Effects of Message Loss on Distributed Termination*

Richard Koo**
Sam Toueg†

87-823

March 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501

---

*The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

**This author was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 5378, Contract MDA903-85-C-0124.

†This author was supported by the National Science Foundation under grant DCR-8601864.

Effects of Message Loss on Distributed Termination*

Richard Koo**
Sam Toueg†

Department of Computer Science
Cornell University
Ithaca, New York 14853

March 21, 1987

Abstract

We study the problem of termination in distributed systems with faulty communication channels. We show that for asynchronous systems, protocols that guarantee knowledge gain via message transfers cannot be guaranteed to terminate even if we assume that only transient communication failures can occur, and want to achieve only a weak kind of termination. The same result holds for synchronous systems as well.
Keywords: distributed systems, fault-tolerance, distributed termination, communication failures.

---

*The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

**This author was supported by the Defense Advanced Research Projects Agency (DoD) under ARPA order 5378, Contract MDA903-85-C-0124.

†This author was supported by the National Science Foundation under grant DCR-8601864.

1 Introduction

In the design of many fault-tolerant distributed protocols, processors are assumed to be the only faulty components; communication channels between processors are assumed to be failure-free. This is particularly evident in the large amount of literature devoted to distributed agreement protocols [Fis83]. This unequal treatment of processor and communication failures has been justified on the grounds that any failures a channel exhibits can be attributed to either one of the two processors the channel connects. This way of modeling channel failures is not satisfactory. Hadzilacos [Had86] wrote:

> ... we prefer to consider a component faulty only if *it* misbehaves, not if other components related to it misbehave. Moreover, in BA [Byzantine Agreement], pronouncing a processor correct or faulty is not merely a question of accounting for faults: a processor that, ... , is faulty is exempted from the requirement to decide on a value subject to the Agreement and Validity conditions.

In this paper, we study the problem of designing protocols that tolerate failures of communication channels. We consider those protocols that guarantee knowledge gain between processors via message transfers. (We call these *non-trivial* protocols.) We show that for asynchronous systems, non-trivial protocols cannot be guaranteed to terminate even if we assume that only transient communication failures can occur, and we only want to achieve a weak kind of termination. The same result also holds for synchronous systems.

Informally, a protocol *weakly terminates* if from every point of its execution, the execution can be continued to a point at which all processors stop. Channel failures are *transient* if any message sent repeatedly is eventually received. Transient channel failure is the model usually adopted by protocol designers to account for the faulty behavior of communication links in long-haul and local computer networks, e.g., ARPANET and Ethernet.

Several previous results are related to the problem of termination in the presence of transient channel failures. With *permanent* channels failures, it is well-known that it is impossible to achieve common knowledge (or "coordinated attack" [Gra78]), or even eventual common knowledge [HM85b]. In contrast to these results, we consider transient channel failures, we do not restrict our study to protocols that achieve either common knowledge or eventual common knowledge, and we concentrate on the problem of termination. These differences are underscored by the fact that with *transient* channel failures, there *is* a protocol for achieving eventual common knowledge. However, this protocol is non-terminating. (See Appendix A.)

In his study of *commit protocols*, Skeen [Ske82] showed that processors may have to block forever, neither committing or aborting, (i.e., the commit protocol may not terminate), if the communication network is permanently partitioned. In contrast to this result, we show that even if only transient channel failures may occur (and hence, even if the network is not partitioned permanently), *any* non-trivial protocol (including commit protocols) cannot be guaranteed to terminate. Therefore, our result complements and/or generalizes previous impossibility results [Gra78,HM85b,Ske82].

In this paper, we prove our result only for asynchronous systems. A similar proof can show that it also holds for synchronous systems. The paper is organized as follows: a formal model of asynchronous system is in Section 2. We present the result in Section 3. Section 4 contains the discussion.

2 Model of an asynchronous distributed system

A distributed system consists of m processors that communicate by messages via communication channels. Each processor is a deterministic state machine, which may have an infinite number of states. In each state, a processor can execute zero or more atomic actions. The states in which a processor cannot execute any atomic actions are called *terminal* states. Executions of atomic actions are called *events*. Any event may cause a processor to change its state. Two possible events of a processor i are:

1. send_i(*processor, message*), which i executes to send *message* to another *processor*; and

2. $\mathsf{receive}_i$(*processor, message*), which i executes to receive *message* from another *processor*.

