[18] R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Trans. on Computer Systems*, Vol. 1, No. 3, Aug. 1983, pp. 222-238.

[19] G. Tel and F. Mattern, "The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes," *ACM Trans. on Programming Languages and Systems*, Vol. 15, No. 1, Jan. 1993, pp. 1-35.

[20] S. Venkatesan, "Reliable Protocols for Distributed Termination Detection," *IEEE Trans. on Reliability*, Vol. 38, No, 1, April 1989, pp. 103–110.

[21] L.-F. Wu, T.-H. Lai and Y.-C. Tseng, "Consensus and Termination Detection in the Presence of Faulty Processes," *Int'l Conf. on Parallel and Distributed Systems*, Taiwan, 1992, pp. 267-274.

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

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

[7] M. J. Fisher, N. A. Lynch and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, Vol. 32, April 1985, pp. 374-382.

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

[9] S.-T. Huang, "A Fully Distributed Termination Detection Scheme," *Information Processing Letter 29*, 1988, pp. 13-18.

[10] S.-T. Huang, "Detecting Termination of Distributed Computations by External Agents," *Int'l Conf. on Distributed Computing Systems*, 1989, pp. 79-84.

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

[12] T.-H. Lai, Y.-C. Tseng and X. Dong, "A More Efficient Message-Optimal Algorithm for Distributed Termination Detection," *Int'l Parallel Processing Symposium*, 1992, pp. 646-649.

[13] T.-H. Lai and L.-F. Wu, "An $(N-1)$-resilient Algorithm for Distributed Termination Detection," *4-th IEEE Symp. on Parallel and Distributed Processing*, 1992, pp. 274-281.

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

[15] F. Mattern, "Global Quiescence Detection Based on Credit Distribution and Recovery," *Information Processing Letters 30*, 1989, pp. 195-200.

[16] J. Misra and K. M. Chandy, "Termination Detection of Diffusing Computations in Communicating Sequential Processes," *ACM TOPLAS*, Vol. 4, No. 1, Jan. 1982, pp. 37-43.

[17] J. Misra, "Detecting Termination of Distributed Computations Using Markers," *ACM Symp. on Principles of Distributed Computing*, 1983, pp. 290-294.

Table 1: Comparison of our algorithm with LW and Ven algorithms. (M = number of basic messages issued, n = number of processes, E = number of channels, k = actual number of faults, and k' = pre-defined tolerable number of faults.)

	Msg. complexity	Overhead on basic msg.	Detection delays	Tolerance model
T	$O(M+kn+n)$	weight appending	$O(k+1)$	fault-reactive
LW	$O(M+kn+n)$	none	$O(n)$	fault-reactive
Ven	$O(k'M+E)$	msg. duplication	$O(M)$	fault-preventive

6 Concluding Remarks

The most common approach to make distributed systems or algorithms fault-tolerant is to make duplicated messages. Such a fault-preventive solution is usually expensive since costs are paid no matter faults will occur or not. In this paper, we have presented a fault-reactive solution for detecting termination in a distributed system in which processes tend to fail. To deal with the possibility of faults, only some bookkeeping computations are necessary. Extra message cost to make our algorithm fault-tolerant is only a linear factor with respect to the actual number of faults. The algorithm is pretty efficient in both message and time complexities as compared to existing algorithms in the literature.

The weight-throwing scheme is also used in solving the *garbage collection problem*. Recently, Tel and Mattern [19] have revealed in general the relationship between the termination detection problem and the garbage collection problem. It deserves further investigation with the presence of our flow-detection scheme.

References

[1] M. Ahuja, "*Flush* Primitives for Asynchronous Distributed Systems," *Information Processing Letters 34*, 1990, pp. 5-12.

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

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

[4] S. Cohen and D. Lehmann, "Dynamic Systems and Their Distributed Termination," *Proc. 1st ACM Symp. on Principles of Distributed Computing*, Aug. 1982, pp. 29-33.

Weight-throwing Overhead: The total number of $C(x)$'s issued is the times processes turning idle. Initially, all n processes are active. If M basic messages are issued, then processes will be activated at most M times. So processes can turn idle no more than $M+n$ times. The number of $C(x)$'s issued is at most $M+n$. The appended weights on basic messages also incur extra cost. However, it is bounded by $O(M)$.

Overall Message Overhead: As a result, the message complexity of our algorithm is $O(M+kn+n)$. It should be noted that failure is a rare event and k is usually a small number. In particular, when $k=0$, our algorithm issues at most $M+n$ control messages. It has been shown in [12] that if the underlying computation is started from any number of processes, then in the worst case, $\Omega(M+n)$ control messages are necessary to correctly detect termination. Our algorithm is thus message-optimal when $k=0$.

