Early Detection of Message Forwarding Faults

Amir Herzberg
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
E-mail: amir@watson.ibm.com

Shay Kutten
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
E-mail: kutten@watson.ibm.com

January 3, 1995

Abstract

In most communication networks, pairs of processors communicate by sending messages over a path connecting them. We present communication-efficient protocols that quickly detect and locate any failure along the path. Whenever there is excessive delay in forwarding messages along the path, the protocols detect a failure (even when the delay is caused by maliciously-programmed processors). The protocols ensure optimal time for either message delivery or failure detection.

We observe that the actual delivery time $\delta$ of a message over a link is usually much smaller than the apriori known upper bound $D$ on that delivery time. The main contribution of the paper is the way to model and take advantage of this observation. We introduce the notion of asynchronously early determinating protocols, as well as protocols that are asynchronously early terminating, i.e., time optimal in both worse case and typical cases. More precisely, we present a time complexity measure according to which one evaluates protocols both in terms of $D$ and $\delta$. We observe that asynchronously early termination is a form of competitiveness.

The protocols presented here are asynchronously early terminating since they are time optimal both in terms of $D$ and of $\delta$. Previous communication efficient solutions were slow in the case where $\delta \ll D$. We observe that this is the most typical case.

*Preliminary reports of parts of the work reported here appeared in the proceedings of the ICCC 88 and the PODC 89 conferences.

1Partially supported by grant No. 86-00301 from the United States-Israel Binational Science Foundation (BSF). Part of the research was done while this author was with the Technion—Israel Institute of Technology.
It is suggested that the time complexity measure introduced, as well as the notion of asynchronously early terminating can be useful when evaluating protocols for other tasks in communication networks. The model introduced can be a useful step towards a formal analysis of real-time systems.

Our protocols have $O(n \log n)$ worst case communication complexity. We show that this is the best possible for protocols that send immediately any acknowledgement they ever send. Then we show an early terminating protocol which uses timing and delay to reduce the communication complexity in the typical executions where the number of failures is small and $\delta \ll D$. In such executions its message complexity is linear, as is the complexity of nonfault tolerant protocols.
1 Introduction

In this paper, we introduce a complexity measure of time complexity for asynchronous networks for which there exists an upper bound on the delivery time of a message over a link. It is suggested that this can be a useful step toward improving the analysis of actual communication networks, as well as a step toward the formal analysis of real-time systems. Using this complexity measure, we develop optimal protocols for dealing with the task of managing end-to-end communication sessions.

The end-to-end delivery of information from source to destination is a basic communication task. The most communication-complexity-efficient method to deliver information is to send it along a fixed (short) path between two processors. There are also some other reasons that make this method the most common (e.g., used in [Tan81, BGF+85, MRR80]). For example, First-In First-Out service for messages is then guaranteed, without the need for expensive hardware or for software intervention for restoring the order of the messages.

Of course, this requires that all the links and the processors along the path are operational. When a link or a processor fails (which does not happen very often, compared to the rate of message traffic), a different fixed path is established instead of the disconnected one. For that, additional mechanisms are used to detect and locate failures. These mechanisms are based on acknowledgements and use a time-out value $D$, which is a bound on the transmission delay over one link. Often fail-stop failures are detected by hop-by-hop acknowledgements, which are control messages sent by a processor to its neighbor upon receiving a data message from that neighbor. If processor $u$ does not receive an acknowledgement from neighbor $v$ within $D$ since $u$ sent a data message to $v$, then either $v$ or the link $(u, v)$ failed.

There are some failures which are not detected by the hop-by-hop acknowledgements mechanism. A simple example is a malicious failure, where $v$ `intentionally' sends an acknowledgement without forwarding the data message towards the destination. A similar outcome may result without malicious intent, i.e., when a processor breaks down after sending the acknowledgement but before succeeding in forwarding the data message. (A more likely case is that a processor did forward the message before it failed, but the message got lost over the link; the failed processor cannot now retransmit the message). This kind of failure is usually dealt with by an end-to-end acknowledgement mechanism. This is an acknowledgement message which is sent from the destination to the source when the destination receives a data message. If the source does not receive an acknowledgement within $2(n-1)D$ since it sent the data message, where $n$ is the number of processors along the path, then a failure has occurred.

For a given execution of the protocol, and for any $f$, let $\delta_f$ denote the maximal transmission delay over a link in this execution, when not counting the worst $f$ links (i.e., the $f$ links whose maximum $\delta_f \leq D$ is unknown initially. Moreover, $\delta$ is not any bound (even unknown), but rather the actual maximal delay as could be observed had there been some outside observer. Intuitively, when $f$ is the number of faults in an execution,
\( \delta_f \) is the maximum delay over nonfaulty links (and between two nonfaulty neighboring processors). However, this definition makes sense also when there are no “real faults” (or at least no faults can be detected, since the delay on all links is still smaller than \( D \)), just some links happened to be slower than others. For simplicity, we prefer to use \( \delta_f \) rather than the actual delay (in the execution) over each link separately. In addition, since our protocols can deal even with malicious faults of processors, most of the paper uses this meaning of \( f \). Thus, \( f \) is clear from the context and we shall use the notation \( \delta \) instead of \( \delta_f \). In Section 7, we analyze briefly the meaning of the results in the case where no faults occur.

Network designers usually choose a bound \( D \) which is much larger than the typical transmission delay. This is due to the unpredictability of the actual delay, and to the huge overhead of disconnecting a link (see, e.g., [GSK87]). Therefore, in typical executions, \( \delta \ll D \) holds. This motivates an analysis of the time complexity which considers both \( D \) and \( \delta \). Intuitively, one would like to derive a bound on the time complexity that will depend, as much as possible, on the (usually small) value of \( \delta \), rather than on the value of \( D \).

We call a protocol *asynchronously early terminating* if its time complexity is optimal for any selection of the number of faults \( f \), of \( D \) and of \( \delta \). (We shorten it to *early terminating*, when there is no ambiguity with the “early stopping” synchronous Byzantine agreement protocols discussed in many papers [DRS].) We present early terminating detection protocols, with time complexity \( O(fD + n\delta) \), where \( f \) is the number of faults. This improves substantially over the common end-to-end mechanism, which requires \( 2(n-1)D \), for typical executions where \( f \ll n \) and \( \delta \ll D \). Typical values of the parameters \( n, f, D \) and \( \delta \) are shown in Table 1.
Table 1: Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Typical value</th>
<th>Initially known?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>Number of processors</td>
<td>20</td>
<td>Yes</td>
</tr>
<tr>
<td>$f$</td>
<td>Number of faulty processors and links</td>
<td>0 or 1</td>
<td>No</td>
</tr>
<tr>
<td>$D$</td>
<td>Bound on the delay on a nonfaulty link</td>
<td>1 sec.</td>
<td>Yes</td>
</tr>
<tr>
<td>$\delta$</td>
<td>Actual delay on a nonfaulty link</td>
<td>10 msec.</td>
<td>No</td>
</tr>
</tbody>
</table>

Another way to see the improvement is when considering the competitiveness of the protocols. Consider the time complexity $T_C(P)$ of a protocol $P$ for a given configuration $C$, that is, given the set $F$ of faulty processors (where $|F| = f$) and the values of $D$ and $\delta$. Define the competitive ratio of a protocol $P_1$ with respect to protocol $P_2$ and the given configuration as $\frac{T_C(P_1)}{T_C(P_2)}$. The competitive ratio of $P_1$ with respect to $P_2$ is $\max C \frac{T_C(P_1)}{T_C(P_2)}$. Clearly, the competitive ratio of the common end-to-end mechanism with respect to our protocols is $\Omega(n)$, where the maximum ratio is achieved then $\delta$ approaches zero.

To compute the distributed competitiveness of a protocol $P_1$ one needs to compare it to the “best” distributed algorithm, rather than to any other. To rephrase the recent definition of [DRS], for a given configuration $C$, one divides $T_C(P_1)$ by the complexity (for configuration $C$) of a distributed protocol $OPT_C$ that achieves the best time complexity for configuration $C$. It is required that $OPT_C$ will perform correctly in any case. The distributed competitiveness of $P_1$ is the maximum (over all the configurations) of such ratio. It can be shown that early terminating protocols for our problem are distributively competitive (i.e., have constant competitive ratio). Thus, our protocols are also distributively competitive.

The protocols presented in this paper allow early terminating and communication-efficient detection of arbitrary faults, while forwarding information from the source to the destination. Our protocols also provide fault location, i.e., when a failure occurs, the source learns of a specific link such that either the link or one of its endpoints is faulty. This is useful for most recovery actions. Both detection and location are done in optimal time for any value of $D$ and $\delta$ (recall that we analyze the time complexity as a function of $D$ and $\delta$).

Table 2 summarizes our results and previous results. Note that all of our protocols provide fault location, which was achieved previously only with $O(n^2)$ communication complexity.
Table 2: Protocols

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Time</th>
<th>Communication</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO 8072/3, CCITT x.224</td>
<td>$O(nD)$</td>
<td>$O(n)$</td>
<td>Not locating</td>
</tr>
<tr>
<td>[Per88, in §3.5]</td>
<td>$O(nD)$</td>
<td>$O(n^2)$</td>
<td>Locating</td>
</tr>
<tr>
<td>End-to-End §4.1</td>
<td>$O(nD)$</td>
<td>$O(n)$</td>
<td>Locating</td>
</tr>
<tr>
<td>Hop-by-Hop §4.2</td>
<td>$O(fD + n\delta)$</td>
<td>$O(n^2)$</td>
<td>Locating</td>
</tr>
<tr>
<td>Immediate Ack §6.1</td>
<td>$O(fD + n\delta)$</td>
<td>$O(n\log(n))$</td>
<td>Locating</td>
</tr>
<tr>
<td>Adaptive §6</td>
<td>$O(fD + n\delta)$</td>
<td>$O(n\log(f + \frac{n\delta}{D}))$</td>
<td>Locating</td>
</tr>
</tbody>
</table>

Our main contribution is the measure of time complexity as depending on $D$ and $\delta$, with the concept that one should strive to obtain time complexity bounds that depend as much as possible on $\delta$ rather than on $D$. Other contributions are communication efficient early terminating protocols. The Immediate Ack protocol, presented in §6.1, has $O(n\log n)$ communication complexity. We present in §6 an early terminating protocol which is adaptive, in the sense that the number of acknowledgement messages sent depends on the delays and the behavior of the adversary in the particular execution. Its communication complexity is $O(n\log(2 + f + \frac{n\delta}{D}))$. We argue that in typical applications of this type of protocol, the number of failures $f$ is zero or very small; otherwise the network designer will avoid transmission of messages over a fixed path. The term $\frac{n\delta}{D}$ is also usually small. Hence, in practice, this communication complexity is close to the optimal communication complexity ($O(n)$).

We show that for other kinds of protocols (i.e., those that are message-driven, except for the decision to disconnect a link that may be time-driven), a lower bound of $\Omega(n\log n)$ exists. The proof of the lower bound shows collections of paths of total length $\Omega(n\log n)$ that any protocol must use for sending messages (acknowledgements). Our adaptive protocol mentioned economizes on message-sending by avoiding using some of these paths when the delays and number of faults are small. However, there are some executions in which this protocol must utilize all the paths in its collections. It is an open problem whether there exists a protocol whose worst case message complexity is better (in order of magnitude). An interesting corollary from the proof of the lower bound is that such a protocol (if exists) will not have an execution that sends messages over every path that is used in some execution.

Related Works

The asynchronous model with bounded delay was previously studied in [AE86, CCGZ88, DHSS84] without considering early termination.

Communication via a fixed path was studied in [SJ86]. Detection of failures was addressed in accepted and proposed standards [Sta87]. Recovery from (detected) errors during such communication was studied in [Gro82]. Detection, location and recovery
from arbitrary transmission failures were studied by Perlman [Per88] who introduced the notions of communication failure detection in the environment of malicious processors, and that of failure location in this context.

Many works presented end-to-end communication protocols which do not depend on a single path [CR87], [Fin79], [Per88], [AG88], [AMS89], [AGH90]. The goal of these works is to increase reliability. In the extreme, these works achieve communication even if there is no moment when there is a nonfaulty path from the source to the destination [AE86, AG88, AMS89]. These methods are useful for source-to-destination communication only in applications where the increased reliability compensates for the much higher communication complexity, storage requirements and local processing.

We permitted the processors to fail in an arbitrary manner. However, we assumed that the links are 'well-behaved'; namely, the links either work correctly or their failure is detected. The justification for this assumption is the known link protocols [BS88, GHM89, Zim80].

