[10] D. Kumar, A Class of Termination Detection Algorithms for Distributed Computations, Tech. Rep. TR-85-07, Dept. of Computer Sciences, Univ. of Texas at Austin, May 1985.

[11] T.-H. Lai, Termination Detection for Dynamically Distributed Systems with Non-first-in-first-out Communication, *Journal of Parallel and Distributed Computing 3,* 1986, pp. 577-599.

[12] L. Lamport, An Assertional Correctness Proof of a Distributed Algorithm, *Science of Computer Programming 2*, 1982, pp. 175-206.

[13] E. L. Lozinskii, A Remark on Distributed Termination, *Proc. Fifth Intl.. Conf. on Distributed Computing Systems,* Denver, 1985, pp. 416-419.

[14] F. Mattern, Algorithms for Distributed Termination Detection, *Distributed Computing 2*, 1987, pp. 161-175.

[15] J. Misra and K. M. Chandy, Termination Detection of Diffusing Computations in Communicating Sequential Processes, *ACM Transactions on Programming Languages and Systems,* Vol. 4, No. 1, January 1982, pp. 37-43.

[16] J. Misra, Detecting Termination of Distributed Computations Using Markers, *Proc. Second Annual ACM Symp. on Principles of Distributed Computing,* Montreal, Aug. 1983, pp. 290-294.

[17] B. Rozoy, Termination for Distributed Systems with Asynchronous Message Passing: Model and Cost, Tech. Rep. 87.20, Laboratoire d'Informatique, Université de CEAN, France, Avril 1987.

[18] R. W. Topor, Termination Detection for Distributed Computations, *Information Processing Letters 18,* 20 January 1984, pp. 33-36.

[19] S. Venkatesan, Reliable Protocols for Distributed Termination Detection, *IEEE Transaction on Reliability*, Vol. 38, No. 1, April 1989, pp. 103-110.

cesses, and step by step reduced the problem of detecting *whether every process is free* to a simpler one of detecting *whether a single process is free*. With this theory at hand, the algorithm became a simple implementation.

The algorithm of [1] was recently extended to a fault-tolerant one [19]: even if a node fails, the algorithm still works correctly. We are currently investigating the possibility of extending our algorithm to tolerate not only one single fault, but any number of faulty nodes as long as the network is still connected.

References

[1] S. Chandrasekaran and S. Venkatesan, A Message-Optimal Algorithms for Distributed Termination Detection, *Journal of Parallel and Distributed Computing 8,* 1990, pp. 245-252.

[2] K. M. Chandy and L. Lamport, Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Transaction on Computer Systems,* Vol. 3, No. 1, February 1985, pp. 63-75.

[3] K. M. Chandy and J. Misra, How Processes Learn, *Distributed Computing (1986) 1,* pp. 40-52.

[4] S. Cohen and D. Lehmann, Dynamic Systems and Their Distributed Termination, *Proc. First Annual ACM Symp. on Principles of Distributed Computing,* Ottawa, Aug. 1982, pp. 29-33.

[5] E. W. Dijkstra and C. S. Scholten, Termination Detection for Diffusing Computations, *Information Processing Letters 11,* August 1980, pp. 1-4.

[6] E. W. Dijkstra, W. H. J. Feijen and A. J. M. van Gasteren, Derivation of a Termination Detection Algorithm for Distributed Computations, *Information Processing Letters 16,* June 1983, pp. 217-219.

[7] N. Francez and M. Rodeh, Achieving Distributed Termination without Freezing, *IEEE Transactions on Software Engineering,* Vol. SE-8, No. 3, May 1982, pp. 287-292.

[8] S.-T. Huang, Termination Detection by Using Distributed Snapshots, *Information Processing Letter 32,* August 1989, pp. 113-119.

[9] Y. M. Kim, T.-H. Lai and Y.-C. Tseng A Wait-Free Algorithm for Deadlock Detection and Resolution, *in preparation.*

while the CV-algorithm still needs $M' + 2 * |E| + n - 1$ message, as if every case were a worst case.

Our algorithm is an improvement on CV-algorithm. Basically, our algorithm uses a single **FINISHED**(k) message to replace k **REMOVE_ENTRY** messages of the CV-algorithm. The CV-algorithm does have an advantage: it's control message is of fixed size, while our algorithm may need as many as $\log M'$ bits for a **FINISHED**(k) message to carry the value of k. But the latter is just our algorithm's strength: by sending a single $\log k$-bit **FINISHED**(k) message, we save k $O(1)$-bit **REMOVE_ENTRY** messages.

Detection Delay: After the system terminates, in the worst case the CV-algorithm may cause a detection delay of M' messages, as compared to ours of n.

Space Complexity: The space required for the CV-algorithm is mainly for each process to maintain a stack. In the worst case, a stack may contain as many as M' entries, so the space complexity is $O(M')$ for each process, as compared to ours of $\Theta(n)$. Note that M' in general is an unbounded quantity not known at compile time, unless the underlying system specifically provides such information. As a result, the stacks used in the CV-algorithm may need to be implemented through dynamic memory allocation. In comparison, ours is a compile-time known quantity.

Postponed Start: Both algorithms provide the "postponed start" feature — they can be initiated sometime *after* the underlying system has started. Strictly speaking, our algorithm starts to operate when the very first basic message is sent. But until the root process enters DT mode and triggers the core of the algorithm, the computational overhead is only to increment a counter while sending/receiving a basic message, and no control message is sent.

FIFO vs. Non-FIFO: The CV-algorithm works only in a FIFO environment. Our algorithm works as well even in a Non-FIFO network.

