SIAM J. COMPUT. © 2000 Society for Industrial and Applied Mathematics
Vol. 30, No. 4, pp. 1169–1196

EARLY DETECTION OF MESSAGE FORWARDING FAULTS[*]

AMIR HERZBERG[†] AND SHAY KUTTEN[‡]

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 δ of a message over a link is usually much smaller than the a priori known upper bound D on that delivery time. The main contribution of this paper is the way to model and take advantage of this observation. We introduce the notion of asynchronously early-terminating protocols, as well as protocols that are asynchronously early-terminating, i.e., time optimal in both worst case and typical cases. More precisely, we present a time complexity measure according to which one evaluates protocols both in terms of D and δ. 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 δ. Previous communication-efficient solutions were slow in the case where $\delta \ll D$. We observe that this is the most typical case.

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 acknowledgment 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.

Key words. real time, distributed algorithms, fault tolerance, competitive algorithms, network protocols, time adaptive

AMS subject classifications. 68W15, 68W10, 68W20, 94C15, 68R25

PII. S0097539796312745

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 for delivering

[*]Received by the editors December 2, 1996; accepted for publication (in revised form) October 13, 1999; published electronically September 20, 2000. Preliminary reports of parts of this work appeared in *Proceedings of the Ninth International Conference on Computers and Communication*, Tel Aviv, Israel, 1988 and *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, Edmonton, Canada, 1989, pp. 339–353.

http://www.siam.org/journals/sicomp/30-4/31274.html

[†]IBM Haifa Reserach Lab, IBM, 5th floor, 2 Weizmann Street, Tel Aviv, Israel, 52960 (amir@il.ibm.com).The research of this author was partially supported by grant 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.

[‡]Faculty of Industrial Engineering and Management, Technion, Haifa 32000, Israel (kutten@ie.technion.ac.il).

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., as used in [Tan81, BGJ$^+$85, MRR80]). For example, "first-in first-out" (FIFO) service for messages is then guaranteed, without the need for expensive hardware or for software intervention to restore 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 acknowledgments 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 acknowledgments*, 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 acknowledgment from neighbor v within D since u sent a data message to v, then either v or link (u, v) failed.

There are some failures which are not detected by the hop-by-hop acknowledgments mechanism. A simple example is a malicious failure, where v "intentionally" sends an acknowledgment without forwarding the data message toward the destination. A similar outcome may result without malicious intent, i.e., when a processor breaks down after sending the acknowledgment 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 acknowledgment* mechanism. This is an acknowledgment message which is sent from the destination to the source when the destination receives a data message. If the source does not receive an acknowledgment 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 δ_f denote the maximal transmission delay over a link in this execution, when not counting the worst f links. Moreover, $\delta_f \leq D$ 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, δ_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); some links just happened to be slower than others. For simplicity, we prefer to use δ_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 δ instead of δ_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 δ; 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 δ, rather than on the value of D.

We call a protocol *asynchronously early-terminating* if its time complexity is op-

TABLE 1
Parameters.

Parameter	Meaning	Typical value	Initially known?
n	Number of processors	< 20	Yes
f	Number of faulty processors and links	0 or 1	No
D	Bound on the delay on a nonfaulty link	1 sec.	Yes
δ	Actual delay on a nonfaulty link	10 msec.	No

timal for any selection of the number of faults f, of D, and, of δ. (We shorten it to *early-terminating*, when there is no ambiguity with the "early-stopping" synchronous Byzantine agreement protocols discussed in many papers [DRS86].) 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 δ are shown in Table 1.

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 δ. 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 when δ 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 [AADW94], 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 ratios). 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 δ. (Recall that we analyze the time complexity as a function of D and δ.)

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.

Our main contribution is the measure of time complexity as depending on D and δ, with the concept that one should strive to obtain time complexity bounds that depend as much as possible on δ rather than on D. Other contributions are communication-efficient early-terminating protocols. The *immediate-Acks protocol*,

TABLE 2
Protocols.

Protocol	Time	Communication	Remarks
ISO 8072/3, CCITT x.224	$O(nD)$	$O(n)$	Not locating
[Per88, section 3.5]	$O(nD)$	$O(n^2)$	Locating
End-to-end (section 4.1)	$O(nD)$	$O(n)$	Locating
Hop-by-hop (section 4.2)	$O(fD + n\delta)$	$O(n^2)$	Locating
Immediate-Acks (section 6.1)	$O(fD + n\delta)$	$O(n \log(n))$	Locating
Adaptive (section 6)	$O(fD + n\delta)$	$O(n \log(f + \frac{n\delta}{D}))$	Locating

presented in section 6.1, has $O(n \log n)$ communication complexity. We present in sections 6.3 and 6.4 an early terminating protocol which is adaptive, in the sense that the number of acknowledgment 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 message (acknowledgments). 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 an optimal time, early terminating 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 it 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 in 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, AAGMRS97, 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, AAGMRS97]. 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 (section 6) is based on the idea that a processor i can "piggyback" its acknowledgment on an acknowledgment of another processor j. This raises the question: How long should processor i wait for the acknowledgment of processor j before it gives up the idea of piggybacking (and sends its acknowledgment)? A similar problem was studied in [AAPS87, AS87, BYKWZ87, K88].

Our protocols can be viewed as being competitive (especially in the time complexity measure). Competitive distributed algorithms were studied further in later papers, e.g., [AKP92, AADW94].

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], which suggests a more detailed model, and gives an algorithm for agreement whose time depends mostly on δ, 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 with respect to 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 (which 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 sections 6.3–6.5, 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 [AE86, 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 ϕ (e.g., an acknowledgment) that was sent by processor $1 \leq j \leq n$ (the *sender* of ϕ) to another processor $1 \leq k \leq j$ (the *recipient* of ϕ). If k is not a neighbor of j, then message Φ 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 $k \leq j$: The protocol in processor i for $(k \leq i \leq j)$ interacts with the links by the events $\texttt{Send}_i^{k\leftarrow j}[\phi]$, and $\texttt{Receive}_i^{k\leftarrow j}[\phi]$. Similarly, in the case that k is larger than j: $\texttt{Send}_i^{j\rightarrow k}[\phi]$, $\texttt{Receive}_i^{j\rightarrow k}[\phi]$. 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 $\texttt{fail}_{i+1}$ event to i (i.e., the failure of the link from i to $i+1$) when more than $2D$ time passed without an acknowledgment 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* $\texttt{Receive}_i^{j\rightarrow k}[\phi]$ *occurs, then* $j < i \leq k$ *and for every* p *between* j *and* i*, previously* $\texttt{Receive}_p^{j\rightarrow k}[\phi]$ *and* $\texttt{Send}_p^{j\rightarrow k}$ *(as well as* $\texttt{Send}_j^{j\rightarrow k}$*) occurred. Similarly, if* $\texttt{Receive}_i^{j\leftarrow k}[\phi]$ *occurs, previously* $\texttt{Receive}_q^{j\leftarrow k}[\phi]$ *and* $\texttt{Send}_p^{j\leftarrow k}$ *(and* $\texttt{Send}_k^{j\leftarrow k}$*) 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 $\texttt{Send}_i^{j\rightarrow k}[\phi]$ (respectively, $\texttt{Send}_i^{j\leftarrow k}[\phi]$) event till the corresponding $\texttt{Receive}_{i+1}^{j\rightarrow k}[\phi]$ (respectively, $\texttt{Receive}_{i-1}^{j\leftarrow k}[\phi]$) event occurs, 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 every D time units. Namely, a "ticker" that sends interrupts every D time actually suffices for our protocols. We express this a little more formally in the following axiom.

