EARLY TERMINATION IN UNRELIABLE COMMUNICATION NETWORKS

by

Amir Herzberg

Technical Report #650

September 1990
Early Termination in Unreliable Communication Networks

Amir Herzberg
Department of Computer Science
Technion, Haifa, Israel
e-mail: arnirh@techunix
September 30, 1990

Abstract

We propose a simple refinement of time complexity, which evaluates the time required for 'typical' executions as well as for 'worst case' executions. This refined measure is particularly useful for analyzing fault-tolerant protocols, which use bounds for the transmission delay. The refined time complexity is a function of both the actual delay over 'most' links, \( \epsilon \), and the bound on the delay, \( B \). Usually \( \epsilon \ll B \). Protocols whose time complexity depends on \( B \) as little as possible are called early terminating.

We present an early terminating protocol for performing queries in unreliable networks. This protocol may be used to locate the processor containing a given user, in order to communicate with this user. The protocol is resilient both to constantly changing topology and to arbitrary, possibly malicious, processor faults. Previous early terminating solutions assumed that the topology changes eventually stabilize, and also did not allow processor faults.

*Partially supported by grant No. 86-00301 from the United States - Israel Binational Science Foundation (BSF).
1 Introduction

Consider the following simple example. Two processors, $P_A$ and $P_B$, are connected by a communication link. Alice, logged on $P_A$, wishes to communicate with Bob, which is not logged on $P_A$. To find out if Bob is logged on $P_B$, processor $P_A$ should query processor $P_B$.

If the network is reliable, then $P_A$ may send “Is Bob there?” to $P_B$, which replies “Yes” or “No”. However, if $P_B$ may fail, then $P_A$ (and Alice) may be left waiting forever. It seems that in this case, after some time, $P_A$ should ‘time-out’ and inform Alice that Bob is unreachable at the moment.

Since $P_A$ should ‘time-out’ after some time, obviously the network is synchronous in the sense that the transmission delay is bounded. Hence, when Bob is not logged on $P_B$, then instead of sending “No”, processor $P_B$ could simply not send any reply to $P_A$. In the traditional analysis of time complexity, there is no difference between the two solutions (sending both “Yes” and “No” or sending only “Yes” and relying on time-out for “No”).

The main conceptual contribution of this paper is a simple refinement of the time complexity analysis, based on the following observations:

- In most executions, the number of faults $f$ is very small. Furthermore, most of the non-faulty processors remain connected through ‘non-faulty links’.
- The actual maximal delay over ‘non-faulty links’ in a given execution, denoted $\epsilon$, is usually much less than the a-priori bound on the delay, denoted $B$.

Hence, in the example above, sending “No” is much quicker in most executions over most networks. Such solutions, which make minimal use of time-outs, are called early-terminating. Since recently introduced in an earlier version of this work [Her88], this approach was adopted by several new works [ADLS90,GSTC90,HK89], which presented early terminating solutions to different problems.

We illustrate the new concepts by presenting an early-terminating query protocol for unreliable networks with arbitrary topology. The protocol is resilient both to arbitrary processor faults and to constantly changing topology (link failures and recoveries). Previous early terminating solutions [BGS88] allowed only a finite number of faults, and only link failures.

Organization. In the rest of the introduction, we describe our contributions and discuss related works. In §2 we present the model and in §3 we define the task. In §4 we present the protocol. In §5 we prove the properties of the protocol.

1.1 Measuring Termination Time

The transmission delay, or simply the delay, is the time since a message is sent over a link and until it is received at the destination. In most networks, the delay of each link depends on the load of the link and of the processors, the number of errors (and subsequent retransmissions), and many other factors. It is extremely difficult to predict the delay, and it changes during the execution.
Synchronous protocols assume an a-priori bound \( B \) on the transmission delay, usually to achieve fault tolerance. If \( B \) is too small, and the delay sometimes exceeds \( B \), then operative links will be considered as faulty; this is extremely undesirable, since each failure involves a huge overhead and degrades the connectivity of the network. Hence, and since the delay is so difficult to predict, the value of \( B \) is much higher than the actual transmission delay in almost every execution \( \epsilon \).

This motivates the asynchronous model, where the processors operate without a bound on the delay. The philosophy of this approach is that any bound \( B \) on the delay is much larger than the actual delay \( \epsilon \), and hence any protocol that depends on \( B \) has an unacceptable termination time, and therefore excluded from discussion. If there are no failures, then we know from \([\text{Awe85}]\) that given a protocol that depends on \( B \), we can produce an asynchronous protocol for the same task with polynomial overhead.

However, it is well known that many tasks, including queries, cannot be accomplished in the presence of processor faults, if there is no bound on the transmission delay \([\text{FLP85,KT88}]\). In order to solve such tasks, we have to use some bound \( B \) on the delay (although we expect that almost every transmission will take much less than \( B \)). Most works using this model, analyze as if the actual delay is equal to the bound on the delay, i.e., that the network is "completely synchronous". This does not distinguish between algorithms that depend heavily on the (extremely large) bound on the delay \( B \) and algorithms with minimal dependency on this bound.

We propose to analyze the time complexity as a function of both the bound on the delay \( B \) and the maximal actual delay \( \epsilon \). We expect the actual delay \( \epsilon \) to be usually much less than \( B \). To compare the time complexity of two algorithms, we mainly consider the dependencies on \( B \) (i.e., assume that \( B >> \epsilon \)). Basically, instead of forbidding the use of timeout (in the asynchronous model), we propose just to charge for the use of timeouts. We stress that processors are not aware of the value of \( \epsilon \).

Since the time complexity is a function of both \( \epsilon \) and \( B \), we can identify the protocols which "make minimal use" of the bound \( B \), namely protocols whose time complexity is minimal when \( \frac{\epsilon}{B} \) approaches zero. We call these early terminating protocols. In particular, asynchronous protocols are always early terminating, although often not fault tolerant.

In this paper, we present an early terminating protocol for queries in the presence of faulty processors. In subsequent works, this approach was used to deal with other tasks which are impossible in a completely asynchronous model \([\text{HK89,GSTC90,ADLS90}]\).

1.2 Specifying Requirements from Fault-Tolerant Protocols

Many tasks which are possible in reliable networks, become impossible if failures are possible. For example, it is impossible to send messages to a processor which is constantly disconnected. Hence, we must make some assumptions regarding the faults, or relax the requirements from the tasks so that they may be fulfilled.

