LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-391

POLYNOMIAL END-TO-END COMMUNICATION

Baruch Awerbuch
Yishay Mansour
Nir Shavit

August 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TM-391	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-85-K-0168 and N00014-83-K-0125
6a. NAME OF PERFORMING ORGANIZATION MIT Laboratory for Computer Science 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Department of Navy
6c. ADDRESS (City, State, and ZIP Code) 545 Technology Square Cambridge, MA 02139	7b. ADDRESS (City, State, and ZIP Code) Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code) 1400 Wilson Boulevard Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Polynomial End-To-End Communication

12. PERSONAL AUTHOR(S)
Awerbuch, B., Mansour, Y., and Shavit, N.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1989 August	23

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Communication networks, unbounded counters, fault-tolerance, ~~end-to-end~~

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

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

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

The best known previous solutions to this problem had exponential complexity. Morever, it has been conjectured in (AG88) that a polynomial solution is impossible.

This paper disproves this conjecture, presenting the first polynomial end-to-end protocol. The protocol uses techniqes adopted from shared memory algorithms, and introduces novel techniques for fast load balancing in communication networks.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Judy Little, Publications Coordinator	22b. TELEPHONE (Include Area Code) (617) 253-5894 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆U.S. Government Printing Office: 1985—607-047

Polynomial End-To-End Communication[1]

Baruch Awerbuch [†] Yishay Mansour [‡] Nir Shavit [§]

August 29, 1989

Abstract

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

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

The best known previous solutions to this problem had exponential complexity. Moreover, it was conjectured in [AG88] that a polynomial solution is impossible.

This paper disproves this conjecture, presenting the first polynomial end-to-end protocol. The protocol uses methods adopted from shared memory algorithms, and introduces novel techniques for fast load balancing in communication networks.

[1] A preliminary version of the results presented in this paper appeared in [AMS89].

[†] Dept. of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139; ARPANET: baruch@theory.lcs.mit.edu Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, and a special grant from IBM.

[‡] Laboratory for Computer science, MIT, 545 Tech. sq., Cambridge, MA 02139. partially supported by NSF 865727-CCR, ARO DALL03-86-K-017 and ISEF fellowship.

[§] Hebrew University, Givat-Ram, Jerusalem 91904; Supported by Israeli Communications Ministry Award. Currently visiting the *TDS* group at *MIT*, supported by *NSF* contract no CCR-8611442, by *ONR* contract no N0014-85-K-0168, by *DARPA* contract no N00014-83-K-0125, and a special grant from *IBM*.

Keywords: Dynamic Networks, Load Balancing, Time Stamping, Amortized Analysis.

1 Introduction

A basic task in any network is that of *end-to-end* communication, that is, delivery in finite time, of data items generated at a designated sender processor, to a designated receiver processor, without duplication, omission or re-ordering of the data items. The data items could represent transactions of a stock exchange, speech or video signals, military commands, etc. In almost all cases, data items are generated on-line and are not available at the beginning of the protocol's execution.

In a reliable network, where communication links never fail, this task is performed easily by establishing a fixed communication path between the sender and the receiver, and sending all data items along this path. Unfortunately, existing communication networks, e.g. the ARPANET [MRR80], DECNET [Wec80], have a *dynamic* topology in the sense that links may repeatedly fail and recover, making it impossible to relay on the use of any single communication path.

The "classical" approach to handle the problem is to construct a new communication path every time the previous path fails, taking care to purge any messages in transit on the old path. However, this approach is very limited, since its implementations (as in [Fin79, Gal76, AAG87, AS88, AAM89, AGH89]) require strong assumptions regarding the allowable patterns of link failures in the network. In [AAG87, AS88, AAM89] for example, the assumption is that the *whole network* stabilizes for a period of time long enough to allow construction of a path and communication over it. The weakest assumption among those above, presented in the broadcast protocol of [AGH89], still requires that all the edges on some path between the sender and receiver be operational for the entire time period required to construct that path and communicate the data over it.

This assumption is overly optimistic, since for example, if every edge has a constant probability of being operational (or not operational) at a given time, then the probability of the whole path being operational at a given time is exponentially small in the length of the path.

However, one can see that the existence of an operational communication path is *not* a necessary condition for communicating between two nodes (processors). In fact, as stated in [AE86], the necessary condition is merely that there is "eventual connectivity" between the sender and receiver, in the sense that there is no permanent cut (see [Vis83]) of failed edges between them. More precisely, there exists no partition of the network into two sets, one containing the sender and the other the receiver, such that from some time and on, no operating edge connects a node in one set to a node the other set.

Early works [Vis83, AE83, AE86] solving the end-to-end problem under the minimal conditions alone, were based on the use "unbounded sequence numbers", implying that both the message size and amount of memory needed, grow with number of data items transmitted. In other words, the complexity of the protocols was *unbounded* in terms of the true input to the problem, namely, the size of the network.

The elegant and surprising work of Afek and [illegible] [AG88] presented the first "bounded complexity" end-to-end protocol. Unfortunately, this solution required *exponential* message complexity (because the number of times a given data item is sent over a network link is exponential). Moreover, it was believed by many researchers, and conjectured in [AG88], that no polynomial solution exists, leaving little hope for a reasonable solution to the end-to-end problem.

In this paper, we disprove this conjecture, presenting an end-to-end protocol that is *polynomial* in both messages and space. The protocol is based on a new technique for sequence numbering, that combines the sequential time-stamp schemes used in shared memory algorithms ([Lam86, IL87, DS89]), with a novel and highly fault tolerant load balancing method allowing to preserve global properties based on local information only (as in Goldberg-Tarjan [GT88] and Ahuja-Orlin [AO87]).

2 Problem Statement

2.1 The network model

Consider a communication network in the form of an undirected graph $G(V, E)$, where the nodes are the processors and the edges are the links of communication. Each *undirected* link consists of two *directed* links, delivering messages in the opposite directions. Below we describe the properties of a directed link.

Each link has a *finite* capacity, in the sense that only constant number of messages is allowed to be in transit on a given link at a given time (I.e. we consider only protocols that obey this property.) The communication over links obeys the FIFO rule, that the sequence of messages *received* over the link is a prefix of the sequence of messages sent over the link. Also, the communication is completely *asynchronous*, namely, there is no a priori bound on link delays.

A link is *non-viable* if starting from some message, and on, it will not deliver any further messages to the other end-point; for those messages the delay is considered to be infinite (∞). The sequence of messages received is in this case a *proper* prefix of the sequence of messages sent. Otherwise, the link is *viable*. An undirected link is *viable* if both of the directed links that it consists of are viable.

We say that the sender is *eventually connected* to the receiver if there exists a (simple) path from the sender to the receiver, consisting entirely of viable links. Clearly, if non-viable links create a *cut* of the network, disconnecting the sender from the receiver, then, eventually, the sender will not be capable of delivering messages to the receiver.

2.2 The end-to-end problem

The purpose of the end-to-end protocol is to establish a (directed) "virtual link" to be used for delivery of data items from the sender to the receiver. It is required that this virtual link be viable if and only if the sender is eventually connected to receiver. This virtual link should have the same properties as a "regular" network link, namely:

Safety: The sequence of data items output by the receiver is a prefix of the sequence of data items input by the sender.

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

2.3 The complexity measures

We consider the following complexity measures.

Communication: The number of bits transferred in the network, per data item delivered. That is, this the total number of bits sent in the period of time between two successive data item deliveries at the receiver.

Space: The maximal amount of space required by a node's program throughout the protocol.

Time: The maximal length of a time interval between two successive data item deliveries at the receiver, under the assumption that delays of viable links are upper-bounded by 1 time unit.

Computation time: The maximal number of local computation steps a node performs in the interval of time between two successive data item deliveries, provided that this data item is not the last data item.

Definition 2.1 A protocol is *bounded* if its communication, space, time, and computation time complexities are independent of the number of data items, depending only on the size of the network.

Definition 2.2 A protocol is *polynomial* if its communication, space, time and computation time complexities are upper-bounded by polynomials of the size of the network.

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

2.4 Relations to other models

The model described above is called the "∞-delay model" in [AG88], and the "fail-stop model" in [AM88]. As mentioned in the introduction, our motivation here is to deal with networks frequently changing topology. In such *dynamic* networks, links may fail and recover many times (yet processors never fail). Each failure or recovery of network link is eventually reported at both end-points by some underlying link protocol. As pointed out by Afek and Gafni [AG88], this dynamic model is easily reducible to the model described above. The simulation of the dynamic model by the fail-stop model is as follows. A message to be forwarded on a link is stored in a buffer, until the link recovers and all the previously sent messages have been delivered. A protocol similar to the data-link initialization protocol of Baratz and Segall [BS88] is used to guarantee that no messages are lost or duplicated. Each link in the dynamic network that does (does not) fail forever is represented by a viable (non-viable) link in the fail-stop model. Any two nodes eventually connected in the dynamic network are eventually connected in the fail-stop network.

3 Informal Description

In this section an informal outline of the protocol and the main ideas leading to it are presented. The presentation begins with a description of a very simple end-to-end protocol using *unbounded sequence numbers*, i.e, messages of unbounded size. This algorithm is then refined through a series of steps, to derive a *bounded* protocol having polynomial complexity.

The simple unbounded protocol involves two types of messages. The "data" message contains some data item which is to be delivered to the receiver; the "acknowledgment" (in short, "ack") message serves to acknowledge the receipt of the data item at the receiver. Both the data and the ack messages carry the (unbounded) sequence number of the data item in question; the data message also carries the data item itself.

The protocol works as follows. Once the sender inputs the data item of sequence number ℓ, it sends the data message (indexed with ℓ) to all its neighbors. Every node, upon receiving this message, forwards it to all its neighbors, unless it already received a data message with higher sequence number. The receiver, upon receiving the data message of sequence number ℓ, sends back an ack message with sequence number ℓ. The sender will input the next data item (with sequence number $\ell + 1$) only after it received the ack for the data item ℓ. Note that the protocol creates a situation where many messages may be in transit on the same link at the same time.

The first very simple modification is intended to guarantee that at a given time, there is at most one message in transit in a given direction on a link. This is achieved by letting every process send the ack of ℓ on every edge the data message with ℓ was received. A process does not send a future data message on an edge, before it receives the ack for the last data message sent on that edge. Though this only means that messages are stored in the process rather than on the channel, the number of messages stored

per edge can be reduced to one, by noticing that it suffices to maintain only the message with highest sequence number. Observe however, that the complexity is still unbounded, since the size of the sequence numbers ℓ is not bounded.

