Quantitative Analysis of Dynamic Network Protocols[*]

Oded Goldreich[†] *Amir Herzberg*[‡] *Adrian Segall*[§]

January 3, 1994

Abstract

We present a quantitative approach to the analysis of dynamic network protocols. Dynamic networks, extensively studied in the last decade, are asynchronous communication networks in which links repeatedly fail and recover. Loosely speaking, we quantify the reliability of a link at a given moment as the time since the link last recovered. This corresponds to the intuition that a link is 'useful' only after some 'warming up' period since it recovered.

To compare the quantitative approach to the existing (qualitative) approaches, we consider the *broadcast* task, used extensively in actual networks. The existing formulations of broadcast seem too difficult for dynamic networks. In particular, protocols with bounded storage must have unbounded time complexity. Our quantitative formulation of broadcast seems to be closer to the realistic requirements, and escapes such difficulties.

We present a protocol for the quantitative formulation of broadcast. Namely, this broadcast protocol operates in networks which satisfy a weak, quantified reliability assumption. The protocol is efficient, with linear message complexity and high throughput.

[*]This work was partially supported by the Fund for the Promotion of Research in the Technion. This manuscript continues and improves our work in PODC 90.

[†]Dept. of Computer Science, Technion, Haifa, Israel. E-mail: oded@csa.cs.technion.ac.il.

[‡]IBM T. J. Watson Research Center, POB 704, Yorktown Heights, NY 10598. E-mail: amir@watson.ibm.com. Part of this work was done while this author was with the Computer Science Dept., Technion, Israel

[§]Dept. of Computer Science, Technion, Haifa, Israel. E-mail: segall@csa.cs.technion.ac.il.

Comment by Oded (21/8/96): This is a working draft of a paper which, being a continuation of work by Awerbuch, Goldreich and Herzberg (presented in *PODC90*), was supposed to provide a jouranl version for the latter. It seems that this intensions will never materialize.

1 Introduction

The modeling and evaluation of protocols for communication networks is among the most evasive issues in computer science. The need for such models and evaluation criteria is evident in view of the increasing importance of communication in the modern society. Precise evaluation criteria are needed in order to compare alternative solutions to communication problems. These evaluation criteria should properly reflect reality. Namely, there should be correspondence between methods that work well in practice, and solutions judged reliable and efficient according to the criteria.

Communication protocols are difficult to model and evaluate, since communication networks exhibit a variety of phenomena that are hard to analyze. There are many factors to consider, and these factors interact in a complex manner. In particular, the delays and failures involved in the transmission of information through the network are critical factors which are difficult to deal with. In this paper we investigate *dynamic networks* [AAG87, AE86], where the communication is asynchronous and where links may fail and recover.

Most works about dynamic networks assume some 'reliability', in order to avoid banal impossibilities due to extreme unreliability. For example, we wish to exclude the absurd scenario where all links are always faulty. It is desirable that different works will use the same assumptions, or comparable assumptions, to enable comparison and composition of results.

Most theoretical works on dynamic networks use either the 'eventual connectivity' assumption or the 'eventual stability' assumption (see §1.1). Each of these assumptions is the weakest sufficient assumption for an important and general class of tasks. In [AAG87, Fin79] it was shown that 'eventual stability' is sufficient for tasks whose output depends on the topology of the network. In [AG91, AE86, AMS89, Vis83] it was shown that 'eventual connectivity' is sufficient for tasks whose output is independent of the topology. Both assumptions are also the weakest possible for the corresponding classes of tasks, and may hold even for extremely unreliable networks, where parts of the network fail frequently for an unbounded duration.

In practice, networks are much more reliable, i.e. almost always the entire network is operational, or at least most of the network is operational. The average time between failures of each link and processor is very large. Of course, in a very large network failures may occur perpetually, simply due to the huge number of components. However, in practice it usually suffices to communicate between two processors via a single path, with a single alternative path to be used if the main path fails. Networks are designed with sufficient redundancy (alternative paths) so that it is easy to ensure reliable communication. Accordingly, protocols employed in actual networks usually make strong reliability assumptions [BGG$^+$85, LR90, Per83, Her92]. These 'practical' protocols are simple and efficient. In particular, these protocols do not wait for 'eventual' conditions, and in this sense are more efficient than the best possible solutions under the 'eventual connectivity' or 'eventual stability' assumptions.

However, if the network satisfies only the 'eventual' assumptions, then the 'practical' protocols are incorrect. Hence, a protocol may work well in practice, yet seem incorrect under the 'eventual' assumptions. It seems that these assumptions are too much concerned with fault tolerance, while the practical intuition is more concerned with efficiency. Furthermore, the 'eventual' assumptions are *qualitative*. Namely, these assumptions are not quantified by some numeric parameter corresponding to the reliability. In reality, reliability is determined by a multitude of factors. Hence, it is desirable to have an assumption that allows a *continuity* from more reliable networks to less reliable networks. In particular, this would allow *tradeoffs* between efficiency and fault tolerance.

We propose a *quantitative* reliability assumption, parameterized by the *degree of reliability assumed*. It seems safe to assume a rather large degree of reliability for most actual networks. This assumption could be used, however, also for less reliable networks, by assuming a smaller degree of reliability. Solutions using different degrees of reliability may be compared, since the degree of reliability assumed by a solution becomes simply an additional measure of the solution.

We illustrate our quantitative approach by presenting an efficient broadcast protocol that assumes a small degree of reliability. The protocol ensures bounded delay, high throughput and bounded storage. In contradiction, under the 'eventual' assumptions, it is impossible to bound the delay of the solutions, and high throughput implies unbounded storage. Our concepts of delay, throughput and communication complexity formalize the notions used in 'practical' works.

1.1 The Qualitative Approaches

The two main known formal approaches to evaluating protocols for dynamic networks have qualitative nature. Each of them takes a radically different attitude towards failures. In the *eventual-stability* approach [AAG87, Awe88, AS88, BGS88, CR87, Fin79, Gal76, Seg83], failures are assumed to cease at some unknown moment. This, of course, is not meant literally, but it is rather assumed that from that moment on, an entire execution of the protocol can be completed within a period containing no new failures. In the *eventual-connectivity* approach [AG91, AE86, AMS89, Vis83], the only restriction on the nature of failures is that they do not disconnect the network forever.

Both approaches are elegant and gave rise to interesting research problems and important techniques. However, there are severe limitations to the application of these approaches to actual networks. We believe that these limitations are due to the fact that both approaches consider only properties of the link which do not change during the execution. In the eventual connectivity approach, the only relevant property of a link is whether it is 'viable', i.e. if it does not fail forever. In the eventual stability approach, the only relevant property of a link is whether it eventually stabilizes as up or as down. These properties, although dealing with dynamic behavior, are nevertheless static.

We believe that the investigation of dynamic networks should focus on dynamic properties of the links. Indeed, in a large actual network, it may be overly optimistic to assume eventual stability, since quite often some link fails or recovers in some part of the network. Namely, the frequency of failures in the entire network is the number of links times the average frequency of failures in each link. Indeed, many large practical networks have taken special precautions to prevent frequent failures and recoveries from degrading the performance of the network by excessive overhead of topology update message and reroutings [MRR80, LR90, RS91]. Doubtlessly, in these networks, we cannot use protocols which restart most or all of the computation at every new failure or recovery, as most 'eventually stable' works do.

On the other hand, actual networks are designed with highly reliable links and sufficient redundancy, to ensure that they remain connected in spite of failures. Obviously this results in much higher reliability than just eventual connectivity. Therefore, it is wasteful to assume only eventual connectivity. Indeed, many formalizations of 'practical' tasks, following both these approaches, give rise to impossibility results, that contradict successful experience with protocols for the same tasks.

A good example is the broadcast task, that is a useful service for network applications and control. Loosely speaking, *broadcast* is the transmission of a sequence of messages from a predetermined processor called *source* to all other processors in the network. All processors would eventually accept a complete copy of the sequence. Suppose that we assume eventual connectivity. Consider a processor that is disconnected from the source when the protocol starts. Every broadcast message must be stored in the network until this processor reconnects. If the source bounds the number of messages broadcasted until the processor reconnects, then the throughput may be arbitrarily small. Hence, assuming only eventual connectivity, broadcast with high throughput requires unbounded storage. Indeed, the solution of [AE86] uses unbounded storage. Furthermore, in any protocol the delay is unbounded, since there is no bound on the time until the processor reconnects. (The delay is the interval from the time when a message is broadcasted and until it is delivered to the last processor.)

Essentially the same problem exists if we assume eventual stability. Consider again a processor that is disconnected from the source for a very long period. Obviously, the delay is unbounded. Furthermore, all broadcast messages sent by the source while the processor is disconnected should be stored, so that they may be delivered if the processor reconnects. If the source sends only a bounded number of messages until the processor reconnects, then the 'throughput' is unboundedly small.

We find that assuming either eventual connectivity or eventual stability, broadcast has, in the worst case, unbounded delay, and for throughput that is bounded away from 0, unbounded storage as well.

1.2 The Quantitative Approach

The *quantitative* approach is based on a dynamic quantity associated with each link at any moment. Loosely speaking, this is the amount of time since the link has last recovered if the link is up, or zero if the link is down.

This quantity is intended to capture the ability of the protocol to utilize the link. Typically, protocols cannot efficiently utilize a link instantaneously after the link recovers, but only after some 'warming up' period. Protocols need the 'warming up' period in order to exchange messages for different purposes. Messages may be needed to compare the state of the end-points of the link, or to exchange information between them. Other messages may be required to inform other processors that the link has recovered. The link becomes useful if it passed, without failures, this 'warming up' period; namely, if it was 'up' for sufficiently long.

We present a broadcast protocol that assumes that the network is connected at all times by 'sufficiently up' links. In this way we can prove bounded delay and throughput even for the worst case, independently of the failures, and with bounded storage.

The 'sufficient up' *amount* required from the links may change from protocol to protocol. For example, in this exposition we present a broadcast protocol that requires less reliability than the protocols in an earlier version [AGH90a]. Hence, we may quantitatively *compare* the reliability requirements of different protocols. It is plausible that there will be tradeoffs between the reliability requirements and the efficiency of protocols. The network designer can pick the most efficient protocol that requires no more than the reliability which is assumed to hold for her network.

Our protocol requires that at any moment, every two processors are connected by some path the links of which were up during the last $3n$ time units, where n is the number of processors. This requires an entire path to be up simultaneously for some time, which is obviously stronger than requiring eventual connectivity. Awerbuch et al. [AMS89] observed, that if the failure probability of each individual link is constant, then the "failure probability" of the path becomes exponentially close to 1 as a function of the length of the path. They conclude that the requirement that the entire path is up is too strong. We disagree with this conclusion, for both practical and theoretical considerations:

Practice: Actual networks [BGG$^+$85, LR90] are designed and implemented so that the probability of a failure along a path is quite small. As a result, many networks use only a small number of the possible alternative paths between each two processors [LR90]. These networks still operate successfully and are considered fault-tolerant, since the probability that there will be a failure in several disjoint paths at about the same time is negligible. Also, many networks perform end-to-end communication by sending messages over a single, efficient path connecting the two end stations [Her92, BGG$^+$85]. Furthermore, it seems that the technological improvements in link reliability are growing faster than the growth of the networks.

Theory: the analysis where the failure probability is constant and only the length of the path grows is misleading. It is reasonable to expect that as networks grow, processors would be connected mostly by links which are highly reliable. It is easy to see that the analysis of [AMS89] fails, for example, if the failure probability of each link is inversely proportional to the length of the path.

We conclude that for most actual networks, the quantified formalizations are sufficient, and the *eventual connectivity* formalizations are *too pessimistic.*

1.3 Complexity Measures for Broadcast

The traditional definitions of time and communication complexities are tailored to tasks where the output is a result of some computation to a given input. Such definitions are inappropriate for the analysis of 'online' tasks, such as broadcast, where the inputs are given and outputs are produced during the (possibly infinite) execution. Furthermore, we consider broadcast in a dynamic network, where links fail and recover during the execution.

We give definitions for throughput and communication complexity for broadcast, which address the 'online' nature of the task and the dynamic nature of the network. Our definitions could be modified easily for other 'online' tasks.

The *throughput* measures, intuitively, the rate by which messages are accepted for broadcast, i.e. average number of broadcast messages per time unit. This rate is limited by the capacity of the network and by the time that it takes the protocol to remove messages from the network.

Throughput, in this sense, is a very practical measure. For example, the advantage of [CR87] over [Fin79] is exactly in improving the throughput. However, it is difficult to define the throughput since it cannot be measured for a single message accepted. As stated above, throughput is related to the average time used to broadcast messages over the number of messages. The problem is to select the right interval over which to average. Short intervals may deliver no (or few) messages, while long intervals may 'hide' bursty behavior. The crux of our definition, presented in section 5.3, is that the selection of the intervals is left as a parameter.

It is similarly difficult to define communication complexity for broadcast (and other 'online' tasks). We cannot truly measure the communication 'per message accepted', and on the other hand amortization may ignore "short-term" disastrous effects. Again, the crux of the solution is to allow the selection of intervals for amortization to be a parameter.

Time and Congestion. When defining time complexity, it is commonly postulated that in case m packets are sent concurrently, over the same link, the last arrives within m time units. An alternative convention postulates that all m packets arrive within one time unit. We believe that, as long as the protocol limits itself to a 'reasonable' number of concurrent messages (i.e., 'reasonable' congestion), the second alternative better reflects reality. (See justification in Section 2.1). We analyze our protocol using the second convention, while bounding the congestion (i.e., the maximal number of packets in transit concurrently) by $O(n)$. It seems that our protocol maintains all its complexities, up to a constant, even when using the first convention for time complexity.

1.4 Our protocol

We present a broadcast protocol, i.e. a protocol which delivers a sequence of messages accepted at a source processor to all of the processors in the network. The protocol operates in the presence of an arbitrary schedule of link failures, provided that the network satisfies a weak, quantified reliability assumption. Loosely speaking, the assumption is that at any time, every pair of processors is connected by a sufficiently reliable path.

The protocol is highly efficient in communication ($O(1)$ message per link), throughput ($\Omega(1)$) and delay ($O(n)$), and uses $O(n)$ storage, where n is the number of processors. In fact, by combining high throughput with bounded ($O(n)$) storage, our protocol improves upon the known, 'classical' broadcast protocols, Echo, PIF and Intelligent Flood, used in many actual networks. (Details follow.) Furthermore, these classical protocols work only for *static* asynchronous networks.

The Echo and PIF protocols [Cha82, DS80, Seg83] have lower throughput. Namely, the maximal time required between broadcasting two messages using Echo is proportional to the diameter of the network, compared to a constant in our protocol It is simple to improve the throughput of PIF using a window, however this seems to increase the worst case delay to $O(n^2)$.

The Intelligent Flood protocol [Per83, Seg83] require unbounded[1] storage, both due to 'infinite counters' and to unbounded link capacity requirements. Our protocol is almost as simple as these 'classical' protocols, especially the version for static networks presented in §3.1. The known fault-tolerant broadcast protocols [AE86] are variations of the Flood protocol, and also require unbounded storage.

We develop and analyse the protocol in a modular manner. We begin with a simple version, which has poor throughput and unbounded storage. This simple version is built as a combination of two complementing mechanisms: a progress mechanism, which guarantees that messages are delivered,

[1]Actual implementations of Intelligent Flood are bounded; they store only the last k messages, where k is a large number, selected experimentally and by bounds on the delays.

and a synchronization mechanism, which ensures that the number of messages delivered by different processors is always roughly the same. We later improve by a series of simple reductions, providing optimal throughput and bounded storage.

The protocol we present is quite simple. Indeed, it may be regarded as a combination of several techniques which are well known, and widely used in practical protocols. Namely, the protocol combines a simple flow control mechanism with the well known flooding mechanism. Failures and recoveries are dealt with, by using standard techniques. We bound the space requirements and the length of messages by using modular counters, as done in practical protocols. Finally, to improve the throughput we use a window mechanism, which is also used in many practical protocols.

The value of these techniques is well recognized in practice. However, there has been limited use of these techniques in previous theoretical works on dynamic networks. The quantitative approach allowed us to formally analyse contributions of these techniques, instead of applying them based only on heuristic arguments. We hope that this indicates that the quantitative approach is more realistic than previous formal approaches to dynamic networks.

Organization

Section 2 contains the definitions of basic dynamic the model, the quantitative reliability assumptions, the broadcast task and the complexity measures. Section 3 presents a simplified version of our protocol, having unbounded storage and low throughput. The analysis of the simplified version is presented in section 4. Section 5 contains enhancements to the simplified protocol, yielding the final version, which achieves bounded storage and optimal ($\Theta(1)$) throughput.

2 Definitions

2.1 The Dynamic Networks Model

We consider the dynamic network model of [AAG87, AE86, BS88]. The network is represented by an undirected graph, with n vertices corresponding to the processors and m edges corresponding to communication links between some pairs of processors (neighbors). There is no assumption about the topology of the graph, and the topology is a-priori not known to the processors. However, the processors know[2] n and each processor has a distinct identity.

The model is asynchronous in that the processors do not have clocks and the delays are finite but unbounded. Hence, some events are concurrent, and events are only partially ordered. Like previous authors [AAG87, AE86, BS88, GHS83], we find it easier to consider a total order between events. This total order should be interpreted as an arbitrary extension of the actual partial order. This extension is unknown to the processors, and our results hold for any such extension. It is possible to obtain equivalent results without assuming total order (see [AGH90b]).