---

1 If the faults consist only of link failures, then it is possible to extend the asynchronous model slightly by assuming that after finite (but unbounded) time, either the message is delivered or a failure is detected; this is the dynamic networks model of \([\text{AE86,AAG87}]\). This is not useful in our case, since we wish to deal with faulty processors as well.
The two main (known) formal approaches to unreliable networks take radically different attitudes towards failures. In the eventual-stability approach [Gal76, AAG87], failures are assumed to cease at some (unknown) point in time. This, of course, is not meant literally, but it is rather assumed that at some point in the future an entire execution of some task can be completed within a period containing no failures. In the eventual-connectivity approach [AE86], the only restriction on the nature of failures is that they do not disconnect the network forever; alternatively, the protocol is required to be successful only for the eventually connected component of the network.

However, how do we define the query task under either of these 'eventual' approaches? The query protocol should terminate either with the reply or with an indication that the reply is not available. If we assume that the network is eventually stable then the query protocol should not respond until the network reached this eventual state, but it is impossible to detect that the network reached stability. On the other hand, if the network is eventually connected, it is unrealistic to require the protocol to wait until every processor replied. Furthermore, the protocol cannot distinguish between processors which are permanently disconnected and processors which are 'eventually connected'; therefore it is impossible to require that the query will terminate after all the 'eventually connected' processors replied.

Furthermore, neither of the 'eventual' approaches is reasonable when processors may be faulty. If some processor stops participating at some moment, it cannot be ignored, since it may reconnect and respond. A faulty processor may, therefore, prevent the protocol from making progress.

In this paper, we present an alternative approach\(^2\) to defining requirements in an unreliable network. Basically, we require that when the query terminates, replies were accepted from any processor which is connected to the source of the query by a path which did not fail during the execution. Note that termination time is guaranteed and bounded, regardless of the faults. Similar definitions may be given to most tasks, and allow solutions resilient to processor faults and repeated link failures and recoveries.

### 1.3 The Query Protocol

The query task is a more general form of the example given in the beginning of the introduction. One processor, called the source \(s\), has some query. When a processor is given the query, it may or may not know a reply. If some processor knows a reply, then a reply should be delivered to the source \(s\). Otherwise, if no processor knows a reply, then the protocol should inform the source.

Query protocols are used in many networks. For example, queries are used in the network layer of SNA/APPN [BGJ*85], to locate the processor where the desired user is connected.

Our objective is to find an early terminating query protocol. It is trivial to solve queries in \(O(d_{max}B)\), where \(d_{max}\) is a bound on the dynamic diameter of the network. Namely, the source 'floods' the query, and any processor which knows a reply will flood it. The source terminates when receiving a reply or, by a timeout mechanism, after \(2d_{max}B\).

---

\(^2\)This approach has been extended and generalized in [AGH90].
We present an early terminating query protocol, which terminates in time \( O(fB+d_{\text{max}}\epsilon) \), where \( f \) is the number of faults. The protocol is based on the third connectivity test protocol (CT-3) in [Seg83]. However, protocol CT-3 is not fault tolerant, while our protocol is resilient to constantly changing topology (link failures) and to arbitrary processor faults.

Previous early terminating query protocols assumed a more benign environment. The echo, or PIF, protocol of [DS80,Cha82,Seg83], and the mobile user protocol of [AP90], both work on reliable networks without any faults. The protocol of [BGS88] deals only with link failures, and terminates only when failures eventually cease. On the other hand, these protocols use bounded sized messages. Our protocol is designed for a single query, and in order to use it for many queries, one has to append a query number field to each message. The length of this field is unbounded as a function of the size of the network, although it is only a logarithm of the number of queries.

Most of the messages required by the protocol, \( O(|E|^2) \), are sent by a topology learning procedure. This procedure is not needed if a previous query was executed; in these cases the message complexity is only \( O(|E|\cdot|V|) \). This topology learning procedure may be used as an early terminating protocol for topology learning, in these applications where the termination of topology learning is important.

Our topology learning protocol may be compared with the source-target-routing protocol presented in [DMP82]. To find a single non-faulty route between the source \( s \) and the target \( t \), the source-target-routing protocol requires that the number of routes between \( s \) and \( t \) would be more than 3 times the number of faulty processors. Our protocol requires only a single non-faulty route, and has lower time and message complexities.
2 Model

The communication network contains $n$ processors and $m$ (bidirectional) communication links between pairs of processors. The set of processors is denoted $V$ and the set of links is denoted $E \subseteq V \times V$; we assume that each processor has a distinct identity. Namely, the set of processors $V$ and the set of links $E$ define a simple graph $(V, E)$.

For consistency with most works dealing with dynamic networks, we consider reliable links, with the following events (see [BS88] for the details):

- **send**: The processor sends a message over the link.
- **receive**: The processor receives a message from the link.
- **fail**: The processor is informed about a failure in the link. Messages already sent from the other end but not received so far, are lost.
- **recover**: The processor is informed that the link recovered from failure. Between the failure and the recovery, messages are not sent or received via the link.

The major deviation of our model from the standard dynamic networks model, is the addition of the following synchronization assumptions. For simplicity of exposition (only), we use global time terminology and assume that all of the clocks have the same rate. We model the clocks by a special event $T_v$ which occurs in processor $v$ once every $B$. Formally, axiom 1 guarantees that the clocks have rate $B$.

**Axiom 1 (tick)** For every processor $v$, a $T_v$ event occurs exactly every $B$ time units.

Note that we assumed only that the clocks operate at the same rate, and not that all ticks occur exactly at once. However, the lag, i.e. the time since $T_v$ until $T_v$, is obviously at most $B$.

**Axiom 2** Assume that processor $u$ sent message $\phi$ over link $(u, v)$ at time $t$. Then before $t + B$, either $(u, v)$ fails or $v$ receives $\phi$. Furthermore, if $(u, v)$ is non-faulty, then $v$ receives $\phi$ before $t + \epsilon$.

The protocol should operate for an arbitrary and unknown selection of any number $f$ of faulty processors and links. Faulty processors may behave in an arbitrary manner. The program at each processor $v$ accepts as parameters the identity of the processor $v$ and a list of identities of its neighbors $N_v = \{u \mid (v, u) \in E\}$, where $N_v$ contains all of the non-faulty neighbors of $v$, and some of the faulty neighbors of $v$.

A path is non-faulty if all of its processors and links are non-faulty. We assume that all non-faulty processors are connected by non-faulty paths.

To allow processors to verify the validity of messages relayed to them, we assume the following primitive $valid$, which may be implemented using cryptographic signatures.

---

Footnote: In fact, our results do not require reliable communication over the links, i.e. we do not require that a data link control protocol [BS88] be used in a lower layer.
Axiom 3 (Validity) Let $\text{type}^*(N)$ be a message consisting of the three fields $\text{type}$, $v$ and $N$. Then $\text{valid}(\text{type}^*(N))$ is true if and only if $v \in V$ and $v$ sent $\text{type}^*(N)$ and $(\forall u \in N)((v, u) \in E)$.