At this point, the following important observation is made. Although the protocol uses an unbounded number of different label values, at each point in time, the number of different label values in the system is linear in the number of edges. Let it first be shown how, assuming that the sender has a *label oracle* telling it whether a given ℓ value exists in the system, a polynomial end-to-end protocol using only bounded size labels can be designed (this protocol is called the *main protocol*). The label oracle enables the sender to compute the set of values that exist in the system, denoted by $\bar{\ell}$.

The idea is that the sender, in order to send a new data item, generates a label that is "greater" than all the labels in $\bar{\ell}$. A well known mechanism that achieves this goal is a bounded sequential time stamp system (see [IL87, DS87].) The time stamp system to be used will be of size N, where N is polynomial in n, and each label has size of $O(N)$ bits. Such a system guarantees that for any set of values $\bar{\ell}$, that has less than N values, a label that is not in $\bar{\ell}$ and is "greater" than any value that is in $\bar{\ell}$ can be found. Note that the system also defines the operation "greater" associated with the labels. Using such a time stamp system and the oracle, the sender can always find a new label, that is greater than all the labels in the system, and is of size polynomial in n.

Most of the effort in the protocol is implementing a mechanism similar to the above label oracle. One would like a node to "know" locally that it is clean of a given label ℓ, that is, all references to it can be eliminated. To achieve this two modifications are introduced. First, the sending of replies to a message with label ℓ is restricted to edges that were traversed by the message ℓ. This implies that a node can receive a reply to message ℓ only on edges on which message ℓ has been sent. Second, a node does not send a new data message on an edge until it receives a reply to the previous message sent along it.

Assume that node v receives a message with label ℓ and forwards it. assume also taht after some time it received a reply for ℓ on all the links on which it forwarded the message, and also received a message labeled later than ℓ. At this point none of the local variables in v need contain reference to label ℓ. Thus, node v creates a *token* that includes its name v, the label ℓ and the list of edges that the message ℓ was sent on. The token is an indication that node v is clear of references to label ℓ.

The remaining unresolved problem is that of getting all the tokens with label ℓ to the sender, so it can deduce that ℓ is not in the system, since all nodes were clean when the tokens were created, and all tokens are in the sender. If the sender collects the tokens from some of the nodes, but not all of them, it can check locally if it has all the tokens created for a given ℓ in the following way. Using the lists of edges (listing on which edges message ℓ was sent) in the tokens of label ℓ, the sender can create a set S of nodes that certainly received the message ℓ. If from every node in S, the sender has received a token, then there is no node in S that sent message ℓ to a node not in S. Since the sender is in S, nodes not in S never received message ℓ.

The problem in getting all the tokens to the sender, is that on one hand, some of the edges that the tokens are sent on may fail. This problem may be solved by duplicating tokens and sending them on different paths. However, duplicating tokens disables the sender from checking locally if it has all the tokens of ℓ that were created.

Assume one could bound the token capacity of the network so that each process apart from the sender (whose capacity is unbounded) could contain at most some fixed number of tokens. After the network's token capacity is reached, the creation of a new token in any process would imply that some token was received by the sender. By simple pigeon hole arguments it can be shown that after a bounded number of such token creations, there would be a message ℓ for which all the tokens were received by the sender.

The solution to the token collection problem is thus a fault tolerant load balancing protocol to assure that tokens are evenly distributed among processes in the system, maintaining the property that every process apart from the sender has bounded capacity. The protocol assures that no matter which com-

munication path becomes eventually connected, there are sufficiently many tokens on this path, and one will eventually be forwarded to the sender.

The basic idea behind the load balancing protocol is the following. Each node has some quantity of tokens. Assume for a moment that the network is synchronous and static. Consider the following protocol. At every even clock tick a node sends a token to each neighbor if it had Δ or less tokens than it, and sends nothing otherwise. At every odd clock tick, each node updates its neighbors about the number of tokens it has. After a polynomial number of iterations the protocol will, given that $\Delta = \Omega(n)$, converge to a steady state (i.e. no tokens are sent). The surprising fact is that a very similar protocol will converge, and with polynomial communication, in an asynchronous network where links may fail.

In the protocol, the number of tokens stored in a node is bounded by a polynomial in n. In order to enforce this bound, a node that has more than a certain amount of tokens is *blocked*, and does not respond to messages of the main protocol. This guarantees that a node will not generate additional tokens locally. Furthermore, it can be proven that this rule does not cause deadlock. The bound on the number of tokens in each node implies that the number of tokens that can be in the network at any point in time is polynomial in n (which is less than N). Any label that is sent, for which not all the tokens have been collected, is assumed to exist in the system. The value of N will be chosen to be more than the "label capacity" of the network, and thus the sender will always be able to generate a new label.

The formal proof of the load balancing protocol uses amortized analysis to show that though it could be that some given token cycles forever in the network, the total number of tokens sent in the period of time between two successive message receipts at the receiver, is bounded by a polynomial in n. In the rest of this section we sketch an intuitive argument for the complexity of the load balancing protocol. Note that every token that is sent creates at most $2n$ updates, therefore , to bound the complexity, it suffices to bound the number of token messages sent. As long as no new token is created, and no token is received by the sender, the number of all the tokens in the network remains unchanged. As mentioned before, the aim of the protocol is to distribute the tokens evenly between the nodes. Consider an energy function $\mathcal{E}$ that is the square of number of tokens in each node at a given time. Clearly, this function achieves its minimum, when the tokens are evenly distributed, that is, when there is not enough energy for a token to be sent, because no two processes have a token difference of Δ. In a static and synchronous system, each token sent from a process to one with Δ less tokens, reduces $\mathcal{E}$ by at least n, for $\Delta \geq 2n$. Unfortunately in our case, due to the asynchrony, updates might be delayed, and based on outdated information, there may be "bad" tokens whose receipt will increase $\mathcal{E}$ since they were mistakenly sent to processes having more tokens. However, in order for such a "bad" token send to occur, updates of many tokens must be delayed. The property that can be shown, and is crucial to the complexity analysis, is that in order to create the many delayed updates necessary for one "bad" token to be sent, many "good" token sends must occur, and so it cannot be that "bad" tokens are continuously providing the energy for more "bad" tokens to be sent.

4 The Main Protocol

4.1 Preliminaries

In the following subsections, the code of the *main protocol* meeting the desired end-to-end properties is described.

In the presentation of the code, we use the language of *guarded commands* of Dijkstra [DF88], where a process code of the form $G_1 \longrightarrow A_1 \Box G_2 \longrightarrow A_2 \Box \ldots G_k \longrightarrow A_k$ is repeatedly executed. In each execution, of all the guards G_i that are true, an arbitrary i is selected and A_i is performed. A guard G_i is a conjunction of predicates. The "**receive** M **on** e" guard is true if message M is available in the "incoming messages" buffer of channel e. The execution of the corresponding statement includes the

receipt of the message, and the deletion of the message from the buffer.

To simplify the description of the properties and proofs, global time is assumed. The execution of each guarded command in the code of a process is thus termed an *event*, and is assumed to be atomic. The state of the system at any time consists of the local process states $S_v^t, v \in V$, and channel states $C_e^t, e \in E$, as they were following the latest event in every process. The subscripts and superscripts, added to variables ($\mathbf{var}_v^t$), denote the local process state.

4.2 Properties of the main protocol

Let the input sequence $I = (D_0, D_1, ...)$ be an infinite sequence of data items to be input, one after the other, to the sender. Similarly, let $O^t = (D_0, D_1, ...)$ be the sequence of data items output by the receiver in all output events preceding some state S^t. Then the following properties must be met by an *End-to-End* communication protocol.

P1 *Safety* In any state S^t, the output sequence O^t is a prefix of I.

P2 *Liveness* For each data element D_k in I, there is a state S^t in which it is added to O^t.

One can easily see that *P1-2* are equivalent to the definition in Section 2.2.

4.3 Creating a Virtual Network

In presenting the protocols below, it is assumed that both sender and receiver have a single link that is always viable, connecting them to the rest of the network. This assumption is made with no loss of generality, since one can effectively "split" the sender (or the receiver) node into two virtual parts, a "special" sender node responsible for the input (output) interface, and an "ordinary" node, responsible for the interface with the rest of the network nodes. Thus, all the network nodes can be partitioned in 3 categories: sender, receiver, and the "ordinary" nodes, where the sender and receiver are each connected to one "ordinary" node. While all the ordinary nodes perform the same protocol, special protocols are designed for the sender and the receiver.

In the *main protocol* presented below, both sender and receiver are split into special and normal nodes. In the *label protocol* presented in the sequel, only the sender is split.

4.4 A sequential time-stamp system

The algorithm used to generate the labels added to the data items transmitted by the main protocol, is a sequential time stamp system algorithm.

A *sequential time stamp system* consists of a set of labels $\bar{\ell} = \{\ell \mid \ell \in L\}$, $|\bar{\ell}| \leq N$ for some constant N, and a labeling function $\mathcal{L}(\bar{\ell})$. The label values in the range L are ordered by the irreflexive and antisymmetric relation $\prec$, described in terms of a *precedence graph* $G = (L, \prec)$. If the cardinality of L is bounded, the time stamp system is said to be bounded, i.e. labels are of a bounded size. The labeling function $\mathcal{L} : L^{N-1} \mapsto L$, given a set of $N-1$ labels *totally ordered* by $\prec$, returns a new label ℓ, greater by the order $\prec$ than all $N-1$ others. A more elaborate description of the properties, upper and lower bounds of sequential time-stamp system constructions, due to Israeli and Li, can be found in [IL87].

