An \((N - 1)\)-Resilient Algorithm for Distributed Termination Detection

Ten-Hwang Lai, Member, IEEE, and Li-Fen Wu

Abstract—This paper presents a fault-tolerant termination detection algorithm based on a previous fault-sensitive scheme by Dijkstra and Scholten. The proposed algorithm can tolerate any number of crash failures. It runs as efficiently as its nonfault-tolerant predecessor if no process actually fails during the computation, and otherwise incurs only a small amount of cost for each actual failure. It is assumed that the underlying communication network provides such services are reliable end-to-end communication, failure detection, and fail flush.

Index Terms—Distributed algorithm, fault tolerance, message complexity, termination detection.

I. INTRODUCTION

A DISTRIBUTED SYSTEM is a set of processes running on top of a communication network (directly or indirectly above the transport layer of the ISO OSI reference model [23]). In the absence of faulty processes, the system is said to have terminated if two conditions are satisfied: 1) all processes are idle and 2) there are no messages in transit. The problem of termination detection is to determine whether a distributed system has terminated. A protocol or algorithm that solves the problem is called a termination detector. The system whose termination is the object of the detection is called the basic system.

Many algorithms for termination detection are available in the literature. Some algorithms assume synchronous communications[8], [9], [12], [17], [20], [24], while others work equally well for asynchronous systems [7], [10], [13], [14], [18], [19]. Some algorithms in the latter category require FIFO channels [7], [19], and some do not [10], [13], [14], [18]. Most algorithms are intended for static systems only; algorithms for dynamic systems can be found in [8], [14]. All these algorithms are fault-sensitive and may work incorrectly in the presence of faulty processes.

Fault tolerance is one of the most desirable properties of distributed systems. Therefore, when a termination detector is applied to a fault-tolerant basic system, it is desirable that the detector itself be fault-tolerant. Unfortunately, the problem of termination detection is harder than the consensus problem [26], and the latter is well known to be unsolvable in the face of faulty nodes under the common assumption of reliable asynchronous communications for the underlying network [11]. Thus, in order to solve the termination detection problem in the presence of faulty processes, the model of the underlying network must be stronger than that assumed in [11].

In a recent paper [25], Venkatesan proposed a fault-tolerant termination detector, called the \(V\) algorithm, in which he assumed that the underlying network has the following additional features: a) should a processor fail, each of its neighbors knows of the failure within a finite amount of time; b) a message sent to a failed processor \(x\) is returned to the sender \(y\) after all messages from \(x\) to \(y\) have been received by \(y\); and c) each processor is able to send two or more messages atomically. (Note that in an environment with these extra features, the consensus problem is solvable.) The \(V\) algorithm is based on a previous nonfault-tolerant version, the CV algorithm [6], and can be \(k\)-resilient for any prespecified value \(k\), \(1 \leq k \leq n - 1\). (A protocol or distributed algorithm is said to be \(k\)-resilient if it works correctly even in the presence of \(k\) faulty processes.) The \(V\) algorithm has several drawbacks. First, the parameter \(k\) needs to be prespecified by the user. Second, the algorithm requires \(\Omega(kM)\) control messages to prepare itself for possible process failures, where \(M\) is the total number of basic messages (i.e., messages of the basic system).

This cost cannot be waivered even if no failure actually occurs during the basic computation. Third, should failures occur, the algorithm requires about \(O(kM/n)\) additional control messages per failure, where \(n\) is the total number of processes in the basic system; this cost may be intolerably high in a message-intensive computation. Fourth, it requires processes to send multiple messages in an atomic way.

In this paper, we assume an underlying communication network that provides such services as reliable end-to-end communication, failure detection, and fail flush. (This model is not much different from the one used in [25], and will be described in the next section.) With the proposed model, we develop an \((n - 1)\)-resilient algorithm based on a variant (the LTD variant [15]) of the well-known Dijkstra-Scholten (DS) algorithm [10], where \(n\) is the total number of processes in the basic system. The LTD variant, like the original DS algorithm, is worst-case message optimal and is one of the most efficient termination detectors available in the literature. Our algorithm has a very interesting property: it achieves fault tolerance without message replication and thereby avoids all preparation cost (in terms of control messages); if no process ever fails during basic computation, our algorithm is as efficient as its predecessor, the LTD algorithm. (For comparison, recall that the \(V\) algorithm needs \(\Omega(kM)\) control messages to get ready...
for possible faults). This property is certainly desirable, since in today’s systems it seems that most computations are done without a process failure. Our algorithm also improves over the V algorithm in several other aspects: 1) should a process fail during the computation, our algorithm recovers with at most O(n) control messages, which is in most cases smaller than the V algorithm’s O(M) (assuming k = n - 1, as in our case); 2) the space complexity is O(n^2) as compared with O(nM) for the V-algorithm (as M is not known at compile time, the V algorithm requires the more expensive dynamic memory allocation); 3) our algorithm has a shorter detection delay of O(n) as compared with the O(M) required by the V algorithm; and 4) our algorithm does not need the service of sending multiple messages atomically, which, in contrast, is crucial for the V algorithm.

We assume that all processes are able to communicate with each other and that the underlying network provides such services as fault-detection and fail-flush. These services are described in the next section, where the problem of termination detection is also defined. We will review the DS algorithm, and the LTD variant in Section III, and describe our algorithm in subsequent sections. We will prove our algorithm correct and analyze it in Sections V and VI, respectively.

II. THE MODEL

In this section, we model the underlying communication network and formally define the problem of fault-tolerant termination detection.

A. The Underlying Communication Network

A distributed system is a set of processes communicating through a communication network. As Fig. 1 shows, each process runs on a station (computer), on top of a communication subsystem (which, for instance, has seven layers in the ISO OSI reference model). All the stations are connected to a communication network. We assume that the network, together with the communication subsystems of the stations where the basic processes reside, provides the following services:

S0: All-Pair Reliable End-to-End Communications: Every process is able to communicate with every other process. The network guarantees delivery of every message to its destination without errors, loss, or duplication—unless the destination process crashes before receiving the message, in which case the message is said to be undeliverable and in discarded by the network without notifying the sender. Messages may experience finite but arbitrary delay in the network, and they are not necessarily delivered in the order they were sent.

S1: Fault Detection: The communication network provides the service of fault detection. A process i may issue a FAULT-DETECT.request(Q) primitive, to request that it be informed should any process in Q fails, where Q is a set of process identities. Every time a process j in Q fails, the network sends to i a FAULT-DETECT.confirm(j) primitive to report that j has failed. We do not assume that failures be reported to processes in any particular order. Thus, it is possible that the network reports failures to different processes in different orders.

S2: Fail Flush: The network also provides a fail-flush service. A process i may issue a FAIL-FLUSH.request(j) primitive to request this service, where j is a process known to have failed. This primitive requests that the network clear the logical channel from j to i. The network issues a FAIL-FLUSH.confirm(j) primitive to process i after all messages sent from j to i have been delivered. (These messages, if any, were sent before j’s failure.) Thus, after receiving the FAIL-FLUSH.confirm(j) primitive, process i knows for sure that no more messages will come from process j.

We make the following definition just for convenience in describing our algorithm in subsequent sections.

Definition 1: We say that “process i has detected process j’s failure” iff process i has received a FAULT-DETECT.confirm(j) primitive.

It must be emphasized that the processes themselves are not capable of detecting failures. It is the underlying communication system that provides the fault-detection service. The reader is referred to [5] for an interesting survey of a variety of failure detectors, and to [3], [2], [21] for further discussion on fault diagnosis.

The fail-flush service as depicted in S2 was inspired by the forward-flush of [1] and by the return flush of [25]. In a return flush, if the destination of a message has failed, the network returns the message to its source node after the network has cleared (flushed) the channel from the destination to the source node. Both return-flush and fail-flush serve the same purpose: to ensure that no basic messages from a faulty process are pending.

B. The Problem

The basic system is a set of n processes, \( P = \{p_1, p_2, \ldots, p_n\} \), which communicate with one another by sending and receiving messages through a communication network. Each process has a unique identity and is aware of other processes’ identities. Without loss of generality, let their identities be 1, 2, . . ., n. We shall interchangeably refer to the process labeled with i either as \( p_i \) or i.

Associated with each process is a state and a basic computation. Before its basic computation gets started, a process’s state is either undefined or faulty, meaning the process has not started its basic computation or has crashed, respectively. Once a process starts its basic computation, it is in exactly one of three possible states: idle, active, or faulty. A process is said to be idle (or active or faulty) if it is in the idle (or active or faulty) state. Each basic computation is a sequence of such events as receiving a message, sending a message,
performing arithmetic/logical (A/L) operations, and changing the process's state, and is subject to the following rules:

- While in the active state, a process can perform A/L operations, send messages, and receive incoming messages without any restriction. It may become idle in an unpredictable fashion.
- Once in the idle state, a process does not perform any A/L operations and cannot send any messages; but it can receive an incoming message, upon which it becomes active immediately.
- An idle or active process may fail (crash). Once a process fails, it stops running and never recovers. (This model of process failure is the same as that assumed in [11], [25].)
- A faulty process is one that has failed. So a faulty process can neither send nor receive messages.

The messages sent by the processes during their basic computations are called basic messages, as to be distinguished from the control messages employed by a termination detector. A basic message is said to be in transit if it has been sent but has not yet been received. A process which is not faulty is nonfaulty.

The basic computations of the n processes, taken together as a whole, are called the computation of the basic system. In the literature, the computation of a basic system without faulty processes is said to have terminated if all processes are idle and there is no basic message that is still in transit; and the problem of termination detection is to design a protocol capable of determining whether the computation of a given basic system has terminated.

In this paper we study the problem of termination detection in the presence of faulty processes. The exact formulation of the problem partly depends on how the underlying communication network handles undeliverable messages, namely, those addressed to faulty processes. We assume that the communication network discards undeliverable messages without notifying their senders, and accordingly define our problem as follows.

**Definition 2:** The computation of a basic system is said to have terminated if 1) every nonfaulty process is idle and 2) no basic message whose destination is a nonfaulty process is still in transit.

Note that we ignore undeliverable messages in the above definition since they are assumed to be eventually discarded by the network. This definition of termination is consistent with the traditional one if no process ever fails during the basic computation.

**Definition 3:** The problem of fault-tolerant termination detection is to design a protocol to determine whether the basic computation has terminated. It is required that 1) after the basic computation terminates, some process in the system will eventually declare termination (unless all processes crash before any process has a chance to do so) and 2) if some process declares termination at some point of time, then the basic computation has really terminated by that moment.

An algorithm that solves the termination detection problem is called a termination detector. We adopt the common view that a termination detector will be "superimposed" on the basic system. That is, the detector will be implemented as an integrated part of the basic system, so that the same process at each station will run both its basic computation and the termination detection algorithm. There is an interesting discussion on the notion of superimposition in [4].

**Remarks:** In our model, we assume that the network discards all undeliverable messages. If undeliverable messages are instead returned back to their senders (as assumed in the V-model), the basic computation of the system should accordingly be defined to have terminated if 1) every nonfaulty process is idle and 2) the source and destination processes of all basic message in transit are presently faulty. In that case, a return-flush service is needed; and with some minor modifications, our algorithm offered in this paper will still work.