Detection Delay: The detection delay is defined to be the maximum number of hops that messages have to travel, after the system is terminated, for an algorithm to find out the terminated state. It reflects the communication delay of the algorithm. In our algorithm, when the last process turns idle, all control messages will take only one hop to reach their destinations. However, the delay of our algorithm depends on whether the leader process has the up-to-date W_H (the remaining weight in the system) or not. If so, the delay is only 1 hop. Otherwise, the leader process has to take a snapshot on the system. Each snapshot has delay of 2 hops. Since the system has stabilized, any snapshot that can successfully complete will work. Thus, the detection delay is bounded by $1+2\tilde{k}$, where $\tilde{k}$ is the number of snapshots that are (continuously) taken after the system is terminated and before the termination is successfully declared. Since $\tilde{k} \leq 2k$, the detection delay is bounded by $O(k+1)$. We emphasize that $\tilde{k}$ should be very small in reality. The detection delay is normally a constant.

Comparison: Table 1 compares our algorithm (denoted by T) with algorithms of Lai and Wu [13] and Venkatesan [20] (denoted by LW and Ven, respectively). It can be seen that our algorithm outperforms the Ven algorithm in both message and time complexities. As compared to the LW algorithm, our algorithm has the same message complexity as theirs, but needs to append weight on each basic message. However, our detection delay is shorter than theirs. Note that when $k=0$, LW [13] still has delay of $O(n)$ while ours is $O(1)$.

Our algorithm, which enjoys the shortest delay, should be most adequate for the high-speed networking environment, in which message length is usually a less important factor while fast revealing the terminated status of the system is more important.

some P_i on some subsystem $H' \subseteq H$ will obtain $sum_i = W_H = w_r$ if $P_r \in H'$, or $sum_i = 0$ if $P_r \notin H'$.

Now, consider time t and suppose the latest snapshot taken by P_r by t was on the subsystem $H' \supseteq H$. We can derive that (at time t):

$$
\begin{array}{ll}
 & W_H = w_r \\
\Longrightarrow & (W_H = w_r) \wedge (H = H' \vee H \subset H') \\
\Longrightarrow & (W_H = w_r \wedge H = H') \vee (W_H = w_r \wedge H \subset H') \\
\Longrightarrow & /* \text{ by Lemma 1, case (a) and (b) } */ \\
 & (W_H = w_r = sum_r) \vee (W_H = w_r \leq sum_r)
\end{array}
$$

If the left term in the above statement is true, then P_r will eventually execute **A5** and announce termination. If the right term is true, then either **A5** will be executed (when $W_H = sum_r$) or another snapshot is needed (when $W_H > sum_r$). The latter is guaranteed to happen because faults are detectable ($H' - H \neq \emptyset$). If P_r fails before it can successfully finish a snapshot, a new leader $P_{r'}$ will be elected eventually. If so, the snapshot will generate $sum_{r'} = 0$. With $w_{r'} = 0$, the proof follows.

Theorem 2 *After the underlying system is terminated, some process will eventually announce termination, unless all processes in the system are faulty.*

5 Performance Analysis

Snapshot Overhead: We first carefully consider the maximum number of snapshots that could be taken by our algorithm. Denote the *snapshot-taken history* in a computation as a sequence $((s_1,1),(s_1,2),\ldots,(s_2,1),(s_2,2),\ldots,(s_3,1),(s_3,2),\ldots)$, where each (s_i,j) denotes the j-th snapshot taken by process P_{s_i}. This sequence includes consistent as well as inconsistent snapshots. We claim that each (s_i,j), except $j=2$, must be *induced by* at least one distinct fault and thus there are at most $2k$ snapshots when the number of faults is k.

Define snapshot $(s_i,1)$ being induced by the failure of the previous leader $P_{s_{i-1}}$, except for $(s_1,1)$ which is defined to be induced by the most recent fault found by P_{s_1} before $(s_1,1)$ is issued. Although an inconsistent $(s_i,1)$ will cause another $(s_i,2)$ being taken, it can be observed that all faults happening before $(s_i,1)$ is issued must be known by P_{s_i} after $(s_i,1)$ (no matter consistent or not) is finished. Thus, we can define (s_i,j), $j \geq 3$, being induced by the most recent fault found by P_{s_i} before (s_i,j) is issued. Under this definition, each (s_i,j), $j \neq 2$, is induced by a distinct fault and we have proved the claim.

Each snapshot costs at most $n-1$ *Request*()'s and $n-1$ *Reply*()'s. The message overhead due to snapshots is bounded by $4kn$.

$$
\begin{array}{ll}
 & P_r \text{ announcing termination} \\
\Longrightarrow & w_r = sum_r \\
 & \geq W_H = \sum_{P_i \in H} w_i + \sum_{P_i,P_j \in H, B(x) \sqsubset C_{i,j}} x + \sum_{P_i,P_j \in H, C(x) \sqsubset C_{i,j}} x \\
 & \geq w_r \\
\Longrightarrow & \sum_{P_i \in H, i \neq r} w_i + \sum_{P_i,P_j \in H, B(x) \sqsubset C_{i,j}} x = 0 \\
\Longrightarrow & /* \text{ by } \mathbf{INV1} \text{ and } \mathbf{INV2} */ \\
 & (\forall P_i \in H, i \neq r :: P_i \text{ is idle}) \wedge (\forall P_i, P_j \in H :: C_{i,j} \text{ is empty})
\end{array}
$$

Combining the last statement with that P_r is idle, we assert that (a) is true.