The sequential time-stamp system to be used in our construction is a variant due to [DS87], of a construction by Lamport [Lam86]. Let the range L of labels (nodes) in the precedence graph G be of size $|L| = N \cdot 2^N$. The label value ℓ of each node is thus a boolean vector of size $N + \log N$. Let $\log N$ bits of the vector ℓ be a "cluster number" (denoted $C(\ell)$), $C(\ell) = \ell[(N-1)...(N-1+\log N)] \in \{0..N-1\}$.

```
Procedure L(ℓ̄);
    ℓ[(N−1)..(N−1+log N)] := (i : ∀ℓ' ∈ ℓ̄, C(ℓ') ≠ i);
    for all ℓ' ∈ ℓ̄ do
        if C(ℓ) > C(ℓ') then ℓ[C(ℓ')] := ℓ'[C(ℓ)] fi;
        if C(ℓ) < C(ℓ') then ℓ[C(ℓ')] := not ℓ'[C(ℓ)] fi;
    od;
    return ℓ;
end L;
```

Figure 1: The labeling function

```
receive REPLY (ℓ) ⟶
    trying := false;
□
free_label_available(ℓ̄); trying := false ; ⟶
    Input value;
    ℓ := L(ℓ̄);
    ℓ̄ := ℓ̄ ∪ {ℓ};
    send MSG (ℓ, value)
    trying := true
```

Figure 2: Code of the main protocol: sender

Let the remaining N bits $\ell[i], i \in \{0..N-1\}$, identify a node in the cluster. The following is thus the definition of the relation $\prec$, where $\ell' \prec \ell$ if there is a directed edge from the node of ℓ to that of ℓ' in G.

$$\ell' \prec \ell = \begin{cases} \mathit{true} & \text{if } (C(\ell') < C(\ell)) \wedge (\ell[C(\ell')] = \ell'[C(\ell)]) \\ & \text{or } (C(\ell') > C(\ell)) \wedge (\ell[C(\ell')] \neq \ell'[C(\ell)]) \\ \mathit{false} & \text{otherwise.} \end{cases}$$

That is, labels of nodes in the same cluster are unrelated, and nodes in different clusters are always related. Figure 1 is the definition of the labeling function $\mathcal{L}$.

Note that for the sake of simplicity, the bit $\ell[C(\ell)]$ exists in every label ℓ, though it is never set (all nodes with either setting of this bit are equally usable). Proof that the above construction has the properties of a sequential time-stamp system, can be found in [DS87].

The sequential time stamp system used in the protocol has a label set of size $N = 1 + (\Delta + 5)n^2$. The predicate function *free_label_available*$[\bar{\ell}]$, that indicates whether there is a free label value that can be used, is $|\bar{\ell}| < N$.

4.5 Sender's protocol

The code of the protocol is presented in Figure 2.

The labeling protocol maintains the set $\bar{\ell}$ of labels which are believed to be existing in the system. The boolean function *free_label_available* $(\bar{\ell})$ returns value *true* only if the labeling function $\mathcal{L}$ can be applied to return a new label. The sender also maintains boolean variable **trying**, which is *true* if the

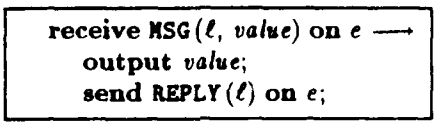

Figure 3: Code of the main protocol: receiver

sender is in the process of delivering next message to the receiver, and a variable **value**, which is the value (contents) of the current data item to be delivered.

The sender reads the input into its variable *value* when both `trying` = *false* (current data item has been delivered) and *free_labelLavailable* ($\bar{\ell}$) = *true*. At that time the sender computes a new label as $\ell := \mathcal{L}(\bar{\ell})$. It then transmits the information message MSG(ℓ, *value*) over its (only) outgoing link.

4.6 Receiver's protocol

The receiver's protocol is given in Figure 3.

For every MSG(ℓ, *value*) received, receiver outputs the contents **value** of the message and sends back REPLY(ℓ).

4.7 Ordinary node's protocol

The code itself is given in Figure 4.

The operations of this protocol are performed only while a node is not "blocked" by the label protocol, a condition that is determined based on the variable **blocked**. For that purpose, the node maintains boolean variable **blocked**, which is *true* if the node is blocked. In the sequel, we describe operations performed at the node while it is not blocked.

An important property of the protocol is that when the sender sends a message MSG(ℓ, *value*), in every node in the network the variables that depend on the value ℓ are at their initial value, in other words, there is no already existing reference to this label in the system.

The node maintains variables **latest_ℓ** - the label of the latest message received, and **latest_value** - the value of the data item sent in of the latest message. Also, it maintains a number of arrays, each indexed by label values. We use arrays with entries ℓ in the code, for the sake of simplicity only. To achieve polynomial space, the actual implementation of these arrays would be in the form of a space efficient data structure such as a linked list.

The variable **rec_msg**[ℓ] (**rec_reply**[ℓ]) is a boolean array, whose ℓ^{th} entry is *true* if MSG(ℓ, *value*) (REPLY(ℓ)) has been was received, but whose token with label ℓ has not been generated yet. The following arrays, **edges_sent_msg**, **edges_rec_msg**, **edges_sent_reply**, and **edges_rec_reply**, are indexed by ℓ. Each entry is a set of edges on which a MSG(ℓ, *value*) or a REPLY(ℓ) message has been sent or received. Also, for each edge e, we define a variable **status** [e], receiving values *clean* or *dirty*. If **status** [e] = *dirty* then there exits ℓ, such that $e \in$ **edges_sent_msg**[ℓ], but, at the same time, $e \notin$ **edges_rec_reply**[ℓ].

There is a simple update rule for the variable **status**. Once a message is sent on e, the sending node sets **status** [e] := *dirty*. Upon receiving a REPLY, **status** [e] := *clean* is performed.

When a node receives MSG(ℓ,*value*) on edge e, it acts as follows. It adds e to **edges_rec_msg** [ℓ], and then checks whether **latest_ℓ** $\prec \ell$. If so, then the message is a *new* one; in this case it updates the variables associated with the latest message, setting **latest_ℓ** := ℓ, **latest_value** := *value*, and **rec_msg**[ℓ] := *true*.

```
¬ blocked; receive MSG(ℓ,value) on e ⟶
   edges_rec_msg[ℓ] := edges_rec_msg[ℓ] ∪ {e};
   if latest_ℓ ≺ ℓ then
      latest_ℓ := ℓ;
      latest_value := value;
      rec_msg[ℓ] := true;
      for all ℓ' if ℓ' ≺ ℓ ∧ rec_msg[ℓ'] then rec_reply[ℓ'] := true fi
□
¬blocked ; ¬rec_reply[ℓ] ; rec_msg[ℓ] ; status[e] = clean; e ∉ edges_sent_msg[latest_ℓ] ⟶
   send MSG(latest_ℓ,latest_value) on e;
   status[e] := dirty;
   edges_sent_msg[latest_ℓ] := edges_sent_msg[latest_ℓ] ∪ {e};
□
¬ blocked; receive REPLY(ℓ) on e ⟶
   status[e] := clean;
   edges_rec_reply[ℓ] := edges_rec_reply[ℓ] ∪ {e};
   for all ℓ', if ℓ' ⪯ ℓ ∧ rec_msg[ℓ'] then rec_reply[ℓ'] := true fi;        /*this includes ℓ itself */
□
¬blocked ; rec_reply[ℓ] ; e ∈ edges_rec_msg[ℓ] ; e ∉ edges_sent_reply[ℓ] ⟶
   send REPLY(ℓ) on e;
   edges_sent_reply[ℓ] := edges_sent_reply[ℓ] ∪ {e};
□
¬blocked; edges_sent_msg[ℓ] = edges_rec_reply[ℓ] ≠ ∅; ℓ ≺ latest_ℓ ⟶
   call Procedure NEW_TOKEN (v, ℓ, edges_sent_msg[ℓ]);        /*triggers the label protocol;*/
   edges_sent_msg[ℓ] := ∅; edges_sent_reply[ℓ] := ∅; rec_reply[ℓ] := false;
   edges_rec_msg[ℓ] := ∅; edges_rec_reply[ℓ] := ∅; rec_msg[ℓ] := false;
```

Figure 4: Code of the main protocol

The receipt of a message MSG(ℓ, *value*) implies the receiving can send a REPLY for all labels $\ell' \prec \ell$, since a reply to all smaller messages must have already been received by the sender. To this end, the node sets **rec_reply**[ℓ'] := *true* for all $\ell' \prec \ell$, for which **rec_msg**[ℓ'] = *true*, i.e. a message has been received.

The latest message is then forwarded on each edge e at most once, if the edge status is clean, and no REPLY has been received for that message. It is clear why a message is forwarded only once, and the reason for not forwarding after a REPLY message was received is simply because the receiver has already received this message.

Whenever REPLY(ℓ) is received on e, the node sets **status**[e] := *clean*, adds e to **edges_rec_reply**[ℓ], and sets **rec_reply**[ℓ'] := *true* for all $\ell' \preceq \ell$, for which **rec_msg**[ℓ'] = *true*. Note that a message MSG(ℓ, *value*) serves as a REPLY only to labels $\ell' \prec \ell$, while a REPLY(ℓ) is also a reply for the label ℓ. The reason for this difference is quite obvious: in this case node can deduce that label, too has already arrived at the receiver.

If **rec_reply**[ℓ] = *true*, the node will forward the reply labeled ℓ on each edge e, provided MSG was received on that edge ($e \in$ **edges_rec_msg**[ℓ]), and this is the first reply on this edge $e \notin$ **edges_sent_reply**[ℓ].

Following the receipt of MSG(ℓ',*value*), with $\ell \prec \ell'$, and the receipt of all replies to the message, there is no reference to this label in the given node. At this time, the node generates a new "token" with label ℓ, to indicate that it and all its outgoing channels are clear of this label. The token contains the name of the node, the label ℓ and the list of edges **edges_sent_msg**[ℓ], on which MSG(ℓ,*value*) has been sent. This serves as input to the label protocol.

```
receive TOKEN(node,ℓ,edges) on e ⟶
    send TOKEN_ACK on e;
    clean_nodes[ℓ] := clean_nodes[ℓ] ∪ {node};
    used_edges[ℓ] := used_edges[ℓ] ∪ {edges}
    if ∀(u → w) ∈ used_edges[ℓ], w ∈ clean_nodes[ℓ] then ℓ̄ := ℓ̄ − {ℓ} fi;        /* ℓ is dead */
```

Figure 5: Code of the label protocol at the sender

5 Label protocol

5.1 Properties of the label protocol

The correctness of the main protocol depends on the properties of the label protocol, providing new unused label values that can be added to the data items transmitted. The main interface between the main protocol and label protocol is as follows. There exists of a set of labels in the range L, and a relation $\preceq$ among them. There is a predicate function *free_labelLavailable*$(\bar{\ell})$, indicating that the labeling function $\mathcal{L}(\bar{\ell})$ can be executed correctly, returning a new label value to be used. The labeling function $\mathcal{L} : L^* \mapsto L$ returns a label $\ell \in L$, and adds this label to $\bar{\ell}$. The label protocol is allowed to block (by changing the value of the variable *blocked*) the progress of the main protocol in any given process, in order to maintain the property of having a free label available to the sender always. It will suffice that in any system state, the *label protocol* will have the following properties[1]:

Q1 *comparability*: In any state S^t, if a label ℓ exists in a process state S_v^t or channel state C_e^t, then $\ell \in \bar{\ell}$ in S_s^t.