Our adaptive protocol (§6) is based on the idea that a processor $i$ can ‘piggyback’ its acknowledgement on an acknowledgement of another processor $j$. This raises the question: How long can processor $i$ wait for the acknowledgement of processor $j$ before it gives up the idea of piggybacking (and sends its acknowledgement)? A similar problem was studied in [AAPAS78, AS87, BYKWZ87, K88].

Finally, this work should be viewed in the context of the important work published later dealing with formal approach and modeling of distributed real time systems. A paper with a strong impact is [ADLS91] that suggests a more detailed model, and an algorithm for agreement whose time depends mostly on $\delta$, and only minimally on $D$. This work was extended by [Pon91] to handle omission and Byzantine faults. In [AL89] formal analysis of timing uncertainties and time bounds is done in respect with another task.

Organization

In the next two sections, we define the communication model and the problem. In section 4, we present two natural, simple protocols that solve the problem. The two protocols are presented mainly in order to demonstrate the problem and the model. One (that is similar to the x.244 protocol [Sta87] is communication-optimal, but is not early-terminating. (The difference between this protocol and that of x.244 is that our protocol also locates the faults.) The other is early-terminating, but has high communication complexity. This protocol is similar to that of [Per88]. In section 5, we present a high level design of a fault isolator. This design is implemented by all the protocols in this paper, and the protocols previously published. In section 6.1, we give an implementation which is early-terminating and with message complexity which is $O(n \log n)$. This message complexity is achieved by optimizing a certain combinatorial cover problem introduced in that section. We also show that every early-terminating protocol sends messages over paths that are included in such a cover. In section 6, we present an early-stopping protocol that economizes on messages by avoiding sending messages over some of these paths in favorable executions. We conclude and discuss open problems in section 7.
2 The Model

Our model is a modification of the standard model of dynamic networks [AES6, AAG87]. Since we are interested in detecting failures, we do not include recoveries. We assume some synchronization, namely a known bound $D$ on the transmission time over a nonfaulty link. We also introduce some new notations and assumptions, since we discuss communication only along a fixed path.

Denote the path as processors $1, \ldots, n$. Even though the task of our protocols is to deal with a message from processor 1 to processor $n$, our protocols send additional messages from intermediate nodes, and to intermediate nodes. Consider a message $\Phi$ (e.g., an acknowledgement) that was sent by processor $1 \leq j \leq n$ (the sender of $\Phi$) to another processor $1 \leq k \leq j$ (the recipient of $\Phi$). If $k$ is not a neighbor of $j$, then message $\Phi$ needs to be received and resent by processors on the path between $j$ and $k$. We use the following (somewhat “visual”) notation (to emphasize that in this case $k < j$): The protocol in processor $i(k \leq i \leq j)$ interacts with the links by the events $\text{Send}_{i}^{k \rightarrow j}[\phi]$, and $\text{Receive}_{i}^{k \rightarrow j}[\phi]$. Similarly, in the case that $k$ is larger than $j$: $\text{Send}_{i}^{j \rightarrow k}[\phi]$, $\text{Receive}_{i}^{j \rightarrow k}[\phi]$. The events have their natural meanings.

In actual networks, following a message sent by $i$ to $i + 1$, the lower layer link level protocol will deliver a fail$_{i+1}$ event to $i$ (i.e., the failure of the link from $i$ to $i + 1$), when more than $2D$ time passed without an acknowledgement whose sender is processor $i + 1$ being received at $i$. For simplicity, we do not use such an event. Our protocol will detect a fault in such a case anyhow. We do not distinguish here between faults of links and those of processors except that when $i$ detects a fault in the communication with $i + 1$, we call it a failure of link $(i, i + 1)$ (although it may be just the fault of processor $i + 1$), and similarly in the case when $i + 2$ detects a fault in its communication with processor $i + 1$, we speak of a failure of link $(i + 1, i + 2)$. Since we do not allow recoveries, we assume that each link fails at most once.

We assume that whenever processor $i$ receives a message, the message was indeed issued by the sender and later forwarded by every processor between the sender and $i$, and moreover, that processor $i$ is between the sender and the recipient. This assumption holds if the failures are nonmalicious, and otherwise can be enforced by cryptographic techniques.

**Axiom 1** If Receive$_{i}^{j \rightarrow k}[\phi]$ occurs, then $j < i \leq k$ and for every $p$ between $j$ and $i$, previously Receive$_{p}^{j \rightarrow k}[\phi]$ occurred. Similarly, if Receive$_{i}^{i \rightarrow k}[\phi]$ occurs, then $j \leq i < k$ and for every $q$ between $i$ and $k$, previously Receive$_{q}^{i \rightarrow k}[\phi]$ occurred.

The major deviation of our model from the standard dynamic network model is the addition of synchronization assumptions. Intuitively, these assumptions imply that the lower layer guarantees that it takes at most $D$ time units from a Send$_{i}^{j \rightarrow k}[\phi]$ (Send$_{i}^{j \rightarrow k}[\phi]$) event till the corresponding Receive$_{i+1}^{j \rightarrow k}[\phi]$ (respectively, Receive$_{i-1}^{j \rightarrow k}[\phi]$) unless the link
(or one of the processors) failed. For simplicity of exposition, we use global time terminology and assume that all of the clocks have the same rate. We model the clocks by an “alarm clock” that generates an event \( tD \) time after it is started by any \( t \) until it is stopped. Moreover, each processor has an unbounded number of such clocks, and every time it starts a clock it starts a different clock. This strong assumption is used just for convenience; clearly it is possible to use just one clock. Moreover, a “ticker” that sends interrupts every \( D \) time actually suffices for our protocols. A little more formally:

**Axiom 2** For every processor \( i \), a \( \text{TICK}_i \) event occurs exactly every \( D \) time units after \( i \) sends (or forwards) a message.

Faulty processors whose number, \( f \), is unknown, are chosen by the adversary. In Section 7, we discuss other meanings of \( f \) (and of the time complexity) in the cases in which there are no faults. A faulty link is one that is adjacent to a faulty processor. (This is a simplification. As mentioned above, in reality, it is possible that a processor will continue to function, and its other links will thus not be faulty.) The time for message delivery over faulty links is bounded by \( D \), after which we say that a fault has occurred. Note that the adversary can choose to deliver messages over faulty links very quickly (e.g., in less than \( \delta \) ) or very slowly (even more than \( D \) ). However, if a message is sent over a link at a time \( t \) and no acknowledgement arrives at time \( t + 2D \) a fault has occurred, and an algorithm is permitted to announce a detected fault. Axiom 3 bounds the message delivery time to \( \delta \) over nonfaulty links.

**Axiom 3** Assume that at time \( t \), a \( \text{Send}_{i \rightarrow k}^{j \rightarrow k}[\phi] \) (\( \text{Send}_{i \rightarrow k}^{j \rightarrow k}[\phi] \)) event occurs. Then, before \( t + \delta \leq t + D \), if the link \((i, i+1)\) (respectively, \((i-1, i)\)) is nonfaulty, then a \( \text{Receive}_{i+1}^{i \rightarrow k}[\phi] \) (\( \text{Receive}_{i-1}^{i \rightarrow k}[\phi] \), respectively) event occurs.

## 3 The Task

Intuitively, we want to deliver a message from a source processor to a destination processor. The protocol should detect any failure of links or processors which may delay (or disable) the transmission along the path. Both the transmission and the detection should be done in minimal time.

In our model processor 1 is the source and processor \( n \) is the destination. The path consists of processors \( 1, \ldots, n \) and the links connecting them. We discuss only the transmission of a single message. There are standard methods to extend the results when many messages are transmitted, e.g., appending counters.

The operation of the protocol is based on transmitting the message and additional (control) information between the processors. Therefore, the protocol accepts a message from the higher layer in the source, and delivers a message to the higher layer in the
destination; and for this purpose, it sends and receives (other) messages between processors along the path. Whenever confusion arises, we use the term data message (recall that in this paper we discuss only the case of a single data message, though, of course, multiple data messages can be handled by the same protocols) to refer to the message accepted from and delivered to the higher layer, and the term control messages to refer to the messages transmitted over the links (by the protocol). Note that some of the ‘control messages’ that we use contain the ‘data message’. (Some papers make instead the distinction between “messages” that arrive at the sender from a higher layer and delivered to a higher layer at the receiver, and “packets” that are sent in the network.)

In the protocols, we use two kinds of control messages (in addition to those that carry the data to be delivered): acknowledgements and disconnection notifications $Disc_i$. A disconnection notification $Disc_i$ means that processor $i$ detected a failure in processor $i + 1$ or in the link $(i, i + 1)$. Such messages are normally flooded in the network, and therefore we assume that the protocol terminates when a nonfaulty processor sends $Disc_i$.

Loosely speaking, the protocols are resilient to a strong ‘adversary’, which ‘knows’ the state of every processor and every link, and ‘controls’ the transmission delays of every link (up to $D$), the actual failures of the faulty links (delay larger than $D$), and the entire behavior of the faulty processors. However, the nonfaulty links never fail (always deliver messages in less than $D$), and the nonfaulty processors always operate according to the protocol. This resiliency is formally stated in the following definition.

**Definition 1** A protocol $(P_1, \ldots, P_n)$ is a resilient forwarding faults detector if for every selection of faulty processors and links, in every execution where every nonfaulty processor $i$ executes $P_i$, the following conditions are kept:

Detection: If the source and the destination are nonfaulty, then within a bounded time from the time the source accepts the message, either the message is delivered or a $Disc_i$ message is sent by some nonfaulty processor.

Location: If a $Disc_i$ message is sent by a nonfaulty processor, the link $(i, i + 1)$ is faulty (that is, either processor $i$ or processor $i + 1$ is faulty).

Note that in the detection condition we do not require that the $Disc_i$ message be issued by a nonfaulty processor $i$.

In fact, a correct protocol should also guarantee the following:

Safety: If the source and the destination are nonfaulty, then the destination delivers a message only if this message was the one accepted at the source.

However, since this is trivial in our model, assuming that the sender is nonfaulty (and in reality is handled by different mechanisms, such as CRC or message authentication)
we assume that when a message is delivered to a processor it indeed knows who was its
initiator.

Complexity Measures

The complexity measure given here is the main difference between the model in the paper and the ones in previous works.

We consider time and communication complexities. Both measures are stated as functions of \( n, f, D, \) and \( \delta \), where the parameters are defined in Table 1. The complexities are the worst case values for any execution over paths of length \( n \) with the actual number (unknown) \( f \) of faulty processors, bound \( D \) on the delay and actual delay \( \delta \) over nonfaulty links whose endpoints are also nonfaulty.

The time complexity is the maximum over all executions of the time since the source accepts the (data) message and till either the destination delivers that message or a failure is detected. To compute the time complexity, we consider only executions where the source and the destination processors are nonfaulty, since otherwise it is impossible to guarantee termination.

The communication complexity is the maximal number of transmissions of messages by nonfaulty processors. Messages transmitted by faulty processors are not counted.

4 Simple Solutions

To demonstrate the problem, this section contains two simple protocols. The first, presented in §4.1, is communication-optimal, but has high time complexity. We point out the cause of the high time complexity. This weakness is removed in the protocol presented in §4.2. The protocol of §4.2 is early-terminating, but has high communication complexity.

4.1 End-to-End Fault Detector

This protocol resembles the time-out mechanism of the data link. The data message \( \phi \) is forwarded towards the destination (by \( \text{Send}_i^{1\rightarrow n}[\phi] \) events). When the destination accepts the data message (\( \text{Receive}_n^{1\rightarrow n}[\phi] \)), it sends an acknowledgement backwards (\( \text{Send}_n^{1\leftarrow n}[\text{Ack}] \)). Every processor \( i < n \) checks whether \( i + 1 \) is faulty. Namely, processor \( i \) expects to receive the acknowledgement (\( \text{Receive}_i^{1\leftarrow n}[\text{Ack}] \)) after \( 3(n - i) \) or less TICK\(_i\) events since \( \text{Send}_i^{1\rightarrow n}[\phi] \). If neither the acknowledgement nor a disconnection message are accepted, then \( i \) disconnects link \( (i, i + 1) \) (by \( \text{Send}_i^{1\rightarrow i}[\text{Disc}_i] \)). Processor \( i \) forwards \( \phi \) at most one TICK\(_{i-1}\) event after \( i - 1 \) forwarded it (assuming \( i, i - 1 \) and \((i - 1, i)\) are nonfaulty). Hence, and from the synchronization axioms, processor \( i - 1 \) will accept
either the acknowledgement or the disconnection from \( i \) at most \( 3(n - (i - 1)) \) \( \text{TICK}_{i-1} \) events since forwarding \( \phi \). This means that nonfaulty processors will not be accidently disconnected.