To prove (b), observe that

$$
\begin{array}{ll}
\text{(b)} & \equiv \text{(there is no in-transit basic message between } H \text{ and } \overline{H}') \wedge \\
 & \qquad \text{(there is no in-transit basic message between } H \text{ and } H' - H).
\end{array}
$$

The first condition has been satisfied at the time the latest snapshot being taken (the channels between $\overline{H}'$ and H have been flushed or frozen then). To prove the second condition, any existence of a basic message $B(x)$ will establish a contradiction that $x + W_H > sum_r$. So (b) is true.

Theorem 1 *When any process announces termination, the underlying system has really terminated.*

Remark: The introduction of H' in the above proof has indicated that even the most recent snapshot includes more processes than the current healthy ones, the terminated state still can be detected correctly.

4.3 Liveness

Now we prove the liveness of our algorithm. We show that after the system terminates, eventually some process will announce termination, should not all processes be failed. Let H be the set of healthy processes at real time t such that at t both S is terminated and the last control message in the system is received. The weight in the subsystem H at time t is:

$$
W_H = \sum_{P_i \in H} w_i + \sum_{P_i,P_j \in H, B(x) \sqsubset C_{i,j}} x + \sum_{P_i,P_j \in H, C(x) \sqsubset C_{i,j}} x = \sum_{P_i \in H} w_i. \qquad (4)
$$

If termination is announced at this time, then we are done. Otherwise, some process will eventually announce termination as shown below.

According to **INV1**, Eq. (4) can be reduced to $W_H = w_r$, where $r = \min\{k | P_k$ is healthy at $t\}$. Because the system has already terminated, w_i, $IN_i[1..n]$, and $OUT_i[1..n]$ of any $P_i \in H$ must be fixed after t. Therefore, after t, a snapshot (if any) taken by

does not necessarily be equal to the real W_H from a *global view*, as there is no guarantee that each process takes snapshot at the same time. Much more misleadingly, there may exist no notion of W_H as there may be no instance in the real time that H being exactly the set of healthy processes. The following lemma establishes a more careful statement on the scenario and an important characterization of our snapshot result.

Lemma 1 *Let P_i take a snapshot on the system using* **F1** *and finish this snapshot at real time t with H being the set of processes from which P_i obtains responses. Also let $t' \geq t$ be any real time and H' be the subsystem consisting of all healthy processes at t'. Then, at time t', if P_i is healthy, the following is true:*
(a) $sum_i = W_{H'}$ if $H' = H$, and
(b) $sum_i \geq W_{H'}$ if $H' \subset H$.

Proof. This lemma is true because after time t all channels between H and $\overline{H}$ have been flushed or frozen. H remains isolated from the rest of the system and can not lose weight after t. H' must be a subset of H. So the obtained sum_i is always no less than the weight held in H'. □

We will use this lemma in the next subsection to show that no termination will be falsely reported.

4.2 Soundness

Now we prove the soundness of our algorithm. Suppose that at real time t a process P_r announces termination through **A5**. Let H be the set of healthy processes at time t. According to Definition 1, we have to prove that

(a) Each $P_i \in H$ is idle and there is no in-transit basic message between any two $P_i, P_j \in H$.

(b) There is no in-transit basic message between H and $\overline{H}$.

At time t, by **A5**, $sum_r = w_r$. However, sum_r was calculated from the latest snapshot[2] on the subsystem, say, $H' \supseteq H$. By Lemma 1, $sum_r \geq W_H$. So we can derive that:

[2]To make this statement sound, we assume that *right before* the system is started, an imaginary snapshot is taken on S.

remark is that the Eq. (2) describes the total weight in terms of real time, while a snapshot is taken on each process at different time. Formal correctness reasoning is given in the next section.

4 Correctness Proof

4.1 Invariants

The following three invariants are preserved by our weight-throwing and flow-detecting schemes.

INV1: *At any time, for any P_i, if $i \neq \min\{k | P_k$ is healthy$\}$, then*

$$P_i \text{ is idle } \Longleftrightarrow w_i = 0.$$

INV2: *For any message $B(x)$ or $C(x)$, the value $x > 0$.*

INV3: *For any healthy process P_i, at any moment,*

$$w_i + \sum_{j=1}^{n} OUT_i[j] = 1/n + \sum_{j=1}^{n} IN_i[j].$$

Invariant **INV3** only involves the local state of one process. We generalize it below to take into account of a subset of processes. Recall the definition of W_H, which is the total weight carried by all processes, in-transit basic messages, and in-transit control messages within the subsystem H. We can formulate this definition as:

$$W_H = \sum_{P_i \in H} w_i + \sum_{P_i, P_j \in H, B(x) \sqsubset C_{i,j}} x + \sum_{P_i, P_j \in H, C(x) \sqsubset C_{i,j}} x \qquad (3)$$

where "$M \sqsubset C_{i,j}$" denotes that M is an in-transit message in channel $C_{i,j}$. Note that in the above W_H is defined provided global views on the whole system are available. Assuming such availability, we can describe the value of W_H in terms of our program variables:

INV4: *Let H be any set of healthy processes. Then, at any time,*

$$W_H = \sum_{P_i \in H} (1/n + \sum_{P_j \in \overline{H}} (IN_i[j] - OUT_i[j])).$$

The above invariant can be easily verified by assuming the computation as a sequence of actions and then checking the equality after each execution of **A1**–**A4**. This is in fact the weight calculation formula used in *snapshot*() and **F3** by the leader process. Nevertheless, the sum_i obtained by a leader P_i from a *snapshot* on subsystem H

/* actions for detecting a fault when no snapshot is in progress */
F1: (P_i detecting P_j faulty) $\wedge$ ($P_j \notin F_i$) $\wedge$ ($SN_i = \emptyset$) $\rightarrow$
"flush or freeze" channel $C_{i,j}$; $F_i := F_i \cup \{P_j\}$;
$leader_i := \min\{k | P_k \in S - F_i\}$;
if ($leader_i = i$) **then call** *snapshot*();

/* actions for receiving a snapshot request */
F2: (P_i receiving $Request(F_j)$ from P_j) $\rightarrow$
$leader_i := j$;
$\forall P_f \in F_j - F_i$ :: "flush or freeze" channel $C_{i,f}$; $F_i := F_i \cup F_j$;
send a $Reply(F_i, IN_i[1..n], OUT_i[1..n])$ to P_j;

/* actions for receiving a snapshot response */
F3: (P_i receiving $Reply(F_j, IN_j[1..n], OUT_j[1..n])$ from P_j) $\rightarrow$
if ($F_i \neq F_j$) $\vee$ $\neg consistent_i$ **then**
$\forall P_f \in F_j - F_i$ :: "flush or freeze" channel $C_{i,f}$; $F_i := F_i \cup F_j$;
$consistent_i := false$;
else $temp_sum_i := temp_sum_i + 1/n + \sum_{P_f \in F_j}(IN_j[f] - OUT_j[f])$;
$SN_i := SN_i - \{P_j\}$;
if $SN_i = \emptyset$ **then**
if $consistent_i$ **then** $sum_i := temp_sum_i$;
else call *snapshot*();
end if;

/* actions for detecting a fault when a snapshot is in progress */
F4: (P_i detecting P_j faulty) $\wedge$ ($SN_i \neq \emptyset$) $\rightarrow$
"flush or freeze" the channel $C_{i,j}$; $F_i := F_i \cup \{P_j\}$;
$consistent_i := false$;
$SN_i := SN_i - \{P_j\}$;
if $SN_i = \emptyset$ **then call** *snapshot*();

Figure 3: The fault-tolerance actions.

Procedure *snapshot*(); /* assuming the caller is P_i */
begin
$SN_i := S - F_i - \{P_i\}$; /* defining the processes to send requests */
$\forall P_k \in SN_i$ :: send a $Request(F_i)$ to P_k;
$temp_sum_i := 1/n + \sum_{P_f \in F_i}(IN_i[f] - OUT_i[f])$;
$consistent_i := true$;
end;

Figure 4: The snapshot-taking procedure.

A1: (P_i sending a basic message B to P_j) $\rightarrow$
partition w_i into x and y such that $x > 0$, $y > 0$ and $x + y = w_i$;
send $B(x)$ to P_j;
$OUT_i[j] := OUT_i[j] + x$;
$w_i := y$;

A2: (P_i receiving a basic message $B(x)$ from P_j) $\rightarrow$
$IN_i[j] := IN_i[j] + x$;
$w_i := w_i + x$;
pass the basic message B to the underlying system;

A3: (P_i becoming idle) $\rightarrow$
if $leader_i \neq i$ **then**
send $C(w_i)$ to P_{leader_i};
$OUT_i[leader_i] := OUT_i[leader_i] + w_i$;
$w_i := 0$;
end if;

A4: (P_i receiving a $C(x)$ from P_j) $\rightarrow$
$IN_i[j] := IN_i[j] + x$;
$w_i := w_i + x$;

A5: (P_i is idle) $\wedge$ ($sum_i = w_i$) $\rightarrow$
announce "TERMINATION";

Figure 2: The weight-throwing and flow-detecting actions.

- $Reply(F_i, IN_i, OUT_i)$ — the state-reporting message sent by P_i. This is to reply the leader's $Request()$ message.

The algorithm consists of 9 event-driven atomic actions, each of the format "(guard) $\rightarrow$ (actions)". Actions are triggered by sending/receiving messages, changing local states, or detecting new faulty processes. The first set of actions **A1**–**A5** (shown in Fig. 2) implements the weight-throwing and the flow-detecting schemes. The second set of actions **F1**–**F4** (shown in Fig. 3) is to deal with faults and take snapshots on the system. These actions will call a procedure *snapshot*() (shown in Fig. 4). We describe the actions for process P_i. Throughout, $i \neq j$ is assumed.