5 Concluding Remarks

We have proposed an algorithm for distributed termination detection. It is optimal in terms of worst-case message complexity, and is more efficient in many other aspects than the existing message-optimal algorithm, namely the one presented in [1].

Before describing the algorithm, we introduced the concept of free and loaded pro-

terminates) plus one, the "one" accounting for the delay caused by the **FINISHED** messages generated by procedure *minor_respond*. So the longest delay is n.

Computation Time: **A1** and **A2** are each executed for exactly M times (for n processes); **A3** exactly n times; **A4** $M + n - 1$ times at most (since there are up to $M + n - 1$ **FINISHED** messages); **A5** no more than $M + n$ times (since there are at most n active processes at initial time and at most M basic messages that can make processes active). Each call to *respond_minor* costs $O(n)$; each call to *respond_major* takes constant time. Thus, the total computational cost for n processes is $O(Mn + n^2)$, or $O(M + n)$ for each process.

Space Complexity: The space complexity is $\Theta(n)$ for each process. If we count the bit complexity, it will be $\Theta(n \cdot \log M)$.

4.2 Compared with Chandrasekaran and Venkatesan's Algorithm

In the CV-algorithm, each process maintains a stack that keeps track of the process's all sending and receiving activities. Whenever a basic message is sent (received), a sending (receiving) record is pushed onto the stack. If a message m is sent from p to q, then the sending-m record in p's stack is locked by the receiving-m record in q's stack. The lock is released when (i) q is idle and (ii) there is no other record on top of q's receiving-m record. On satisfying these conditions, q pops its receiving-m record and sends a **REMOVE_ENTRY** signal to p for p to remove its sending-m record. These rules apply to all processes and a single stack in each process is used for all its communication activities. When a process is idle and has an empty stack, it makes a **TERMINATE** report. When every process has reported, the system is terminated.

Message Complexity: If M' is the sum of the number of basic messages sent by each process after entering DT mode, then the CV-algorithm uses precisely $M' + 2 * |E| + n - 1 = \Theta(M' + |E|)$ control messages — M' **REMOVE_ENTRY** messages, $2 * |E|$ **START** (named as **WARNING** in the original paper) messages, and $n - 1$ **TERMINATE** messages. For a meaningful comparison, assume that both algorithms — the one presented in this paper and the CV-algorithm — use the same strategy for the root process to decide when to enter DT mode, so that both have the same M' value. (This assumption is in favor of the CV-algorithm, for even if the root enters DT mode in both algorithms at the same time a process may receive a **START** message earlier in the CV-algorithm than in ours.) Even in the worst case, our algorithm issues no more than a total of $M' + 2 * (n - 1)$ control messages, always less than theirs. In a non-worst case computation, our algorithm uses less than $M' + 2 * (n - 1)$ messages,

exactly $n-1$ of them. **FINISHED** messages are used to report the finish of basic messages (including the $n-1$ imaginary messages introduced to S'). If, as in a worst case, a **FINISHED**(1) is used for every basic message, then a total of $M+n-1$ **FINISHED**'s will be needed. Thus, our algorithm requires $M+2(n-1) = O(M+n)$ control messages in all in the worst case. This complexity is asymptotically optimal since any termination detection algorithm requires at least $\Omega(M+n)$ control message in the worst case. ($\Omega(n)$ is an obvious lower bound since in the worst case each process, except for one, has to report at least once. The $\Omega(M)$ lower bound was established in [3].)

Although our algorithm may use up to $M+2(n-1)$ control messages in the worst case, fewer than that is necessary on the average. Rigorously analyzing its average-case performance is very difficult, if not impossible, due to lack of knowledge of patterns in which processes in a distributed system exchange messages and switch between active and idle states. Here we satisfy ourself by considering a simple, hypothetical distributed system, in which traffic loads on any two channels are independent of each other. Focus on an interval of time $I = (t_1, t_2)$ in which a particular process, p_i, becomes active at t_1, turns idle at t_2, and otherwise is active and in DT mode for the entire period. Let p_j be any arbitrary but fixed process connected to p_i by a channel. During interval I, assume that basic messages from p_j arrive at p_i according to some stochastic process. Let X be the total number of such messages arriving at p_i during I. Let q_k, $k \geq 0$, be the probability that p_i receives exactly k basic messages from p_j during period I; i.e., $q_k = Prob(X = k)$. At the end of I, if $parent_i \neq j$, then p_i sends a **FINISHED**(X) message to p_j iff $X > 0$. The expected number of control messages sent from p_i to p_j in interval I is $0 \cdot q_0 + 1 \cdot (1 - q_0) = 1 - q_0$, and even smaller if $parent_i = j$. If the random variable X follows a Poisson distribution ($q_k = \lambda^k e^{-\lambda}/k!$, where λ is the expected value of X), then the ratio of the expected number of control messages from p_i to p_j, to the expected number of basic messages sent from p_j to p_i in period I is

$$R = \frac{1 - q_0}{\text{mean of } X} = \frac{1 - e^{-\lambda}}{\lambda} \tag{1}$$

For $\lambda = 1, 2, 3, 4, \ldots, 9, 10$, $R = 0.63, 0.43, 0.31, 0.24, \ldots, 0.11, 0.09$, respectively.

These discussions, not by any means a formal analysis, should have shed some light on our algorithm's potential in saving control messages in non-worst cases.