We now make a further important simplification, again following [AAG87, AE86, BS88, GHS83]. Namely, we associate a positive number $time(e)$ with each event e. The number $time(e)$ represents the 'normalized time' of event e. The time is 'normalized' in the sense that for complexity analysis purposes, one time unit is defined as the maximal transmission delay. The definition of the maximal transmission delay is given later, when discussing time complexity measures.

At first sight, this simplification seems to introduce some synchronization. However, like the total order itself, the association of the time to the events is only for the sake of analysis, and is completely transparent to the protocol. In particular, the processors are not aware of the 'time' of events during the execution. Hence, the model remains completely asynchronous. Namely, associating time to events simplifies the formalization, but is not essential (see again [AGH90b]).

We consider *send, receive, fail* and *recover events*, each referring to a specific processor and link of that processor and having the natural meanings. The *send* event is an *output event*; the other events are *input events*. Additional task specific events are described in subsection 2.3. Each send

[2]In fact a reasonable upper bound for n is enough, since the protocol needs to know n only for allocation of storage and sizing of counters.

and receive event is associated to a specific *packet*[3], which is the information sent or received.

We use the natural notions of a link being up or down at a processor, and of an up interval, following [AAG87, AE86, BS88]. Messages are sent and received only when the link is up.

Definition 1 *A time interval $[t_1, t_2]$ is an* up interval *of link (u, v) at processor v if (u, v) recovers at processor v at time t_1, and does not fail at v during $[t_1, t_2)$. A link (u, v) is* up *at processor v at any time during an up interval of (u, v) at v, and* down *at any other time.*

We assume that the events satisfy DLC (Data Link Control) reliability. This is guaranteed by using a reliable DLC procedure as a lower layer on all links. DLC reliability is defined in [BS88, GS92], and is restated below in Definition 2.

Normally, the DLC procedure ensures that all packets sent are received at the other end of the link, in the correct sequence. However, if the link becomes inoperational for too long, it is declared faulty and the DLC procedure throws away any undelivered packets. Namely, a reliable DLC procedure ensures that the sequence of packets received is the same as the sequence sent from the other end, except that some subsequences sent prior to failures may be missing.

Definition 2 A set of events with associated times satisfy DLC reliability if the following properties holds for any link (u, v):

1. Processor u sends and receives packets over (u, v) only during up intervals.
2. **Follow-up:** If (u, v) fails at u while up at v, then it would also fail in v within finite time.
3. **Delivery:** If (u, v) is up at u at t and u sends a packet to v at t, then within finite time this packet is received by v or the link fails.
4. **Crossing:** If (u, v) fails at u at some time t, there is a time t_c after t but before (u, v) recovers at u such that (u, v) is also down in v at t_c and no packet sent over (u, v) from either end before time t_c can be received by the other end after time t_c.

[3]The term *packet* is used for the control messages sent by the protocol. This convention is intended to avoid confusion between packets, i.e. control messages sent by the protocol, and messages accepted from the higher layer.

5. **FIFO:** Suppose u receives at time t_u a packet which was sent at time t_v by v over (u, v). Let t'_u denote the last time before t_u when (u, v) went up at u. Similarly, let t'_v denote the last time before t_v when (u, v) went up at v. Then the sequence of packets sent by v to u during $[t'_v, t_v]$ is identical to the sequence of packets received by u from v during $[t'_u, t_u]$.

The last two properties ensure a reliable one-to-one correspondence between up intervals. The Crossing property ensures that the link is cleared of old messages following each failure. The FIFO property ensures that the packets in two 'corresponding' intervals are delivered in the same order, with no gaps or duplicates.

We restrict our attention to the case where the delays of the Follow-up and Delivery properties are at most one time unit. Namely, there is at most one time unit between a failure at one end and the corresponding failure at the other end, or between a send event and the corresponding receive (or failure) event. In this case we say that the link satisfies *normalized DLC reliability*.

We stress that with this normalization, the delay does not depend on the number of packets in transit concurrently. Namely, we allow an unbounded number of packets to be transmitted at the same time unit over a link. This is a simplification of reality, where links have a specific capacity and therefore the delay grows when there are many packets in transit. However, for a reasonable amounts of concurrent transmissions by the protocol, the changes in the delay *caused by packets sent by the protocol itself* are negligible, as the delay is mostly dominated by the packets sent concurrently by other protocols running in the network. The reason being that the delay of a packet is determined by the total number of packets already queued for transmission, counting all packets independent of the protocols which have placed them. Hence, in case it is guaranteed that only one protocol sends packets on a link, the actual link delay could have been determined by the protocol. Typically, when the asynchronous model is used, it is believed that a significant part of the communication is caused by other (unknown) protocols running concurrently in the network. Indeed, the belief that the delay is determined by the other (unknown) protocols, is the very justification for the use of the asynchronous model.

We now define our model of an execution, called a timed dynamic execution. Where no ambiguity may arise, we sometimes say simply *execution*, meaning a timed dynamic execution. This definition is simplified, e.g. by ignoring local processing. For a more precise definition, see [ADLS90].

Definition 3 *An* algorithm Π *is a mapping from a* state *and a set of input events to a (new) state and a set of output events. A* timed dynamic execution *of algorithm* Π *is a sequence of tuples* $\{(t_i, p_i, s_i, I_i, n_i, O_i)\}$ *where* t_i *are monotonously increasing positive real numbers (representing time),* p_i *is a processor identity from* $1, \ldots n$*,* s_i *is the state,* n_i *is the new state,* I_i *is the set of input events and* O_i *is the set of output events, such that for every* i *holds* $(n_i, O_i) = \Pi(s_i, I_i)$ *and the* $s_i = n_k$ *where* $k < i$ *is the last tuple before* i *of processor* p_i*, and where the events on each link satisfy normalized DLC reliability.*

Throughout the rest of this section, we consider a specific timed dynamic execution α.

2.2 Quantitative Connectivity Assumptions

We now present the main conceptual contribution of this paper: quantitative assumptions of connectivity. The assumptions are, basically, extensions of the (qualitative) assumption that the network is 'always connected'. The extensions are based on the observation that for many natural protocols, connectivity at any moment is not enough, since it takes some time for the protocol to utilize a new path between two processors. More specifically, it usually takes some time since a link recovers and until the protocol can 'really use it' as if it never failed. This time may be used, for example, to synchronize across the link or to notify other processors of the recovery.

Essentially, a link is $l-$Up if it is up during the last l time units. Note that l time units are sufficient for transmission of a packet over a path of length l.

Definition 4 *We say that link* (u, v) *is* $l-$Up *at time* $t \geq 0$*, if link* (u, v) *is up at both* v *and* u *during the entire interval* $[\max(t-l, 0), t]$*.*

Note that v knows, at any moment, if (u, v) is up at v. However, processor v does not know if (u, v) is currently $l-$Up. One reason is that the network is asynchronous and hence v cannot detect when l time units have elapsed. Another reason is that processor v does not know if (u, v) is up at u.

A path is $l-$Up at time t if all of its links have been up during the l time units before t, i.e. if all of its links are $l-$Up at t.

Definition 5 *A* path is $l-$Up *at time* t*, if all of its links are* $l-$*Up at* t*.*

Usually, we are not interested in the reliability of a specific path, but rather about the existence of some reliable path between the processors, i.e. reliable connectivity. Up-connected processors are processors connected by a path of links which have been up for sufficiently long time. This is formalized as follows.

Definition 6 *Processors u and v are $l-$*Up-Connected *at time t if there exists a path connecting u and v that is $l-$Up at time t.*

Two processors may be $l-$Up-Connected during a long time interval, without having any path connecting them for the entire interval. For example, suppose there are two different paths p, q between processors u and v. Suppose further that p is $l-$Up at $[t_1, t_2]$ and that q is $l-$Up at $[t_2, t_3]$. Then u and v are $l-$Up-Connected during the entire $[t_1, t_3]$. Note that all the links in p should be up during $[t_1 - l, t_2]$ and all the links in q should be up during $[t_2 - l, t_3]$.

A natural reliability assumption, which will be used in this paper, is that every two processors in the network are reliably connected at any time. In other words, the network is l-Up at any time.

Definition 7 *A* network is $l-$Up *at time t if every two processors are $l-$Up-Connected at time t.*

2.3 The Broadcast Task

All broadcast messages are accepted from the 'higher layer' at a single processor, called the *source* and denoted by s. The interaction between the broadcast protocol and the 'higher layer' that uses it consists of the following events:

accept The protocol accepts a new message from the 'higher layer' (at processor s).

deliver The protocol delivers a message to the 'higher layer'.

ready The protocol is ready to accept a new message from the 'higher layer' (at processor s).

The *ready* event is essential in order to bound the storage required in the source to hold messages which were accepted and are still being broadcasted. Namely, the higher layer is required to wait for a ready event between every two accept events. Formally, we assume that every two accept events at s are interleaved by a ready event at s.

The goal of the broadcast protocol is to deliver the exact sequence of messages accepted at s to the higher layer at each processor.

Definition 8 *Execution α is a* correct broadcast from *s if at any time the sequence of messages delivered at any processor to the higher layer is a prefix of the sequence of messages accepted from the higher layer at the source s.*

Note that we define above only correctness, i.e. it is possible to satisfy this definition in an empty manner by never delivering any message. The 'liveness' requirement is implied by the throughput and delay complexity measures, defined in the next subsection.

2.4 Complexity Measures

We now present complexity measures for broadcast. We begin by suggesting refined complexity measures for the communication and the throughput. We then formalize the notion of congestion (over links), i.e. the maximal number of packets in transit over a link at the same time. Finally, we define delay and space complexity.

2.4.1 Communication Complexity Measures

The communication complexity should reflect the amount of network bandwidth consumed by the protocol. Most works measure the amount of communication per message accepted from the higher layer. Namely, the communication complexity in these works is often defined as the ratio of the number of packet transmissions over the number of accept events, both counted during an entire execution or any prefix of an execution. This is natural in protocols where each packet sent may be easily associated with a single corresponding message, as in most protocols using infinite counters, e.g. [AE86], [JBS86].

However, in many works the same packet may serve several accept events. In fact, many protocols improve complexities by performing some tasks periodically, instead of doing them for each message, thereby amortizing their costs over many messages. For example, in [AGH90a] we suggested an end-to-end protocol which transmitted an exponential number of packets for a 'clean-up' task which was performed once per an exponentially long period of time. The amortized communication complexity, i.e. the ratio of packets sent to messages accepted during *prefixes* of the execution, was only $O(m)$. However, in the 'bursty' intervals, the ratio between packets transmissions and messages accepted was exponential.

These short 'bursty' intervals may cause congestion in the network. It seems desirable to reflect such 'bursty' intervals, i.e. the congestion caused by the protocol, in a refined measure of communication complexity. In particular, there seems no apparent reason for considering only prefixes of the execution and not any interval.

Possibly with this motivation, the communication complexity of [AG91, AMS89] is defined as the maximal number of packet transmissions between two accept events. Namely, [AG91, AMS89] measure the number of packet transmissions in intervals with exactly one accept event, instead of amortizing the ratio of sends to accepts over prefixes of the execution. Therefore, a protocol with low complexity as defined in [AG91, AMS89] would not cause congestion by 'bursty' intervals.

However, we believe that in most realistic scenarios, performing some tasks periodically, once for several messages, is useful and would not cause congestion. Unfortunately, the measure of [AMS89] does not amortize such periodic tasks at all. Furthermore, even a protocol which transmits the same number of packets for each message may have higher complexity with the measure of [AMS89], if it may transmit packets due to a message even after accepting a new message (pipeline). As a result, this measure seems to be artificially high for many protocols that do not cause congestion in practice. For example, under this measure, the communication complexity of the Intelligent Flood protocol is $O(nm)$ instead of $O(m)$ when amortization is for prefixes [AG91].

We therefore propose a new complexity measure. Our goal is to identify the 'bursty' congestion caused by some protocols, but to allow some amortization and pipeline operation. Instead of defining the amortization intervals explicitly, we define the communication complexity with respect to a given predicate $P(\alpha, t, l)$. This predicate determines if interval $(t, t+l]$ in execution α is an amortization interval. Note that the previous complexity measures become special cases. Another simple and natural special case is sufficient for our work, namely when the predicate is simply a bound on the length l of the interval; this is defined in subsection 2.4.3 below.

Definition 9 *Consider a broadcast protocol, a predicate $P(\alpha, t, l)$ where α is an execution and t, l are positive real numbers, and a function $C : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. We say that C is the* communication complexity of the protocol over intervals satisfying P, *if for every execution α and t, l such that $P(\alpha, t, l)$ holds, the number of receive events during $(t, t+l]$ is at most*

$$C(n, m, A([t-l, t+l)), F([t-l, t+l)), R([t-l, t+l)))$$

where:

$$\begin{array}{lcl} A([t-l,t+l)) & \overset{\text{def}}{=} & \textit{The number of } \mathsf{A}\textit{ccept events (in } s\textit{) during } [t-l, t+l). \\ F([t-l,t+l)) & \overset{\text{def}}{=} & \textit{The number of } \mathsf{F}\textit{ail events during } [t-l, t+l). \\ R([t-l,t+l)) & \overset{\text{def}}{=} & \textit{The number of } \mathsf{R}\textit{ecover events during } [t-l, t+l). \end{array}$$

There are two subtle issues with the definition:

- Since we deal with dynamic networks, we allow for some communication overhead per failures and recoveries. Furthermore, we count the number of packets received, and not of the packets sent. In dynamic networks, the distinction is meaningful, since some packets sent just before a failure may not be received. In particular, our protocol sends $O(n)$ packets per recovery, but *receives* only $O(1)$ packets per recovery; the potential large difference is due to messages which were sent but not received (lost) due to a failure. The justification for counting receive events rather than send events is that the protocol cannot utilize packets sent but not received.

- We amortize over the number of accept, fail and recover events not only in $[t, t+l)$, but also in the l preceding time units. This is needed since some of the messages during $[t, t+l)$ may be due to events before $[t, t+l)$. In the extreme case, there may be no accept, fail or recover events in $[t, t+l)$, but packets may be transmitted to deal with previous events. To minimize notations, we use l also for the length of the interval before t where we count accept, fail and recover events.

Definition 9 defines the communication complexity as a function of five arguments. This general form seems necessary, and hopefully sufficient as well, to allow us to express correctly the communication complexity of any algorithm. However, it is inconvenient. We next define a simple special case of the definition, which is sufficient for this paper and seems sufficient for many protocols.

Works following the eventual stability approach usually consider communication complexity as directly proportional to the number of topology changes. This implies the simplified form $C(n, m, a, f, r) = f \cdot \bar{C}(n, m, a)$, where $\bar{C}$ is the *amortized communication complexity* [AAG87]. Expressing the communication complexity in this way is justified when the actual communication is indeed linear in the number of failures. This is indeed the case in many of these works, where every failure or recovery may cause the protocol to restart operating from the beginning. Many eventual-stability protocols work in this *'blast-away'* technique [Gal76], [Fin79], [Seg83], [AAG87].

However, the 'blast-away' technique is wasteful; it is desirable, and often possible, to recover from a failure at a cost much smaller than restarting the task. In particular, our protocol recovers from failures with a small fixed cost, independent of the number of broadcasts done so far. There have been several previous works which featured a smaller cost per recovery than the 'blast-away' approach [ACK90], [Awe88], [AS88],[BGS88], [MS79],[SS81].

We therefore propose another simplified form for the communication complexity. This communication complexity consists of fixed cost per each accept, failure and recovery event. The cost depends on the kind of the event.

Definition 10 *Consider functions $C_A : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, $C_F : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ and $C_R : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ and broadcast protocol with communication complexity C over intervals satisfying some predicate P. We say that the communication complexity over intervals satisfying P is C_A per* accept*, C_F per* fail *and C_R per* recovery *if for every n, m, a, f and r, the following holds:*

$$C(n, m, a, f, r) = a \cdot C_A(n, m) + r \cdot C_R(n, m) + f \cdot C_F(n, m).$$

Length of Packets

In the definition of the communication complexities measures above we ignored the *length* of the packets. This allows protocols to use extremely long packets or packets whose length is not bounded as a function of the size of the network. For example, the classical fault tolerant versions of the Intelligent Flood protocol use packets with unbounded length. The final version of our protocol uses only short packets, as defined below. Note that we allow a packet to contain a message accepted from the higher layer.

Definition 11 *Let M be the maximal number of bits in messages accepted. We say that a protocol* uses only short packets *if the maximal number of bits in any packet sent in any execution of the protocol over networks with n processors is $O(M + \log(n))$.*

2.4.2 Throughput

We now define the throughput of a protocol. Intuitively, the throughput is the worst-case bound on the rate in which messages may be accepted from the higher layer. For simplicity, assume that the higher layer always has packets to broadcast. (This assumption does not effect our results.)

Like communication complexity, the throughput is traditionally measured for the worst case prefix of an execution of the protocol. As for communication complexity, we see no apparent reason to consider only prefixes of executions, which may hide important transient effects. Instead, we propose to consider the throughput of any *interval* of an execution of the protocol.

Definition 12 *Consider a broadcast protocol, a predicate $P(\alpha, t, l)$ where α is an execution and t, l are positive real numbers, and a function $T : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}$. We say that T is the* throughput of the protocol over intervals satisfying *P, if for every execution α and t, l such that $P(\alpha, t, l)$ holds, the number of accept events during $(t, t + l]$ is at least $l \cdot T(n, m)$.*

This definition seems to formalize practical notions of the throughput of protocols. In particular, this definition allows to analyse the advantage of window mechanisms. Such mechanisms are used, to enhance throughput, in many protocols, e.g. [CR87]. In fact, we also improve the throughput of our protocol by using a window mechanism.