Axiom 2. *For every processor* i*, a TICK event occurs exactly once every* D *time units.*

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 δ) or very slowly (e.g., even more than D). However, if a message is sent over a link at a time t and no acknowledgment 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 δ over nonfaulty links.

Axiom 3. *Assume that at time* t*, a* $\texttt{Send}_i^{j\rightarrow k}[\phi]$ *(*$\texttt{Send}_i^{j\leftarrow k}[\phi]$*) event occurs. Then, before* $t+\delta \leq t+D$*, if the link* $(i, i+1)$ *(respectively,* $(i-1, i)$ *is nonfaulty, then a* $\texttt{Receive}_{i+1}^{j\rightarrow k}[\phi]$ *(respectively,* $\texttt{Receive}_{i-1}^{j\leftarrow k}[\phi]$*) 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 instead make 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): acknowledgments 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 resilience 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, from Axiom 1 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 δ, 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 (unknown) number f of faulty processors, bound D on the delay, and actual delay δ 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 until 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 maximum 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 section 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 section 4.2. The protocol of section 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 ϕ is forwarded towards the destination (by $\texttt{Send}_i^{1\rightarrow n}[\phi]$ events). When the destination accepts the data message ($\texttt{Receive}_n^{1\rightarrow n}[\phi]$), it sends an acknowledgment backward ($\texttt{Send}_n^{1\leftarrow n}[Ack]$). Every processor $i < n$ checks whether $i+1$ is faulty. Namely, processor i expects to receive the acknowledgment ($\texttt{Receive}_i^{1\leftarrow n}[Ack]$) after $3(n-i)$ or less TICK_i events since $\texttt{Send}_i^{1\rightarrow n}[\phi]$. If neither the acknowledgment nor the disconnection message is accepted, then i disconnects link $(i, i+1)$. Processor i forwards ϕ 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 acknowledgment or the disconnection from i at most $3(n-(i-1))$ TICK_{i-1} events since forwarding ϕ. This means that nonfaulty processors will not be accidentally 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 acknowledgments. In the end-to-end detector, we use only one acknowledgment message, which signals the completion of the transmission. The idea is to use additional acknowledgments, 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 acknowledgment towards the source immediately upon receiving the message ϕ en route 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

```
Constants of processor $i$:
  $A_i$ ($integer$)): array of integer; (different for each protocol)
  (If $A_i(t) \neq \perp$, then after $tD$ since processor $i$ accepts $\phi$, it sends $Ack$ 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 $TICK_i$ events since start)
  $AckSet_i$ : set of intervals (init: $\emptyset$); (Intervals of $Ack$ which $i$ received or sent)
}
```

FIG. 1 *Design of forwarding faults detector: declarations.*

disconnects from $i+1$ if it does not receive the acknowledgment from the destination n after more than $3(n-i)$ $TICK_i$ events since i forwarded ϕ toward the destination. (In fact, for this specific protocol we could have written $2(n-i)$, but $3(n-i)$ is used for compatibility with protocols presented later.) In essence, i waits the time needed in the worst case for the data message to reach its final destination n and for an acknowledgment to arrive from n to i. Consider the case that $i+1$ never forwards the message to $i+2$. Intuitively, this can be detected by i (using another protocol in which $i+2$ sends an acknowledgment 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 acknowledgments from $k = i+2, i+3, \ldots, n$ after more than $3(k-i)$ $TICK_i$ events. Intuitively, this mechanism guarantees that every $3D$ time unit the message processes toward the destination over an additional link (if this link is faulty; otherwise traversing the link costs δ 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 section 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 are given below.

Different implementations are defined by different selections of values for the array $A_{node}(time)$. Basically, if $A_{node}(time) \neq \perp$, 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 section 4 are implemented by using $A_i(t) = \perp$, 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 acknowledgment

```
Algorithm for the source $i = 1$:
⟨B1⟩  On accepting $\phi$:
⟨B2⟩      {   $\texttt{Send}_1^{1\rightarrow n}[\phi]$;
⟨B3⟩          repeat WORKLOOP() until $done_i$;}
Algorithm for the destination $i = n$;
⟨C1⟩  On $\texttt{Receive}_n^{1\rightarrow n}[\phi]$ : { deliver $\phi$; $\texttt{Send}_n^{1\leftarrow n}[Ack]$; $done_n \leftarrow TRUE$;}
Algorithm for intermediate processors $1 < i < n$:
⟨D1⟩  On $\texttt{Receive}_i^{1\rightarrow n}[\phi]$:
⟨D2⟩      {   $\texttt{Send}_i^{1\rightarrow n}[\phi]$; (forward message)
⟨D3⟩          if $A_i(0) \neq \perp$ then
⟨D4⟩              { $\texttt{Send}_i^{A_i(0)\leftarrow i}[Ack]$ ;
⟨D5⟩                $AckSet_i \leftarrow \{[A_i(0), i]\}$;   }
⟨D6⟩          repeat WORKLOOP() until $done_i$;   }
Procedure WORKLOOP():{
⟨E1⟩  On $\texttt{Receive}_i^{j\leftarrow l}[Ack]$ such that
          $((\exists t)A_l(t) = j) \wedge ((\forall[j', l'] \in AckSet_i)(j < j') \vee (l' < l))$ :
⟨E2⟩  {  if $j = 1$ and $l = n$ then $done_i \leftarrow TRUE$;
⟨E3⟩          if $j < i$ then $\texttt{Send}_i^{j\leftarrow l}[Ack]$ ; (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 $(A_i\ (time_i) \neq \perp) \wedge ((\forall[j', l'] \in AckSet_i)A_i\ (time_i) < j')$ then
⟨F4⟩              { $\texttt{Send}_i^{A_i(time_i)\leftarrow i}[Ack]$ ; $AckSet_i \leftarrow AckSet_i \cup \{[A_i(time_i), i]\}$ }
⟨F5⟩          if $\exists l > i$ and $(\exists t)t < time_i - 3(l - i)$ such that $i \geq A_l(t) \neq \perp$ and
                  $(\forall[j', l'] \in AckSet_i)(A_l(t) < j' \vee l' < l)$ then DISCONNECT() ;}
⟨G1⟩  On $\texttt{fail}_i$ : DISCONNECT() ;
⟨H1⟩  On $\texttt{Receive}_i^{1\leftarrow j}[Disc_j]$ :
⟨H2⟩      {  $done_i \leftarrow TRUE$; $\texttt{Send}_i^{1\leftarrow j}[Disc_j]$ ; }
      }
Procedure DISCONNECT() ; (Disconnect processor $i$ from $i + 1$)
⟨I1⟩      {   $\texttt{Send}_i^{1\leftarrow i}[Disc_i]$ ;
⟨I2⟩          $done_i \leftarrow TRUE$; }
```

FIG. 2 *Design of a forwarding faults detector for processor i.*

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

The operation begins when the source (processor 1) accepts the message ϕ from the higher layer. Each processor $i > 1$ begins operating when receiving ϕ from $i - 1$. When the message is received, every processor i (except for the destination) forwards it to the next processor $i + 1$. If $i = n$, the message is 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 $A_i(0) \neq\perp$) 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 also issue *Ack* 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 $t \cdot D$ after 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 implementation; however, it proves very useful in the implementations of section 6.) Assume that $t_1 D$ time units after i started operating, it forwarded some *Ack* whose destination is some node k, and later, after $t_2 D$ time units, it is supposed to send an *Ack* to some $j > k$. (That is, $A_i(t_2) = j$.) Intuitively, this later *Ack* is no longer necessary, since the earlier *Ack* (that of time $t_1 D$) was already supposed to tell j (as well as k) that processor i received the message.