The communication complexity of the End-to-End detector is optimal \((3(n - 1))\). The time complexity is \( 3(n - 1) \cdot D \). (This is the complexity in the case that processor 2 is faulty.) When \( \delta \ll D \), this time complexity is much inferior to the early-terminating time complexity \( O(fD + n\delta) \), achieved by the protocols presented later.

### 4.2 Hop-by-Hop Detector

The End-to-End detector suffers from \( O(nD) \) time complexity. We now describe a detector with time complexity \( O(n\delta + fD) \), which is later shown to be optimal. We do this by extending the use of the acknowledgements. In the End-to-End detector, we use only one acknowledgement message, which signals the completion of the transmission. The idea is to use additional acknowledgements, which signal that the transmission is progressing properly.

In the Hop-by-Hop fault detector, we carry this idea to the extreme, thereby obtaining optimal time complexity. Namely, each processor sends an acknowledgement towards the source immediately upon receiving the message \( \phi \) enroute to the destination.

The improvement in time complexity is obtained by a tighter time-out check in the processors. Consider an arbitrary processor \( i \). In the End-to-End detector, processor \( i \) disconnects from \( i + 1 \) if it does not receive the acknowledgement from the destination \( n \) after more than \( 3(n - i) \) \( \text{TICK}_i \) events since \( i \) forwarded \( \phi \) towards the destination. In essence, \( i \) waits the time needed in the worst case for the data message to reach its final destination \( n \), and for an acknowledgement to arrive from \( n \) to \( i \). Consider the case that \( i + 1 \) never forwards the message to \( i + 1 \). Intuitively, this can be detected by \( i \) (using another protocol in which \( i + 2 \) sends an acknowledgement to \( i \)) with \( O(D) \) time. However, the End-to-End protocol detects the disconnection only in \( \Omega((n - i)D) \) time.

In the Hop-by-Hop detector, processor \( i \) disconnects from \( i + 1 \) if it does not receive any of the acknowledgements from \( k = i + 2, i + 3, \ldots, n \) after more than \( 3(k - i) \) \( \text{TICK}_i \) events. Intuitively, this mechanism guarantees that every \( 3D \) time unit the message processes towards the destination over an additional link (if this link is faulty; otherwise traversing the link costs \( \delta \) time).

### 5 A Design of Resilient Detectors

Both simple detectors presented in section 4 are extremely inefficient in one measure (either time or communication), and optimal in the other measure. In the rest of the
paper, we present detectors which are efficient in both measures, by providing reasonable trade-offs between them. In particular, the detectors are time-optimal up to a constant factor. It is also easy to present implementations of the design which are communication-optimal but with suboptimal time complexity.

Instead of presenting each detector ‘from scratch’, we regard them all as implementations of a common ‘design’. The End-to-End and the Hop-by-Hop detectors may also be regarded as implementations of this design. We prove that every implementation of this design, which satisfies a simple condition, is a resilient forwarding faults detector. Furthermore, we give a simple yet useful bound on the time complexity of implementations. In particular, these general results are used to prove that the detectors of §4 are resilient and that the Hop-by-Hop detector is time-optimal.

The design is presented in Figures 1 and 2, and the explanations below.
Constants of processor $i$:

- $A_i(t)$, array of integer; (different for each protocol)
- $(\text{if } A_i(t) \neq -, \text{then after } tD \text{ since processor } i \text{ accepts } \phi, \text{it sends } \text{Ack} \text{ to processor } A_i(t).)$

Variables of processor $i$:

- $done_i$ : logical (init: FALSE); (True after $i$ terminated (\phi delivered or failure detected))
- $time_i$ : integer; (The current time, i.e., number of TICK events since start)
- $AckSet_i$ : set of intervals (init: $\emptyset$); (Intervals of $\text{Ack}$ which $i$ received or sent)

Figure 1: Design of Forwarding Faults Detector: Declarations

Different implementations are defined by different selections of values for the array $A_{node}(time)$. Basically, if $A_{node}(time) \neq -$, then processor node will send an acknowledgment to processor $A_{node}(time)$ after time events of type $D_{node}$ (i.e., additional $D$ time units elapsed) occurred since node entered the protocol. The detectors of §4 are implemented by using $A_i(t) = -$, except for the following:

- **For the End-to-End detector**: use $A_n(0) = 1$.

- **For the Hop-by-Hop detector**: for every $1 < i \leq n$, use $A_i(0) = 1$.

That is, in the End-to-End detector, only processor $n$ initiates an acknowledgement (thus $A_i(0) = -$ for every $n \neq n$). Moreover, processor $n$ sends the acknowledgement to processor 1, and after 0 time, i.e., immediately on receiving the message. In the Hop-by-Hop detector every processor sends an acknowledgement to processor 1, immediately on receiving the message. For now, it will be easier to think of the more intuitive case that $A_i(t) = -$ for every $t > 0$. The usefulness of the case that $A_i(t) \neq -$ is demonstrated in Subsection 6.3.

The operation begins when the source (processor 1) accepts the message $\phi$ from the higher layer. Each processor $i > 1$ begins operating when receiving $\phi$ from $i - 1$. When the message is received, very processor $i$ (except for the destination) forwards it to the next processor $i + 1$. If $i = n$, the message is also delivered to the higher layer, an Ack is sent to the source, and the protocol terminates. In the other processors ($i < n$), an Ack is sent to $A_i(0)$ (provided that $A_i(0) \neq -$), and $i$ starts executing its WORK_LOOP procedure. The WORK_LOOP is executed repeatedly, terminating only if a failure is detected or the Ack from $n$ is accepted.

Processor $i$ may issue Ack also later, sometime after it started operating. This is done according to the protocol-dependent $A_i(t)$ array. The value of $A_i(t)$ is the identity of the processor to which $i$ should send an Ack at $t \cdot D$ since $i$ began executing. We say that processor $A_i(t)$ is the recipient of this Ack message.

Some economizing is done at that point. (This economizing does not occur in the Hop-by-Hop and the End-to-End implementations; however, it proves very useful in the
implementations of Section 6.) Assume that \( t_1D \) time unit after \( i \) started operating, it forwarded some \( A\text{ck} \) whose destination is some node \( k \), and later, after \( t_2D \) time unit, it is supposed to send an \( A\text{ck} \) to some \( j > k \). (That is, \( A_i(t_2) = j \).) Intuitively, this later \( A\text{ck} \) is no longer necessary, since the earlier \( A\text{ck} \) (the one of time \( t_1D \) was already supposed to tell \( j \) (as well as \( k \)) that processor \( i \) received the message.

Thus, the \( A\text{ck} \) is sent only if its recipient \( A_i(t) \) is farther from \( i \) than the most distant recipient of some previous \( A\text{ck} \) which \( i \) already forwarded. The set \( A\text{ckSet}_i \) holds all the intervals of \( A\text{ck} \) which \( i \) already forwarded. Processor \( i \) checks every \( T\text{ICK}_i \) event, while in \textsc{WorkLoop}, if it should issue an \( A\text{ck} \). The time since \( i \) began executing is approximated (in variable \textit{time}_i) by counting the number of \( T\text{ICK}_i \) events since \( i \) began executing.

The \( A\text{ck} \) messages are forwarded to their recipients by the nodes along the path. Namely, when processor \( i \) receives an \( A\text{ck} \) (from \( i + 1 \)) then \( i \) sends this \( A\text{ck} \) to processor \( i - 1 \). There are three exceptions. First, if \( i \) is the recipient, then, of course, it does not forward the \( A\text{ck} \) any further. Second, processor \( i \) checks that this \( A\text{ck} \) is not ‘bogus’, namely that for some \( t \) and some \( l > i \) the value of \( A_i(t) \) is \( j \). This prevents \( A\text{ck} \) messages from ‘maliciously’ increasing message complexity.

The third exception is that the \( A\text{ck} \) is forwarded only if it may give some processor new information about the progress of the protocol. When a processor \( i \) receives an \( A\text{ck} \) which cannot give new information about the progress of the protocol, we say that the \( A\text{ck} \) is \textit{redundant} and \( i \) does not forward it toward the recipient. Formally, an \( A\text{ck} \) from \( l \) to \( j \) is \textit{redundant} when received by \( i \), if \( i \) already sent an \( A\text{ck} \) from some \( l' \geq l \) to some \( j' \leq j \).

Let us comment about nonredundant \( A\text{cks} \). Note that for \( i \) to forward \( l \)'s \( A\text{ck} \) it is unnecessary for \( i \) to learn anything new from that \( A\text{ck} \). For example, it may be the case that \( i \) already received an \( A\text{ck} \) from \( l + 1 \), and thus \( i \) already “knows” that \( l \) received the messages. Still, it may be the case that this \( A\text{ck} \) of \( l + 1 \) was not forwarded to \( j \), and that \( j \) does not “know” that the message arrived at \( l + 1 \), or even at \( l \). Thus, this \( A\text{ck} \) may be nonredundant.

Every \( T\text{ICK}_i \) event, processor \( i \) checks for time-out of any expected \( A\text{ck} \). A time-out is a failure of \( i + 1 \) to deliver the acknowledgement in time or to disconnect from \( i + 2 \). Note that if \( i \) does not receive an \( A\text{ck} \) on time from any processor \( l > i + 1 \) that is supposed to send an \( A\text{ck} \) to \( i \), then \( i + 1 \) has the opportunity to discover that before \( i \) does. In this case, if \( i + 1 \) is not faulty, it must detect a disconnection of its link to \( i + 2 \) and tell it to \( i \). If this did not happen, then \( i \) concludes that \( i + 1 \) is faulty.

Let us now elaborate on instruction \(<\text{F5}>\) where the mentioned check is done. Processor \( i \) checks if there exists some processor \( l > i \) that was supposed to send an \( A\text{ck} \) to be received by \( i \). That is \( A_i(t) \) equals some \( j \) that is either \( i \) or smaller than \( i \). As mentioned above, such an \( A\text{ck} \) message should pass \( i \). If such an \( A\text{ck} \) was supposed to be sent, it may have not been sent yet, since the original message did not have time to reach \( l \) yet. Alternatively, the \( A\text{ck} \) may have already been sent by \( l \), but did not have enough time to reach \( i \). That is why \( i \) also checks in instruction \(<\text{F5}>\) that there was enough time for
the Ack to reach it, if there were not disconnections. This is the meaning of the check on \( t \) in instruction \(<F5>\). Finally, it could be the case that the Ack was redundant, and thus was never sent, or was omitted. Processor \( i \) verifies that this did not happen, by checking that other Ack messages it received or forwarded were not sent over intervals that contained \( l \) and \( j \).

The check is done using the values of \( A_i(\cdot) \) of the different processor \( j > i \), and the fact that \( time_{i+1} \geq time_i - 2 \). This fact follows from the synchronization axioms and the fact that \( \phi \) is forwarded immediately. If the check fails, procedure DISCONNECT() is used to disconnect processor \( i \) from \( i + 1 \) (since processor \( i + 1 \) of the link to it, or both, are faulty).

5.1 Resiliency of the Design

In this subsection, we prove that every implementation of the design, which satisfies a simple condition, is a resilient forwarding faults detector. Most of the effort is required to prove the location property, which shows that a nonfaulty link \((i, i + 1)\) between nonfaulty processors \( i, i + 1 \) will not be disconnected. We begin with several simpler observations regarding such a link. First, we show that if \( i + 1 \) finishes operating, then \( i \) will also finish operating, after at most \( \delta \).

**Lemma 1** Consider an execution in which link \((i, i + 1)\) and processors \( i \) and \( i + 1 \) are nonfaulty. Processor \( i \) sets \( done_i \leftarrow TRUE \) at most \( \delta \) after processor \( i + 1 \) sets \( done_{i+1} \leftarrow TRUE \).

**Proof** Processor \( i + 1 \) sets \( done_{i+1} \leftarrow TRUE \) only after \( Send_{i+1}^{\rightarrow n}[Ack] \) or \( Send_{i+1}^{\rightarrow j}[Disc_j] \). In any case, the corresponding \( Receive_i^{\leftarrow [\phi]} \) will occur at most after \( \delta \), from axiom 3. Either message will cause \( done_i \leftarrow TRUE \) unless it is already TRUE. □

We now prove that until the processors \( i \) and \( i + 1 \) finish, they are ‘nearly synchronized’ in the values of the variable \( time \).

**Lemma 2** Consider an execution in which link \((i, i + 1)\) and processors \( i \) and \( i + 1 \) are nonfaulty. Whenever \( done_{i+1} = FALSE \), then \( time_i - 2 \leq time_{i+1} \).