2.4.3 Predicate defined by length of interval.

Typically, and in this work, the predicate P of definitions 9, 10 and 12 would be the length of the interval in time units. Namely, $P(\alpha, t, l) = \text{TRUE}$ if $l = L(n, m)$, for some function L. In this case, we will talk about intervals *of length* $L(n, m)$, instead of 'intervals satisfying P'.

Note that replacing $l \geq L(n, m)$ instead of $l = L(n, m)$ would at most double the communication complexity and reduce the throughput by half. This is the motivation for using the (slightly) simpler $l = L(n, m)$.

2.4.4 Congestion over links

Our definition of the normalized DLC reliability says that the maximal delay of a message is one time unit, regardless of the number of messages concurrently in transit. This convention has been justified by noting that as long as the number of messages sent concurrently by the protocol is not 'too large', it is reasonable to expect that the delay would be mostly determined by the number of concurrent messages generated by *other* protocols. Using the same reasoning, we do not count the storage used for (data-link) queueing of the message after it has been sent, since this storage is shared among many protocols. Both simplifications seem justified provided that the protocol does not send an excessive number of concurrent messages on any link, namely the protocol is not the cause of *congestion* on any link. We now define the congestion of the protocol; i.e., the maximal number of packets (of the protocol) which are concurrently in transit over any link.

Definition 13 *The* congestion *of a protocol is an upper bound to the number of messages sent by any processor u to neighbor v during an operating interval $[t, t']$ of (u, v) at u, but not received by v until t'.*

2.4.5 Delay

The delay of a protocol is the maximal delay over all possible executions of the protocol. We now formalize the notion of the delay of a given execution α of a broadcast protocol.

Definition 14 (Delay) *Consider a* timed dynamic execution $\alpha = \{(t_i, p_i, s_i, I_i, n_i, O_i)\}$. *For every $j \in \mathbb{N}$, let $t_a(i)$ be the value of t_i where $(t_i, p_i, s_i, I_i, n_i, O_i)$ is the j^{th} event in α where I_i includes an accept event (with $p_i = s$), and ∞ if there is no such event. Similarly, let $t_d(v, i)$ be the value of t_i where $(t_i, p_i, s_i, I_i, n_i, O_i)$ is the j^{th} event in α where O_i includes a deliver event and with $p_i = v$, and ∞ if there is no such event.*

The delay *of execution α of a broadcast protocol is:*

$$\sup\{t_d(v, i) - t_a(i) | (\forall v \in V) (t_a(i) < \infty)\}.$$

The delay is measured in the worst case, since it may be important to bound the maximal time since a message is accepted from the higher layer, at the source, and until it is delivered to the higher layer at the last processor.

2.4.6 Space Complexity

We use the standard measure of space complexity used in most works dealing with dynamic networks [AAG87, AE86]. Namely, the space complexity includes the maximal number of words stored by the processor, where every word contains $O(M + \log n)$ bits, where M is the maximal number of bits in a message accepted from the higher layer.

Definition 15 (Space) *Consider a broadcast protocol and a function* $S : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$. *The* space complexity of the protocol *is bounded by* S*, if in every execution the number of allocated bits at every processor and at any time is* $O(S(n, m) \cdot (M + \log n))$.

3 Simplified Broadcast Protocol

In this section we present a broadcast protocol for $O(n)-$Up dynamic networks which uses unbounded storage and has low throughput. This is a simplified version of our main protocol. In the next section we analyse this (simplified) version of the protocol. The enhancements to bounded storage and optimal ($\Omega(1)$) throughput are given in Section 5. The advantage in describing and analyzing the simplified version is that its *analysis* is much simpler. Furthermore, the analysis of the enhancements in Section 5 is done by reducing executions of the enhanced versions to executions of the simplified version, and using properties from the analysis of the simpler version.

We first describe the protocol informally, beginning with the basic mechanisms and gradually augmenting them. A concise description is given in Fig. 1, and the formal code is presented in the appendix (Figs. 8-11). To refer to line l in the code described in figures 10 and 11, we use the notation $< l >$.

The protocol contains three main components, each using a different type of packets. The first component uses *sync* packets and its goal is to ensure synchronization of the delivery of messages. Namely, it ensures that the number of messages delivered by a processor would be the same, up to one, as that of a neighbor connected by a link which is up for 'enough time'. This component is described informally in §3.1, and formally by the lines in the code marked by S in the right margin. The second component uses *recover* and *update* packets and its goal is to deal with link failures and recoveries. This component is described informally in §3.2, and formally by the unmarked lines in the code. The third component uses $flood$ packets and its goal is to speed up the delivery of messages. This component is described informally in §3.3 and formally by the lines in the code marked by F in the right margin.

3.1 Operation in Static Networks

We first try to give some intuition to the design goals and the main ideas of the protocol. On one hand, the protocol should have *high throughput* (rate). To achieve this, the source does not wait for network-wide progress, before enabling a new accept event. Instead, the source waits only for progress of its immediate neighbors. This should be contrasted with the *Echo* and *PIF* protocols [DS80, Cha82, MRR80, Seg83], that wait before accepting a new message until an entire spanning tree converges.

On the other hand, the protocol should *not cause high congestion* on links. High congestion may be caused, for example, by the *Intelligent Flood* protocol and its variants [Per83, Seg83, AE86], since the source may issue messages much faster than some links can transfer them. Hence, these messages must be stored in the source (resulting in unbounded space complexity) or in the links (resulting in unbounded link concurrency).

Our protocol prevents high congestion by limiting the number of messages in transit over any link. This is achieved by synchronizing between each pair of neighbors. This synchronization resembles synchronizer α of [Awe85] and the minimal hop protocol of [Gal76, Seg83].

In the rest of this subsection we informally describe this synchronization mechanism, which is the core of the protocol, and suffices for operation in static networks. In the code, appearing in Figs. 8-11, the lines which implement this mechanism are marked by S in the right margin.

The basic idea is that a processor sends message i to all of its neighbors together <H5>, after receiving message $(i-1)$ from *all* neighbors <H7>. Since this condition is observed by all the neighbors of the processor, a processor would not *receive* message $i+1$ from any neighbor before sending message i. Hence, at most two messages are in transit over any link at any moment. This prevents congestion over links.

To implement this mechanism, the protocol sends each message together with its index in the sequence of accepted messages. The pair of a message and its index is called a *sync packet*. We denote the sync packet which contains message m the index of which is i by $sync(m, i)$.

In static networks, once a packet has been sent to the neighbors, it may be discarded (to free memory). Since a message is sent to a neighbor only after the neighbor sent (and discarded) the previous message, it follows that all processors except the source have to store at most one message. The source has also to store the messages accepted but not sent yet. In order to prevent congestion at the source, the protocol must control the rate of accept events. The solution is to use the assumption that the message accept events are interleaved with ready events at the source. The source enables a ready event for a message only after having received a sync packet containing the previous message from all of its neighbors <H7> and therefore discarding this previous message. Hence, the source will not accept a message before sending to its neighbors the previous message. This ensures that at any time, there is at most one message accepted by the source but not yet sent to its neighbors. Namely, the storage required in the source is bounded (in fact just one message is stored).

The protocol also synchronizes the number of messages delivered to the higher layer by any two neighbors. This is achieved by delivering a message to the higher layer simultaneously to sending a sync packet containing that message to all neighbors. Hence, the number of messages delivered by any processor v is at most one more than the number of messages delivered by any of the neighbors of v. Since every processor is connected to the source, it follows that at any time, in static networks, any processor has delivered all but at most the last n messages accepted from the higher layer in the source. In the final protocol, for dynamic networks, similar properties hold when considering $3n-$Up links and assuming that the network is connected via $3n-$Up paths (see Lemma 23). Namely, the protocol ensures that the following two conditions hold at any time:

Definition 16 Link Synchronization: *We say that the* link synchronization condition holds at time *t if for every link (u, v) which is $3n-$ Up at time t, processor v delivers until t at most one message more than u delivers until t.*

Synchronization: *We say that the* synchronization condition holds at time *t if the number of messages accepted by the source until t is at most n more than the number of messages delivered by any processor.*

Note that if the network is connected via $3n-$Up paths at time t, then link synchronization at t implies the synchronization condition at t, due to the interleaving of accept, deliver and ready events in the source.

As described in this section, the protocol uses infinite counters. However, we show later that if the network is always $3n-$Up, and in particular if the network is static, then it suffices to use (finite length) modular counters instead. Essentially, this follows from proving that the synchronization condition holds.

3.2 Dealing with Topology Changes

When a processor v detects a failure on the link to its neighbor u, it stops waiting for messages from u. Namely, while (u, v) is down, v behaves as if (u, v) never existed. The challenge is to obtain the complementing situation: that 'soon' after (u, v) recovers, it would appear as if it never failed.

After processor v detects the recovery of the link to u, two types of actions are taken in order to simulate the situation as if (u, v) never failed: updating and re-synchronizing. Updating is needed if processor v received more messages than u, and in this case processor v sends to u the messages that u may have not received yet. This updating action is a standard practice [MRR80, Per83, JBS86].

Our protocol requires precise synchronization between the messages delivered by two neighbors connected by a $3n-$Up link. The update operation cannot achieve such precise synchronization; in particular, it does not prevent v from delivering more messages. This is dealt with by the re-synchronization action, whose goal is to reach a state where the number of messages delivered by u and v differ at most by one, like the situation over a link which has never failed.

To re-synchronize, processor v delivers a new message *following* the recovery only when it 'knows' (from a sync packet, or from update packet to be described) that u has delivered the previous message. However, there is an exception: v may deliver *one* message after (u, v) recovered, *without* waiting to 'know' that u has delivered the previous message. This exception is necessary to avoid (unlikely) 'livelock' situations where v does not deliver messages due to a long sequence of alternating failures and recoveries in two or more of its neighbors. Details follow.

To understand the possible 'livelock', suppose for a moment that processor v, with neighbors u and w, waits before delivering messages until it 'knows' that all of the neighbors whose link is up have delivered the previous message. Consider time t when links (u, v) and (w, v) are both up, and processor v waits for both u and w to deliver the previous message in order to deliver a new message. We will show a (very unlikely) sequence of frequent repeated failures and recoveries of (u, v) and (w, v), which are furthermore interleaved (in an unlikely manner) so that at any moment either (u, v) is up or (u, w) is up. Yet, the failures would be so frequent that v will not receive messages from either u or w and hence v never learns that u or w delivered the last message delivered by v. (Note that such scenario is impossible to achieve by frequent failures and recoveries on a *single* link e.g. (u, v), since upon its failure v would immediately be able to deliver the message.) During the entire scenario, v does not receive any message from either u or w.

The sequence begins with (u, v) failing at $t + .25$ and recovering at $t + .5$. During this time, v does not deliver messages since it waits for response from w. From the DLC properties, this response should arrive before $t+1$ or (w, v) must fail. Indeed assume that it fails at $t + .75$ and recovers at $t + 1$. During this period, v does not deliver messages since it waits for response from u. Again from DLC, this response should arrive before $t + 1.5$ (since (u, v) is up since $t + .5$). However, (u, v) will fail again during $[t + 1.25, t + 1.5]$, and this alternating sequence of failures and recoveries could continue forever. This problem is easily prevented by allowing v to deliver one message after recovery and before re-synchronization.

To allow the updating and re-synchronization, the processors connected by the link should exchange the indices of the last message received and the last message delivered. There could be a substantial difference between the number of messages received and the number of messages delivered by a processor, since the processor may receive a 'bunch' of several messages following a recovery, and only then it would deliver them one by one, synchronizing with its neighbors. (In static networks, each processor receives at most one message more than it had delivered.)

Technically, this exchange is performed by sending over a link, upon its recovery, an *update packet* containing the indices of the last delivered and received messages by this processor <F2>. The fields containing the indices are called the *deliver-counter* and the *receive-counter* of the update packet, and are denoted by c_d, c_r respectively. The update packet is different from the sync packets and hence ignored by the mechanisms described in §3.1.

The deliver-counter c_d is used to let v 'know' the index of the last message delivered by u until the recovery. Even if v has not delivered these messages yet, processor v does not wait for sync packets with these indices from u. This is important, since u has already sent these sync packets when delivering the messages, and therefore would not send these sync packets to v.

The receive-counter c_r is used to identify the situation where v have received more messages than u. Processor v should send these messages to u, to ensure that u would receive them. In this case, v sends to u the messages received which are numbered more than the value of c_r received from u.

However, from the synchronization condition follows that, assuming that the network is connected by $3n-$Up paths, then u is guaranteed to have delivered (and certainly received) all but the last n messages accepted by the source. Hence, u has certainly received all but at most the last n messages received by v. We therefore modify the rule above; if v received more than $c_r + n$ messages upon receiving the update packet from u, then v sends to u only the last n messages received. Essentially, if the c_r received from u is very small (compared to the number of messages received by v), than v 'knows' that both v and u received messages after u sent the update packet containing c_r. Hence, it always suffices to send at most the last n messages received. In section 5.2 we show that this implies that the protocol has to store only the n last messages received.

3.3 Improving Complexities by Using Flood

It can be shown that the protocol presented so far is correct. Furthermore, if the network is always $O(n^2)-$Up, then the delay is $O(n^2)$. We now present a modification that achieves $O(n)$ delay in $O(n)-$Up networks.

The improvement is by adding a *flooding* mechanism. In the code, appearing in Figs. 8-11, the lines which implement the flood mechanism are marked by F on the right margin.

The flood mechanism is an application of the Intelligent Flood protocol [Per83] or PI protocol [Seg83]. The flood mechanism uses a new type of packets, called *flood packets.* A flood packet, like a sync packet, consists of a pair of a message and its index.

The flood packets are generated by the source. Whenever the source accepts message number i, it sends a flood packet containing the message and number i to all of its neighbors <I5>. This is in addition to the sync packet with the same message and number, which is sent as before <H5>.

Whenever a processor v receives a flood packet whose number is one more than the highest number in a flood packet sent by v <I2>, then this processor sends this flood packet to all neighbors <I5>. (This is exactly the operation of the Intelligent Flood protocol.)

The idea is that flood packets normally propagate much faster than sync packets. Therefore, we use the flood packets for fast dissemination of the messages to the processors. Once the messages are accepted at a processor, they may be used to construct new sync packets. In this way, the flood packets may speed up the sync mechanism.

More specifically, recall that by the sync mechanism, a processor delivers message $i+1$ to the higher layer, and at the same time sends sync packet number $i+1$ to its neighbors, only after having received sync packet i from *all* neighbors and message $i+1$ from *some* neighbor. In the absence of flood packets, message $i+1$ would be received only in a sync packet; therefore in order for v to send sync packet $i{+}1$, and deliver message $i{+}1$, it has to receive sync packet $i{+}1$ from some neighbor u. But in order for u to send sync packet $i+1$, it must first *receive* sync packet i from *all* of its neighbors... The flood mechanism allows u to send message $i+1$ to v immediately upon receiving it, in a flood packet, without delaying and waiting for other neighbors of u. In this manner, the flood mechanism speeds-up the protocol. Of course, u

would still send v sync packet $i+1$ after receiving sync packet i from all of its neighbors.

We now describe how the flood mechanism deals with recoveries. As described in §3.2, whenever a link recovers, the two processors connected to it exchange an update packet. The update packet contains two counters: the deliver-counter c_d and the receive-counter c_r. The flood mechanism is concerned only with the receive counter c_r, i.e. the number of messages received by the processor. When a processor receives an update packet from a neighbor, it sends back any flood packet received with number higher than the receive-counter c_r in the packet <G2>. (There is no need to send also sync packets with the same numbers, as in the original description; indeed the code does not send them in this case.)

3.4 Towards Bounded Congestion, Counters and Storage

The protocol as described so far is correct, when implemented with unbounded counters in the packets and in the program. However, recall that in Section 5 we intend to present enhancements to the protocol which achieve bounded storage and counter. The analysis of these enhancements is simplified by reducing executions to executions of the 'simplified' protocol described in this section. We now present two minor modifications to the protocol as presented so far, the aim of which is to facilitate the reductions in Section 5. In addition, the second modification below is required to ensure bounded congestion.

In order to use modular counters in its internal computations, the maximal difference between two values compared must be bounded. One of the values kept by the protocol at v, for every neighbor u, is $D'_v(u)$: the highest number of sync packet or deliver counter in an update packet received from u. Under 'normal' conditions, when u and v are connected by a $3n-$Up link, this value is the actual number of packets delivered by u, possibly minus one. However, while u is disconnected, v would not receive sync or update packets from u, and therefore this value could become *'outdated'* - much smaller than the actual number of messages delivered by u or v.

In order to be sure that we do not compare this 'outdated' value to other values in v, e.g. to the number of messages delivered by v denoted D_v, we modify the protocol slightly. When the link to u falls, we set the value of $D'_v(u)$ to a special value, denoted undef. This ensures that processors never keep 'outdated' values.

The second modification deals with a subtle aspect of the data link control protocol. This aspect is due to the counter-intuitive fact that the link does not recover at both ends at the same time. Therefore, the end of the link where recovery occurred earlier, say u, may start sending packets to the other end, say v, before the link (u, v) recovered at v. These messages would be received by v after the link (u, v) recovers. The problem is that, due to the asynchronous model, there is no bound on the number of packets which may be thus buffered in the link. In principle, the rest of the network may operate much faster than this recovering link, so that while the link has still not recovered at v, processor u would send an unbounded number of packets over the link. This appears to require unbounded link capacity, and also v should be able to deal, after recovery, with 'old' packets - e.g., sync packets numbered much less than the number of deliveries in u at that time.

