Early stopping algorithms for distributed agreement under fail-stop, omission, and timing fault types

P.D. Ezhilchelvan
Early stopping algorithms for distributed agreement under fail-stop, omission, and timing fault types

P.D. Ezhilchelvan

Abstract

Reaching agreement in the presence of faults is a fundamental problem in fault tolerant distributed computing. This paper describes, and establishes the correctness of, algorithms for reaching distributed agreement which are tolerant of faults of certain types. Fail-stop, omission, and timing faults are the three types of faults considered. Taking an upper bound on the number of processors that can possibly fail, an agreement algorithm is developed for each type of fault. Execution of these algorithms will be stopped early, when the number of processors that actually failed happens to be significantly less than the upper bound. It is observed that, for the same number of processor failures, the fail stop fault tolerant algorithm can be potentially faster than the omission fault tolerant algorithm which is in turn faster than the algorithm tolerant of timing faults.


Series Editor: M.J. Elphick

© 1987 University of Newcastle upon Tyne. Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
Bibliographical details

EZHILCHELVAN, Paul Devadoss

Early distributed agreement under fail-stop, omission, and timing fault types  [By] P.D. Ezhilchelvan.


(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 235.)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.

Abstract

Reaching agreement in the presence of faults is a fundamental problem in fault tolerant distributed computing. This paper describes, and establishes the correctness of, algorithms for reaching distributed agreement which are tolerant of faults of certain types. Fail-stop, omission, and timing faults are the three types of faults considered. Taking an upper bound on the number of processors that can possibly fail, an agreement algorithm is developed for each type of fault. Execution of these algorithms will be stopped early, when the number of processors that actually failed happens to be significantly less than the upper bound. It is observed that, for the same number of processor failures, the fail stop fault tolerant algorithm can be potentially faster than the omission fault tolerant algorithm which is in turn faster than the algorithm tolerant of timing faults.

About the author

Mr. Ezhilchelvan has been a Research Associate in the Computing Laboratory since September, 1983.

Suggested keywords

ALGORITHMS
DISTRIBUTED AGREEMENT
EARLY STOPPING
FAULT TOLERANCE
FAULT TYPES
MESSAGE DELAY
SYNCHRONISED CLOCKS

Suggested classmarks (primary classmark underlined)
Dewey (18th):  001.64404  621.381958
U.D.C.  519.687  681.322.06
1. INTRODUCTION

In this paper, we describe early stopping algorithms for reaching distributed agreement which are tolerant of faults of certain types. The problem of reaching distributed agreement, named by Lamport, Pease, and Shostak [Lampo82] as the Byzantine generals problem, can be described as follows: Consider a distributed system made up of n, n > 1, potentially faulty processors that are capable of communicating with each other only by message passing. It is assumed that the communication medium is fault free and that faults occur only in processors. Any non-faulty processor can not, however, ascertain which other processors are faulty. The type of faults considered for the agreement problem is often taken to be the least restrictive one in which no assumptions are made on responses produced by faulty processors. A faulty processor might exhibit what is called the Byzantine behaviour of transmitting conflicting values and of colluding with other faulty ones in trying to "sabotage" any attempt to solve the agreement problem.

Among these n processors in the system, one processor is designated as the sender and the others receiver processors. The sender is to choose a value from many potential values and send it to all receiver processors so that they all receive the same value. When the sender is faulty, it cannot be guaranteed that all non-faulty receiver processors receive the same value directly from the sender. The problem is to develop an agreement algorithm that can be executed to guarantee the following two, unanimity and validity, conditions to be met by all nonfaulty receiver processors for the given sender.

C1(unanimity):
all non-faulty receiver processors agree on the same value;

C2(validity):
if the sender is non-faulty, all non-faulty receiver processors agree on the value the sender chose to send.

When these two conditions are met, the receiver processors will be said to have reached agreement (on the sender's value).

The development of agreement algorithms has been explored extensively [Pease80, Dolev83, Dolev82a]. Agreement algorithms are developed by parameterising the maximum number of processors that can fail as t. Dolev and Strong [Dolev82b] established that reaching agreement in the presence of at most t distinct processor failures requires at least (t+1) synchronised phases or rounds of information exchange to be carried out by the processors. Dolev, Reischuk, and Strong [Dolev82c] were the first to investigate agreement algorithms whose execution can terminate taking fewer than (t+1) rounds of information exchange. They came up with two types of agreement that can be reached: immediate and eventual. This classification led the authors to the following definition of early stopping: an agreement algorithm tolerant of at most t distinct processor failures is said to exhibit early stopping if all non-faulty processors are guaranteed to reach agreement in strictly less than (t+1) rounds of information exchange in all executions in which at most f, f < t, distinct processors can fail. An area of application for early stopping agreement algorithms will be distributed transaction commit where non-faulty processors on different sites have to unanimously agree on commit/abort decision on results of a transaction. These algorithms will enable the processes to reach agreement as quickly as possible with each non-faulty process knowing the time by which all
other non-faulty ones will have agreed on the decision it has agreed on.