2.1 Runs of processors

Let $state_{i,0}$ be an initial state of processor i, and s_i be a sequence of events of i. The pair $h_i = (state_{i,0}, s_i)$ is a *local history* of i if s_i is a sequence of events that i executes beginning at state $state_{i,0}$. Let $h_i.state$ denote $state_{i,0}$, and $h_i.events$ denote s_i. A local history h_i is *finite*, if $h_i.events$ is finite.

A m-tuple of local histories, $hist = \{h_1, h_2, \ldots, h_m\}$, is a *system history* if

1. $\forall i : 1 \leq i \leq m$, h_i is a local history of processor i; and
2. $h_i.events$ contains $\mathsf{receive}_i(j, msg)$, only if $h_j.events$ contains $\mathsf{send}_j(i, msg)$.

If all h_i's are finite local histories, $hist$ is a finite system history. Since all processors are deterministic, any finite system history unambiguously specifies the state of each processor.

An *asynchronous run* r is a pair ($hist$, $mesg$) such that $hist$ is a finite system history, and $mesg$ is a subset of the set of messages that are not yet delivered; i.e., those messages that are sent and not received in $hist$.[1] The two components of r are denoted by $r.hist$ and $r.mesg$, respectively. Messages that are sent and not received in $hist$, and also are not in $mesg$ are *lost*.

To model message losses, we introduce the following notation: $r' = failure(r)$, if runs r' and r are identical except that some messages that are not yet delivered in r are lost in r'. To be more precise, let $r = (hist$, $mesg)$ and $r' = (hist'$, $mesg')$. If $hist' = hist$ and $mesg' \subset mesg$, then $r' = failure(r)$. (See Figure 1.)

An *asynchronous system* is defined as the set of all asynchronous runs. Each run corresponds to a possible state of the system. Changes of system states are modeled by *continuations* of runs. A run r' is a continuation of a run r, if r' and r meet the following conditions: Let h_i be $proj_i(r.hist)$, and h'_i be $proj_i(r'.hist)$.

[1] A lock-step synchronous run (***hist***, ***mesg***) [DDS87] must meet the following additional requirements. For all processors i and j, the numbers of events that i and j have respectively in ***hist*** differ by at most one; and for all $msg \in mesg$, if the sender of msg is i, then the sending of msg is i's last event in ***hist***.