Q2 *ordering*: The labels in $\bar{\ell}$ are totally ordered by $\prec$, where if *free_labelLavailable*$(\bar{\ell})$ holds, for any label $\ell' \in \bar{\ell}$, it is the case that $\ell' \prec \mathcal{L}(\bar{\ell})$.

Q3 *availability*: In any state S_s^t, *free_labelLavailable*$(\bar{\ell})$ holds.

Q4 *non-blocking*: Let B^t be the set of nodes for which in any time $t' \geq t$, **blocked** = *true*. There is no time t'', such that $B^{t''}$ forms a cut between the sender and the receiver.

The above properties formalize the idea that: 1. the order $\prec$ indicates the order in which labels were generated, 2. all label values in the system are totally ordered by $\prec$, 3. a free label is always available, and 4. the nodes that are blocked by the label protocol, never form a cut between the sender and receiver.

As mentioned earlier, in the label protocol, only the *sender* is split into a special node and normal node. All other nodes, including the receiver, are treated as normal nodes.

5.2 Sender's protocol

The sender's protocol is given in Figure 5.

The purpose of this protocol is to determine which labels are no longer in use in the system. For each label ℓ, the sender tries to establish the nodes and edges having a variable or message of this label in the network. The variable **used_edges**$[\ell]$ denotes the set of edges traversed by messages **MSG**$(\ell,value)$, and variable **clean_nodes**$[\ell]$ denotes all the nodes which will not send label ℓ any more.

[1]The exact definition of *exists* as in *Q1* is given in Definitions 6.1 and 6.2.

As explained in the previous section, a token labeled ℓ and generated by a given node, contains the set of edges on which MSG(ℓ,*value*) has been forwarded by this node.

In general, whenever the sender receives TOKEN(*node*,ℓ,*edges*) message, it adds *node* to clean_nodes[ℓ], and adds *edges* to used_edges[ℓ]. Whenever for each edge $(u,w) \in$ used_edges[ℓ], both u and w exist in clean_nodes[ℓ], the sender deduces that label ℓ does not exist in the network and thus deletes it from $\bar{\ell}$.

5.3 Ordinary node's protocol

The ordinary node's protocol is presented in Figure 7.

```
Procedure NEW_TOKEN (node, ℓ, edges)
   tokens := tokens+1;
   if tokens = n · Δ then blocked := true fi;
   token_set := token_set ∪ {(node, ℓ, edges)};
   for all e ≠ (v, s) do
      unreported[e] := unreported[e] + 1 od fi;
end NEW_TOKEN;
```

```
Procedure UPDATE (e)
   send UPDATE(unreported[e]) on e;
   wait_update_ack[e] := true;
   unreported[e] := 0
end UPDATE;
```

Figure 6: The procedures

The ordinary nodes coordinate the delivery of tokens to the sender. The variable token_set denotes the set of tokens which have accumulated at this node, and the variable tokens denotes the cardinality of this set. A node updates its neighbors about the change in tokens. In order not to have many UPDATE messages in transit at the same time on a single edge e, the node accumulates the net change between two UPDATE messages in the local variable unreported[e]. In the proof we will claim that at any time t, the sum of $\texttt{estimate}_u[v]$, $\texttt{unreported}_v[(v,u)]$, and the value in the UPDATE message in transit from v to u (if one exists) is equal to $\texttt{tokens}_v$.

In general, whenever TOKEN(*node*,ℓ,*edges*) arrives at a node on edge e, the node adds it to token_set, sets tokens := tokens + 1, and sends a TOKEN_ACK message back, which acknowledges receipt of TOKEN message. Also, for all edges e, the node increments unreported[e] by 1. For each adjacent edge e, a node maintains a boolean variable wait_token_ack[e] which receives value *false* after TOKEN message was sent on e and before TOKEN_ACK is received.

The label protocol has a parameter $\Delta = 112n$. Each time the cardinality of the token set exceeds a certain constant $n \cdot \Delta$, the node sets blocked := *true*, disabling the operation of the main protocol. This effectively bounds the number of tokens in a node.

The purpose of the algorithm is to push all the tokens towards the sender. The tokens are carried by messages TOKEN. When such message is received, TOKEN_ACK message is sent back to acknowledge its arrival.

The updating on edge e is performed by sending message UPDATE(unreported[e]) on that edge. Receipt of this message is acknowledged by a special message UPDATE_ACK. For each edge e, node maintains boolean flag wait_update_ack[e], which receives value *false* after sending UPDATE one e and before receiving UPDATE_ACK from e. The UPDATE(unreported[e]) message is sent on e whenever wait_update_ack[e] = *false* and unreported[e] $\neq 0$. At this time, unreported[e] := 0 is set. Each node keeps for each edge u an estimate estimate[u] for the variable tokens on the other end-point of that edge. Whenever node receives UPDATE(x) message on edge e, it adds x to estimate[u], and sends back UPDATE_ACK.

As it is not clear which path to the sender is operational, the algorithm simply tries to balance the tokens more or less evenly between the nodes. That is, a node tries to push tokens to its neighbors

```
receive TOKEN(node,ℓ,edges) on e ⟶
   if ¬wait_update_ack(e) ∧ unreported[e] ≠ 0 then call Procedure UPDATE (e) fi
   send TOKEN_ACK on e;                                   /* ack the message */ ;
   call Procedure NEW_TOKEN (node, ℓ, edges)
□
¬wait_update_ack(e); unreported[e] ≠ 0 ⟶
   call Procedure UPDATE (e)
□
tokens - estimate[u] > Δ; ¬ wait_token_ack(e); (unreported[e] = 0) ∨ wait_update_ack; ⟶
   some_token := select_token (token_set);                /*select an arbitrary token*/
   send TOKEN(some_token ) on e;
   wait_token_ack[e] := true;
   token_set := token_set − {some_token};
   tokens := tokens − 1;
   if tokens < n · Δ then blocked := false fi;
   for all e ≠ (v, s) do unreported[e'] := unreported[e'] − 1 od;
□
receive UPDATE(x) on e ⟶
   estimate[u] := estimate[u]+x;
   send UPDATE_ACK on e;
□
receive TOKEN_ACK on e ⟶
   wait_token_ack[e] := false;
□
receive UPDATE_ACK on e ⟶
   wait_update_ack[e] := false;
```

Figure 7: Code of the label protocol

along any *any* edge u, such **tokens** - **estimate**$[u] > \Delta$, i.e. the amount of tokens on the other side is estimated to be less than amount of tokens at the node by at least Δ. However, the actual transmission is postponed until no **TOKEN** or **TOKEN_ACK** message is pending on the link, namely **wait_token_ack**$(e) =$ *false*.

One of subtleties of the protocol, is that upon arrival of a **TOKEN** a node must send an **UPDATE** message if there is one to be sent, before it sends the **TOKEN_ACK**. This is intended to achieve the same effect as the update phases in the synchronous version of the algorithm informally described before. Failure to do so would allow the actual token distribution to differ significantly from the one known to the processors, and would would result in an exponential increase in complexity.

6 Correctness of the Main Protocol

In this section, it is proven that under the assumption that the *label protocol* is correct, that is, has properties *Q1-4*, the *main protocol* meets properties *P1-2*. Then, in Section 7, the proof is completed by proving that the *label protocol* meets properties *Q1-4*. Finally, in Section 8, the complexity of the complete *End-to-End* protocol is shown to be polynomial.

Recall again that as a convention we superscript variables by time and subscript them by the node they belong to, e.g. $\mathbf{tokens}_v^t$ is the value of the local variable **tokens** of node v at time t (i.e. at S_v^t).

Definition 6.1 If in a processor state S_v^t, there is an entry for label ℓ in one of the variables `edges_rec_msg`, `edges_sent_msg`, `edges_sent_reply`, `edges_rec_reply`, `rec_reply`, or `rec_msg`, or `latest_`$\ell = \ell$, then ℓ is said to *exist* in node v in state S_v^t.

Definition 6.2 If in a channel state C_e^t, a message MSG$(\ell, value)$ or REPLY(ℓ) is in transit, then ℓ is said to *exist* in the channel e in state S^t.

Lemma 6.3 Let t_0 be a time at which the sender added ℓ to $\bar{\ell}$, and t_1 the earliest time after t_0 at which the sender deleted ℓ from $\bar{\ell}$. On each edge $e = (u, v)$, the message MSG$(\ell, value)$ was sent at most once between time t_0 and t_1.

Proof: Assume by way of contradiction that it was sent twice at times t_3 and t_4, where $t_3 < t_4$. The processor u sending at time t_3 the MSG$(\ell, value)$ on edge e, had `latest_`$\ell = \ell$, and added e to `edges_sent_msg`$[\ell]$. In order to have sent the message at time t_4, u must not have e in `edges_sent_msg`$[\ell]$ in $S_u^{t_4}$. The edge e could have been deleted from `edges_sent_msg`$[\ell]$ only by this variable being set to $\emptyset$. Together with the setting of `edges_sent_msg`$[\ell]$ to $\emptyset$, `rec_msg`$[\ell]$ must have been set to *false*. By the code, the setting could have been done only after `latest_`ℓ was set to a label ℓ', $\ell \prec \ell'$. By *Q1-2*, at time t_0, ℓ was greater than all labels in S^t, and since ℓ remains in $\bar{\ell}$ till time t_1, any new label ℓ' added at time t, $t_0 < t < t_1$, is greater than any of the labels in S^t.

We claim that `latest_`ℓ could not have been less than ℓ during $[t_3, t_4]$. Indeed, consider the first time $\hat{\tau}$, $t_3 \leq \hat{\tau} \leq t_4$ `latest_`$\ell = \hat{\ell} \prec \ell$; moreover assume that $\tilde{\ell}$ has been the previous value of `latest_`ℓ. Clearly, $\tilde{\ell} \prec \hat{\ell}$. Assume labels $\hat{\ell}$, $\tilde{\ell}$ were generated at the sender at times $\hat{t}$, and $\tilde{t}$, respectively. It must be that $\hat{t} \notin [t_0, t_1]$, since in this interval, $\ell \in \bar{\ell}$, and it would be the case that $\ell \prec \hat{\ell}$, a contradiction. Consequently, $\hat{t} < t_0$ and $\hat{\ell} \in S^{t_0}$.