The valid primitive checks three things: that the message really was sent by $v$, that $v$ is in the network, and that the parameter $N$ is a subset of the actual neighbors of $v$. In our protocol, this last check is important to prevent a faulty processor $v$ from flooding in the network messages $\text{type}^*(N)$, where the $N$ field is extremely long, much more than the list of the neighbors of $v$. This may increase the communication complexity.

Complexity Measures

Essentially, we use the traditional worst-case complexity measures: the communication complexity is the number of messages transmitted, of length $\log(s)$ or less, and the time complexity is the maximal time since the protocol starts in a non-faulty processor and until it terminates in that processor. However, the complexities are analysed as functions of both $\epsilon$ and $B$, as well as of $n$, $m$ and $f$ (the number of faults).

3 The Tasks

3.1 Query

We begin with informal description of the query task. The protocol starts in the source $s$, with some query $q$ (say, a user name). Each processor $v$ computes a boolean response as a function of the query $\text{response}_v(q)$ (say, true if the user is logged on that processor). If some non-faulty processor, which is connected to the source, responds true, then the protocol in the source terminates with an identity of some processor that responded true. Otherwise, the protocol terminates with NoReply. In any case, we require that the termination time will be bounded.

Definition 3.1 A protocol $P$ performs query if in every execution of $P$, started from some non-faulty processor $s$ at $t_Q \geq 0$, the following hold:

Termination There is some bound $T$ such that $P$ produces a response $r$ at $s$ at most at $t_Q + T$.

Correctness: Suppose $P$ terminates in $s$ with response $r$. Then:

Safety Either $r \in V$ or $r = \text{NoReply}$. If $r \in V$, then either $r$ is faulty or $\text{response}_v(q) = \text{true}$.

Completeness If there is a non-faulty path which emanates from $s$ to a processor $v$ such that $\text{response}_v(q) = \text{true}$, then $r \neq \text{NoReply}$.

3.2 Topology Learning

Our query protocol uses, as a procedure, a topology learning protocol. A single application of the topology learning procedure suffices for several executions of the query protocol. We now define the topology learning task, since it may be used for other applications as well.
In a static network, the meaning of topology learning is obvious. When the topology may change during the execution of the protocol, there are several variations of the topology learning problem (cf. [SG89, HS89]). Our definition takes the following approach: the protocol should give its estimate of the topology after *bounded time*. This time should not depend on the topology changes.

Obviously, we cannot require the topology estimate to precisely reflect the actual topology. However, the topology estimate should be 'useful', namely it should include every path from the source which was non-faulty since the protocol 'began'. The protocol 'begins' whenever a (non-faulty) processor accepts any message associated with the protocol; without loss of generality, assume that this is at time $0$.

**Definition 3.2** A protocol $P$ performs topology learning if in every execution of $P$, first started at some non-faulty processor at time $0$, the following hold:

**Termination** Let $v$ be a non-faulty processor in which $P$ starts at time $t_v \geq 0$. There is some bound $T$ such that $P$ terminates at $v$ before $t_v + T$.

**Correctness** Suppose $P$ terminates in a non-faulty processor $v$ with output estimate $(V_v, E_v)$. Then:

- **Safety** The estimate $(V_v, E_v)$ is a subset of the actual topology, namely $V_v \subseteq V$ and $E_v \subseteq E$.
- **Completeness** If there is a non-faulty path in $(V, E)$ emanating from $v$, then the path appears in $(V_v, E_v)$. 


4 Protocols

4.1 Informal Description

The topology learning protocol and the query protocol are both based on three mechanisms: flood, termination detection, and remote time-out, as described in the rest of this section. The formal code is presented in figures 1 to 3, and explained in §4.2.

The topology update protocol produces a graph \((V_v, E_v)\), which is the ‘topology estimate’. This graph is read by the query protocol. Initially, the estimate is only \(v\) itself and its neighbors. Namely, initially \(V_v = \{v\} \cup N_v\) and \(E_v = \{v\} \times N_v\). This graph is directed.

4.1.1 Operation without Failures

If there are no faults, the topology learning protocol is the same as protocol CT-3 of [Seg83], except for two small changes described below. For the sake of self-containment, we describe the entire modified protocol and not just the modifications. Each processor \(v\), upon receiving the first message of the topology update protocol, floods its neighbors list \(N_v\). Whenever a neighbors list \(N_u\) is received at \(v\), processor \(v\) adds to \((V_v, E_v)\) the directed edges from \(u\) to the processors in \(N_u\).

The termination of the topology learning is detected in \(v\) after it receives the neighbor lists \(N_u\) from every \(u\) to which there is a path in \((V_v, E_v)\).

The differences between the operation described above and CT-3 are needed only to deal, later, with faults. One change is that we use directed edges, to ensure that if there is a link from \(u\) to \(w\), then \(w\) indeed appeared in the neighbor list sent by \(u\). The other change is that CT-3 terminates after receiving the neighbor lists from every processor in \(V_v\). If failures occur, as we show later, the neighbor list may be never received from some processors in \(V_v\).

The operation of the query protocol, when there are no faults, is even simpler. The source begins by starting the topology learning protocol, and flooding a message containing the query. Whenever a processor receives the query, it floods a reply. The source terminates when it receives the replies from every \(u\) to which there is a path in \((V_v, E_v)\). This completes the description of the non-resilient version of the protocols.

4.1.2 Dealing with Faults

The protocols are enhanced as follows to deal with faults. Whenever a failure is detected in a link or a processor, this link or processor is removed from \((V_v, E_v)\). Therefore, termination may occur while the reply was not received from some processors in \(V_v\) which are not reachable from \(v\) in \((V_v, E_v)\). For this purpose, messages announcing such faults are flooded in the network whenever the faults are detected.

We now come to the main idea of the protocol, which is a remote time-out technique. This technique may be useful for other tasks as well, and indeed was used also in a subsequent work

\footnote{Flood is a widely used primitive protocol, explained in §4.1.3, and suggested (in a slightly simpler form) in [Seg83]. Basically, this protocol sends a message to every processor in the network.}
The remote time-out procedure detects faults in a neighbor or a link to the neighbor. This procedure essentially checks that the response from every processor \( u \in V_v \) is received 'in time'. Namely, if \( u \) is \( l \) hops from \( v \) in \((V_v,E_v)\), then the reply should reach \( v \) within \( 3IB \) since \( v \) entered the protocol. The factor of 3 is needed to allow for the time it takes the message to reach \( u \), for the time it takes the reply to reach \( v \), and the lag of up to \( B \) between two different clocks.