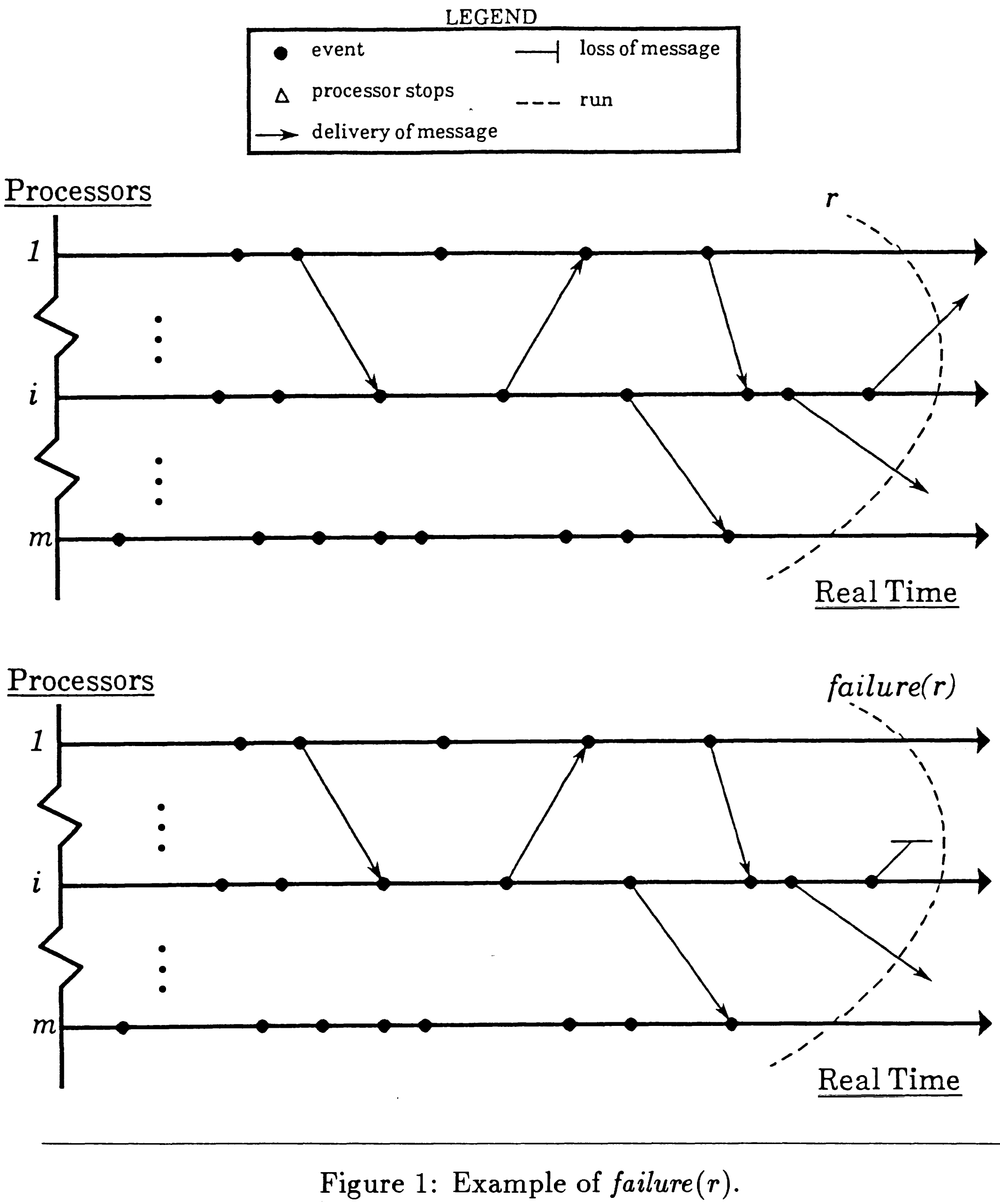

Figure 1: Example of *failure*(r).

1. $\forall i : 1 \leq i \leq m$, $h'_i.state = h_i.state$, and $h_i.events$ is a prefix of $h'_i.events$;

2. for all $\mathsf{receive}_i(j, msg)$ that are in r' and not in r, either $msg \in r.mesg$, or the event $\mathsf{send}_j(i, msg)$ is in r' but not in r; and

3. for all messages $msg \in r'.mesg$, either $msg \in r.mesg$, or the sending of msg is an event in r' but not in r.

We say $r = \mathit{prefix}(r')$, if and only if r' is a continuation of r.

We now describe the channel failure model. We assume that all channels exhibit only transient omission failures. If copies of a message are sent repeatedly over a channel, at least one copy will eventually be received. However, there is no bound on the number of messages that may be lost. To simplify our discussion of lost messages, we assume that every copy of a message is unique. From this assumption and the definitions of runs and of continuations, a message lost in a run is also lost in all of its continuations.

Events are partially ordered by the *after* relation [Lam78]. Event e' is after event e in run r if and only if

1. e and e' are events of the same processor, and a prefix of r contains e but not e';

2. e is the sending of a message msg and e' is the receipt of msg; or

3. there is an event e'' such that e' is after e'', and e'' is after e.

If e' is after e, then we say that e is *before* e'.