Since at time t_3, `latest_`$\ell = \ell$, and by the fact that $\hat{t}$ is the first time since t_3 that `latest_`$\ell \prec \ell$, it follows that $\ell \preceq \tilde{\ell}$. Therefore, $\tilde{t} \in [t_0, t_1]$. Since $\hat{\ell} \in S^{t_0}$, and could not be generated in $[t_0, t_1]$, by definition, $\hat{\ell} \in S^{\tilde{t}}$, and therefore it is in $\bar{\ell}$ at time $\tilde{t}$ (by *Q1*). By *Q2*, at time $\tilde{t}$, $\tilde{\ell} \succ \ell'$, for all $\ell' \in \bar{\ell}$. In particular $\tilde{\ell} \succ \hat{\ell}$, since $\hat{\ell} \in \bar{\ell}$ at time $\tilde{t}$.

Thus, `rec_msg`$[\ell]$ could not have been set to *true* once again before time t_4, a contradiction to the fact that processor u sent MSG$(\ell, value)$ on e at time t_4. □

Theorem 6.4 If the *labeling protocol* has properties *Q1-2*, then the *main protocol* has property *P1*.

Proof: Assume by way of contradiction that the above does not hold. By the senders protocol, the values in I are input one after the other, the order of input time corresponding to the order in I. One of the following cases must thus be true:

1. There are values D_k and D_{k+1}, input respectively at times t_1 and t_2, $t_1 < t_2$, and there is no output of D_k at a time t_3, $t_3 < t_4$, where t_4 is the output time of D_{k+1}.
2. There is a value D_k input at time t_1 and output at two different times t_2 and t_3.

Case 1: By the protocol, with each value D_k, a label ℓ is associated by the sender. In all processors, the data item sent in a message is `latest_value`, and is associated with `latest_`ℓ, both always updated in the same event, so it is never the case that the label ℓ associated in a MSG(ℓ, D_k) with a value D_k, is ever changed. Let us thus denote the label assigned by the sender to D_k by ℓ_k. By the senders protocol, a MSG(ℓ_k, D_k) must have been sent by the sender and REPLY(ℓ_k) received prior to the input of D_{k+1}, that is, between t_1 and t_2. By property *Q1*, immediately before the input of D_k was performed, ℓ_k did not exist in S^t. Thus, in order for a REPLY(ℓ_k) to have been received by the sender, it must be that the receiver

performed an output event at some time $t_3 < t_2$. Since $t_2 < t_4$ by definition, a contradiction to the first case is derived.

Case 2: There is only one edge $e = (u, r)$ leading to the receiver processor, on which a $\texttt{MSG}(\ell_k, D_k)$ could have been received. By the protocol, a value D_k is always associated with one label ℓ_k, and as long as D_k is in S^t, this label is in $\bar{\ell}$. Thus, by Lemma 6.3 the receiver could have received a $\texttt{MSG}(\ell_k, D_k)$ only once on e, and output the value D_k only once. □

The following is the proof of the liveness property *P2* of the main protocol.

Claim 6.5 There exists in the communication graph $G = (V, E)$, a path $p = s, v_1, .., v_k, r$ of nodes, each edge of which is eventually connected and each node of which is eventually ¬blocked.

Proof: Define the following graph G'. The set of nodes includes those of processors v, for which given any time t, there is a time $t' > t$, such that in state $S_v^{t'}$ the value of $\texttt{blocked}_v$ is *false*. The set of edges includes the eventually connected edges among nodes. By *Q4*, there is no cut between the receiver and sender in G'. It is known, from graph theory, that if is there is no cut between two nodes, then there is a path connecting them. □

Lemma 6.6 If $\texttt{MSG}(\ell, value)$ is sent on edge (v_i, v_{i+1}) in p, (as in Claim 6.5) then eventually a $\texttt{REPLY}(\ell)$ must be received on (v_{i+1}, v_i).

Proof: Assume by way of contradiction that there is an edge for which the lemma does not hold. Consider the edge (v_i, v_{i+1}) *closest to the receiver* for which it does not hold. Consider the last unreplied message $\texttt{MSG}(\ell, value)$ sent on (v_i, v_{i+1}). Since the edge (v_i, v_{i+1}) is eventually connected, there exists a time t_1 such that $\texttt{MSG}(\ell, value)$ is received at node v_{i+1}. If at state $S_{v_{i+1}}^{t_1}$, $\ell \prec \texttt{latest\_}\ell_{v_{i+1}}$ then, by the code of the main protocol, a message $\texttt{REPLY}(\ell)$ is sent from v_{i+1} to v_i, and eventually will be received. Therefore, at state $S_{v_{i+1}}^{t_1}$, $\texttt{latest\_}\ell_{v_{i+1}} \preceq \ell$ and $\texttt{rec\_reply}[\ell]$ is *false*.

Since the Lemma holds for edge (v_{i+1}, v_{i+2}), there is a time t_2 such that at state $S_{v_{i+1}}^{t_2}$ the value of $\texttt{state}(v_{i+1}, v_{i+2})$ is *clean*.

If $\texttt{rec\_reply}[\ell]$ is *true* in $S_{v_{i+1}}^{t_2}$ a $\texttt{REPLY}(\ell)$ is sent to v_i and we are done. Therefore, the interesting case is when $\texttt{rec\_reply}[\ell]$ is *false* in $S_{v_{i+1}}^{t_2}$. In this case v_{i+1} sends $\texttt{MSG}(\ell, value)$ on (v_{i+1}, v_{i+2}). Since the Lemma holds for (v_{i+1}, v_{i+2}), there exists a time t_3 such that a $\texttt{REPLY}(\ell)$ is received from v_{i+2}, and in state $S_{v_{i+2}}^{t_3}$, $\texttt{rec\_reply}[\ell]$ is thus *true*. Therefore, at some time between t_2 and t_3, a $\texttt{REPLY}(\ell)$ was sent to v_i.

Since the edge (v_{i+1}, v_i) is eventually connected, the $\texttt{REPLY}(\ell)$ is received at v_i, contradicting the assumption. □

Corollary 6.7 If in some state $S_{v_i}^t$ the $\texttt{status}[(v_i, v_{i+1})] = \textit{dirty}$ then there exists a state $S_{v_i}^{t'}$, $t < t'$, in which $\texttt{status}[(v_i, v_{i+1})] = \textit{clean}$. □

Theorem 6.8 If the *labeling protocol* has properties *Q1-4*, then the *main protocol* has property *P2*.

Proof: Consider the sender, in a state S_s^t in which **trying** is true. By Corollary 6.7, there is a time $t_1 > t$, such that in $S_s^{t_1}$, $\texttt{status}[(s, v_1)] = \textit{clean}$. By the code of the protocol, the sender sends $\texttt{MSG}(\ell, value)$ to v_1. By Lemma 6.6, there is a time $t_2 > t_1$, such that a $\texttt{REPLY}(\ell)$ is received. □

7 Correctness of the Label Protocol

The following is the proof that the label protocol meets properties *Q1-4*.

Theorem 7.1 The *labeling protocol* has property *Q2*.

Proof: Follows directly from the properties of the *sequential time stamp system* of size $N = 1+(\Delta+5)n^2$, where the predicate function *free_labeL_available*$(\bar{\ell})$ is just $|\bar{\ell}| < N$. □

Definition 7.2 A *token interval* is the interval $[t_0, t_1]$ from the time t_0 in which the TOKEN was sent, till the time t_1 in which a TOKEN_ACK was received for it.

Claim 7.3 In any channel $e = (u, v)$ from, the the *token intervals* of TOKENs sent in a given direction, are disjoint.

Proof: Follows from the code, since a REPLY must be received for the latest sent TOKEN, before the following TOKEN can be sent. □

Definition 7.4 Let UPDATE(x) be a message that is sent from v to u before time t, and not received until time t. (There is at most one such message UPDATE at any time t.) Define the "dummy variable" $\texttt{UPDATE}_v^t[u]$ to have the value x, if such a message UPDATE(x) exists, otherwise $\texttt{UPDATE}_v^t[u] = 0$ (this dummy variable will also be used in the complexity proof).

The following will lead to the proof that property *Q3* is met.

Lemma 7.5 In any state S^t, if in S_v^t **tokens**$= x1$ and $\texttt{unreported}_v[e] = x2$, and in S_u^t **estimate**$= y$, and $x1 + x2 \neq y$, then in $C_{(u,v)}^t$, there is an UPDATE$(x1 - x2 - y)$ message from v to u.

proof: Proof is by induction on the sequence of all events by v and u. Initially $x1 = x2 = y = 0$. Assume the claim holds in state S^{t_1}, and let it be proven for any following state S^{t_2}. If the event between t_1 and t_2 was an *increment or decrement of* **tokens**, there was also a corresponding increment or decrement of $\texttt{unreported}_v[e]$, and the claim holds. If the event is a send event of an UPDATE(x) message, there is a corresponding assignment of **unreported**$[e]$ to 0. If the event is a receive event of an UPDATE(x) message by u, there is a corresponding adding of x to **estimate**(v). In all other events there is no change of any of the related variables, and so the claim holds. □

Lemma 7.6 In any state S_v^t, **tokens** is at most $\Delta n + 3n$.

Proof: Assume by way of contradiction that the claim does not hold. Consider the earliest time t in which a process v in state S_v^t, had **tokens** $> \Delta n + 3n$. Let t' be the latest time before t, in which v in $S_v^{t'}$ had **tokens** $= \Delta n$ and following which **tokens** $\geq \Delta n$ in any $S_v^{t''}$, $t'' \in [t', t]$. (I.e. the maximal interval that ends in t in which **blocked**=*true* in v.)

Since $3n$ more tokens were added to **tokens** during $[t', t]$, and no **new_token** was generated by v, because it was in state **blocked**, there exists a process u, from which v received at least 3 tokens on channel $e = (u, v)$ during $[t', t]$. Recall that by Claim 7.3, the token intervals of these three TOKENs, are disjoint.