**III. PREVIOUS WORK**

As mentioned in the introduction, our algorithm is based on the LTD variant of the Dijkstra-Scholten algorithm [10], and will be evaluated against the resilient protocol of Venkatesan [25]. In this section, we review the DS algorithm and summarize the V algorithm. A thorough understanding of the DS algorithm will later help in understanding ours.

**A. Dijkstra Scholten Algorithm and its Variants**

The DS algorithm is applicable to diffusing computing, in which a distinguished process, called the root, starts the basic computation, with all others initially idle. (Except for this constraint, diffusing computing is no different from a general distributed computation as described in the preceding section. In particular, each process is either idle or active during the computation.) Each process is regarded either as neutral or as engaged, depending on its status. The concept of "neutral/engaged" is a product of the algorithm; it is related, but not equivalent, to that of "idle/active." Initially, only the root is engaged; all others are natural. If a neutral process p receives a basic message, say, from q, it becomes engaged as well as active; p is said to be engaged by q if p is introduced between the two processes; the involved message is called an engagement message. In this way, all engaged processes form a tree with edges directed from children to parents. Every basic message is acknowledged by its receiver with a SIGNAL. A process can acknowledge an incoming nonengagement message any time after the receipt. However, it can acknowledge an engagement message only if the following three conditions are satisfied: 1) the process is idle, 2) it has acknowledged all incoming nonengagement messages, and 3) all its outgoing messages have been acknowledged. An engaged process becomes neutral again once it has acknowledged its incoming engagement message. The diffusing computation is declared to have terminated when the root becomes neutral (or more precisely, when the root satisfies the above three conditions).

The algorithm is spelled out in Fig. 2 with p_r acting as the root. Each process p_i maintains three variables: in_i[1 . . . n], out_i, and parent_i where in_i[j] is the number of yet-to-be-acknowledged messages received from p_j, out_i is the total number of p_i's outgoing messages which have not been acknowledged, and parent_i is the process that engaged p_i. (Initially, out_i = in_i[j] = 0 and parent_i = NULL for all i and
j). Note that a process \( p_i \) is neutral iff \( p_i \) is idle, \( \text{inj}_i[j] = 0 \), and \( \text{out}_i = 0 \). Also note that the variables \( \text{parent}_i \) define a tree consisting of all engaged processes. The tree changes shape as time passes: a neutral process may become engaged and join the tree as a leaf and a leaf may become natural and drop off.

The DS algorithm requires exactly \( M \) control messages, where \( M \) is the number of basic messages. This is optimal in terms of worst-case complexity. The DS algorithm has two drawbacks: it always requires \( M \) control messages and it works only for diffusing computations. These, however, can be easily overcome as shown in the LTD variant [15]:

1) Instead of acknowledging one message per signal as in the original DS algorithm, let each SIGNAL carry an integer \( c \) so that it acknowledges \( c \) basic messages at a time. (This in general will reduce the number of control messages, although in the worst case it still needs \( M \).

2) Instead of initializing for all \( i \), let \( \text{parent}_i := \text{NULL} \), \( \text{inj}_i := 0 \) and \( \text{out}_i := 0 \) as in the original DS algorithm, the LTD variant initializes these variables as follows:

   - Let \( \text{parent}_1 := \text{NULL} \) and \( \text{parent}_i := 1 \) for all other processes \( i \) (so that the \( n \) processes form a tree of height 1 with \( p_1 \) at the root).

   - Set \( \text{in}_i \) and \( \text{out}_i \) as above: 1) for all processes \( i,j \), let \( \text{in}_i[j] = 1 \) if \( j = \text{parent}_i \), and 0 otherwise; and 2) for all \( i \), let \( \text{out}_i = \sum_j \text{in}_j[i] \).

These simple changes in initialization enable the DS algorithm to work for nondiffusing computations. The reader is referred to [15] for a proof.

The worst-case message complexity of the LTD variant is \( M + n \), which is optimal. We shall adopt the LTD extension in our effort to make the DS algorithm fault-tolerant.

The LTD algorithm is asymmetric. In [22], a nondiffusing computation is treated as a collection of diffusing computations, resulting in a beautiful symmetric termination detector that works for both diffusing and nondiffusing computations. Its worst-case message complexity is \( O(M + n * E) \), where \( E \) is the number of communication links.

B. A Fault-Tolerant Termination Detector

Venkatesan's algorithm [25] is a fault-tolerant version of a previous algorithm by Chandrasekaran and Venkatesan [6]. The main idea of the CV algorithm is similar to that of the DS algorithm. But, instead of counting messages, the CV algorithm uses stacks, one per process, to keep record of incoming and outgoing messages. Each process's transactions of sending or receiving a message are sequentially logged into its stack. When a process becomes idle, it removes all "receiving" entries on the top of the stack until a "sending" entry is reached. The process acknowledges the messages corresponding to these removed receiving entries. A sending entry is removed from the stack when an acknowledgment to the corresponding message is received. Like the original DS algorithm, the CV algorithm, once invoked, needs a control message for each basic message. The most interesting feature of the CV algorithm is that it need not start running right from the beginning of the basic computation. It may start later than the basic computation and still work correctly. This useful feature of "postponed start" has been incorporated into the DS algorithm as shown in the LTD variant [15].

The CV algorithm was transformed into the fault-tolerant V algorithm by message replication. Each node is associated with \( k \) other nodes, called its representatives, where \( k \) is the number of tolerable faults. (The user of the algorithm needs to specify the value of \( k \).) Whenever a node enters and exits its stack, it atomically replicates the entry and sends one copy to each representative. Each representative keeps a stack for each node represented by it. In this way, should a node fail, at least one of its representatives is still alive and will be able to simulate the faulty node's local stack.

IV. \((N - 1)\)-RESILIENT ALGORITHM

As was mentioned earlier, our algorithm is based on a variant of the DS algorithm. We first discuss some of the problems that need to be solved in order to make the DS algorithm fault-tolerant; then we describe our algorithm.

A. Problems Caused by Faulty Processes

Recall that in the DS algorithm all engaged processes form a tree \( T \). When the tree degenerates to nil, the computation is known to have terminated. There are at least three problems that need to be solved in order to make the algorithm fault-tolerant. First, if the coordinator (the root) fails, some surviving process must succeed as the coordinator. Second, a nonfaulty process must be able to tell whether an inbound channel from a faulty process is empty. Third, if an engaged process, say \( p \), fails, then the subtree of \( p \) rooted at \( p \) will break off from \( T \), and the nonfaulty nodes in \( T \) will be unable to report to the coordinator when they become neutral. To be resilient to faulty processes, it is thus necessary to re-attach to \( T \) the nonfaulty nodes in \( T \) or, in other words, to establish new child/parent relationships between the offspring of \( p \) and the remaining
nodes in T. This requires making both a child and its new parent aware of the new relationship.

The second problem can be easily solved using the FAIL-FLUSH service provided by the network (see Section II-A). To solve the first problem, let process $p_1$ be the coordinator initially. If it fails, let process $p_2$ take over the job of coordination. If process $p_2$ also fails, then process $p_3$ succeeds as the coordinator. In general, we let the nonfaulty process with the smallest index be the coordinator. In our algorithm, a process $p_i (i \neq 1)$ which becomes a coordinator will know its new status only after it has received from the network a FLUSH package. Now consider issue c). With the above strategy, the coordinator now knows that all engaged processes will eventually become its children after the failure of a process. The problem is that the coordinator doesn’t know which processes are engaged and which are not. We need a scheme to ensure that the coordinator will know whether it has received a report from every child. That is, when an engaged process detects a fault, it adopts as its new parent the process that it believes is the coordinator. When the engaged process becomes neutral, it will send a report (signal) to the coordinator.

The third problem is considerably harder, especially if more than one process fails. There are three issues here:

a) Which nodes in $T_p$ should these nodes be attached back to $T$? every node in $T_p$ or just the children of $p$?

b) Which nodes in $T$ should these nodes be attached to?

c) How is a node $q$ in $T_p$ actually attached to a node $r$ in $T$? It is not sufficient for $q$ to simply send a notice to inform $r$ of the new child/parent relationship, because due to message delay, it is possibly that $r$ becomes neutral and leaves tree $T$ before receiving the notice.

Issues a) and b) are not trivial because each process has only a little knowledge of the structures $T$ and $T_p$. The only thing an engaged process knows about tree $T$ is the identity of its own parent and that of the root; the latter may or may not be up-to-date information. A process in general doesn’t know whether it is a descendant of the detected fault $p$ (unless $p$ is its parent), and which processes are in $T$ and which are not. With little information about $T$ and $T_p$, we will simply let every engaged process that detects a fault become a child of the coordinator. That is, when an engaged process detects a fault, it adopts as its new parent the process that it believes is the coordinator. When the engaged process becomes neutral, it will send a report (signal) to the coordinator.

Now consider issue c). With the above strategy, the coordinator now knows that all engaged processes will eventually become its children after the failure of a process. The problem is that the coordinator doesn’t know which processes are engaged and which are not. We need a scheme to ensure that the coordinator will know whether it has received a report from every child. Not that all the problems mentioned above are compounded by the facts that there may be multiple process failures, that not all processes detect a failure immediately or simultaneously, that the coordinator may fail, and that processes may have out-of-date knowledge about the coordinator.

Despite these tangled problems, the basic ideas of our solution are simple. First, when a nonfaulty process, engaged or neutral, detects a fault in the system, let it adopt the coordinator as its parent and let it send a signal (report) to the latter when it is done. Second, let the signal not only carry a number as in the DS algorithm to acknowledge basic messages, but also a set of processes that contains all faults the process has so far detected. The set of detected faults is intended to help the coordinator determine whether any child has signaled its completion of processing. Roughly speaking, if $S_i = F$ for every process $i \notin F$, then the coordinator has received the most updated report (signal) from each child and thus has sufficient information for checking if the system is terminated, where $S_i$ is the set of faults reported to the coordinator by process $i$ and $F$ is the set of all faults detected by the coordinator.

Having pointed out the major problems caused by faulty processes as well as our basic strategies, we are in a position to describe the algorithm.

### B. Algorithm Description

Let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of processes of the basic system. Process $p_i$ is identified by the integer $i$ and is often referred to as process $i$ or just $i$. A faulty process is simply referred to as a fault.

**Overview** Our fault-tolerant algorithm is based on the LTD variant of the DS algorithm. It is basically the LTD variant plus the following extensions:

- Each process $i$ records the faults that have been reported to it by the underlying network. Let $DF_i$ denote the set of all such faults.
- When a process $i$ sends a signal as in the DS algorithm, the set $DF_i$ is included in the signal. In this way, processes share information about faults.
- Each process $i$ keeps track of the faults that have been reported to it by every other process. Let $RF_i[j]$ denote the set of faults reported by process $j$.
- When a process $i$ is informed of a fault by the underlying network, it issues a FAIL-FLUSH request to clear the channel from the faulty process to $i$ itself, and changes its parent to the coordinator.
- The coordinator, say process $r$, makes use of $DF_r$ and $RF_r[1 \ldots n]$ to avoid premature declaration of termination. Before the coordinator can check whether the basic computation has terminated, it first checks whether $DF_r = RF_r[j]$ for every $j \notin DF_r$. The latter condition roughly means that the coordinator has received a most recent signal from each nonfaulty process. Only if this condition holds can the coordinator go ahead to check for system termination in a way similar to the DS algorithm.