Action **A1** is activated when P_i wants to send a basic message to another process. **A2** is triggered by receiving a basic message. **A3** is the weight-reporting action executed when P_i does not consider itself as the leader. **A4** describes P_i's response on receiving a control message. In all **A1**–**A4**, the weight-flowing history is recorded in the corresponding arrays $IN_i[1..n]$ and $OUT_i[1..n]$. Action **A5** is where P_i, as a leader, announces termination. It is executed when P_i is idle and has accumulated all weight in the system.

Action **F1** is activated on P_i detecting other process P_j to be faulty for the first time. Also, no snapshot is currently in progress (ensured by $SN_i = \emptyset$). Channel $C_{i,j}$ is flushed or frozen. Then P_i elects the healthy process with the least id as the leader (it does not hurt if the elected process does not know this fact or even is already dead but not being known to P_i yet). If P_i finds itself to be the leader, then it invokes procedure *snapshot*() to initiate a snapshot.

In procedure *snapshot*(), SN_i is first set to the set of processes to which $Request()$'s are to be sent. Note that this prevents **F1** from being executed until this snapshot finishes. Assuming the current healthy set is $S - F_i$ and this snapshot will be consistent, we implement part of Eq. (2) in variable $temp_sum_i$. More weight is to be added to $temp_sum_i$ subsequently. Action **F2** describes P_i's response on receiving such a request.

Then P_i (the snapshot initiator) waits for each process in SN_i on the event of either a reply arriving (**F3**) or the process being dead (**F4**). If **F3** happens, the consistency of the snapshot is checked (the condition $\neg consistent_i$ indicates that some earlier events have destroyed the consistency). A consistent response will update $temp_sum_i$ (using Eq. (2)). Then the barrier (SN_i) is reduced by one. When all barriers are removed, we, depending on the consistency, either start another snapshot or assign the new total weight to sum_i. If **F4** happens, this snapshot must be inconsistent. Again, another snapshot can only be started when $SN_i = \emptyset$.

Such a procedure is repeated until a consistent snapshot is obtained. Since a fail-stop model is assumed, the number of healthy processes is a non-increasing function (with respect to time) and so eventually the process will terminate. One interesting

system. Snapshot requests are sent by the leader process on two occasions: (1) when it becomes leader for the first time and (2) when a new fault is detected. During taking a snapshot, processes exchange their the flow-in and flow-out records. They also exchange the set of faulty processes known to them to check the consistency of the snapshot. Snapshots might be repeatedly taken until a consistent one is obtained. However, note that no two snapshots will be in progress at the same time.

The data structures kept by P_i, $i = 1..n$, are defined as follows.

- $leader_i$ — id of the current leader known to P_i. Initially, $P_i = 1$.
- w_i — a real number, which contains the weight currently held by P_i. Initially, $w_i = 1/n$.
- sum_i — a real number, which is the system's total weight assumed by P_i. It is only used when P_i is a leader. Initially, $sum_i = 1$.
- $IN_i[1..n]$ — an array of real numbers. $IN_i[j]$ records the total weight that has been thrown from P_j to P_i. Initially, $IN_i[j] = 0$ for all $j = 1..n$.
- $OUT_i[1..n]$ — an array of real numbers. $OUT_i[j]$ records the total weight that has been thrown to P_j from P_i. Initially, $OUT_i[j] = 0$ for all $j = 1..n$.
- F_i — a set of processes. A process $P_j \in F_i$ iff P_i knows P_j to be faulty and P_i has "flushed or frozen" the channels connected to P_j. Initially, assume $F_i = \emptyset$.
- SN_i — a set of processes. When P_i initiates a snapshot, SN_i is set to the set of processes to which snapshot requests are sent. A $P_j \in SN_i$ is removed from the SN_i under the following possible events: (1) P_j replies the request and (2) P_j is found to be faulty by P_i. Note that SN_i performs like a *barrier*. Before SN_i becomes an empty set, no second snapshot will be started. Initially, $SN_i = \emptyset$ and we use this to indicate that no snapshot is in progress.
- $temp_sum_i$ — a real number, for temporarily calculating the total remaining weight while a snapshot is in progress.
- $consistent_i$ — a boolean, for recording the consistency of a snapshot.

There are 4 types of messages that will be sent in the system:

- $B(x)$ — a basic message B appended with weight x.
- $C(x)$ — a control message that is used to report weight x to the leader.
- $Request(F_i)$ — a snapshot-requesting message sent by the current leader P_i. Set F_i informs the receiver the set of faulty processes known to P_i.

2.3 Flushing/Freezing Channels and Taking Snapshots

We assume that the underlying network provides the service of flushing or freezing a channel connected to a faulty process. The purpose of flushing or freezing channels between H and $\overline{H}$ is to cease any further interaction between H and $\overline{H}$, so as to find out a stable value W_H. Let us consider the channel $C_{i,j}$ between a $P_i \in H$ and a $P_j \in \overline{H}$ (we assume channels are bi-directional). Observe Fig. 1. Messages directed to P_j will be discarded by the network and weights carried by them will be lost. For messages directed to P_i, we assume that P_i can send a special *flush* message to the faulty P_j. This message, on arriving P_j, will bounce back to P_i and, on its way back, will pump out all messages directed to P_i. We note that such a mechanism is called a *return flush* in [13]. A similar assumption (but is for FIFO channels) is also used in [20]. *Flush* has been proposed as a network primitive [1]. If the flush mechanism is not available, we assume that P_i can freeze this channel $C_{i,j}$. A frozen channel is disconnected and all its contents are discarded. In our algorithm, we will simply say "flush or freeze channel $C_{i,j}$" to refer to the above operations. After $C_{i,j}$ is flushed or frozen, P_i can not use this channel any more. Most networks satisfy this assumption.