2.2 Distributed Protocols

A *local protocol* P_i of a processor i is a function from i's current state and the sequence of messages that i has received to the next atomic action to be executed by i. (Our results can be generalized to allow non-deterministic protocols.) A *distributed protocol* is a n-tuple $P = \{P_1, P_2, \cdots, P_n\}$ such that for all $1 \leq i \leq n$, P_i is a local protocol of processor i. A system history $\{h_1, \cdots, h_n\}$ is *consistent* with P if for all $1 \leq i \leq n$, the sequence of events in h_i corresponds to an execution of local protocol P_i beginning from the initial state of h_i. A run r is consistent with protocol P if $r.hist$ is consistent with P. If r is consistent with P, so do *prefix*(r) and *failure*(r). For convenience, protocol P can be identified with the set of asynchronous runs that are consistent with it.

The termination property of a protocol P is characterized by the runs in P. A *terminating* run is one in which all processors enter terminal states. A *weakly terminating* run is one that has at least one terminating continuation in P. Runs that are not weakly terminating are *non-terminating*. Examples of non-terminating runs are runs in which the system is deadlocked, and runs in which one processor is in an infinite loop. Protocol P is *weakly terminating*, if all its runs are weakly terminating; it is *non-terminating* otherwise.

3 Problem of Protocol Termination

In this section, we prove that any protocol that guarantees knowledge transfer despite transient communication failures is non-terminating. Roughly, our argument goes as follows. First, given any weakly terminating protocol and *any* initial system state, we show that starting from this state, this protocol must have a run that terminates *without* message transfers. Then, we note that without message transfers, knowledge cannot be transferred in asynchronous systems [CM86]. Hence, in asynchronous systems with transient communication failures, weakly terminating protocols cannot guarantee knowledge transfer.

3.1 Termination without Message Exchange

An event is a *last* receive event in a run if it is not before another receive event in this run. A run r is an *initial* run if its contains no events; it includes only the initial states of the processors.

Theorem 1 *Let P be a weakly terminating protocol. For all initial runs r in P, r has a terminating continuation in P in which no processor receives any messages.*

Proof: By contradiction. Let r be an initial run in P such that every terminating continuation of r in P contains at least n ($n > 0$) receive events. Let $r' = (hist', mesg')$ be a terminating continuation of r in P that contains exactly n receive events. Let e be a last receive event in r', and suppose that e occurs at processor i.

Delete e and all events that are after e from $hist'$. It is easy to see that this results in a system history $hist''$ (See Figure 2). Let run $r'' = (hist'', \phi)$. Note that $r'' = failure(prefix(r'))$. Since r' satisfies P, r'' must also satisfy P; thus, r'' is also in P. Moreover, r'' is a run that has only $n-1$ receive events.

Since e is a last receive event of r', all events that are deleted from r' in the construction of r'' occur only at processor i. The histories of all processors except i in r' and r'' are the same. Since r' is a terminating run, i is the only processor in r'' that has not terminated. By construction, the channels of r'' are empty. Hence, in any continuation of r'', i receives the same number of messages as it does in r'', namely $n-1$. Since P is weakly terminating, r'' has a terminating continuation r^* in P. However, run r^* contradicts the minimality of n. □

3.2 Processors' Knowledge

3.2.1 Syntax

We adopt the notation used by Halpern and Moses [HM85a] to describe the knowledge of processors. Let Φ be a set of primitive propositions $\{p_1, p_2, \ldots\}$. The language $\mathcal{L}(\Phi)$ is the smallest set of formulas containing Φ, closed under $\neg$, $\wedge$, and modal operators $K_1, K_2, \ldots,$ and K_m. Formulas of the form $p \vee q$ are abbreviations for $\neg(\neg p \wedge \neg q)$, $p \supset q$ are for $\neg(p \wedge \neg q)$.

3.2.2 Semantics

For all processors $\forall i : 1 \leq i \leq m$, i's view of a run r is the projection of $r.hist$ $proj_i(r.hist)$, on i. The views of processor i divide runs of a protocol into equivalence classes. Runs r and r' are in the same equivalence class (with respect to i), if and only if $proj_i(r.hist) = proj_i(r'.hist)$. The equivalence class of r according to i, denoted by $poss_i(r)$, determines what i can know at the end of r.