If the reply from \( u \) is not received within \( 3IB \), then either the link \((v,u)\) is faulty or \( u \) is faulty. Otherwise, the remote time-out procedure in \( u \) would have already detected the failure and flooded a disconnection message. (Recall that \( u \) starts the protocol at most \( B \) after \( v \), if the link between them is non-faulty.)

Thus, the remote time-out procedure ensures that after \( 3IB \), every processor connected to \( v \) in \((V_v,E_v)\) by a path of length \( l \) or less has already replied. This suggests the following method to detect termination. The idea is that \( v \) should terminate after it received replies from every processor reachable from \( v \) in \((V_v,E_v)\), since afterwards it may not receive additional replies. The replies are either the responses to the query, or the neighbor lists in the topology learning protocol.

We now give some intuition to the time complexity of \( O(\pi e + fB) \). The flood mechanism ensures that processors which have a non-faulty path to the source, receive the message within \( O(\pi e) \). The remote time-out mechanism, operating in the non-faulty processors adjacent to faulty components, will either disconnect from the component or receive all the acknowledgements from it within additional \( O(fB) \).

### 4.1.3 Failure-Resilient Flood

It remains to describe the flood procedure, which basically allows a processor to send a message to the entire network. This procedure is a simple extension of the PI protocol from [Seg83]. For the sake of self-containment, we now recall this mechanism. Every processor \( v \), which receives a flood message for the first time, forwards this message to all of its neighbors. Each flood is uniquely identified, therefore \( v \) will not forward the same flood several times (upon receiving it again).

The only modification we made to this 'standard' flood mechanism is the addition of a check, which deals with the case where a faulty processor \( u \) uses the same flood identifier for more than one message. When a non-faulty processor \( v \) receives the two different messages with the same flood identifier from \( u \), then \( v \) detects that \( u \) is faulty. To allow the other processors to detect that \( u \) is faulty, processor \( v \) also floods the two different messages with the identical flood identifier. However, processor \( v \) does not flood additional messages with the same identifier, to prevent bandwidth waste. Hence, every processor \( v \) sends and stores at most two messages per each flood identifier.

### 4.2 Program

In this subsection, we present and explain the code of the protocols. The code, for a generic processor \( v \), is presented in figures 1 to 3. The query protocol consists of all three figures; the code of the topology learning protocol does not include figure 3. The declarations of the variables are presented in figure 1.
Variables at processor $v$:

Note: Names with hats are used for the query protocol (e.g. $\hat{FE}_v$).

- $N_v$: The neighbors known to $v$.
- $FE_v$: Faulty edges known to $v$. Initially empty.
- $FV_v$: Faulty processors known to $v$. Initially empty.
- $(V_v^*, E_v^*)$: Processors and links known to $v$ (even if in $FE_v$, $FV_v$). Initially empty.
- $mode_v$: Either Idle or Work. Initially Idle.
- $time_v$: The number of $B_v$ events since $v$ started. Initially zero.
- $Replay_v$: An array, containing the replies of the different processors. Initially all entries are $\lambda$.
- $flooded_v$: Used flood identifiers, i.e. $(\psi, u)$ s.t. $\text{flood}^u(\psi, \phi)$ was called.
- $contents_v$: An array, containing the contents of flooded messages (indexed by the identifiers).

Messages

- $T^u(N)$: Local topology of processor $u$.
- $F^w(u)$: Failure of the link from $u$ to $w$. The query protocol uses $F^w(u)$.
- $q^u()$: A query $q$ issued from $s$.
- $R^v()$ (or $R^s()$): A positive (negative) reply from $u$ to the query.

Figure 1: Protocol declarations.

For simplicity, the two protocols are presented separately. Many procedures and variables have the same function in both protocols. In these cases, we use the same name for both protocols, distinguishing the names of the query protocol by putting a hat over them. For example, the protocols are started by the event $\text{Start}$ (for the topology learning) or $\text{Start}$ (for the query).

The protocols do not directly maintain $(V_v^*, E_v^*)$. Instead, the topology learning builds the graph $(V_v^*, E_v^*)$, which contains every link and processor known (even after they fail). To compute $(V_v^*, E_v^*)$ from $(V_v^*, E_v^*)$, both protocols remove faulty processors and links. The topology learning protocol keeps faulty processors in the list $FV_v$ and faulty edges in the list $FE_v$. The query protocol uses $(V_v^*, E_v^*)$, $FV_v$ and $FE_v$ maintained by the topology learning protocol, and also maintains additional lists of faulty processors ($\hat{FV}_v$) and edges ($\hat{FE}_v$).

The only interaction between the two protocols is when the query protocol reads the values of $(V_v^*, E_v^*)$, $FV_v$ and $FE_v$ which were set by the topology learning protocol. This allows us to analyze the topology learning protocol completely independently of the query protocol, and to analyze the query protocol with minimal reference to the topology learning mechanism.
Remark: the code does not refer to link failure or recovery events. This is since the remote time-out mechanism effectively removes the need for the data link reliability. We included link failure and recovery in the model just to be consistent with most other works about dynamic networks.
On Start: 

- If mode = Idle then { mode = Work; topo(N, T'(N)); }
- On receive(T(N)) s.t. valid(T(N)) : topo(N, T'(N));
- On receive(F(w)) s.t. valid(F(w)) : efail(w, F(w));

On Tv : if mode = Work then

- increment time;
- For every v = Uo, Ul, ... , Uk ∈ V
  where (λ < j ≤ k)((uj, uj+1, uj) ∈ E - FEv)
  and k ≤ tmax and Reply(u) = λ
  do: efail(u, F(u));
- if (Reply(u) ≠ λ for every u reachable from v in (V себя - FV, Ev - FEv))
  then Ready; (Remote time-out)
- (Termination detection)

Procedure flood(ψ, φ);

- if mode = Idle then { mode = Work; topo(N, T'(N)); }
- if u ∈ FV then return;
- if (ψ, u) ∈ flooded then (Ignore flood from faulty proc. to prevent waste.)
- \{ if contents(ψ, u) ≠ φ then (Ignore extra copies)
  - send φ to all neighbors;
  - add u to FV;
- return;
- send φ to all neighbors;
- return;

Procedure efail(u, φ);

- add (u, φ) to FEv;
- flood(φ, φ);

Procedure topo(N, φ);

- flood(T, φ);
- Reply(u) = true;
- add u and N to V;
- for every x ∈ N do: add (u, x) to Ev;

Figure 2: Topology Learning Algorithm at v.
On $\text{Start}(q)$ ($v = s$): call $\text{query}^*(q, q^*)$;

On $\text{receive}_v(q^*)$ s.t. $\text{valid}(q^*)$: $\text{query}^*(q, q^*)$;

On $\text{receive}_v(Y^*)$ s.t. $\text{valid}(Y^*)$: $\text{reply}^*(\text{true}, Y^*)$;

On $\text{receive}_v(W^*)$ s.t. $\text{valid}(W^*)$: $\text{reply}^*(\text{false}, W^*)$;

On $T_v$: if $\text{mode}_v = \text{Work}$ then

Figure 3: Query Algorithm at $v$. 