Thus, the *Ack* is sent only if its recipient $A_i(t)$ is farther from i than the most distant recipient of some previous *Ack* which i already forwarded. The set $AckSet_i$ holds all the intervals of *Ack* which i already forwarded. Processor i checks every $TICK_i$ event while in WORK_LOOP, if it should issue an *Ack*. The time since i began executing is approximated (in the variable $time_i$) by counting the number of $TICK_i$ events since i began executing.

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

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

Let us comment about nonredundant *Acks*. Note that for i to forward l's *Ack* it is unnecessary for i to learn anything new from that *Ack*. For example, it may be the case that i already received an *Ack* from $l + 1$, and thus i already "knows" that l received the messages. Still, it may be the case that this *Ack* 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 *Ack* may be nonredundant.

In every $TICK_i$ event, processor i checks for time-out of any expected *Ack*. A time-out is a failure of $i + 1$ to deliver the acknowledgment in time or to disconnect from $i + 2$. Note that if i does not receive an *Ack* on time from any processor $l > i + 1$ that is supposed to send an *Ack* 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 $\langle$F5$\rangle$, where the mentioned check is done. Processor i checks if there exists some processor $l > i$ that was supposed to send an *Ack* to be received by i. That is $A_l()$ equals some j that is either i or smaller than i. As mentioned above, such an *Ack* message should pass i. If such an *Ack* 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 *Ack* may have already been sent by l, but did not have enough time to reach i. That is why in instruction $\langle$F5$\rangle$ i also checks that there was enough time for the *Ack* to reach it if there were no disconnections. This is the meaning of the check on t in instruction $\langle$F5$\rangle$. 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 $A_i()$ 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 ϕ is forwarded immediately. If the check fails, procedure DISCONNECT() is used to disconnect processor i from $i+1$ (since processor $i+1$ or the link to it, or both, are faulty).

5.1. Resilience 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 δ.

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 δ after processor $i+1$ sets $done_{i+1} \leftarrow TRUE$.*

Proof. Processor $i+1$ sets $done_{i+1} \leftarrow TRUE$ only after $\mathtt{Send}_{i+1}^{1\leftarrow n}[Ack]$ or $\mathtt{Send}_{i+1}^{1\leftarrow j}[Disc_j]$. In any case, the corresponding `Receive` will occur at most after δ, by Axiom 3. Either message will cause $done_i \leftarrow TRUE$ unless it is already *TRUE*. $\square$

We now prove that until 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 $\mathtt{Send}_i^{1\rightarrow n}[\phi]$ immediately upon starting operation. From Axiom 3, processor $i+1$ accepts ϕ 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. $\square$

We now prove that if $i+1$ sends an *Ack* to i, then this *Ack* will not be ignored by (statement $\langle$E1$\rangle$ 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 $\mathtt{Send}_{i+1}^{j\leftarrow l}[Ack]$ occurs at time τ for $j \leq i$, $l \geq i+1$, while $done_i = done_{i+1} = FALSE$. Then, at time $\tau + \delta$, either $done_i = TRUE$ or $(\exists [j', l'] \in AckSet_i)(j' \leq j < l \leq l')$.*

Proof. From Axiom 3, event $\mathtt{Receive}_i^{j\leftarrow l}[Ack]$ will occur before time $\tau+\delta$. Assume that at time $\tau + \delta$, the following holds: $done_i = FALSE$. Therefore, statement $\langle$E1$\rangle$ is executed in i upon $\mathtt{Receive}_i^{j\leftarrow l}[Ack]$. However, since $i+1$ is nonfaulty, it also used statement $\langle$E1$\rangle$, $\langle$D3$\rangle$, or $\langle$F3$\rangle$ before $\mathtt{Send}_{i+1}^{j\leftarrow l}[Ack]$. Hence $(\exists t) A_l(t) = j$. This proves the claim, since if the other check of statement $\langle$E1$\rangle$ fails, then the claim holds trivially; and if both checks succeed, then statement $\langle$E4$\rangle$ is executed and the claim follows. $\square$

We now prove the core of the location property. This claim is still slightly weaker than the location property, since it deals only with the case that i sends the $Disc_i$ 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* $\texttt{Send}_i^{1\leftarrow i}[Disc_i]$.

Proof. The proof is by contradiction. Assume that $\texttt{Send}_i^{1\leftarrow i}[Disc_i]$ occurs at time τ. By definition, a $\texttt{fail}_i$ event does not occur. Therefore, statement $\langle$F5$\rangle$ in the design caused the $\texttt{Send}_i^{1\leftarrow i}[Disc_i]$ event at τ. Namely, while $done_i = FALSE$, a $TICK_i$ event occurs where for some $l > i$ and $t < time_i - 3(l-i)$ holds $i \geq A_l(t) \neq \perp$ and $(\forall [j', l'] \in AckSet_i)(A_l(t) < j' \vee l' < l)$.

The proof is based on considering the state of $i+1$ at $\tau - D$. From Lemma 1, if $done_{i+1}$ is $TRUE$ at $\tau - D$, then $done_i$ is $TRUE$ at $\tau - D + \delta < \tau$. Assume, therefore, that $done_{i+1} = FALSE$ at $\tau - D$. From Axiom 2, a $TICK_i$ event occurred at $\tau - D$ with $time_i$ less by one than at τ; namely, at time $\tau - D$ the following holds: $time_i = t + 3(l-i)$. (Recall that t, l, i are integers.) From Lemma 2, the maximum value of $time_i - time_{i+1}$ until $\tau - D$ is 2 (since $done_{i+1}$ is $FALSE$). Hence, the $TICK_{i+1}$ event in which $time_{i+1} \leftarrow t + 1 + 3(l - (i+1))$ is before $\tau - D$. Denote the time of this $TICK_{i+1}$ event by τ'. We derive a contradiction from considering the state of $i+1$ at τ'. 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 Ack from l (to some $A_l(t) \leq i$) or to disconnect from $i+2$. At τ', since $done_{i+1}$ is $FALSE$, then $(\exists [j', l'] \in AckSet_{i+1})(j' \leq A_l(t) \leq i < l \leq l')$. (Otherwise, processor $i+1$ would have invoked statement $\langle$F5$\rangle$.) An interval $[j', l']$ is added to $AckSet_{i+1}$ only by one of statements $\langle$F4$\rangle$, $\langle$D5$\rangle$, or $\langle$E4$\rangle$. Since $j' < i+1$, exactly before any of these statements is executed, a $\texttt{Send}_{i+1}^{j'\leftarrow l'}[Ack]$ occurs (see Figure 2). Hence, at (or before) $\tau - D$ a $\texttt{Send}_{i+1}^{j'\leftarrow l'}[Ack]$ occurs with $j' \leq A_l(t) < l \leq l'$. From this follows, using Lemma 3, that at τ there must be some $[x', y'] \in AckSet_i$ such that $x' \leq j' < l' \leq y'$. This contradicts the assumption that statement $\langle$F5$\rangle$ was invoked at τ.