Early stopping algorithms have been developed[DoLev82c, Perry85] under the least restrictive failure assumptions in which faulty processors are capable of exhibiting Byzantine behaviour. Here, we develop early stopping algorithms considering restricted types of faults, namely fail-stop, omission, and timing faults. Our fail-stop fault tolerant, and omission fault tolerant algorithms are found to be faster than those presented in[Hadzia, Lampa84]. The algorithms are developed in the context mentioned above: in a distributed system of \( n \) processors that are capable of reliably communicating with each other by message passing, at most \( t \) processors can fail, while the receiver processors are to reach agreement on the sender's value which is to be broadcast by the sender at some known time. Modifying these algorithms to the general context of every processor in the system being a sender and sending a value to all other processors is straightforward.

The rest of this paper is organised as follows: In the next section, the three fault types are described. The section 3 explains the assumptions involved in the design of the algorithms. The sections 4, 5, and 6 respectively describe, and establish the correctness of, the fail-stop fault tolerant, the omission fault tolerant, and the timing fault tolerant algorithms. In each of these three sections, important observations about the respective algorithm are presented.

2. DESCRIPTION OF THE FAULT TYPES

A processor fails when its behaviour deviates from that specified. Failures are caused by faults. As a consequence of having received a service request from the environment, a correctly functioning processor will produce (i.e. respond with) an output value that is in accordance with its specification. The response of a processor for a given service request will be correct, if the output value is as expected and produced on time[Kopet85]. The response of a faulty processor need not be correct. The characterisation of fail-stop, omission, and timing faults in a processor is fairly well known[Schli83, Mohan83, Crist85, Ezhi86] and is briefly described below: A fail-stop fault causes a processor to stop functioning; the corresponding failure is called a fail-stop failure. A fault that causes a processor, for a given service request, not to produce a response is called an omission fault and the corresponding failure an omission failure. A timing fault causes a processor to produce the expected value for a given service request either early or late; the corresponding failure is a timing failure.

A fail-stop fault causes a component not to respond for any service request. Thus, fail-stop faults can be seen to form a proper subset of omission faults. A processor's not producing a response, for a given request, due to an omission fault can be regarded as that processor's producing the expected value for the request infinitely late. Thus, omission faults form a proper subset of timing faults. The relationship, "form a proper subset of", among the types of faults concerned here would imply that the omission fault tolerant algorithm can be expected to be fail-stop fault tolerant; and the timing fault-tolerant one can be expected to work tolerating omission faults.

3. ASSUMPTIONS

We make two major assumptions in the design of our algorithms.
Assumption A1:

If, at time T, an event occurs in a non-faulty processor p and causes a message to be formed and sent to another non-faulty processor q, then that message will be received in q at time \( T_r, T \leq T_r < T + d \) - where time is measured according to either processor's clock.

The assumptions underlying A1 are: non-faulty processors are reliably connected; d is fixed independent of events that cause messages to be sent, of message routing and message traffic in the network, and of processing loads in processors.

Assumption A2:

At any time, the clock readings of any two non-faulty processors differ by at most e.

The processors’ clock readings tend to drift apart due to small differences in their running rates. Therefore A2 requires that processors' clocks be periodically resynchronised. Fault tolerant clock synchronisation algorithms appropriate to the system context of our algorithms do exist[Halpe84, Crist86] and we simply assume their proper implementation.

Based on A1 and A2, the following can be stated:

If an event occurs in a non-faulty processor p at time T according to p’s clock and causes a message to be formed and sent by p to another non-faulty processor q, then the message will be received in q at time \( T_r, T - e \leq T_r < T + d + e \), according to q’s clock.

It should be noted that the assumptions A1 and A2 are essential for developing agreement algorithms; without them, a processor cannot decide whether a message has not been sent at all, or is yet to be sent by another processor. With processors not being able to solve this ambiguity, agreement algorithms cannot be designed[Fisch83].

We explain a few notations that are frequently used in latter sections. The sender processor is denoted by s and its value to be broadcast to all receiver processors by V. \( T_0 \) is the time when s has to broadcast its value, V. (\( T_0 \) is known to all receiver processors.) The set of all processors in the system is denoted by P and \( |P| = n \). N and F are any two subsets of P.

The messages exchanged by processors during an execution of an agreement algorithm are denoted by \( m(v,i) \) explicitly expressing two important items of their contents: v, the value a given message intends to deliver to a processor that receives it and i, an integer, is used to identify a given message to group it with other relevant messages.

In all the three types of faults concerned, the sender processor, if faulty, is prevented from broadcasting messages with different values, and any faulty receiver processor from attempting to alter the value of the message to be relayed. Hence the receiver processors, in any execution of the algorithms, have to decide either to agree on V or to conclude that the sender failed to broadcast its value. We call the latter situation deciding to agree on 'default' value as the sender's value. In the same way, a processor's decision on a non-default value will be taken to mean that it is deciding on V.
4. FAIL-STOP FAULT TOLERANT ALGORITHM

A description of the early stopping algorithm tolerant of fail-stop faults is as follows.