The main argument of the proof is that in the state before u sent the third TOKEN to v during $[t', t]$, $\texttt{estimate}_u[v]$ was at least Δn. Since u sent the third TOKEN, $\texttt{tokens}_u$ was at least $\Delta n + \Delta$. Noting that $\Delta > 3n$, a contradiction to the fact that v was the first node to have $\Delta n + 3n$ tokens is reached. The rest of the proof will show that in fact at the time the third TOKEN was sent, $\texttt{estimate}_u[v] \geq \Delta n$.

By Lemma 7.5, at time t', $\texttt{tokens}_v = \texttt{unreported}_v[e] + \texttt{estimate}_u[v] + \texttt{UPDATE}(e)$. Since the number of tokens in v, during the time interval $[t', t]$, is at least Δn, the value of $\texttt{unreported}_v[e]$, till the next send event of an UPDATE message, is always at least $\texttt{unreported}_v^{t'}[e]$, and this number has to be positive.

Consider the first time after t', where $\texttt{unreported}_v[e] = 0$. This time occurs before v receives the second TOKEN from u. At time t', if $\texttt{unreported}_v[e] > 0$, then v must be waiting for an UPDATE_ACK. Node u will send the UPDATE_ACK before it sends the second token (since it receives the UPDATE before the token_ack). Therefore, v will send an UPDATE before the second TOKEN is received. Just after the send of the UPDATE, the value of $\texttt{unreported}_v[e] = 0$. Thus, $\texttt{estimate}_u[v] + \texttt{UPDATE}(e) \geq \Delta n$. The UPDATE must be received at node u before the receipt of TOKEN_ACK for the second TOKEN. This implies that $\texttt{estimate}_u[v]$ at this time is at least Δn. It is clear that $\texttt{estimate}_u[v]$ will remain larger than Δn at least till time t, implying that at the time TOKEN was sent, $\texttt{estimate}_u[v] \geq \Delta n$. This contradicts the assumption that such a time t exists. □

Lemma 7.7 In any state S^t of the network, the sum of the number of tokens in each node (i.e. $\sum_v \texttt{tokens}_v^t$), plus the tokens in all the channels, is at most $(\Delta + 3)n^2$.

Proof: By Lemma 7.6 the number of tokens in a node in the tokens protocol is bounded by $(\Delta + 3)n$. We claim that the number of tokens in a node plus the number of tokens on the incoming edges to a node in bounded by $(\Delta + 3)n$. This follows from the observation that the adversary, by delivering all the tokens on the incoming edges to a node, can make the number of tokens in a node equal to the number of tokens it had plus the number of tokens on the edges. Therefore the overall number in all the node is $(\Delta + 3)n^2$. □

Lemma 7.8 In any state S_v^t, the number of entries for different values of ℓ in the variables $\texttt{edges\_sent\_msg}[\ell]$, $\texttt{edges\_rec\_reply}[\ell]$, $\texttt{edges\_rec\_msg}[\ell]$, $\texttt{edges\_sent\_reply}[\ell]$, or $\texttt{latest}\_\ell = \ell$, is bounded by $2n$.

Proof: In any process, for any label ℓ, if one of the sets or variables corresponding to ℓ is not empty, there is an edge e in $\texttt{edges\_sent\_msg}[\ell]$ which is not in $\texttt{edges\_rec\_reply}[\ell]$, or an edge in $\texttt{edges\_rec\_msg}[\ell]$ which is not in $\texttt{edges\_sent\_reply}[\ell]$. Since in this case, a new message cannot be received on the edges on which replies were not sent, and new messages cannot be sent on edges on which replies have not yet been received, there is one edge at least corresponding to each non-empty entry, and the edges are different. The number of different possible entries is thus bounded by $2(n-1)$, twice the number of incident edges, which in addition to the one additional value in $\texttt{latest}\_\ell$ is less than $2n$. □

Theorem 7.9 The *labeling protocol* has property *Q3*.

Proof: It will suffice to prove that in any state S_s^t, the size of $\bar{\ell}$ is at most $(\Delta + 5)n^2$. By Lemma 7.7 the number of token in the node, is at most $(\Delta + 3)n^2$. By Lemma 7.8, each node has at most $2n$ different $\ell \in \bar{\ell}$. Therefore, the size of $\bar{\ell}$ is bounded by $(\Delta + 5)n^2$. □

Theorem 7.10 The *labeling protocol* has property *Q1*. □

Proof: Let it be shown that in a state S^t, if ℓ exists in some process or channel, then in S_s^t, $\ell \in \bar{\ell}$. Initially the claim holds. Assume inductively that the claim holds in any state prior to S^t. Since by the code, no process apart from the sender ever adds a variable entry or message of a *non-existing* label ℓ without priorly receiving a message containing ℓ, it will suffice to prove that the claim will hold in state S^{t_1}, $t < t_1$ following an event in which the sender deleted ℓ from $\bar{\ell}$.

Since in the state S^{t_2}, where t_2 is the latest time in which ℓ was not in $\bar{\ell}$ (there is such a last time since initially $\bar{\ell}$ is empty), ℓ was not by the induction hypothesis in any entry in a process or message on a channel in S^{t_2}.

Thus, by the main algorithm, any process v having value ℓ in the time interval $[t_2, t_1]$, must have received a MSG (ℓ, *value*) from some process u in a state following S^{t_2}. The *edges* field in any TOKEN (u, ℓ, *edges*)

created in a process u, contains all edges to processes to which it sent **MSG**$(\ell, value)$. It follows that the condition $\forall(u \rightarrow w) \in$ **used_edges**$[\ell], w \in$ **clean_nodes**$[\ell]$ holds at time t_1, only after a token has been received from every node that received a **MSG**$(\ell, value)$ a some time in the interval $[t_2, t_1]$.

Since a **TOKEN**$(u, \ell, edges)$ can be created in u only after all messages sent by it have been replied, all its outgoing channels do not contain messages with label ℓ. Recall that the creation event of the token removes all entries of ℓ. By Lemma 6.3 the message **MSG**$(\ell, value)$ is sent on each edge at most once in the time interval $[t_2, t_1]$, therefore each node v generates **TOKEN**$(\ell, \cdot, \cdot)$ at most once in the time interval $[t_2, t_1]$. If $\forall(u, w) \in$ **used_edges**$[\ell], u, w \in$ **clean_nodes**$[\ell]$ holds at time t_1, then ℓ does not exists is any process or channel. Therefore, the sender can delete ℓ from $\bar{\ell}$. □

The proof of the following Theorem 7.11 depends on the proof that the message complexity of the tokens protocol is bounded.

Theorem 7.11 The *labeling protocol* has property *Q4*.

Proof: Assume by way of contradiction that there is such a time t_0, in which the set of nodes for which **blocked** is *true* in every state S^t, $t \geq t_0$, forms a cut between the receiver and sender. By the code of the main protocol, the nodes with **blocked** = *true* do not send **MSG** and **TOKEN-ACK** messages. In any state, either **trying**=*true*, or since by *Q3 free_label_available*$(\bar{\ell})$ holds, **trying** will become *true*. Since the nodes with **blocked** = *true* form a cut between the receiver to the sender, from the first time in which **trying** becomes *true* after t_0, it will remain *true* forever, and eventually the sender will not send any new **MSG** messages.

Assume that no **new_token**$_v$ is generated by any node v. By Theorem 8.21 the message complexity is bounded as a function of n, therefore eventually there exists a time in which no more **TOKEN**s are sent. Let this time be t_1.

In any state of the sender, **tokens**$_s = 0$ always. Any neighbor v of s, has **estimate**$_v[s] = 0$. By the assumption, after time t_1, node v does not send any more tokens. ¿From the code of the label protocol, this implies that either **tokens**$_v$ − **estimate**$_v[s] \leq \Delta$ or **wait_token_ack**(v, s) is *true*. Since the edge (v, s) is eventually connected, eventually a **token_ack** will be returned to v, and **wait_token_ack**(v, s) will become *false*. Therefore, **tokens**$_v$ − **estimate**$_v[s] \leq \Delta$. Since **estimate**$_v[s] = 0$, **tokens**$_v \leq \Delta$.

By induction on the distance from the sender (in the eventually connected network), the above observation can be extended to show that a node that has an eventually connected path of length i to the sender, has **tokens** $\leq i\Delta$, after time t_1. Any node that that is not in the cut, and has an eventually connected path to the sender, that does not pass through the cut, has a path of distance at most $n - 2$ (since the receiver is not in the cut).

The nodes that have an eventually connected edge to a node that is in the cut, by the previous claim have **tokens** $\leq (n-2)\Delta$. Therefore, either new tokens are generated, or one of the nodes in the cut is in a state in which ¬**blocked**.

To complete the proof, it is sufficient to show that the number of times a token can be generated locally, while no token is received by the sender, is bounded by n^3. This will imply that one of the nodes in the cut changes its state from **blocked** to ¬**blocked**, contradicting the assumption about the nodes in the cut. The following paragraph is devoted to shows that the number of tokens generated, while no token is received by the sender, is bounded by n^3.

The number of tokens that can be created by a single **MSG** of label ℓ is at most n (one per process). Since the sender is in a **trying** state, it will not add any **MSG** with new label values in any state following time t_0. Since the number of **MSG** messages in the channels in state S^{t_0} is at most n^2, the number of new tokens added is bounded by n^3. □

8 The Complexity

The proof that the space, time and computation time of the protocol are polynomial follow immediately from the code of protocol, given that the communication complexity is polynomial. In this section we therefore prove that the communication complexity of the end-to-end protocol is polynomial.

To assist in the analysis, let the following two functions be introduced. The first is an energy function, $\mathcal{E}$, and the second is a potential function ϕ. It will be shown that each TOKEN message received, reduces the sum of $\mathcal{E} + 28\phi$ by n, and that the sum $\mathcal{E} + 28\phi$ is monotonically non-increasing in time, as long as no new token is generated, and no TOKEN is received by the sender. Since the sum is bounded from above and below by a polynomial in n, the number of TOKENs sent is polynomial. The number of UPDATEs is bounded by $2n$ times the number of TOKENs sent. Since by the argument used in the proof of Theorem 7.11, the number of times a new token can be generated is at most n^3, the entire scheme has a polynomial message complexity.

Definition 8.1 For a node v at time t, let $intransit_v^t$ be all the TOKEN messages sent by v before time t, and not receive by time t. Let $\mu_v^t = \mathtt{tokens}_v^t + \mathsf{intransit}_v^t$.

