[15] A. D. Kshemkalyani & M. Singhal, "Invariant-based verification of a distributed deadlock detection algorithm," *IEEE trans. Software Engineering* Vol. 17, No. 8 (Aug. 1991), 789–799.

[16] T. -H. Lai, Y. -C. Tseng & X. Dong, "A more efficient message-optimal algorithm for distributed termination detection," *6th Intl. Parallel Processing Symposium* (March 1992), 646–649.

[17] T.-H. Lai, "Termination detection for dynamically distributed systems with non-first-in-first-out communication," *Journal of Parallel and Distributed Computing* 3 (1986), 577–599.

[18] E. L. Lozinskii, "A remark on distributed termination," *Proc. Fifth Intl. Conf. on Distributed Computing Systems* (1985), 416–419.

[19] F. Mattern, "Algorithms for distributed termination detection," *Distributed Computing* 2 (1987), 161–175.

[20] J. Misra, "Detecting termination of distributed computations using markers," *ACM Symp. on Principles of Distributed Computing* (Aug. 1983), 290–294.

[21] J. Misra & K. M. Chandy, "Termination detection of diffusing computations in communicating sequential processes," *ACM trans. on Programming Languages and Systems* 4,1 (Jan. 1982), 37–43.

[22] R. D. Schlichting & F. B. Schneider, "Fail-stop processors: an approach to designing fault-tolerating computing systems," *ACM trans. on Computer Systems* 1, 3 (Aug. 1983), 222–238.

[23] A. S. Tanenbaum, *Computer Networks, Ed. 2, Prentice-Hall, Inc.* (1988).

[24] G. Tel & F. Mattern, "Comments on "ring based termination detection algorithm for distributed computations"," *Information Processing Letters* 31 (May 1989), 127–128.

[25] R. W. Topor, "Termination Detection for distributed computations," *Information Processing Letter* 18, 20 (Jan. 1984), 33–36.

[26] S. Venkatesan, "Reliable protocols for distributed termination detection," *IEEE trans. on Reliability* 38, 1 (Apr. 1989), 103–110.

BIBLIOGRAPHY

[1] M. Ahuja, "Implementation of F-channels, a preferable alternative to FIFO channels," *IEEE Proceedings of the Eleventh Intl. Conf. on Distributed Computing Systems* (1991), 180–187.

[2] H. Attiya, A. Bar-Noy, D. Dolev, D. Koller, D. Peleg & R. Reischuk, "Achievable cases in an asynchronous environment," *IEEE Foundation of Computer Science* (1987), 337–346.

[3] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg & R. Reischuk, "Renaming in an asynchronous environment," *Journal of ACM* Vol. 37, No. 3 (July 1990), 524–548.

[4] F. M. Bridgleand & R. J. Watro, "Fault-tolerant decision making in totally asynchronous distributed systems," *ACM Symp. on Principles of Distributed Computing* (1987), 52–63.

[5] S. Chandrasekaran & S. Venkatesan, "A message-optimal algorithm for distributed termination detection," *Journal of Parallel and Distributed Computing* 8 (1990), 245–252.

[6] K. M. Chandy & L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM tran. on Computer Systems* 3, 1 (Feb. 1985), 63–75.

[7] V. Cherkassky, H. Lari-Najafi & R. Rooholamini, "Fault-tolerant communications processing," *IEEE Intl. Symp. on Fault-Tolerant Computing* 21 (1991), 344–351.

[8] A. L. Choudhary, W. H. Kohler, J. A. Stankovic & D. Towsley, "A modified priority based probe algorithm for distributed deadlock detection and resolution," *IEEE trans. Software Engineering* Vol. 15, No. 1 (Jan. 1989), 10–17.

[9] S. Cohen & D. Lehmann, "Dynamic systems and their distributed termination," *Proc. First Annual ACM Symp. on Principles of Distributed Computing* (Aug. 1982), 29–33.

[10] E. W. Dijkstra, W. H. Feijen & A. J. van Gasteren, "Derivation of a termination detection algorithm for distributed computations," *Information Processing Letter* 16, 5 (June 1983), 217–219.

[11] E. W. Dijkstra & C. S. Scholten, "Termination detection for diffusing computations," *Information Processing Letter* 11 (Aug. 1980), 1–4.

[12] M. J. Fischer, N. A. Lynch & M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of ACM* Vol. 32, No. 2 (Apr. 1985), 374–382.

[13] N. Francez & M. Rodeh, "Achieving distributed termination without freezing," *IEEE trans. on Software Engineering* SE-8,3 (May 1982), 287–292.

[14] S.-T. Huang, "Termination detection by using distributed snapshots," *Information Processing Letter* 32 (Aug. 1989), 113–119.

mance measures in Table 1. 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 non-fault-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.

We assumed in our model that the underlying communication network provides a reliable, fully connected communication medium. This assumption is not unrealistic, for termination detectors are presumably to be run at a layer directly or indirectly above the transport layer and the latter by definition provides reliable end-to-end transport service to its users. Nevertheless, the assumption of fully-connectedness of the underlying graph is not crucial for our algorithm. (The underlying graph is one that represents the underlying system, with nodes representing processes and edges representing communication patterns.) If the underlying graph is not fully connected and if a termination detector is allowed to communicate only between adjacent nodes (which, in our opinion, is a requirement of no practical necessity), our algorithm with slight modification will still work, as long as the underlying graph is not disconnected. In such an environment, the gauges shown in the above table are still correct except that the cost per fault becomes $O(kM)$ for the V algorithm and $O(n^2)$ for our algorithm and that the detection delay becomes $O(n^2)$ for our algorithm and $O(nM)$ for the V algorithm.

7 Conclusion

We have proposed a fault-tolerant algorithm for termination detection that can tolerate any number of stop failures. This algorithm improves over existing ones in many aspects, including worst-case message complexity, average-case message complexity, storage overhead. Most important, it has two nice features: 1) No overhead is incurred for being fault-tolerant—if no process fails during the computation, the algorithm is comparable to the best existing non-fault-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.

The main idea of our algorithm is a barrier set at a particular process. It is possible that other existing termination detection algorithms can be made fault-tolerant using similar techniques.

$\diamond$ *Preparation cost* — There is no preparation cost at all for our algorithm; if no process fails during the computation, our algorithm becomes the LTD algorithm. The V algorithm, by comparison, has a preparation cost of $O(kM)$, requiring that much more control messages than its predecessor should both be applied to a system without faulty processes.

$\diamond$ *Cost per actual fault* — In our algorithm, for each fault that actually occurs during the computation, every nonfaulty process needs to send one CLEAR message and at most one SIGNAL message. Overall, at most $2n$ messages per fault are needed. (If the postponed start feature is considered, extra n START messages for each failure should be added as each new coordinator may send up to n START messages.) By comparison, each time a fault occurs, the V algorithm takes a snapshot and redirects each message to k representatives. On the average, about 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 in the order of M that 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 underlying 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 [16], could be as large as M.

• *Atomic sending* — Our algorithm does not need any network service other than those described in Section 2.2. In addition to these services, however, the V algorithm requires the underlying network to 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 be correctly replicated at k sites.

• *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 does.

We have compared our algorithm with the V algorithm and summarized their perfor-

	# Faults Tolerable	Overall Message Cost		
		Base Cost	Preparation Cost	Cost Per Actual Fault
V Algorithm	$k\ (< n)$	$M + n - 1$	$\approx 3kM + k(n-1)$	$O(kM/n)$
Ours	$n - 1$	$\leq M + n - 1$	0	$O(n)$

	Storage	Detection Delay	Atomic Sending	Message Length
V Algorithm	$O(kM)$	O(M)	*Required*	$2\log n + \log M$
Ours	$O(n^2)$	O(n)	*Not Required*	$f\log n + \log M$

Table 1: Performance Measures.

Theorem 2 *If a process declares termination at time $t_{\rm d}$, then the underlying computation is really terminated at the time.*

6 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 underlying 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 be the number of actual faults.

• *#Faults Tolerable*, the number of faults tolerable — Our algorithm can tolerate any number of faults. The V algorithm can tolerate up to a *pre-specified* number, k, of faults.

• *Message Cost* — Our algorithm and the V algorithm each have a non-fault-tolerant predecessor; the former is based on the LTD algorithm, while the latter is on the CV algorithm. In order to get a better view of the two algorithms' performances, 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 non-fault-tolerant predecessor. The preparation cost is defined to be the number of control messages that is required 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 [16], both LTD and CV algorithms are message-optimal in the worst case, but LTD is more efficient than CV on the average; the former requires far less than $M + n - 1$ messages in an average case, while the latter incurs a flat cost $M + n - 1$ in all cases.