**Local Variables and Control Messages** A control message $\text{SIGNA}(c, S)$ is used in our algorithm to acknowledge the receipt of $c$ basic messages and to report that the sender has detected all the processes in $S$ as faulty ones. A single $\text{SIGNA}(c, S)$ message of our algorithm has the effect of $c$ signals of the DS algorithm. The $S$ parameter is used to help the coordinator determine whether it has received a report from every nonfaulty process.

Each process $p_i$ maintains the following local variables:

- $in_i[1 \ldots n]$ — an integer array, where $in_i[j]$ indicates the number of messages that $p_i$ has received from $p_j$ but has not yet acknowledged.
- $out_i[1 \ldots n]$ — an integer array, where $out_i[j]$ records the number of messages sent to $p_j$ for which an acknowledgment has not yet been received.
- $\text{parent}_i$ — indicating which process to report to when $p_i$ becomes neutral.
- $\text{crd}_i$ — a boolean variable indicating whether $p_i$ is the coordinator.
• $DF_i$—the set of all processes $x$ for which $p_i$ has received a FAULT-DETECT.confirm($x$). By Definition 1, this set consists of all processes that $p_i$ has detected as faulty. Thus, one may regard $DF$ as meaning detected failures.

• $RF_i$—the union of all sets $S$ contained in the SIGNAL messages that $p_i$ has so far received. The processes in this set are those which have been reported to be faulty by other processes. The mnemonics RF stands for reported failures.

• NumFault$_i$[1…$n$]—an integer array, where NumFault$_i$[j] indicates $p_i$’s knowledge about $|DF_i|$, the number of faults that $p_j$ has detected. (Rather than keep track of the set $RF_i[j]$ for each $j$ as mentioned above in the overview, it turns out to be sufficient to just record the number $\text{NumFault}_i[j] = |RF_i[j]|$ for each $j$, and keep one single set $RF_i$.)

Note that a process may know of a fault through a SIGNAL or a FAULT-DETECT.confirm primitive, and it stores this information in $RF_i$ or $DF_i$ accordingly. In this paper, whenever we say that a process “detects” a fault, we exclusively refer to the event as defined in Definition 1, not counting that of knowing a fault through a SIGNAL.

**Initialization of Variables**—The above mentioned variables are initialized as follows.

- Each process $i$ initially does not know of any process failure, so $DF_i = RF_i = \emptyset$ and $\text{NumFault}_i = 0 = (0, . . . , 0)$.
- Let $p_i$ be the first coordinator. So initially $crd_1 = true$ and $crd_i = false$ for $i \neq 1$.
- Let parent$_1 = NULL$ and parent$_i = 1$ for all other processes $i$, so that each process will later report to $p_1$. Note that the $n$ processes form a tree with $p_1$ at the root.
- Initialize in$_i$ and out$_i$ in accordance with the above tree: for all processes $i, j$, let in$_i[j] = out_i[j] = 1$ for all $i \neq 1$, and in$_i[j] = out_i[j] = 0$ otherwise. (This tree structure accommodates the cases where there are multiple initially active processes.)

**The Algorithm**—The algorithm for each process $p_i, 1 \leq i \leq n$, is given in Fig. 3 as a number of event-driven actions. After the process gets started, there are six events that may trigger an action—sending or receiving a basic message, becoming idle, receiving a signal, receiving a FAIL-FLUSH.request, and detecting a fault (i.e., receiving a FAULT-DETECT.confirm)—some of them may involve multiple actions. Actions A1–A5 correspond to actions D1–D5 of the DS algorithm, and are responsible for termination detection. The other two, F1 and F2, cope with faulty processes.

A0 initializes the variables in$_i$, out$_i$, etc., as described in the above, and issues a FAULT-DETECT.request($P$). It takes place when a process starts its basic computation. Action A1 simply counts outgoing messages, and A2 counts incoming ones. Moreover, in A2, if, upon receiving a basic messages, $p_i$ is not the coordinator and has no parent, then it becomes a child of the process from which the message came. Intuitively, this means $p_i$ becomes engaged.

Action A3 is executed on two occasions: 1) when a process becomes idle (i.e., when the process’s state changes to idle from either undefined or active, or 2) when a process detects a fault. In the latter case, F1 and A3 are both triggered and must be executed in that order. The main function of A3 is to acknowledge nonengagement messages. So a SIGNAL is sent to every nonfaulty process $j (j \neq parent_i)$ from which there are still unacknowledged messages. The signal carries two pieces of information: the number of messages $p_j$ wants to acknowledge (in$_j[j]$) and the set of faults $p_j$ has so far detected ($DF_i$). Note that no signal is sent to $parent_i$. So, should $p_i$ ever change the value of $parent_i$ after it has become idle, it may need to send a signal to its original parent. Since $parent_i$ may change value in F1, A3 is executed after F1.

The purpose of action A4 is for $p_i$ to send signal to its parent (parent$_i$) or, if it is the coordinator, to check if the system has terminated. This must be done when $p_i$ is ready to become neutral, i.e., when these conditions are satisfied: $p_i$ is idle, out$_i = 0, \sum_{k \neq \text{parent}_i} \text{in}_i[k] = 0$, and $DF_i \subseteq DF_i$. So A4 is executed immediately following A3, A5, or F2, each of which has a chance to bring about at least one of the above conditions. In this action, it is first checked whether $p_i$ is ready to become neutral. If so and if $p_i$ is not the coordinator, then it reports to its parent with a SIGNAL and updates in$_i[\text{parent}_i]$ and parent$_i$ to reflect its neutral status. If $p_i$ is the coordinator, then it checks whether NumFault$_i[j] = |DF_i|$ for all $j \in P - DF_i - \{i\}$. If so, $p_i$ declares termination. We shall prove in the next section that if $p_i$ is the coordinator, then the conditions that $p_i$ is idle, out$_i = 0, \sum_{k \neq \text{parent}_i} \text{in}_i[k] = 0, RF_i \subseteq DF_i,$
and $\text{NumFault}_i[j] = |DF_i| \text{ for all } j \in P - DF_i - \{i\}$ together imply the termination of the basic computation.

A5 is triggered when $p_i$ receives a SIGNAL($c$, $S$) from $p_j$. The signal carries two pieces of information: 1) an integer $c$ for $p_j$ to acknowledge $c$ messages and 2) a set $S$ for $p_i$ to report detected faults. So in this action, the number of unacknowledged messages $\text{out}_i[j]$ is reduced by $c$ and the set $S$ is added to $RF_i$, the set of faults that have been reported to $p_i$ by other processes through SIGNALS. Also, $\text{NumFault}_i[j]$ is raised to $|S|$ if its current value is smaller than $|S|$. This variable records how many faults have been reported by $p_j$. It is updated only if the new signal reports more faults than the previous ones (from $p_j$). It is possible that the $S$ contained in the current signal be smaller than previous ones (in which case $\text{NumFault}_i[j]$ will not be changed) because the signals of $p_j$ may arrive at $p_i$ in a non-FIFO order.

When $p_i$ receives a FAULT-DETECT.confirm($x$) primitive, F1 is invoked. The newly detected fault is added to $DF_i$, and $\text{out}_i[x]$ is incremented by $c$, any positive number. (For understanding the algorithm, it is convenient to think of $c$ as a small number. In implementation, one may let $c = 1$.) In normal cases, $\text{out}_i[x]$ counts only outgoing basic messages.

Here, we artificially increase the value of $\text{out}_i[x]$ so that $p_i$ can ensure that there are no messages in the channel from $x$ to it. It works as follows: 1) $p_i$ issues a FAIL-FLUSH.request($x$) primitive; 2) the network will clear the logical channel from $x$ to $i$ and inform $p_i$ of the fact by a FAIL-FLUSH.confirm($x$) primitive; 3) when $p_i$ receives a FAIL-FLUSH.confirm($x$) primitive in F2, $\text{in}_i[x]$ and $\text{out}_i[x]$ are set to zero to reflect the fact that the channel between $p_x$ and $p_i$ is now free of deliverable basic messages; 4) before that, $\text{out}_i[x]$ stays positive, thereby falsifying the condition of the outer if-statement of A4 and preventing $p_i$ from prematurely becoming neutral or announcing termination. In F1, since a new fault is detected, there is a chance that $p_i$ will become neutral. So $p_i$ checks if its index is the smallest among all processes which it has not detected as faulty. If so, $p_i$ becomes the coordinator and accordingly updates $\text{crdi}$ and $\text{parent}_i$. Otherwise, $p_i$ becomes a child of process $r = \min(P - DF_i)$, which is believed to be the coordinator (that of course may or may not be true, depending on whether or not $r$ is still alive). Note that in A1, one may substitute $P - DF_i - RF_i$ for $P - DF_i$ in the if-else-statement.

Before proceeding to prove the algorithm correct, we demonstrate how it works by examples.

**Example 1:** Consider a distributed system consisting of seven processes $p_1$, $p_2$, ..., $p_7$, in which $p_7$ is neutral and the others are engaged as shown in Fig. 4(a). If no process fails during the rest of the computation, the tree defined by the $\text{parent}_i$ variables will evolve as in the DS algorithm. For instance, if processes $p_6$, ..., $p_2$ successfully become neutral, they will leave the tree one after another. On leaving the tree, $p_2$, in particular, will send a signal to $p_1$. That will enable $p_1$ to become neutral and declare termination as soon as it becomes idle.

**Example 2:** Consider again the system as shown in Fig. 4(a), but this time $p_4$ fails, resulting in a structure as shown in Fig. 4(b). Because of $p_4$'s failure, $p_5$ is no longer able to propagate its signal along the original path ($p_4$, $p_3$, $p_2$, $p_1$). Our algorithm lets every surviving process (except for $p_1$ itself) attach to $p_1$ as a child on detecting $p_4$'s failure. Suppose that $p_3$ is the first process to detect the failure. It not only changes its parent to $p_1$ [see Fig. 4 (c)], but also sends a signal to its original parent, $p_2$, to acknowledge $p_2$'s messages including the engagement message [see A3, which is invoked after F1]. So $p_2$ no longer holds responsibility for $p_3$ status. Suppose now that $p_2$ becomes neutral and so signals $p_1$, while $p_3$ is still active [see Fig. 4(d)]. In Example 1, $p_2$'s signal to $p_1$ enabled $p_1$ to declare termination. Here, $p_2$'s signal to $p_1$ will not cause a premature declaration of termination because the information about $p_4$'s failure has been passed to $p_1$ along the path ($p_3$, $p_2$, $p_1$). Aware of $p_4$'s faulty status, $p_1$ will not declare termination until after it has received from each nonfaulty process a signal that reports $p_4$'s faulty status. In particular, when $p_1$ receives the signal (reporting $p_4$'s failure) from $p_3$, $p_1$ can ensure that $p_3$ (which was engaged by $p_2$) by finished. The following are some other subtle points about our algorithm:

- Although $p_7$ is neutral, it also has to send a signal to $p_1$ after detecting $p_4$'s failure, because $p_1$ expects to receive a signal from every nonfaulty process.
- After detecting $p_4$'s failure and before signaling $p_1$ (i.e., while waiting for a FAIL-FLUSH, confirm($p_4$) primitive), $p_7$ may possibly receive a basic message (either from $p_4$ or from any other process). If that happens, $p_7$ becomes active and remains as a child of $p_1$; it does not change parent to the sender of that message (see A2).