14
5 Analysis

We begin the analysis by proving several properties of the flood mechanism. Then we show termination and later correctness. Last, we analyze the complexities.

5.1 The Flood Mechanism

We begin by investigating the flood mechanism, which is used by both protocols. As described in §4.1.3, the flood procedure is a simple enhancement which adds fault-tolerance to the well known Propagation of Information protocol (PI), presented in [Seg83].

One notable difference between the two versions is that in [Seg83], flood was considered as a stand-alone protocol. As such, flood was invoked whenever a processor received a message. In our case, this property is not immediate, since flood is invoked only as a procedure of the other protocols. However, the interaction between our protocols and their flood procedure has the following similar property. Namely, flood is always invoked (with the same parameters) whenever a non-faulty processor receives a valid ‘protocol message’.

**Lemma 1** Let φ be one of the messages listed in figure : \( T^u(N), F^u(w), F^u(w), q^t(), v^t(), \) or \( N^u() \). Then there are a string \( \psi \) and a processor \( u \in V \) such that whenever a non-faulty processor \( v \) receives \( \phi \) and valid(\( \phi \)) = true, then \( v \) calls \( \text{flood}^v(\psi, \phi) \) or \( \text{flood}^u(\psi, \phi) \).

**Proof:** Follows immediately from the code. □

The values of \( \psi, u \) associated with a ‘protocol message’ \( \phi \) are the ‘flood identifiers’ of \( \phi \). Many messages \( \phi \) may have the same identifiers; these messages are ‘equivalent’ to flood.

**Definition 5.1** Let \( \phi \) be one of the messages listed in figure 1. Then \( \psi^v \) is called the flood identifier of \( \phi \), where \( \psi \) and \( u \) are according to Lemma 1.

In [Seg83], when the flood procedure (PI) is given a message, then it either sends that message to all its neighbors, or ignores that message. This is also kept by our flood procedure. However, there is a difference: we ignore the message not only if we flooded the same message before, but also if we flooded (two) messages with the same flood identifier, and hence identified the origin of the message as faulty.

**Lemma 2** Assume that a non-faulty processor \( v \) executes \( \text{flood}^v(\psi, \phi) \) or \( \text{flood}^u(\psi, \phi) \) at time \( t \). If \( v \) has not sent so far \( \phi \) and \( u \notin FV_v \) (or \( FV_u \)), then \( v \) will send \( \phi \) to all neighbors. The claim also holds for \( \text{flood}^u(\psi, \phi) \) and \( u \notin FV_u \cup FV_v \).

**Proof:** The proof is similar for \( \text{flood} \) and for \( \text{flood} \); we discuss only \( \text{flood} \). Whenever \( v \) executes \( \text{flood}^v \) with \( u \notin FV_v \), then it executes <B3>. If \( (\psi, u) \notin \text{flooded}_v \), then \( \phi \) is sent by <B9>. Otherwise, since \( \phi \) was not flooded before, then contents(\( \psi, u \)) was set to some other string, and the condition of <B4> will hold and \( \phi \) is sent by <B5>. □

The flood procedure operates like the PI protocol [Seg83] on a reliable network which consists of the connected sub-network of non-faulty links and processors. Namely:
Lemma 3 Assume that non-faulty processor \( v_0 \) executes \( \text{flood}^n(\psi, \phi) \) at time \( t \) (\( t \geq 0 \)), and for some \( l, \epsilon \), there is a non-faulty path of length \( l \) from \( v_0 \) to \( v_l \). Then before \( t + l \epsilon \), either \( u \in FV_{v_l} \) (and \( u \) is indeed faulty) or \( \text{flood}^n(\psi, \phi) \) occurs in \( v_l \). The claim holds also for \( \text{flood}^n(\psi, \phi) \) and \( u \in FV_{v_l} \cup \overline{FV}_{v_0} \).

Proof: By induction on \( l \) and since, roughly speaking, every processor forwards the flooded messages immediately (this basically extend the proof of [Seg83]). We prove the lemma for \( \text{flood} \); the claim for \( \text{flood} \) follows in the same manner. Assume first that \( u \notin FV_{v_0} \) at \( t \). Then from lemma 2, processor \( v_0 \) sends \( \phi \) to \( v_1 \) at \( t \) or sooner. Since \((v_0, v_1)\) is non-faulty, processor \( v_1 \) will receive \( \phi \) before \( t + \epsilon \). Since processor \( v_0 \) executed \( \text{flood}^n(\psi, \phi) \), surely \( \text{valid}(\phi) = \text{true} \). The claim follows from Lemma 1.

To complete the proof for \( l = 1 \), consider the case where \( u \in FV_{v_0} \) at \( t \). This means that \( v_0 \) flooded before \( t \) two different messages which have the same flood identifier and originated from \( u \). Since \((v_0, v_1)\) is non-faulty, processor \( v_1 \) will receive these two different messages before \( t + \epsilon \). Both of these messages are valid, since otherwise \( v_0 \) would not flood them. The claim follows from lemma 1 and from statements <B3>, <B4> and <B6>.

Assume now that the claims hold for \( l \); let \( t' \) be the time when \( \text{flood}^n(\psi, \phi) \) occurred in \( v_1 \), or when \( u \) was added to \( FV_{v_1} \). If \( \text{flood}^n(\psi, \phi) \) occurred then the claim follows by considering \( u_0 = v_l, v_l = v_{l+1} \) and applying the proof of \( l = 1 \). If \( u \) was added to \( FV_1 \) then the claim follows in the same manner, by considering the two activations of \( \text{flood} \) which caused \( v_1 \) to add \( u \) to \( FV_{v_1} \).

We now use lemma 3 to prove, essentially, that information propagates over a non-faulty link by at most \( \epsilon \) (see table 1).