Due to the lack of global clock, the global views on W_H have to be obtained by taking snapshots on H. We define a snapshot on H to be *consistent* iff every $P_i \in H$ sees the same set of faulty processes $\overline{H}$ and has "flushed or frozen" its channels connected to $\overline{H}$. A consistent snapshot is not hard to obtain because the fail-stop behavior is a *stable* property, in the sense that a process, once failing, remains so for good. This is what makes a global view possible. In our algorithm, in the presence of faults, the leader sends a snapshot request to each process in H. On receiving such a request, a process "flushes or freezes" its channels connected to $\overline{H}$ and then reports its in_i and out_i to the leader. The values of in_i and out_i will remain unchanged since then, unless the definition of H changes. The leader then uses these values to calculate W_H. It is worth noting that the above obtained W_H may not be up-to-date because more faults may occur during the snapshot. However, this W_H is always greater than or equal to the current $W_{H'}$, where H' is the current healthy set. This eliminates the possibility of reporting fake termination. Also, if fortunately $W_H = W_{H'}$, then the algorithm will still work correctly.

3 Fault-tolerant Termination Detection Algorithm

The algorithm incorporates the above weight-throwing and flow-detecting schemes and a snapshot-taking scheme. Leaders are elected according to their indices. We let process P_i elect itself as a leader if it knows all $P_j, j < i$, to be faulty. The leader process is responsible for taking snapshots to calculate the remaining weight in the

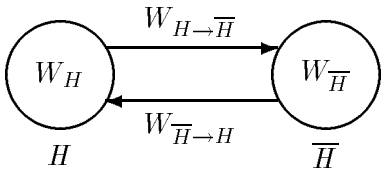

Figure 1: Conceptual weight distribution diagram between subsystems H and $\overline{H}$.

To implement this invariant in our algorithm, let us impose two variables, in_i and out_i, on each $P_i \in H$. Initially, $in_i = out_i = 0$. On the top of the weight-throwing scheme, we add the following flow-detecting rules:

(i) Whenever $P_i \in H$ receives a (basic or control) message with weight x from another $P_j \notin H$, add x to in_i.

(ii) Whenever $P_i \in H$ sends a (basic or control) message with weight x to $P_j \notin H$, add x to out_i.

Now suppose *global views* on the subsystem H is available and thus we can define the *weight within* H (denoted by W_H) at some point of time to be the sum of weights held by each $P_i \in H$ and by each in-transit message sending from a $P_i \in H$ to another $P_j \in H$. It can be observed that at any moment the following equality holds:

$$W_H = \sum_{P_i \in H} (in_i - out_i + 1/n), \tag{2}$$

where the first and second terms are weights flowing respectively into and out of H and the last term is the initial weight held by each P_i. Clearly, in a fault-free system, $W_S = 1$.

Next, we address some problems that have to be taken care of in the presence of faults. Suppose H is the set of all healthy processes in the system. Denote the faulty set as $\overline{H} = S - H$. Let us consider the interactions between H and $\overline{H}$. Weights, according to their distribution, can be partitioned into 4 portions: (a) W_H, (b) $W_{\overline{H}}$, (c) $W_{H\to\overline{H}}$ (the weights holding by in-transit messages sent from H to $\overline{H}$) and (d) $W_{\overline{H}\to H}$ (the weights holding by in-transit messages sent from $\overline{H}$ to H). Such a scenario is depicted in Fig. 1. Messages carried by portion $W_{H\to\overline{H}}$ will automatically be discarded by the network. If any of portions $W_{\overline{H}}$ and $W_{H\to\overline{H}}$ is nonzero, then the leader can never accumulate the sufficient weight of 1. Moreover, the leader may belong to $\overline{H}$.

To solve these problems, in our algorithm, we shall "flush or freeze" the channels between H and $\overline{H}$, take a *snapshot* on the subsystem H, use Eq. (2) to find out W_H, and let the leader collect the new weight W_H. We will also re-elect a new leader from H if necessary. The detailed algorithm is presented in Section 3. In the following, we address issues concerning flushing or freezing channels and obtaining snapshots.

The weight-throwing scheme works as follows. Whenever an active process P_i wants to send a basic message B to another process P_j, P_i performs: (1) partitioning w_i into two positive reals x and y such that $x + y = w_i$, (2) sending $B(x)$ to P_j, where $B(x)$ denotes the basic message B appended with the number x, and (3) letting $w_i = y$. Accordingly, on P_j receiving $B(x)$, P_j retrieves the appended weight x and combines x to its current weight, i.e., assigning $w_j + x$ to w_j.