Detection Delay: Detection delay is defined to be the length of the longest communication path along which the algorithm, after the system terminates, needs to sequentially pass control messages in order to detect the termination. Clearly, the detection delay of our algorithm is the height of T_1 (as of the time when the system

Proof. When p_1 enters procedure *respond_major* and reports termination, p_1 is A-free. By Corollary 1, p_1 is the only process in T_1. By **P3** of Lemma 3, all other processes are A-free, and, then by Lemma 2, the system is terminated. □

The above theorem shows that the algorithm never mistakenly reports termination. The next theorem shows the algorithm to be deadlock-free.

Theorem 5 *Assume that p_1 is guaranteed to enter DT mode at some point of time. Then after the underlying distributed computation is terminated, the algorithm will report termination in finite time.*

Proof. Assume that the underlying system will eventually terminate and p_1 will eventually switch to DT mode. With reliable networks, there is a time instant t at which the system is terminated and every process is in DT mode. If by this time p_1 has reported termination, then the theorem is proved. So assume that p_1 has not done so.

Observe that at time t, $in_i[j] = 0$ for all i and j unless $j = parent_i$. This is because when the last time p_1 turned idle or entered DT mode, whichever occurring later, it invoked procedure *respond_minor* from either A_5 or A_3. Also observe that for any p_i, after time t, if $parent_i \neq NULL$, then the only reason is that $out_i \neq 0$.

Now, consider the tree T_1 established in **P1** of Lemma 3. Let p_i be any leaf node with a non-zero out_i. By Lemma 1, there must exist some **FINISHED** message(s) (with counter(s) summing up to out_i), which will eventually arrive at p_i, making p_i leave T_1 through **A4**. By simple induction, one readily sees that T_1 will eventually shrink to a single node and that p_1 will eventually have $out_1 = 0$ and report termination. □

4 Performance Analysis and Comparisons

We now analyze our algorithm's performance in terms of number of control messages, detection delay, time complexity, and space complexity. Then, we compare our algorithm with the CV-algorithm [1], which is the only message-optimal termination detection algorithm known to us.

4.1 Performance

Message Complexity: Our algorithm uses two types of control messages, **START** and **FINISHED**. **START** messages are diffused through the spanning tree, so there are

Initially, $parent_i$ is set to p_i's parent in T. T_1 is identical to T if the edges of T are regarded as directed from children to parents. The three properties are obviously true.

Assume that the three properties hold immediately before an action α, which is anyone of **A1** – **A5**. We show that after the execution of α they still hold. We only have to consider these events: (i) some process becomes A-loaded, (ii) some process becomes a child of another process, and (iii) some process in T_1 breaks away from T_1. If none of them occurs, then **P1** – **P3** obviously remain valid. Consider these cases one by one.

(i) Suppose that an A-free process p_i changes status to A-loaded in action α. We have to show p_i to be part of T_1 after α. It follows from definitions of A-free and A-loaded that in α either out_i is incremented or p_i turns active, and thus α is either an **A1** or **A2** action. The former case cannot happen since p_i is idle prior to α. In the latter case, if $p_i = p_1$ then there is no change in F's structure and the lemma remains true. If $p_i \neq p_1$ in **A2**, then $parent_i$ is set to j in the **if** statement of **A2**, with $in_i[j]$ just being incremented. In this case, by Corollary 1, $out_j \neq 0$ and p_j is A-loaded (at the time immediately before α), and by **P2** $p_j \in T_1$. After α, being a child of $p_j \in T_1$, p_i is a member of T_1, and **P2** holds. **P3** still holds, since aside from p_i all nodes are intact in α. Since $parent_i = NULL$ before α — meaning $p_i \notin T_1$ — T_1 remains an in-tree after p_i joins it.

(ii) Suppose that process p_i becomes a child of p_j in executing action α. **A2** is the only action in which such an event may occur; this has been considered in (i).

(iii) Suppose now that some process p_i breaks away from T_1 by changing $parent_i$ to NULL. One readily sees that it may occur in procedure *respond_major* only, and that when it occurs p_i is idle and has $out_i = 0$, or equivalently, p_i is A-free. Furthermore, with $out_i = 0$, p_i is a leaf node (by Corollary 1). After setting $parent_i$ to NULL, p_i leaves T_1 and becomes an isolated node. All the three properties hold after α. □

From the above proof, one may observe that before the algorithm enters DT mode, some leaf nodes of T_1 may be A-free. Once entering DT mode, such nodes immediately leave T_1 and thereafter T_1 contains only A-loaded nodes. During the computation, T_1 sometimes grows, sometimes shrinks. Eventually it shrinks to a single node and that is when p_1 detects termination. We have to show that the algorithm won't report termination by mistake.

Theorem 4 *When p_1 reports termination, the underlying system is really terminated.*

Proof. The first part is a direct implication of the lemma. The second part follows from the lemma and the observation that whenever $parent_j = i$, $in_j[i] \neq 0$. □

In our algorithm, a process p_i is thought to be free when $out_i = 0$ and p_i is idle. As mentioned above, there is a difference between facts and observations. Thus, call a process p_i *A-free* (*approximately free*) iff p_i is idle and $out_i = 0$; and call a process *A-loaded*(*approximately loaded*) iff it is not A-free. Define *A-minor* and *A-major* messages, as well as *A-finished* and *A-unfinished* messages, as in Definitions 2 and 3, but with respect to A-free and A-loaded processes. One may prove the algorithm correct by establishing Theorems $1-3$ for A-free processes. But here we choose to take a slightly different approach.