In any execution of the algorithm, the sender processor, at clock time $T_0$, is to broadcast its message $m(V, 0)$ containing its value $V$ to all receiver processors. Each receiver processor waits for a message that contains a value to be decided on - i.e. either for a message $m(v = V, i)$ for some $i$, $0 \leq i \leq t$, or $m(v = \text{default}, i)$ for some $i$, $3 \leq i \leq t$. On receiving such a message it decides on the value contained in the message and stops after broadcasting $m(v, i+1)$ to other processors, if $i < t$. When such a message is not received, the processor continues to wait for it; while waiting, it executes the following steps:

S1 When the clock reads $T_0 + id + e$ for $i \geq 1$, it broadcasts to all other receiver processors a message $m(?, i)$ meaning "are you decided?" If $i = 1$ or 2, it forms a set $F_i$ containing the sender processor. The set $F_i$, for any $i$, $i \geq 1$, will contain processors that have been observed, at time $T_0 + id + e$, to have failed.

S2 Whenever it receives a message $m(?, j)$, for some positive integer $j$, from another processor, it replies by sending a message $m(\text{X}, j+1)$ meaning "I am not yet decided".

S3 When clock reads $T_0 + id + e$, for $i$, $i \geq 3$, it collects those receiver processors that replied by sending $m(\text{X}, (i-1))$ in response to its message $m(?, (i-2))$ into a set $N$ and computes the set $F_i = P \cap N$.

S4 If $|F_i| \leq i-2$ or $|F_i| = |F_{i-2}|$, it decides on default and stops after broadcasting $m(v = \text{default}, i)$ to all receiver processors.

If a processor cannot decide either by receiving a message containing another processor's decision or by computing appropriate $F$'s until its clock reads $T_0 + (t+1)d + e$, it decides to agree on default value and stops the execution.

In the following, we explain the algorithm by presenting it in a program-like format. The presentation of the algorithm assumes the use of an 'if' construct - if $<\text{boolean}>$ then $<\text{statements}>$ fi - in which statements will be executed, if the boolean statement is found true and a 'while' construct - while $<\text{boolean}>$ do $<\text{statements}>$ od - in which statements are executed while the boolean statement remains true.

The algorithm is presented below:

```
sender-processor:
begin
  broadcast(m(V,0)) to all receiver-processors;
  stop
end.
```
receiver-processor:

decided:boolean; i, j: integer;
begin

decided := false; i := 1;
while not decided and clock ≥ T₀ do
receive-message(m(v, j ≥ 0));
1a if received is m(v = V, j) or m(v = default, j) then
   decide(v); decided := true;
   if (j < t) then
      broadcast(m(v,(j+1))
   fi;
1b fi;
2a if received is m(v = "?", j) and not decided then
   reply(m("X",(j+1))
2b fi;
3a if not decided and clock = T₀ + (t+1)d + e then
   decide(default); decided := true
3b fi;
6a if not decided and clock = T₀ + id + e and i ≤ t then
   if i ≤ 2 and i ≤ t-2
      then broadcast(m("?",i)); Fᵢ := \{s\}
   4a fi;
5a if i ≥ 3 then
      Nᵢ := \{processors replied m("X",(i-1))\};
      Fᵢ := P \setminus Nᵢ;
      if |Fᵢ| ≤ (i-2) or |Fᵢ| = |Fᵢ-1|
      then
         decide(default); decided := true;
         broadcast(m(default,i))
      fi;
      if not decided and i ≤ (t-2)
      then broadcast(m("?",i))
5b fi;
fi;
6b i := i + 1;
fi;
6c od;
stop
end.

Explanation

In the above presentation of the algorithm, the boolean variable "decided" indicates whether a decision on sender’s value has been made and is initially set to false by all receiver processors. By executing the algorithm, the sender processor broadcasts its value to all receiver processors and each receiver processor, after initialising the variables, starts receiving messages m(v, j ≥ 0). In the first
block of statements [from 1a to 1b], if a message received happens to have v = V or v = default, a decision is made on v, setting "decided" to true. Also, if the message received has j < t, then a message \( m(v, j+1) \) is relayed. Note that j = t (j > t) would mean that the value of the message received has been relayed by (at least) t processors apart from the sender. Therefore, such a message need not be, and is not, relayed any further. The third block of statements [from 3a to 3b] leads to termination of the execution in the worst case of the execution not being terminated before the clock reads \( T_0 + (t+1)d + e \).