<table>
<thead>
<tr>
<th>Scope</th>
<th>Number</th>
<th>Cause (at ( t_0 ))</th>
<th>Effect (at ( t_0 + \epsilon ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both</td>
<td>1</td>
<td>( v_0 ) started</td>
<td>( v_l ) started</td>
</tr>
<tr>
<td>Protocols</td>
<td>2</td>
<td>( u \in FV_{v_0} )</td>
<td>( u \in FV_{v_l} )</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>( (u, w) \in FE_{v_0} )</td>
<td>( (u, w) \in FE_{v_l} ) or ( u \in FV_{v_l} )</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>( \text{Reply}_{v_0}(u) \neq \lambda )</td>
<td>( \text{Reply}<em>{v_0}(u) \neq \lambda ) or ( u \in FV</em>{v_l} ) or ( w \in FV_{v_l} )</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>( (u, w) \in E^*_{v_0} )</td>
<td>( (u, w) \in E^*<em>{v_l} ) or ( u \in FV</em>{v_l} )</td>
</tr>
<tr>
<td>Query</td>
<td>6</td>
<td>( u \in FV_{v_0} )</td>
<td>( u \in FV_{v_l} \cup \overline{FV}_{v_l} )</td>
</tr>
<tr>
<td>Only</td>
<td>7</td>
<td>( (u, w) \in FE_{v_0} )</td>
<td>( (u, w) \in FE_{v_l} ) or ( {u, w} \cap (FV_{v_l} \cup \overline{FV}_{v_l}) \neq \emptyset )</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>( \text{Reply}_{v_0}(u) \neq \lambda )</td>
<td>( \text{Reply}<em>{v_0}(u) \neq \lambda ) or ( {u, w} \cap (FV</em>{v_l} \cup \overline{FV}_{v_l}) \neq \emptyset )</td>
</tr>
</tbody>
</table>

Table 1: Propagation of Information over \( v_0, \ldots, v_l \).

Lemma 4 Assume that the cause of line \( i \) in table 1 holds in processor \( v_0 \) at time \( t_0 \) (\( t_0 \geq 0 \)), and that for some \( l, \epsilon \), there is a non-faulty path of length \( l \) from \( v_0 \) to \( v_l \). Then the effect of line \( i \) will hold at \( t_0 + l \epsilon \).

Proof: The proof of all lines is straightforward, along the following outline. We identify a call to 16
flood (or \textit{flood}), with a specific message, which must have been executed at \( v_0 \) before \( t_0 \). The claim then follows from lemma 3 and the response to that message.

We give the details only for line 5 (as an example). Only statement \( \langle \text{D4} \rangle \) adds to \( E^*_{u_0} \). Hence, before \( t_0 \), procedure \textit{topo} was executed (with the same \( u \) and with \( \psi \in N \)). Consider all statements which call \textit{topo}, namely \( \langle \text{B1} \rangle \), the 'on start' statement and the 'on receive (\( T^{\psi}(N) \))' statement. In all of them, \( \phi = T^{\psi}(N) \). Hence, before \( t_0 \), statement \( \langle \text{D1} \rangle \) called \( \text{flood}^{\psi}(T, T^{\psi}(N)) \). From lemma 3, before \( t_0 + \ell_t \), either \( \text{flood}^{\psi}(T, \phi) \) will be executed in \( v_1 \) or \( u \) will be added to \( FV_{u_0} \). Assume that \( u \) was not added to \( FV_{u_0} \), otherwise we are done. Only statement \( \langle \text{D1} \rangle \) calls \( \text{flood}^{\psi} \) with \( \psi = T \). Hence, before \( t_0 + \ell_t \), procedure \textit{topo} was invoked with \( w \in N \); the claim follows from statement \( \langle \text{D4} \rangle \).

\section{5.2 Termination}

We begin by proving a property of the remote time out mechanism, which is the base of the termination proof.

**Lemma 5** Let \( u_0 \) be a non-faulty processor, and let \( t_{u_0} \) denote the time when \( u_0 \) first started the topology learning protocol. Let \( t' \) be defined as \( t_{u_0} + (3k + 1) \cdot B \). Consider any path of \( j \) links \( u_0, u_1, \ldots, u_j \), where \( j \leq k \), which is in \( (V^*_{u_0} - FV_{u_0}, E^*_{u_0} - FE_{u_0}) \) at \( t' \). Then \( \text{Reply}_{u_0}(u_j) \neq \lambda \). Furthermore, \( \text{flood}^{u_j}(T, T^{u_j}(N)) \) was called in \( u_{0} \) before \( t' \).

**Proof:** From Axiom 1, a \( T_{u_0} \) event occurs once every \( B \) time units. In particular, the number of \( T_{u_0} \) events until \( t' \) is at least \( 3k \). Statement \( \langle \text{B1} \rangle \) is executed at \( t_{u_0} \), setting \( \text{mode}_{u_0} = \text{Work} \). Hence statement \( \langle \text{A1} \rangle \) is executed every \( T_{u_0} \) event after \( t_{u_0} \), and at \( t' \) holds \( \text{time}_{u_0} \geq 3k \).

Consider the \( T_{u_0} \) event number \( 3k \) since \( t_{u_0} \). This event occurs at some time \( t_{3k,B} \) before \( t' \). If \( \text{Reply}_{u_0}(u_j) = \lambda \), then \( \langle \text{A3} \rangle \), and thereby \( \langle \text{C1} \rangle \), are invoked, and \( (u_0, u_1) \) is added to \( FV_{u_0} \). This contradicts the fact that at \( t' \), the path \( u_0, u_1, \ldots, u_j \) is in \( (V^*_{u_0} - FV_{u_0}, E^*_{u_0} - FE_{u_0}) \). Hence, \( \text{Reply}_{u_0}(u_j) \neq \lambda \) at \( t_{3k,B} \).

However, only \( \langle \text{D2} \rangle \) could make \( \text{Reply}_{u_0}(u_j) \neq \lambda \). Hence, \( \langle \text{D1} \rangle \) was executed before \( t_{3k,B} \), and called \( \text{flood}^{u_j}(T, T^{u_j}(N)) \). The claim follows since \( t_{3k,B} \leq t' \).

A similar claim holds for the query protocol.

**Lemma 6** Let \( u_0 \) be a non-faulty processor, and let \( t_{u_0} \) denote the time when \( u_0 \) first started the query protocol. Let \( t' \) be defined as \( t_{u_0} + (3k + 1) \cdot B \). Consider any path of \( j \) links \( u_0, u_1, \ldots, u_j \), where \( j \leq k \), which is in \( (V^*_{u_0} - FV_{u_0}, E^*_{u_0} - FE_{u_0}) \) at \( t' \). Then \( \text{Reply}_{u_0}(u_j), \text{Reply}_{u_0}(u_j) \neq \lambda \). Furthermore, \( \text{flood}^{u_j}(T, T^{u_j}(N)) \) was called in \( u_0 \) before \( t' \).