We now deal with the case $l = i+1$. First, assume $t = 0$. Namely, $\texttt{Send}_{i+1}^{j\leftarrow i+1}[Ack]$ occurs when $i+1$ starts operating, i.e., at most δ after i starts operating. Hence, from Lemma 3, there will be some $[j', l'] \in AckSet_i$ such that $j' \leq j$ and $i+1 \leq l'$, after at most an additional δ (i.e., 2δ since i started). From Axiom 2 and the design, the value of $time_i$ at 2δ since i started is at most 2. Hence, when $time_i > 3$ statement $\langle$F5$\rangle$ is not invoked by i with $l = i+1$ and $t = 0$.

Assume, therefore, that $t > 0$ and $l = i+1$. Recall that $done_{i+1} = FALSE$ at τ'. Processor $i+1$ checks, at τ', the condition of statement $\langle$F3$\rangle$. If the condition holds, then statement $\langle$F4$\rangle$ is executed, i.e., a $\texttt{Send}_{i+1}^{j\leftarrow i+1}[Ack]$ occurs at τ', and the contradiction again follows from Lemma 3.

Assume that the condition of $\langle$F3$\rangle$ does not hold, namely, $(\exists [j', l'] \in AckSet_{i+1})$ $j' \leq j$. Then, previously one of statements $\langle$F4$\rangle$, $\langle$D5$\rangle$, or $\langle$E4$\rangle$ was executed, adding $[j', l']$ to $AckSet_{i+1}$. Since $j' < i+1$, exactly before any of these statements is executed, a $\texttt{Send}_{i+1}^{j'\leftarrow l'}[Ack]$ occurred (see Figure 2). Hence, at or before τ', a $\texttt{Send}_{i+1}^{j'\leftarrow l'}[Ack]$ occurs with $j' \leq j$. The contradiction follows from Lemma 3. ∎

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

THEOREM 5. *Every implementation* $A()$ *of the design such that* $A_n(0) = 1$ *is a resilient forwarding faults detector.*

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 $\texttt{Send}_n^{1\leftarrow n}[Ack]$ event. From Axiom 1, there will be no $\texttt{Receive}_1^{1\leftarrow n}[Ack]$ event. We assumed that $A_n(0) = 1$. Hence, after $time_1 = 3(n-1)$, statement ⟨F5⟩ sets $done_1 \leftarrow TRUE$ (unless $done_1 = TRUE$ already). But since processor 1 did not accept Ack from n, when it sets $done_1 \leftarrow TRUE$ it also did $\texttt{Send}_1^{1\leftarrow 1}[Disc_j]$.

We now prove the location property. Consider an execution where $Disc_i$ is sent by a nonfaulty processor j, and i is nonfaulty. The only event in which j sends, according to the design, a $Disc_i$ message, is by a $\texttt{Send}_j^{1\leftarrow i}[Disc_i]$ event. From Figure 2, a nonfaulty processor $j \neq i$ sends $Disc_i$ only if a $\texttt{Receive}_j^{1\leftarrow i}[Disc_i]$ occurred. Since $j \neq i$, it follows from Axiom 1 that $j < i$ and that before the $\texttt{Send}_j^{1\leftarrow i}[Disc_i]$ event, a $\texttt{Send}_i^{1\leftarrow i}[Disc_i]$ event occurred. From Lemma 4, either $i+1$ or the link $(i, i+1)$ or both are faulty. ∎

Since both detectors of section 4 are implementations of the design with $A_n(0) = 1$, we conclude with the following corollary.

COROLLARY 6. *The end-to-end and the hop-by-hop detectors are both resilient forwarding faults detectors.*

5.2. A bound on time complexity. 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 present later, as well as the hop-by-hop detector, are early-terminating.

Let $T_A(n, f, \delta, D)$ (respectively, $T_{\text{opt}}(n, f, \delta, D)$) 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 ϕ or an error is detected. We want a bound for the worst ratio $\frac{T_A(n,f,\delta,D)}{T_{\text{opt}}(n,f,\delta,D)}$ over every selection of n, f, δ, and D.

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. Likewise, we bound T_A by the sum of times required to reach from 1 to 2, from 2 to 3, and so on. 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. 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. 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 acknowledgment whose recipient is $j' \leq j$. 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 $F_{[j,l]}(A())$ and defined as follows:*

$$(1) \qquad F_{[j,l]}(A()) \overset{\text{def}}{=} \min_{t',j',l'} \left\{ \frac{(3(l'-j)+t') | (A_{l'}(t') = j') \wedge (j' \leq j < l \leq l')}{3(l-j)} \right\}.$$

Note that $A_i(t)$ is a set of intervals. The covering factor $F_{[j,l]}(C)$ for any set of intervals 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 $F(A())$ and is defined as follows:*

$$F(A()) \overset{\text{def}}{=} \max_{1 \le j < l \le n} F_{[j,l]}(A()). \tag{2}$$

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 τ_i denote the time when processor i received the message and entered the protocol, i.e., the time of $\texttt{Receive}_i^{1\to n}[\phi]$.

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 $Disc_j$, 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 τ the time when the $1 + 3 \cdot (i - i') \cdot F(A())^{th}$ event of the kind $TICK_{i'}$ occurs since i' received the message. From Axiom 2, time τ is not more than $3 \cdot (i - i') \cdot F(A()) \cdot D + 2D$ time units since processor i' received the message, namely,

$$\tau \le i' \cdot \delta + 8D \cdot F(A()) \cdot f_{i'} + 3(i - i') \cdot F(A()) \cdot D + 2D.$$

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 τ, 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 τ. The first case we consider is that at time τ holds $done_{i'} = TRUE$; later we deal with the other case.

Since at τ holds $done_{i'} = TRUE$, then previously either statement ⟨H2⟩ of procedure DISCONNECT() or statement ⟨E2⟩ was executed at processor i'. If procedure

DISCONNECT() was executed, then $\texttt{Send}_{i'}^{1\leftarrow i'}[Disc_{i'}]$ occurred already, and the claim for i follows. Similarly, if statement $\langle$H2$\rangle$ was executed, then $\texttt{Send}_{i'}^{1\leftarrow j}[Disc_j]$ occurred already for some j, and the claim for i follows.

If statement $\langle$E2$\rangle$ was executed, then statement $\langle$E3$\rangle$ was also executed, since $j = 1 < i'$. Hence, $\texttt{Send}_{i'}^{1\leftarrow n}[Ack]$ 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 τ it holds that $done_{i'} = FALSE$. Hence, at τ processor i' executes $\langle$F2$\rangle$, after which $time_{i'} = 1 + 3 \cdot (i - i') \cdot F(A())$.