In the sixth block of statements [from 6a to 6b], attempts at early stopping the execution are provided. It will be executed once whenever clock reads \( T_0 + id + e \), \( 1 \leq i \leq t \). For \( i = 1 \) or \( i = 2 \), \( F_i \) is formed, in the fourth block of statements [from 4a to 4b], as a singleton set containing s and a message \( m(?, i) \) is broadcast to all receiver processors, if \( i \leq t-2 \). For \( i > (t-2) \), \( m(?, i) \) is not broadcast, since the corresponding reply messages, if any, can only be guaranteed to be received just before the clock is to be read \( T_0 + (t+1)d + e \), at which time any attempt for early stopping would be of little use, as the execution is any way going to be terminated at clock time \( T_0 + (t+1)d + e \). An undecided receiver processor, on receiving a message \( m(?, j) \), \( j \geq 1 \), replies, in the second block of statements [from 1a to 1b], by sending \( m(X, j+1) \). In the fifth block of statements [from 5a to 5b], the processors that replied by sending \( m(X, i-1) \), \( i \geq 3 \), in response to messages \( m(?, i-2) \) are collected in N and \( F_i \) is computed. If the early stopping conditions are satisfied, decision is taken on default and the boolean "decided" is set to true; otherwise, a message \( m(?, i) \) is broadcast. The variable i is incremented by 1 every time the sixth block is executed. When "decided" becomes true, 'stop' is executed to terminate the execution of the algorithm.

4.1. Correctness Of The Algorithm

The algorithm is shown to be correct by establishing a series of lemmas. From here on, \( T_i \) is used to denote \( T_0 + id + e \), for \( i, i \geq 1 \). Also, it is assumed that the statements in the algorithm can be executed in no time (this will require an increase on the value of \( d \)).

Lemma 1

Let p be an undecided non-faulty processor that computes \( F_i \), for some \( i, i \geq 3 \), at its clock time \( T_i \). Any processor that is an element of \( F_i \) must have halted functioning at some time \( T, T < T(i-1) \), according to p's clock.

Proof

Throughout this proof, we measure time according to p's clock. The \( F_i \) of p contains the sender processor which has obviously halted at some time \( T, T \leq T_0 + e < T_i \), \( i \geq 1 \). For every \( i, i \geq 3 \), the non-faulty processor p broadcasts a message, \( m(?, (i-2)) \), to all receiver processors at time \( T(i-2) \). By assumption A1, p's message \( m(?, (i-2)) \) can be received by any other receiver processor at some time \( T_r, T_r < T(i-1) \) and the reply message \( m(X, (i-1)) \), if sent, must be received by p before \( T_i \). If any receiver processor in \( F_i \) had decided and stopped the execution without any failure by \( T_r \), then p must have decided by \( T_i \). But p is undecided. So, every processor in \( F_i \) must have failed and halted by \( T_r \). Hence the lemma.
Lemma 2

Let p be an undecided non-faulty processor that computes $F_i$ at its clock time $T_i$, $i \geq 3$. Any processor that is not in p's $F_i$ must have remained undecided at time $T$, $T \leq T_{(i-2)}$, according to p's clock.

Proof

Lemma is obviously true for p which will not be in its $F_i$. Any other receiver processor that is not in p's $F_i$ would have replied to p's message $m(, (i-2))$, only if it has been undecided at the time p's message was received. The p's message can be received by any other receiver processor, due to A1, at time $T_r$, $T_{(i-2)} \leq T_r < T_{(i-1)}$, according to p's clock. Hence the lemma.

Remark

The proof of this lemma makes use of two facts: a processor does not suffer any undue delay in replying to $m(, j \geq 1)$ and does not produce messages containing incorrect information (such as indicating undecidability when it is indeed decided). So, the above lemma will also hold true for the omission fault type.

Lemma 3

If an undecided non-faulty processor p finds $|F_i| = |F_{(i-2)}|$ at time $T_i$, according to its clock, for some $i$, $i \geq 3$, then no non-faulty processor could have decided, and can ever decide, to agree on a non-default value as the sender's value.

Proof

Throughout this proof, it is supposed that p's clock will be used to measure time. Under fail-stop assumptions, a processor, on failing, halts for ever. Therefore any $F_i$ will be a subset of $F_j$ for j, $j > i \geq 1$. According to the algorithm, p will have $F_1 = F_2 = \{s\}$. The condition that $|F_3| = |F_1|$ would imply that $F_3 = \{s\}$. This would mean that all receiver processors had remained undecided, when p's clock read $T_1$. Hence s must not have sent its message to any of the receiver processors (due to assumptions A1 and A2). Hence the lemma is true for $i = 3$. With similar reasoning, the lemma can be shown to be true for $i = 4$, though this situation can never occur, since the execution would be stopped for $i = 3$ itself.

We prove the lemma for $i, i \geq 5$, by showing that any processor, say q, that is in $F_{(i-2)}$ could not have sent a decision message containing either V or default to any processor, say r, that is not in $F_{(i-2)}$. Assume that q has sent a decision message to r, before halting at $T$, $T < T_{(i-3)}$ (lemma 1). Under this assumption, r cannot be non-faulty, because p is undecided at $T_i$. Hence it must have failed before $T + d < T_{(i-2)}$, and must be counted in p's $F_i$. But $|F_i| = |F_{(i-2)}|$.