Lemma 2 *During the execution of the algorithm, when all processes are A-free, the system is terminated.*

Proof. If all processes are A-free, by definition they are all idle. Besides, since $out_i = 0$ for all p_i's, by Corollary 1 no process has undelivered messages in its outgoing channels. So the system is terminated. □

The variables $parent_i$, $1 \leq i \leq n$, altogether define a directed graph: $F = (V, E)$, where $V = \{p_1, \ldots, p_n\}$ is the original process set and $E = \{\langle p_i, parent_i \rangle : parent_i \neq NULL, 1 \leq i \leq n\}$. Let $V_1 = \{p_i :$ there is a path in F from p_i to $p_1\}$, and let T_1 be the subgraph of F that is induced on V_1. The following lemma indicates that F is a forest of in-trees (trees with edges directed from children to parents), and that T_1 is the only possible nontrivial tree in F — every process not in T_1 is an isolated node, having no child and no parent. Besides, all A-loaded processes belong to T_1.

Lemma 3 *At any point of time during the algorithm's execution, the following properties hold:*

P1 *: T_1 is an in-tree.*

P2 *: If p_i is A-loaded then it belongs to T_1.*

P3 *: If p_i is not in T_1, then it is an isolated node.*

Proof. Recall that the execution of the algorithm is assumed to be a single sequence of actions. Thus we can prove by induction (or assertional proof as called in [12]).

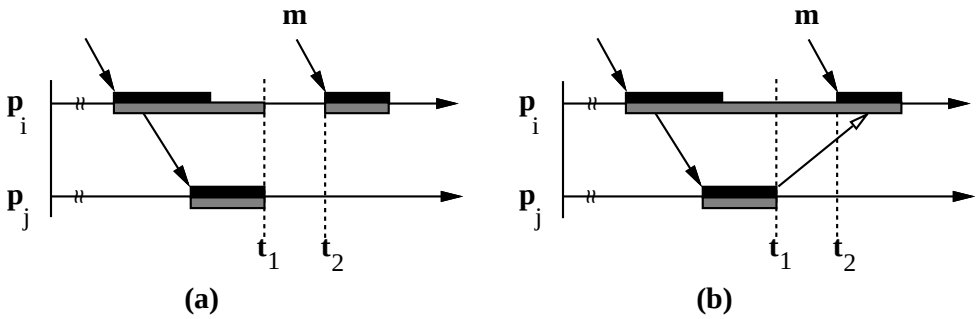

Figure 5: Difference between absolute facts and processes' observations.

The computation of the system coupled with the termination detection algorithm is a set of n sequences of atomic actions, one sequence for each process. However, we may regard each action as occurring at a single point of time without overlapping any other process's action, and, as a result, the n sequences can be combined as a single sequence. It has been shown in [9] that if an algorithm works correctly for "serialized" computations, it will work correctly for actual, non-serialized computations.

Through out this subsection, when we say *any time*, we mean any time between atomic actions, not within an action.

The next lemma describes the invariant property maintained by our message-counting scheme. This lemma holds no matter the channels are FIFO or Non-FIFO.

Lemma 1 *Let p_i and p_j be different processes with a channel in between. Let b_j be the number of basic messages still in transit in the channel from p_i to p_j, and let* **FINISHED**$(c_{j,1})$, **FINISHED**$(c_{j,2})$, ..., **FINISHED**(c_{j,m_j}) *be all in-transit* **FINISHED** *messages from p_j to p_i. Then*

$$out_i = \sum_{j \neq i} (b_j + in_j[i] + \sum_{k=1}^{m_j} c_{j,k}).$$

Proof. Initially, $out_i = \sum_{j \neq i} in_j[i]$ = (number of p_i's children), and all channels are empty, so the equation holds. By induction, it is easy to verify that after any execution of **A1**–**A5**, the equation still holds. □

Corollary 1 *If $out_i = 0$, then all p_i's outgoing channels are empty and $in_j[i] = 0$ for all p_j; besides, $parent_j = i$ for no p_j.*

messages. Since minor messages are finished on their arrival, p_i could send a notice to the sender whenever it receives a minor message. But for the sake of saving control messages, we let p_i send such notices only when it is idle and, of course, in DT mode. Thus, the procedure is called only if p_i is idle when entering DT mode (**A3**), or if p_i is in DT mode when turning idle (**A5**). Recall that $in_i[j]$, $j \neq parent_i$, keeps the number of "unacknowledged" minor messages. Procedure *respond_minor*(i), when invoked, sends a **FINISHED**($in_i[j]$) message to every p_j, $j \neq parent_i$, such that $in_i[j] \neq 0$.

Procedure *respond_major*(i) is for p_1 to report termination and for p_i, $i \neq 1$, to report the finish of major messages. A process makes such reports only when it becomes free (or more precisely, when it detects that it has become free) and is in DT mode. In our algorithm, p_i regards itself free when it is idle and $out_i = 0$. Thus, the procedure may be invoked from any of **A3**, **A4**, and **A5**. Procedure *respond_major*(i), once invoked, first checks if $out_i = 0$; and if so, then either p_i ($i = 1$) reports termination immediately, or p_i ($i \neq 1$) sends a **FINISHED**($in_i[parent_i]$) message to $parent_i$. In the latter case, $parent_i$ is set to NULL to reflect p_i's free status. Note that, in **A2**, upon receiving a basic message, if p_i is indicated by a null $parent_i$ to be free, then $parent_i$ is set to the message's sender to reflect p_i's new loaded status. A non-null $parent_i$ also serves to indicate that one of the messages counted by $in_i[parent_i]$ is a major message. When $in_i[parent_i] > 1$, the **FINISHED**($in_i[parent_i]$) message actually reports the finish of one major message and $in_i[parent_i] - 1$ minor one(s) at one time.

A4 describes a process's reaction to an incoming **FINISHED**(k) message. Note that there is no need to call procedure *respond_minor* in **A4** since all values in array $in_i[1\,..\,n]$, except for $in_i[parent_i]$, are zero.