The solution is simple: we ensure that (update, sync or flood) packets received by v following a recovery of (u, v) at time t, were also sent by u after t. This additional 'data-link' property is natural and useful, and allows us to guarantee certain relations between the contents of the packets received after a recovery and the state of the processor at the recovery.

To ensure that update, sync and flood packets were not sent before the recovery (time t above), we introduce another kind of packet, called *recover packet*. This packet would be sent (only) immediately upon recovery. Furthermore, no update, sync or flood packet would be sent to the neighbor until receiving the recover packet from it. In the analysis (Lemma 9) we show that this ensures that any packet, except recover, received following a recovery at time t was also sent after t.

Note that the new property essentially defines a new state for the link: 'partially operational (real only)' or 'recovery in progress'. In order to achieve the new property, the link is first declared to be in this new state - partially operative - where packets may only be received. Only later (after the other end is known to have recovered) the link is declared fully operational (allowing the processor to send packets to the other end). Therefore, if we wanted to add this property to the data link protocol, we would be forced to (slightly) modify its interface.

We note that the fact that the protocol has bounded congestion - concurrent number of messages sent over a link - follows from the modification above, which bounds the number of messages in transit during link recovery, and from the fact that at most n 'old' messages are sent upon recovery (as justified at the end of subsection 3.2).

Variables: $\begin{cases} R_v & \text{number of messages received by } v\text{, or accepted for } v = s \\ D_v & \text{number of messages delivered by } v\text{to the higher layer} \end{cases}$

WHEN: (only in s) Accepting a new message, THEN: Send the new message and its number to all neighbors, in both sync and flood packets.

WHEN: (only in s) Holds $R_s \leq D_s$, THEN: Enable a ready event.

WHEN: Holds $D_v < R_v$, and for every neighbor u of v such that the link to u is up at v since the last deliver event at v holds that v has received from u, after (u, v) has last recovered at v, an update packet with deliver-counter $\geq$ D_v, or a sync packet numbered D_v, THEN: Deliver message $D_v + 1$ and send it in a sync packet to all neighbors w such that a recover packet was received from w since (v, w) has last failed.

WHEN: Receiving a flood packet with number $R_v + 1$, THEN: Send this packet to all neighbors w such that a recover packet was received from w since (v, w) last failed.

WHEN: Link to u recovers, THEN: Send to u a recover packet.

WHEN: Receiving a recover packet from u, THEN: Send to u an update packet, with receive-counter R_v and deliver-counter D_v.

WHEN: Receiving an update packet from neighbor u, THEN: If the receive-counter $<$ R_v, then send to u flood packets containing the messages numbered between the receive-counter and R_v; if the gap is larger than n, send only the last n messages.

Figure 1: Concise description of the algorithm at processor v.

4 Analysis of the Protocol

We now analyse the protocol of Figs. 8-11. The following theorem states the properties of the protocol, as proved in the rest of this section. Note that the complexities are guaranteed only if the network is always $3n-$Up, i.e. if at any time t there is a spanning tree whose links were up during the entire interval $[t-3n, t]$.

Theorem 1 *Every execution of the protocol in Figs. 8-11 is a correct broadcast from s. Furthermore, if the network is $3n-$ Up at all times, then the following holds:*

- *The delay is at most $3n$.*
- *The congestion is $O(n)$.*
- *The communication complexity over intervals of length $3n+3$ is $C_A = 4m$ per accept, $C_F = 0$ per fail and $C_R = 2$ per recovery.*

Note that the communication complexity bounds the number of packets sent for each message accepted and for each link recovery, when averaged over intervals of length at least $3n+3$. Namely, it may be possible, although improbable, that all of the packets would be sent at the same instance in the $3n+3$ interval. Also note that the messages are not short, in fact they contain counters which are unbounded in the size of the network. (This would be fixed in the next section.)

Theorem 1 above does not yield good bounds on the space complexity or throughput[4], and also does not claim that the protocol uses only short messages. We claim these properties only in section 5, for a slightly modified version of the protocol.

Organization of the proof. The proof begins with local properties of the protocol, which concern only one processor or one link. We then prove the correctness of the protocol. The rest of the analysis proves the stated complexities.

The complexities are based on the combination of two complementing mechanisms of the protocol. First, we analyse the progress mechanism in the protocol, which guarantees that messages are delivered. Then, we analyse the synchronization mechanism, which ensures that the number of messages

[4]A trivial bound for the throughput is that it is one over the delay, i.e. $\frac{1}{3n}$. However this is a rather low throughput; in Section 5 we achieve an $\Omega(1)$ throughput.

delivered by different processors is always roughly the same. Finally, we combine the analysis of the two mechanisms, and prove Theorem 1.

Notations: We denote the value of variable X_v in processor v at time t by $X_v(t)$. For example, $R_v(t)$ is the value of R_v at time t. We denote the value of variable X_v in processor v just before (after) time t by $X_v(t-)$ (respectively, $X_v(t+)$).

4.1 Local Properties.

In this subsection we present several properties of the protocol, which are 'local' to one processor or link. These properties are later used in the rest of the analysis. The following Lemma shows the validity of most descriptions of the variables given in Fig. 9.

Lemma 2 *The following hold at any time and at any processor v:*

A) *The value of D_v is the number of messages delivered by processor v to the higher layer, and it is also the number of sync packets sent by v.*

B) *The value of A_s is the number of messages accepted from the higher layer.*

C) *The set G_v contains the neighbors u of v such that (u, v) is up at v.*

D) *The set G_v^D contains the neighbors u of v such that (u, v) is up at v since v last delivered a message.*

E) *The set G_v^R contains the neighbors u of v such that (u, v) is up at v and v received a recover packet from u and sent an update packet to u since the last recovery of (u, v) at v.*

Proof: All properties follow immediately by considering the statements in the protocol which modify the relevant variables. □

The next Lemma states that the values of R_v, D_v and A_s are non-decreasing. We later show that the value of $D'_v[u]$ is also non-decreasing.

Lemma 3 *For any processor v, the values of the variables R_v and D_v are non-decreasing. The variable A_s of the source processor is also non-decreasing.*

Proof: The values of R_v, D_v and A_s change only when incremented in <I3>, <H3> and <A2> respectively. □

We next observe that $D_v \leq R_v$.

Lemma 4 *For any processor v and at any time holds $D_v \leq R_v$.*

Proof: Variable D_v changes only when incremented by one in <H3>. The claim follows from the condition $D_v < R_v$ in <H2>. $\square$

We now analyze the recover mechanism. The following two Lemmas observe that the first (second) packet received at every up interval is always a recover (respective, update) packet.

Lemma 5 *The first packet received at processor v from link (u, v) during any up interval of (u, v) at v is always a recover packet.*

Proof: From the FIFO property, the first packet received by v during an up interval of (u, v) at v is the first packet sent by u to v at some up interval of (u, v) at u. The first packet sent by u to v at some up interval of (u, v) at u is always a recover packet <E2>. The claim follows. $\square$

Lemma 6 *The second packet received at processor v during any up interval of link (u, v) at v is always an update packet. Furthermore, from the time when u sends this packet and until the link (u, v) next fails at u holds $v \in G^R_u$.*

Proof: Let p be the second packet received by v from u during some up interval of link (u, v) at v. From the FIFO property, p was sent by u to v during some up interval of (u, v) at u. We show that p must be an update packet. Obviously, p cannot be a recover packet, since a recover packet is sent only upon recovery. It remains to show that p cannot be a sync or flood packet.

Sync packets are sent in <H5> and flood packets are sent in <I5> and <G2>. We first show that p is not sent by u to v in <H5> or <I5>. A necessary condition for p to be sent by either <H5> or <I5> is $v \in G^R_u$ at the time. We show that when p is sent $v \notin G^R_u$. Lemma 2 E) shows that $v \in G^R_u$ only if u sent an update packet to v during the same up interval of (u, v) at u. But since p is the second packet sent by u during this up interval, and the first packet is always a recover packet, it follows that p is not sent by <H5> or <I5>.

It remains to show that p is not sent by <G2>. Processor u sends flood packet to v by <G2> upon receiving an update packet from v. From Lemma 5, previously during this up interval, processor u received a recover packet from v. Upon receiving a recover packet from v, processor u sends an update packet to v <F2>. Hence, p cannot be sent by <G2>. $\square$

We conclude that the value of $D'_v[u]$ increases by exactly one whenever v receives a sync packet from u.

Lemma 7 *Whenever a processor v receives a sync packet from its neighbor u then the value of $D'_v[u]$ increases exactly by one.*

Proof: Suppose that v receives a sync packet from u at time t; we want to show that $D'_v[u](t+) = D'_v[u](t-) + 1$. Let t_u be the time when v received the last update packet from u before t, and let c_d be the value of the deliver-counter in this packet. Let i be the number of sync packets which v received from u during $(t_u, t]$. We prove by induction on i that $D'_v[u](t+) = c_d + i$; the Lemma follows.

From <G3> follows that $D'_v[u](t_u+) = c_d$. Therefore, it suffices to show that if $D'_v[u](t-) = c_d + i - 1$ then $D'_v[u](t+) = c_d + i$. From <H3> and <H5>, the i^{th} sync packet sent by u to v since t_u is numbered $c_d + i$. The claim follows from the FIFO property. $\square$

The following Lemma shows that at most two time units since a link (u, v) recovers at v, it either fails or v receives the recover and update packets from u.

Lemma 8 *Suppose that link (u, v) recovers at processor v at time t and stays up until $t + 2$. Then during $[t, t + 2]$ processor v receives a recover and an update packet from u.*

Proof: Upon recovery at time t, processor v sends a recover packet to u <E2>. Since the delay is at most one time unit and link (u, v) stays up at v until $t + 2$, follows that u receives this recover packet during $[t, t + 1]$.

Upon receiving the recover packet from v, processor u sends to v an update packet <F2>. Again, since the delay of the recover packet is at most one time unit and since (u, v) stays up at v till $t + 2$, follows that v receives this update packet during $[t, t + 2]$.

From Lemma 5, the first packet received after a link recovers is always a recover packet. Hence, during $[t, t + 2]$ and before v receives the update packet, processor v receives also a recover packet from u. $\square$

The purpose of the recover packet is to ensure that the other packets received from u has been sent by u after the time when link (u, v) has last recovered at v (see Fig. 2). Namely, we want to ensure that *both* processors are aware of the recovery before either one of them sends any (non-recover) packet over the link. This property, while natural, is *not* guaranteed by the data link protocol (compare to the crossing property), and therefore we prove it below.

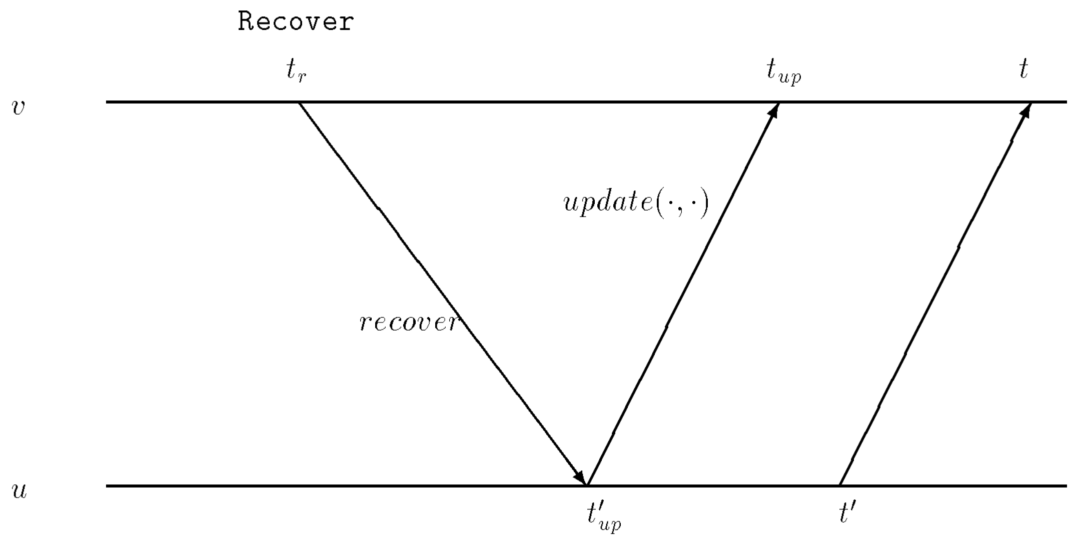

Figure 2: The operation of the recover mechanism.

Lemma 9 *Suppose that processor v receives some packet other than recover from neighbor u at time t. Then link (u, v) is up at v continuously since this packet was sent by u and until time t.*

Proof: Let t_r denote the time when link (u, v) has last recovered at v before t. Namely, link (u, v) is up at v during $[t_r, t]$. From Lemmas 5 and 6, at some time t_{up} during $[t_r, t]$, processor v receives an update packet from u. Denote the time when processor u sent the packet received by v at time t (t_{up}) by t' (respectively, t'_{up}). From the FIFO property of the link holds $t'_{up} \leq t'$.

Update packets are sent only upon receiving a recover packet <F2>, and recover packets are sent only upon recovery <E2>. Hence, from the crossing property of the link at time t'_{up} processor u received the recover packet sent by v at t_r. Thus, $t_r \leq t'_{up}$ (see Fig. 2). Since $t'_{up} \leq t'$ and (u, v) is up at v during $[t_r, t]$, the claim follows. □

The next several Lemmas show that $D'_v[u]$ is indeed a lower bound estimate for D_u, as stated in Fig. 9. We first show that $D'_v[u]$ is the value of

D_u at the time when u has sent to v the last sync or accept packet received by v from u.

Lemma 10 *Consider any link (u,v) and time t. Let t_u denote the last time before t when processor v receives an update or sync packet from neighbor u. Let t'_u denote the time when processor u has sent the packet received by v at time t_u. Then either $D'_v[u](t) = D_u(t'_u)$ or $D'_v[u](t) = \mathsf{undef}$ and (u,v) failed at v during $[t_u, t]$.*

Proof: The value of $D'_v[u]$ changes only in <D3>,<B2> and <G3>. If (u,v) fails at v during $[t_u,t]$ then $D'_v[u]$ is last changed in <D3> to undef, and the claim follows. Otherwise, the value of $D'_v[u]$ does not change during $(t_u,t]$, namely $D'_v[u](t) = D'_v[u](t_u+)$. It remains to show that in this case $D'_v[u](t_u+) = D_u(t'_u)$.

Either a sync or an update packet is received at t_u. Sync packets are sent only in <H5>. $i = D_u(t'_u)$. Therefore, if the packet received at t_u is a sync packet, then it was sent with $i = D_u(t'_u)$. In this case the claim follows from <B2>.

Update packets are sent only in <F2>. Therefore, if the packet received at t_u is an update packet, then it was sent with $c_d = D_u(t'_u)$. In this case the claim follows from <G3>. □

We conclude that at any time either $D'_v[u] \leq D_u$ or $D'_v[u] = \mathsf{undef}$. Furthermore, as long as (u,v) does not fail in v, the value of $D'_v[u]$ is non-decreasing.

Lemma 11 *For any link (u,v) and at any time, either $D'_v[u] \leq D_u$ or $D'_v[u] = \mathsf{undef}$. Furthermore, $D'_v[u]$ is non-decreasing during up intervals of (u,v) at v.*

Proof: The claims are immediate from Lemma 10, in view of the fact that packets are received in the order in which they were sent. □

We next observe that:

$$D'_v[u] \neq \mathsf{undef} \Rightarrow D_u(t-1) \leq D'_v[u](t). \tag{1}$$

This shows that $D'_v[u]$ is a meaningful estimate of D_u whenever $D'_v[u] \neq$ undef.

Lemma 12 *At any time t, either $D_u(t-1) \leq D'_v[u](t)$ or $D'_v[u](t) = \mathsf{undef}$.*

Proof: Suppose that $D'_v[u](t) \neq \textsf{undef}$. Hence, at some time $t_{up} \leq t$ processor v receives an update packet from u and (u,v) is up at v during $[t_{up}, t]$ (see Fig. 3). Let t'_{up} denote the time when u has sent the update packet received by v at time t_{up}. From Lemma 9, link (u,v) is up at v during $[t'_{up}, t_{up}]$. Hence (u,v) is up at v during $[t'_{up}, t]$.

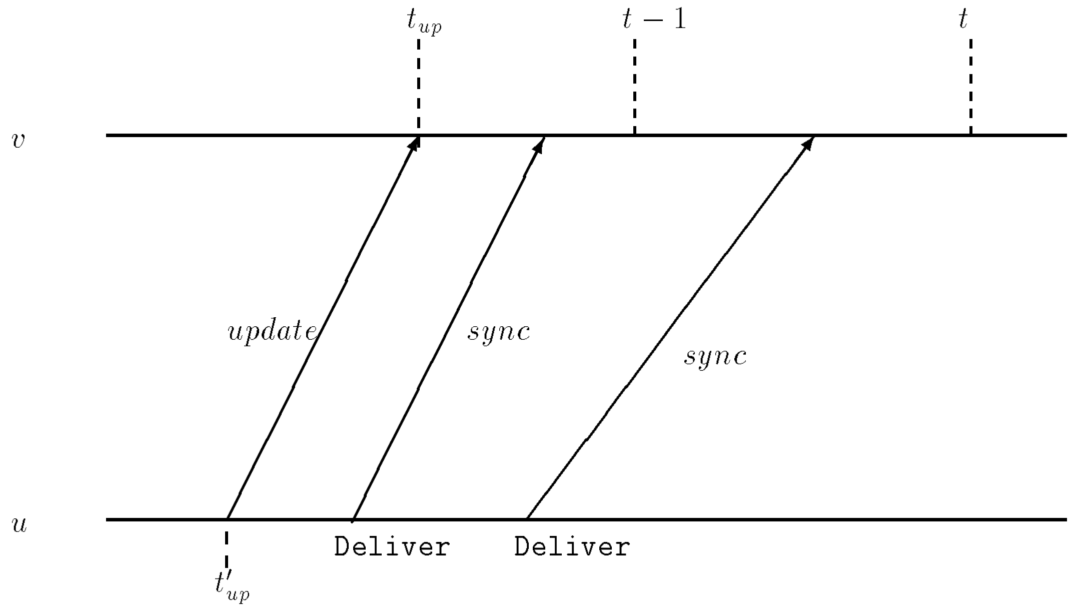

Figure 3: Proof that if $D'_v[u] \neq \textsf{undef}$ then $D_u(t-1) \leq D'_v[u](t)$.