Definition 8.2

$$\mathcal{E}^t = (\sum_{v \in V} (\mu_v^t)^2) + 28(\sum_{(u,v) \in E} |\mathtt{unreported}_v^t[u]| + |\mathtt{UPDATE}_v^t[u]|)$$

Claim 8.3

$$0 \leq \mathcal{E}^t \leq 30(\Delta + 3 + n)^2 n^3 = O(\Delta^2 n^3)$$

Proof: The proof follows from the fact that each expression is non-negative, and by Lemma 7.6, is bounded from above by $(\Delta + 3 + n)n$. □

Definition 8.4 An *unreported interval* in a node v is a maximal time interval $[t_0, t_1]$, where t_0 is the latest time prior to t_1, in which $\mathtt{unreported}_v[e] \neq 0$.

Definition 8.5 An *update interval* of an UPDATE message sent from u to v at time t_1 and received at time t_2, is the time interval $[t_0, t_2]$, where $[t_0, t_1]$ is an unreported interval.

Claim 8.6 On a channel from u to v, an *update interval* intersects at most seven (7) *token intervals.*

Proof: In the subinterval $[t_0, t_1]$ of the *update interval* from u to v, $\mathtt{unreported}_u[e] \neq 0$. If during this interval, $\mathtt{wait\_update\_ack}_u = \mathit{false}$, then a TOKEN could not have been sent by u. If $\mathtt{wait\_update\_ack}_u = \mathit{true}$, then an UPDATE message must be in transit from u to v, or an UPDATE_ACK message is in transit from v to u. Since the receive event of the UPDATE message, includes a sending of the UPDATE_ACK, a TOKEN sent by u in the interval $[t_0, t_1]$, would have its corresponding TOKEN_ACK received by u in the interval $[t_1, t_2]$. A TOKEN sent following this one (i.e. in $[t_1, t_2]$), must by Claim 7.3 be sent after the TOKEN_ACK of the first was received, and would have its TOKEN_ACK received later than t_2. Since before the TOKEN sent by u during $[t_0, t_1]$, at most one TOKEN could have been sent and received prior to t_0, at most three *token intervals* overlapped $[t_0, t_2]$ for TOKENs sent from u to v.

In the subinterval $[t_0, t_1]$ of the *update interval* from u to v, $\mathtt{unreported}_u[e] \neq 0$. If during this interval, $\mathtt{wait\_update\_ack}_u = \mathit{false}$, then a TOKEN_ACK could not have been sent by u. If $\mathtt{wait\_update\_ack}_u = \mathit{true}$, then an UPDATE message must be in transit from u to v, or an UPDATE_ACK message is in transit from v to u. Let α be the first TOKEN message sent from v to u at time $t_3 > t_0$. Let t_4 be the time that

the TOKEN_ACK for α was received at v. If at time t_0 there was an UPDATE in transit between u and v, it must have been received at v by time $t_5 < t_4$. Since the receive event of the UPDATE message, includes a sending of the UPDATE_ACK, a message UPDATE_ACK must have been sent to u before t_4. Let β be the first TOKEN message sent from v to u at time $t_5 > t_4$ and t_6 the time at which it was received at v. Since the UPDATE_ACK was sent before t_5, it will be received at u at time $t_7 < t_6$. At time t_7, wait_update_ack= *false*. Therefore, at time t_6, when β is received at u, either an UPDATE message was sent between t_6 and t_7, or wait_update_ack= *false*. In the latter case an UPDATE message is sent to v at time t_6. If t_8 is the time at which the TOKEN_ACK for β was received at v, then the UPDATE was received before t_8. Since the receive event of the UPDATE message, includes a sending of the UPDATE_ACK, a message UPDATE_ACK will be sent to u before t_8. Finally, a token γ sent after t_8 will be received by u after the UPDATE_ACK. Since before the TOKEN α, at most one additional TOKEN could have been sent and received prior to t_0, at most four *token intervals* overlapped $[t_0, t_2]$ for TOKENs sent from v to u. Thus, at most seven *token intervals* could have intersected $[t_0, t_2]$ in both directions combined. □

By the same arguments, the following claim is also true.

Claim 8.7 On a channel from u to v, an *unreported interval* intersects at most seven (7) *token intervals.*

We define a potential function ρ. The main purpose of this potential function is to enable to amortize in a given state, over events that will happen in the future.

Definition 8.8 For every TOKEN message α, define an potential function $\rho(\alpha)$, and let it change in the following way:

1. At the time when α is sent, t, $\rho^t(\alpha)$ is decremented by n.
2. For an update interval that intersects the token interval of α, at the time t that the message UPDATE(x) was received, $\rho(\alpha)$ is incremented to $\max\{\rho(\alpha) + |x|/7, 2\Delta n\}$.
3. At time t in an *unreported interval* (either in u or v) that intersects the token interval of α, such that at time t, either at u or v a token was sent or received and $|\text{unreported}[e]|$ was reduced by one, $\rho^t(\alpha)$ is incremented to $\max\{\rho(\alpha) + 1/7, 2\Delta n\}$.
4. Let t be the time that α is received, and t' the time just before t. Then $\rho^t(\alpha)$ is set to $\rho^{t'} - 1/14(\Delta - n - (\mu_v^t - \mu_u^t)) + n$.

Claim 8.9 $|\rho^t(\alpha)| \leq O(\Delta n)$ □

The following two lemmi provide a lower bound on the increase in ρ with respect to UPDATE messages and changes in **unreported**.

Lemma 8.10 Let α be a TOKEN message sent from v to u. Consider an *unreported interval* $[t_0, t_1]$ in v (u resp.), that is not a part of an update interval and intersects the token interval of α. Let K be the maximum value of $|\text{unreported}_v[e]|$ ($|\text{unreported}_u[e]|$ resp.) in this interval. Then the sum of the increases of $\rho(\alpha)$ in this interval is at least $K/7$.

Proof: Consider the time t' such that $|\text{unreported}_v[e]| = K$. At time t_1, by definition $\text{unreported}_v[e] = 0$. Between t' and t_1 at least K times there was a decrease in $|\text{unreported}_v[e]|$. Each such decrease by definition contributes $1/7$. □

Lemma 8.11 Let α be a TOKEN message sent from v to u. Consider an *update interval* $[t_0, t_2]$ in v (resp. u) that intersects the token interval of α. Let t_1 be the time at which UPDATE was sent. Let K be the maximum value of $|\mathbf{unreported}_v[e]|$ ($|\mathbf{unreported}_u[e]|$ resp.) in $[t_0, t_1]$. Then the sum of the increases of $\rho(\alpha)$ in this interval is at least $K/7$.

Proof: Consider the time t' such that $|\mathbf{unreported}_v[e]| = K$. At time t_1 the value of $\mathbf{unreported}_v[e]$ was x. Between t' and t_1 at least $|K| - |x|$ times there was a decrease in $|\mathbf{unreported}_v[e]|$. Each such decrease by definition contributes $1/7$. At time t_1 the message UPDATE(x) was sent. At time t_3, when the message was received at u, $\rho(\alpha)$ increased by $|x|/7$. Thus, the sum of the increases is $K/7$. □

The following lemma provides a lower bound on the total increase in ρ as a function of the final difference between the number of tokens in the two end processors.

Lemma 8.12 Let t_0 be the time that TOKEN α was sent from v to u, and t_1 the time it was received. The sum of all increases of $\rho(\alpha)$ (over all time) is at least $[\Delta - (\mu_v^{t_1} - \mu_u^{t_1}) - n]/14$.

Proof:

Let x_1 be the value of the UPDATE that crosses α from u to v and x_2 be the value of the UPDATE that crosses α from v to u.

Let η be $\mathbf{estimate}_v^{t_0}(u)$, and τ_0 the value of $\mathbf{tokens}_v^{t_0}$, and ν_0 the value of $\mathbf{unreported}_v^{t_0}[e]$. By the code, $\tau_0 - \eta \geq \Delta$.

Let τ_1 the value of $\mathbf{tokens}_v^{t_1}$, ν_1 the value of $\mathbf{unreported}_v^{t_1}[e]$, and ξ_1 the value of $intransit_v^{t_1}$. This means that $\mu_v^t = \tau_1 + \xi_1$. By Lemma 7.5, the value of τ_1 (i.e. $\mathbf{tokens}_v^t$) is equal to $\tau_0 + x_2 + (\nu_1 - \nu_0)$.

Let τ_2 the value of $\mathbf{tokens}_u^{t_1}$, ν_2 the value of $\mathbf{unreported}_u^{t_1}[e]$, and ξ_2 the value of $intransit_u^{t_1}$. This means that $\mu_u^{t_1} = \tau_2 + \xi_2$. The value τ_2 (i.e. $\mathbf{tokens}_u^{t_1}$) is equal to $\eta + x_1 + \nu_2$.

$$\mu_v^{t_1} - \mu_u^{t_1} = [(\tau_0 + x_2 + \nu_1 - \nu_0) + \xi_1] - [(\eta + x_1 + \nu_2) + \xi_2]$$

Which we can rewrite as,

$$(\tau_0 - \eta) - (\mu_v^{t_1} - \mu_u^{t_1}) = [(x_1 + \nu_2) + \xi_2] - [(x_2 + \nu_1 - \nu_0) + \xi_1]$$

Since $\xi_2 \leq n - 1$ and $\xi_1 \geq 0$, $|\xi_2 - \xi_1| \leq n$,

$$(\tau_0 - \eta) - (\mu_v^{t_1} - \mu_u^{t_1}) - n \leq |\nu_0| + |x_2| + |\nu_1| + |x_1| + |\nu_2|$$

Recall that $\Delta \leq \tau_0 - \eta$, hence,

$$2(\Delta - (\mu_v^{t_1} - \mu_u^{t_1}) - n) \leq \max\{|\nu_0|, |x_2|\} + |\nu_1| + |x_1| + |\nu_2|$$

Since every term in the sum is identified with a distinct *update* or *unreported interval*. By Lemma 8.11 and Lemma 8.10, each term contributes to $\rho(\alpha)$ one seventh of the maximum value. This implies that the lemma follows. □

Definition 8.13 Let T_e^t be the set of tokens, α, sent on edge e, such that either t is in the token interval of α, or at time t, there is an update or an unreported interval that intersect the token interval of α. Denote by $T^t = \cup_{e \in E} T_e^t$. Let,

$$\phi^t = \sum_{\alpha \in T^t} \rho^t(\alpha)$$

Claim 8.14 $|\phi^{t_1}| = O(\Delta n^3)$

Proof: By Claim 8.9, $|\rho^t(\alpha)| = O(n\Delta)$, and by Claim 8.6, the size of T^t is bounded by $7n^2$. □

Lemma 8.15 Let α be a **TOKEN** message for which the token interval ends before time t, and $\alpha \notin T_e^t$, then for any $t' > t$, $\rho^t(\alpha) = \rho^{t'}(\alpha)$.