3.2 Correctness Proof

Since p_1 reports termination when it detects its "free" status, the correctness of the algorithm seems to directly follow from Theorem 3. Unfortunately that is not the case. In Section 2, we took a "global view" and defined the terms free, loaded etc. in terms of *absolute facts* rather than *what observed by processes*. Owing to message delay, a process's observation of a message's finished/unfinished status — or even its own free/loaded status — may be different from the fact. For instance, referring to Fig. 5(a), suppose that p_j becomes free at time t_1, and, accordingly, so does p_i. Also suppose that the message m received by p_i at t_2 is a major message. However, in our algorithm (see Fig. 5(b)), if the **FINISHED** message sent by p_j arrives at p_i after t_2, then m will be treated by p_i as a minor message. Fortunately, despite such possible difference between observations and facts, the algorithm still works correctly.

```
Procedure respond_minor(i: integer)
begin
    for each j ≠ parent_i with in_i[j] ≠ 0 do
        send a FINISHED(in_i[j]) to p_j;
        in_i[j] := 0;
    end for;
end;

Procedure respond_major(i: integer)
begin
    if (out_i = 0) then
        if (i = 1) then report termination
        else
            send a FINISHED(in_i[parent_i]) to parent_i;
            in_i[parent_i] := 0;
            parent_i := NULL;
        end if;
    end if;
end;
```

Figure 4: Procedures to respond major and minor messages.

Initially p_i is in NDT mode. In this phase, p_i only counts incoming and outgoing messages; no control message is sent. **A1** and **A2**, activated by the events of sending and receiving basic messages respectively, perform the counting actions. The **if** statement in **A2** is of no use at this point since its condition is always false.

Process p_i enters DT mode by executing **A3**, which is activated at p_1 and at other processes by different events: p_1 by its deciding to enter DT mode and other processes by receiving a **START** message. (For flexibility, we leave it unspecified as to when p_1 should enter DT mode. A reasonable choice is when p_1 becomes idle for the first time.) By executing **A3**, p_i switches to DT mode and sends a **START** message to each child — recall that initially the processes are organized as a tree. The **if** statement of **A3** is addressed below.

After entering DT mode, p_i occasionally has to inform other processes of the number of messages it has finished, by sending a **FINISHED**(k) message, where k indicates the number of messages finished. It does so by calling procedure *respond_minor* and/or *respond_major*.

Procedure *respond_minor*(i) is for p_i to send notices to report the finish of minor

A1: (Upon sending a basic message to p_j)
$out_i := out_i + 1$;

A2: (Upon receiving a basic message from p_j)
$in_i[j] := in_i[j] + 1$;
if $(parent_i = \text{NULL}) \wedge (i \neq 1)$ **then** $parent_i := j$;

A3: (Upon deciding to switch to DT mode) /* for p_1 */
or (Upon receiving a **START** message) /* for p_i, $2 \leq i \leq n$ */
$mode_i :=$ DT;
for each child p_j of p_i **do**
send a **START** message to p_j;
end for;
if (p_i is idle) **then**
call *respond_minor(i)*;
call *respond_major(i)*;
end if;

A4: (Upon receiving a **FINISHED(k)** from p_j)
$out_i := out_i - k$;
if $(mode_i = \text{DT}) \wedge$ (p_i is idle) **then call** *respond_major(i)*;

A5: (Upon turning idle)
if $(mode_i = \text{DT})$ **then**
call *respond_minor(i)*;
call *respond_major(i)*;
end if;

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

START message to every other process along the edges of T. (We leave it unspecified as to how the root decides when to start the algorithm. A reasonable approach is to start it when the root becomes idle for the first time.) Strictly speaking, our algorithm starts to operate at the same time as the underlying system, because each process has to count the messages it sends. But until the root issues a **START** message, no control message has ever been sent between processes. Before that, only simple, passive bookkeeping needs to be done. Thus, we use the word *start* in the sense of starting to use control messages. The reason for not starting earlier is simply for message efficiency.

The **FINISHED(k)** message is used in our message-counting scheme to inform a process that k messages sent by it have been finished.

Our algorithm checks if S is terminated by checking if the augmented system S' as defined preceding Theorem 3 is terminated. Each process p_i, $1 \leq i \leq n$, maintains four variables, each initialized to reflect the imaginary events occurring during time $(-1, 0)$:

- $in_i[1\,..\,n]$ — an integer array, where $in_i[j]$ counts the number of messages received from p_j which either have not been finished, or have been finished but have not yet been so noticed to p_j. Initially, $in_i[j]$ is 1 if p_j is p_i's parent in T, and 0 otherwise. Note that p_1 has no parent.

- out_i — an integer that indicates the number of unfinished (or thought-to-be-unfinished) messages sent by p_i. Initialize out_i to the number of children p_i has in T.

- $mode_i$ — a boolean variable indicating whether p_i is in DT (Detecting Termination) or NDT (Not Detecting Termination) mode. Initially, p_i is in NDT mode; only after receiving a **START** message, it changes to DT mode.

- $parent_i$ — a pointer for a loaded p_i to indicate where the most recent major message came from. Since each message introduced to S' between times -1 and 0 is a major message, $parent_i$, $i \neq 1$, is initialized to p_i's parent in T, and $parent_1$ = NULL.

The algorithm for p_i, $1 \leq i \leq n$, is given in Fig. 3. The algorithm makes use of two procedures which are given in Fig. 4.