By Definition 3, $F(A()) \geq F_{[i',i]}(A())$. By Definition 2, there are l, t such that $A_l(t) \leq i'$ and $i \leq l$ and

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

Hence, when processor i' executes $\langle$F5$\rangle$ at time τ, then

$$\begin{aligned} time_{i'} &= 1 + 3 \cdot (i - i') \cdot F(A()) \\ &> 3 \cdot (i - i') \cdot F_{[i',i]}(A()) \\ &= 3 \cdot (i - i') \cdot \frac{3 \cdot (l - i') + t}{3(i - i')} \\ &= 3 \cdot (l - i') + t. \end{aligned}$$

Hence, $t < time_{i'} - 3 \cdot (l - i')$ at time τ. Namely, at time τ, either $\texttt{Send}_{i'}^{j'\leftarrow l'}[Ack]$, $\texttt{Receive}_{i'}^{j'\leftarrow l'}[Ack]$ occurred with $j' \leq A_l(t) \leq i' < l \leq l'$, or processor i' executes procedure DISCONNECT() due to $\langle$F5$\rangle$. If processor i' executes procedure DISCONNECT(), then $\texttt{Send}_{i'}^{1\leftarrow i'}[Disc_{i'}]$ occurs and the claim follows. On the other hand, from Axiom 1, if $\texttt{Receive}_{i'}^{j'\leftarrow l'}[Ack]$ occurred, then $\texttt{Send}_{i}^{j'\leftarrow l'}[Ack]$ 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 δ. □

We deduce the following.

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 $A_l(t) = 1$. By definition, for every j, l, $F_{[j,l]}(A()) = 1$; hence, $F(A()) = 1$. The claim follows. □

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 2. This at most quadruples the complexities of the solutions, when applied to paths were $n - 1$ is not an even power of 2. 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 2.

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

$$F_{[j,l]}(A()) = \min_{j',l'} \left\{ \frac{l'-j \mid (A_{l'}(0)=j') \wedge (j' \le j < l \le l')}{l-j} \right\}. \tag{3}$$

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 $j' \le j < l \le l'$ and $\frac{l'-j}{l-j}$ is bounded by a constant.

A natural selection of $A()$ is to send acknowledgments 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 $2^k \le (n-1)$ 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. All acknowledgment intervals of length 2^k that start at a processor, i, end 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 $\frac{n}{4} < L < \frac{n}{2}$ is covered with a covering factor of less than 2 if and only if it is contained in one of the two acknowledgment intervals (described above) of length $\frac{n}{2}$. However, if it partially intersects with both, then only the end-to-end acknowledgment 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 acknowledgment 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 $r>0$ 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 $\frac{3}{4}(n-1)+1$. The acknowledgment 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 acknowledgment intervals of the first kind, e.g., intervals which include processor $\frac{n-1}{2}+1$. This selection of acknowledgment intervals ensures that every interval $[j,l]$ is covered by an acknowledgment interval which is not "much larger," as we now formalize.

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

$$A_i^{(1)}(0) \overset{\text{def}}{=} i - \max_{k\ge 0}\left\{ 2^k \,\middle|\, (\exists r\in\mathbb{N})\left((i=2^k\cdot r+1) \text{ or } \left(i=2^k\cdot\left(r+\frac{1}{2}\right)+1\right)\right)\right\}. \tag{4}$$

(No acknowledgment interval is defined for the case that $t>0$.)

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

Proof. From Corollary 9, it suffices to show that $F(A^{(1)}())$ is bounded by a constant. The proof is by showing that every interval $[j,l]$ is "covered" by an interval in $A^{(1)}()$ whose length is at most four times $l-j+1$.

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

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

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

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 acknowledgments. It is natural to classify such protocols by the way they handle the acknowledgments. An important subset, termed *oblivious protocols*, sends any acknowledgments they wish to send immediately, without delaying it. Similarly, if they receive an acknowledgment 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 $t \ne 0$.

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

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 there is a run where 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. Let an *interval* $c = [i,j]$ for $1 \le i < j \le n$ be the set $\{i, i+1, \ldots, j\}$. The *length* of interval c, denoted $L(c)$, is $j - i$. An *interval cover* C of an interval $[1,n]$ is a set of intervals that includes interval $[1,n]$.

The claim is that for an interval cover C, if $F(C)$ is a constant, then $L(C) = \Theta(n \log n)$, where $L(C) = \sum_{c \in C} L(c)$.

Intuitively, 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 such that $F(C)$ is a constant, the total* *length $L(C)$ is $\Omega(n \log n)$.*

Proof. Without loss of generality, assume $(\exists k) n-1 = 2(F(C))^k$. (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 $(F(C)-1)x+1 \leq (F(C)-1)x+x \leq F(C)x$ intervals in I_x with covering factor of $F(C)$. Consider the sets $I_{x(i)}$, where $x(i) = \frac{n-1}{2F(C)^{i+1}}$ for $1 \leq i \leq -1 + \log_{F(C)}(\frac{n-1}{2})$. Clearly $|I_{x(i)}| \geq \frac{n-1}{2}$. Thus, the observation implies that C must contain at least $\frac{|I_{x(i)}|}{F(C)x(i)} \geq \frac{(n-1)2F(C)^{i+1}}{2(n-1)F(C)} \geq F(C)^i$ intervals, the length of each at least $\frac{n-1}{2F(C)^{i+1}}$.

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. Let $C_0 \subset C$ include the interval $[n, 1]$. For $i = 1$ we have that $x(i) = \frac{n-1}{2F(C)^2}$. To cover $I_{x(1)}$ cover C must include $F(C)$ intervals, the length of each is at least $x(1) = \frac{n-1}{2F(C)^2}$. One of them can be the $[n, 1]$ interval, but additional $F(C)-1$ intervals are needed. Let $C_1 \subset C$ be a set of such additional intervals. Note that $C_1 \cap C_0 = \emptyset$. The total sum of C_1 is at least $\frac{(F(C)-1)(n-1)}{2F(C)^2}$. We continue to construct the C_i's inductively. For intervals in $I_{x(i)}$, C must include at least $F(C)^i$, where the length of each is at least $\frac{n-1}{2F(C)^{i+1}}$. As before, $F(C)^{i-1}$ such intervals are already included in the sets $C_0, C_1, C_2, \ldots, C_{i-1}$. Thus, we can construct a set $C_i \subset C$ such that $\forall_{1 \leq j < i} C_i \cap C_j = \emptyset$ and the cardinality of C_i is $F(C)^i - F(C)^{i-1}$. Thus, the total sum of C_i is at least $\frac{F(C)^{i-1}(F(C)-1)(n-1)}{2F(C)^{i+1}} = \frac{(n-1)(F(C)-1)}{2F(C)^2}$. Since we can construct $r(\log_{F(C)} n)$ such disjointed sets, the total length of C is $\Omega(\frac{n}{F(C)} \log_{F(C)} n)$. Since $F(C)$ is a constant, the theorem follows. $\square$

6.3. The ideas behind the adaptive implementations. The immediate-Ack implementation sends $r(n \log n)$ messages even in executions where there are no faults ($f = 0$) and δ is small. We next present implementations of the design which are early-terminating but use less messages when f and δ 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 acknowledgments upon beginning execution. Therefore, the immediate-Ack implementation of the design sends all of its acknowledgments in every execution (without failures).