From Lemma 10 holds $D_u(t'_{up}) = D'_v[u](t_{up})$. Since $D'_v[u]$ is non-decreasing during up intervals of (u,v) at v (Lemma 11), holds $D'_v[u](t_{up}) \leq D'_v[u](t)$. If $t-1 \leq t'_{up}$, then the claim follows since D_u is non-decreasing and hence holds $D_u(t-1) \leq D_u(t'_{up}) \leq D'_v[u](t_{up}) \leq D'_v[u](t)$. Assume, therefore, that $t'_{up} < t-1$.

From the follow-up property, link (u,v) is up at u during $[t'_{up}, t-1]$, since it is up at v during $[t'_{up}, t]$. From Lemma 6 follows that $v \in G^R_v$ during $[t'_{up}, t-1]$. Hence processor u sends $sync(\cdot, D_u)$ to v at any time when D_u is increased during $[t'_{up}, t-1]$ <H3>. From the deliver property of the link, processor v receives these sync packets before time t. In particular, if

$D_u(t'_{up}) < D_u(t-1)$, then processor v receives $sync(\cdot, D_u(t-1))$ before time t. From <B2> and since $D'_v[u]$ is non-decreasing, holds $D_u(t-1) \leq D'_v[u](t)$, i.e. the claim holds. The claim also holds if $D_u(t'_{up}) = D_u(t-1)$ since $D_u(t'_{up}) \leq D'_v[u](t_{up}) \leq D'_v[u](t)$. □

We now conclude that if link (u, v) is up at v during $[t-2, t]$, then Eq. (1) holds at time t.

Lemma 13 *If link (u, v) is up at v during $[t-2, t]$ then $D_u(t-1) \leq D'_v[u](t) \neq$ undef.*

Proof: From Lemma 12, it suffices to show that $D'_v[u](t) \neq$ undef. Let t_r denote the last recovery of link (u, v) at v prior to $t-2$. Namely, link (u, v) recovers at processor v at time t_r and stays up until t, where $t_r \leq t-2$. From Lemma 8, within two time units after recovery either the link fails or an update packet is received. Since the link does not fail until $t_r + 2$, processor v receives an update packet from u between t_r and $t_r + 2$, which is before t. The claim follows from <G3>. □

We now state the relation between R_v and the indices in sent and received packets.

Lemma 14

A) Just before processor v sends either $sync(m, i)$ or $flood(m, i)$, the following holds: $M_v[i] = m$ and $R_v \geq i$.

B) If $i \leq R_v(t)$ then one of the following holds:

1. *v has received before t packet $sync(M_v[i], i)$ or packet $flood(M_v[i], i)$.*
2. *$v = s$ and before time t message $M_s[i]$ was accepted in the i^{th} accept event.*

Proof: Lemma 14 A) follows immediately from the fact that sync packets are sent only in <H5> and flood packets are sent only in <I5> and <G2>.

We now prove Lemma 14 B). The variable R_v is incremented only in procedure *message* <I3>, when called at some time t with parameter $R_v(t-) + 1$. Procedure *message* is called only in <B3>, <C2> or <A3>. In Statement <B3> and <C2>, $message(m, i)$ is called only after s has received a packet with index number i, and thus the claim follows. Therefore it suffices to consider the case where *message* is called in <A3>. In this case, the claim follows from Lemma 2 B), which says that A_s is the number of accept events. □

4.2 Correctness

The correctness of the protocol is based on the simple mechanism of associating indices to messages and including them in sync and flood packets. We now show formally that the association is correct.

Lemma 15 *If processor v receives $sync(m, i)$ or $flood(m, i)$, then m was the i^{th} message accepted by the source s.*

Proof: Every packet received by some processor was previously sent by some other processor. Consider the first send event, at some processor u, of a packet containing message m and counter i. Let t denote the time of this send event. From Lemma 14 A) holds $R_u(t-) \geq i$ and $M_v[i] = m$. From Lemma 14 B) either $u = s$ and m was the i^{th} message accepted, or processor u has received before t a packet containing m and i. However, processor u could not have received such a packet before t, since then the packet must have been sent before t, and t is the time of the first send event of packets containing m and i. $\square$

It is now easy to prove the correctness of the protocol.

Lemma 16 *Every execution of the protocol, as specified in Figs. 8-11, is a* correct broadcast from *s.*

Proof: Messages are delivered to the higher layer only in <H3>. Hence, the i^{th} message delivered by processor v is the contents of $M_v[i]$, and when the latter is delivered, holds $i = D_v$. From Lemma 4 holds $D_v \leq R_v$. Hence, from Lemma 14 B) either $v = s$ and $M_v[i]$ is indeed the i^{th} message accepted by the source, or v has previously received $sync(M_v[i], i)$ or $flood(M_v[i], i)$. The correctness follows from Lemma 15. $\square$

Lemma 15 allows us also to show that $R_s = A_s$ and that for every processor v, we have $R_v \leq A_s$. These relations are useful in the rest of the analysis.

Lemma 17 *At any time $R_s = A_s$ holds, and $R_v \leq A_s$ holds for every $v \in V$.*

Proof: The fact that $R_v \leq A_s$ follows since v sets R_v to i only after having received $sync(\cdot, i)$, which can happen only after the time when the i^{th} message is accepted at the source s.

The claim $R_s = A_s$ follows by induction on the events. Obviously, the claim holds before the first event. Consider some event e at time t, and assume that for any $t' < t$ holds $R_s(t') = A_s(t')$. If e is an accept event, then

A_s is incremented in <A2> and procedure $message(m, i)$ is called in <A3>, with m the accepted message and $i = A_s(t+) = R_s(t-)+1$. The claim holds after e since R_s is incremented as well, in <I5>, since $i = R_s(t-) + 1$ as required by <I2>.

It remains to consider the case when e is not an accept event. In this case, the value of A_s does not change. We now show that the value of R_s also does not change. The value of R_s changes only when <I5> is executed at s, as a result of a call to $message(m, R_s+1)$ <I2>. Such a call is possible only if s has previously received either $sync(m, R_s+1)$ or $flood(m, R_s+1)$, by <B3> or <C2>. From Lemma 15 above, this occurs only *after* the R_s+1 accept event. The claim follows since A_s is the number of messages accepted (Lemma 2 B)). □

4.3 The progress mechanism.

The complexities of the protocol are ensured by two complementing mechanisms. The *synchronization mechanism* ensures that no processor is delivering 'much more' messages than other processors, i.e. it delays the delivery of messages in the 'faster' processors to ensure that processors are loosely synchronized. The *progress mechanism* ensures that each message is indeed delivered by all processors within finite time after it is accepted by the source.

It is trivial to achieve either synchronization or progress; the difficulty is to achieve both properties together. Synchronization alone is trivially achieved, for example, if no messages are delivered, or by the PIF protocol. Progress alone is achieved by the intelligent flood protocol [AE86, Per83, Seg83].

We begin by presenting a weak progress property, that does not use the flood mechanism. Namely, we show that if processor v receives before t at least one message more than the number of messages delivered by the slowest processor, then v delivers before $t + 2$ more messages than these delivered by the slowest processor at t.

Lemma 18 *For every processor v and time t, if $R_v(t) > \min_{w\in V} D_w(t)$, then $D_v(t+2) > \min_{w\in V} D_w(t)$.*

Proof: We prove the claim by contradiction. Let v be a processor such that $D_v(t+2) \leq \min_{w\in V} D_w(t)$. Since D_v is non-decreasing, it follows that $D_v(t+2) = D_v(t) = \min_{w\in V} D_w(t)$. We show that $R_v(t) \leq D_v(t) = \min D_w(t)$, from which the claim follows.

Consider some $u \in G_v^D(t+2)$ (if $G_v^D(t+2)$ is not empty). Namely, link (u, v) is up at v from the time when v has last delivered a message prior to $t+2$ and until time $t+2$ (Lemma 2 D)). Since D_v does not change between t and $t+2$, processor v delivers no messages during $[t, t+2]$. Hence, link (u, v) was up at v during $[t, t+2]$. From Lemma 13 follows that $D'_v[u](t+2) \geq D_u(t+1)$ and $D'_v[u](t+2) \neq \mathsf{undef}$. Also, since R_v is non-decreasing, $R_v(t) \leq R_v(t+2)$. Consider the first time t' such that during $(t', t+2]$ none of G_v^D, $D'_v[u]$ or R_v changes. (Note that t' may be before or after t.) Namely:

- For every $t'' \in (t', t+2]$, holds $R_v(t) \leq R_v(t'')$, and for every processor u in $G_v^D(t'')$ holds $D'_v[u](t'') \neq \mathsf{undef}$ and $D'_v[u](t'') \geq D_u(t+1)$.
- Either $R_v(t'-) < R_v(t+2)$ or there is some u in $G_v^D(t'-)$ such that $D'_v[u](t'-) < D_u(t+1)$ or $D'_v[u](t'-) = \mathsf{undef}$.

By definition of t', one of the following holds:

- The value of R_v increases at t'. This can happen only in <I3>.
- A processor is removed from G_v^D at t'. This can happen only in <D2>.
- The value of $D'_v[u]$ is increased at t'. This can happen in either <B2> or <G3>.

It is easy to see that in all cases, procedure *proceed* is executed at t'. The second condition of <H2> holds at $t'+$. Hence, after t', the first condition of <H2> does not hold. Namely, $D_v(t'+) \geq R_v(t'+)$. Since $t' \leq t+2$, holds $D_v(t'+) = D_v(t+2) = D_v(t)$. On the other hand, $R_v(t) \leq R_v(t'+)$. Hence $R_v(t) \leq D_v(t)$. $\square$

We now extend Lemma 18 to longer time periods.

Lemma 19 *For every processor v, time t and $i \geq 1$:*

$$D_v(t) \geq \min\{D_v(t-2i) + i, \min_{w \in V} R_w(t-2i)\} \tag{2}$$

Proof: Lemma 18 shows that the claim holds for $i = 1$, since $D_v \leq R_v$. The induction step follows by another application of Lemma 18. $\square$

The stronger progress properties, to be proven later, hold in conjunction with the synchronization mechanism. As shown later, this combination of the progress mechanism and the synchronization mechanism ensures that the synchronization condition (Def. 16) holds at any time. Namely, $A_s(t) \leq$

$D_v(t) + n$ for any time t. The progress properties stated below require the synchronization condition to hold at certain times.

We now prove a simple property of the flood mechanism used in the protocol. This property shows that messages are indeed 'flooded'. Namely, if u_k and u_0 are $O(k)-$Up-Connected at time t, then every flood packet sent by u_k at $t-k$ is received by u_0 before t.

Lemma 20 (Flood) *Consider the path $u_0 - u_1 - \cdots - u_{k-1} - u_k$ consisting of k links. If the path is $(k+2)-$Up at some time t, and the synchronization condition holds during $[t-(k+2), t]$, then $R_{u_0}(t) \geq R_{u_k}(t-k)$.*

Proof: This Lemma extends Theorem PI-1 of [Seg83] to deal with recoveries. We first prove the Lemma for $k = 1$. Namely, we assume that link (u_0, u_1) is $3-$Up at t, and prove that $R_{u_1}(t-1) \leq R_{u_0}(t)$.

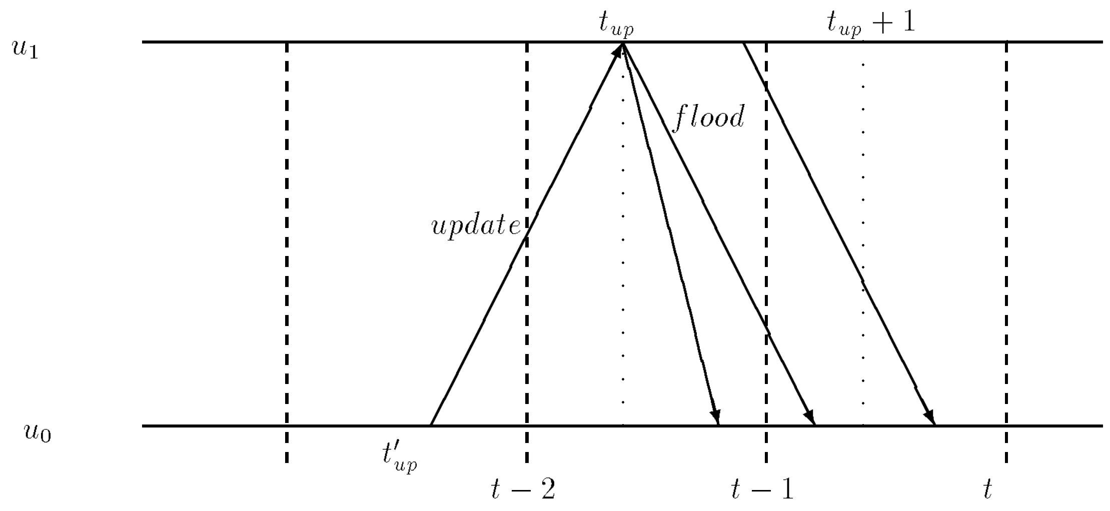

Figure 4: The flood mechanism.

From Lemma 8, processor u_1 receives an update packet from u_0 at some time t_{up} before $t-1$. Let t'_{up} denote the time when u_0 sends this update packet to u_1 (see Fig. 4).

Upon receiving the update packet, processor u_1 executes <G2> with $c_r = R_{u_0}(t'_{up})$. Hence, at time t_{up}, processor u_1 sends to u_0 flood packets numbered $\max\{R_{u_0}(t'_{up}) + 1, R_{u_1}(t_{up}) - n + 1\}$ to $R_{u_1}(t_{up})$ <G2>. Since (u_0, u_1) does not fail until t, and the delay is at most one time unit, processor u_0 receives these packets before $t_{up} + 1 \leq t$. From the Synchronization condition (Def. 16) holds $R_{u_0}(t_{up}) \geq A_s(t_{up}) - n \geq R_{u_1}(t_{up}) - n$. Hence, from <I3> follows that $R_{u_0}(t_{up} + 1) \geq R_{u_1}(t_{up})$.

Since R_{u_1} increases only in <I3>, then processor u_1 sends to u_0 flood packets numbered $R_{u_1}(t_{up}) + 1, \ldots, R_{u_1}(t-1)$ between t_{up} and $t-1$. Since the link does not fail until t and the delay is at most one, all of these packets are received by u_0 before t. From <I3>, and the fact that $R_{u_0}(t_{up} + 1) \geq R_{u_1}(t_{up})$, follows that $R_{u_1}(t-1) \leq R_{u_0}(t)$. This proves the claim for $k = 1$.

By repeating the inequality $R_{u_1}(t-1) \leq R_{u_0}(t)$, we get

$$R_{u_k}(t-k) \leq \ldots \leq R_{u_1}(t-1) \leq R_{u_0}(t)$$

which proves the Lemma. $\square$

Recall that the synchronization condition holds at time t if the number of messages accepted by the source until t is at most n more than the number of messages delivered by any processor (Def. 16). We now prove the Progress Lemma based on the Flood Lemma, assuming that the Synchronization Condition holds. Later we show that this condition is always satisfied.

Lemma 21 (Progress) *Assume that the network is $(n+1)-$Up at $(t-2n)$ and that the synchronization condition holds during $[t-3n-1, t-2n]$. Then every processor delivers until time t every message accepted until $(t-3n)$, i.e. for every $v \in V$ holds $A_s(t-3n) \leq D_v(t)$.*

Proof: The proof consists of two parts. We first show, using the Flood Lemma, that every processor receives until time $(t-2n)$ flood packets containing every message accepted until time $(t-3n)$. Namely, $A_s(t-3n) \leq \min_{w \in V} R_w(t-2n)$. Second, we use Lemma 19 to show that until t, every processor delivers each of these messages. Namely, $R_v(t-2n) \leq D_v(t)$.

The network is $(n+1)-$Up at time $(t-2n)$. Hence, there is an $(n+1)-$Up path consisting of at most $(n-1)$ links from s to every other processor v at time $(t-2n)$. Also, we have assumed that the synchronization condition holds during $[t-3n-1, t-2n]$. Hence, from the Flood Lemma, for every processor $w \in V$ holds $R_w(t-2n) \geq R_s(t-3n)$. The first part of the proof follows since always holds $A_s = R_s$ (Lemma 17).

From Lemma 19, for every processor v holds $D_v(t) \geq \min\{D_v(t-2n)+n, \min_{w \in V} R_w(t-2n)\}$. Since $A_s(t-3n) \leq \min_{w \in V} R_w(t-2n)$ and $A_s(t-3n) \leq A_s(t-2n) \leq D_v(t-2n)+n$, follows that for every processor v holds $A_s(t-3n) \leq D_v(t)$. $\square$

4.4 The synchronization mechanism.

We now analyze the synchronization mechanism. We begin with a simple synchronization property, which shows, loosely speaking, that a processor delivers new messages only after all neighbors in G_v^D deliver the previous message. Recall that neighbor u is in G_v^D if the link (u,v) is up at v since v has last delivered a message to the higher layer.