A processor's knowledge at the end of a run is defined inductively. Let P be a protocol, and π be a function mapping from Φ to the set of subsets

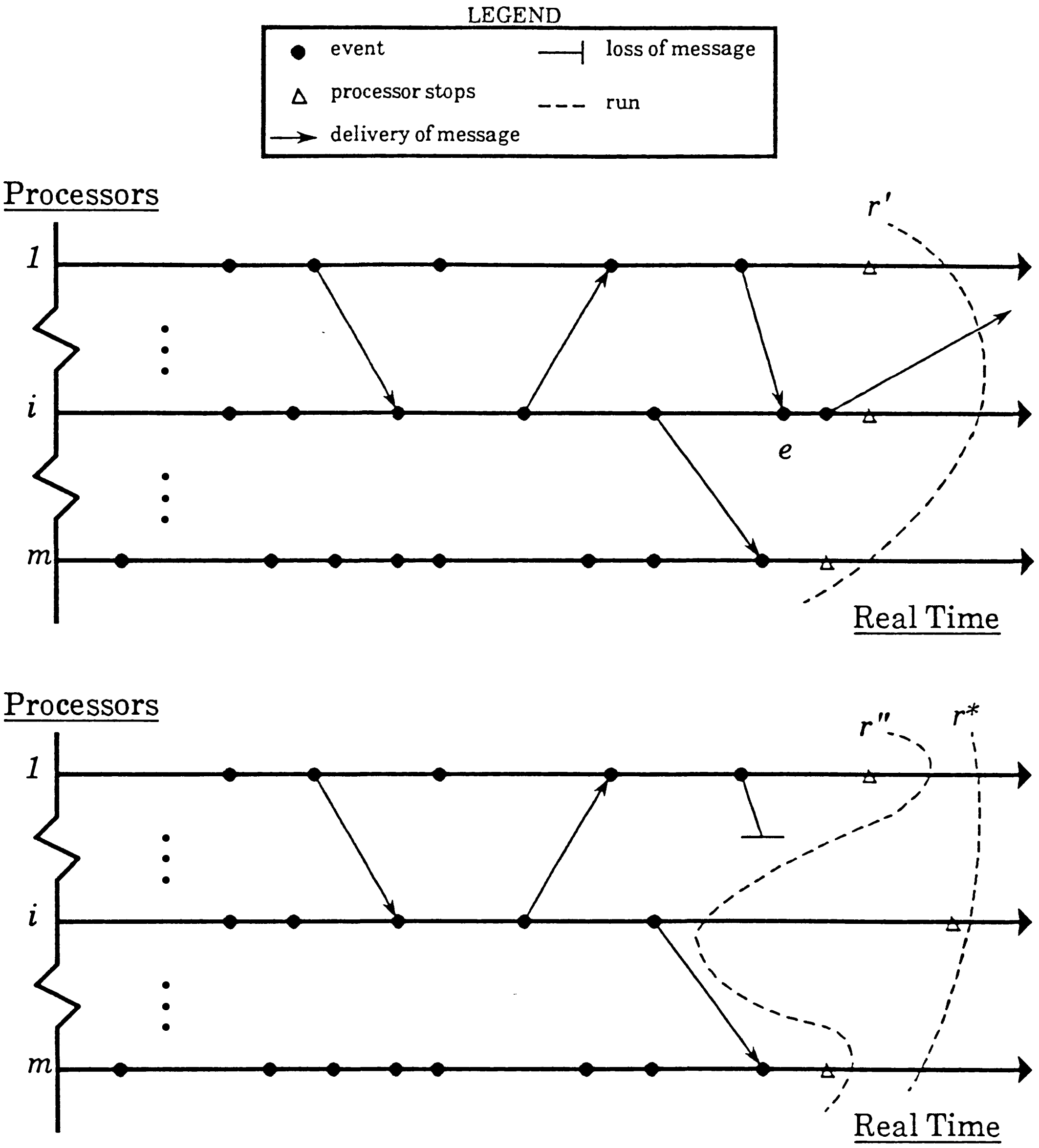

Figure 2: Construction of r'' from r', and r^* from r''. Note that the histories of all processors save that of i are unchanged.