However, if some of the processors *delay* issuing their acknowledgments, then it is possible that a delayed acknowledgment will become "redundant." For example, suppose that each of the processors delay all their acknowledgments by D except for 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 acknowledgments become redundant.

Obviously, the delay until the acknowledgments are sent increases the maximal time until the protocol terminates. For example, if the processors i s.t. $1 < i < n$ delay their acknowledgments 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 comes, therefore, at 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 acknowledgments sent is a function of the time complexity, which is lowest when there are no faults. The second adaptive implementation saves 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 long as possible, before sending intermediate acknowledgments (i.e., *Ack* from $l < n$ to $j \geq 1$). The longer the delay in sending an intermediate acknowledgment, the larger the hope that the acknowledgment from n to 1 will make the intermediate acknowledgment 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 $O(n \log(1 + \frac{n\delta}{D}))$ 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 $A^{(1)}()$. 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 acknowledgment is needed (although acknowledgments must be sent, since the number of faults is not known). Thus, a delay in any acknowledgment does not increase the time complexity in such executions. Finally, in executions with more than one fault, the acknowledgments over intervals of length 2 do not help, so any delay in them cannot increase the time complexity.

The adaptive detector sends the acknowledgments of the intervals of length 2 of $A^{(1)}()$, 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 δ), this achieves both early termination and optimal message complexities. The optimal message complexity is achieved, in this case, since the acknowledgment from processor n arrives at every other processor before it sends any acknowledgment of its own. Thus, all the other acknowledgments 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 acknowledgments 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 2. For $1 < i \leq n$ and $0 \leq t < n-1$, define $A_i^{(2)}(t)$ as follows:

$$(5) \qquad A_i^{(2)}(t) \overset{\text{def}}{=} \begin{cases} 1 & \text{if } i = n, \\ i - t & \text{if } (\exists k) t = 2^k \wedge (\exists r \in \mathbb{N})\ i = rt + 1, \\ i - t & \text{if } (\exists k) t = 2^k \wedge (\exists r \in \mathbb{N})\ i = (r + \frac{1}{2})t + 1, \\ \perp & \text{otherwise.} \end{cases}$$

The implementation of this subsection achieves communication complexity $O(n\log(\frac{n\delta}{D}))$ if $f = 0$. We do not prove this here, since it is a corollary of a theorem presented in what follows.

6.5. Saving acknowledgments. 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 acknowledgments 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., $O(n\log(f + \frac{n\delta}{D}))$. However, there is a bad scenario in which the complexity is $\Omega(nf)$. Indeed, in this scenario, the number of acknowledgments 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 acknowledgments 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 δ is small). However, we did not prove that the acknowledgment from n to 1 is delivered in such a time. In fact, for the implementation of the previous subsection, one can show a case where for $f = O(\log n)$ faults the time for delivering n's acknowledgment 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 acknowledgment 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 δ 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 acknowledgment to $n-4$. Thus, the next acknowledgment that $n-4$ is waiting for is a length 4 interval acknowledgment, expected to arrive after a time that is double that of the length 2 acknowledgment.

Carrying this argument further, when a length 2^i interval acknowledgment Ack_i arrives at its destination, j, the next acknowledgment expected by processor $j-1$ is a length 2^{i+1} interval acknowledgment Ack_{i+1}, that is expected in double the time. Even if both Ack_i and Ack_{i+1} arrive at j at the same time, processor j can delay 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_j}. Thus, the length $n-1$ interval acknowledgment from n to 1 (i.e., the end-to-end acknowledgment) can also be delayed 2^{f_j} without causing $j-1$ to detect the fault.

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

A natural selection is the set of intervals whose length is $\sqrt{n-1}$. Let us comment that using *only* this set, without using the shorter acknowledgments, we could obtain $O(n\delta + f\sqrt{n}D)$ 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" acknowledgments over "long" intervals, i.e., intervals whose length is *more* than $\sqrt{n-1}$. By sending acknowledgments 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 (5), are sent in order of increasing length. Namely, the *Ack* of interval of length l is sent after lD. The "long acks" are acknowledgments 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 lengths are *more* than $\sqrt{n-1}$.

Formally, the implementation of this subsection is presented in (6) below. For simplicity, we assume that $n-1$ is an even power of 2. For $1 < i \leq n$ and $0 \leq t < n-1$ define $A_i^{(3)}(t)$ as follows:

$$(6)\qquad A_i^{(3)}(t) \overset{\text{def}}{=} \begin{cases} 1 & \text{if } i = n, \\ i - \sqrt{n-1} & \text{if } (\exists r \in \mathbb{N}) i = r\sqrt{n-1} + 1, \\ i - \sqrt{n-1} & \text{if } (\exists r \in \mathbb{N}) i = (r + \frac{1}{2})\sqrt{n-1} + 1, \\ i - \frac{n-1}{t} & \text{if } (\exists k) t = 2^k \wedge (\exists r \in \mathbb{N}) i = r\frac{n-1}{t} + 1, \\ i - \frac{n-1}{t} & \text{if } (\exists k) t = 2^k \wedge (\exists r \in \mathbb{N}) i = (r + \frac{1}{2})\frac{n-1}{t} + 1, \\ i - t & \text{if } (\exists k) t = 2^k \wedge (\exists r \in \mathbb{N}) i = rt + 1, \\ i - t & \text{if } (\exists k) t = 2^k \wedge (\exists r \in \mathbb{N}) i = (r + \frac{1}{2})t + 1, \\ \perp & \text{otherwise.} \end{cases}$$

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

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, they send only $O(n \log(\frac{n\delta}{D}))$ 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 $A_{l'}^{(1)}(0) = j'$ 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 $A_{l'}^{(3)}(t') = j'$. Hence,

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

which completes the proof. □

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

LEMMA 14. *Let j, i be processors such that $0 < i - j < \frac{\sqrt{n-1}}{2}$. If $\texttt{Send}_i^{j \leftarrow i}[Ack]$ occurs, then either there is a faulty processor in $[i, i+\sqrt{n-1}\,]$ or in $i - j \leq \sqrt{n-1} \cdot \frac{2\delta}{D}$.*

Proof. Since $i - j < \frac{\sqrt{n-1}}{2}$ and from (6), there is some processor l such that $A_l^{(3)}(0) \neq \perp$ 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 acknowledgments immediately, then $\texttt{Receive}_i^{A_l^{(3)}(0) \leftarrow l}[Ack]$ 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 acknowledgment to j after $\texttt{Receive}_i^{A_l^{(3)}(0) \leftarrow l}[Ack]$. Hence, $\texttt{Send}_i^{j \leftarrow i}[Ack]$ may occur only before $\texttt{Receive}_i^{A_l^{(3)}(0) \leftarrow l}[Ack]$ occurs, i.e., before $2\sqrt{n-1} \cdot \delta$ since processor i started.

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

Lemma 14 shows that a "short acknowledgment" is issued only if it is one of the very short ones which are required since δ is not negligible, or if it is "close" to a faulty processor. We now bound the maximal number of "short" acknowledgments issued due to "close" faulty processors.