We describe the algorithm as five event-driven atomic actions: **A1** – **A5**. Whenever the event for an action occurs, the corresponding action is taken. We explain the algorithm by following p_i's three phases: when it is in NDT mode, when it switches to DT mode, and when it is in DT mode.

4. By time 0, each process except for p_1 has received exactly one message; p_1 receives no message during $(-1, 0)$.

5. At time 0, let the active/idle status of each process be exactly the same as in S.

6. After time 0, S' works exactly the same as S.

Clearly, S' is terminated if and only if S is terminated. The termination of S' is somewhat easier to detect owing to the following theorem, which follows directly from Theorem 2 and the fact that every process p_i, $i \neq 1$, is free at time -1.

Theorem 3 *S' is terminated if and only if p_1 is free.*

In the next section we shall develop an algorithm that detects S's termination by checking whether S' is terminated. In order to do so, it is not necessary to actually convert S to S'. All what we need is to properly initialize some variables so as to reflect the extra computation of S' during time interval $(-1, 0)$. In the algorithm, p_1 will report termination once it finds itself to be free.

3 Termination Detection Algorithm

3.1 Algorithm

Our algorithm is based on the concepts developed in the preceding section. In particular, we try to apply Theorem 3. The key problem in taking this approach is how to decide whether the messages sent by a process have all been finished. We solve the problem by *message counting*. Each process knows how many messages it has sent. By letting the receiver of a message responsible for informing the sender when the message is finished, each process is able to decide if all of its outgoing messages have been finished.

Let $S = \{p_1, p_2, \ldots, p_n\}$ be the process set of the underlying system. Let the processes be organized as a spanning tree T; that can be done at compile time. Without loss of generality, let p_1 be the root of T.

Two types of control messages are used in our algorithm — **START** messages and **FINISHED**(k) messages, where k is an integer. The **START** messages are used to "start" the execution of the algorithm. The root starts the algorithm by sending a

Now consider the major message $m'(p,q)$ that loads q at time $t^-(I_q)$. As of time t, m' is still unfinished, and thus p, the sender of m', is still loaded. With p, following a similar argument to that for q, there is a message m'' that loads p at time $t^-(I_p)$, where I_p is the loaded interval of p that spans from $t^-(I_p)$ to t and possibly beyond. Clearly, m'' and m' are different messages, since receive-time(m'') $<$ send-time(m').

By a simple induction, we have an infinite sequence of distinct messages, which of course is impossible since the number of messages sent before time t is finite. This contradiction establishes the theorem. □

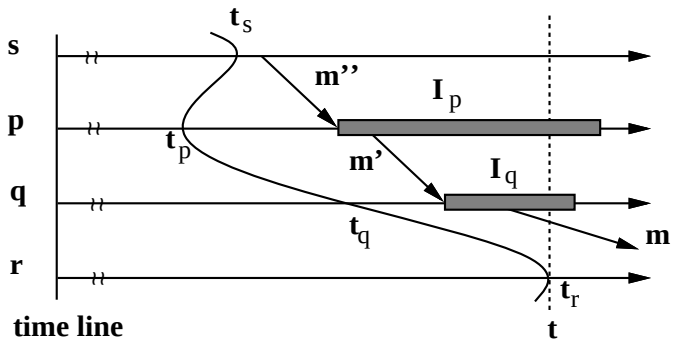

Figure 2: A sequence of unfinished messages.

Theorem 2 is certainly easier to apply than Theorem 1, but further simplification is still possible.

Let $G = (S, E)$ be a distributed system, where $S = \{p_1, p_2, \ldots, p_n\}$ is the process set and E the channel set. Organize the processes in S as a spanning tree T along some channels in E, with p_1 at the root. Assume that S starts to operate at time 0. Define an *imaginary* system S' that starts at time -1. During time interval $(-1, 0)$, an imaginary message is passed from the root process to every other process along the edges of T, and starting from time 0, S' behaves exactly the same as S. More specifically, S' works as below:

1. At time -1, p_1 is active and all other processes are idle.
2. p_1 sends a message to each of its children in T; these message are immediately received.
3. On receiving such a message, each process in turn sends a message to each of its children (if any) in T. Again, these messages are immediately received.

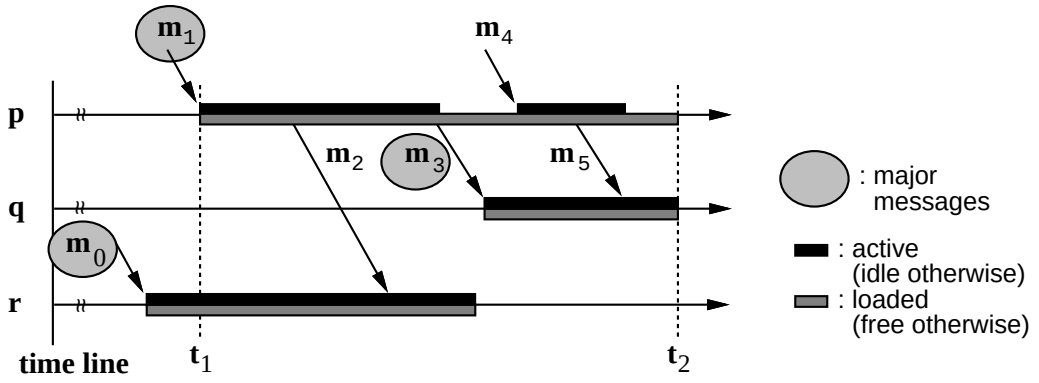

Figure 1: Example of major/minor messages and free/loaded processes.