Lemma 22 *Assume that link (u,v) is up at v during $[t_1,t_2]$. If processor v delivers more than one message during $[t_1,t_2]$, then $D_v(t_2) \leq D'_v[u](t_2)+1$.*

Proof: The intuition behind the proof is that at some time before the $(D_v(t_2))^{th}$ deliver event in v, processor v has already received sync packet number $D_v(t_2)-1$ from u.

Let t' be the time when v last delivers a message before t_2. Holds $t_1 < t' \leq t_2$. Since v delivers at least two messages during $[t_1,t_2]$, then v delivers at least one message during $[t_1,t']$. Link (u,v) is up at v during $[t_1,t_2]$, hence $u \in G_v^D(t'-)$. From the second condition of <H2> holds $D_v(t'-) \leq D'_v[u](t'-)$. Also $D_v(t'+) = D_v(t'-)+1$, and D_v does not change between $t'+$ and t_2. Thus, since $D'_v[u]$ is non-decreasing, holds $D_v(t_2) \leq D'_v[u](t_2)+1$. $\square$

The synchronization mechanism ensures that the synchronization condition and the link synchronization condition, both of Def. 16, hold at any time t. We prove both synchronization properties together with the progress property of Lemma 21.

Lemma 23 (Synchronization and Progress) *Assume that the network is $3n-$Up at any time. Then the following properties hold at any time t:*

Link Synchronization: *The Link Synchronization condition holds. Namely, if link (u,v) is $3n-$Up at time t, then processor v delivers until t at most one more message than u delivers until t. In other words, $D_v(t) \leq D_u(t)+1$.*

Synchronization: *The Synchronization condition holds. Namely, $(\forall v \in V)\ A_s(t)-n \leq D_v(t)$.*

Progress: *Every processor delivers until time t every message accepted until $t-3n$. Namely, $(\forall v \in V)\ A_s(t-3n) \leq D_v(t)$.*

Proof: The proof is based on three main observations. First, from Lemma 21, the Progress property for time t follows from the synchronization condition until $t-2n$. Second, the Link Synchronization property for time t follows from the Progress property for time t and from Lemma 22. Last, the synchronization condition for time t follows easily from the Link Synchronization condition for time t. We now proceed with the proof.

The properties hold trivially for $t \leq 0$. We show that they hold for every t by assuming that they hold until some time t_0, and proving that they hold until time t_0+2n.

Consider any time t' between t_0 and t_0+2n. First, from Lemma 21, and since the synchronization condition holds until t_0, the progress property holds for time t'. Namely, $A_s(t'-3n) \leq D_v(t')$ for all $v \in V$. We now prove that the Link Synchronization property holds at time t'.

Consider some link (u,v) which is $3n-$Up at t'. From the Progress property for time t' follows that $A_s(t'-3n) \leq D_u(t')$. Since $D_v \leq A_s$ always, follows that $D_v(t'-3n) \leq D_u(t')$ holds.

If $D_v(t') \leq D_v(t'-3n)+1$, the Link Synchronization property for t' follows. On the other hand, if $D_v(t') > D_v(t'-3n)+1$, then v delivers between $(t'-3n)$ and t' more than one message. Thus Lemma 22 implies that $D_v(t') \leq D'_v[u](t')+1$. From Lemma 11 holds $D'_v[u](t') \leq D_u(t')$. Hence, the Link Synchronization property holds at t'.

It remains to prove that the Synchronization condition holds at t' as well, i.e. that for every processor v holds $D_v(t') \geq A_s(t')-n$. In the statement of the Lemma we have assumed that the network is $3n-$Up. Hence, there is a $3n-$Up path from v to s at any time. From the Link Synchronization condition, applied at time t' to each link along this path, follows that $D_s(t') \leq D_v(t')+(n-1)$, since this $3n-$Up path contains at most $n-1$ links. The Synchronization condition follows since $A_s \leq D_s+1$ <H7>. □

4.5 Bounded Congestion

In the following Lemmas we show that at any time, any link contains $O(n)$ packets in each direction. We begin by bounding the number of sync packets in transit. We first bound the number of sync packets in transit from u to v, if u has received at least two sync packets from v since (u,v) had last recovered at u (see Fig. 5).

Lemma 24 *Consider any link (u, v) and time t, and let t_r denote the time of the last recovery of (u, v) at u before t. Assume that processor u receives at least two sync packets from v during $[t_r, t]$. Then all but at most two sync packets sent by u to v during $[t_r, t]$ are received by v before time t.*

Proof: Let i, $i+1$ denote the sequence numbers of the last two sync packets received by u from v during $[t_r, t]$. Namely, $D'_u[v](t) = i + 1$.

The idea of the proof is that $i \leq D'_v[u](t)$ and $D_u(t) \leq D'_u[v] + 1 = i + 2$. Hence, at most two sync packets are in transit from u to v at time t, with sequence numbers $i + 1$ and $i + 2$. (See Fig. 5).

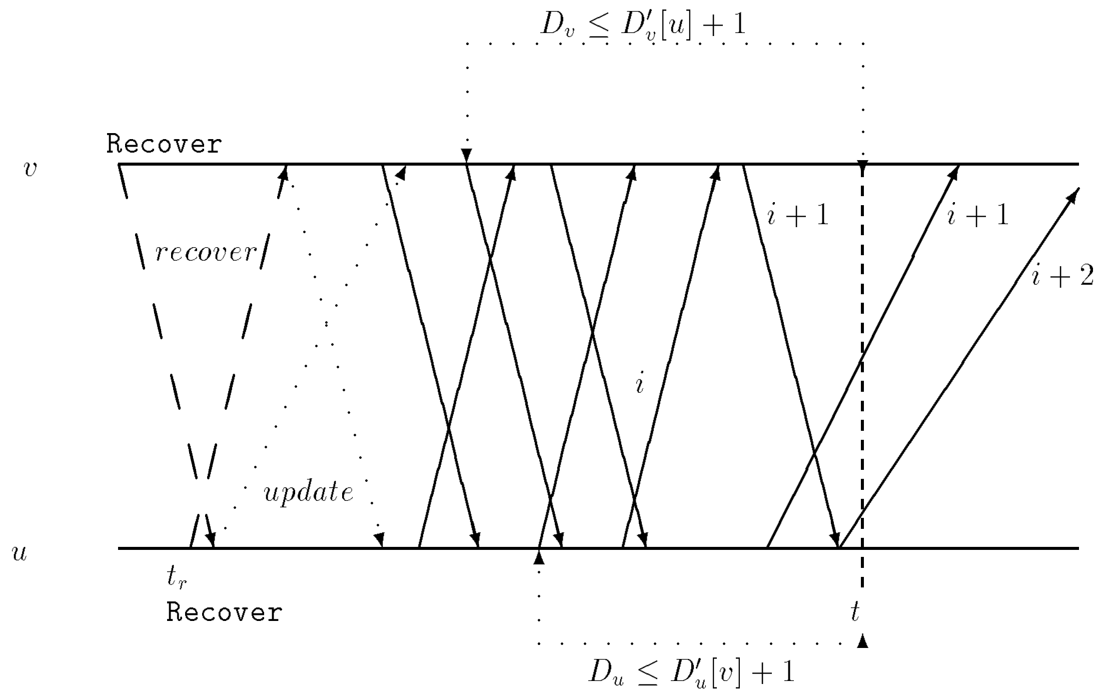

Figure 5: Number of sync-packets in transit if u received two or more sync packets during $[t_r, t]$.

From the FIFO property, link (u, v) did not fail or recover at processor v from the time when v sends sync packet i to u and until v sends sync packet $i+1$ to u. Hence, when processor v sends sync packet $i+1$ to u, executing <H5>, holds $u \in G^D_v$. From <H2>, at that time holds $i \leq D'_v[u]$. Since $D'_v[u]$ is non-decreasing during operating intervals (Lemma 11), holds $i \leq D'_v[u](t)$.

The claim is trivial if u sends less than two sync packets to v during $[t_r, t]$. Assume that u sends two or more sync-packets during $[t_r, t]$, and we show that $D_u(t) \leq D'_u[v] + 1 = i + 2$.

Whenever u sends a sync-packet, in <H5>, then u also delivers a message in <H4>. Hence before u sends the last sync packet to v until t, holds $v \in G^D_u$. From <H2>, at that time holds $D_u \leq D'_u[v]$. Since $D'_u[v]$ is non-decreasing and D_u changes only when u delivers messages, holds $D_u(t) \leq D'_u[v](t) + 1 = i + 2$. Since $i \leq D'_v[u](t)$, it follows that at most two sync-packets are in transit from u to v at time t, namely $sync(\cdot, i+1)$ and $sync(\cdot, i+2)$. $\square$

It remains to consider intervals where u receives less than two sync packets from v. We first prove some useful synchronization properties between the number of messages received and delivered in v and u.

Lemma 25 *If the network is always* $3n-$*Up, then the following holds at any time* t*:*

$$(\forall v \in V)\ R_v(t) \leq D_v(t) + n \tag{3}$$

$$(\forall v, u \in V)\ D_u(t) \leq D_v(t) + n \tag{4}$$

$$(\forall v, u \in V)\ (D'_v[u](t) \leq D_v(t) + n)\ \vee\ (D'_v[u](t) = \mathsf{undef}) \tag{5}$$

Proof: From Lemma 23 follows that $A_s - n \leq D_v$. From Lemma 17 holds $R_v \leq A_s$. This proves that $R_v \leq D_v + n$.

From Lemma 4, for every processor u holds $D_u \leq R_u$. Again from Lemma 17 holds $R_u \leq A_s$. Hence $D_u \leq D_v + n$. Equation (5) holds as well, since either $D'_v[u] \leq D_u$ or $D'_v[u] = \mathsf{undef}$, from Lemma 11. $\square$

We now bound the number of sync packets sent by u to v during intervals where u does not receive an update packet from v.

Lemma 26 *Assume that the network is* $3n-$*Up at any time. Consider any link* (u, v) *which is up at processor* u *during some interval* $[t_1, t_2]$*. Assume that processor* u *does not receive update packets from* v *during* $[t_1, t_2]$*. Let* i *denote the number of sync packets received by* u *from* v *during* $[t_1, t_2]$*. Then processor* u *sends to* v *during* $[t_1, t_2]$ *at most* $i + n + 1$ *sync packets.*

Proof: The proof is based on three observations. First, $D'_u[v](t_2) \leq D'_u[v](t_1) + i$, as we show later. Second, if processor u delivers more than one message during $[t_1, t_2]$, then $D_u(t_2) \leq D'_u[v](t_2) + 1$ (Lemma 22). Third, $D'_u[v](t_1) \leq D_u(t_1) + n$ (Eq. (5) of Lemma 25). The claim follows since the number of sync packets sent by u during $[t_1, t_2]$ is exactly $D_u(t_2) - D_u(t_1)$.

It remains to show that $D'_u[v]$ increases during $[t_1, t_2]$ by at most i. Since processor u does not receive update packets from v during $[t_1, t_2]$, then $D'_u[v]$ changes during $[t_1, t_2]$ only when u receives sync packets from v. From Lemma 7, during $[t_1, t_2]$ the value of $D'_v[u]$ increases exactly by i. $\square$

We now conclude that at any time there are at most $2n + 3$ sync packets in transit from u to v.

Lemma 27 *Assume that the network is always $3n-$Up. Consider any link (u, v) and time t, and let t_r denote the time of the last recovery of (u, v) at u before t. Then at most $2n + 3$ sync packets which were sent by u to v between t_r and t are not received by v until t.*

Proof: If processor u receives during $[t_r, t]$ two or more sync packets from v then the claim follows from Lemma 24. Assume, therefore, that u receives at most one sync packet from v during $[t_r, t]$.

The rest of the proof is organized as follows. We first show that processor u receives at most one update packet from v during $[t_r, t]$, say at time t_u. Then we apply Lemma 26 twice: first for the period $[t_r, t_u]$ and second for the period $[t_u, t]$.

Update packets are sent only when receiving a recover packet by <F2>. Recover packets are sent only upon recovery <E2>. Hence, from the FIFO property, processor u receives at most one update packet during $[t_r, t]$, say at time t_u.

From Lemma 26, the number of sync packets sent by u to v during $[t_r, t_u]$ ($[t_u, t]$) is at most $n + 1$ more than the number of sync packets received by u from v during $[t_r, t_u]$ (respectively, $[t_u, t]$). Since u receives from v at most one sync packet during $[t_r, t]$, the claim follows. $\square$

We now proceed to show that the maximal number of flood packets in transit over a link is also $O(n)$.

Lemma 28 *Assume that the network is always $3n-$Up. Consider any link (u, v) and time t, and let t_r denote the last recover of (u, v) in u before t. Then at most $4n + 6$ flood packets sent by u between t_r and t are not received by v until t.*

Proof: The proof is based on Lemma 27, which shows that at most $2n+3$ sync packet are in transit from u to v at t, and on the following two relations, proven below, between the numbers of sync packets and flood packets in transit:

1. Let i_{sync} (i_{flood}) be the sequence number of the *last* sync (respectively, flood) packet sent by u to v before t. Then
$$i_{flood} \leq (i_{sync}+1)+n.$$

2. Let i'_{sync} (i'_{flood}) be the sequence number of the *first* sync (respectively, flood) packet sent by u to v between t_r and t, which was not received by v until t. Then
$$i'_{flood} \geq (i'_{sync}-1)-n.$$

We first show that $i_{flood} \leq (i_{sync}+1)+n$. Intuitively, this means that u does not send flood packets with numbers 'much higher' than sync packets. Processor u sends flood packet number i_{flood} before delivering $i_{sync}+1$ messages. From the Synchronization Lemma, at most $(i_{sync}+1)+n$ messages are accepted before the time when u sent sync packet $i_{sync}+1$ to v, i.e. $A_s \leq (i_{sync}+1)+n$. Since the message in the flood packet i_{flood} is accepted before the time when u sends sync packet $i_{sync}+1$, holds $i_{flood} \leq (i_{sync}+1)+n$.

We now prove the second relation, $i'_{flood} \geq (i'_{sync}-1)-n$. Processor u sends flood packet i'_{flood} *after* sync packet $i'_{sync}-1$. Namely, when u sends flood packet i'_{flood}, then $D_u \geq i'_{sync}-1$. Since $R_u \geq D_u$ from Lemma 4, follows that at that time $R_u \geq i'_{sync}-1$. Since R_u is non-decreasing, holds $R_u \geq i'_{sync}-1$ also when flood packet number i'_{flood} is sent. Hence if the flood packet was sent in <I5> then $i'_{flood} = R_u \geq i'_{sync}-1$. Also, if the flood packet was sent in <G2>, then $i'_{flood} \geq R_u - n \geq i'_{sync}-1-n$.

We conclude that $(i'_{sync}-1)-n \leq i'_{flood} \leq i_{flood} \leq (i_{sync}+1)+n$. But Lemma 27 shows that at most $2n+3$ sync packets sent by u between t_r and t are not received by v until t. Namely, $i_{sync} - i'_{sync} \leq 2n+3$. Hence, $i_{flood} - i'_{flood} \leq 4n+5$. Namely, at most $4n+6$ flood packets sent by u between t_r and t are not received by v until t. $\square$

We now use Lemmas 27 and 28 to show that the number of packets in transit over a link is $O(n)$.

Lemma 29 *Assume that the network is always $3n-$Up. Consider any link (u,v) and time t, and let t_r denote the time of the last recovery of (u,v) at u*

before t. Then only $O(n)$ packets sent by u between t_r and t are not received by v until t.

Proof: The claim holds for sync packets from Lemma 27, and for flood packets from Lemma 28. It remains to consider update and recover packets.

Recover packets are sent only in <E2>, i.e. once after each recovery. From the FIFO property, at most one recover packet is received from a link while a link is up. In particular, at most one recover packet sent by u between t_r and t is not received by v until t.

Similarly, update packets are sent only in <F2>, upon receiving a recover packet. Since at most one recover packet is received from a link while the link is up, then at most one update packet is sent over a link while it is up. The claim follows as for recover packets. $\square$

4.6 Communication Complexity

We now show that Theorem 1 holds for the protocol shown in Figs. 8-11. The correctness follows from Lemma 16. The delay follows directly from the Progress property of Lemma 23. The congestion follows from Lemma 29. It remains to prove the communication complexity.

Lemma 30 *Consider an execution where the network is $3n-$ Up at all times. Then the communication complexity over intervals of length $3n+3$ is $C_A = 4m$ per accept, $C_F = 0$ per fail and $C_R = 2$ per recovery.*

Proof: Consider times t_1, t_2 and t_3 according to Def. 9. Namely:

$$t_1 \le t_2 \le t_3 \quad ; \quad t_2 - t_1 \ge 3n+3 \quad ; \quad t_3 - t_2 \ge 3n+3 \quad ;$$

We wish to show that the number of receive events during $(t_2, t_3]$ is at most

$$A([t_1, t_3)) \cdot 4m \ + \ R([t_1, t_3)) \cdot 2$$

Where $A([t_1, t_3))$ ($R([t_1, t_3))$) is the number of accept (respectively, recover) events during $[t_1, t_3)$.

From Defs. 9 and 10 follows that if the number of receive events during $[t_2, t_3)$ is bounded by the expression above, then the message amortized complexity is $4m$ and the fail amortized complexity is 2, both for intervals of length $3n$.