of runs of P. Let $r \models p$ denote that formula p is *interpreted* to be true in r. If p is a primitive proposition ($p \in \Phi$),

$$r \models p \text{ iff } r \in \pi(p).$$

For formulas $\neg p$ and $p \wedge q$ where $p, q \in \mathcal{L}(\Phi)$,

$$r \models \neg p \text{ iff not } r \models p;$$
$$r \models p \wedge q \text{ iff } r \models p \text{ and } r \models q.$$

And finally, processor i knows p in r, if and only if p is true in all the runs that are equivalent to r. $\forall p \in \mathcal{L}(\Phi)$,

$$r \models K_i p \text{ iff } \forall r' \in poss_i(r), r' \models p.$$

A protocol P is said to *guarantee* a formula p, if every run r of P has a continuation r' in P such that $r' \models p$.

3.3 Termination without Knowledge Transfer

In this section, we show that in the presence of transient communication failures, a weakly terminating protocol cannot guarantee the transfer of a processor's local knowledge to another processor. In other words, protocols that guarantee knowledge transfer are necessarily non-terminating.

Chandy and Misra [CM86] defined that a formula p is *local* to a processor i with respect to protocol P, if $\forall r \in P$, $r \models K_i p \vee K_i \neg p$. They show that in an asynchronous system, a processor must receive messages to acquire knowledge of a non-local formula:

Lemma 1 (CM86) *Let p be a formula that is not local to processor i and is local to some other processors in protocol P. If for some run r and its continuation r' in P, $r \models \neg(K_i \vee K_i \neg p)$ and $r' \models K_i p \vee K_i \neg p$, then i receives at least one more message in r' than it does in r.* □

A formula p is an *a priori* formula of i in protocol P, if for all initial runs r of P, $r \models K_i p \vee K_i \neg p$. Obviously, if p is not a priori to i, then it is not local to i.

Theorem 2 *Let P be a protocol, and $p \in \mathcal{L}(\Phi)$ be a formula that is not a priori to processor i and is local to some other processors in P. If P is weakly terminating, it cannot guarantee $K_i p \vee K_i \neg p$.*

Proof: By contradiction. Without loss of generality, suppose that P guarantees $K_ip \vee K_i\neg p$ and is weakly terminating. Since p is not a priori to i, P has an initial run r such that $r \models \neg(K_ip \vee K_i\neg p)$. Since P is weakly terminating, by Theorem 1, r has a terminating continuation r' in P in which no messages are received. Furthermore, since $r \models \neg(K_ip \vee K_i\neg p)$ and no messages are received in r', by Lemma 1, $r' \models \neg(K_ip \vee K_i\neg p)$. Thus P does not guarantee $K_ip \vee K_i\neg p$, a contradiction. $\Box$

4 Discussion

We have showed that in asynchronous systems with transient channel failures, only non-terminating protocols can guarantee transfer of knowledge. This result can be extended to systems with synchronous processors and synchronous communication.

Synchronicity is a critical parameter of the problem of reaching agreement in the presence of processor failures. In asynchronous systems, there are no deterministic solutions even if only one processor may fail by halting [FLP85]; in synchronous systems, however, several solutions are known [DDS87,Fis83]. In contrast, our negative result holds for both synchronous and asynchronous systems.

Since weakly terminating protocols do not guarantee knowledge transfer, we may have to settle for protocols that guarantee only that all but one processors will terminate. The following problem serves as an illustration. Processors i and j are connected by a link with transient failures. We want a protocol that i can use to send a message m to j such that

1. j receives m from i, and

2. i and j are eventually allowed to forget m.

Such a protocol P is given in Figure 3 [Ske86]. Since channel failures are transient, one of the copies of m that i sends repeatedly to j is guaranteed to be received by j. Similarly, the acknowledgements $ack(m)$ from j, and $ack(ack(m))$ from i, will also be received by i and j, respectively. It is now easy to see that protocol P achieves the two goals. Note, however, that processor i never terminates (it will remain in the do-forever loop).

Let *ack* denote acknowledgements.

processor i	processor j
repeat until receipt of $ack(m)$	wait to receive m;
send m to j;	**repeat until** receipt of $ack(ack(m))$
od;	send $ack(m)$ to i;
forget m;	**od**;
do forever	forget m;
if $ack(m)$ is received **then**	**stop**.
send $ack(ack(m))$ to j;	
od.	