So, processors in $P-F_{(i-2)}$ could not have received any message containing either V or default from any processor in $F_{(i-2)}$ at any time before $T_{(i-3)}$ and, by lemma 2, all processors in $P-F_{(i-2)}$ must have remained undecided before $T_{(i-4)}$. An undecided receiver processor can decide on a non-default value only by receiving a message containing V. Hence the Lemma.
Lemma 4

If a non-faulty undecided processor p finds $|F_i| \leq (i-2)$ at its clock $T_i$, for some $i$, $i \geq 3$, then no non-faulty processor could have decided, and can ever decide, on a non-default value.

Proof

Under fail-stop (also omission) fault assumptions, faulty processors do not introduce any undue delay in relaying a received message. Therefore, by A1 and A2, when the clock of any non-faulty processor reads a value greater than or equal to $T_{(j+1)}$, $j \geq 0$, no undecided processor can receive a message $m(V, j)$ which has been sequentially relayed by $j$ distinct receiver processors after being sent by the sender. By lemma 2, all processors in $P-F_i$ must have remained undecided at p's clock time $T$, $T \leq T_{(j-2)}$. If any processor in $P-F_i$ is to receive $m(V, j)$, then that message has to come through processors in $F_i$ (which also includes the sender) such that $j < |F_i|$. But p finds $|F_i| \leq (i-2)$, when its clock has past reading $T_{(j-2)}$. Hence the Lemma.

Lemma 5

When a non-faulty processor decides on default, no other non-faulty processor could have decided, and can ever decide, on a non-default value.

Proof

If a non-faulty processor can decide on default because either of the following conditions $|F_i| = |F_{(i-2)}|$ or $|F_i| \leq (i-2)$ has come true for some appropriate $i$, then, by lemma 3 or by lemma 4, the above lemma is true. Alternatively, a non-faulty processor can decide on default by receiving a message containing default value from another receiver processor. Under the three types of faults concerned, any response of a processor will always be correct. Hence the other processor, while deciding on default, must have functioned like a non-faulty processor, correctly making, and broadcasting, the decision on default value. Hence the Lemma.

Theorem 1

In any execution of the algorithm in which the sender is to broadcast its value at $T_0$ according to its clock, every non-faulty processor reaches agreement on the sender's value in the presence of $f$, $f \leq t$, distinct processors out of $n$ processors suffering fail-stop faults, and stops the execution by not later than $T_0 + \min((t+2)d+e,(t+1)d+e)$ according to its clock after transmitting a total of $O(nf)$ messages.

Proof

Consider an execution of the algorithm. If the sender is non-faulty, every nonfaulty processor would receive the sender's message by $T_0+d+e$ according to its clock, due to assumptions A1 and A2. Hence the validity condition is realised. If the sender is faulty, it is impossible, by lemma 5, to have one non-faulty receiver processor deciding on a non-default value and another one deciding on default value. Hence all non-faulty processors eventually decide either on default or on a non-default value. Thus the unanimity condition is realised.
Suppose that the sender is faulty and that all non-faulty receiver processors decide on V. All non-faulty receiver processors receive \( m(V,i \leq f) \) before \( T_0 + (f+1)d + e \) in their clocks. Suppose that all non-faulty receiver processors decide on default. By Lemma 4, it can be seen that each of them stops the execution at its clock reading \( T_0 + (f+2)d + e \), when \( f < t-1 \). If \( f \geq (t-1) \), each processor will, however, stop at its clock reading \( T_0 + (t+1)d + e \). Therefore, every non-faulty processor reaches agreement not later than \( T_0 + \min((f+2)d + e, (t+1)d + e) \) at its clock.

During an execution of the algorithm, a receiver processor is to broadcast either \( m(?',i) \) or \( m(V \lor default, i) \) to all other receiver processors at its clock reading \( T_0 + id + e \), \( i \geq 1 \), until a decision for agreement is reached. By this way, it broadcasts at most \((n-2)\min(f+2, t)\) messages. Also, it has to reply at most \( i(n-2) \) messages of the form \( m(X', i) \) before \( Ti \), \( i \geq 1 \), until it stops. Hence it broadcasts a total of \( O(nf) \) messages. Hence the theorem.

In the following, we make two observations about the algorithm. The first one concerns early stopping conditions used and the second one illustrates that this algorithm will not work in the presence of omission faults.

4.2. Observation 1

The above algorithm employs two early stopping rules: (i) \( |F_i| \leq (i-2) \) and, (ii) \( |F_i| = |F_{i-2}| \). Satisfying either condition leads to stopping the execution early. The first stopping rule verifies whether the size of \( F \) computed for every given \( i \), \( i \geq 3 \), is sufficiently small to decide on default. Therefore, during an execution, the smaller is the number of failed processors, the more quickly it is to be satisfied.

The second rule works by relating the size of \( F \) computed for a given value of \( i \) to that for \( (i-2) \), \( (i-2) \geq 1 \). Hence, satisfying the second rule in an execution would require the number of failed processors to remain constant over a period of time i.e. no functioning processor being observed to have failed during the period. Therefore, during an execution, the sooner the faulty processors fail, the more quickly the second rule remains to be satisfied. In other words, if all faulty processors fail before \( T_0 + id + e \), \( i > 0 \), according to a nonfaulty processor’s clock, then the processor will stop the execution, by the second rule, not later than \( T_0 + (i+4)d + e \), irrespective of the actual number of failed processors. To illustrate this, consider an execution with the following characteristics: exactly five processors (including the sender), named \( s \), \( w, x, y \), and \( z \), fail; \( p \) is a non-faulty receiver processor whose messages take a non-zero transmission time; and \( t > 6 \).