**C. Postponed Start**

As mentioned in Section III-B, both the CV and the V algorithms start to run only after they switch to detection mode. While the original DS algorithm does not have a “postponed start” feature, the LTD variant implements it in a way such that no control message is sent before the detector switches to detection mode; although the detector is actually running while in nondetection mode, its work involves only simple computation such as counting outgoing and incoming basic messages and is cost-negligible. For clarity and simplicity, we chose in the previous subsection not to include the postponed-start feature in the algorithm. In the following, we briefly discuss how to incorporate this feature into the algorithm.

A new local variable $\text{mode}_i$ is introduced to indicate whether process $i$ is in detection-termination (DT) or nondetection-termination (NDT) mode; and a new type of control message, called START, is used. All the actions in Fig. 3 remains the
same except for A0, A3 and A4. We need to modify A0 so that mode; gets initialized to NDT. The other two actions are modified as follows. First, a new condition "mode; = DT" is added to the (outer) if-statements of A3 and A4, so that no process in NDT mode will send any SIGNALS. Second, the event "pi changes from NDT to DT mode" is added to the two actions’ lists of guarding conditions, so that both actions will be invoked soon after a process enters DT mode. Thus, for instance, A3 will become:

A3: (When pi becomes idle or enters DT mode or whenever F1 is executed)
   if mode; = DT and pi is idle then . . ;

Two new actions are introduced to deal with START messages:

A6: (The coordinator spontaneously executes this action exactly once)
   if mode; = NDT then mode; := DT;
   for all j ∈ P - DFi - RF; - {i} do
      send a START message to pj ;

A7: (When pi receives a START message)
   if mode; = NDT then mode; := DT;

A6 indicates that the algorithm must be implemented in a way such that the process with crdi = true will execute the action once, and only once, unless it fails before having a chance to do so. In this action, the coordinator sends a START message to every nonfaulty process. Once a process receives a START message, it enters DT mode (A7). Note that the coordinator may enter DT mode by itself or by receiving a START message from a former coordinator. The reason for a coordinator already in DT mode to execute A6 is that the command "send a START message to every nonfaulty process" is not atomic. If the former coordinator from which the present coordinator received its START message had crashed in the course of sending START messages, then some processes may not receive a START from it. So, the new coordinator has to send STARTs to ensure that every nonfaulty process will eventually enter DT mode.

In the next section we will establish the correctness of the algorithm as presented in Fig. 3. It will be not hard to see that, with the above modifications, the algorithm still can be proved along the same line of reasoning, requiring only a few minor changes.

V. CORRECTNESS PROOF

We establish the correctness of the algorithm by proving that 1) if some process declares termination at some point of time, then the basic computation is really terminated at that moment, 2) after the basic computation terminates, some process in the system will eventually declare termination (unless all processes crash before any process has a chance to detect the termination).

As was mentioned earlier, we assume that the termination detector will be superimposed on the basic system. The computation of a basic system and that of a superimposed termination detector together are called a combined computation, which is the computation of n concurrent processes.

As is [16], we adopt the common approach of defining an execution of a concurrent algorithm to be a sequence of atomic actions in which concurrent actions of separate processes are assumed to be interleaved in an arbitrary manner. It is thus necessary to specify which of the algorithm’s operations are atomic. There is a Folk Theorem, in [16] that reads "when reasoning about a multiprocess program, one can combine into one atomic action any sequence of operations that contains only a single access to a single shared variable." Applying this theorem, we model the combined computation (of the basic system and the superimposed termination detector together) as a single sequence of atomic events, where each event is one of the following.

AE1 A process fails.
AE2 A process becomes idle.
AE3 A process issues or receives a FAULT-DETECT or FAIL-FLUSH primitive.
AE4 A process performs a sequence of operations that involve only local variables.
AE5 A process receives or sends a single (basic or control) message and performs a sequence of operations that involve only local variables.

In particular, we regard the body of action A1 together with its "guarding condition" as an atomic event (of type AE5); the same for A2 and A5. Each iteration of the for-loop of A3 (i.e., sending a signal and updating ini [j]) is an AE5 event. In A4, sending a signal and updating ini [parenti] and parenti, is also an AE5 event.

Conventions: For ease of presentation, we adopt the following conventions throughout this section.

- We assume the existence of an external clock to which the processes have no access. If C = (e1, e2, . . . , em) is a combined computation, let (t0, t1, . . . , tm) be any sequence of points of time (according to the external clock) such that ei occurs between ti-1 and ti.
- When we say any time, we mean any time between two consecutive events, not within an event.
- When we refer to a local variable of a faulty process, we mean its value as of the time when the process failed.
- For any variable x, x(t) indicates the value of x as of time t; t may be omitted if no confusion may arise.

With these conventions, we formally state as follows the two properties, soundness and completeness, of our algorithm that will be established later.

Soundness: If (e1, e2, . . . , ed) is a combined computation such that some process declares termination in event ed, then the basic computation has really terminated by time td-1.

Definition 4: A combined computation C is said to be complete if either of the following two statements is true.

1) C contains an infinite number of events.
2) C = (e1, . . . , em) is finite and satisfies both of the following two conditions:
   - there is at least one nonfaulty process at time tm;
   - unless one of these processes crashes (after tm),
A. Message Counting and Process Status

Let \((e_1, e_2, \ldots, e_m)\) be any computation. With respect to this computation, we first state two simple facts concerning the numbers of unacknowledged basic messages. Consider any two nonfaulty processes \(p_i\) and \(p_j\). Recall that \(\text{out}_i[j]\) indicates the number of basic messages that \(p_i\) has sent to \(p_j\) for which \(p_j\) has not received an acknowledgment, and that \(\text{in}_i[j]\) is the number of basic messages from \(p_i\) that \(p_j\) has received but has not yet acknowledged. Also recall that the \(c\) parameter of a SIGNAL\((c, -)\) message is meant to acknowledge \(c\) basic messages. Let \(b\) be the number of basic messages in the channel from \(p_i\) to \(p_j\) (which have been sent by \(p_i\) but not yet received by \(p_j\)). Let \(\text{SIGNAL}(c_1, -)\), \(\text{SIGNAL}(c_2, -), \ldots\), \(\text{SIGNAL}(c_q, -)\) be all the SIGNAL messages in the channel from \(p_i\) to \(p_j\) (which have been sent by \(p_j\) but not yet received by \(p_i\)). We immediately have the following.

**Lemma 1:** At any time during the computation, if \(p_i\) and \(p_j\) are both nonfaulty, then

\[
\text{out}_i[j] = b + \text{in}_j[j] + \sum_{k=1}^{q} c_k.
\]

**Lemma 2:** If process \(j\) receives a basic message from process \(i\) in event \(e_k\) and does not send any SIGNAL to \(i\) during the period \([t_k, t'_k]\), where \(k < k'\), then \(\text{out}_i[j] > 0\) at time \(t'_k\).

The terms neutral and engaged were employed in the description of our algorithm without formal definition. They basically mean the same things as in the DS algorithm, but there is a subtle difference. In the DS algorithm, a process \(i\) is neutral if \(p_i\) is idle and \(\text{in}_i = \text{out}_i = 0\). We take this to mean that \(p_i\) has not received any basic message but has not yet sent any SIGNAL to its parent. If \(p_i\) is not neutral, then it is engaged, which means that \(p_i\) has sent a SIGNAL to its parent. In our algorithm, a process \(i\) is neutral if \(p_i\) is idle and \(\text{in}_i = 0\), \(\text{out}_i \approx 0\) (i.e., \(\text{out}_i[j] = 0\) or \(\epsilon\) for every \(j\)), and \(p_i\) has not reported (via a SIGNAL) to its parent \(p_i\). (The difference is due to our design that \(p_i\) may increase the value of \(\text{out}_i[x]\) and adopt a new parent on detecting fault \(x\) in A4.) The above statement is not a definition of neutral, because the expression “\(p_i\) has reported to its parent” is not unambiguous and it does not apply very well to the coordinator, who has no parent. The neutral/engaged status of a process is formally defined below.

**Definition 5:** Every process is neutral before it gets started, and becomes engaged in the event of its initialization (i.e., A0 action). An engaged process becomes neutral in the last event of a “successful” A4 action. A neutral process becomes engaged in an A2 action. Should a process fail, its engaged-neutral status stays unchanged thereafter.

In the above, a “successful” A4 action is one in which the condition of the outer if-statement is true and so the inner if-command is executed.

The notion of the neutral/engaged status of a process plays a critical role in our correctness proof. It follows from A4 and from the above definition that if process \(i\) is neutral, then it is idle, and \(\text{in}_i = 0\), and \(\text{out}_i \approx 0\).

**Definition 6:** A neutral process \(i\) is said to be strongly neutral if \(\text{out}_i = 0\), and weakly so otherwise.

Note that a strongly neutral process \(i\) becomes weakly neutral on setting \(\text{out}_i[x] := \epsilon\) in F1, in which a FALL-FLUSH.request\((x)\) is also issued. A weakly neutral process becomes engaged on receiving a basic message. If it does not get engaged in this way, then after all its FALL-FLUSH.requests have been confirmed, it will become strongly neutral again.

B. Structure of Engaged Processes

**Definition 7:** At any time \(t\), the root of the system or system root, denoted by \(r(t)\) (or simply by \(r\) if there is no confusion), is the process whose index is smallest among all nonfaulty processes, i.e., \(r(t) = \min\{i: i\text{ is nonfaulty at } t\}\).

The system always has a root. When the current root fails, another process immediately succeeds (by definition) as the root. The new root, however, is not aware of its having become the root until later when it detects the failure of the previous root. When the root recognizes that it is the root, then it becomes the “coordinator.”

**Definition 8:** The system root \(r\) is also called a coordinator if \(\text{crd}_r = true\).

The variables \(parent\) define a directed graph \(G\) with vertex set \(V = P\) and edge set \(E = \{(i, parent_i): parent_i \neq NULL\}\). The edges of the graph indicate whom a process will report to when it becomes strongly neutral. Through \(G\) can be directly used in our correctness proof, we found it more convenient to deal with the following subgraph of \(G\) that includes only essential edges and, as will be seen, has a more regular structure than \(G\).

**Definition 9:** The hierarchy graph of the system is the directed graph \(H = (V, E)\) where \(V = P\) and \(E = \{(i, parent_i): i \neq r, parent_i \text{ is engaged and nonfaulty}\}\).

Note that \(parent_i \neq NULL\) if \(p_i\) is engaged and is not the system root. So \(H\) is well-defined and is really a subgraph of \(G\). The edges of \(G\) that are missing from \(H\) are those corresponding to nonessential or delayed edges and, as will be seen, has a more regular structure than \(G\).

**Definition 10:** An edge \((i, parent_i)\) in \(H\) is said to be a fault-detection edge due to \(p_i\) if \(parent_i\) was set in F1 because of \(p_i\) detecting the failure of \(p_\_\). An engagement edge is one that was created in A2 when \(p_i\) received a basic message.

It should be clear from the algorithm that an edge in \(H\) is either an engagement edge or a fault-detection edge.