Figure 3: Protocol P.

Our result shows that this is not a deficiency of this particular protocol: in a system with transient channel failures, any protocol that achieves goals (1) and (2) also guarantees that j knows m; therefore, it must be non-terminating.

Acknowledgement

We would like to thank Amr El Abbadi and Tommy Joseph for their valuable comments on previous drafts of the paper, and Gil Neiger for his contribution to Appendix A.

References

[CM86] K. M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1:40–52, 1986.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of ACM*, 34(1), January 1987.

[Fis83] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Intl. Conference on Foundations of*

Computations Theory, Lecture Notes in Computer Science, Volume 158, pages 127–140, Springer-Verlag, New York, 1983.

[FLP85] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of ACM*, 32(2):374–382, 1985.

[Gra78] J. Gray. Notes on database operating systems. In *Lecture notes in computer science 60*, Springer Verlag, 1978.

[Had86] Vassos Hadzilacos. Connectivity requirements for byzantine agreement under restricted types of failures. 1986. Unpublished manuscript.

[HM85a] J. Y. Halpern and Y. Moses. A guide to the modal logics of knowledge and belief. In *Ninth International Joint Conference on Artificial Intelligence*, pages 480–490, 1985.

[HM85b] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. 1985. To appear in JACM. An earlier version of the paper appeared in the Third ACM Symposium on Principles of Distributed Computing, 1984.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Ske82] D. M. Skeen. *Crash recovery in a distributed database system.* PhD thesis, Computer Science Division, University of California, Berkeley, 1982.

[Ske86] D. M. Skeen. Private Communication, April 1986.

Let $ack^0(m)$ denote m, and $ack^{k+1}(m)$ denote the acknowledgement to $ack^k(m)$.

```
processor i
k := 0;
do forever
   repeat until receipt of ack^{k+1}(m)
      send ack^k(m) to j;
   od;
   k := k + 2;
od.
```

```
processor j
k := 1;
do forever
   wait to receive ack^{k-1}(m);
   repeat until receipt of ack^{k+1}(m)
      send ack^k(m) to i;
   od;
   k := k + 2;
od.
```

Figure 4: Protocol Q allows processors i and j to attain eventual common knowledge of m.

Appendix A: Achieving Eventual Common Knowledge

We present a protocol that allows processors i and j to gain eventual common knowledge of a fact m despite transient channel failures. We give only an informal description here. (The formal definition of eventual common knowledge is in [HM85b].)

A fact is *eventually* true in a run r, if it must become true some time in the "future" of r. Furthermore, a fact m is *stable* with respect to a protocol P, if for all run $r \in P$, once m becomes true in a run r, it remains true in all continuations of r in P. Eventual common knowledge of a stable fact m is achieved in a run r, if in r, m is true, and that eventually every processor knows that m is true, and that eventually every processor knows that eventually every processor knows that m is true, $\cdots$, *ad infinitum*.

Let m be a stable fact and suppose that processor i knows that m is true. We claim that the non-terminating protocol Q in Figure 4 allows processors i and j to attain eventual common knowledge of m, despite transient channel failures. It is clear that the repeated sending of $ack^0(m)$ by processor i guarantees that at least one $ack^0(m)$ will arrive at processor j. Hence, j will send $ack^1(m)$ to i repeatedly until at least one $ack^1(m)$ arrives at i. Thus, it follows by induction that for all $k \geq 0$, at least one $ack^{2k}(m)$ will arrive at j, and at least one $ack^{2k+1}(m)$ will arrive at i

despite transient channel failures. Let $E^1 m$ denote that both i and j know m, and $E^{k+1} m$ denote that both i and j know $E^k m$. It is easy to see that the receiving of $ack^0(m)$ by j implies $E^1 m$. In general, for all $k \geq 0$, the receiving of $ack^{2k}(m)$ by j implies $E^{k+1} m$. Thus, despite transient channel failures, i and j achieve eventual common knowledge of m by executing protocol Q. Note, however, that neither i nor j ever stop executing Q.