Let the time be measured according to \( p \)’s clock and \( Ti = T_0 + id + e \), \( i > 0 \). In the execution, let \( s \) fail before \( T_0 \) without broadcasting its value and the other four receiver processors fail before \( T_1 \). The processor \( p \) will now compute all the five faulty processors in \( F_3, F_4 \), and \( F_5 \) respectively at \( T_3, T_4 \), and \( T_5 \). Hence, by the second rule, it stops the execution at \( T_5 \). If it were to stop by the first rule only, it should have stopped at \( T_7 \). If some of the faulty receiver processors, say, \( y \) and \( z \), fail after \( T_1 \) and before \( T_2 \), then, by the second rule, \( p \) will stop the execution not later than \( T_6 \) with all the five faulty ones guaranteed to be computed in \( F_4, F_5 \), and \( F_6 \) respectively at \( T_4, T_5 \), and \( T_6 \).

Under fail-stop assumptions, a failed processor halts functioning for ever. In practical systems, it could often be the case that none of the functioning processors fail during a given execution of the algorithm. In such cases, when \( n > t > 4 \), the effect of the second stopping rule is more significant.
in bringing the execution to an earlier stop (at $T_5$ of respective non-faulty processor's clock), irrespective of the number of processors that have halted functioning before the execution started.

4.3. Observation 2

In an execution of the above algorithm, processors are not guaranteed to reach agreement in the presence of omission faults.

Let us consider an execution in which $q$ is another non-faulty processor in the context characterized in the previous observation. Let the five faulty processors, in this execution, fail before $T_5$, suffering omission faults and not respond to messages of $p$ and $q$ at all. Let the sender's message be transmitted from $s$ only to $w$, from $w$ only to $x$, from $x$ only to $y$, from $y$ only to $z$, and from $z$ only to $q$. Let $q$ receive the message just before $p$'s clock is to read $T_5$. If $p$ does not receive $q$'s message before $T_5$, it will decide on default. So the agreement is not reached.

5. OMISSION FAULT TOLERANT ALGORITHM

It has been observed that the previous algorithm cannot be guaranteed to work in the presence of omission faults. A closer look into the early stopping conditions employed in the previous algorithm will reveal that only the early stopping condition, $|F_i| = |F_{i-2}|$, makes use of the fail-stop feature. The other one, $|F_i| \leq (i-2)$, works by recognizing the situation in which the number of processors that did not respond at some time $T_i$, $i \geq 1$, during the execution, becomes so small and $T_i$ is so late that no undecided processor will ever be able to receive a message from the failed ones. Thus, it can work in the presence of omission faults in processors. So the agreement algorithm for the omission fault type can be derived from the previous algorithm by simply removing the early stopping condition $|F_i| = |F_{i-2}|$. But for this, the omission fault tolerant algorithm is no different from the previous one; in the following, the part of the algorithm containing the only appropriate early condition is presented:

5a

\[
\begin{align*}
\text{if } i \geq 3 \text{ then} \\
N &:= \{ \text{processors replied } m(\text{'X',}(i-1)) \}; \\
F_i &:= P-N; \\
\text{if } |F_i| \leq (i-2) \\
&\quad \text{then} \\
&\quad \text{decide(default); decided: = true;} \\
&\quad \text{broadcast(m(default,i))} \\
&fi;
\end{align*}
\]

5.1. Correctness Of The Algorithm

Theorem 2

In any execution of the algorithm in which the sender is to broadcast its value at $T_0$ according to its clock, every non-faulty receiver processor reaches agreement on the sender's value in the presence of $f$, $f \leq t$, distinct processors out of $n$ processors suffering omission faults, and stops the execution not later than $T_0 + min((f+2)d+e, (t+1)d+e)$ time according to its clock after transmitting a
total of $O(nf)$ messages.

**Proof**

This theorem can be shown correct by restating the correctness of theorem 1, after showing that the fourth and the fifth lemmas that are used to establish the correctness of theorem 1 are also true for any execution of this algorithm. We leave the verification to the reader.

In the following, we make two observations about the above algorithm.

5.2. Observation 3

For the same, $f$, failures of respective types, the fail-stop fault tolerant algorithm will be either as fast as or faster than the above algorithm.

Given the same $f$ failures of respective types, the executions of both the algorithms are to stop as early as $\min \left( (f+2)d + e , (t+1)d + e \right)$ after the start of the execution. The observation 1 illustrates that it is sometimes possible for the execution of the fail-stop fault tolerant algorithm to be stopped, due to the condition $|F_i| \leq |F_{i+2}|$ becoming true, earlier than it would have stopped, if it were to stop by the other condition. Thus, in some execution scenarios, the previous algorithm can be faster than the above one, for the same $f$ failures of respective types.