We first show that at most one recover packet is received during $[t_2, t_3)$ per each recover event during $[t_1, t_3)$. Since the transmission delay is at most one time unit, any packet received during $[t_2, t_3)$ was sent during $[t_2 - 1, t_3)$. Since recover packets are sent only upon recovery, it follows that at most one recover packet is received during $[t_2, t_3)$ per each recover event during $[t_2 - 1, t_3)$.

We now show that at most one update packet is received during $[t_2, t_3)$ per each recover event during $[t_1, t_3)$. Update packets are sent only upon receiving a recover packet <F2>. By the same argument as above, at most one recover packet is received during $[t_2 - 1, t_3)$ per each recover event during $[t_2 - 2, t_3)$. Hence, at most one update packet is received during $[t_2, t_3)$ per each recover event during $[t_2 - 2, t_3)$.

We now show that at most $2m$ sync packets and $2m$ flood packets are received during $[t_2, t_3)$ per each accept event during $[t_1, t_3)$. From the protocol, processors send each sync and flood packet at most once to each neighbor. Hence, it suffices to show that every sync and flood packet received during $(t_2, t_3]$ contains a message accepted during $[t_1, t_3]$. Since the transmission delay is at most one time unit, it suffices to show that every sync and flood packet *sent* during $[t_2 - 1, t_3)$ contains a message accepted during $[t_1, t_3)$. From the progress condition and since $t_1 < t_2 - 1 - 3n$ follows that the claim holds for packets sent in <H5> or <I5>.

It remains to prove that every flood packet sent in <G2> during $[t_2 - 1, t_3)$ contains a message accepted during $[t_1, t_3]$. Update packets received during $[t_2 - 1, t_3)$ were sent during $[t_2 - 2, t_3)$. Therefore, the flood packets sent in <G2> after $t_2 - 1$ have indices at least $\min_{v \in V}\{D_v(t_2 - 2) + 1\}$. From the progress property, and since $t_1 < (t_2 - 2) - 3n$, follows that $A_s(t_1) \leq \min_{v \in V}\{D_v(t_2 - 2)\}$. The claim follows. $\square$

5 The Enhanced Broadcast Protocol

In this section we present three minor enhancements to the protocol, and prove the properties of each enhanced version by reduction arguments to the previous version. In the first subsection we show how to use only short packets, by using modular counters instead of unbounded counters. In the second subsection we bound the space, by observing that it suffices to store only the last n messages received. In the third subsection we improve the throughput, by using a 'window' of messages in the source.

5.1 Using Bounded Counters.

We now show that modular counters can be used in the protocol, instead of the unbounded counters. Modular counters suffices, intuitively, since the maximal difference between counter values compared in the protocol is always bounded. From the synchronization condition, until any given moment all processors deliver roughly the same number of messages. On the other hand, there is a difference of at most $O(n)$ between the counters compared by the protocol and the number of messages delivered by this processor or by one of its neighbors. We formalize this intuition in the following Lemmas.

There are two kinds of comparison operations in the protocol: comparison of two numeric variables (counters) and comparison of a counter to a value in an incoming packet. We begin by showing that the whenever the protocol compares two counters, the difference between them is bounded by $O(n)$. Later we bound the difference between counters and the values they are compared to from incoming packets.

From Lemmas 4 and 25 it follows immediately that if the network is $3n-$Up at any time, then:

$$R_v - n \leq D_v \leq R_v.$$

There are only two places in the protocol where variables other than R_v and D_v are compared, in <H2> and in <H7>. In <H7>, the values of A_s and D_s are compared. However, $D_s \leq A_s \leq D_s + 1$. It remains to consider <H2>, which compares $D'_v[u]$ to D_v.

Lemma 31 *Assume that the network is $3n-$Up at any time. Then either $D_v(t) - n - 1 \leq D'_v[u](t) \leq D_v(t) + n + 1$ or $D'_v[u](t) = \mathsf{undef}$.*

Proof: From Eq. (5) of Lemma 25 holds either $D'_v[u] = \mathsf{undef}$ or $D'_v[u](t) \le D_v(t) + n$. It remains to show that either $D_v(t) - n - 1 \le D'_v[u](t)$ or $D'_v[u](t) = \mathsf{undef}$. Assume that $D'_v[u](t) \ne \mathsf{undef}$.

Since $D'_v[u](t) \ne \mathsf{undef}$ follows that at some time $t_{up} \le t$ processor v receives an update packet from u, and (u, v) is up at v during $[t_{up}, t]$. Let t'_{up} denote the time when processor u sent this update packet (see Fig. 6).

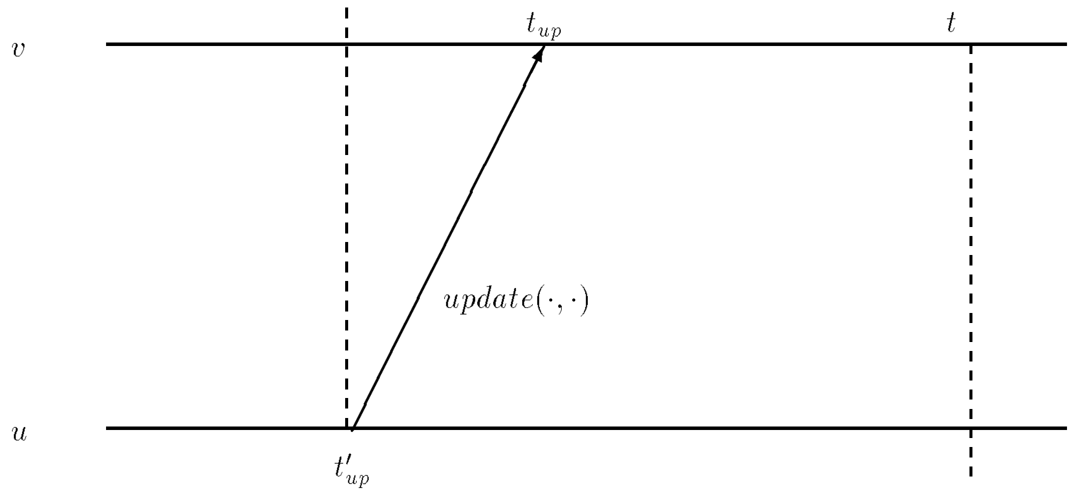

Figure 6: Proof that either $D_v - n - 1 \le D'_v[u]$ or $D'_v[u] = \mathsf{undef}$.

The idea of the proof is that if v delivers two or more messages between t'_{up} and t then $D_v(t) \le D'_v[u](t) + 1$ and the claim holds. Otherwise, the claim holds since $D'_v[u](t_{up}+) = D_u(t'_{up})$ and $D_v(t'_{up}) \le D_u(t'_{up}) + n$.

From Lemma 9, link (u, v) is up at v during $[t'_{up}, t_{up}]$. Hence, link (u, v) is up at v during $[t'_{up}, t]$. From Lemma 18, if v delivers two or more messages during $[t'_{up}, t]$ then $D_v(t) - 1 \le D'_v[u](t)$ and the claim follows.

Assume, therefore, that v delivers at most one message during $[t'_{up}, t]$. Namely:

$$D_v(t) \le D_v(t'_{up}) + 1 \tag{6}$$

From Eq. (4) of Lemma 25 holds

$$D_v(t'_{up}) \le D_u(t'_{up}) + n \tag{7}$$

From Lemma 10 holds:

$$D'_v[u](t_{up}) = D_u(t'_{up}) \tag{8}$$

Since $D'_v[u]$ is non-decreasing while (u, v) is up at v (Lemma 11) follows that:

$$D'_v[u](t_{up}) \leq D'_v[u](t) \tag{9}$$

The claim follows from Eqs. (6-9). □

We now show that the difference between a counter and a value from an incoming packet to which it is compared is at most $O(n)$. We first note that a processor never receives at most $O(n)$ messages more than it had delivered to the higher layer.

Lemma 32 *Assume that the network is always $3n-$Up and that processor v receives a packet containing number i at time t. Then $i \leq D_v(t) + n$.*

Proof: If v receives a packet numbered i, then one of its neighbors, say u, has previously sent this packet. From Lemma 17, holds $R_u \leq A_s$. Since A_s is non-decreasing, then $i \leq A_s(t)$. The claim follows since the Synchronization Lemma shows that $D_v(t) > A_s(t) - n$. □

We now show that the dual claim also holds: whenever a processor receives a packet numbered i, then i is not much smaller than the number of deliveries in this processor.

Lemma 33 *Assume that the network is always $3n-$Up, and that processor v receives a packet containing number i at time t. Then $D_v(t) \leq i + 2n$.*

Proof: Let u be the neighbor from which v receives the packet at time t, and let t' denote the time when u has sent the packet to v. From Lemma 9 follows that link (u, v) is up at v during $[t', t]$.

If processor v delivers more than one message during $[t', t]$, then from Lemma 22 holds $D_v(t) - 1 \leq D'_v[u](t)$. From Lemma 10, processor v receives from u before t either sync packet number $D_v(t) - 1$ or update packet with deliver-counter at least $D_v(t) - 1$. From the FIFO property of the link, u sends this sync or update packet before t'. Namely, at that time, before t', the value of D_u is $D_v(t) - 1$. Since D_u is non-decreasing (Lemma 3), we know that in this case $D_u(t') \geq D_v(t) - 1$.

In the other case, when v delivers at most one message during $[t', t]$, then immediately from Lemma 2 A) follows $D_v(t) \leq D_v(t') + 1$. We have therefore bounded $D_v(t)$ in both cases:

$$D_v(t) \leq \max\{D_v(t') + 1, D_u(t') + 1\}. \tag{10}$$

From Eq. (4) of Lemma 25, holds $D_v(t') \leq D_u(t') + n$. Therefore, from Eq. (10) follows that $D_v(t) \leq D_u(t') + n + 1$. The claim follows immediately if the packet is a sync, update or recover packet. The claim holds also for flood packets, since the number of a flood packet sent by u is either R_u in <I5> or in the range $R_u - n + 1$ to R_u in <G2>, and by Lemma 4 holds $D_u \leq R_u$. $\square$

We now conclude that the difference between numbers compared during any execution of the protocol is bounded by $O(n)$.

Lemma 34 *The difference between two numbers compared by the protocol in a timed dynamic execution where the network is always* $3n-$ *Up is bounded by* $3n + 1$.

Proof: The protocol compares values in statements <G2>, <H7>, <H2> and <I2>. We now consider each of these statements. The claim is trivial for <H7>, since $D_s \leq A_s \leq D_s + 1$.

In <G2> the received value of c_r is compared to R_v. From Lemma 32 holds $c_r \leq D_v + n$, and from Lemma 4 holds $D_v \leq R_v$. Hence, $c_r \leq R_v + n$. On the other hand, from Lemma 33 holds $D_v \leq c_r + 2n$, and from Lemma 25 holds $R_v \leq D_v + n$. Hence $R_v \leq c_r + 3n$. Namely, the claim holds regarding <G2>. A similar argument holds for <I2>, where the received value of i is compared to $R_v + 1$.

In <H2>, the value of D_v is compared to $D'_v[u]$. The claim holds in this case from Lemma 31. Also, in <H2>, the value of D_v is compared with R_v. In this case the claim holds from Lemmas 4 and 25. $\square$

From Lemma 34 it follows that it suffices to use numeric variables reduced modulo $6n+3$ (or more), when the comparisons are performed in the standard manner when using a cyclic counter [Per83]. Namely, x is larger than y if $|x - y| \leq 3n + 1$. By performing this modification, the protocol uses only short messages. The properties of this version of the protocol are summarized in the following Theorem.

Theorem 35 *Consider the protocol in Figs. 8-11, when all arithmetics and comparisons are performed modulo* $6n + 3$. *This protocol uses only short packets. Furthermore, if the network is 3n-Up at all times, then this protocol is correct, with delay at most* $3n$, *congestion* $O(n)$ *and communication complexity* $C_A = 4m$ *per accept,* $C_F = 0$ *per fail and* $C_R = 2$ *per recovery over intervals of length* $3n + 3$.

Proof: It is obvious that the modified protocol uses only short packets; it remains to show that correctness and efficiency are not lost. Consider any timed dynamic execution $(\alpha, time)$ of the modified protocol.

From Theorem 1, it suffices to prove that there is a corresponding execution $(\alpha', time')$ of the original (non-modular) protocol, s.t. the two executions are identical except for the use of modular counters. Namely, let $\alpha = e_1, e_2, \ldots e_n$ and $\alpha' = e'_1, e'_2, \ldots e'_n$. We show that for every i, the event e_i is identical to e'_i except that all values are reduced modulo $6n + 3$, and that $time(e_i) = time'(e'_i)$.

Suppose to the contrary that there is no such α'. Consider the shortest prefix $e_1, \ldots, e_k$ of α which does not have a corresponding prefix of an execution of the original protocol. Let $e'_1, \ldots, e'_{k-1}$ be the prefix of an execution of the original protocol corresponding to $e_1, \ldots, e_{k-1}$. By considering all possible event types for e_k, it follows from Lemma 34 that it is possible to select e'_k which corresponds to e_k and such that $e'_1, \ldots, e'_k$ is a prefix of an execution of the original protocol. $\square$

5.2 Bounded Storage

The previous subsection shows a minor fix to the protocol, which maintains all of the properties of Theorem 1 but uses only short packets. We now observe that from Eq. 3 of Lemma 25 follows that the protocol always uses only the last n messages received. This suggests an additional minor fix: store only the last n messages received. It easily follows that the combination of these two simple fixes gives $O(n)$ space complexity.

Theorem 36 *Consider the protocol in Figs. 8-11, when all arithmetics and comparisons are performed module $6n + 3$ and where processors store only the last n messages received. This protocol has $O(n)$ space complexity and uses only short packets. Furthermore, if the network is 3n-Up at all times, then this protocol is correct, with delay at most $3n$, congestion $O(n)$ and communication complexity $C_A = 4m$ per accept, $C_F = 0$ per fail and $C_R = 2$ per recovery over intervals of length $3n + 3$.*

Proof Sketch: The proof follows like that of Theorem 35, by observing that the protocol uses the stored messages in two ways. First, the protocol sends, upon receiving an update packet, at most n flood packets, each containing a message not received yet by the neighbor <G2>. Obviously, for this purpose it is sufficient to store only the last n received messages. The second use of messages in the protocol is to deliver them to the higher layer. The protocol in processor v delivers messages whose sequence number is higher than D_v. Eq. 3 of Lemma 25 shows that storing the n last received messages suffices for this purpose as well. $\square$

5.3 Improving Throughput Using a Window

In the protocol as presented so far, the throughput is bounded by one over the delay, namely $\Omega(\frac{1}{n})$. It is possible to show that the throughput is not better than $O(\frac{1}{\sqrt{n}})$. We now show how the protocol may be modified to achieve $\Omega(1)$ throughput.

As described so far, the source performs a ready event, i.e. enable the higher layer to broadcast another message, only after receiving from *all* neighbors the sync packet containing the last message accepted.

We change the operation of the source processor by implementing a *window* of $n+1$ messages. Namely, the source processor enables a ready event whenever $A_s \leq D_v + n$. This condition replaces the existing condition in <H7>, which is: $A_s \leq D_s$.

This is a typical implementation of the window technique, which is often used to improve throughput while avoiding excessive congestion. The window does involve some overheads: it requires the source to store n messages, and, as we shortly show, the rest of the network operates as if there are additional n processors. In our case, all complexities are linear in the number of processors, namely all complexities are at most doubled by the use of a window of size n.

To distinguish between the two versions of the protocol, we refer in this section to the protocol as described before this section as the 'non-window version' and to the protocol with the modification described above as the 'window version'. The window version preserves all the properties we proved for the non-window version, up to a constant. This is in addition to the substantial improvement in the throughput.

Theorem 37 *Consider the protocol in Figs. 8-11 with the modification described above in this section (window of n messages and arithmetics modulo* $12n+3$*). This protocol has* $O(n)$ *space complexity and uses only short packets. Furthermore, if the network is* $6n-$*Up at all times, then this protocol is correct, with throughput at least* $\frac{1}{6}$ *for intervals of length* $6n+1$*, delay at most* $6n$*, congestion* $O(n)$ *and communication complexity* $C_A = 4(m+n)$ *per accept,* $C_F = 0$ *per fail and* $C_R = 2$ *per recovery over intervals of length* $6n+3$*.*

Proof: We first prove that the protocol still has all of the properties of the previous versions. We map every timed dynamic execution $(\alpha, time)$ of the window protocol into a timed dynamic execution $(\alpha', time')$ of the non-window version. The execution $(\alpha, time)$ is in an arbitrary network (V, E) containing n processors and m links; execution $(\alpha', time')$ is in a fictitious network (V', E') containing $2n$ processors and $n + m$ links. The mapping is shown in Fig. 7. Namely, if we assume that processor names $s_1, \ldots, s_n$ are not used in V:

- $V' = V \cup \{s_1, \ldots, s_n\}$
- $E' = E \cup \{(s, s_1)\} \cup \{(s_i, s_i + 1) | i = 1, \ldots, n\}$

The links added in the non-window execution operate without failures and with zero delay. The source processor of $(\alpha, time)$ is, as usual, processor s. The source processor of $(\alpha', time')$ is s_n.

The sequence of events and their timing in $(\alpha', time')$ is the same as in $(\alpha, time)$. In particular, if at time t the network in $(\alpha, time)$ is $6n-$Up, then the network in $(\alpha', time')$ is also $6n-$Up at t. As $|V'| = 2|V| = 2n$, the properties stated in Theorem 37 follow from the corresponding properties proved for the non-window version in Theorem 36.