Subcase (i): Suppose, at time $t-1$, T is rooted at r and C is an engagement chain. Then at time t, after the crash of x, $r \stackrel{x}{\leftarrow} parent_x$, from which it follows $x \in DF_r(t_{\rm d})$ (by Lemma 6).

Subcase (ii): Suppose at time $t-1$, T is rooted at r and C is not an engagement chain. The edge $\langle c, r\rangle$, where c is the child of r on the chain C, must be a fault-detection edge that was introduced into G due to c's detection of some fault z. (This edge is the only non-engagement edge on the path). At time $t-1$, as a node with a parent in G, c is engaged and so z is critical. By the induction hypothesis, $z \in DF_r(t_{\rm d})$. According to Corollary 2, c has not reported z to r (through a major SIGNAL) as of time t. If $c = x$, it follows from Corollary 1 that $x \in DF_r(t_{\rm d})$. Otherwise, c is nonfaulty and $r \stackrel{z}{\not\leftarrow} c \stackrel{x}{\leftarrow} parent_x$ at t; by Lemma 6, $x \in DF_r(t_{\rm d})$.

Subcase (iii): T is rooted at an engaged faulty node y. This case is not much different from (ii). Let c be the child of y on the chain between x and y. Since y is engaged, $y \in CF(t-1)$, and by the induction hypothesis, $y \in DF_r(t_{\rm d})$. By Corollary 2, c has not reported fault y to r by time t and hence $r \stackrel{y}{\not\leftarrow} c$ holds at time t. Following a similar argument as in subcase (ii), one readily sees $x \in DF_r(t_{\rm d})$ whether $x = c$ or $x \neq c$.

E4. A neutral process b becomes engaged at time t. Let x be any fault that becomes critical because of b. Thus, $x \in F_b(t)$. Let a be the process that makes b engaged. At time t, process a is in a tree T rooted either at r or at an engaged faulty process y. Following arguments similar to those for case E2, x can be shown to be in $DF_r(t_{\rm d})$.

E5. An engaged process a detects a fault x at time t. Similar arguments as in case E2 can be applied. □

With properties 2 and 3 established, the remaining one is relatively easy to prove.

Lemma 8 *When r declares termination at time $t_{\rm d}$, $E = \emptyset$.*

Proof. Assume for contradiction that $E \neq \emptyset$ at $t_{\rm d}$. By Lemma 2, there is a nonfaulty engaged process i, $i \neq r$ with $parent_i$ equal to r, a faulty engaged process, or a former coordinator. In the first case, since r is neutral at $t_{\rm d}$, the edge $\langle i, r\rangle$ must be a fault-detection edge due to detection of some fault x. By definition, x is a critical fault because of i'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 $DF_r(t_{\rm d})$ (by Lemma 7). It follows from Lemma 4 that x (or y) has been reported to r by i through a major SIGNAL, in contradiction to Corollary 2. Therefore, $E = \emptyset$. □

We have proved the three properties mentioned in the beginning of this subsection. As remarked there, the soundness of the algorithm follows directly from these properties. We have thus proved the following.

E4: A neutral process becomes engaged. In this case, $r \stackrel{x}{\leftarrow} b$ at t and $r \stackrel{z}{\not\leftarrow} a \stackrel{x}{\leftarrow} b$ at t imply $r \stackrel{x}{\leftarrow} b$ at t' and $r \stackrel{z}{\not\leftarrow} a \stackrel{x}{\leftarrow} b$ at t', respectively, and so $x \in DF_r(t_\mathrm{d})$ by the induction hypothesis.

E5. A process c in C detects a fault y in event e.

- If $r \stackrel{x}{\leftarrow} b$ at t and $c = r$, then $r \stackrel{x}{\leftarrow} b$ at $t' = t + 1$. By the induction hypothesis, $x \in DF_r(t_\mathrm{d})$.
- If $r \stackrel{z}{\not\leftarrow} a \stackrel{x}{\leftarrow} b$ at t and $c = a$, then $r \stackrel{z}{\not\leftarrow} a \stackrel{x}{\leftarrow} b$ at t' and, again by the induction hypothesis, $x \in DF_r(t_\mathrm{d})$.
- If $r \stackrel{x}{\leftarrow} b$ at time t and $c \neq r$, then by Lemma 5, we have $r \stackrel{y}{\leftarrow} parent_c$ or $r \stackrel{c}{\leftarrow} parent_c$, and $c \stackrel{b}{\leftarrow} x$, all at t'. By the induction hypothesis, $y \in DF_r(t_\mathrm{d})$ or $c \in DF_r(t_\mathrm{d})$. In the latter case, Corollary 1 indicates that $y \in DF_r(t_\mathrm{d})$. So in both cases $y \in DF_r(t_\mathrm{d})$. Since c just detects y, $r \stackrel{y}{\not\leftarrow} c$ at t'. Combining it with $c \stackrel{b}{\leftarrow} x$ yields $r \stackrel{y}{\not\leftarrow} c \stackrel{x}{\leftarrow} b$ at t'. It thus follows from the induction hypothesis $x \in DF_r(t_\mathrm{d})$.
- If $r \stackrel{z}{\not\leftarrow} a \stackrel{x}{\leftarrow} b$ at t and $c \neq r$, $c \neq a$, then $r \stackrel{z}{\not\leftarrow} a \stackrel{y}{\leftarrow} parent_c$ or $r \stackrel{z}{\not\leftarrow} a \stackrel{c}{\leftarrow} parent_c$ and $c \stackrel{x}{\leftarrow} b$, all at t'. An argument similar to the preceding one concludes that $r \stackrel{y}{\not\leftarrow} c \stackrel{x}{\leftarrow} b$ at t' and, hence, $x \in DF_r(t_\mathrm{d})$.

Others. It is not hard to see that in any other event, $r \stackrel{x}{\leftarrow} b$ at t and $r \stackrel{z}{\not\leftarrow} a \stackrel{x}{\leftarrow} b$ at t imply $r \stackrel{x}{\leftarrow} b$ at t' and $r \stackrel{z}{\not\leftarrow} a \stackrel{x}{\leftarrow} b$ at t', respectively, and so $x \in DF_r(t_\mathrm{d})$ by the induction hypothesis. □

We are now ready to prove the second property.

Lemma 7 $CF(t_\mathrm{d}) \subseteq DF_r(t_\mathrm{d})$.

Proof. We prove by induction on t that $CF(t) \subseteq DF_r(t_\mathrm{d})$ for all $t \in [t_c, t_\mathrm{d}]$. For the induction base, we have $CF(t_c) \subseteq DF_r(t_\mathrm{d})$. It is so because, by Corollary 1, every fault that occurs at or before t_c is in $DF_r(t_\mathrm{d})$. Now assume as the induction hypothesis that $CF(t-1) \subseteq DF_r(t_\mathrm{d})$, where $t > t_c$. We prove $CF(t) \subseteq DF_r(t_\mathrm{d})$, showing that any new critical fault introduced at time t (i.e., in the event that occurs immediately before t), if any, will be detected by r before time t_d. Such a fault may be introduced in any of the following events. (**E1**, **E2**, ...refer to the same events as in the proofs of Lemma 6, and 2.)

E2. An engaged process $x \neq r$ fails at t (and so x is a new critical fault). At time $t-1$, x belongs to a non-trivial tree T (according to Lemma 2). The tree T may be rooted at r or at a faulty node y, x's parent may or may not be r (or y), and the path C from x to r (or y) may or may not be an engagement chain.

The importance of the notions $r \overset{x}{\leftarrow} b$ and $r \overset{z}{\not\leftarrow} a \overset{x}{\leftarrow} b$ are reflected in the following lemma.

Lemma 6 *Let x be faulty at time $t \in [t_c, t_d]$. If at t there are nonfaulty processes a, b and faulty process z such that $r \overset{x}{\leftarrow} b$ at t or $r \overset{z}{\not\leftarrow} a \overset{x}{\leftarrow} b$ at t, then $x \in DF_r(t_d)$.*

Proof. The proof is by induction on t, starting with $t = t_d$.