**Proof** Processor \( i \) performs \( Send_i^{\rightarrow n}[\phi] \) immediately upon starting operation. From axiom 3, processor \( i + 1 \) accepts \( \phi \) at most \( \delta < D \) later. As long as \( done_{i+1} = FALSE \), the value of \( time_{i+1} \) is the number of \( TICK_{i+1} \) events since \( i + 1 \) began executing. Similarly, the value of \( time_i \) is at most the number of \( TICK_i \) events since \( i \) began executing. The claim follows from axiom 2. □

We now prove that if \( i + 1 \) sends an Ack to \( i \), then this Ack will not be ignored by (statement \(<E1>\) of the design) processor \( i \).
Lemma 3 Consider an execution in which link \((i, i+1)\) and processors \(i\) and \(i+1\) are nonfaulty. Assume that a \(\text{Send}_{i+1}^{i-1} [\text{Ack}]\) occurs at time \(\tau\), for \(j < i, l > i + 1\), while \(\text{done}_i = \text{done}_{i+1} = \text{FALSE}\). Then at time \(\tau + \delta\), either \(\text{done}_i = \text{TRUE}\) or \((\exists [j', l'] \in \text{AckSet}_i) (j' \leq j < l \leq l')\).

Proof From axiom 3, event \(\text{Receive}^{i-1}_{l} [\text{Ack}]\) will occur before time \(\tau + \delta\). Assume that at time \(\tau + \delta\), holds \(\text{done}_i = \text{FALSE}\). Therefore, statement \(<E1>\) is executed in \(i\) upon \(\text{Receive}^{i-1}_{l} [\text{Ack}]\). However, since \(i + 1\) is nonfaulty, it also used statement \(<E1>\) before \(\text{Send}_{i+1}^{i-1} [\text{Ack}]\). Hence \((\exists t) \text{A}_i(t) = j\). This proves the claim, since if the other check of statement \(<E1>\) fails, then the claim holds trivially; and if both checks succeed, then statement \(<E4>\) will be executed and the claim follows.

We now prove the core of the location property. This claim is still slightly weaker than the location property, since we deal only with the case that \(i\) sends the \(\text{Disc}_i\) message.

Lemma 4 Consider an execution in which link \((i, i+1)\) and processors \(i\) and \(i+1\) are nonfaulty. Then the execution does not contain a \(\text{Send}_{i+1}^{i-1} [\text{Disc}_i]\).

Proof The proof is by contradiction. Assume that \(\text{Send}_{i+1}^{i-1} [\text{Disc}_i]\) occurs at time \(\tau\). By definition, a \(\text{fail}_i\) event does not occur. Therefore, statement \(<F5>\) in the design caused the \(\text{Send}_{i+1}^{i-1} [\text{Disc}_i]\) event at \(\tau\). Namely, while \(\text{done}_i = \text{FALSE}\), a \(\text{TICK}_i\) event occurs where for some \(l > i\) and \(t < \text{time}_i - 3(l - i)\) holds \(i \geq \text{A}_i(t) \neq -\) and \((\forall [j', l'] \in \text{AckSet}_i) (\text{A}_i(t) < j' \land l' < l\).

The proof is based on considering the state of \(i+1\) at \(\tau - D\). From Lemma 1, if \(\text{done}_{i+1}\) is \(\text{TRUE}\) at \(\tau - D\), then \(\text{done}_i\) is \(\text{TRUE}\) at \(\tau - D + \delta \leq \tau\). Assume, therefore, that \(\text{done}_{i+1} = \text{FALSE}\) AT \(\tau - D\). From axiom 2, a \(\text{TICK}_i\) event occurred at \(\tau - D\) with \(\text{time}_i\) one less than at \(\tau\), namely at time \(\tau - D\) the following holds: \(\text{time}_i \geq t + 3(l - i)\) (recall that \(t, l, i\) are integers). From lemma 2, the maximum value of \(\text{time}_i - \text{time}_{i+1}\) until \(\tau - D\) is 2 (since \(\text{done}_{i+1} = \text{FALSE}\)). Hence, the \(\text{TICK}_{i+1}\) event in which \(\text{time}_{i+1} \leftarrow t + 1 + 3(l - (i + 1))\) is before \(\tau - D\). Denote the time of this \(\text{TICK}_{i+1}\) event by \(\tau'\). We derive a contradiction from considering the state of \(i + 1\) at \(\tau'\). Consider the two cases: \(l > i + 1\) and \(l = i + 1\).

We first deal with the case \(l > i + 1\), i.e., processor \(i + 1\) failed to forward to \(i\) the \(\text{Ack}\) from \(l\) (to some \(\text{A}_i(t) \leq i\) or to disconnect from \(i + 2\). At \(\tau'\), since \(\text{done}_{i+1}\) is \(\text{FALSE}\), then \((\exists [j', l'] \in \text{AckSet}_{i+1}) (j' \leq j < l \leq l')\). (Otherwise, processor \(i + 1\) will invoke statement \(<F5>\).) An interval \([j', l']\) is added to \(\text{AckSet}_{i+1}\) only by one of statements \(<F4>\), \(<D5>\) or \(<E4>\). Since \(j' < i + 1\), exactly before any of these statements is executed, a \(\text{Send}_{i+1}^{i-1} [\text{Ack}]\) occurs (see Figure 2). Hence, at (or before) \(\tau - D\), a \(\text{Send}_{i+1}^{i-1} [\text{Ack}]\) occurs, with \(j' \leq j < l \leq l'\). From this follows, using lemma 3, that at \(\tau\) there will be some \([x', y'] \in \text{AckSet}_i\) such that \(x' \leq j' < l' \leq y'\). This contradicts the assumption that statement \(<F5>\) was invoked at \(\tau\).

We now deal with the case \(l = i + 1\). First, assume \(t = 0\). Namely, \(\text{Send}_{i+1}^{i-1} [\text{Ack}]\) occurs when \(i + 1\) starts operating, i.e., at most \(\delta\) after \(i\) starts operating. Hence, from
lemma \ref{lem:condition}, there will be some \([j', l'] \in \text{AckSet}_i\) such that \(j' \leq j\) and \(i + 1 \leq l'\), after at most \(\delta\) (i.e., \(2\delta\) since it started). From axiom 2 and the design, the value of \(\text{time}_i\) at \(2\delta\) since \(i\) started is at most 2. Hence, when \(\text{time}_i > 3\), statement \(<F5>\) is not invoked by \(i\) with \(l = i + 1\) and \(t = 0\).

Assume, therefore, that \(t > 0\) and \(l = i + 1\). Recall that \(\text{done}_{i+1} = \text{FALSE}\) at \(\tau'\). Processor \(i+1\) checks at \(\tau'\), the condition of statement \(<F3>\). If the condition holds, then statement \(<F4>\) is executed, i.e., a \(\text{Send}_{i+1}^{j+i}|A\text{ck}\) occurs at \(\tau'\), and the contradiction again follows from lemma \ref{lem:condition}.

Assume that the condition of \(<F3>\) does not hold, namely \((\exists [j', l'] \in \text{AckSet}_{i+1}) j' \leq j\).
Then, previously one of statements \(<F4>\), \(<D5>\) or \(<E4>\) was executed, adding \([j', l']\) to \(\text{AckSet}_{i+1}\). Since \(j' < i + 1\), exactly before any of these statements is executed, a \(\text{Send}_{i+1}^{j-i}|A\text{ck}\) occurred (see Figure 2). Hence, at or before \(\tau'\), a \(\text{Send}_{i+1}^{j-i}|A\text{ck}\) occurs with \(j' \leq j\).
The contradiction follows from lemma \ref{lem:condition}.

We now complete the proof that every implementation of the design, which satisfies a simple condition, is a resilient forwarding faults detector.

\textbf{Theorem 5} Every implementation \(A()\) of the design such that \(A_n(0) = 1\) is a resilient forwarding faults detector.

\textbf{Proof} The safety property follows immediately from the design and axiom 1. To prove the detection property, consider an execution where no message is delivered, and the source and destination are nonfaulty. Since the destination is nonfaulty and no message is delivered, there will be no \(\text{Send}_n^{i-n}|A\text{ck}\) event. From axiom 1, there will be no \(\text{Receive}_1^{i-n}|A\text{ck}\) event. We assumed that \(A_n(0) = 1\). Hence, after \(\text{time}_1 = 3(n-1)\), statement \(<F5>\) sets \(\text{done}_1 \leftarrow \text{TRUE}\) (unless \(\text{done}_1 = \text{TRUE}\) already). But since processor 1 did not accept \(A\text{ck}\) from \(n\), when it sets \(\text{done}_1 \leftarrow \text{TRUE}\) it also did \(\text{Send}_1^{i-1}|D\text{isc}_j\).

We now prove the location property. Consider an execution where \(D\text{isc}_i\) is sent by a nonfaulty processor \(j\), and \(i\) is nonfaulty. The only event in which \(j\) sends, according to the design, a \(D\text{isc}_i\) message, is by a \(\text{Send}_j^{i-1}|D\text{isc}_i\) event. From Figure 2, a nonfaulty processor \(j \neq i\) sends \(D\text{isc}_i\) only if a \(\text{Receive}_j^{i-1}|D\text{isc}_i\) occurred. Since \(j \neq i\), it follows from axiom 1 that \(j < i\) and that before the \(\text{Send}_j^{i-1}|D\text{isc}_i\) event, a \(\text{Send}_i^{i-1}|D\text{isc}_i\) event occurred. From lemma 4, either \(i + 1\) or the link \((i, i + 1)\) or both are faulty.

Since both detectors of \(\S 4\) are implementations of the design with \(A_n(0) = 1\), we conclude:

\textbf{Corollary 6} The End-to-End and the Hop-by-Hop detectors are both resilient forwarding faults detectors.
In this subsection, we show a bound on the time complexity of implementations of the design. This bound suffices to show that the implementations we will present later, as well as the Hop-by-Hop detector, are early-terminating.

Let $T_A(n, f, \delta, D)$ (respectively, $T_{opt}(n, f, \delta, D)$) be the maximal time since an execution of implementation $A$ starts (respectively, since an execution of a time-optimal implementation starts) and until the destination receives the message $\phi$ or an error is detected. We want a bound for the worst ratio $\frac{T_A(n, f, \delta, D)}{T_{opt}(n, f, \delta, D)}$ over every selection of $n, f, \delta$ and $D$.

One can view $T_A$ as the time required by implementation $A$ to either detect a failure or to reach from the source, processor 1, to the destination, processor $n$. Obviously, we can bound $T_A$ by the time required to reach from 1 to any processor $i$, plus the time required to reach from $i$ to $n$. More generally, we bound $T_A$ by the sum of times required to reach from 1 to 2, from 2 to 3, and so on. Note that if every processor and link from $j$ to $l$ is nonfaulty, then it takes exactly $(l - j)\delta$ to reach from $j$ to $l$. Also, if some processor $k$ between $j$ and $l$ is nonfaulty, then the time to reach from $j$ to $l$ is bounded by the time to reach from $j$ to $k$ plus the time to reach from $k$ to $l$. Therefore, we can bound the time complexity by regarding the worst ratio of the times required to reach from processor $j$ to $l$ when all of the processors between $j$ and $l$ are faulty.

The best time to reach from $j$ to $l$ is achieved if $j$ expects $l$ to acknowledge immediately; then the delay is $3D(l - j)$. The time of a specific implementation $A$ is the minimal value of $3D(l' - j) + t$, where $l'$ is a processor after $l$ which sends at time $t$, according to $A$, an acknowledgement whose recipient is $j' < j$. We call this ratio the covering factor of the interval $[j, l]$.

**Definition 2** Let $j, l$ be processors such that $1 \leq j < l \leq n$. The covering factor of $[j, l]$ with respect to $A_i(t)$ is denoted $F_{[j, l]}(A_i(t))$ and defined as follows:

$$F_{[j, l]}(A_i(t)) \overset{def}{=} \min_{i, j', l'} \left\{ \frac{(3(l' - j) + t')A_i(t') = j' \wedge (j' \leq j < l \leq l')}{3(l - j)} \right\}$$

Note that $A_i(t)$ is a set of intervals. The covering factor $F_{[j, l]}(C)$ for any set of interval $C$ (called a cover) is defined similarly.

We now define the covering factor of an implementation, which is the worst covering factor of any interval.

**Definition 3** The covering factor with respect to $A_i(t)$ is denoted $F(A_i(t))$ and defined as follows:

$$F(A_i(t)) \overset{def}{=} \max_{1 \leq j < l \leq n} F_{[j, l]}(A_i(t)) .$$

19
The covering factor \( F(C) \) for a cover \( C \) is defined in a similar way.

The covering factor of an implementation gives an upper bound on the time complexity of this implementation, as follows.