\(H\) is a dynamic graph, its structure changing during the computation. In the following, we list all the events that may possibly change the structure of \(H\), and then show that \(H\) is
always a forest. (A forest is a number of mutually disjoint trees.) There are six events that may add edges to or delete edges from $H$:

**E0:** A process $i \neq r$ initializes its local variables in execution of $A0$. (Edge $(i, 1)$ is introduced.)

**E1:** The present system root fails.

**E2:** An engaged nonfaulty process $x \neq r$ fails. (The edge emanating from $x$ disappears from $H$.)

**E3:** An engaged nonfaulty process $i \neq r$ executes the inner if-statement of $A4$ and becomes neutral. (The outbound edge of $i$ is deleted.)

**E4:** A neutral nonfaulty process $i \neq r$ receives a basic message and performs $A2$ (and thus becomes engaged). (A new edge $(i, \text{parent}_i)$ is introduced.)

**E5:** An engaged process $i \neq r$ updates the value of $\text{parent}_i$ (and other local variables) in $F1$. (A new edge $(i, \text{parent}_i)$ replaces on old one, if the value of $\text{parent}_i$ changes in $F1$.)

Note that each of these events is atomic and belongs in one of the event categories as listed in the beginning of Section 5. For instance, $E4$ is in category $A4$, and $E5$ in $A4$. Also note that in event $E1$ the edge from the new root, if it exists, is, by definition, immediately excluded from $H$, even though $\text{parent}_r$ may still have a non-NULL value. Defining $H$ like this results in a simple structure that would otherwise be hard to describe. The following lemma shows that $H$ is a forest of trees. The four categories (of trees) described in the lemma are not necessarily disjoint: for instance, a tree in category 3 may also be in category 2 and/or category 4, because a former coordinator could be engaged or neutral. Fig. 4 shows the structure of an example $H$ at various times.

**Lemma 3:** At any time $t$ during the computation, $H$ is a forest comprising the following trees:

1. A single tree rooted at the system root—called the main tree;
2. Zero or more trees rooted at engaged, faulty nodes;
3. Zero or more trees rooted at former coordinators;
4. Zero or more neutral isolated nodes (faulty or nonfaulty).

**Proof:** Let $(e_1, e_2, \ldots, e_m)$ be any computation. At $t_0$, before any event occurs, every process is neutral and there is no edge at all in $H$. So $H$ contains the main tree and $n - 1$ trees of type 4—the lemma is true at the beginning. (The main tree is also of type 4, but that doesn’t matter.)

Assume that the lemma is true at time $t_{k-1}$, before event $e_k$. We show that the lemma remains valid after the event (i.e., at $t_k$). It suffices to consider the events $E0$–$E5$ since other events do not change the structure of $H$.

**E0:** A process $p_i, i \neq r$, executes $A0$, resulting in edge $(i, 1)$ being added to $H$. Before the event, $p_i$ constitutes a single-node tree in category 4. In the event, $p_i$ joins the tree rooted at $p_1$, which is in category 3 or 1 depending on whether $p_1$ is faulty or nonfaulty at $t_{k-1}$. The lemma holds at $t_k$.

**E1:** The present root of the system $r$ fails. In this case, a process $r'$ succeeds as the new system root and edge $(r', \text{parent}_{r'})$, if it existed at $t_{k-1}$, now disappears from $H$ (for $H$ does not include any edge emanating from the system root). Thus, after the event, the tree rooted at $r$ becomes a tree in category 3, and the subtree rooted at $r'$ becomes the main tree. The lemma is still true.

**E2:** An engaged nonfaulty process $x \neq r$ fails. In this event, edge $(x, \text{parent}_x)$ disappears from $H$ (by definition $H$ does not include outgoing edges of faulty processes) and the original subtree rooted at $x$ now becomes an individual tree with $x$ at the root. The new tree is in category 2. The tree from which $x$ breaks off is still a tree. So the lemma remains valid after the failure of $x$.

**E3:** An engaged nonfaulty process $i \neq r$ becomes neutral in the last event of $A4$. The edge $(i, \text{parent}_i)$ is deleted from $H$. The precondition out$_i = 0$ together with the assumption $i \neq r$ implies that node $i$ has no child in $H$ (for, by Lemma 1, $in_j[i] = 0$, and thus $\text{parent}_j \neq i$ for every nonfaulty process $j$.) Thus, after the event, $i$ becomes a neutral isolated node and the lemma holds.

**E4:** A neutral nonfaulty process $i \neq r$ receives a basic message from $p_j$ (and hence executes $A2$ ) and becomes engaged. There are two cases depending on whether $p_i$ is strongly or weakly neutral.

* $\text{parent}_i = \text{NULL}$ before $A2$ ($p_i$ is strongly neutral)—Process $j$, whose message makes $i$ engaged, must be engaged itself (whether it is faulty or nonfaulty) and therefore belongs in a tree in categories 1–3. Setting $\text{parent}_j$ to $j$ attaches the tree rooted at $i$ to the tree containing $j$ without violating the lemma.

* $\text{parent}_i \neq \text{NULL}$ before the action ($p_i$ is weakly neutral)—In this case, $\text{parent}_j$ points either to the present root or to a former root of the system. The edge $(i, \text{parent}_i)$, which was not in $H$ because of $i$’s neutral status, is now added to $H$. It connects the tree containing $i$ either to the main tree or to a tree in category 3. The lemma remains valid

**E5:** An engaged process $i \neq r$ performs action $F1$. This results in process $i$ changing its parent in $H$ either to the present system root $r$ or to a former one. Again, the lemma remains true.

**C. Completeness**

The following theorem establishes the completeness property of our algorithm.

**Theorem 1:** Given any complete combined computation $C = (e_1, e_2, \ldots, e_t, \ldots)$ in which the basic computation terminates in event $e_t$, there is an event $e_d$ in $C$ where $d > t$, such that some process declares termination in $e_d$.

**Proof:** Let $C = (e_1, e_2, \ldots, e_t, \ldots)$ be a complete combined computation in which the basic computation terminates in event $e_t$. We first show that $C$ contains only a finite number of events.

As the basic computation has terminated, events occurring after $e_t$ in $C$ are those of the termination detector. Each action of our algorithm consists of a finite number of events. Thus, $C$ is finite if the number of actions occurring after $e_t$ is finite. Actions $A0$, $A1$, and $A2$ do not occur after $e_t$. At most $O(n)$ $F1$s (and the same for $F2$s, and $A3$s) may occur at each process after $e_t$. There are at most $O(M + n^2)$ in-transit SIGNALs—$O(M + n)$ for acknowledging basic messages and $O(n^2)$ for fault reporting—so the number of $A5$s that may oc-
cur is finite. Since the execution of A4 depends on the occurrence of A3, A5, or F2, the number of A4 occurrences is also finite. This establishes that the number of events is C in finite.

Let C = \( (e_1, e_2, \ldots, e_t, \ldots, e_m) \). Since C is a finite complete computation, by Definition 4 there is at least one nonfaulty process at time \( t_m \) and no other events will occur after time \( t_m \) unless some currently nonfaulty process will crash (after \( t_m \)). Obviously, \( m > t \), since a basic computation always ends with either a process becoming idle or with a process crashing, in either case there being actions after \( t \). Let \( F \) be the set of all faulty processes as of time \( t_m \). By condition i), \( F \neq \emptyset \). Let \( e_k \) be the last event in C that occurs at a process \( i \not\in F \). Due to condition ii), the following observations are true at \( t_k \), the time before event \( e_k \).

1) Process \( i \not\in F \) has detected all faults in \( F \) (i.e., \( DF_i (t_k) = F \)), and all those \( F1, A3, \) and \( F2 \) actions associated with detecting of the failures in \( F \) have been finished. This also means that process \( i \) has no unfinished (unconfirmed) fail-flush requests.

2) The A3 action triggered because of process \( i \) being idle has been finished.

3) No basic message or SIGNAL is in transit to \( i \).

From these observations, one may readily verify that \( e_k \), is an event associated with A4. Defining a leaf to a node without any child, we make two claims.

Claim 1: If in computation \( C \) process \( i \not\in F \) is a leaf (in the hierarchy graph \( H \)) at time \( t_{k-1} \) and remains so thereafter, then the inner if-elseif-statement of A4 is executed in event \( e_k \).

Proof of Claim 1: It suffices to show that the condition of the outer if-statement of A4 is satisfied at time \( t_{k-1} \). Obviously, process \( i \) is idle at \( t_{k-1} \) and \( e_k \) is its last event. Since \( e_k \) is an event associated with action A4, it follows from observation 1 that \( RF_i (t_{k-1}) \subseteq F = DF_i (t_k) = DF_i (t_{k-1}) \), and that \( in_i [x] = 0 \) and \( out_i [x] = 0 \) for all \( x \in F \). Observation 2 implies that \( in_i [j] = 0 \) for all \( j \neq \text{parent} \). Since \( i \) is a leaf, all basic messages that \( i \) has sent to other nonfaulty processes must have been acknowledged; i.e., \( in_j [i] = 0 \) for all nonfaulty \( j \), and no basic message is in transit from \( i \) to any nonfaulty \( j \). (Otherwise, \( in_j [i] \neq 0 \) for some nonfaulty \( j \), then \( j \) becomes idle or when it becomes a coordinator, \( i \) will send a SIGNAL to \( j \), contradicting the assumption that \( e_k \) is the very last event at \( i \). Similarly, if there exists a basic message in transit from \( i \) to a nonfaulty \( j \), then either \( j \) becomes a child of \( i \), or \( j \) will send a SIGNAL to \( i \) upon becoming idle or upon becoming a coordinator—a contradiction in either case.) By Observation 3 and Lemma 1, \( out_i [j] = 0 \) for all \( j \not\in F \). Thus, at time \( t_{k-1} \), \( out_i = 0 \) and \( in_i [j] = 0 \) for all \( j \not\in \text{parent} \), and the claim is proved.

Claim 2: At time \( t_{m-1} \), \( r \) is a leaf in \( H \).

Proof of Claim 2: Let \( e_l \) be any event between \( e_2 \) and \( e_m \) such that by time \( t_l \) not only all faults in \( F \) have been detected by all nonfaulty processes, but all F1 and F2 actions associated with these fault detection have been finished. After time \( t_l \), no edge will be added to \( H \); if \( H \) changes shape, it will be that edges are removed in \( E_3 \) events. Consider any leaf \( i \) in the main tree (the one rooted at \( r \)) as of time \( t_l \). If \( i = r \), we have verified \( r \)’s leaf status. If \( i \neq r \), we show that i’’s very last event (i.e., \( e_k \)) occurred after time \( t_l \). Assume otherwise that \( e_k \) occurred before \( t_l \). At that time, process \( i \) was already a leaf. This is because 1) \( i \)’s former children (former as relative to time \( t_l \) at which \( i \) has no child), if any, each sent a SIGNAL to \( i \) when terminating their child/parent relationship with \( i \); and 2) these signals obviously had arrived at \( i \) before the occurrence of \( e_k \). Thus, by Claim 1, \( i \) executed the inner if-statement of A4 and terminated its child/parent relationship with its parent, contradicting the fact that \( i \) still has a parent at \( t_l \). So, \( e_k \) could not have occurred before \( t_l \); it must have occurred after \( t_l \) (while \( i \) is a leaf). By Claim 1 again, \( i \) executes the inner if-statement of A4 and leaves the main tree in event \( e_k \). Applying this argument to each leaf leads to the conclusion that every leaf in the main tree eventually leaves the tree. So by the time \( e_m \) occurs, \( r \) itself has become a leaf, and claim 2 is proved.