Proof: Since α was already received at time t, $\rho(\alpha)$ can not decrease after time t. The fact that it can not increase after time t is immediate from the definition of the increases. □

The following lemma shows that the value of each $\rho(\alpha)$ becomes eventually non-negative, and therefore dropping them from the sum in ϕ can not increase the value of ϕ.

Lemma 8.16 Let α be a **TOKEN** message for which the token interval ends before time t, and $\alpha \notin T_e^t$, then $\rho^t(\alpha) \geq 0$.

Proof: By Lemma 8.15 after time t, $\rho(\alpha)$ does not change. The function $\rho(\alpha)$ can be decreased at most twice. When α is sent, $\rho(\alpha)$ is decreased by $1/28n$. The value of $\rho(\alpha)$ is decreased at the receipt by $1/14(\Delta - n - (\mu_v^t - \mu_u^t)) - n$. By Lemma 8.12, the sum of the increases is at least $[\Delta - (\mu_v^{t_1} - \mu_u^{t_1}) - n]/14$. Therefore, $\rho(\alpha) \geq 0$. □

The following lemma shows that the sum $\mathcal{E} + 28\phi^t$ decreases by at least n, after each receive of a **TOKEN** message. This implies that the number of such messages can be bounded by $[\mathcal{E} + 28\phi^t]/n$.

Lemma 8.17 Let α be a **TOKEN** message receive from v to u at time t. Let t', $t < t'$ be the time immediately after the receive event of α. Then

$$(\mathcal{E}^t + 28\phi^t) - (\mathcal{E}^{t'} + 28\phi^{t'}) \geq n$$

Proof: At the receipt of α, μ_v^t is decremented by one, and μ_u^t is incremented by one. This implies that $(\mu_v^t)^2 + (\mu_u^t)^2$ is changed to $(\mu_v^t - 1)^2 + (\mu_u^t + 1)^2$, hence the difference is $2\mu_v^t - 2\mu_u^t - 2$. Since the increases in **unreported** is bounded by n, $\mathcal{E}^t - \mathcal{E}^{t'} \geq 2\mu_v^t - 2\mu_u^t - 2 - 28n$.

The value of ϕ^t is changed to $\phi^t - 1/14(\Delta - n - (\mu_v^t - \mu_u^t)) + n$. Therefore $28\phi^t - 28\phi^{t'} = 2\Delta - 2n - 2(\mu_v^t - \mu_u^t) - 28n$. Hence, the sum of the two is $(\mathcal{E}^t + 28\phi^t) - (\mathcal{E}^{t'} + 28\phi^{t'}) \geq 2\Delta - 58n - 2 \geq n$. For $\Delta \geq 30n$, this holds. □

The following lemma establishes the invariant that $\mathcal{E}^t + 28\phi^t$ is monotonically non increasing in time. This fact, combined with Lemma 8.17, will establish the polynomial convergence of the algorithm.

Lemma 8.18 Let S^{t_0} and S^{t_1} be two states, such that $t_0 < t_1$, then $\mathcal{E}^{t_0} + 28\phi^{t_0} \geq \mathcal{E}^{t_1} + 28\phi^{t_1}$.

Proof: The only events that can affect the value of $\mathcal{E} + 28\phi$, are the sending of a **TOKEN** message, the receipt of a **TOKEN** message, the sending of an **UPDATE** message, or the receipt of an **UPDATE** message.

When a receive token event occurs, by Lemma 8.17, the sum $\mathcal{E} + 28\phi$ is decremented by n.

When a **TOKEN** message, α, is sent from v, the value of μ_v does not change. The **unreported** variables that are updated, contribute at most $28n$ to $\mathcal{E}$. Since $\rho(\alpha)$ is decremented by n, ϕ is decremented by n, and therefore the sum $\mathcal{E} + 28\phi$ can only decrease in this case.

When an **UPDATE**(x) message is sent, the value of $\mathcal{E}$ does not change. The increase in |**UPDATE**| is equal to the decrease in |**unreported**|. The value of ϕ clearly remains unchanged.

When an **UPDATE**(x) message is received, the value of $\mathcal{E}$ is decremented by $28|x|$. By Lemma 8.6, an update interval can intersect at most seven token intervals. Each of them will increase ϕ by $|x|/7$. Therefore the increase in ϕ is bounded by $|x|$. □

Lemma 8.19 As long as no new token is generated locally and no **TOKEN** is received at the sender, the number of **TOKEN** messages sent in the *labeling protocol*, is bounded by $O(\Delta^2 n^2)$, and the number of **UPDATE** messages is bounded by $O(\Delta^2 n^3)$.

Proof: At any time $|\mathcal{E}^t + 28\phi^t| = O(\Delta^2 n^3)$, by Claims 8.3 and 8.14. By Lemma 8.18 and 8.17, the maximum number number of tokens transmitted is bounded by $O(\Delta^2 n^2)$. The number of UPDATE messages is bounded by $O(\Delta^2 n^3)$. □

In order to show that the algorithm is polynomial it would have been sufficient to multiply the above complexity by n^3, the number of tokens that can be generated, without any TOKEN returned to the sender. In the rest of the complexity analysis we show how to analyze the influence of the tokens that are generated, without a great penalty in the message complexity. The main idea is to analyze the influence of creating a TOKEN on the energy function.

Lemma 8.20 If k new TOKENs are generated and no TOKEN is received at the sender, the number of TOKEN messages sent in the *labeling protocol*, is bounded by $O(\Delta^2 n^3 + k\Delta^2 n)$, and the number of UPDATE messages is bounded by $O(\Delta^2 n^3 + k\Delta^2 n^2)$.

Proof: Each token that is generated increases $\mathcal{E} + 28\phi$ by at most $\Delta^2 n^2$. The total increase is bounded by $k\Delta^2 n^2$. Therefore, the number of tokens is bounded by $O(\Delta^2 n^2 + k\Delta^2 n)$, and the UPDATE messages by $O(\Delta^2 n^3 + k\Delta^2 n^2)$. □

Theorem 8.21 The communication complexity of the end-to-end protocol is $O(n^9)$ bits.

Proof: The complexity of the main protocol is $O(n^4)$ bits per data item transmitted. Since the number of TOKENs generated locally by the labeling algorithm is bounded by $O(n^3)$, and since each time stamp label ℓ is of $O(n^3)$ bits, and each Update message is of $O(\log n)$ bits, the communication complexity of the protocol is by Lemma 8.20 $O(n^9)$ bits. □

9 Acknowledgements

We wish to thank Yehuda Afek and Adi Rosen for their helpful comments, especially in pointing out an error in an earlier version of the complexity counting arguments.

References

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

[AAM89] Yehuda Afek, Baruch Awerbuch, and Hezi Moriel. Overhead of resetting a communication protocol is independent of the size of the network. May 1989. Unpublished manuscript.

[AE83] Baruch Awerbuch and Shimon Even. *A Formal Approach to a Communication-Network Protocol; Broadcast as a Case Study.* Technical Report TR-459, Electrical Engineering Department, Technion-I.I.T., Haifa, December 1983.

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

[AG88] Yehuda Afek and Eli Gafni. End-to-end communication in unreliable networks. In *Proceedings of the 7^{th} Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada*, pages 131–148, ACM SIGACT and SIGOPS, ACM, 1988.

[AGH89] Baruch Awerbuch, Oded Goldreich, and Amir Herzberg. A quantitative approach to dynamic networks. May 1989. Unpublished manuscript.

[AM88] Baruch Awerbuch and Yishay Mansour. Efficient topology update algorithms. 1988. Unpublished manuscript.

[AMS89] Baruch Awerbuch, Yishay Mansour, and Nir Shavit. Polynomial end-to-end communication. In 30^{th} *Annual Symposium on Foundations of Computer Science*, Comp. Soc. of the IEEE, IEEE, 1989.

[AO87] Ravindra K. Ahuja and James B. Orlin. *A Fast and Simple Algorithm for the Maximum Flow Problem.* Working Paper 1905-87, MIT Sloan School of Management, June 1987.

[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.

[BS88] Baratz and Segall. Reliable link initialization procedures. *IEEE Trans. Comm.*, February 1988.

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

[DS87] D. Dolev and N. Shavit. A note on bounded time-stamp systems. July 1987. Unpublished manuscript.

[DS89] D. Dolev and N. Shavit. Bounded concurrent time-stamp systems are constructible. In *Proceedings of the* 21^{st} *Annual ACM Symposium on Theory of Computing, Seattle, Washington*, ACM SIGACT, ACM, 1989.

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

[Gal76] Robert G. Gallager. *A Shortest path Routing Algorithm with Automatic Resynch.* Technical Report , MIT Lab. for Information and Decision Systems, March 1976.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *J. ACM*, 35(4):921–940, October 1988. Also appeared in 18th STOC (1986).

[IL87] A. Israeli and M. Li. Bounded time stamps. In 28^{th} *Annual Symposium on Foundations of Computer Science, White Plains, New York*, pages 371–362, IEEE, 1987.

[Lam86] Leslie Lamport. The mutual exclusion problem.part ii: statement and solutions. *J. ACM*, 33(2):327–348, 1986.

[LMF88] Nancy A. Lynch, Yishay Mansour, and Alan Fekete. The data link layer: two impossibility results. In *Proceedings of the* 7^{th} *Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada*, pages 149–170, ACM SIGACT and SIGOPS, ACM, Toronto, Canada, 1988. Also, Technical Memo MIT/LCS/TM-355, May 1988.

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

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

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

OFFICIAL DISTRIBUTION LIST

Director Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 copies
Office of Naval Research 800 North Quincy Street Arlington, VA 22217 Attn: Dr. R. Grafton, Code 433	2 copies
Director, Code 2627 Naval Research Laboratory Washington, DC 20375	6 copies
Defense Technical Information Center Cameron Station Alexandria, VA 22314	12 copies
National Science Foundation Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 copies
Dr. E.B. Royce, Code 38 Head, Research Department Naval Weapons Center China Lake, CA 93555	1 copy