**Theorem 7** The time complexity of every implementation \( A() \) of the design is \( O(n\delta + f \cdot F(A()) \cdot D) \).

**Proof** For simplicity, we ignore link failures, which may be emulated by corresponding processor failures. Also, we assume that the source accepted the message from the higher layer at time 0. Finally, we assume that the source and the destination are nonfaulty, since the time complexity is defined under this assumption. We use the following notations:

**Notations:** Let \( f_i \) be the number of faulty processors before processor \( i \). Also, let \( \tau_i \) denote the time when processor \( i \) received the message and entered the protocol, i.e., the time of Receive\( \rightarrow n[\phi] \).

We prove the following claim for every processor \( i \): if \( i \) is nonfaulty, then before time \( i\delta + 8D \cdot F(A()) \cdot f_i \) one of the following happens: either some nonfaulty processor sent Disc\(_j\), for some \( j \), or processor \( i \) received the message and entered the protocol. The theorem follows by considering \( i = n \).

The claim is trivial for \( i = 1 \). We now prove the claim for processor \( i \) assuming that it holds for every processor before \( i \). If processor \( i \) is faulty, then the claim holds trivially. If both processors \( i \) and \( i - 1 \) are nonfaulty, then the claim holds since processor \( i - 1 \) forwards the message to processor \( i \) immediately.

Assume, therefore, that processor \( i \) is nonfaulty, but processor \( i - 1 \) is faulty. Let \( i' \) be the last nonfaulty processor before \( i \), i.e., \( i' < i \) and every processor in \([i' + 1, i - 1]\) is faulty. By the induction hypothesis, before time \( i' \cdot \delta + 8D \cdot F(A()) \cdot f_{i'} \), either some nonfaulty processor sent Disc\(_j\), for some \( j \), or processor \( i' \) received the message. In the first case, where some nonfaulty processor sent Disc\(_j\), then the claim for \( i \) holds trivially.

Assume, hence, that processor \( i' \) received the message before \( i' \cdot \delta + 8D \cdot F(A()) \cdot f_{i'} \). Denote by \( \tau \) the time when the \( 1 + 3 \cdot (i - i') \cdot F(A())^{th} \) event of the kind \( T I C K_{i'} \) occurs since \( i' \) received the message. From Axiom 2, time \( \tau \) is not more than \( 3 \cdot (i - i') \cdot F(A()) \cdot D + 2D \)

Thus
\[
\tau \leq i' \cdot \delta + 8D \cdot F(A()) \cdot f_{i'} + 3(i - i') \cdot F(A()) \cdot D + 2D .
\]

Since \( f_i = f_{i'} + (i - i' - 1) \) and \( i' + 1 < i \) then
\[
\tau < i \cdot \delta + 8D \cdot F(A()) \cdot f_i .
\]

Hence, it suffices to show that at time \( \tau \), either processor \( i \) received the message or processor \( i' \) sent Disc\(_j\) for some \( j \). We now consider two cases, depending on the state of
processor \( i' \) at \( \rho \). The first case we consider is that at time \( \tau \) holds \( \text{done}_{i'} = \text{TRUE} \); later we deal with the other case.

Since at \( \tau \) holds \( \text{done}_{i'} = \text{TRUE} \), then previously one of procedure \text{DISCONNECT()}\), statement \( \langle \text{H2} \rangle \) or statement \( \langle \text{E2} \rangle \) was executed at processor \( i' \). If procedure \text{DISCONNECT()} was executed, then \( \text{Send}_{i' \leftarrow i'}^{i} [\text{Disc}_{i'}] \) occurred already, and the claim for \( i \) follows. Similarly, if statement \( \langle \text{H2} \rangle \) was executed, then \( \text{Send}^{i' \leftarrow j}_{i} [\text{Disc}_{j}] \) occurred already for some \( j \), and the claim for \( i \) follows.

If statement \( \langle \text{E2} \rangle \) was executed, then statement \( \langle \text{E3} \rangle \) was also executed, since \( j = 1 < i' \). Hence, \( \text{Send}^{i' \leftarrow j-n}_{i} [\text{Ack}] \) occurred. This happens only if \( i \) previously received the message, and then the claim for \( i \) follows.

Consider now the second case, where at time \( \tau \) holds \( \text{done}_{i'} = \text{FALSE} \). Hence, at \( \tau \) processor \( i' \) executes \( \langle \text{F3} \rangle \), after which \( \text{time}_{i'} = 1 + 3 \cdot (i - i') \cdot F(A()) \).

By Def. 3, \( F(A()) \geq F_{[i' + 1, i - 1]}(A()) \). By Def. 2, there are \( l, t \) such that \( A_{l}(t) \leq i' \) and \( i \leq l \) and

\[
F_{[i', l]}(A()) = \frac{3(l - i') + t}{3(i - i')}.
\]

Hence, when processor \( i' \) executes \( \langle \text{F5} \rangle \) at time \( \tau \), then:

\[
\text{time}_{i'} = 1 + 3 \cdot (i - i') \cdot F(A())
\]
\[
> 3 \cdot (i - i') \cdot F_{[i', l]}(A())
\]
\[
= 3 \cdot (i - i') \cdot \frac{3(l - i') + t}{3(i - i')}
\]
\[
= 3 \cdot (l - i') + t.
\]

Hence, \( t < \text{time}_{i'} - 3 \cdot (l - i') \) at time \( \tau \). Namely, at time \( \tau \), either \( \text{Send}^{i' \leftarrow l'}_{i} [\text{Ack}] \), \( \text{Receive}^{i' \leftarrow l'}_{j} [\text{Ack}] \) occurred with \( j' \leq A_{l}(t) \leq i' < l \leq l' \), or processor \( i' \) executes procedure \text{DISCONNECT()} due to \( \langle \text{F5} \rangle \). If processor \( i' \) executes procedure \text{DISCONNECT()} then \( \text{Send}^{i' \leftarrow l'}_{i} [\text{Disc}_{i'}] \) occurs and the claim follows. On the other hand, from Axiom 1, if \( \text{Receive}^{i' \leftarrow l'}_{i} [\text{Ack}] \) occurred, then \( \text{Send}^{i' \leftarrow l'}_{i} [\text{Ack}] \) occurred before, and this happens only after \( i \) entered the algorithm.

We now observe that the optimal time complexity is bounded by \( (n - f){\delta} + fD \).

**Lemma 8.** Every forwarding faults detector has a run with time complexity at least \( (n - f){\delta} + fD \).

**Proof** Consider the execution where processors \( 2, \ldots, 2 + f - 1 \) are faulty. The fault merely causes the delay upon forwarding the message through these processors to be \( D \) instead of \( \delta \).

We deduce that:
Corollary 9 A detector that implements the design such that $F(A())$ is bounded by a constant, is early-terminating. In particular, the Hop-by-Hop detector is early-terminating.

Proof The general claim follows from Theorem 7 and Lemma 8. The Hop-by-Hop detector has $A_i(t) = 1$. By definition, for every $j,l, F_{t_1}[t](A()) = 1$; hence, $F(A()) = 1$. The claim follows. □

Algorithm for the source $i = 1$:

< B1 > On accepting $\phi$:
< B2 >   \{ Send$_i^{n\rightarrow}[\phi]$ ;
< B3 >   repeat WORK LOOP() until done$_i$; \}

Algorithm for the destination $i = n$:

< C1 > On Receive$_n^{n\rightarrow}[\phi]$ : \{ deliver $\phi$ ; Send$_n^{n\rightarrow}[Ack]$ ; done$_n$ \rightarrow TRUE \}

Algorithm for intermediate processors $1 < i < n$:

< D1 > On Receive$_i^{n\rightarrow}[\phi]$ :
< D2 >   \{ Send$_i^{n\rightarrow}[\phi]$ ; (forward message)
< D3 >   if $(A_i(0))n$ then
< D4 >     \{ Send$_i^{A_i(0)\rightarrow}[Ack]$ ;
< D5 >     AckSet$_i$ \leftarrow $\{[A_i(0), i] \}$ ; \}
< D6 >   repeat WORK LOOP() until done$_i$; \}

Procedure WORK LOOP();:

< E1 > On Receive$_i^{i\rightarrow}[Ack]$ such that

\[ \exists j \in [1, n] \land (A_i(t) = j) \land (\forall [j', l'] \in AckSet_i) \{(j < j') \lor (l' < l) \} \]

< E2 >   \{ if $j \geq 1$ and $l = n$ then done$_i$ \rightarrow TRUE ;
< E3 >   if $j < i$ then Send$_i^{i\rightarrow}[Ack]$ ; (Forward Ack if i is not its destination)
< E4 >   AckSet$_i$ \leftarrow AckSet$_i$ $\cup$ \{[j, l]\}; \}

< F1 > On TICK$_i$:
< F2 >   \{ increment time$_i$ ;
< F3 >   if $(A_i(time$_i$))n$ then
< F4 >     \{ Send$_i^{A_i(time$_i$)\rightarrow}[Ack]$ ; AckSet$_i$ \leftarrow AckSet$_i$ $\cup$ \{[A_i(time$_i$), i] \};
< F5 >     if $\exists i > j$ and $(\exists t) t < time$ \land 3 \land i$ such that $i \geq A_i(t)$ and

\[ (\forall [j', l'] \in AckSet_i) (A_i(t) < j' \lor l' < l) \] then DISCONNECT() ; \}

< G1 > On fail$_i$ : DISCONNECT() ;

< H1 > On Receive$_i^{i\rightarrow}[Disc]$ :
< H2 >   \{ done$_i$ \rightarrow TRUE ; Send$_i^{i\rightarrow}[Disc]$ ; \}

Procedure DISCONNECT() : (Disconnect processor i from i + 1)

< I1 >   \{ Send$_i^{i\rightarrow}[Disc]$ ;
< I2 >   done$_i$ \rightarrow TRUE ; \}

Figure 2: Design of a forwarding faults detector for processor $i$. 22
6 Optimal Time and Communication Efficient Implementations

In this section we present three implementations of the design, which ensure early termination (time complexity $O(n\delta + fD)$) and efficient ($O(n \log n)$ in the worst case) communication. Each implementation is a refinement of the previous one.

Throughout this section, we make the simplifying assumption that $n - 1$ is an even power of two. This at most quadruples the complexities of the solutions, when applied to paths where $n - 1$ is not an even power of two. In these cases, the source processor may ‘play the rule’ of a sufficient number of processors, to extend $n$ so that $n - 1$ will become an even power of two.

6.1 The Immediate Acknowledgements Implementation

We begin by considering implementations where every acknowledgement is sent immediately upon receiving the message. For such an implementation $A()$, the covering factor of an interval $[j, l]$, as defined in Eq. (1), has the following simplified form:

$$F_{[j,l]}(A()) = \min_{j',l'} \left\{ \frac{l' - j'}{(A_{l'}(0) = j'(j' \leq j < l \leq l'))} \right\}$$  \hspace{1cm} (3)

We are interested in implementations which are early-terminating. From Corollary 9, such are the implementations where $F(A())$ is bounded by a constant. Namely, for every $[j, l]$ where $j < l$ there is some interval $[j', l']$ such that $A_{l'}(0) = j'$ and $j' \leq j < l \leq l'$ and $\frac{l - j}{l - j}$ is bounded by a constant.

A natural selection of $A()$ is to send acknowledgements over intervals of lengths which are powers of two, i.e., $1, 2, 4, \ldots, (n - 1)$. Let us describe the set of intervals used (see also a definition below). For every length $2^k \leq (n - 1)$ there are two types of intervals. Intervals of the first type start at every processor in a position of the form $r \cdot 2^k + 1$ for every $r$ for which such a processor exists. For example, if $2^k = \frac{n-1}{2}$ one such interval starts at processor $2^k + 1$ (for $r = 1$ and the other starts at $n$. Any acknowledgement intervals of length $2^k$ that starts at a processor, $i$, ends at processor $i - 2^k$. For example, the interval that starts at processor $2^k + 1$ ends at processor 1. Note, for example, that a subpath (of the message path) of length $L$ such that $\frac{1}{2} < L < \frac{3}{2}$ is covered with a covering factor of less than two if and only if it is contained in one of the two acknowledgement intervals (described above) of length $\frac{n}{2}$. However, if it partially intersects with both, then only the end-to-end acknowledgement interval covers it. This effect becomes more damaging to the covering factor when we consider a shorter subpath.