However, this theorem does not seem to make the job of termination detection any easier — a free process may become loaded again by receiving a message, just as an idle process may become active on that occasion. Detecting the global state of *all processes being free* seems to be as hard as the original termination detection problem. Further reduction is desirable.

Call every maximal interval of time in which a process is loaded a *loaded interval* for that process, and similarly define a *free interval*. Thus, a process's loaded intervals alternate with its free intervals without overlapping. To avoid unnecessary confusion, regard each interval as being left-closed and right-open.

While Theorem 1 indicates that the system is terminated at time t if all processes are free *at* that moment, the next theorem tells us that the system is terminated at time t if each process has ever been once free *by* the time.

Theorem 2 *During the computation of a distributed system, if every process has ever been free for at least once, then the system is terminated.*

Proof. Let S be a distributed system. For every process $p \in S$, let t_p be the first time p becomes free ($t_p = 0$ if p is initially free). We show S terminated at time $t = \max\{t_p : p \in S\}$. Assume, to the contrary, that S is not terminated at time t. Then there exists either an active process, say, q or a still in-transit message, say, $m(q, r)$ at time t. (We denote by $m(q, r)$ a message m with sender q and receiver r.) In both cases, q is a loaded process at t, by Definition 1. Let I_q be the loaded interval of q such that $t \in I_q$. Let $t^-(I_q)$ denote the starting point of interval I_q. Consider the relationship between $t^-(I_q)$ and t_q, the first time q turning free. Obviously, $t_q < t^-(I_q)$, since q is free at $t_q \leq t$ and loaded in the entire period $I_q \ni t$. (See Fig. 2.)

is, with this interpretation, we may call a minor message finished once it is received. With minor messages being finished as soon as they are received, once the the major message m is finished, the process p becomes free again.

We now formally define the terms *loaded, free, major, minor, finished*, and *unfinished* recursively in terms of each other.

Definition 1 At any instant of time during the system's computation, a process is called *free* iff it is idle and all messages it has sent to other processes have been finished. A process is *loaded* iff it is not free.

Definition 2 A *major* message is one that is received by a free process. A *minor* message is one that is received by a loaded process.

Definition 3 A minor message is *finished* once it is received by the receiving process. A major message is *finished* when the process loaded by it once again becomes free. A message is *unfinished* until it is finished. (Thus, in particular, a message in transit is unfinished.)

Fig. 1 gives an example of the above definitions. The period in which a process is active (loaded) is represented by a black bar (gray bar). Note that an active process must be loaded, but the reverse is not necessarily true. At t_1, message m_1 arrives at a free process p, making p loaded and active, and thus is a major message. The message m_2 received by a loaded process r is a minor message and is finished upon receipt. After p sends message m_3 and turns idle, it remains loaded, waiting for m_3 to be finished. (m_3 is a major message which makes q loaded.) But before m_3 is finished, m_4 activates p to later send a message m_5; both m_4 and m_5 are minor messages. Finally, m_3 is finished at t_2, at which time both q and p become free.

The following theorem relates the concept of free processes to that of system termination.

Theorem 1 *When all processes are free, the system is terminated.*

Proof. If all processes are free, then every process is idle and every message that has been sent is finished and thus not in transit. By definition, the system is terminated. □

2 Foundation

We assume the common model of a distributed system as used by most researchers in the literature of termination detection. Thus, a distributed system consists of a finite set of processes and a finite set of channels. It is described as an undirected graph in which vertices represent processes and edges represent channels. We assume asynchronous, bidirectional, Non-FIFO channels, in which messages are delivered without errors; with arbitrary, finite delays; but not necessarily in the order in which they were sent.

The computation of a process is a sequence of atomic actions. In an action, a process may send or receive a message (but not both) and change local states, or it may just change local states without involving any message. Each process switches between *active* and *idle* states in an unpredictable fashion. Active processes send and receive arriving messages freely. An idle process can only receive messages, and on such an occasion it becomes active immediately. The system is *terminated* if and only if (i) all processes are idle and (ii) no message is in transit. The *termination detection problem* for a distributed system is to detect the system's termination after it occurs.

In a distributed system, processes communicate by messages. In order to solve the termination detection problem, we *interpret* each message as a job: by sending a message, the sending process asks the receiving process to do a job. (The job represented by a message m will simply be referred to as job m.) Furthermore, we regard each process as being either *loaded* (with a job) or *free*. Before formally defining these terms, let us explain their intuitive meanings.

Informally, when the system starts running, each active process is loaded and each idle process is free. Later on if a free process p receives a message m then it becomes loaded, and it is p's responsibility to make sure that the job represented by m be finished. If p later ever sends a message m' to q, we regard that activity as p delegating part of job m to q. Thus, job m can be regarded finished only after m' is finished.

We distinguish between two types of messages. A *major* message is one that is received by a free process; a *minor* message is one that is received by an already loaded process. Whether a message is major or minor depends upon the receiver's status as of the time when it receives the message. The message m mentioned in the preceding paragraph is a major message. Let us go back to the above scenario. Suppose that after p becomes loaded (because of m) but before m is finished, p receives another message m'', which is a minor one. If we imagine that job m'' is merged into job m and still call the resulting job m, then we can regard job m'' as having been finished. That

Until recently, all existing general-purpose termination detection algorithms require $\Omega(Mn)$ control messages in the worst case, where M and n are defined as before. The gap between this barrier and the above mentioned theoretical lower bound was recently closed by Chandrasekaran and Venkatesan with an $O(M + |E|)$ algorithm [1]. This algorithm (called CV-algorithm in the following) is not only message-optimal, it also enjoys a useful feature: it doesn't have to start simultaneously with the underlying computation. Thus its message complexity is actually $O(M' + |E|)$, where M' is the number of basic messages issued by the underlying system after the termination detection algorithm has started to operate.