Now we show that process \( r = \min(F - F) \) declares termination in event \( e_m \). First, \( r \) already recognizes itself as the coordinator by time \( t_{m-1} \), since all faults have been detected. Second, \( e_m \) is an event at process \( r \), as otherwise there would be a SIGNAL in transit at time \( t_m \) towards a \( j \not\in F \). Third, by Claim 1 and Claim 2, process \( r \) executes the elseif statement of A4. So we only have to show that the condition “\( \forall j \in P - DF_r - \{r\}; NumFaults, [j] = \|DF_r \| \)” holds true at time \( t_{m-1} \), where we already know \( DF_r = F \). Note that every nonfaulty process except for \( r \) has once changed its parent to \( r \) (after detecting the last fault in \( F \)), but now \( r \) has no child at time \( t_{m-1} \). So, each process \( j \in P - F - \{r\} \) must have sent a SIGNAL(=S) to \( r \) with \( S = F \), and, by observation 3, the signal has been received by \( r \) by time \( t_{m-1} \). So \( NumFaults, [j] = \|F \| \) for all \( j \in P - F - \{r\} \) at time \( t_{m-1} \) and, therefore, \( r \) declares termination in event \( e_m \). (Note that \( r \) may or may not be the only process that declares termination in C; some other processes—former coordinators—might have done so before \( r \).) This proves the theorem.

D. Soundness

We now prove the soundness of our algorithm, which, as will be seen, is a much harder task than providing the completeness.

Outline: Suppose in a combined computation \( (e_1, e_2, \ldots, e_d) \) some process \( r \) declares termination in event \( e_d \). (Process \( r \) must be the coordinator of the system, since only the coordinator may declare termination, owing to the design of A4). We show that the basic computation has terminated by time \( t_d \). As \( e_d \) is not an event of the basic computation, it will then follow that the basic computation had actually terminated by time \( t_{d-1} \).

We shall establish three important properties of the algorithm:

1) At time \( t_d \), if \( x \in DF_r \) (i.e., \( x \) has been detected by \( r \) to be faulty), then every nonfaulty process has reported fault \( x \) to \( r \) in a SIGNAL.
2) At time \( t_d \), if \( x \) is an engaged faulty process, then \( x \in DF_r \).
3) At time \( t_d \), there is not edge in the hierarchy graph H.
From these properties, the soundness of the algorithm can be readily proved as follows.

**Theorem 2:** If some process declares termination, then the basic computation is terminated at that time.

**Proof:** Assume that in a computation \((e_1, e_2, \ldots, e_d)\) some process \(r\) declares termination in event \(e_d\). Property 3 and Lemma 3 together imply that, at time \(t_d\), all nonfaulty processes are neutral and hence idle. (Note that \(r\) is also neutral because it executes action \(A4\) and declares termination.) As neutral processes have no unacknowledged basic messages (by definition and Lemma 1), only a channel \([x, i]\) from an engaged faulty process \(x\) to a nonfaulty one \(i\) may possibly contain deliverable basic messages. Properties 1 and 2 together indicate that \(i\) has reported \(x\) to \(r\) in a SIGNAL, which, according to the algorithm, may happen only if \(i\) has received a \(FAIL\)-\(FLUSH\).confirm\((x)\) primitive (see \(F2\)). Thus, channel \([x, i]\) is free from basic messages. By definition, the basic computation is terminated at \(t_d\).

So, all what we need to do is to prove the three mentioned properties.

**Property One:** This section establishes property one: If \(x \notin DF_r(t_d)\), then every nonfaulty process (except for \(r\) itself) has reported \(x\)'s failure to \(r\) through a SIGNAL. (Recall that \(DF_r(t_d)\) is the value of \(DF_r\) at time \(t_d\).)

**Lemma 4:** For each \(i \in P - DF_r(t_d) - \{r\}\), process \(r\) has received by time \(t_d\) a SIGNAL \((-, S_i)\) from \(i\) with \(S_i = DF_r(t_d)\).

**Proof:** Suppose \(i \in P - DF_r(t_d) - \{r\}\). When \(r\) declares termination in event \(e_d\), these two conditions hold: \(RF_r \subseteq DF_r\) and \(|DF_r| = NumFaults_r[i]\). From the statements of \(A5\), we have \(NumFaults_r[i] \leq |RF_r|\). Combining all these conditions yields \(|RF_r| = |DF_r| = NumFaults_r[i]\), which implies that \(r\) has received a SIGNAL \((-, S_i)\) from \(i\) such that \(S_i = RF_r = DF_r\).

**Property Two:** We establish property two in this section: if a faulty process \(x\) is engaged when it fails, then \(x \notin DF_r(t_d)\). The task turns out to be very difficult. Our strategy is to identify all faults that are definitely detected by \(r\) at time \(t_d\). We will define a class of faults called critical faults which includes engaged faults, and show that all critical faults are detected by \(r\) at time \(t_d\).

Let \((e_1, e_2, \ldots, e_d)\) be the computation in which \(r\) declares termination in event \(e_d\). If \(r = 1\), let \(c = 0\); otherwise, let \(c = 1\). Then, there are \(t_c\) and \(t_d\) during which \(r\) is the system root.

**Definition 11:** At any time \(t \in [t_c, t_d]\), a faulty process \(x\) is said to be a critical fault if at least one of the following is true:
- \(x\) is a former system root (i.e., \(x \leq r\), where \(r\) is the system root at time \(t_d)\);
- \(x\) was engaged at the time of its failure;
- \(x\) was once detected by some engaged process (i.e., there was a time \(t' \leq t\), and a process \(i\), such that \(i\) was engaged at \(t'\) and \(x \in DF_i(t')\)).

Let \(CF(t)\) denote the set of all critical faults as of time \(t\). \(CF(t)\) is a “nondecreasing” function of \(t\), since a fault remains critical once it is critical.

Our goal here is to show \(CF(t_d) \subseteq DF_r(t_d)\). The question facing us is: Under what conditions can one assert that a certain fault \(x\) will definitely be in \(DF_r(t_d)\)? The definition of coordinator suggests one such condition: every former system root is in \(DF_r(t_d)\), since by time \(t_d\) process \(r\) already recognizes itself as the coordinator (or it wouldn’t have declared termination). We show some other conditions in Lemmas 7 and 8. These conditions will be used to prove property two in Lemma 9.

Call an ordered pair of processes \((a, b)\) a semi-engagement edge if \(in_a[b] > 0\). This term is only introduced to facilitate the next definition. Note that an engagement edge is always a semi-engagement edge, and a semi-engagement edge may or may not be an edge in the hierarchy graph.

**Definition 12:** A sequence of \(t \geq 1\) nonfaulty processes \((a_1, \ldots, a_l)\) is said to be a semi-engagement chain if the first edge \((a_1, a_2)\), if it exists, is a semi-engagement edge and the rest of the edges (i.e., \((a_2, a_3), \ldots, (a_{l-1}, a_l)\)), if they exist, are all engagement edges.

If \((a_1, \ldots, a_l)\) is a semi-engagement chain, \(a_{l+1}\) is said to be the predecessor of \(a_l\) in the chain. In most cases, the predecessor of a node is also its parent in the hierarchy graph. By definition, every nonfaulty process \(a\) alone constitutes a semi-engagement chain \((a)\), whether \(a\) is neutral or engaged. If a semi-engagement chain has a nonzero length, every process on it is engaged and nonfaulty.

**Definition 13:** Denote by “\(a \not\leq b\) at \(t_k\)” that at time \(t_k\) process \(x\) is faulty and there exists a semi-engagement chain \(C\) from \(b\) to \(a\) such that at least one of the following statements is true:
- \(x \in DF_b(t_k)\).
- All edges in \(C\) are engagement edges and \((a, b)\) is a semi-engagement edge.
- All edges in \(C\) are engagement edges and there is some SIGNAL\((c, S)\) in transit for \(b\) such that \(c > 0\) and \(x \in S\).

The intuitive meaning of “\(a \not\leq b\) at \(t\)” is that, if no process on the chain (from \(b\) to \(a\)) ever fails after \(t\), then the information “\(x\) is faulty” will propagate along the chain and eventually reach \(a\). This property is due to the following lemma that indicates that if \(a \not\leq b\) holds at \(t_k\), then after time \(t_k\) \(b\) may be neutral only if it has detected \(x\)'s failure.

**Lemma 5:** If \(a \not\leq b\) holds at time \(t_k\), then for all \(k', k \leq k' \leq d\), either \(x \in DF_b(t_k)\) or \(b\) is engaged at \(t_k\).

**Proof:** Assume \(a \not\leq b\) holds at \(t_k\). Also, assume \(x \not\in DF_b(t_k)\), or we are done. Thus, condition b) or c) of Definition 13 holds. In either case, \(b\) still has unacknowledged outgoing messages (i.e., \(out_b[j] \geq 1\) for some \(j\)) as of time \(t_k\). \(b\), \(b\) is engaged at time \(t_k\), and it remains so as long as \(out_b[j] \geq 1\) for some \(j\). In case b), \(out_b[x]\) remains positive until \(b\) has detected \(x\)'s failure and subsequently finished the corresponding F1 and F2 actions. In case c), \(out_b\) may become 0 only after \(b\) has received the SIGNAL containing \(x\). Thus, \(b\) may become neutral only after it has detected \(x\)'s failure.

In the algorithm, a SIGNAL is sent either in A3 or A4. A SIGNAL sent in an A4 action is called a major SIGNAL.

**Definition 14:** If \(a \not\equiv x\) and \(a \not\equiv r\), we denote by “\(r \not\equiv a\) at time \(t_k\)” that by time \(t_k\) process \(a\) has not yet sent a major
SIGNAL(\(\neg, S\)) \(\) to r with \(z \in S\).

**Definition 15.** Write "\(r \xrightarrow{z} a \xleftarrow{z} b \) at \(t_k\)" iff both \(r \xrightarrow{z} a\) and \(a \xleftarrow{z} b\) hold true at time \(t_k\).

Intuitively, if \(r \not\xrightarrow{z} a \xleftarrow{z} b\) at \(t_k\) and no process in the chain (from \(b\) to \(r\)) ever fails after time \(t_k\), then the information about \(a\)'s failure will propagate along the chain to \(a\) and be reported together with \(z\) by \(a\) to root \(r\). Before we proceed to prove this, we first establish some conditions that ensure \(r \not\xrightarrow{z} a\).

**Lemma 6:** If any one of the following statements is true, then \(r \not\xrightarrow{a} a\) holds at time \(t_k\).

- (a) \((a, j)\) is a fault-detection edge in \(H(t_k)\) due to a faulty process \(z\).
- (b) \((a, z)\) is an edge in \(H(t_k)\).
- (c) Process \(a\) becomes engaged in event \(e_k\) because of receiving a basic message from process \(z\).