To alleviate this effect, we introduce the second type of acknowledgement interval used. A subpath not covered (with a covering factor of 2 or less) by an interval of the first type, will be covered (with a covering factor of 4 or less) by an interval of the second
type. The intervals of the second type (still of length $2^k$) start at $(r + \frac{1}{2}) \cdot 2^k + 1$ for every $r$ for which there is such a processor. For example, for $2^k = \frac{n-1}{2}$ there is only one interval of the second type, and it starts at $2^{k-1} + 1$. The acknowledgement intervals which are ‘shifted’, and start at $(r + \frac{1}{2}) \cdot 2^k + 1$, are needed to cover intervals which span over the connection between the acknowledgement intervals, e.g., intervals which include processor $\frac{n+1}{2} + 1$. This selection of acknowledgement intervals ensures that every interval $[j, l]$ is covered by an acknowledgement interval which is not ‘much larger’, as we now formalize.

We specify this implementation in Eq. (4). We now prove that $F(A(t))$ is bounded by a constant, and hence that it is early-terminating.

\[
A^{(1)}_k(t) \defeq i - \max_{k \geq 0} \left\{ 2^k \left( (i = 2^k \cdot r + 1) \text{ or } \left( i = 2^k \cdot \left( r + \frac{1}{2} \right) + 1 \right) \right) \right\}. \quad (4)
\]

**Lemma 10** The implementation $A^{(1)}(t)$ is early-terminating.

**Proof** From Corollary 9, it suffices to show that $F(A^{(1)}(t))$ is bounded by a constant. The proof is by showing that every interval $[j, l]$ is ‘covered’ by an interval in $A^{(1)}(t)$ whose length is at most four times $l - j + 1$.

Consider an interval $[j, l]$ such that $1 \leq j < l \leq n$. Without loss of generality, assume that $l - j < \frac{n+1}{2}$. Let $k$ be the minimal such that $l - j < \frac{2^k}{4}$, i.e., $k \overset{df}{=} \lceil \log_2(4 \cdot (l - j + 1)) \rceil$. Let $r$ be the minimal such that $l < 2^k \cdot r + 1$. Namely, $2^k \cdot (r - 1) + 1 < l$.

If $2^k \cdot (r - 1) + 1 < j$ then $[j, l]$ is covered by the acknowledgement interval $[A^{(1)}_{2^k \cdot r + 1}(0), 2^k \cdot r + 1]$. From Eq. (3), $F_{[j, l]}(A^{(1)}(t)) \leq 4$.

Assume, therefore, that $j \leq 2^k \cdot (r - 1) + 1$. In this case, $2^k \cdot (r - \frac{3}{2}) + 1 < j \leq l < 2^k \cdot (r - \frac{1}{2}) + 1$, since $l - j < 2^{k-2}$ and $2^k \cdot (r - 1) + 1 \leq l$. Again, from Eq. (3), $[j, l]$ is covered by the acknowledgement interval whose last processor is $2^k \cdot (r - \frac{1}{2}) + 1$. Hence, again $F_{[j, l]}(A^{(1)}(t)) \leq 4$.

To complete the analysis of this implementation, we note that the communication complexity is obviously the total length of the intervals, which is $O(n \log(n))$.

### 6.2 A Tight Lower Bound for Oblivious Protocols

We consider protocols whose operations include: forwarding the message, computing, using time-outs, and sending acknowledgements. It is natural to classify such protocols by the way they handle the acknowledgements. An important subset, termed *oblivious protocols* sends any acknowledgements they wish to send immediately, without delaying it. Similarly, if they receive an acknowledgement to be forwarded, they forward it immediately. All the previously published protocols, as well as all the protocols up to this point in this paper are oblivious. In the design, this family of protocols is captured by having $A_i(t) = 0$ for every $t \neq 0$. 


Theorem 11 An oblivious protocol is time-optimal if and only if in it $F(A())$ is a constant.

Proof The “if” part follows from Corollary 9. For the “only if” part, consider an oblivious protocol for which $F(A())$ is some $f(n)$. (Notice that for an oblivious protocol $F(A())$ does not depend on $t$.) Let $j,l$ be two processors such that $F_{[j,l]}(A()) = f(n)$ (see Definitions 2 and 3) and let $l'$ be the one mentioned in Definition 2. Consider the case that all the processors in the closed interval $[j,l]$ are faulty, but no other processor is faulty. Consider the state of knowledge (see e.g., [HM90]) of processor $j'$ at any time before $(l'-j)D$. Clearly it did not receive any message from a nonfaulty processor $p > l$. Thus, the state of knowledge of processor $j$ at such a time is the same as in the case that all the processors in the interval $[j,l']$ are faulty. The theorem now follows from Lemma 8.

We now prove that the set of intervals used by the previous implementation is optimal in the sum of the lengths of the intervals. Note that this sum determines the message complexity of the protocol. The claim is that for an interval cover (a set of intervals $C$, if $F(C)$ is a constant then $L(C) = \Theta(n \log n)$, where $L(C) = \sum_{x \in C}$.

Intuitively, even a long interval can cover only a few short intervals. To cover long intervals, the cover must contain some long intervals. In fact, a few long intervals in the cover suffices to cover every long interval. The main observation in the proof is that a long interval cannot cover too many short intervals. Thus, additional intervals must be introduced into the cover. These intervals may be short and thus, it may seem that the contribution of each of them to $L(C)$ is small. However, many short intervals are needed in the cover, to cover all the short intervals. Thus, the total contribution of the short intervals to $L(C)$ is large.

Theorem 12 For every interval cover $C$ s.t. $F(C)$ is a constant, the total length $L(C)$ is $\Omega(n \log n)$.

Proof Without loss of generality, assume $(3k)n - 1 = 2(F(C))^k$. (Otherwise, prove for $n' < n$ for which there exists such $k$; this adds only a constant factor.)

Let $I_x$ be the set of intervals of path $< n, 1 >$ that contain (each) exactly $x$ links. The key observation is that any single interval in $C$ (even a very long one) can cover at most $(F(C) - 1)x + 1 \leq (F(C) - 1)x + x \leq F(C)$ intervals in $I_x$ with covering factor of $F(C)$. Consider the sets $I_x(i)$ where $x(i) = \frac{n - 1}{2F(C)x(i)}$, for $1 \leq i \leq -1 + \log_{F(C)} n$. Clearly $|I_x(i)| \geq \frac{n - 1}{2}$. Thus, the observation implies that $C$ must contain at least $\frac{|I_x(i)|}{F(C)x(i)} \geq \frac{(n - 1)2F(C)x + 1}{2(n - 1)F(C)} \geq F(C)^x$ intervals, the length of each at least $\frac{n - 1}{2F(C)x}$.

We now partition $C$ into sets of intervals, and give a lower bound for the total sum of each set. The sum of these bounds will later give us a lower bound for the total sum of $C$. 25
Let $C_0 \subseteq C$ include the interval $< n, 1 >$. For $i = 1$ we have that $x(i) = \frac{n-1}{2F(C)}$. To cover $L_{x(1)}$ cover $C$ must include $F(C)$ intervals, the length of each is at least $x(1) = \frac{n-1}{2F(C)}$. One of them can be the $< n, 1 >$ interval, but additional $F(C) - 1$ intervals are needed. Let $C_1 \subseteq C$ be a set of such additional intervals. Note that $C_1 \cap C_0 = \emptyset$. The total sum of $C_1$ is at least $\frac{F(C) - 1}{2F(C)}$. We continue to construct the $C_i$’s inductively. For intervals in $L_{x(i)}$, $C$ must include at least $F(C)^i$, where the length of each is at least $\frac{n-1}{2F(C)^i}$. As before, $F(C)^i$ such intervals are already included in the sets $C_0, C_1, C_2, \ldots, C_{i-1}$. Thus, we can construct a set $C_i \subseteq C$ such that $\forall 1 \leq j < i; C_i \cap C_j = \emptyset$ and the cardinality of $C_i$ is at least $F(C)^i - F(C)^{i-1}$. Thus, the total sum of $C_i$ is at least $\frac{F(C)^i - 1}{2F(C)}$ messages even in executions where $f$ and $\delta$ are small. The worst case communication complexity remains $O(n \log n)$.

Recall that in the design (Figure 2), processor $i$ does not forward an Ack from processor $l$ to processor $j$ if this Ack is redundant, namely if $i$ already forwarded an Ack from some $l' \geq l$ to some $j' \leq j$. However, in the Immediate-Ack implementation, each processor sends its acknowledgements upon beginning execution. Therefore, the Immediate-Ack implementation of the design sends all of its acknowledgements in every execution (without failures).

However, if some of the processors delay issuing their acknowledgements, then it is possible that a delayed acknowledgement will become ‘redundant’. For example, suppose that each of the processors delay all their acknowledgements by $D$, except the destination. If Ack from the destination $n$ reaches the source 1 before $D$ since the source started the protocol, then all of the other acknowledgements become redundant.

Obviously, the delay until the acknowledgements are sent increases the maximal time until the protocol terminates. For example, if the processors $i$, s.t. $1 < i < n$ delay their acknowledgements by $2Dn$, then the resulting implementation has the same time complexity, up to a constant, as the End-to-End fault detector.

The decrease in message complexity is, therefore, in the cost of an increase in the time complexity. The implementations presented in this section are early-terminating, since the delay is bounded by twice the time required for the Immediate-Ack implementation. In fact, the adaptive implementations, presented in the rest of this section, are modifications of the Immediate-Ack implementation.

The first adaptive implementation saves messages mainly if there are no faulty processors. The number of acknowledgements sent is a function of the time complexity, which
is lowest when there are no faults. The second adaptive implementation saves additional messages when there are faulty processors. In the following subsections we explain each of these implementations and analyze their properties.

6.4 Send *Ack* Only When Really Needed

The first idea is to wait, as much as possible, before sending intermediate acknowledgments (i.e., *Ack* from \(l \leq n\) to \(j \geq 1\)). The longer is the delay in sending an intermediate acknowledgement, the larger is the hope that the acknowledgement from \(n\) to \(1\) will make the intermediate acknowledgement redundant. In this subsection, we study how much any specific processor can wait without increasing the time complexity “too much”. By implementing the idea of this subsection, the communication complexity is reduced to 
\[
O\left( n \log \left( 1 + \frac{n^2}{D} \right) \right),
\]
if \(f = 0\). In the next subsection, we show how to keep the communication complexity low, even when \(f > 0\).

To demonstrate the idea, let us first investigate the case that there is exactly one faulty processor \(i\) (although the algorithm must still be early-terminating for any number of faults). Early termination is assured in the Immediate-Ack implementation since \(i\) is covered by the interval \([i-1, i+1]\) of length two in \(A^{(1)}(i)\). Note that an *Ack* from \(i+1\) to \(i-1\) is allowed to take at least \(D\) times if \(i\) is faulty. (Recall also that time complexity in the presence of one fault is \(\Omega(D)\).) Hence, if \(i+1\) waits \(D\) times before sending the *Ack*, it at most doubles the protocol’s time complexity in executions with one fault. In executions with no faults, no acknowledgement is needed (although acknowledgements must be sent, since the number of faults is not known). Thus, a delay in any acknowledgement does not increase the time complexity in such executions. Finally, in executions with more than one fault, the acknowledgements over intervals of length 2 do not help, so any delay in them cannot increase the time complexity.

The adaptive detector sends the acknowledgements of the intervals of length two of \(A^{(1)}(i)\), but only after waiting \(O(D)\) for *Ack* from \(n\). Early termination is always obtained, as explained above. However, in executions with exactly one fault (and a small \(\delta\)), this achieves both early termination and optimal message complexities. The optimal message complexity is achieved, in this case, since the acknowledgement from processor \(n\) arrives at every other processor before it sends any acknowledgement of its own. Thus, all the other acknowledgements become redundant, and are not sent.

In general, if there are at most \(f\) faults, then the time complexity with \(f\) faulty processors is at least \(O(f \cdot D)\). Hence, the adaptive implementation delays sending the acknowledgements of intervals of length \(f\) by \(f \cdot D\).

We now formally present the implementation of this subsection. It is easy to see that the covering factor is bounded by a constant. Hence, by Corollary 9, the implementation is early-terminating; we later show this formally. For simplicity, we assume that \(n-1\) is an even power of two. For \(1 \leq i \leq n\) and \(0 \leq t < n-1\) define \(A^{(2)}_i(t)\) as follows:

\[
A^{(2)}_i(t) \overset{\text{def}}{=} \left\{ \begin{array}{ll}
1 & \text{if } i = n \\
i - t & \text{if } (\exists k) k = 2^k \land (\exists r \in \mathbb{N}) i = rt + 1 \\
i - t & \text{if } (\exists k) k = 2^k \land (\exists r \in \mathbb{N}) i = (r + \frac{1}{2})t + 1 \\
- & \text{otherwise}
\end{array} \right.
\]

(5)
The implementation of this subsection achieves communication complexity \(O\left(n \log \left(\frac{m^2}{D}\right)\right)\) if \(f = 0\). We do not prove this here, since it is a corollary of a theorem presented in the sequel.