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

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

Lemma 15 bounds the communication due to acknowledgment intervals shorter than $\frac{\sqrt{n-1}}{2}$. In order to bound the entire communication complexity, we have to consider also longer acknowledgment intervals. We begin by bounding the maximal number of intervals either containing, or "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 (6), and since we assumed that $\sqrt{n-1}$ is a power of 2, it follows that if $A_i^{(3)}(t) = i - 2^k$, then $(\exists r \in \mathbb{N})(i = r \cdot 2^k + 1) \vee (i = (r+\frac{1}{2}) \cdot 2^k + 1)$. 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 $\texttt{Send}_i^{(i-2^k)\leftarrow i}[Ack]$ occurs, then either there is a faulty processor in $[i, i+2^k]$ or in $2^k > \sqrt{\frac{D\cdot(n-1)}{8\cdot\delta}}$.*

Proof. Assume that $\texttt{Send}_i^{(i-2^k)\leftarrow i}[Ack]$ occurs. Since $\sqrt{n-1} < 2^k$, from subsection 6.4 we know that processor $i+2^k$ is to send an acknowledgment over an interval of length 2^{k+1} (if $2^{k+1} \le n-1$). That is,

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

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 $\frac{n-1}{2^{k+1}} \mathit{TICK}_{i+2^k}$ events, processor $i+2^k$ either sends its own length 2^{k+1} interval acknowledgment, or this acknowledgment is already redundant since it already performed some other $\texttt{Send}_{i+2^k}^{j\leftarrow l}[Ack]$ such that $j \le i-2^k < i < i+2^k \le l$. From Axiom 2, this occurs after at most $(\frac{n-1}{2^{k+1}}+1)\cdot D$ since processor $i+2^k$ started. Since every processor in $[i, i+2^k]$ is nonfaulty, it follows that $\texttt{Receive}_i^{j'\leftarrow l'}[Ack]$ occurs at most $2^k \cdot \delta$ after $\texttt{Send}_{i+2^k}^{j\leftarrow l}[Ack]$ with $j' \le j < l \le l'$. Namely, $\texttt{Receive}_i^{j'\leftarrow l'}[Ack]$ occurs after at most $2\cdot 2^k \cdot \delta + (\frac{n-1}{2^{k+1}}+1)\cdot D$ since processor i started.

From (6) and since $\sqrt{n-1} < 2^k$, it follows that $\texttt{Send}_i^{(i-2^k)\leftarrow i}[Ack]$ occurs only after $\mathit{time}_i \ge \frac{n-1}{2^k}$. From Axiom 2, this occurs at least $\frac{n-1}{2^k}\cdot D$ since processor i starts executing. However, $\texttt{Send}_i^{(i-2^k)\leftarrow i}[Ack]$ does not happen after $\texttt{Receive}_i^{j'\leftarrow l'}[Ack]$ where $j' \le i-2^k < i < l'$. Since we assumed that $\texttt{Send}_i^{(i-2^k)\leftarrow i}[Ack]$ does occur, it follows that

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

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* $O(n\log(f+\frac{n\delta}{D}))$.

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 acknowledgments.

Let n_k be the number of $\texttt{Send}_i^{(i-2^k)\leftarrow i}[Ack]$ events during the execution. The communication complexity due to acknowledgments is at most

$$C_{Ack} \leq \sum_{k=0}^{\log(n-1)} n_k \cdot 2^k \tag{7}$$

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

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

$$C_{Ack} = O(n\log(n)). \tag{8}$$

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 $\frac{\sqrt{n-1}\cdot 2\delta}{D} < 2^k < \frac{\sqrt{n-1}}{2}$. Lemmas 16 and 17 give tighter bounds of n_k for $\sqrt{n-1} < 2^k \leq \sqrt{\frac{D\cdot(n-1)}{8\cdot\delta}}$. We combine these tighter bounds with the simple bound of $\frac{(n-1)\cdot 2}{2^k}$ for other values of k and obtain the desired bound on the communication complexity.

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

$$\sum_{k=0}^{\lceil\log\frac{\sqrt{n-1}\cdot 2\delta}{D}\rceil} n_k\cdot 2^k \leq \sum_{k=0}^{\lceil\log\frac{\sqrt{n-1}\cdot 2\delta}{D}\rceil} 2(n-1) = O\left(n\log\frac{n\delta}{D}\right), \tag{9}$$

$$\sum_{k=(\log\sqrt{n-1})-1}^{\log\sqrt{n-1}} n_k\cdot 2^k \leq \sum_{k=(\log\sqrt{n-1})-1}^{\log\sqrt{n-1}} 2(n-1) = O(n), \tag{10}$$

$$\begin{aligned}\sum_{k=\lceil\log\sqrt{\frac{D(n-1)}{8\delta}}\rceil}^{\log(n-1)} n_k\cdot 2^k &\leq \sum_{k=\lceil\log\sqrt{\frac{D(n-1)}{8\delta}}\rceil}^{\log(n-1)} 2(n-1)\\ &\leq 2(n-1)\cdot\log\sqrt{\frac{8\delta(n-1)}{D}} = O\left(n\log\frac{n\delta}{D}\right),\end{aligned} \tag{11}$$

$$\begin{aligned}\sum_{k=\lceil\log\frac{2(n-1)}{5f}\rceil}^{\log(n-1)} n_k\cdot 2^k &\leq \sum_{k=\lceil\log\frac{2(n-1)}{5f}\rceil}^{\log(n-1)} 2(n-1)\\ &\leq 2(n-1)\cdot\log\frac{5f}{2} = O(n\log(f)).\end{aligned} \tag{12}$$

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

$$\sum_{k=1+\log\sqrt{n-1}}^{\lceil\log\sqrt{\frac{D\cdot(n-1)}{8\delta}}\rceil-1} n_k\cdot 2^k \leq \sum_{k=1+\log\sqrt{n-1}}^{\lceil\log\frac{2(n-1)}{5f}\rceil} 5f\cdot 2^k + \sum_{k\lceil\log\frac{2(n-1)}{5f}\rceil}^{\log(n-1)} n_k\cdot 2^k. \tag{13}$$

Obviously,

$$\sum_{k=1+\log\sqrt{n-1}}^{\lceil \log \frac{2(n-1)}{5f} \rceil} 5f \cdot 2^k \leq 5f \cdot \frac{2(n-1)}{5f} \cdot 2 = O(n). \tag{14}$$

From inequalities (12), (13), and (14), we obtain

$$\sum_{k=1+\log\sqrt{n-1}}^{\lceil \log \sqrt{\frac{D\cdot(n-1)}{8\delta}} \rceil - 1} n_k \cdot 2^k \leq O(n) + O(n\log(f)) = O(n\log(f)). \tag{15}$$

From Lemma 15, it follows that if $\frac{\sqrt{n-1}\cdot 2\delta}{D} < 2^k < \frac{\sqrt{n-1}}{2}$, then $n_k \leq 2f \cdot \frac{\sqrt{n-1}+1}{2^k}$. Hence,

$$\begin{aligned} \sum_{k=1+\lceil \log \frac{\sqrt{n-1}\cdot 2\delta}{D} \rceil}^{(\log\sqrt{n-1})-2} n_k \cdot 2^k &\leq \sum_{k=1+\lceil \log \frac{\sqrt{n-1}\cdot 2\delta}{D} \rceil}^{(\log\sqrt{n-1})-2} 2f \cdot \frac{\sqrt{n-1}+1}{2^k} \cdot 2^k \\ &= O(\sqrt{n} \cdot f \cdot \log(n)). \end{aligned} \tag{16}$$

From inequalities (7), (9), (10), (11), (15), and (16), we obtain

$$C_{Ack} \leq \sum_{k=0}^{\log(n-1)} n_k \cdot 2^k = O\left(n \log \frac{n\delta}{D}\right) + O(n\log(f)) + O(\sqrt{n}\cdot f \cdot \log(n)). \tag{17}$$

For $f \geq n^{\frac{1}{4}}$ the claim follows from (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 (17) since then $O(\sqrt{n}\cdot f\cdot\log(n)) \leq O(n)$. ∎

7. Conclusions.

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 [AADW94], 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 $O(n\log(f + \frac{n\delta}{D}))$, where $\frac{\delta}{D}$ is the ratio between the actual delay and the a priori bound on the delay, and f is the number of faults. Since usually $\frac{\delta}{D} \ll 1$ 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.