**Proof:** In case (a), edge \((a, j)\) would no longer exist in \(H(t_k)\) if \(a\) has reported the failure of \(z\) in a major SIGNAL to \(r\). In case (b), \(a\) has not detected \(z\)'s failure as of time \(t_k\). In case (c), one observes that \(a\) may report \(z\) in a major SIGNAL only after it has received the FAIL-FLUSH.confirm(\(z\)). But after the FAIL-FLUSH.confirm(\(z\)), no basic message from \(z\) will be received by \(a\).

We are now ready to show that under certain conditions it can be concluded that \(x \in DF_r(t_k)\). This is done in the next two lemmas.

**Lemma 7:** Assume \(z \in DF_r(t_k)\) and \(a \not\xrightarrow{z} a\). If, by time \(t_k\), \(r\) has not received a major SIGNAL(\(\neg, S\)) with \(z \in S\) from process \(a\), then \(a \in DF_r(t_k)\). In particular, if \(r \not\xrightarrow{z} a\) at time \(t_k\), then \(a \in DF_r(t_k)\).

**Proof:** We show that if \(a \in DF_r(t_k)\), where \(a \not\xrightarrow{z} a\), then \(r\) has received from \(a\) by time \(t_k\) a major SIGNAL(\(\neg, S\)) with \(z \in S\) for all \(z \in DF_r(t_k)\).

By Lemma 4, process \(r\) has received a SIGNAL(\(\neg, S\)) with \(S = DF_r(t_k)\) from \(a\). This SIGNAL is evidently a major one (i.e., at the time the SIGNAL is sent, \(parent_a = r\)), since \(parent_a\) was set to \(r\) after \(a\) had detected all faults in \(S\).

We next show that if \(r \xleftarrow{z} b\) or \(r \not\xrightarrow{z} a \xleftarrow{z} b\) holds at some point of time between \(t_c\) and \(t_d\), where \(z \in DF_r(t_k)\), then \(x \in DF_r(t_k)\).

**Lemma 8:** If \(r \xleftarrow{z} b\) or \(r \not\xrightarrow{z} a \xleftarrow{z} b\) at time \(t_k \in [t_c, t_d]\), where \(z \in DF_r(t_k)\), then \(x \in DF_r(t_k)\).

**Proof:** The proof is by induction on \(t_k\), for \(k = d, d - 1, \ldots, c\). Recall that event \(e_k\) occurs between time \(t_{k-1}\) and time \(t_k\).

**Induction Base:** Suppose \(r \xrightarrow{z} b\) \(r \not\xrightarrow{z} a \xleftarrow{z} b\) holds at times \(t_d\) for some \(z \in DF_r(t_d)\). The latter case is impossible since by Lemma 7 every nonfaulty process \(a\) (in particular) has reported \(z\) to \(r\) by time \(t_d\). In the case of \(r \xrightarrow{z} b\), it must be \(r \xleftarrow{z} r\) as \(r\) is neutral at \(t_d\). It follows from Lemma 5 that \(x \in DF_r(t_k)\).

**Induction Hypothesis:** Assume the lemma is true for \(t_{k+1}, t_{k+2}, \ldots, t_d\) where \(k \geq c\).

**Induction Step:** We show the lemma true for \(t_k\). Thus, assume "\(r \xrightarrow{z} b\) at time \(t_k\)" or "\(r \not\xrightarrow{z} a \xleftarrow{z} b\) at time \(t_k\)" with \(z \in DF_r(t_k)\). We need to show \(x \in DF_r(t_k)\). Let \(C\) be the semi-engagement chain involved in \(r \xrightarrow{z} b\) or in \(a \xleftarrow{z} b\), and for simplicity let \(k' = k + 1\). Consider all possible cases for \(e_{k'}\), the event occurring between \(t_k\) and \(t_{k'}\).

**E1:** The present coordinator \(r\) crashes in event \(e_{k'}\). This case is impossible by definition of \(t_k\).

**E2:** A process \(y \not\xrightarrow{z} r\) in \(C\) fails in event \(e_{k'}\). By Definitions 13(b) and 14, we have the following:

1. If \(r \xleftarrow{z} b\) at time \(t_k\), then \(r \not\xrightarrow{z} p\) at time \(t_{k'}\), where \(p\) is the predecessor of \(y\) in chain \(C\).
2. If \(r \not\xrightarrow{z} a \xleftarrow{z} a\) at \(t_k\) and \(y \not\xrightarrow{z} a\), then \(r \not\xrightarrow{z} a \xleftarrow{z} y\) at \(t_{k'}\), where \(p\) is the predecessor of \(y\) in chain \(C\).
3. If \(r \not\xrightarrow{z} a \xleftarrow{z} b\) at \(t_k\) and \(y = a\), then \(r \not\xrightarrow{z} a\) at \(t_{k'}\) which implies \(r \not\xrightarrow{z} a\) at \(t_d\).

In the first two cases, the induction hypothesis indicates that \(y \in DF_r(t_{k'})\). In the third case, \(y(=a)\) is also in \(DF_r(t_{k'})\) as implied by Lemma 7. So, in all cases, \(y \in DF_r(t_{k'})\) and as \(y\)'s failure occurs after \(x\)'s, it follows from Lemma 7 that \(x \in DF_r(t_{k'})\).

**E3:** An engaged process in \(C\) either sends a SIGNAL(\(c, S\)) to its predecessor \(p\) in the chain or becomes neutral (or both). This process must be \(b\), for the only process in a semi-engagement chain that may send a SIGNAL to its predecessor or become neutral is the process where the chain starts. First observe that if \(b\) becomes neutral in \(e_{k'}\), then \(x \in DF_b(t_{k'})\) (by Lemma 5), which implies \(x \in DF_b(t_{k'})\) since \(e_{k'}\) occurs in an A4 action in which \(DF_b\) does not change value. There are three different cases which we need to consider separately:

1. \(b\) becomes neutral in \(e_{k'}\) without sending a SIGNAL to its predecessor in \(C\). This case is possible only if \(b\) has no predecessor in \(C\) (i.e., \(b = r\) or \(b = a\)).

   - If \(r \xrightarrow{z} r\) at \(t_k\), then \(x \in DF_r(t_k)\).
   - If \(r \not\xrightarrow{z} a \xleftarrow{z} a\) at \(t_k\), then \(a\) sends a SIGNAL to \(parent_a\) in \(e_{k'}\). The SIGNAL contains \(x\). If \(parent_a \not\xrightarrow{z} r\), then \(r \not\xrightarrow{z} a \xleftarrow{z} a\) still holds at \(t_{k'}\), and hence \(x \in DF_r(t_{k'})\) by the induction hypothesis. Otherwise, suppose \(parent_a = r\) (so the SIGNAL is destined for \(r\)). If the SIGNAL is received by \(r\) before time \(t_d\), then \(x \in DF_r(t_{k'})\). If it is not received before \(t_d\), then no SIGNAL from \(a\) that contains \(z\) is received before \(t_d\).

2. \(b\) sends a SIGNAL(\(c, S\)) to \(p\) and becomes neutral. In this case, \(c > 0\) (because \(im_{b}[p] > 0\) at \(t_k\) and \(x \in S\)). By Definition 13(c), the following hold:

   - If \(r \xleftarrow{z} b\) at \(t_k\), where \(r \not\xrightarrow{z} b\) then, \(r \not\xrightarrow{z} p\) at \(t_{k'}\).
   - If \(r \not\xrightarrow{z} a \xleftarrow{z} b\) at \(t_k\), where \(a \not\xrightarrow{z} b\), then \(r \not\xrightarrow{z} a \xleftarrow{z} p\) at \(t_{k'}\).

   In both cases, \(x \in DF_r(t_{k'})\) by the induction hypothesis.

3. \(b\) sends a SIGNAL(\(c, S\)) to \(p\) and remains engaged. In this case, \(e_{k'}\) occurs in an A3 action, which implies that \(im_{b}[p] > 0\) and \((b, p)\) is not an engagement edge. Condition (a) of Definition 13 holds at time \(t_k\) (i.e., \(x \in DF_b(t_k)\) and thus \(c > 0\) and \(x \in S\)). An argument similar to 2) yields \(x \in DF_r(t_{k'})\).
E5: A process $q$ in $C$ performs Fl (in which it updates $parent_q$ and thus may affect chain $C$). If $q = r$ or $q = a$, then $r \not\equiv b$ or $q \not\equiv a \not\equiv b$ continues to hold at $t_k'$ and thus $x \in DF_r(t_d)$ by the induction hypothesis. So assume $q \not= r$ and $q \not= a$. Let $y$ be the fault that triggered the Fl action.

- If $r \not\equiv b$ at time $t_k$, then using the subchain (of $C$) from $q$ to $r$, we have $r \not\equiv q$ at $t_k'$ by Definition 13(a).

By the induction hypothesis, $y \in DF_r(t_d)$. Since $q$ just detected $y$, $r \not\equiv q$ at $t_k'$. Combining it with $q \not\equiv b$ yields $r \not\equiv a \not\equiv b$ at $t_k'$, which implies $x \in DF_r(t_d)$ by the induction hypothesis.

- If $r \not\equiv a \not\equiv b$ at $t_k$, then $r \not\equiv a \not\equiv q$ at $t_k'$. An argument similar to the preceding one yields $r \not\equiv a \not\equiv b$ at $t_k'$, and, hence $x \in DF_r(t_d)$.

Others: It is not hard to see that in any other event (in particular, EO or E4), $r \not\equiv b$ at $t_k$ and $r \not\equiv a \not\equiv b$ at $t_k$ imply $r \not\equiv b$ at $t_k'$ and $r \not\equiv a \not\equiv b$ at $t_k'$, respectively, and $x \in DF_r(t_d)$ by the induction hypothesis.

We are now in a position to prove $CF(t_d) \subseteq DF_r(t_d)$, from which will follow the second property of the algorithm that by time $t_d$, all engaged faults are detected by coordinator $r$. 

Lemma 9: $CF(t_d) \subseteq DF_r(t_d)$.

Proof: We show by induction that $CF(t_k) \subseteq DF_r(t_d)$ for all $k, c \leq k \leq d$.

Induction Base: For the induction base, we have $CF(t_c) \subseteq DF_r(t_d)$. The induction base holds because the failure that occurred in event $e_c$ has been detected by $r$ by time $t_d$; and every fault that occurred before $t_c$ is in $DF_r(t_d)$, as implied by Lemma 7.

Induction Hypothesis: Now assume as the induction hypothesis that $CF(t_{k-1}) \subseteq DF_r(t_d)$, where $k > c$.

Induction Step: We show $CF(t_k) \subseteq DF_r(t_d)$.

Claim: If $a$ is an engaged process at time $t_k$ at such that 1) $a \not\equiv b$ at $t_k$, and 2) event $e_k$ involves no SIGNAL sending and does not change the path from $a$ to the root of the tree in the hierarchy graph that contains $a$, then $x \in DF_r(t_d)$.

To prove the claim, let $T$ be the tree in the hierarchy graph that contains $a$. By Lemma 3, $T$ is rooted at $r$ or at a faulty node $y$. Let $C$ be the path from $a$ to the root of $T$. By condition 2) of the claim, $C$ does not change in event $e_k$. There are three possibilities:

1) $T$ is rooted at $r$ and all edges in $C$ are engagement edges. (This does not exclude the possibility that $a = r$.) In this case, we have $r \not\equiv b$ at $t_k$, from which it follows, by Lemma 8, that $x \in DF_r(t_d)$.