The CV-algorithm, however, has several undesirable properties. First, it always requires $M' + 2 * |E| + n - 1$ control messages, whether in the worst or the best case. Second, after the underlying system terminates, the algorithm may have to sequentially send as many as M' control messages in the worst case. In other words, its worst-case detection delay is $\Theta(M')$. In a message-intensive computation, that might not be tolerable. Third, the maximum amount of space needed by each process is $\Theta(M')$, a quantity generally not known at compile time, making it necessary to use the more expensive dynamic memory allocation. Last, it works only for FIFO channels.

The purpose of this paper is to remedy these drawbacks, while keeping its strength. Working along the line of [1], we propose an improved algorithm that requires $M' + 2(n-1)$ control messages in the worst case, but much fewer on the average; in the best case, it uses only $2(n-1)$ control messages, no matter how large M' is. The worst-case detection delay is $\Theta(n)$, as compared to the CV-algorithm's $\Theta(M')$; and the space complexity is $\Theta(n)$ for each process, a compile-time known quantity. The algorithm works equally well for both FIFO and Non-FIFO channels. Like the CV-algorithm, the proposed algorithm can "start" at a later time after the underlying system starts, and thus the M' here has a similar meaning as for the CV-algorithm. Strictly speaking, our algorithm always starts to operate when the first basic message is sent. But until its core procedure starts — and that is what we meant by "start" — no control message is involved and only little computation is required.

The theory behind our algorithm is developed in Section 2. The algorithm itself is described in the Section 3, where it is also proved to be correct. Section 4 analyzes its performance and compares it to the CV-algorithm.

1 Introduction

In a distributed system, a number of processes cooperate and communicate with one another by message-passing. Each Process switches between *active* and *idle* states in an unpredictable manner. Active processes send and receive arriving messages freely. An idle process can only receive messages, and on such an event it becomes active immediately. The system is *terminated* if and only if (i) all processes are idle and (ii) no message is in transit. The *termination detection problem* for a distributed system is to detect the system's termination, should it occur.

An algorithm that solves the termination detection problem often requires processes to exchange messages in addition to those used by the original distributed system. Such messages are called *control messages*, as distinguished from the *basic messages* or simply *messages* of the original system. The original system is the *underlying system* and its computation *underlying computation*.

In the literature, algorithms for termination detection are most often evaluated by the number of control messages required in the worst case. Chandy and Misra [3] established a lower bound on this worst-case message complexity: for any termination detection algorithm and any integer M, there exists a computation in which M basic messages are sent and at least M control messages are needed to correctly detect the system's termination. In other words, any termination detection algorithm requires $\Omega(M)$ control messages in the worst case, where M is the number of basic messages sent in the underlying computation. This lower bound can be more precisely written as $\Omega(M + n)$, where n is the number of processes in the system, since every process except possibly for the one who collects information needs to send at least one control message to inform other processes of its status. If a termination detection algorithm does not necessarily start its computation together with the underlying computation, then in the worst case as many as $\Omega(M + |E|)$ control message may be required [1], where $|E|$ is the number of channels in the system.

Many algorithms for termination detection have been proposed with various characteristics. Some algorithms work only for distributed systems with special properties such as diffusion (e.g., [5, 15, 17]), but most are for general purpose. The aforementioned lower bounds only apply to algorithms in the latter category. Some algorithms assume synchronous communications (e.g., [4, 6, 7, 13, 15, 18]), while others work as well for asynchronous ones [2, 8, 10, 11, 14, 16]. Some of the latter algorithms require FIFO channels [2, 16], and some don't [8, 10, 11, 14]. Most algorithms are intended for static systems only; algorithms for dynamic systems can be found in [4, 11].

A More Efficient Message-Optimal Algorithm for Distributed Termination Detection*

Ten-Hwang Lai, Yu-Chee Tseng, Xuefeng Dong

Department of Computer and Information Science
The Ohio State University
2036 Neil Ave.
Columbus, OH 43210

Tel: (614) 292-7036
E-mail: lai, tseng, dong@cis.ohio-state.edu

October 20, 1992

Abstract

Termination detection is a fundamental problem in distributed computing. Many algorithms have been proposed, but only the Chandrasekaran and Venkatesan (CV) algorithm [1] is known to be optimal in worst-case message complexity. This optimal algorithm, however, has several undesirable properties. First, it always requires $M' + 2 * |E| + n - 1$ control messages, whether it is worst case or best case, where M' is the number of basic messages issued by the underlying computation after the algorithm starts, $|E|$ is the number of channels in the system, and n is the number of processes. Second, its worst-case detection delay is $O(M')$. In a message-intensive computation, that might not be tolerable. Third, the maximum amount of space needed by each process is $O(M')$, a quantity not known at compile time, making it necessary to use the more expensive dynamic memory allocation. Last, it works only for FIFO channels.

The purpose of this paper is to remedy these drawbacks, while keeping its strength. We propose an algorithm that requires $M' + 2(n-1)$ control messages in the worst case, but much fewer on the average, and in the best case, it uses only $2(n-1)$ control messages, no matter how large M' is. The worst-case detection delay is $O(n)$, as compared to the CV-algorithm's $O(M')$; and the space complexity is $\Theta(n)$ for each process, a compile-time known quantity. The algorithm works equally well for both FIFO and Non-FIFO channels.

Index terms: Distributed systems, termination detection, algorithms, message complexity.

*This research was supported in part by National Science Foundation under Grant CCR-9010589.