**Proof:** Statement \( \langle \text{F1} \rangle \) ensures that the topology learning protocol starts at \( u_0 \) no later than the query protocol. Hence, from Lemma 5, before \( t' \) holds \( \text{Reply}_{u_0}(u_j) \neq \lambda \) and \( \text{flood}^{u_j}(T, T^{u_j}(N)) \) was called in \( u_0 \). The rest of the proof is basically the same as of Lemma 5.

We now prove the termination property (and the time complexity). This 'fast' termination property is guaranteed only for processors which are connected to the initiator of the protocol by a non-faulty path.
Lemma 7 Let \( s \) be the processor where the topology learning (respectively, query) protocol initiated. Let \( t_v \) be the time when the topology learning (respectively, query) protocol started at processor \( v \), and let \( t'_v \overset{\Delta}{=} t_v + 2n\epsilon + (3\Delta + 1) \cdot B \). If there is a non-faulty path from \( v \) to \( s \), then \( v \) terminates before \( t'_v \). In the above, \( \Delta \) is the number of faulty processors and links.

Proof: The claim is, basically, based on the two following observations. First, the communication between processors connected by non-faulty paths takes at most \( n\epsilon \). Second, after at most \( 3\Delta B \) since transmission, the remote time-out mechanism will disconnect links to components of the network which do not respond in time. The proof is presented for the topology learning protocol; the proof for the query protocol follows the same lines.

The proof is by contradiction: assume that there is some processor \( u_k \) such that at \( t'_v \) holds \( \text{Reply}_v(u_k) = \lambda \) and there is a path from \( v \) to \( u_k \) in \((V_u^*, FV_v, E_u^* - F E_u)\). Let \( u_1, \ldots, u_k \) be the shortest such path, where \( k \) is the length of the path. Let \( i \) be the maximal index such that \( u_i \) is connected to \( v \) by a path which is non-faulty. Since we assumed that all non-faulty processors are connected by non-faulty paths, \( k - i \leq \Delta \). From line 1 of table 1, processor \( u_i \) starts the protocol before \( t_v + n\epsilon \). Denote the time when \( u_i \) started the protocol by \( t_{ui} \) (\( t_{ui} \leq t_v + n\epsilon \)) and denote \( t'_w \overset{\Delta}{=} t_v + (3\Delta + 1) \cdot B \).

Let \( j \leq k \) be the maximal such that at \( t'_{ui} \) the path \( u_i, u_{i+1}, \ldots, u_j \) is in \((V_u^*, FV_u, E_u^* - F E_u)\).

Assume first that \( j = k \). From Lemma 5, \( \text{Reply}_{u_i}(u_j) \not= \lambda \) at \( t'_{ui} \leq t'_v - n\epsilon \). Then, from line 4 of table 1, before \( t'_w \) either \( \text{Reply}_{ui}(u_k) \not= \lambda \) or \( u_k \in F V_v \). This gives the required contradiction.

Consider the other case, where \( j < k \). Namely, \((u_j, u_{j+1}) \not\in (V_u^*, FV_u, E_u^* - F E_u)\). One of the following cases hold:

1. Processor \( u_{j+1} \in F V_{u_i} \). Then, from line 2 of table 1, before \( t'_w \) holds \( u_{j+1} \in F V_v \). This gives the required contradiction.

2. Link \((u_j, u_{j+1}) \in F E_{u_i} \). Then, from line 3 of table 1, before \( t'_w \) either \((u_j, u_{j+1}) \in F E_v \) or \( u_{j+1} \in F V_v \) or \( u_j \in F V_v \). This gives the required contradiction.

3. Link \((u_j, u_{j+1}) \not\in (V_u^*, E_u^*) \). However, from Lemma 5, \( \text{flood}^{u_i}(T, T^{ui}(N)) \) was called in \( u_i \). Hence, \( u_{j+1} \not\in N \). From Lemma 3, before \( t'_w \) either \( u_j \in F V_v \) or \( \text{flood}^{v_i}(T, T^{ui}(N)) \) was called in \( v \). Either case contradicts the assumption.

Since all cases result in a contradiction, the claim is true. \( \square \)

5.3 Correctness

We now proceed to the correctness properties. The following lemma is the core of the completeness proof; it shows that the protocol always utilizes non-faulty links, since it never adds such a link to \( F E \) or \( \overline{F E} \).

Lemma 8 Let \((w, u_0)\) be a non-faulty link, where both \( w \) and \( u_0 \) are non-faulty processors. Then, for every non-faulty processor \( v \), and at any time, the link \((w, u_0)\) is not in \( F E_v \) (or in \( \overline{F E}_v \)).
Proof: The idea of the proof is that the link will be added to \( FE_w \) only if one of \( w, u_0 \) disconnects it. It suffice to show that \( w \) does not disconnect the link. This is basically ensured since \( u_0 \) basically performs the same remote time-out check as \( w \). The proofs for both protocols are almost identical, and we present the slightly cleaner proof of the topology learning protocol.

Assume to the contrary, namely that statement \(<C1>\) added \((w,u_0)\) to \( FE_w \). Since only \( e \)feil floods \( V^w(u_0) \), then previously processor \( w \) executed statement \(<A3>\). This happened at some \( T_w \) event, say the \( 3(k+1) \)th since \( w \) started. Let \( w, u_0, u_1, \ldots, u_k \) be the relevant path. Namely, at that time, \( \text{Reply}_w(u_k) = \lambda \).

Let \( t_w \) be the time when processor \( w \) (respectively, \( u_0 \)) started executing the protocol. From line 1 of table 4 follows that \( t_0 \leq t_w + t \).

Let \( t' \triangleq t_w + (3k + 1) \cdot B \leq t_w + (3k + 2)B \). Let \( j \leq k \) be the maximal index such that \( x_0, u_1, \ldots, u_j \) is in \((V^w_{u_0} - FV_{u_0}, E^w_{u_0} - FE_{u_0})\) at \( t' \). From 5, \( \text{Reply}_{u_0}(u_j) \neq \lambda \). Furthermore, \( \text{flood}^{u_0}(T, T^{u_0}(N)) \) was called in \( u_0 \) before \( t' \).

The rest of the proof follows like in Lemma 7. Assume first that \( j = k \). Then the contradiction follows from line 4 of table 1, since at \( t' \) holds \( \text{Reply}_{u_0}(u_j) \neq \lambda \), and since \( t' \leq t_w + (3k + 2)B \).

Consider the other case, where \( j < k \). Namely, \((u_j, u_{j+1}) \notin (V^w_{u_0} - FV_{u_0}, E^w_{u_0} - FE_{u_0})\). One of the following cases hold:

1. \( u_{j+1} \in FV_{u_0} \). Then the proof follows from line 2 of table 1.

2. \((u_j, u_{j+1}) \in FE_{u_0} \). Then the proof follows from line 3 of table 1.