Induction Base: Suppose at time $t = t_d$ there are nonfaulty processes a, b and faulty process z such that $r \overset{x}{\leftarrow} b$ or $r \overset{z}{\not\leftarrow} a \overset{x}{\leftarrow} b$. The latter case is impossible since by Lemma 4 every nonfaulty process (a in particular) has reported to r by time t_d. In the case of $r \overset{x}{\leftarrow} b$, it must be $r \overset{x}{\leftarrow} r$ as r is neutral at t_d and hence has no unacknowledged basic messages. Since r is neutral, it follows from Definition 9 that $x \in DF_r(t_d)$.

Induction Hypothesis: Assume the lemma true for $t \in [t', t_d]$, where $t_c < t'$.

Induction Step: Let $t = t' - 1$. Let a, b, z be processes such that "$r \overset{x}{\leftarrow} b$ at time t" or "$r \overset{z}{\not\leftarrow} a \overset{x}{\leftarrow} b$ at time t". We want to show $x \in DF_r(t_d)$. Let C be the engagement chain involved in $r \overset{x}{\leftarrow} b$ or in $a \overset{x}{\leftarrow} b$. Let e be the event that occurs between time t and $t + 1$. Consider all possible cases.

E1: The present coordinator r crashes at t. This case is impossible by definition of t_c.

E2: A process $y \neq r$ in C fails in event e. By Lemma 5, we have

- $r \overset{y}{\leftarrow} parent_y$ at time $t' = t + 1$, if $r \overset{x}{\leftarrow} b$ at time t;
- $r \overset{z}{\not\leftarrow} a \overset{y}{\leftarrow} parent_y$ at t', if $r \overset{z}{\not\leftarrow} a \overset{x}{\leftarrow} b$ at t and $y \neq a$;
- $r \overset{z}{\not\leftarrow} a$ at t', if $r \overset{z}{\not\leftarrow} a \overset{x}{\leftarrow} b$ at t and $y = a$;

In the first two cases, the induction hypothesis indicates that $y \in F_r(t_d)$. In the third case, y $(= a)$ is also in $DF_r(t_d)$ as implies by Corollary 1. So, in all cases, $y \in DF_r(t_d)$, and as y's failure occurs after x's, it follows from Corollary 1 that $x \in DF_r(t_d)$.

E3: An engaged process in C becomes neutral in e. This process must be b because only a leaf node in G can become neutral. For $t' = t + 1$, the followings hold:

- $r \overset{x}{\leftarrow} parent_b$ at t', if $r \overset{x}{\leftarrow} b$ at t and $b \neq r$;
- $r \overset{x}{\leftarrow} r$ at t', if $r \overset{x}{\leftarrow} b$ at t and $b = r$;
- $r \overset{z}{\not\leftarrow} a \overset{x}{\leftarrow} parent_b$ at t', if $r \overset{z}{\not\leftarrow} a \overset{x}{\leftarrow} b$ at t and $a \neq b$;
- $r \overset{z}{\not\leftarrow} a \overset{x}{\leftarrow} a$ at t', if $r \overset{z}{\not\leftarrow} a \overset{x}{\leftarrow} b$ at t and $a = b$.

In all cases, $x \in DF_r(t_d)$ by the induction hypothesis.

Proof. In case a), the edge $< i, j >$ that was created due to the detection of x would have been deleted if i has reported x to r through a major SIGNAL. In case b), i has not yet detected x as of time t. □

We now proceed to prove the second property mentioned above. We need the following definitions and lemmas. These definitions are made with respect to the computation $(e_1, e_2, \ldots, e_d)$ in which process r declares termination in event e_d.

Definition 8 A directed path of length zero or more in G, going from b to a, is said to be an *engagement chain* and denoted as $a \leftarrow b$ iff the ending point a is nonfaulty and all edges on the path are engagement edges.

Definition 9 For processes a, b, x and time $t \leq t_{\rm d}$, we write "$a \stackrel{x}{\leftarrow} b$ *at* t" iff the following two statements are both true:

1. there exists at time $t < t_{\rm d}$ an engagement chain $a \leftarrow b$ in $G(t)$.
2. $\forall t' \in [t, t_{\rm d}]$, b is engaged at t' or $x \in F_b(t')$ (i.e., if b ever turns neutral after t, then it must have detected x's failure.)

Note that x may or may not be faulty at time t and the path $a \leftarrow b$ may or may not continue to exist *after* t.

Lemma 5 *Assume* $a \neq b$.
a) *If* $a \leftarrow b$ *at time* $t-1$, *and* b *fails in event* e_t, *then* $a \stackrel{b}{\leftarrow} parent_b$ *at* t, *right after* b*'s failure.*

b) *If* $a \leftarrow b$ *at time* $t-1$, *and* b *detects a failure* y *in event* e_t, *then* $a \stackrel{y}{\leftarrow} parent_b$ *or* $a \stackrel{b}{\leftarrow} parent_b$ *at* t, *right after the detection.*

c) *If* $a \stackrel{x}{\leftarrow} b$ *at time* $t-1$, *and* b *becomes neutral in event* e_t, *then* $a \stackrel{x}{\leftarrow} parent_b$ *at* t.

Proof. We prove only statement b); the other two are quite obvious. Suppose $a \leftarrow b$ at $t-1$ and b detects a failure y in event e_t. Let $p = parent_b$. By Lemma 1, $out_p[b] \geq in_b[p] > 0$ at t. Assume that p becomes neutral sometime between t and $t_{\rm d}$ or we are done. When p becomes neutral, $out_p[b] = 0$. But $out_p[b]$ may become 0 only in two occasions: p detects the failure of b, or p receives a SIGNAL which was sent by b after t and hence contains y in its S set. In the former case we have $a \stackrel{b}{\leftarrow} parent_b$ at t, while in the latter $a \stackrel{y}{\leftarrow} parent_b$ at t. □

Definition 10 For $x \in DF_r(t_{\rm d})$ and $a \neq x$, we write "$r \stackrel{x}{\not\Leftarrow} a$ *at time* t" iff by time t process a has not reported x to r as a fault through a major SIGNAL.

Definition 11 Write "$r \stackrel{z}{\not\Leftarrow} a \stackrel{x}{\leftarrow} b$ at t" iff $r \stackrel{z}{\not\Leftarrow} a$ and also $a \stackrel{x}{\leftarrow} b$ at t.

From these properties, the soundness of the algorithm can be readily proved. In fact, property 1, together with Lemma 2, implies that, at time $t_{\rm d}$, all nonfaulty processes are neutral and hence idle. (Note that r is also neutral because it executes action **A4** to declare termination.) As neutral processes have no unacknowledged basic messages, only a channel connecting an engaged faulty process x to a nonfaulty one i may possibly contain deliverable basic messages. By definition, x is critical and hence, by properties 2 and 3, in $MS_i(t_{\rm d})$. That means i has reported x to r through a major SIGNAL message, which, according to the algorithm, could have happened only after i had received the returned CLEAR message from x (see **F2**). Thus, channel $[x, i]$ is free from basic messages. By definition, the underlying computation is terminated at $t_{\rm d}$.

So, all what we need to do is prove the three properties. We start with the third one.

Lemma 4 *$DF_r(t_{\rm d}) \subseteq MS_i(t_{\rm d})$ for all processes $i \in P - DF_r - \{r\}$.*

Proof. Assume $DF_r(t_{\rm d}) \neq \emptyset$, or the lemma is proved. Let i be a process i in $P - DF_r - \{r\}$. When r declares termination at time $t_{\rm d}$, $DF_r \supseteq KF_r$ and $|KF_r| \geq NumFault_r[i] = |DF_r|$ for all i. As $|KF_r| \leq NumFault_r[i]$, and $NumFault_r[i] = |DF_r| \geq |KF_r|$, $DF_r = KF_r$. This implies that for each i, process r has received a SIGNAL$(-, S_i)$ from i such that $DF_r = S_i$. The question is whether such a SIGNAL is a major one.