Acknowledgments. 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, Tsipi Barzilai, Inder Gopal, Madan Gopal, George Grover, Radia Perlman, Ken Perry, and Moti Yung for helpful discussions and motivations for this work.

REFERENCES

[AAG87] Y. Afek, B. Awerbuch, and E. Gafni, *Applying static network protocols to dynamic networks*, in Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1987, pp. 358–370.

[AAPS87] Y. Afek, B. Awerbuch, S. A. Plotkin, and M. Saks, *Local management of a global resource in a communication network*, J. ACM, 43 (1996), pp. 1–19.

[AKP92] B. Awerbuch, S. Kutten, and D. Peleg, *Competitive distributed job scheduling*, in Proceedings of the 24th ACM Symposium on Theory of Computing, Victoria, Canada, 1992, pp. 571–580.

[AL89] H. Attiya and N. Lynch, *Time bounds for real-time process control in presence of timing uncertainty*, Inform. and Comput., 110 (1994), pp. 183–232.

[ADLS91] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer, *Bounds on the time to reach agreement in the presence of timing uncertainty*, in Proceedings of the 23rd ACM Symposium on Theory of Computing, New Orleans, LA, 1991, pp. 359–369.

[AE86] B. Awerbuch and S. Even, *Reliable broadcast protocols in unreliable networks*, Networks, 16 (1986), pp. 381–396.

[AG88] Y. Afek and E. Gafni, *End-to-end communication in unreliable networks*, in Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, Toronto, Ontario, Canada, 1988, pp. 131–148.

[AGH90] B. Awerbuch, O. Goldreich, and A. Herzberg, *A quantitative approach to dynamic networks*, in Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing, Quebec City, Quebec, Canada, 1990, pp. 189–204.

[AAGMRS97] Y. Afek, B. Awerbuch, E. Gafni, E. Rosen, and N. Shavit, *Slide—the key to polynomial end-to-end communication*, J. Algorithms, 22 (1997), pp. 158–186.

[AS87] Y. Afek and M. Saks, *Detecting global termination conditions in the face of uncertainty*, in Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing, Vancouver, Canada, 1987, pp. 109–124.

[BGJ+85] A. E. Baratz, J. P. Gray, P. E. Green, Jr., J. M. Jaffe, and D. P. Pozefsky, *SNA networks of small systems*, IEEE J. Selected Areas in Comm., 3 (1985), pp. 416–426.

[BS88] A. E. Baratz and A. Segall, *Reliable link initialization procedures*, IEEE Trans. Comm., 36 (1988), pp. 144–152.

[BYKWZ87] R. Bar-Yehuda, S. Kutten, Y. Wolfstahl, and S. Zaks, *Making distributed spanning tree algorithms fault resilient*, in Proceedings of the Fourth Symposium on Theoretical Aspects of Comptuer Science, Passau, Germany, Lecture Notes in Comput. Sci. 247, Springer-Verlag, Berlin, 1987, pp. 432–444.

[CCGZ88] C. T. Chou, I. Cidon, I. Gopal, and S. Zaks, *Synchronizing asynchronous bounded delay networks*, in Distributed Algorithms: Second International Workshop, J. van Leeuwen, ed., Lecture Notes in Comput. Sci. 312, Springer-Verlag, New York, 1988, pp. 212–218.

[CR87] I. Cidon and R. Rom, *Failsafe end-to-end protocols in computer networks with changing topology*, IEEE Trans. Comm., 35 (1987), pp. 410–413.

[DHSS84] D. Dolev, J. Halpern, B. Simons, and R. Strong, *Dynamic fault-tolerant clock synchronization*, J. ACM, 42 (1995), pp. 143–185.

[DRS86] D. Dolev, R. Reischuk, and H. R. Strong, *Early stopping in Byzantine agreement*, J. ACM, 37 (1990), pp. 720–741.

[AADW94] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts, *A theory of competitive analysis for distributed algorithms*, in Proceedings of the 36th IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1994, pp. 401–411.

[Fin79] S. G. Finn, *Resynch procedures and failsafe network protocol*, IEEE Trans. Comm., 27 (1979), pp. 840–846.

[GHM89] O. Goldreich, A. Herzberg, and Y. Mansour, *Source to destination communication in the presence of faults*, in Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, Edmonton, Canada, 1989, pp. 85–101.

[Gro82] G. Grover, *High Availability for Networks (HAPN)—Non-Disruptive VR Switching*, internal memorandum, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1982.

[GSK87] P. M. Gopal, R. A. Sultan, and B. K. Kadaba, *Performance Limits of APPN Architecture*, internal memorandum, IBM T. J. Watson Research Center, Yorktown Heights, NY, 1987.

[Her88] A. Herzberg, *Network management in the presence of faults*, in Proceedings of the Ninth International Conference on Computers and Communication, Tel Aviv, Israel, 1988.

[HK89] A. Herzberg and S. Kutten, *Efficient detection of message forwarding faults*, in Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing, Edmonton, Canada, 1989, pp. 339–353.

[K88] S. Kutten, *Optimal fault tolerant distributed construction of a spanning forest*, Inform. Process. Lett., 27 (1988), pp. 299–307.

[MRR80] J. M. McQuillan, I. Richer, and E. C. Rosen, *The new routing algorithm for the ARPANET*, IEEE Trans. Comm., 28 (1980), pp. 711–719.

[HM90] J. Y. Halpern and Y. Moses, *Knowledge and common knowledge in a distributed environment*, J. ACM, 37 (1990), pp. 549–587.

[Per88] R. Perlman, *Network Layer Protocols with Byzantine Robustness*, Ph.D. thesis, MIT Laboratory for Computer Science, Cambridge, MA, 1988.

[Pon91] S. Ponzio, *Network consensus in the presence of timing uncertainty: Omission and Byzantine failures*, in Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, 1991, pp. 125–138.

[SJ86] A. Segall and J. M. Jaffe, *Route setup with local identifiers*, IEEE Trans. Comm., 34 (1986), pp. 45–53.

[Sta87] W. Stallings, *Handbook of Computer Communication Standards*, Vol. 1, Macmillan, New York, 1987.

[Tan81] A. Tannenbaum, *Computer Networks*, Prentice–Hall, Englewood Cliffs, NJ, 1981.

[Zim80] H. Zimmerman, *OSI reference model—the ISO model of architecture for open systems interconnection*, IEEE Trans. Comm., 28 (1980), pp. 425–432.