**Remark**

From the above observation, it can be noted that it is possible to take advantage of the special features of fail-stop faults over omission faults in achieving distributed agreement. There are system contexts where this possibility cannot exist. For example, in [Babao85], the component classification of the system, and the intended information exchange between processors are such that the assumption of fail-stop faults in processors does not render the distributed agreement problem any easier.

5.3. Observation 4

The above algorithm is not tolerant of timing faults.

The correctness of the above algorithm is based on the results of the lemma 4 in which it is stated that no processor can receive $m(v, j \geq 0)$ when, or after, the clock of any non-faulty processor has read $T_0 + (j+1)d + e$. Under the assumptions of timing faults, a faulty processor can be untimely in transmitting/relaying its messages. With $e$ being a non-zero quantity, the above statement, hence the lemma 4, will no longer hold true under timing fault assumptions.

**Remark**

Extending the above algorithm to the one tolerant of timing faults will require: non-faulty processors carry out timeliness checks before accepting a message and $e = 0$. The second requirement can not be realised in practice. However, its equivalence, so far as message transmission delays are concerned, can be obtained by making the lower and the upper bounds on message transmission delays, denoted respectively as $d_{min}$ and $d_{max}$, to be: $d_{min} = -e$ and $d_{max} = (d+e)$. The statement equivalent to that in lemma 4 will become: no non-faulty processor accepts $m(v, j \geq 0)$, when, or after,
any nonfaulty processor's clock has read $T_0 + (j+1)d_{max} = T_0 + (j+1)(d+e)$ (and before any nonfaulty processor's clock had read $T_0 -(j+1)d_{min} = T_0 -(j+1)e$). Based on these facts, the timing fault tolerant algorithm is developed.

6. TIMING FAULT TOLERANT ALGORITHM

The receiver processors, while executing the algorithm, exchange messages in synchronised phases that are uniformly of length $(d+e)$. With the sender processor broadcasting its message $m(V,0)$ at its clock time $T_0$, each receiver processor looks for a decision message, $m(V,i \geq 0)$ or $m(\text{default}, i > 0)$, to be received between $T_0 + i(d+e)$ - $e$ and $T_0 + (i+1)(d+e)$ according to its clock. On receiving such a message during that interval, it decides on the value contained in the message and stops after broadcasting $m(v, i+1)$ at $T_0 + (i+1)(d+e)$ according to its clock. On receiving no such message, it broadcasts $m(X', (i+1))$ at its clock time $T_0 + (i+1)(d+e)$, if $(i+1) < t$. At clock time $T_0 + (i+2)(d+e)$, $(i+2) < (t+1)$, if it finds the number, $|F|$, of processors not having broadcast $m(X', (i+1))$ to be less than $(i+2)$, then it decides on default and stops after broadcasting $m(\text{default}, (i+2))$. If no decision can be made until clock reads $T_0 + (t+1)(d+e)$, the default value is taken to be the sender's value and the execution is stopped.

The algorithm is presented below:

sender-processor:

begin
  broadcast($m(V,0)$)
  to all receiver-processors;
  stop
end.

receiver-processor:

decided:boolean;
i, j:integer;
M: set-of-messages;
T: clock;

begin
  decided := false;
i := 1;
while not decided and clock $\geq T_0 - e$
   do
      receive-message(m(v, j));
      if clock = $T_0 + i(d + e)$
         then
         M = (m(v, (i-1)) received at T,
              $T_0 + (i-1)(d + e) - e \leq T < T_0 + i(d + e)$
         if m(v = V, i-1) or m(v = default, i-1) in M
            then
            decide(v); decided := true;
            if $i \leq t$ then
               broadcast(m(v, i))
            fi;
         fi;
      if not decided and clock = $T_0 + (t+1)(d + e)$
         then
         decide(default); decided := true
      fi;
      if not decided and $i > 1$
         then
         N := \{processors from which m('X', (i-1))
               in M was received\};
         $F_i := P - N$;
         if $|F_i| < i$
            then
            decide(default); decided := true;
            broadcast(m(default, i))
         fi;
      fi;
      if not decided and $i < t$
         then broadcast(m('X', i))
      fi;
   i := i + 1;
   od;
stop
end.

6.1. Correctness Of The Algorithm

In establishing the correctness of the algorithm, we let $T_i = T_0 + i(d + e)$, for $i \geq 0$. 
Theorem 3

In any execution of the above algorithm in which the sender is to broadcast its value at $T_0$ according to its clock, every non-faulty receiver processor reaches agreement on the sender's value in the presence of $f$, $f \leq t$, distinct processors out of $n$ processors suffering timing faults, and stops the execution not later than $T_0 + (f+1)(d+e)$ according to its clock after transmitting a total of $O(nf)$ messages.

Proof

Consider an execution of the above algorithm. When the sender is non-faulty, all non-faulty receiver processors receive $m(V, 0)$ from the sender and stop the execution at $T_1$ after taking, and broadcasting, the decision on $V$. Hence, the validity condition is met.