### 6.5 Saving Acknowledgements

In this section, we improve the previous implementation to save messages in a particular, bad scenario. This improves the worst case message complexity for \(f > 0\).

Recall that acknowledgements are sent (in the previous implementation) at times \(2^k \cdot D\) for \(k = 0, 1, 2, \ldots\). At first glance, this seems to imply that the communication complexity is \(O(n)\) times the logarithm of the time complexity divided by \(D\), i.e., \(O\left(n \log \left(f + \frac{n^\delta}{D}\right)\right)\). However, there is a bad scenario in which the complexity is \(\Omega(n f)\). Indeed, in this scenario, the number of acknowledgements sending (and forwarding) events until the message is delivered at processor \(n\) (at time that is \(O(fD)\)) is just \(O(n \log f)\). However, additional acknowledgements are sent after the message is delivered.

Put differently, what we did prove (regarding time complexity) is that if there are \(f\) faults, then the message must be delivered at node \(n\) at time that is \(O(fD)\) (if \(\delta\) is small). However, we did not prove that the acknowledgement from \(n\) to \(1\) is delivered in such a time. In fact, for the implementation of the previous section, one can show a case where for \(f = O(\log n)\) faults the time for delivering \(n\)'s acknowledgement in that implementation is \(\Omega(2^f D)\). The way the time complexity is defined (only until the delivery of the message, or the disconnection of a link), we do not care about the time it takes the acknowledgement to arrive after the message is delivered. Still, this increases the message complexity of the previous implementation, since its message complexity is, actually, in the order of \(n\) multiplied by the logarithm of the time until all protocol-related communication ceases. The improvement of the message complexity in this section is obtained by shortening that ceasing time.

Let us demonstrate a bad scenario. Assume that processor \(n - 1\) is faulty; more specifically, assume that a message over any link of processor \(n - 1\) is delivered exactly after \(D\) time (rather than after \(\delta\) time). Processor \(n - 2\) expects (and receives) an Ack from \(n\) within \(4D\) time after \(n - 2\) forwarded the message to \(n - 1\). By that time, processor \(n - 2\) already sent a length 2 interval acknowledgement to \(n - 4\). Thus, the next acknowledgement that \(n - 4\) is waiting for is a length 4 interval acknowledgement, expected to arrive after a time that is double that of the length two acknowledgement.

Carrying this argument further, when a length \(2^i\) interval acknowledgement \(\text{Ack}_i\) arrives at its destination, \(j\), the next acknowledgement expected by processor \(j - 1\) is a length \(2^{i+1}\) interval acknowledgement \(\text{Ack}_{i+1}\), that is expected in double the time. Even if both \(\text{Ack}_i\) and \(\text{Ack}_{i+1}\) arrive at \(j\) at the same time, processor \(j\) can delay \(\text{Ack}_{i+1}\) without \(j - 1\) noticing a fault and disconnecting the link. Let \(f_j\) be the number of faults in the interval \([j, n]\). The state of knowledge of \(j - 1\) at this point is the same as in the case that the number of faults is \(2^f\). Thus, the length \(n - 1\) interval acknowledgement from \(n\) to 1
(i.e., the end-to-end acknowledgement) can also be delayed $2^j$ without causing $j - 1$ to detect the fault.

Let us now describe the improvement to the previous implementation. Surprisingly, the idea is to send (few) other acknowledgements for ‘long’ intervals. Acknowledgements sent quickly over ‘long’ intervals which do not contain faults will reach every processor in the interval quickly. Therefore, many ‘short’ acknowledgements whose intervals are contained in the ‘long’ interval will become redundant, and therefore, these ‘short’ acknowledgements will not be sent.

A natural selection is the set of intervals whose length is $\sqrt{n - 1}$. Let us comment that had we used only this set, without using the shorter acknowledgements, we could obtain $O(n\delta + f\sqrt{nD})$ time with $O(n)$ communication, which is a communication optimal but time suboptimal solution. However, to achieve time-optimality, we do combine this set with the ‘short’ intervals.

Furthermore, as we now explain, we need also to send ‘quickly’ acknowledgements over ‘long’ intervals, i.e., intervals whose length is more than $\sqrt{n - 1}$. By sending acknowledgements over the intervals of length $\sqrt{n - 1}$ immediately upon forwarding the message, we prevent scenarios as described above for intervals whose length is less than $\sqrt{n - 1}$, except for the few intervals of length $\sqrt{n - 1}$ which contain faulty processors. However, we still have to deal with the intervals whose length is more than $\sqrt{n - 1}$.

Recall that the ‘short acks’, as defined in Eq. (5), are sent in order of increasing length. Namely, the Ack of interval of length $l$ is sent after $lD$. The ‘long acks’ are acknowledgements sent in the reverse order, from the longest to the shortest. Namely, the ‘long ack’ of interval of length $\frac{n - 1}{l}$ is sent after $lD$. (The length of the intervals divide $n - 1$.) The reason the ‘long’ intervals are sent in this order is similar to the idea behind the intervals of length $\sqrt{n - 1}$—a single successful longer interval may make many relatively shorter intervals redundant. Note that the intervals which we want to become redundant as a result of the ‘long’ intervals are not really short; their length is more than $\sqrt{n - 1}$.

Formally, the implementation of this subsection is presented in Eq. (6) below. For simplicity, we assume that $n - 1$ is an even power of two. For $1 \leq i \leq n$ and $0 \leq t < n - 1$ define $A_i^{(3)}(t)$ as follows:

$$A_i^{(3)}(t) = \begin{cases} 
1 & \text{if } i = n \\
\sqrt{n - 1} & \text{if } (\exists r \in \mathbb{N}) i = r\sqrt{n - 1} + 1 \\
\sqrt{n - 1} & \text{if } (\exists r \in \mathbb{N}) i = (r + \frac{1}{2})\sqrt{n - 1} + 1 \\
\frac{n - 1}{l} & \text{if } (\exists k) t = 2^k \land (\exists r \in \mathbb{N}) i = r\frac{n - 1}{l} + 1 \\
\frac{n - 1}{l} & \text{if } (\exists k) t = 2^k \land (\exists r \in \mathbb{N}) i = (r + \frac{1}{2})\frac{n - 1}{l} + 1 \\
- t & \text{if } (\exists k) t = 2^k \land (\exists r \in \mathbb{N}) i = rt + 1 \\
- t & \text{if } (\exists k) t = 2^k \land (\exists r \in \mathbb{N}) i = (r + \frac{1}{2})t + 1 \\
- & \text{otherwise.}
\end{cases}$$

Note that the first, sixth, and seventh lines correspond to acknowledgements that are sent also by the previous implementation. We term the acknowledgements defined in the
sixth and seventh lines “short” acknowledgements. The new acknowledgements defined in the fourth and fifth lines are termed “long” acknowledgements. The new acknowledgements defined in the second and third lines are termed “medium” acknowledgements.

Intuitively, this helps since ‘long’ intervals which contain no faulty processors are ‘almost unaffected’ by the faults ‘outside’. For example, if all the processors from 1 till \((\sqrt{n-1} + 1)\) are nonfaulty, then send only \(O(n \log(\frac{n^2}{D}))\) messages.

6.6 Complexities of the Adaptive Implementations

We begin by showing that the time complexity has not deteriorated significantly.

Lemma 13 The implementations \(A^{(2)}()\) and \(A^{(3)}()\) are early-terminating.

Proof We state the proof for \(A^{(3)}()\), but all of our arguments hold for \(A^{(2)}()\) as well. From Corollary 9, it is sufficient to bound \(F(A^{(3)}())\) by a constant. We use the similarity between \(A^{(3)}()\) and \(A^{(1)}()\).

From Lemma 10 we know that \(F(A^{(1)}())\) is bounded by a constant. For every \(j, l\) such that \(1 \leq j < l \leq n\), we show that \(F_{[j,l]}(A^{(3)}()) \leq \frac{1}{3} + F_{[j,l]}(A^{(1)}())\). Let \(j', l'\) be such that \(j' \leq j < l \leq l'\) and \(A^{(1)}(0) = j'\) and \([j', l']\) is the ‘best cover’ of \([j, l]\) in \(A^{(3)}()\), namely

\[
F_{[j,l]}(A^{(1)}()) = \frac{l' - j}{l - j}.
\]

By the definition of \(A^{(3)}()\) there is some \(t' \leq (l - j)\), such that \(A^{(3)}(t') = j'\). Hence,

\[
F_{[j,l]}(A^{(3)}()) \leq \frac{3(l' - j) + t'}{3(l - j)} \\
\leq \frac{3(l' - j) + (l - j)}{3(l - j)} \\
\leq \frac{1}{3} + \frac{3(l' - j)}{3(l - j)} \\
\leq \frac{1}{3} + F_{[j,l]}(A^{(1)}())
\]

which completes the proof. \(\square\)

We now turn to the proof of the communication complexity. We begin by showing a simple necessary condition for sending a particular ‘short acknowledgement’.

Lemma 14 Let \(j, i\) be processors such that \(0 < i - j < \frac{\sqrt{n-1}}{2}\). If \(\text{Send}_{j, i}^{(j + i)} [\text{Ack}]\) occurs, then either there is a faulty processor in \([i, i + \sqrt{n-1}]\) or \(i - j \leq \sqrt{n-1} \cdot \frac{2A}{D}\).
Proof Since \( i - j < \sqrt{n-1} \) and from Eq. (6), there is some processor \( l \) such that \( A_l^{(3)}(0) \neq -1 \) and \( l - A_l^{(3)}(0) = \sqrt{n-1} \) and \( A_l^{(3)}(0) \leq j < i \leq l \). If there is a faulty processor in \([i, l]\) then the claim holds. Assume, therefore, that there is no faulty processor in \([i, l]\).

Since every nonfaulty processor forwards the message and the acknowledgements immediately, then \( \text{Receive}^{A_i^{(3)}(0) - l}_l[Ack] \) occurs not later than \( 2\sqrt{n-1} \cdot \delta \) after processor \( i \) entered the protocol. From the second condition of \( <F3> \), processor \( i \) does not issue the acknowledgement to \( j \) after \( \text{Receive}^{A_i^{(3)}(0) - l}_l[Ack] \). Hence, \( \text{Send}^{i \leftarrow i}_l[Ack] \) may occur only before \( \text{Receive}^{A_i^{(3)}(0) - l}_l[Ack] \) occurs, i.e., before \( 2\sqrt{n-1} \cdot \delta \) since processor \( i \) started.

On the other hand, from Eq. (6) and since \( i - j < \sqrt{n-1} \), then \( \text{Send}^{i \leftarrow i}_l[Ack] \) occurs only when \( \text{time}_i > i - j \). From \( <F2> \) and Axiom 2 holds \( i - j < \text{time}_i \) only after at least \( (i - j) \cdot D \) since processor \( i \) started. Hence, \( \text{Send}^{i \leftarrow i}_l[Ack] \) occurs only if \( (i - j) \cdot D \leq 2 \cdot \sqrt{n-1} \cdot \delta \). □

Lemma 14 shows that a ‘short acknowledgement’ is issued only if it is one of the very short which are required since \( \delta \) is not negligible, or if it is ‘close’ to a faulty processor. We now bound the maximal number of ‘short’ acknowledgements issued due to ‘close’ faulty processors.

**Lemma 15** For every \( k \) such that \( \sqrt{n-1} \cdot \frac{2^k}{D} < 2^k < \sqrt{n-1} \), there are at most \( f \cdot 2 \cdot \sqrt{n-1} + 1 \) events of type \( \text{Send}^{i - 2^k \leftarrow i}_l[Ack] \).

Proof From Lemma 14, if \( \text{Send}^{i - 2^k \leftarrow i}_l[Ack] \) occurs as specified, then there is a faulty processor in \([i, i + \sqrt{n-1}]\). From Eq. (6), for some integer \( r \) holds either \( i = r2^k + 1 \) or \( i = (r + \frac{1}{2})2^k + 1 \). Hence, for every faulty processor \( l \), there are at most \( 2 \cdot \sqrt{n-1} + 1 \) intervals \([i - 2^k, i] \) such that \( l \in [i, i + \sqrt{n-1}] \). □

Lemma 15 bounds the communication due to acknowledgement intervals shorter than \( \sqrt{n-1} \). In order to bound the entire communication complexity, we have to consider also longer acknowledgement intervals. We begin by bounding the maximal number of intervals containing, or are “near to”, faulty processors.