Consider any $i \in P - DF_r(t_{\rm d}) - \{r\}$. Let x be the process in $DF_r(t_{\rm d})$ that was the last one, among all processes in $DF_r(t_{\rm d})$, to be detected by i as faulty. On detecting x, $parent_i$ was set to r, for r was not faulty and $DF_r \supseteq \{1, \ldots, r-1\}$. Let i detect x at time t. We make three observations. First, the first SIGNAL to r sent by i after t must be a major signal. Second, any SIGNAL$(-, S)$ sent by i after time t must have $S \supseteq DF_r(t_{\rm d})$. Third, i did send some SIGNAL message(s) to r after time t, for at least the above mentioned SIGNAL$(-, S_i)$ was sent after t. The lemma follows directly from these observations. □

Corollary 1 *If x is faulty at $t_{\rm d}$ and $z \notin MS_x$ for some $z \in DF_r(t_{\rm d})$, then $x \in DF_r(t_{\rm d})$.*

The above lemma gives a case in which we are able to tell what processes must have sent a major SIGNAL to r containing what faults. The following corollary does just the opposite; it shows two cases in which a nonfaulty process must have not reported a fault to r by a major SIGNAL.

Corollary 2 *Suppose that at time t, $t_c \leq t \leq t_{\rm d}$, process x is faulty. As of time t, a process i must have not sent to r any major SIGNAL containing x if either of the following is satisfied:*

a) *$\langle i, j \rangle$ is a fault-detection edge in $G(t)$ due to a fault x.*

b) *$\langle i, x \rangle$ is an edge in $G(t)$.*

eventually becomes strongly neutral (Lemma 3), F is sent to r via a SIGNAL message. When the message eventually arrives at r, $NumFault_r[b]$ is set to $|F|$. Thus, eventually $NumFault_r[b] = |F| = |DF_r|$ for all $b \in P - DF_r - \{r\}$.

So, process r eventually satisfies all of the above six conditions and declares termination. □

5.4 Soundness

This section proves the soundness of the proposed algorithm: If some process declares termination at time $t_{\rm d}$ then the underlying computation is really terminated at that time.

Assume that the coordinator r declares termination at time $t_{\rm d}$ (it is clear that only the coordinator can do that). Let $(e_1, e_2, \ldots, e_d)$ be the sequence of events that occur before r's declaration, with r declaring termination at the end of e_d. For ease of presentation, assume that event e_i occurs at the moment just before time $t = i$. Let t_c be the time at which process r becomes the coordinator, i.e.,

$$t_c = \begin{cases} 0 & \text{if } r = 1, \\ \text{the time when the coordinator right before } r \text{ fails} & \text{if } r \neq 1. \end{cases}$$

Definition 6 At any time $t \in [t_c, t_{\rm d}]$, a faulty process x is said to be *critical* if at least one of the following is true:

- x is a former coordinator (i.e $x < r(t)$, $r(t)$ being the coordinator of time t);
- x is engaged (at time t);
- x was once known by some engaged process (i.e., there was a time $t' \leq t$ such that $x \in DF_i(t')$ for some engaged (as of time t') process i).

Let $CF(t)$ denote the set of all critical faulty processes as of time t. (Note that once a faulty process becomes critical, it remains critical thereafter.)

Definition 7 A SIGNAL is said to be a *major* one if it is sent by process i to its parent in G (while executing **A4**). Denote by $MS_i(t)$ the set S carried in the most recent major SIGNAL that i has sent to r by time t (whether it has or has not been received by r).

Our plan of proof is as follows. First, we establish three important properties of the algorithm:

1. $E(t_{\rm d}) = \emptyset$.
2. $CF(t_{\rm d}) \subseteq DF_r(t_{\rm d})$.
3. $DF_r(t_{\rm d}) \subseteq MS_i(t_{\rm d})$ for all processes $i \in P - DF_r(t_{\rm d}) - \{r\}$.

Proof. If all the n processes eventually become faulty, then the lemma is proved. Now assume that not all processes eventually fail. Let $t_{\rm f}$ be the time at which the very last failure occurs. Let F be the final set of faulty processes. By the assumption of fault detectability, there is a time $t_{\rm d} \geq t_{\rm f}$ when all faults in F are known to all nonfaulty processes (i.e., $DF_i = F$ for all $i \notin F$). Let $t = \max\{t_{\rm d}, t_{\rm term}\}$.

Consider the hierarchy graph of time t. Since the system is terminated and no fault or fault detection occurs after time t, it is clear from the proof of Lemma 2 which shows how G may change its structure, that after time t the only event that may occur to G is E3, in which a leaf node becomes neutral and leaves a tree. We show that every leaf of the hierarchy graph eventually does become neutral. Let i be any engaged process that has no child in G. We claim that out_i eventually becomes $\vec{0}$. To see this, observe that for any nonfaulty node j, $in_j[i] = 0$ as of time t; by Lemma 1, $out_i[j]$ becomes zero when all SIGNAL messages, if any, in channel $[j, i]$, eventually arrive at i. For every faulty process x, $out_i[x]$ is set to 0 when i receives back the CLEAR message sent to x. Thus, out_i eventually becomes $\vec{0}$. Since p_i is idle and $\sum_{k \neq parent_i} in_i[k] = 0$ after an **A3**, p_i eventually executes **A4** and becomes strongly neutral, thereby isolated in G. Repeating this argument, one readily sees that every nonfaulty process that is engaged at time t eventually becomes strongly neutral, and thus eventually $E = \emptyset$.

As for those nonfaulty processes which are weakly neutral at time t, it is not hard to see that they will become strongly neutral once their CLEAR messages all return. □

Theorem 1 *After the underlying computation terminates, the algorithm declares termination within a finite amount of time (unless all processes fail).*

Proof. Assume that the underlying computation terminates at time $t_{\rm term}$, and that not all processes eventually fail (or we are done). Let F be the final set of faulty processes. By Lemma 3 and the assumption of fault detectability, there is a time instant $t > t_{\rm term}$ at which $E = \emptyset$ and $DF_i = F$ for all $i \notin F$. Assume that no process has declared termination by time t, or the theorem is proved.

According to the algorithm, a process i declares termination if all these conditions are satisfied: 1) $root_i = true$, 2) p_i is idle, 3) $in_i = \vec{0}$, 4) $out_i = \vec{0}$, 5) $DF_i \supseteq KF_i$, and 6) $NumFault_i[b] = |DF_i|$ for all $b \in P - DF_i - \{i\}$. We show that these conditions eventually hold at process $r = \min(P - F)$, the coordinator of time t.

The first five conditions can be easily seen to hold: As r has detected all faults by time t, $root_r = true$ at t. Since the system has terminated, r is idle; and this eventually triggers **A3** and sets $in_i = \vec{0}$. By Lemma 3, r eventually becomes strongly neutral and $out_r = \vec{0}$. Obviously, $DF_r = F \supseteq KF_r$ at time t.

As for the last condition, it obviously holds if $F = \emptyset$. So assume $F \neq \emptyset$. After a nonfaulty process b detects the last fault in F, $parent_b$ is set to r. After that, when b

It is necessary to consider only those events that may possibly change the structure of G. There are five such events.

E1: The present coordinator r fails. In this case, a process r' succeeds as the new coordinator, and the edge $\langle r', parent_{r'} \rangle$, if exists, disappears from G. 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: A nonfaulty engaged process $x \neq r$ fails. In this case, the edge $\langle x, parent_x \rangle$ disappears from G and the subtree rooted at x becomes a tree rooted at an engaged faulty node, which is in category 2. The lemma remains valid.

E3: An engaged nonfaulty process $i \neq r$ finishes action **A4** and becomes neutral (the event e in question is the last event of **A4**). The edge $\langle i, parent_i \rangle$ is deleted from G. The precondition $out_i = \vec{0}$ together with the fact that p_i is not the coordinator ensures that p_i was a leaf in G before event e. 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 possibilities as i can be strongly or weakly.

1. $parent_i$ = NULL before **A2** — In this case, p_j must be engaged and therefore belongs to a tree in categories 1-3. Setting $parent_i$ to j attaches the tree rooted at i to the tree containing j without violating the lemma.

2. $parent_i \neq$ NULL before the action — In this case, $parent_i$ points to either the present or a former coordinator. The new edge $\langle i, parent_i \rangle$ (which did not exist because i was neutral) thus connect the tree containing i to either the main tree or to a tree in category 3. The lemma holds.

E5: An engaged process $i \neq r$ changes its value of $parent_i$ in **F1**. This results in process i changing its parent in G to either the present or a former coordinator. Again, the lemma remains true. □

5.3 Completeness

Assume that the underlying computation terminates at time t_{term}. We show that some process eventually detects and declares termination unless all the n processes fail before anyone has a chance to do that.