Suppose that the sender is faulty. Suppose also that a non-faulty processor, say $p$, accepts $m(V, i-1)$, and decides on $V$ at its clock time $T_i$, $i \leq (t+1)$. This would mean that $(i-1)$, $(i-1) \leq f \leq t$, processors including the sender are faulty. The processor $p$ stops at clock time $T_i$, i.e., by $T(f+1)$. If $i < (t+1)$ and if there are undecided non-faulty processors at $T_i$, then $i \leq f$ and all those undecided processors will decide on $V$ at $T_{(i+1)}$ by receiving $m(V, i)$ from $p$. Suppose that, in another execution, the sender is faulty and the non-faulty processor $p$ finds, at its clock time, $T_i, 2 \leq i \leq t$, that $|F| < i$. The processor $p$ decides on default and stops the execution by $T(f+1)$, since $f \geq |F|$. Note that the processor $p$, on computing $F$, forms the set $N$ by collecting all processors from which $m(X', (i-1))$ was received before its clock time $T_i$. Since processors, under timing fault assumptions, do not fail by producing messages of incorrect information, all processors in $N$ must have remained undecided at their clock reading $T_{(i-1)}$. Since $|F| < i$, no processor in $p$'s $F$ can ever receive a valid message containing $V$. That is, when $p$ decides on default, no non-faulty processor would have ever received, and will ever receive, a valid message containing $V$ and all non-faulty processors can decide only on default. Every undecided processor in $p$'s $N$ will decide on default by receiving appropriate message from $p$ at clock time $T_{(i+1)}$. Since these processors did not have $|F| < i$ at clock time $T_i$, $f \geq i$ and thus they all stop the execution by $T(f+1)$. If, in an execution, $t$ distinct processors including the sender fail, then every non-faulty processor will decide on default and stop executing the algorithm when its clock reads $T_0 + (t+1)(d+e)$. Thus, the unanimity condition is met.

It can be seen that a non-faulty processor, during any execution, has to broadcast, until it stops, either $m(X', i \leq 1)$ or $m(V, i \leq 1)$ to all other receiver processors at its clock time $T_i$, for every $i$, $i > 0$. Hence the number of messages broadcast will be not more than $(n-2)(min((f+1), t))$. Thus the theorem is proved.

6.2. Observation 5

For the same $f$, $f \geq 3$, failures of respective types, the fail-stop fault tolerant, and omission fault tolerant, algorithms are faster than the above algorithm.

Any execution of the above algorithm will be stopped as early as $T_0 + (f+1)(d+e)$ according to a non-faulty processor's clock. Whereas the execution of the previous two algorithms can be expected to stop by $T_0 + max((f+2)d+e, (t+1)d+e)$. Dolev and Halpern[Dolev84] established that $e$ would be at least as large as $d/2$. Assuming that $e = d/2$, the first two algorithms can be seen to be faster.
than the third one, when \( f > 3 \).

**Remark**

The design of early stopping algorithms presented in [Lampo84, Hadzia] is not effectively influenced by the no-untimely-response nature of fail-stop and omission failures. As a result, the fail-stop fault, and omission fault, tolerant algorithms presented there turn out to be as slow as the timing fault tolerant algorithm presented here.

7. **CONCLUSIONS**

Early stopping algorithms for reaching agreement in the presence of fail stop, omission, and timing faults have been presented. Fail-stop faults form a proper subset of omission faults which, in turn, form a proper subset of timing faults. We observed that the algorithm tolerant of fail-stop faults can be sometimes faster than the omission fault tolerant algorithm, for the same number of failures of respective types. Since a fail-stop faulty processor fails to halt functioning for ever, the earlier all faulty processors fail, the earlier the execution of the algorithm tends to stop. With omission fault assumptions, a processor, once failed, can later produce correct responses. So, stopping the execution of the omission fault tolerant algorithm gets delayed in direct proportion to the number of processors that failed. We also observed that the omission fault tolerant algorithm is faster than the algorithm tolerant of timing faults due to a timing faulty processor's failure of producing untimely responses. The relative earlier stopping capabilities of these algorithms reveal the fact that each algorithm has been designed without failing to make complete use of the distinct features that characterise the respective type of faults to be tolerated. So, we believe, the algorithms presented here make a significant contribution to the area of early stopping algorithms for reaching agreement in distributed systems.

8. **ACKNOWLEDGEMENTS**

This work was supported by the U.K. Science and Engineering Research Council. My thanks are due to Professor S. K. Shrivastava, Professor B. Randell, and the two anonymous referees of the sixth SRDSDS who commented on this paper.
References

Babao85.

Crist85.

Crist86.

Dolev82a.

Dolev82b.

Dolev82c.

Dolev83.

Dolev84.

Ezhil86.

Fisch83.

Hadzia.
Halpe84.

Kopet85.

Lampo82.

Lampo84.

Mohan83.

Pease80.

Perry85.

Schli83.