Whenever a process $P_i, i \neq 1$, switches from active to idle, it sends a weight-reporting message $C(w_i)$ to the leader P_1 and sets its w_i to 0. Accordingly, P_1, on receiving $C(w_i)$, combines w_i to its current weight. The above rules preserve two important invariants:

- At any moment, every active process holds a non-zero weight. Every in-transit basic/control message also carries a non-zero weight.
- At any moment, the sum of weights held by all active processes, in-transit basic messages and in-transit control messages is 1.

The weights have jointly represented the process idleness (i.e., a non-leader P_i is idle iff $w_i = 0$) and the emptiness of channels (i.e., $\sum_{i=1..n} w_i = 1$ iff all channels are empty) at the same time. Thus, when P_1 becomes idle and has accumulated all the weights in the system (i.e., $w_1 = 1$), it can declare termination.

We have described the weight-throwing scheme in an abstract manner. Note that weights must be expressed precisely so as to guarantee the correct detection of termination. Using floating point values is not appropriate due to possible rounding errors. This problem can be easily solved by using a pair of integers to represent the weights (e.g., the initial weight of $1/n$ is $[1, n]$). This may, theoretically, lead to another problem of very long messages. The reader is requested to refer to [10, 15], where several variants are discussed.

2.2 Flow-detecting Scheme

In a faulty system, weights might be lost due to faulty processes holding weights at the time becoming dead or due to undeliverable messages carrying weights. Hence the leader process (if still alive) may never accumulate sufficient weight. To solve this problem, we introduce a *flow-invariant* concept below.

Consider any subsystem H which is a subset of S. (By "subsystem" we mean all processes in H and all channels connecting two processes in H.) Let I be any time interval during which the system is doing computation. The flow-invariant concept can be stated with the following equality:

$$\begin{aligned} & \text{(the weight change in } H \text{ during time interval } I) \\ = \; & \text{(weights flowing into } H \text{ during } I) - \text{(weights flowing out of } H \text{ during } I). \end{aligned} \tag{1}$$

(or Byzantine-fault) model is assumed, one famous *impossibility* result [7] — that in an asynchronous network, a set of processes can not reach *consensus* even in the presence of one faulty process — has refused us from developing a fault-tolerant termination detection algorithm [20]. In fact, recently it has been proved that the termination detection is (properly) more difficult than the consensus problem [21]. Fortunately, even with these strong negative results, designing fail-stop processors using prone-to-be-malicious processors is still possible [18], if the number of faults is limited.

The weight-throwing scheme and the flow-detecting scheme are introduced in Section 2. The complete algorithm is presented in Section 3. Correctness reasoning is given in Section 4 and performance analysis is in Section 5. Concluding remarks are in Section 6.

2 Basic Idea

Let $S = \{P_1, P_2, \ldots, P_n\}$ be the set of distributed processes. In the beginning of a computation, all processes are assumed to be in the active state. We re-define the terminated state of a fault-tolerant underlying system as follows:

Definition 1 A distributed system is said to be *terminated* iff (1) each healthy P_i is idle, (2) there is no in-transit message in the channel between any two healthy processes, and (3) there is no in-transit message in the channel between any faulty process and healthy process.

Note that the above definition is independent of the states of faulty processes and undeliverable messages (which will be automatically discarded by the underlying networks). It is also consistent with the conventional definition if no fault occurs. To detect such a state, in the following we first summarize a termination detection algorithm called *weight-throwing scheme* reported in [10, 15], which is vulnerable to faults. Then, we develop on the top of this scheme a *flow-detecting scheme* to derive a fault-tolerant termination detection algorithm.

2.1 Weight-throwing Scheme

For now, we assume that S is fault-free. Each P_i keeps a real variable w_i, which indicates the *weight* that P_i is holding at hand. Initially, each process is active[1] and $w_i = 1/n$. So the total weight in the system is 1. One special process in S (say P_1) is designated as the *leader* of the system. The leader is responsible for collecting weights in the system and on accumulating all weights in the system, declaring termination.

[1] In [10], the underlying system is assumed to start from P_1. So $w_1 = 1$ and $w_i = 0$, $i = 2..n$. Here we assume all processes are active initially. This generalization is straight-forward.

and detection delay of $O(n)$, where k is the *actual* number of faults that occur in the computation. This is a *fault-reactive* algorithm since the extra message costs due to faults are proportional to the number of faults that really occur. Furthermore, it remains message-optimal when no fault occurs (i.e., $k = 0$). Finally, we note that both these algorithms are modified from their fault-intolerant precedents [2] and [12], respectively.

One interesting question is: whether there is a fault-tolerant version for the algorithm in [10, 15].