Lemma 3 *After the underlying computation terminates, each nonfaulty process eventually becomes either faulty or strongly neutral.*

Lemma 1 *If p_i and p_j are both nonfaulty, then*

$$out_i[j] = b + in_j[i] + \sum_{k=1}^{q} c_k.$$

5.2 Structure of Engaged Processes

Definition 4 At any time t, the *coordinator* of the system, 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., $r(t) = \min\{i : i \text{ is nonfaulty at time } t\}$.

In the algorithm, when process r recognizes itself as the coordinator, $root_r$ is set to *true* in **F1**.

In the DS algorithm all engaged processes form a tree. In our algorithm, as to be seen soon, the variables $parent_i$ of all nonfaulty engaged processes form a forest, which we will call it a hierarchy graph.

Definition 5 The directed graph $G = (V, E)$, where $V = P$ and $E = \{\langle i, parent_i\rangle$: $i \neq r$, p_i is engaged and nonfaulty$\}$ is called the *hierarchy graph*. (Note : if p_i, $i \neq r$, is engaged then $parent_i \neq$ NULL.)

The hierarchy graph indicates whom an engaged nonfaulty process will report to when it becomes neutral. We distinguish between two types of edges in G: engagement edges and fault-detection edges. An edge $\langle i, parent_i\rangle$ in G is said to be a *fault-detection edge due to p_x* if $parent_i$ is set in **F1** because of p_i detecting the failure of p_x. An *engagement edge* is one that is created in **A2** on receiving a basic message. It should be clear from the algorithm that an edge is either an engagement or a fault-detection edge.

The hierarchy graph G plays an important role in our proof. We first show that at any time it forms a forest.

Lemma 2 *At any time t before the algorithm declares termination, G forms a forest comprising the following trees:*

1) A tree rooted at the coordinator — 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. Initially, before any event occurs, G forms a single tree with $r = 1$ at the root and the lemma is true. Assume the lemma is valid immediately before an event e. We show it remains valid after the event.

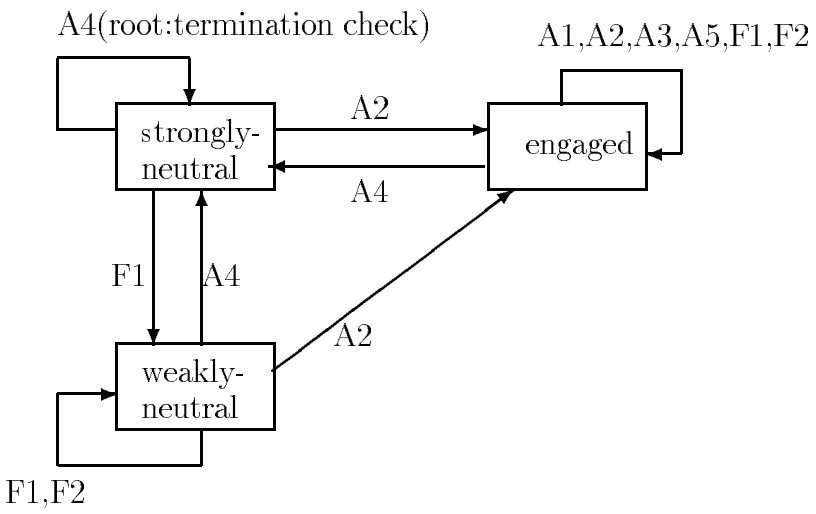

Figure 4: State transitions of a process

Should a process fails, its engaged/neutral status stays unchanged.

It is not hard to see from the above definition and **A4** that if process i is neutral then it is idle and has no unacknowledged incoming or outgoing basic message, and, besides, $in_i = \vec{0}$, $parent_i$ = NULL. However, out_i may or may not be $\vec{0}$.

Definition 3 A neutral process i is said to be *strongly* so if $out_i = \vec{0}$, and *weakly* so otherwise.

Fig. 4 shows how a process changes its neutral/engaged status. Note that a strongly neutral process i sets $out_i[x]$ to 1 and becomes weakly so upon detecting a fault x. For a weakly neutral process i, there is an outstanding CLEAR message from i to x iff $out_i[x] > 0$. After all these CLEAR messages return to i, it becomes strongly neutral again.

In the following we state a simple invariant involving the number of unacknowledged incoming and outgoing messages. Consider any two nonfaulty processes p_i and p_j. Recall that $out_i[j]$ indicates the number of basic messages sent from p_i to p_j which have not yet been acknowledged, and $in_j[i]$ is the number of basic messages which p_j has received but not yet acknowledged. Also recall that the c parameter of a SIGNAL 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 SIGNAL$(c_1, -)$, SIGNAL$(c_2, -)$, ..., SIGNAL$(c_q, -)$ be all the SIGNAL messages in the channel from p_j to p_i (which have been sent by p_j but not yet received by p_i). We immediately have the following.

receiving START messages. In the first case, the coordinator sends a START message to every other process. A coordinator does not necessarily enter DT mode in this way— it may have received a START message and entered DT mode before becoming coordinator. To avoid possible deadlock, it is necessary to ensure that each coordinator, while alive and whether in DT or non-DT mode, eventually sends one START message to every other nonfaulty process, unless it becomes faulty before being able to do so. The reason for a coordinator already in DT mode to send START messages is due to the command "send a START message to every nonfaulty process" is not an atomic action and thus the former coordinator which switched the present coordinator to DT mode could have crashed in the course of sending START messages.

In the next section we establish the correctness of our algorithm as presented in the previous subsection. It is 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.

5 Correctness Proof

We now prove the algorithm described in the preceding section correct. Specifically, we prove that: 1) if some process declares termination, then the underlying computation is really terminated at the time, and 2) after the underlying computation terminates, some process will declare termination in a finite amount of time.

The underlying computation together with the execution of the termination detection algorithm is modeled as a single sequence of *events*. There are five different types of events: a process sends a single message, receives a message, stop-fails, detects the failure of another process, performs a local atomic computation. Each event is regarded as occurring at one single point of time without mutual overlapping. According to this model, an action of our algorithm, in general, contains more than one event.

Convention: Throughout this section, when we say *any time*, we mean any time between two events, not within an event; and when we refer to a local variable of a faulty process, we mean its value as of the time when the process failed. Also, for any variable x, $x(t)$ indicates the value of x as of time t; t may be omitted if no confusion may arise.

5.1 Process Status and Message Counting

The terms *neutral* and *engaged* are employed in the description of our algorithm without definition. They have similar meanings as in the DS algorithm and are formally defined below.

Definition 2 Initially, all processes are *engaged*. An engaged process becomes *neutral* once it successfully executes **A4**. A neutral process becomes *engaged* on finishing an **A2** action.

updated in $NumFault_i[j]$ — it could happen that $|S| < NumFault_i[j]$ as messages do not necessarily arrive in the FIFO order.

We have assumed in our model that every fault eventually be detected by every nonfaulty process. We have also seen that a process may know of a fault through the S set of a SIGNAL. In the rest of this paper, whenever we say a process detects a fault, we exclusively refer to the event of detecting the fault as assumed in the model, not counting that of knowing a fault through a SIGNAL.

When a faulty process x is detected by p_i, **F1** is invoked. The newly detected fault is added to DF_i and $out_i[x]$ is incremented by one. In normal case, $out_i[x]$ counts only outgoing basic messages. Here, we artificially increase the value of $out_i[x]$ so that p_i has chances to clean up the channel from x to it. It works as follows: 1) p_i sends a CLEAR message to x in **F1**; 2) as a return-flush, the message will be returned to p_i; 3) when p_i receives the returned CLEAR message in **F2**, $in_i[x]$ and $out_i[x]$ 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, $out_i[x]$ stays positive and thereby prevents p_i from becoming neutral and invoking **A4**. In **F1**, since a new fault is detected, there is a chance that p_i emerges as the new coordinator. So p_i checks if its index is the smallest among all nonfaulty processes. If so, p_i becomes new coordinator and updates $root_i$ and $parent_i$ accordingly. Otherwise, p_i becomes a child of process r, where $r = \min(P - DF_i)$. In the latter case, to the knowledge of p_i, process r is the coordinator. Of course, that may or may not be correct, depending on whether or not r is still alive.

4.3 Postponed Start

We mentioned in Section 3.2 that both the CV and the V algorithms start to run only after they switch to *detection mode*. The original DS algorithm does not have this "postponed start" feature, but the LTD variant implements it to the extent that no control messages need to be sent until the detector switches to the detection mode. Before that happens, although the detector is actually running, its work involves only simple computation such as counting outgoing and incoming basic messages and is cost-negligible.