3. \((u_j, u_{j+1}) \notin (V^w_{u_0}, E^w_{u_0}) \). However, from Lemma 5, \( \text{flood}^{u_0}(T, T^{u_0}(N)) \) was called in \( u_0 \). Hence, \( u_{j+1} \notin N \). The claim follows from Lemma 3.

Since all cases result in a contradiction, the claim is true. \( \Box \)

Using lemmas 8 and 7, we prove the properties of the two protocols.

**Theorem 1** The protocols presented in §4.3 perform topology learning and query as defined in def. 3.2 and 3.1.

**Proof** The termination property follows immediately from lemma 7. The safety property follows from the code.

From the code also follows easily that if \( v, u \) are non-faulty then \( u \notin FV_v \). To prove the completeness property, we use this observation, together with the dual observation about links. Namely, lemma 8 shows that a link which is non-faulty, and whose endpoints are non-faulty, is not added to \( FE_v \). It follows that every non-faulty path will be in \((V^w_{u_0}, E^w_{u_0})\) upon termination. \( \Box \)

**5.4 Complexities**

We now analyze the communication complexities. We first show that messages with a specific flood identifier \( \psi^w \) are sent at most \( 4 \) times over each link. (Recall that \( \psi^w \) is the flood identifier of protocol message \( \phi \) if when a non-faulty processor receives \( \phi \), it calls \( \text{flood}^u(\psi, \phi) \), see def. 5.1.)
Lemma 9 For every flood identifier \( \psi^u \), in every non-faulty processor \( v \), and in every execution of the topology learning (query) protocol, processor \( v \) sends at most two messages.

Proof: The send events are performed only by either flood or \( \overline{\text{flood}} \). For simplicity, we consider only flood; the proof for \( \overline{\text{flood}} \) is basically the same.

The proof is by observing the operation of the flood algorithm, when called with the same flood identifier, in the first time, second time, and following times. Let \( \phi \) be the message with which the flood identifier \( \psi^u \) was first called. Assume that at this first call, \( u \not\in \mathcal{F}V_v \). Hence, statements \( <B8> \) to \( <B10> \) are executed. Statement \( <B9> \) sends \( u \) to all neighbors. Statement \( <B8> \) adds \( (u, \psi) \) to \( \text{flooded}_u \), and sets \( \text{contents}_u(u, \psi) \) to \( \phi \).

Subsequent calls with \( \phi \) are ignored by statements \( <B3> \) and \( <B4> \). Also, calls when \( u \in \mathcal{F}V_v \) are ignored. Hence, assume that \( u \not\in \mathcal{F}V_v \), and consider a call to flood with a message \( \psi' \) with flood identifier \( \psi^u \), such that \( \psi' \neq \phi \).

Consider the first call to flood\( ^u(\psi, \psi') \) with \( \psi' \neq \phi \). At this time, \((u, \psi) \in \text{flooded}_u \) and \( \text{contents}_u(u, \psi) \rightarrow \phi \). Hence, statements \( <B5> \) and \( <B6> \) are executed, and \( u \) is added to \( \mathcal{F}V_v \).

Hence, subsequent calls to flood\( ^u(\psi, \psi') \) with \( \psi'' \neq \phi \) are ignored by statement \( <B2> \). To summarize, at most two calls to flood\( ^u(\psi, \psi') \) with the same \( \psi \) caused processor \( v \) to send a message. \( \Box \)

Lemma 9 shows that for every flood identifier \( \psi^u \), the protocol will perform a message with flood identifier \( \psi^u \), at most 4m times. It is easy to see that the protocol uses \( O(n + f) \) different flood identifiers; hence the total number of send events is \( O(m \cdot (n + f)) \). However, for communication complexity analysis, we count the number of \( \log(n) \) bit long messages, i.e. messages containing a single processor identity. Hence, a single long message may be computed as several \( \log(n) \) bit long messages.

To perform this analysis, we denote by \( L(\psi^u) \) the maximal length of a message sent with flood identifier \( \psi^u \). The following lemma bounds \( \sum_{\mathcal{V}_u, \psi} L(\psi^u) \) for both protocols; the message complexity follows by combining this and Lemma 9.

Lemma 10 In the topology learning protocol, \( \sum_{\mathcal{V}_u, \psi} L(\psi^u) = O(m) \). In the query protocol, \( \sum_{\mathcal{V}_u, \psi} L(\psi^u) = O(n + f) \).

Proof: We count the part contributed by each type of message. Edge failure message (\( \mathcal{F}\psi^u(w) \) or \( \not\mathcal{F}\psi^u(w) \)) are of length 2, and from Lemma 8, the number of such failures is at most \( f \); this contributes \( O(f) \) (to both protocols). The other types of messages are different in the two protocols; we begin with the query protocol.

The query itself is of length 2 (the query and the source), and is issued only by \( s \), and hence contributes only \( O(1) \). There are two types of responses (\( Y \) and \( N \)), each also of length 2, and they are issued by each processor; hence they contribute the term of \( O(n) \) to the complexity of the query protocol.
In the topology learning protocol, there are two types of messages: $F^u(w)$, which contribute at most $O(f)$ as counted above, and $T^u(N)$ messages. From the validity axiom, in every message $T^u(N)$ such that $\text{valid}(T^u(N)) = \text{true}$ holds $(\forall w \in N)((u, w) \in E)$. Hence

$$\sum_{V_u} L(T^u) \leq \sum_{V_u} (1 + |\{v|(u, w) \in E]|) \leq |V| + |E| = O(m)$$

Since $m \geq f$, this completes the proof. □

We now conclude the complexities of the protocols.

**Theorem 2** The time complexity of both protocols is $O(nc + fB)$ (optimal). The message complexity of the query protocol is $O((n + f) \cdot m)$, and of the topology learning protocol is $O(m^2)$.

**Proof:** The time complexity follows from lemma 7; its optimality follows by considering a path of $f$ links whose delay is exactly $B$, which is connected to the rest of the network by a single link at one of its ends.

The message complexities follow by multiplying the values of $\sum_{V_u, \psi} L(\psi^u)$ from lemma 10 by $4m$. The value $4m$ is the total number of (directed) links $(2m)$ multiplied by the bound of 2 obtained in lemma 9 (on the number of sends, by a non-faulty processor, of messages with the same $\psi^u$). □
Acknowledgements

Many thanks to Oded Goldreich and Adrian Segall, my advisors, for guidance, criticism, entertainment, patience and impatience which have improved both the results and the exposition. Hagit Attiya, Baruch Awerbuch, Benny Chor, Shimon Even, Shay Kutten, Alain Mayer, Mike Merritt and Rafi Rom helped me with fruitful comments and discussions.

References