**Lemma 16** For every \( k \in \mathbb{N} \), there are at most \( 5 \cdot f \) processors \( i \) such that for some \( t \) holds \( A_i^{(3)}(t) = i - 2^k \) and the interval \([i - 2^k, i + 2^k] \) contains a faulty processor.

Proof From Eq. (6), and since we assumed that \( \sqrt{n-1} \) is a power of two, it follows that if \( A_i^{(3)}(t) = i - 2^k \), then \((\exists r \in \mathbb{N})(i = r \cdot 2^k + 1) \lor (i = (r + \frac{1}{2}) \cdot 2^k + 1)\). Hence, for any faulty processor \( l \), there are at most five processors \( i \) such that \( A_i^{(3)}(t) = i - 2^k \) and \( l \in [i - 2^k, i + 2^k] \). □

All that remains is to bound the communication in ‘long’ intervals that do not contain, and are not near, a faulty processor.
Lemma 17 Consider $k$ such that $\sqrt{n-1} < 2^k$. If $\text{Send}_{i,i+2^k}^{\text{ack}}$ occurs, then either there is a faulty processor in $[i, i+2^k]$ or $2^k > \sqrt{\frac{D(n-1)}{8\cdot \delta}}$.

Proof Assume that $\text{Send}_{i,i+2^k}^{\text{ack}}$ occurs. Since $\sqrt{n-1} < 2^k$, from Eq. (6) we know that processor $i + 2^k$ is to send an acknowledgement over an interval of length $2^{k+1}$ (if $2^{k+1} \leq n-1$). That is,

$$A^{(3)}_{i+2^k} \left( \frac{n-1}{2^{k+1}} \right) = (i + 2^k) - 2^{k+1} = i - 2^k.$$ 

Assume that there is no faulty processor in $[i, i+2^k]$. Every nonfaulty processor forwards the message immediately. Hence, processor $i + 2^k$ receives the message and starts executing at most $2^k \cdot \delta$ after processor $i$ started executing. After at most $\frac{n-1}{2^{k+1}}$ \text{TICK}_i$ events, processor $i + 2^k$ either sends its own length $2^{k+1}$ interval acknowledgement, or this acknowledgement is already redundant since it already performed some other $\text{Send}_{i,i+2^k}^{\text{ack}}$ such that $j \leq i - 2^k < i < i + 2^k \leq l$. From Axiom 2, this occurs after at most $\left( \frac{n-1}{2^{k+1}} + 1 \right) \cdot D$ since processor $i + 2^k$ started. Since every processor in $[i, i+2^k]$ is nonfaulty, it follows that $\text{Receive}_{i,i+2^k}^{\text{ack}}$ occurs after $\text{Send}_{i,i+2^k}^{\text{ack}}$, with $j' < l' \leq l$. Namely, $\text{Receive}_{i,i+2^k}^{\text{ack}}$ occurs after at most $2^k \cdot \delta + \left( \frac{n-1}{2^{k+1}} + 1 \right) \cdot D$ since processor $i$ started.

From Eq. (6) and since $\sqrt{n-1} < 2^k$ follows that $\text{Send}_{i,i+2^k}^{\text{ack}}$ occurs only after time $t_i \geq \frac{n-1}{2^k}$. From Axiom 2, this occurs at least $\frac{n-1}{2^k} \cdot D$ since processor $i$ starts executing. However, $\text{Send}_{i,i+2^k}^{\text{ack}}$ does not happen after $\text{Receive}_{i',i'+l'}^{\text{ack}}$ where $j' \leq i - 2^k < i < l'$. Since we assumed that $\text{Send}_{i,i+2^k}^{\text{ack}}$ does occur, it follows that

$$\frac{n-1}{2^k} \cdot D < 2^k \cdot \delta + \left( \frac{n-1}{2^{k+1}} + 1 \right) \cdot D$$

from which the claim follows.

We now use Lemmas 14–17 to compute the communication complexity.

Theorem 18 The communication complexity of implementation $A^{(3)}$ is $O(n \log(f + \frac{n}{D}))$.

Proof It is immediate that the communication complexity due to the data message and to the disconnection messages is $O(n)$. Therefore, we consider only acknowledgements.

Let $n_k$ be the number of $\text{Send}_{i,i+2^k}^{\text{ack}}$ events during the execution. The communication complexity due to acknowledgements is at most

$$C_{\text{ack}} \leq \sum_{k=0}^{\log(n-1)} n_k \cdot 2^k$$  \hspace{1cm} (7)

32
directly from Eq. (6) and since $\sqrt{n - 1}$ is a power of two, $n_k \leq \frac{n - 1}{2^n} \cdot 2$ holds.

By substituting this bound for $n_k$ in Eq. (7), we obtain

$$C_{AeR} = O(n \log(n)) \cdot (8)$$

The rest of the proof is needed to refine this bound. Let us first outline the proof. Lemma 15 gives tighter bounds of $n_k$ for $\frac{\sqrt{n - 1} \cdot \delta}{D} < 2^k < \frac{\sqrt{n - 1}}{2}$. Lemmas 16 and 17 give tighter bounds of $n_k$ for $\sqrt{n - 1} < 2^k \leq \frac{D \cdot (n - 1)}{8 \delta}$. We combine these tighter bounds with the simple bound of $\frac{(n - 1)}{2^n}$ for other values of $k$, and obtain the desired bound on the communication complexity.

Directly since $n_k \leq \frac{n - 1}{2^n} \cdot 2$, i.e., $n_k \cdot 2^k \leq (n - 1) \cdot 2$, it follows that:

$$\sum_{k=0}^{\log \sqrt{\frac{\sqrt{n - 1} \cdot \delta}{D}}} n_k \cdot 2^k \leq \sum_{k=0}^{\log \sqrt{\frac{\sqrt{n - 1} \cdot \delta}{D}}} 2(n - 1) = O\left(n \log \frac{n \delta}{D}\right)$$

$$\sum_{k=\log \sqrt{\frac{\sqrt{n - 1} \cdot \delta}{D}}}^{\log \sqrt{n - 1}} n_k \cdot 2^k \leq \sum_{k=\log \sqrt{\frac{\sqrt{n - 1} \cdot \delta}{D}}}^{\log \sqrt{n - 1}} 2(n - 1) = O(n)$$

$$\sum_{k=\log \frac{D \cdot (n - 1)}{8 \delta}}^{\log (n - 1)} n_k \cdot 2^k \leq \sum_{k=\log \frac{D \cdot (n - 1)}{8 \delta}}^{\log \frac{D \cdot (n - 1)}{8 \delta}} 2(n - 1) \leq 2(n - 1) \cdot \log \frac{8 \delta(n - 1)}{D} = O\left(n \log \frac{n \delta}{D}\right)$$

$$\sum_{k=\log \frac{D \cdot (n - 1)}{8 \delta}}^{\log (n - 1)} n_k \cdot 2^k \leq \sum_{k=\log \frac{D \cdot (n - 1)}{8 \delta}}^{\log \frac{D \cdot (n - 1)}{8 \delta}} 2(n - 1) \leq 2(n - 1) \cdot \log \frac{5f}{2} = O(n \log(f))$$

From Lemmas 16 and 17, if $\sqrt{n - 1} < 2^k \leq \frac{D \cdot (n - 1)}{8 \delta}$, then $n_k \leq 5 \cdot f$. Therefore, the following holds:

$$\sum_{k=1+\log \sqrt{n - 1}}^{\log \frac{D \cdot (n - 1)}{8 \delta} - 1} n_k \cdot 2^k \leq \sum_{k=1+\log \sqrt{n - 1}}^{\log \frac{2(n - 1)}{5f}} 5f \cdot 2^k + \sum_{k=\log \frac{2(n - 1)}{5f}}^{\log \frac{n - 1}{2^n}} n_k \cdot 2^k .$$

$$\sum_{k=1+\log \sqrt{n - 1}}^{\log \frac{2(n - 1)}{5f}} 5f \cdot 2^k \leq 5f \cdot \frac{2(n - 1)}{5f} \cdot 2 = O(n) .$$
From inequalities (12), (13) and (14), we obtain
\[
\log \sqrt[\log \sqrt{n - 1}]\frac{D}{8s} - 1 = \sum_{k=1+\log \sqrt{n - 1}}^{\log \sqrt{n - 1}} n_k \cdot 2^k \leq O(n) + O(n \log(f)) = O(n \log(f)). \tag{15}
\]

From Lemma 15, it follows that if \(\frac{\sqrt{n - 1} - 2\delta}{D} < 2^k < \frac{\sqrt{n - 1}}{2^x}\), then \(n_k \leq 2f \cdot \sqrt{n - 1} + 1\). Hence,
\[
\sum_{k=1+\left\lfloor \log \sqrt{n - 1} \right\rfloor}^{\log \sqrt{n - 1} - 1} n_k \cdot 2^k \leq \sum_{k=1+\left\lfloor \log \sqrt{n - 1} \right\rfloor}^{\log \sqrt{n - 1} - 2} 2f \cdot \sqrt{n - 1} + 1 \cdot 2^k = O(\sqrt{n} \cdot f \cdot \log(n)). \tag{16}
\]

From inequalities (7), (9), (10), (11), (15) and (16), we obtain
\[
C_{Ack} \leq \sum_{k=0}^{\log(n - 1)} n_k \cdot 2^k = O\left(n \log \frac{n^0}{D}\right) + O(n \log(f)) + O(\sqrt{n} \cdot f \cdot \log(n)). \tag{17}
\]

For \(f \geq n^\frac{1}{4}\) the claim follows from Equation 8, since \(C_{Ack} = O(n \log n)\), and in this case \(O(n \log(f)) = O(n \log(n))\). For the case that \(f < n^\frac{1}{4}\) the claim follows from Equation 17 since then \(O(\sqrt{n} \cdot f \cdot \log(n)) \leq O(n)\).

7 Conclusions

@speak of the meaning of \(f\) if there are no faults, or if the actual number \(A\) of actual faults is different than \(f\). In the paper, the lower message complexity bound for the adaptive protocol can be had when considering the value of \(f\) for which the message complexity gets a minimum, and this may be different than \(A\)—only if we expand the model a little and say that not all the deltas are the same. In addition, if there is at least one fault \(f\) may get into the time complexity too (check). In addition, the notion of \(f\), different than \(A\), may be useful in other papers or tasks. @@

We have observed that the actual delivery time in asynchronous bounded networks is much shorter than the known bound on the delivery time. We introduced a way to model and take advantage of that fact, and the notion of early termination for protocols in the asynchronous bounded network. Following [DRS], we observed that early termination is a form of distributed competitiveness.

The protocols presented ensure early-terminating detection of arbitrary failures in forwarding a message along a fixed route. The protocols are quite simple and need only finite memory. The penalty in message complexity is acceptable for most applications. The message complexity is at most \(O(n \log n)\), where \(n\) is the length of the path, compared to \(O(n)\) for the trivial protocol which cannot overcome faults other than those detectable by the link protocol. The message complexity of our adaptive detector is \(O(n \log(f + \frac{nD}{f}))\),
where $\frac{\hat{D}}{D}$ is the ratio between the actual delay and the apriori bound on the delay, and $f$ is the number of faults. Since usually $\frac{\hat{D}}{D} \ll 1$ and $f = 0$ (or $f = O(1)$), the communication complexity is nearly optimal. Theorems 5 and 7 may be used to achieve other trade-offs by different implementations of the design, some of which may be better in practical applications. For example, it is easy to keep the communication complexity optimal (i.e., $O(n)$) while still improving the time complexity from the $O(nD)$ of the trivial protocol to $(O(n\delta + \sqrt{n \cdot f \cdot D}))$.

Further work is needed to find the best communication complexity for early-terminating protocols, possibly generalizing our lower bound to hold for a general protocol. Let us list some additional open problems: generalizing our results to networks and rings (rather than a path); considering probabilistic protocols; dealing with clock drifts; and efficient handling of many messages. Additional further work is needed in order to understand the implications of the model for other tasks and, possibly, to generalize the model further. As mentioned in the introduction, some of this further research has meanwhile already taken place.

Acknowledgements

Special thanks to Oded Goldreich and Adrian Segall, who supervised the work of the first author, for their encouragement and help in obtaining the results and improving the exposition.

It is a pleasure to thank Hagit Attiya, Baruch Awerbuch, Gil Barzilai, Tsiyi Barzilai, Inder Gopal, Madan Gopal, George Grover, Radia Perlman, Ken Perry, and Moti Yung for helpful discussions and motivations for this work.

References