For simplicity and clarity, the algorithm described in the previous subsection does not implement the postponed-start feature. It is not hard, though, to modify the algorithm to include that feature.

Basically, let each process initially at NDT (not detecting termination) mode. In this mode, each process passively collects information, updating local variables as in **A1**, **A2** and **F1**, but executes none of **A3**, **A4**, **A5**, **F2** and thus sends no control message.

Processes switch to DT (detecting termination) mode in one of two ways: 1) the present coordinator spontaneously decides to enter DT mode, or 2) processes enter DT mode on

A1: (When p_i sends a basic message to p_j)

$out_i[j] := out_i[j] + 1$; (* p_i counts outgoing messages. *)

A2: (When p_i receives a basic message from p_j)

if $(\neg root_i)$ and $(parent_i = \text{NULL})$ **then** $parent_i := j$;
$in_i[j] := in_i[j] + 1$; (* p_i counts incoming messages. *)

A3: (When p_i is idle and $\sum_{k \neq parent_i} in_i[k] > 0$)

for each j such that $j \neq parent_i$, $j \notin DF_i$ and $in_i[j] \neq 0$ **do**
send a SIGNAL($in_i[j]$,DF_i) message to p_j;
$in_i[j] := 0$; (* p_i signalled all messages received from p_j. *)
endfor

A4: (When p_i is idle, $out_i = \vec{0}$, $\sum_{k \neq parent_i} in_i[k] = 0$, and ($root_i$ or $parent_i \neq$ NULL))

if $\neg root_i$ **then**
send a SIGNAL($in_i[parent_i], DF_i$) message to $parent_i$;
$in_i[parent_i] := 0$; $parent_i :=$ NULL;(* p_i signalled parent messages. *)
elseif $(DF_i \supseteq KF_i)$ and $(\forall b \in P - DF_i - \{i\}$: $NumFault_i[b] = |DF_i|)$ **then**
declare termination; (* The barrier conditions establish. *)
endif

A5: (When p_i receives a SIGNAL(c, S) from p_j)

$out_i[j] := out_i[j] - c$; (* p_i records number of unacknowledged messages. *)
$KF_i := KF_i \cup S$;
$NumFault_i[j] := \max\{|S|, NumFault_i[j]\}$;

F1: (When p_i detects p_x's failure)

$DF_i := DF_i \cup \{x\}$;
$out_i[x] := out_i[x] + 1$; (* p_i is responsible for in-transit messages from p_x. *)
send a CLEAR message to x;
if $i = \min(P - DF_i)$ **then** (* p_i becomes coordinator. *)
$root_i :=$ true; $parent_i :=$ NULL;
else $parent_i := \min(P - DF_i)$; (* p_i becomes a child of coordinator. *)

F2: (When p_i receives a returned CLEAR message from p_x)

$in_i[x] := 0$; $out_i[x] := 0$; (* no in-transit deliverable messages between p_i and p_x *)

Figure 3: Algorithm for $p_i, 1 \leq i \leq n$.

3. Between time -1 and 0, no message is sent. At time 0, the original basic computation starts.

This artificially introduced computation between -2 and 0 has no effect on the original basic computation and, in particular, the original and the augmented will terminate at the same time. This enables us to observe the termination of the augmented instead of the original. This technique has been used in the LTD variant to make the DS algorithm applicable to even non-diffusing computations.

The algorithm for p_i, $1 \leq i \leq n$, comprised of a number of event-driven actions, is specified in Figure 3. Actions **A1**–**A5**, corresponding respectively to actions **D1**–**D5** of the DS algorithm, are responsible for termination detection; the other two actions, **F1** and **F2**, cope with faulty processes.

Action **A1** simply counts outgoing messages, and **A2** counts incoming ones. Moreover, in **A2**, upon receiving a basic message, if p_i is not the coordinator and has no parent, then it becomes a child of the process that sent the message. Intuitively, that means p_i becomes engaged.

When a process is idle, action **A3** is invoked to acknowledge all the unacknowledged received incoming messages not sent by its parent ($j \neq parent_i$) or detected faulty processes ($j \notin DF_i$). A SIGNAL message is issued to report the number of unacknowledged basic messages received ($in_i[j]$) and the set of faults detected (DF_i) so far.

Action **A4** is invoked when 1) p_i is idle, 2) all messages sent have been acknowledged ($out_i = \vec{0}$), 3)p_i has acknowledged all received incoming messages from processes other than its parent ($in_i[k] = 0$ for all $k \neq parent_i$, i.e., **A3** has been executed), and 4) either p_i is the coordinator or $parent_i \neq$ NULL. The condition $parent_i \neq$ NULL is included here to prevent this action from being repeatedly executed. When the first three conditions hold, p_i has essentially become neutral. Thus, if p_i is not the coordinator, it reports to its parent with a SIGNAL, and updates $in_i[parent_i]$ and $parent_i$ to reflect its neutral status. Otherwise, if p_i is the coordinator, then it checks whether $DF_i \supseteq KF_i$ and $NumFault_i[b] = |DF_i|$ for all $b \in P - DF_i - \{i\}$. If so, p_i declares termination. Since $NumFault_i[b] \leq |KF_i|$, when these two conditions hold, $DF_i = KF_i$. These two conditions roughly mean that all the processes that are nonfaulty according to the coordinator's knowledge have each reported to the coordinator the same set of faulty processes as have been detected by the coordinator itself. This provides a consistent view about the set of engaged faulty processes that have broken up the main tree, and thereby avoids otherwise possible premature, incorrect declaration of termination.

A5 is triggered when p_i receives a SIGNAL(c, S) message. In this action, $out_i[j]$, the number of unacknowledged messages, is reduced by c, and the set S is added to KF_i, the set of all faults known to i. Also, the cardinality of S, if bigger than previous ones, is

its "local" status.

- $in_i[1..n]$ — an integer array, where $in_i[j]$ indicates the number of messages received from p_j that p_i has not yet acknowledged.
- $out_i[1..n]$ — an integer array, where $out_i[j]$ records the number of messages sent to p_j that have not yet been acknowledged.
- $parent_i$ — indicating which process to report to when p_i becomes neutral.
- $root_i$ — indicating whether p_i is the coordinator.
- KF_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 known by p_i to be faulty.
- DF_i — the set of all processes that p_i itself has detected to be faulty. (By the assumption of fault detectability, every nonfaulty process eventually detects every fault.) DF_i and KF_i are updated independently; they may have intersection, but are not necessarily equal.
- $NumFault_i[1..n]$ — an integer array, where $NumFault_i[j]$ indicates p_i's knowledge about $|DF_j|$, the number of faults p_j has detected.

These variables are initialized as follows. Each process i at the beginning does not know any process failures, so $DF_i = KF_i = \emptyset$ and $NumFault_i = \vec{0} = (0, \ldots, 0)$. Let p_1 be the first coordinator and set $root_i$ accordingly: $root_1 = true$ and $root_i = false$ for $i \neq 1$. Variables $parent_i$ are initialized in such a way that the n processes form a tree (any tree) rooted at p_1 (e.g., $parent_i = 1$ for $i \neq 1$ and $parent_1 =$ NULL). Now initialize out_i and in_i according to the structure of this tree. Specifically, let $out_i[j] = in_j[i] = 1$ if p_i is p_j's parent in the tree, and 0 otherwise. (In the example tree, $out_1[j] = in_j[1] = 1$ for all $j \neq 1$.) Here we *envision* that each child has received an inherent basic message from its parent even before the basic computation starts. Or, more formally, we *augment* the basic computation which is assumed to start at time 0 by introducing an artificial computation between time -2 and 0:

1. Between time -2 and -1, let the root send a basic message to each of its children and let each node, upon receiving such a message, in turn send a basic message to each of its own children; repeat this until each leaf receives a basic message. Let all of these be finished by time -1.
2. At time -1, let each process become either active or idle according to its initial state as in the original basic computation.

The third problem is considerably harder, especially if more than one process fails. There are three issues here: a) *Which* nodes in T_p are to directly attach to T (every node or just the children of p)? b) *Which* nodes in T to attach to? c) *How* to actually attach a node q in T_p to a node r in T? It is not sufficient for q to simply send a message to r to inform it of the new child/parent relationship — due to message delay, by the time r receives the notice, it may have become neutral and left tree T.