We now prove the assertion concerning the throughput. This part of the proof does not use the mapping to α'. Recall that the throughput is defined over executions where there is never a delay from the time of a *ready* event and until the following *accept* event. Namely, the higher layer always has additional messages for broadcast.

Since *accept* events immediately follow each *ready* events, and the capacity in the source is limited, it follows from the modified <H7> that at any time t holds $A_s(t) > D_s(t) + n$.

We now prove that at least $n+1$ messages are accepted in every interval of length $6n+1$; hence the throughput is at least $\frac{1}{6}$. Consider any interval $[t, t+6n+1]$. From the first part of the proof, the delay is at most $6n$. Therefore, after $(t+6n)$ all processors have delivered every message accepted until time t. Namely, for every processor v holds $D_v((t+6n)+) \geq A_s(t)$. In particular, $D_s((t+6n)+) \geq A_s(t)$. By substituting $A_s(t+6n+) > D_s(t+6n+) + n$, we get $A_s(t+6n+) > A_s(t) + n$. The proof follows since A_s is non-decreasing. $\square$

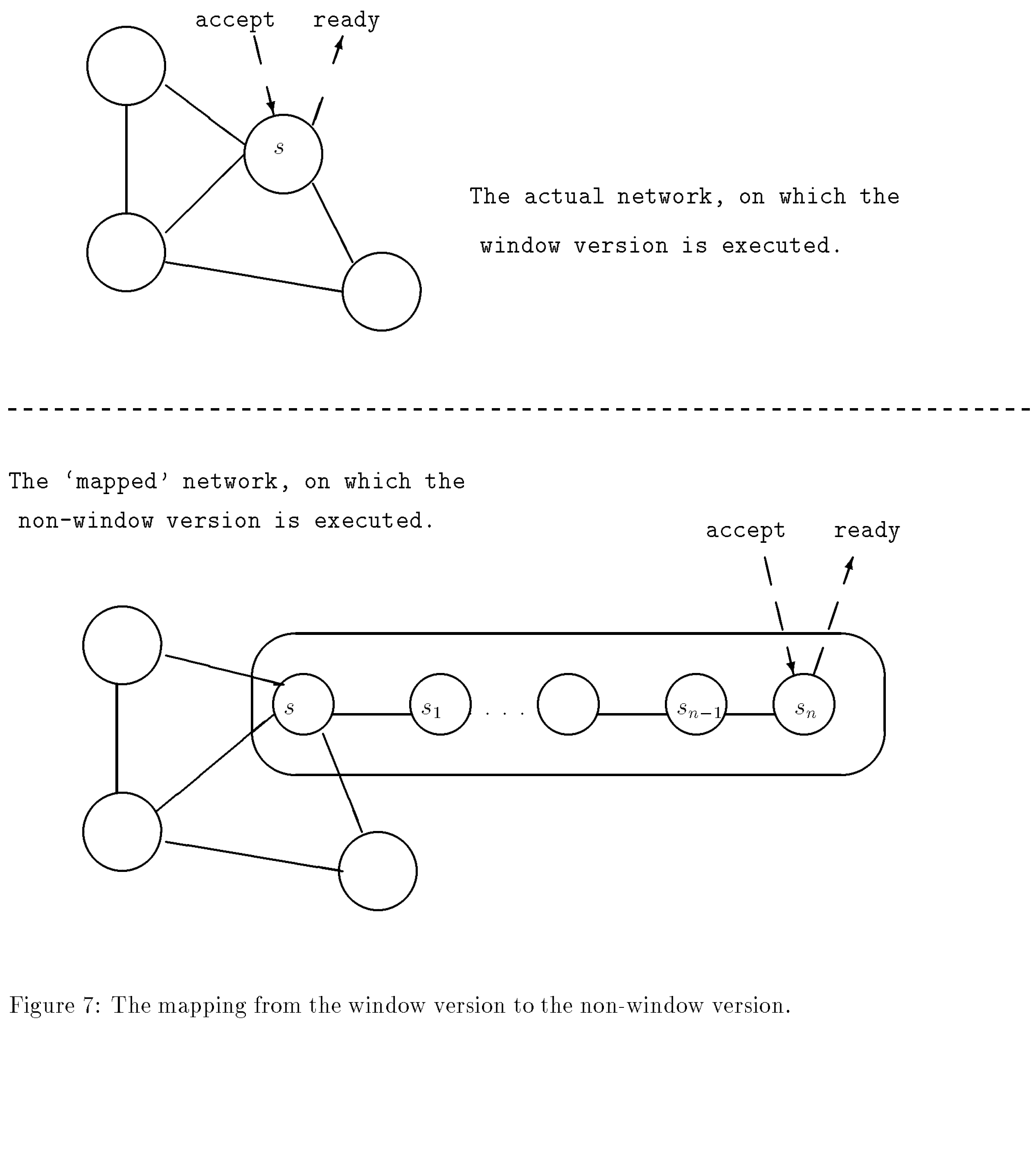
accept
ready
s
The actual network, on which the window version is executed.
The 'mapped' network, on which the non-window version is executed.
accept
ready
s
s_1
. . .
s_{n-1}
s_n

Figure 7: The mapping from the window version to the non-window version.

6 Concluding Remarks

We have presented a quantitative approach to dynamic networks, and illustrated this approach by analyzing a new, efficient broadcast protocol. Our objective was to enable realistic evaluation of protocols. Namely, the practical value of protocols and ideas should be reflected by the formal evaluation within a theoretical framework.

We have also suggested new complexity measures, especially for the throughput and the communication complexities. The main idea is that the complexities are amortized but only over intervals which satisfy certain minimal requirements, e.g. intervals which are not 'too short'. These measures might be useful for analysis of other interactive tasks as well, and even without failures. In particular, the advantage of the window mechanism is revealed by our definition of throughput.

Further research is required in several directions. It is interesting to find if we can reduce the reliability requirements of broadcast. In particular, the assumption that the network is reliable is too strong for some networks, which may be disconnected for substantial amounts of time. Is it possible to solve [5] such problems, using bounded resources, and allowing cuts in the network, or processor crashes? Also, it is interesting to try to combine the approach of this work with a more realistic approach to time complexity which permits 'timeouts' such as suggested in [Her88, HK89].

The broadcast task addressed in this work is closely related to practical broadcast and multicast problems of audio and video data streams, such as video-conferencing and radio/video broadcasts over computer networks. It appears that the protocol described here deals with aspects which so far found little reflection in the protocols proposed and used in experiments for such tasks. Namely, the protocol deals with allowing the broadcast to flow over a *dynamic* graph which may contain *cycles*. This could be very helpful in some realistic scenarios, where specific links or gateways may be temporary congested; a protocol which allows redundant connectivity (cycles) may be able to route around the congested areas.

Another challenge is to apply the quantitative approach to additional tasks. In [Her91] we have briefly discussed how the protocol presented in this paper may be adapted to end to end communication and to implement a fault-tolerant synchronizer. In [Her92] we used the quantitative approach

[5]The construction of [AGH90a] does not require that the entire network would be reliable, and allowed crashes. However, a full proof did not appear.

to analyze and improve connection-based communication schemes.

Acknowledgments

We thank Baruch Awerbuch for collaborating with us in earlier stages of this research [AGH90a]. Special thanks to Hagit Attiya, who made a substantial contribution to the present definitions of reliability, and for other encouraging and helpful remarks. We thank Shimon Even for discussing shortcomings of the eventual stability approach, and thereby planting the seeds of the quantitative approach. We also thank Benny Chor, Israel Cidon, Shlomi Dolev, Guy Even, Shlomo Moran, David Peleg, Benny Pinkas, and Sergio Rajsbaum for their comments.

References

[AAG87] Yehuda Afek, Baruch Awerbuch, and Eli Gafni. Applying static network protocols to dynamic networks. In 28th *Annual Symposium on Foundations of Computer Science*. IEEE, October 1987.

[ACK90] Baruch Awerbuch, Israel Cidon, and Shay Kutten. Optimal maintenance of replicated information. In *Proc.* 31st *Symp. on Foundations of Computer Science*, October 1990.

[ADLS90] Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. In STOC 1991, 1990.

[AE86] Baruch Awerbuch and Shimon Even. Reliable broadcast protocols in unreliable networks. *Networks*, 16(4):381–396, Winter 1986.

[AG91] Yehuda Afek and Eli Gafni. Bootstrap network resynchronization. In *Proceedings of the* 10th *Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada*, pages 295–307. ACM SIGACT and SIGOPS, ACM, August 1991.

[AGH90a] Baruch Awerbuch, Oded Goldreich, and Amir Herzberg. A quantitative approach to dynamic networks. In *Proceedings of the* 9th *Annual ACM Symposium on Principles of Distributed Computing*, pages 189–204, August 1990.

[AGH90b] Baruch Awerbuch, Oded Goldreich, and Amir Herzberg. A quantitative approach to dynamic networks (version without global time). Technical Report 624, Computer Science Dept., Technion, May 1990.

[AMS89] Baruch Awerbuch, Yishay Mansour, and Nir Shavit. Polynomial end-to-end communication. In *Proc. of the 30th IEEE Symp. on Foundations of Computer Science*, pages 358–363, October 1989.

[AS88] Baruch Awerbuch and Michael Sipser. Dynamic networks are as fast as static networks. In 29^{th} *Annual Symposium on Foundations of Computer Science*, pages 206–220. IEEE, October 1988.

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

[Awe88] Baruch Awerbuch. On the effect of feedback in dynamic network protocols. In *Proc. 29th IEEE Symp. on Foundation of Computer Science*, pages 231–245, October 1988.

[BGG$^+$85] A. E. Baratz, J. P. Gray, P. E. Green Jr., J. M. Jaffe, and D. P. Pozefsky. SNA networks of small systems. *IEEE Journal on Selected Areas in Comm.*, SAC-3(3):416–426, May 1985.

[BGS88] Alan E. Baratz, Inder Gopal, and Adrian Segall. Fault tolerant queries in computer networks. In J. van Leeuwen, editor, *Distributed Algorithms: Second International Workshop*. Springer-Verlag, 1988. Lecure notes in computer science number 312.

[BS88] Alan E. Baratz and Adrian Segall. Reliable link initialization procedures. *IEEE Trans. on Communication*, COM-36:144–152, February 1988.

[Cha82] Ernest J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *IEEE Trans. Software Eng.*, 8(4):391–401, July 1982.

[CR87] Israel Cidon and Raphael Rom. Failsafe end-to-end protocols in computer networks with changing topology. *IEEE Trans. Comm.*, COM-35(4):410–413, April 1987.

[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 a failsafe network protocol. *IEEE Tran. Comm.*, COM-27(6):840–846, June 1979.

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

[GHS83] Robert G. Gallager, Pierre A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Prog. Lang. and Syst.*, 5(1):66–77, January 1983.

[GS92] George Grover and Adrian Segall. A full duplex dlc protocol on two links. *IEEE Transactions on Communications*, COM-40(1):210–223, January 1992.

[Her88] Amir Herzberg. Network management in the presence of faults. In *Ninth International Conference on Computers and Communication (ICCC)*, October 1988. Updated version: 'Early Termination in Unreliable Communication Networks' is technical report TR-650 of computer science dept., Technion, September 1990.

[Her91] Amir Herzberg. *Communication Networks in the Presence of Faults*. PhD thesis, Computer Science Faculty, Technion, Israel, 1991. In Hebrew.

[Her92] Amir Herzberg. Connection-based communication in dynamic networks. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 13–24, August 1992.

[HK89] Amir Herzberg and Shay Kutten. Efficient detection of message forwarding faults. In *Proceedings of the* 8th *Annual ACM Symposium on Principles of Distributed Computing*, pages 339–353, August 1989.

[JBS86] Jeff Jaffe, Alan E. Baratz, and Adrian Segall. Subtle design issues in the implementation of distributed dynamic routing algorithms. *Computer networks and ISDN systems*, 12(3):147–158, 1986.

[LR90] K. Lougheed and Yacov Rekhter. A border gateway protocol. Internet RFC 1163, Network Working Group, June 1990.

[MRR80] John M. McQuillan, Ira Richer, and Eric C. Rosen. The new routing algorithm for the ARPANET. *IEEE Trans. Comm.*, 28(5):711–719, May 1980.

[MS79] P. Merlin and A. Segall. Failsafe distributed routing protocol. *IEEE Trans. Comm.*, COM-27:1280–1288, September 1979.

[Per83] Radia Perlman. Fault tolerant broadcast of routing information. *Computer Networks*, December 1983.

[RS91] T. L. Rodeheffer and M. D. Schroeder. Automatic reconfiguration in autonet. In *Symp. on Principles of Operating Systems*, 1991.

[Seg83] A. Segall. Distributed network protocols. *IEEE Trans. on Information Theory*, IT-29(1), January 1983.

[SS81] A. Segall and M. Sidi. A failsafe distributed protocol for minimum delay routing. *IEEE Trans. Comm.*, COM-29(5):689–695, May 1981.

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

A Code of the simplified protocol

A high-level concise description of the algorithm is given in Figure 1. A more complete and formal description is given in Figures 8-11. The formal description (code) contains labels, of the form 3.2, which refers to a particular line (number) within a particular block of the code (letter). These labels would be used for detailed proofs.

- Sync-packet $sync(m, i)$, where m is a message and i is its index. This packet is used to synchronize neighboring processors.
- Update-packet $update(c_d, c_r)$, where c_d is the *deliver-counter* and c_r is the *receive-counter*. Namely, c_d is the highest index of messages delivered by the processor to the higher layer and c_r is the number of the last message received by the processor. This packet is used to re-synchronize between two processors upon recovery of the link connecting them.
- Recover-packet *recover*. This packet is used to signal to the neighbor that the link has recovered at both ends.
- Flood-packet $flood(m, i)$, where m is a message and i is its index. This packet is used to distribute the message to the processors in the network.

Figure 8: Types of packets used in the protocol.

$M_v(i)$: Buffer for the i^{th} message, initially empty. It suffices to store only the last n messages.

R_v: The highest index of messages placed in M_v. Initially 0.

D_v: Counts the number of messages delivered by v to the higher layer (which equals the number of sync packets sent by v). Initially 0.

$D'_v(u)$: A lower bound estimate for D_u, i.e. for the number of messages delivered by neighbor u. Initially, and after (u, v) fails at v, has the value `undef`.

A_s: The number of accept events in the source s. Initially 0.

G_v: The set of neighbors u of v such that (u, v) is up at v.

G_v^D: The set of neighbors u of v such that (u, v) is up at v since v has last delivered a message.

G_v^R: The set of neighbors u of v such that (u, v) is up at v and v has received a recover packet from u since the last recovery of (u, v) at v.

Figure 9: Variables at processor v.

```
<A1> (for $v = s$) When accepting a message $m$ from the higher layer:        S
<A2> {    increment $A_s$;                                                     S
<A3>      call procedure $message(m, A_s)$;                                    S
<A4>      call procedure $proceed()$;                                          S
     }
<B1> When receiving $sync(m, i)$ from $u$:                                     S
<B2> {    $D'_v[u] \leftarrow i$;                                              S
<B3>      call procedure $message(m, i)$;                                      S
<B4>      call procedure $proceed()$;                                          S
     }
<C1> When receiving $flood(m, i)$:                                             F
<C2> {    call procedure $message(m, i)$;                                      F
<C3>      call procedure $proceed()$;                                          F
     }
<D1> When the link to $u$ fails:
<D2> {    remove $u$ from $G_v$, $G_v^D$ and $G_v^R$;
<D3>      $D'_v[u] \leftarrow$ undef;
<D4>      call procedure $proceed()$;
     }
<E1> When the link to $u$ recovers:
<E2> {    send $recover$ to $u$;
<E3>      add $u$ to $G_v$;
     }
<F1> When receiving $recover$ from $u$:
<F2> {    send $update(D_v, R_v)$ to $u$;
<F3>      add $u$ to $G_v^R$;
     }
<G1> When receiving $update(c_d, c_r)$ from $u$:
<G2> {    if $c_r < R_v$ then for $r = \max\{c_r + 1, R_v - n + 1\}$ to $R_v$ do:
                  send $flood(M_v[r], r)$ to $u$;                              F
<G3>      $D'_v[u] \leftarrow c_d$;
<G4>      call procedure $proceed()$;
     }
```

Figure 10: Algorithm at processor v.

```
<H1>  procedure $proceed()$comment: tries to cause $ready$ (for $v = s$) or $deliver$ (for $v \neq s$).
<H2>  {    while $(D_v < R_v) \wedge (\forall u \in G_v^D)\,((D'_v[u] \neq \mathsf{undef}) \wedge (D_v \leq D'_v[u]))$    S
<H3>            do   {    increment $D_v$;                                                S
<H4>                      deliver $M_v[D_v]$ to the higher layer;                         S
<H5>                      send $sync(M_v[D_v], D_v)$ to all $u \in G_v^R$;                S
<H6>                      $G_v^D \leftarrow G_v$;
                     }
<H7>       if $(v = s) \wedge (A_s \leq D_s)$                                             S
<H8>            then signal $ready$ to the higher layer;                                  S
      }
<I1>  procedure $message(m, i)$     comment: deals with message $m$ numbered $i$.
<I2>  {    if $i = R_v + 1$                                                               S
<I3>            then {    increment $R_v$;                                                S
<I4>                      $M_v[R_v] \leftarrow m$;                                        S
<I5>                      send $flood(m, R_v)$ to all $u \in G_v^R$;                      SF
                     }
      }
```

Figure 11: Procedures at processor v.