2) $T$ is rooted at $r$ and there is a nonengagement edge in $C$. The edge $(c, r)$, where $c$ is the child of $r$ on $C$, is a fault-detection edge that was introduced into $H$ due to $c$'s detection of some fault $z$. (This edge is the only nonengagement edge on the path). At time $t_{k-1}$, as a node with a parent in $H$, $c$ is an engaged process and so $z$ is critical. By the induction hypothesis, $z \in DF_r(t_d)$. By Lemma 6(a), $r \not\equiv z$ holds at time $t_{k-1}$; and it continues to hold at time $t_k$ since no SIGNAL is sent in event $e_k$.

Thus $r \not\equiv c \equiv b$ at $t_k$ and, by Lemma 8, $x \in DF_r(t_d)$.

3) The root of $T$, say $y$, is an engaged faulty node or a former coordinator. Since $y$ is either an engaged process or a former coordinator, $y \in CF(t_{k-1})$ by definition, and thus $y \in DF_r(t_d)$ by the induction hypothesis. Let $c$ be the child of $y$ on path $C$. By Lemma 6(b), $r \not\equiv y$ holds at time $t_{k-1}$, and it continues to hold at $t_k$. So $r \not\equiv c \equiv b$ at $t_k$ and, by Lemma 8, $x \in DF_r(t_d)$.

So the Claim is correct, and with it the induction step is relatively easy to establish. Let $x$ be any new critical fault introduced in event $e_k$, if any. We show $x \in DF_r(t_d)$. Such a fault may be introduced only in an E2, E4, or E5 event, where E1, E2, etc. refer to the same events as listed in the paragraph preceding Lemma 3. Consider the three events separately:

E2: An engaged process $x \neq r$ fails in event $e_k$ (so $x$ is a new critical fault). Let $y = parent_x$.

1) If $y$ is nonfaulty at $t_k$ and $(x, y)$ is an engagement edge at $t_{k-1}$, then $y \not\equiv x$ at $t_k$ (by Definition 13(b1), from which it follows $x \in DF_r(t_d)$ (by the Claim).

2) If $y$ is nonfaulty at $t_k$ and $(x, y)$ is a fault-detection edge due to some fault $w$, then $r \not\equiv x$ holds at time $t_k$ (by Definition 6(a)). Since $w \in DF_x(t_{k-1})$, $w$ is a critical fault at $t_{k-1}$. By the induction hypothesis, $w \in DF_r(t_d)$. Since $r \not\equiv x$ at $t_k$ and $x$ is faulty at $t_k$, $r \not\equiv x$ at $t_d$. By Lemma 7, $x \in DF_r(t_d)$.

3) If $y$ is faulty at $t_k$ (and hence at $t_{k-1}$ as well), then $y$ is either an engaged faulty process or a former coordinator. By definition, $y \in CF(t_{k-1})$; and by the induction hypothesis, $y \in DF_r(t_d)$. Evidently, $r \not\equiv y$ at $t_d$ (since $y$ fails before $x$), and so $x \in DF_r(t_d)$ (by Lemma 7).

E4: A neutral process $b$ becomes engaged in event $e_k$. Let $x \in DF_b(t_k)$ be any fault that becomes critical because of $b$'s new engagement due to the receipt of a basic message from, say, process $a$.

1) If $a$ is nonfaulty at $t_k$, then by Definition 13(a), $a \not\equiv b$ at $t_k$, and by the Claim, $x \in DF_r(t_d)$.

2) Now if $a$ is faulty at $t_k$, then it was faulty at $t_{k-1}$ also. By definition, $a \in CF(t_{k-1})$; and by the induction hypothesis, $a \in DF_r(t_d)$. By Lemma 6(c), $a \not\equiv b$ holds at $t_k$. Evidently, $r \not\equiv b \equiv b$ holds at $t_k$, so $x \in DF_r(t_d)$ (by Lemma 8).

E5: An engaged process $b \neq r$ performs Fl after detecting a fault $x$. Let $a$ be the original parent of $b$; i.e., $a = parent_b$ as of time $t_{k-1}$. There are the possibilities: i) $a$ is nonfaulty at $t_k$ and $(b, a)$ was an engagement edge at $t_{k-1}$, ii) $a$ is nonfaulty at $t_k$ and $(b, a)$ was a fault-detection edge due to some fault $w$, and iii) $a$ is faulty at $t_k$. The arguments for these cases are similar to those for cases E4 1), E2 2), and E4 2), respectively.

Property Three: With properties 1 and 2 having been established, the remaining one is relatively easy to prove.

Lemma 10: When $r$ declares termination at time $t_d$, the hierarchy graph $H$ contains no edge.
Proof: Assume for contradiction that \( E(H) \neq \emptyset \) at \( t_d \).
By Lemma 3, there is a nonfaulty engaged process \( i, i \neq r \) with \( parent_i \) pointing to either \( r \), or a faulty engaged process, or a former coordinator. In the first case, since \( r \) is neutral at \( t_d \) the edge \( (i, r) \) must be a fault-detection edge due to detection of some fault \( x \). By definition, \( x \) is a critical fault because of \( r \)'s engaged status. In the other two cases, \( y = parent_i \) is readily seen to be critical. As a critical fault, \( x \) (or \( y \)) belongs to \( DF, (t_d) \) (by Lemma 9). It follows from Lemma 7 that \( x \) (or \( y \)) has been reported to \( r \) by \( i \) through a major SIGNAL, in contradiction to Lemma 6. Therefore, \( E(H) = \emptyset \). \( \square \)

VI. PERFORMANCE ANALYSIS AND COMPARISON

In this section, we analyze the performance of our algorithm and compare it with the V algorithm.

Let \( M \) be the total number of basic messages sent during the entire basic computation (or just those sent in DT mode if the feature of postponed start is implemented), and let \( n \) be the number of processes, \( k \) the maximum number of faults tolerable, and \( f \) the number of actual faults.

- **Faults Tolerable**—Our algorithm can tolerate any number of faults. The V algorithm can tolerate up to a prespecified number, \( k \) of faults.
- **Message Cost**—Our algorithm and the V algorithm are based on the LTD algorithm and the CV algorithm, respectively. In order to get a better view of the performance of the two algorithms, we decompose the overall message complexity of each algorithm into three components: base cost, preparation cost, and cost per actual fault. The base cost is the number of control messages required by each algorithm's nonfault-tolerant predecessor. The preparation cost is defined to be the number of control messages required by the algorithm in addition to the base cost, even if no process fails during the computation: it is the cost of preparing the algorithm for possible faulty processes. The cost per actual fault accounts for the number of control messages required by the algorithm in order to recover from an actual failure. Each process-to-process message is counted as one.

- **Base cost**—As shown in [15], both LTD and CV algorithms are message-optimal in the worst case; but the former requires \( M + n - 1 \) messages only in the worst case, while the latter incurs a flat cost \( M + n - 1 \) in all cases.
- **Preparation cost**—There is no preparation cost at all for our algorithm: if no process fails during the computation, our algorithm essentially becomes the LTD algorithm. The V algorithm, by comparison, has a preparation cost of \( O(kM) \): if both the V algorithm and its predecessor are applied to a system devoid of faulty processes, the V algorithm will use \( O(kM) \) more control messages than its predecessor.
- **Cost per actual fault**—In our algorithm, for each fault that actually occurs during the computation, every nonfault process needs to send at most one extra SIGNAL message (to the system root). Overall, at most \( n \) SIGNALs per fault are needed. (If the postponed start feature is included, then \( n \) START messages should be added for each faulty process, since each new coordinator may send that many STARTs.) By comparison, each time a fault occurs, the V algorithm takes a snapshot and redirects each message to \( k \) representatives. According to [25], about \( O(M/n) \) messages need to be redirected to at most \( k \) representatives, resulting in an overhead of \( O(kM/n) \) per fault. If the user wants the algorithm to be \((n-1)\)-resilient, the quantity \( kM/n \) is on the order of \( M \), which is usually greater than \( n \).

- **Storage**—Our algorithm uses \( O(n) \) space for each process, or \( O(n^2) \) in all. The V algorithm uses \( O(kM) \), as each message transaction is stacked in \( k \) duplicates. Note that \( M \) is a quantity unknown at compile time and thus the V algorithm requires the more expensive dynamic storage allocation to maintain stacks.

- **Detection Delay**—The detection delay is defined to be "the length of the longest possible communication path along which control messages need to be sequentially passed after the basic computation has terminated but before the termination is detected." The detection delay of our algorithm is \( O(n) \). The worst-case detection delay of the V algorithm is the same as that of the CV algorithm, which, as pointed out in [15], could be as large as \( M \).
- **Message Length**—The longest message used in our algorithm is SIGNAL\( (c, S) \). The first parameter \( c \) is an integer not exceeding \( M \) and thus requires at most \( \log M \) bits. The second parameter \( S \) is a set of process identities that can be represented by \( f \) integers, each of \( \log n \) bits. The longest control message used in the V algorithm is remove-entry\( (p, q, t) \) (where \( p \) and \( q \) are process identities and \( t \) is a time stamp), which needs \( 2\log n + \log M \) bits. For an extremely unreliable system in which more than two processes may fail during the computation, our algorithm has to use longer control messages than the V algorithm uses.

- **Atomic sending**—Our algorithm uses no atomic sending. The V algorithm requires that the underlying network be able to support \( (k+1)\)-message atomic sending, which allows a process to atomically send up to \( k+1 \) (possibly different) messages so as to ensure that each message transaction is correctly replicated at \( k \) sites.

- **Network Services**—Aside from atomic sending, the network services required by both algorithms are comparable: reliable communication, fault detection, and fail/return flush.

We have compared our algorithm with the V algorithm and summarized their performance measures in Table I. It is our view that a fault-tolerant algorithm should be able to tolerate any number of faults, should run as efficiently as the best nonfault-tolerant algorithm available if no process fails during the computation, and should incur only a reasonable amount of cost for each process failure that actually occurs. The above analyses indicate that our algorithm does have these nice properties.

VII. CONCLUSION

We have proposed a fault-tolerant algorithm for termination detection that can tolerate any number of crash failures. This
algorithm improves over existing ones in many aspects, including worst-case message complexity, storage overhead, and detection delay. Most important, it has two nice features: 1) no overhead is incurred for being fault-tolerant—it no process fails during the computation, the algorithm is comparable to the best existing nonfault-tolerant algorithm; 2) the cost for processing each fault occurrence is not high and is independent of both \( M \), the number of basic messages, and \( k \), the maximum number of faulty processes the algorithm can tolerate. These nice features, of course, do not come free: our algorithm requires that every process knows the identifications of all processes in the basic system. We don’t know whether it is possible to somehow relax this requirement without trading off the above mentioned nice properties.

Our algorithm is based on the well-known Dijkstra-Scholten termination detector. It achieves fault-tolerance not by message replication, but by appropriately setting a barrier at the root of the system. Similar techniques might be applicable to other termination detectors. In particular, it might be possible to convert the symmetric algorithm in [22] to a fault-tolerant one.

ACKNOWLEDGMENT

The authors would like to thank J. Ward and the anonymous referees for their constructive criticisms and suggestions that have considerably improved this work.

REFERENCES