In our approach, issues a) and b) are resolved by introducing a new child/parent relationship between *every* engaged process and the coordinator (the root of T.) If no process fails during the basic computation, the tree T evolves as in the DS algorithm. Whenever an engaged process detects a failure, it becomes a child of the coordinator. If, in the simplest case, all the engaged processes detect the failure at about the same time, then we have a new tree T' in which every surviving engaged process is a child of the coordinator. However, since not every process detects the failure at the same time, the transformation from T to T' does not achieve instantaneously but gradually. During the course of transformation, leaf nodes may become neutral and leave the tree, and neutral nodes may become engaged and join the tree (not necessarily as a child of the coordinator). The idea behind this scheme is that the coordinator, now aware that all engaged processes are to report to it, knows to wait for such reports.

Two questions immediately arise: How does the coordinator know whether or not it has received a report from *every* engaged process? (The coordinator does not know who is and who is not engaged.) How does the coordinator know that the report it has received from a process is the most recent one, as a process may send more than one report? Our strategy is to let neutral processes also send reports and to set a barrier synchronization at the coordinator with two conditions (to be described in the next subsection). When these two conditions are satisfied, the coordinator is able to answer the above two questions.

4.2 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 often referred to as process i or just i.

Two types of control messages are used: CLEAR and SIGNAL(c, S). A CLEAR message is a return-flush whose purpose, as described in Section 2.2, is to flush a channel between a nonfaulty process and a faulty process. A SIGNAL(c, S) message is sent 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 SIGNAL(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 variables, each subscripted with i to emphasize

it need not start running right from the beginning of the basic computation. It may start at a later time of the basic computation and still work correctly. This useful feature of "postponed start" has been incorporated into the DS algorithm in [16].

The CV algorithm was made fault-tolerant by 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 an entry into 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.

It should be noted that the V algorithm presented in [26] is actually only 1-resilient, but it was claimed by its author to be extendable along the directions as mentioned above.

4 $(N-1)$-Resilient Algorithm

As mentioned before, our algorithm is based on the LTD variant of the DS algorithm. We shall first discuss some of the problems that need to be solved in order to make the DS algorithm fault-tolerant.

4.1 Problems Caused by Faulty Processes

Recall that in the DS algorithm, all engaged processes form a tree. When this tree degenerates to nil, the computation is known to be terminated. At least three problems 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. Next, a nonfaulty process has to determine if the channel from a faulty process is empty. Third, after an engaged process p fails, the tree of all engaged processes, referred to as T, may break up into two subtrees: the subtree T_p rooted at p and the rest of T, which we shall still call T. In order for the DS algorithm or the LTD variant to work, it is necessary to re-attach to T the nonfaulty nodes in T_p or, in other words, to establish new child/parent relationships between the offspring of p and the nodes in T. It requires both the child and the parent to be aware of the existence of such a new relationship.

The second problem can be easily solved: the return-flush messages described in Section 2.2 helps a nonfaulty process determine whether there are any messages in transit from a faulty process. The first problem is easy to solve. Initially, let process p_1 be the coordinator. If it fails, let process p_2 take over the job of coordination. If process p_2 also fails, let process p_3 succeed, and so on. In general, just let the nonfaulty process with the smallest index be the coordinator.

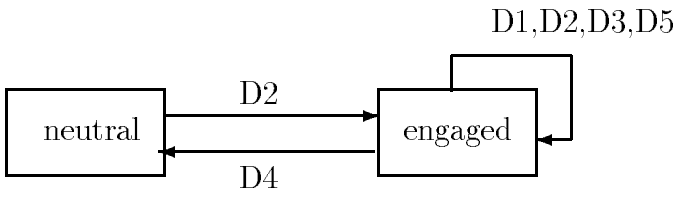

Figure 2: State Transitions of the DS algorithm

maintains three variables: $in_i[1..n]$, out_i, and $parent_i$, where $in_i[j]$ is the number of yet-to-acknowledge 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.) Figure 2 shows how a process changes between the engaged and neutral states. Note that a process $p_i \neq p_1$ is neutral iff $\sum_j in_i[j] = 0$ and $out_i = 0$; p_1 is neutral iff p_1 is idle and $out_1 = 0$. Also note that the variables $parent_i$ 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 neutral and drop off the tree. When the root becomes neutral, the computation is declared to be terminated.

The DS algorithm requires exactly M control messages, where M is the number of basic messages. That is optimal in terms of worst-case complexity. The DS algorithm has two drawbacks: it always requires M control messages and can only be applied to diffusing computations. These, however, can be easily overcome [16]. First, by letting each SIGNAL carry an integer c so that it acknowledges c basic messages at a time, the total number of control messages will be tremendously reduced. Second, by properly initializing the variables in_i, out_i and $parent_i$, the algorithm can be applied to non-diffusing computations as well. We shall adopt these extensions in our effort to make the DS algorithm fault-tolerant.

3.2 A Fault-Tolerant Termination Detector

Venkatesan's algorithm [26] is a fault-tolerant version of a previous algorithm by Chandrasekaran and Venkatesan [5].

The main idea of the CV algorithm is similar to that of the DS algorithm. But, instead of counting messages as in DS, 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 turns 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 acknowledgement to the corresponding message is received. Like the original DS algorithm, the CV algorithm, once provoked, needs a control message for each basic message. The most interesting feature of the CV algorithm is that

D1: (When p_i sends a basic message)

$out_i := out_i + 1$;

D2: (When p_i receives a basic message from p_j)

$in_i[j] := in_i[j] + 1$;
if $(p_i \neq p_1)$ and $(parent_i = \text{NULL})$ **then** $parent_i := j$; (* p_i becomes engaged. *)

D3: (When $\sum_k in_i[k] > 1$)

select a process p_j such that $(in_i[j] > 0)$ and $(j \neq parent_i$ if $in_i[parent_i] = 1)$;
send a SIGNAL to p_j;
$in_i[j] := in_i[j] - 1$;

D4: (When p_i is idle, $out_i = 0$ and $(p_i = p_1$ or $\sum_k in_i[k] = 1))$

if $p_i = p_1$ **then**
    declare termination;
else (* p_i becomes neutral. *)
    send a SIGNAL to $parent_i$;
    $in_i[parent_i] := in_i[parent_i] - 1$;
    $parent_i = \text{NULL}$;
endif

D5: (When p_i receives a SIGNAL from p_j)

$out_i := out_i - 1$;

Figure 1: Dijkstra-Scholten Algorithm for $p_i, 1 \leq i \leq n$.

but not equivalent to that of "idle/active". Initially, only the root is engaged; all others are neutral. 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 and an edge $p \rightarrow q$ 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 non-engagement message any time after the receipt. However, it can acknowledge an engagement message only if the three conditions are satisfied: i) the process is idle, ii) it has acknowledged all incoming non-engagement messages, and iii) 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 Figure 1 with p_1 acting as the root. Each process p_i

S1: Fault Detectability. After a process fails, every process knows of the failure within a finite amount of time.

S2: Forward Reliability. If a process does not fail, it will eventually receive all messages sent to it, but not necessarily in the order they were sent.

S3: Return Flush. If process p sends a *return-flush* message to a process q that is known to have failed, the communication network will return the message to p *after* having delivered all other messages from q to p.

S4: Undeliverable Messages. A message which is not a return-flush and whose destination process has failed is said to be *undeliverable*. The network discards all undeliverable messages.

Some comments about these assumptions are in order. First, fault detectability is necessary for fault-tolerant termination detection. It can be proved that without S1, no fault-tolerant termination detection is possible. The problem of detecting faulty processes is the topic of many papers (e.g. [7,22]). Second, the concept of return-flush is inspired by that of forward-flush [1]: If a message m is sent as a *forward flush*, every message sent before it to the same destination is guaranteed to be received before m. The return-flush primitive can be implemented using the technique of [1]. Third, S4 is not crucial. If the underlying network chooses to return all undeliverable messages back to their senders, our algorithm will still work well. However, in that case, a return-flush m from p to q must be implemented in a way such that it can "flush" the channel between p and q in both directions.

3 Previous Work

As mentioned in the introduction, our algorithm is based on the LTD variant of the Dijkstra-Scholten algorithm [11], and will be evaluated against the resilient protocol of Venkatesan's [26]. In this section, we review the DS and summarize the V algorithm. A thorough understanding of the DS algorithm will later help understand ours.