This paper is motivated by the elegant message-optimal but fault-intolerant termination detection algorithm proposed in [10, 15] called *weight-throwing scheme* (or called *credit distribution* by the latter). We develop on the top of this scheme a new *flow-detecting scheme* and propose a fault-tolerant termination detection algorithm that has comparable message complexity to the existing algorithms and better detection delay. This algorithm shares the elegance of [10, 15] and has the following characteristics: (1) the message complexity is $O(M + kn + n)$, where k is as defined above, (2) the worst-case detection delay is $O(k + 1)$ and the worst case only happens in the rare event that a sequence of k processes continuously fail *right after* the system terminates and *right before* any of them were able to successfully claim termination detected, and (3) the extra computational overhead to handle the possibility of faults is very light. It is also a fault-reactive algorithm. In particular, when no fault occurs ($k = 0$), the algorithm sends at most $M + n$ control messages and remains message-optimal (we assume that computations are started from all processes and thus the lower bound of $\Omega(M + n)$ mentioned above applies). We emphasize that the detection delay should normally be a constant as faults are usually improbable in reality.

The environment assumed by this paper is an asynchronous network. Communications are reliable and NON-FIFO. The system is static, but an arbitrary number of processes may fail. Processes are fully connected and thus the network always remains connected in the presence of faults. A fail-stop model [18] is assumed and thus a failed process can not rejoin the computation in the current session. Faults are detectable within a finite amount of time. Undeliverable messages (those directed to faulty processes) are discarded by the networks. We assume that channels between faulty and healthy processes can be "flushed or frozen." This "channels are flushable" assumption is made in both [13, 20]. If this function is not provided by the network, we assume freezing a channel is possible, by which this frozen channel is disconnected and all its contents are discarded. After a channel is flushed or frozen, it is not usable any more.

It is worth noting that the assumption of faults being detectable is a necessary condition to solve the fault-tolerant termination detection problem, as proved in [21]. It is also worth noting that the assumption of fail-stop model is crucial. If a malicious-fault

1 Introduction

In a distributed system, a set of processes cooperate and communicate with one another by message-passing. Each process switches between *active* and *idle* states. An active process is free to send/receive messages and may become idle spontaneously, but an idle process can only receive messages, on which event it becomes active immediately. For clarity, sometimes the above messages are called *basic messages* and the distributed system is called the *underlying system*. Conventionally, in the fault-free case, the underlying system is said to be *terminated* iff (i) all its processes are idle and (ii) no message is in transit. A termination detection algorithm is to find out such a global quiescent state, should it occur. For this detection purpose, extra messages, or explicitly *control messages*, might be sent by the algorithm.

Many termination detection algorithms have been proposed (e.g., for diffusion computations [5, 16], for synchronous communications [6, 8], for asynchronous but FIFO communications [2, 17], for asynchronous NON-FIFO communications [9, 10, 11, 12, 14, 15], and for dynamic systems [4, 11]). The worst-case message complexity for this problem is $\Omega(M)$ (if the underlying system is started from one process) [3], $\Omega(M+n)$ (if the underlying system can be started from any number of processes) [12], or $\Omega(M+E)$ (if the detection algorithm is launched later than the underlying system) [2], where n is the number of processes, E is the number of channels in the system, and M is the number of basic messages issued. All these bounds have been achieved respectively by [10, 15], [12] and [2]. Another performance measure is the *detection delay*, which is defined to be "the length of the longest possible communication path along which control messages need to be sequentially passed *after* the underlying system has terminated but *before* the termination is detected." The above four message-optimal algorithms have detection delays of $O(1)$ ([10, 15]), $O(n)$ ([12]), and $O(M)$ ([2]), respectively. But the first two need to append extra information to basic messages, while the rest two do not.

All above algorithms are for fault-free distributed systems. In comparison, less emphasis has been given to the problem of detecting termination in a distributed system in which processes tend to fail. As one of the merits of using distributed computing is its fault-tolerant capability, it is desirable that a termination detection algorithm work correctly on an underlying system which can tolerate faults. Recently, this problem has started to receive attention. Venkatesan [20] first proposed such an algorithm. This algorithm has detection delay of $O(M)$ and, to tolerate k' faults, incurs message complexity of $O(k'M + E)$, where k' is the *predefined* fault-tolerant level. This is a *fault-preventive* algorithm since the factor k' (which is in fact incurred from the cost of duplicating messages) is always added no matter whether or not faults occur. Lai and Wu [13] presented an algorithm which has message complexity of $O(M + kn + n)$

Detecting Termination by Weight-throwing in a Faulty Distributed System

Yu-Chee Tseng

Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210
Tel: 614-292-5813, Fax: 614-292-9021
Email: tseng@cis.ohio-state.edu

Abstract

This paper presents a fault-tolerant termination detection algorithm for a distributed system in which processes tend to fail. Allowing arbitrary number of processes to have fail-stop behavior, the algorithm can detect termination efficiently with $O(M + kn + n)$ control messages and $O(k + 1)$ detection delays, where M is the number of basic messages issued, n is the number of processes, and k is the actual number of processes that fail. This algorithm has less detection delays than existing algorithms in the literature and comparable performance in terms of message complexity. In particular, when no fault occurs, the algorithm has constant detection delay and it uses, in the worst case, optimal number of messages.

Key Words: distributed system, termination detection, fault-tolerance, reliability, fail-stop processor, weight-throwing.