3.1 Dijkstra-Scholten Algorithm

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 not 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 to be either *neutral* or *engaged* depending on its status. The concept of "neutral/engaged" is a product of the algorithm. It is related

when a process *fails*, it stops running forever. A *faulty process* is one that has failed. This model of process failure is the same as that assumed in [12,26].

Assume that the underlying system can tolerate stop-failures. (Otherwise, there would be little point to use a *fault-tolerant* termination detector.) If processes fail during the basic computation, some basic messages (e.g. those addressed to faulty processes) may become *undeliverable* and eventually be discarded by the communication network. Exactly what messages are undeliverable is a characteristic of the underlying communication network and will be addressed in the next subsection. Here we just assume that undeliverable messages are eventually discarded by the network.

Definition 1 The computation of a fault-tolerant distributed system is *terminated* iff 1) every nonfaulty process is idle and 2) no deliverable basic message is still in transit in the underlying communication network. The problem of *termination detection* is to design a protocol that determines whether the computation of a given distributed system has terminated.

This definition is consistent with the traditional one. They refer to the same thing if no process fails during the basic computation and if every message is deliverable.

2.2 The Underlying Communication Network

As pointed out in [26], no fault-tolerant termination detector is possible in an asynchronous network in which logical channels are reliable but may incur arbitrary delays for messages. It is so even if messages are delivered in a FIFO fashion. To overcome the problem, [26] assumed three additional features of the underlying network: 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 is returned to the sender; c) each processor is able to send two or more messages *atomically* (the last feature was not explicitly stated in the model but did be used in their algorithm). Under these conditions, [26] presented a fault-tolerant algorithm for termination detection.

A close examination of the algorithm reveals that the above assumptions are not sufficient for the algorithm to work correctly. An additional condition is necessary: when a message sent from p to q is returned to p because q is faulty, the returned message must be able to "flush" all basic messages in the network that had been sent to p by q before its failure. This "flush" property is *not* a logical consequence of FIFO channels and condition b) above. A network that provides FIFO channels and can return undelivered messages does not necessarily provides the "flush" service. In fact, it can be shown that without additional assumptions no termination detector can tolerate even a single faulty process.

In this paper, we assume that the underlying communication network provides the following service primitives:

The main result of this paper is a fault-tolerant termination detector that meets the above requirements. Indeed, we have an algorithm based on a variant (the LTD variant [16]) of the well-known Dijkstra-Scholten (DS) algorithm [11]. The proposed algorithm is $(n-1)$-resilient, where n is the number of processes. It achieves fault tolerance by properly synchronizing the surviving processes and thus incurs no cost at all (in terms of control messages) in getting ready for possible faulty processes — if no process fails during the underlying computation our algorithm is as efficient as its predecessor (the LTD algorithm). The latter is one of the most efficient that are available in the literature; it requires only $M+n-1$ control messages in the worst case, which is optimal, and far less on the average. Should a process fail, 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).

Our algorithm also improves in other aspects: 1) the space complexity is $O(n^2)$ as compared with $O(nM)$ of the V-algorithm (as M is not known at compile time, the V algorithm requires the more expensive dynamic memory allocation); 2) our algorithm has a shorter detection delay of $O(n)$ as compared with the $O(M)$ required by the V algorithm; and 3) our algorithm does not need to send multiple messages atomically, which is crucial for the V algorithm.

The next section defines the problem to be studied. Section 3 reviews the DS algorithm and its variants. Section 4 proposes a fault-tolerant algorithm which is later proved correct and analyzed in Sections 5 and 6, respectively.

2 The Problem and the Underlying Network

2.1 Problem Definition

For our purpose, a *distributed system* is simply a set of n processes running on top of a communication network (e.g., at the application layer of the OSI ISO reference model). Each process switches between *active* and *idle* states. An active process can send and/or receive messages at will. An idle process can only receive incoming messages, and upon such an event it becomes active immediately. The underlying communication network (the lower four layers of the OSI model) is assumed to provide reliable end-to-end transport service.

In the literature, the computation of a distributed system without faulty processes is regarded as terminated iff all processes are idle and no basic messages are still in transit in the underlying communication network. The problem of termination detection is to design a protocol that determines whether the computation of a given distributed system has terminated.

We study in this paper the problem of termination detection *in face of faulty processes*. There are many kinds of process failures. The one studied in this paper is the stop-failure:

computation is called the *basic* or *underlying* computation, has terminated. A termination detector needs to exchange messages among its processes. These messages are called *control messages*, as distinguished from the *basic messages* of the underlying system.

Many algorithms for termination detection are available in the literature. Some algorithms assume synchronous communications [9,10,13,18,21,25], while others work equally well for asynchronous systems [6,11,14,17,19,20]. Some algorithms in the latter category require FIFO channels [6,20], and some do not [11,14,17,19]. Most algorithms are intended for static systems only; algorithms for dynamic systems can be found in [9,17].

Fault tolerance is one of the most desirable properties of distributed systems. While some problems are notoriously known to be unsolvable in the presence of faulty nodes[12], many distributed systems (algorithms) are able to tolerate failures of processes [2,3,4]. When a termination detector is applied to a fault-tolerant underlying system, it is desirable that the detector itself be fault-tolerant.

Unfortunately, none of the aforementioned termination detectors can tolerate even a single faulty process. That is not particularly surprising because termination detection is not easier than the consensus problem which is unsolvable in face of faulty nodes under the common assumption of reliable, FIFO channels for the underlying network [12].

There does exist a fault-tolerant termination detector — the V algorithm [26]. This algorithm is based on a previous non-fault-tolerant version, the CV algorithm [5]. The V algorithm is 1-resilient and was claimed by its author to be extendable to a k-resilient one. (A protocol or distributed algorithm is said to be *k-resilient* if it works correctly even in the presence of k faulty processes.) However, as so many protocols in the literature are known to be incorrect (the reader is referred to [8,15,24] for further comments on this), one does not know for sure whether such extension is possible without actually doing it. Even if the algorithm can be successfully extended to a k-resilient one, it has several drawbacks. First, the parameter k needs to be *pre-specified* by the user. Second, since the algorithm achieves fault-tolerance by replication, k copies are needed for each message transaction. As a result, the algorithm requires $\Omega(kM)$ control messages just to get itself prepared for possible process failures, where M is the total number of basic messages. This cost cannot be waived even if there is no process at all that fails during the computation. Third, for each failure that occurs, the algorithm requires about kM/n additional control messages on the average; that could be intolerably high in a message-intensive computation.

In most existing distributed systems, processes certainly may fail, but only occasionally or even rarely. Thus, a desirable fault-tolerant algorithm is one that can tolerate any number of faults, that runs as efficiently as the best available non-fault-tolerant algorithm if no process fails during the computation, and that incurs only a reasonable amount of cost for each process failure that actually occurs.

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

Ten-Hwang Lai and Li-Fen Wu

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

(614) 292-2146, lai@cis.ohio-state.edu

Abstract: A *termination detector* is a distributed algorithm that, when superimposed on a distributed system of n processes, is able to determine whether the computation of the underlying system has terminated. Fault tolerance is one of the most desirable properties of distributed systems. While some problems are notoriously known to be unsolvable in the presence of faulty processes, many distributed algorithms are able to tolerate failures of processes. When a termination detector is applied to a fault-tolerant underlying system, it is desirable that the detector itself be fault-tolerant. In most existing distributed systems, processes certainly may fail, but only occasionally or even rarely. Thus, a desirable fault-tolerant algorithm is one that can tolerate any number of faults, that runs as efficiently as the best available non-fault-tolerant algorithm if no process fails during the computation, and that incurs only a reasonable amount of extra cost for each process failure that actually occurs. In this paper, we present a fault-tolerant termination detection algorithm with these nice properties. The algorithm is much more efficient than existing (fault-tolerant) ones.

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

1 Introduction

A *distributed system* is a set of processes running on top of a communication network (e.g., at the application layer of OSI ISO reference model [23]). Each process switches between two possible states: *active* and *idle*. Active processes may send or receive messages at will. Idle processes can receive but cannot send messages. If an idle process receives a message, it becomes active immediately. The system is said to be *terminated* if two conditions are satisfied: i) all processes are idle and ii) there are no messages in transit. The problem of *termination detection* is to determine whether a distributed system has terminated. An algorithm that solves the termination detection problem is called a *termination detector*.

A termination detector itself may be regarded as a distributed system whose function is to determine if another distributed system, called the *basic* or *underlying system* whose