Consensus and Termination Detection
in the Presence of Faulty Processes

Li-Fen Wu  Ten-Hwang Lai  Yu-Chee Tseng
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210

Abstract

The consensus problem is well known to be unsolvable in a totally asynchronous system even in the face of a single faulty node. It has been believed in the literature that as achieving consensus is impossible in the presence of faulty processes, so is detecting termination. However, to the best of our knowledge, no one has proved it. In this paper, we not only confirm this belief but also show that termination detection is actually harder than consensus. Necessary/sufficient conditions for each problem to be solvable in the presence of faulty nodes are also obtained.

Key Words: Fault tolerance, termination detection, consensus, distributed algorithm, fault detectability.

1 Introduction

A distributed system involves \( n \) processes that run on top of a communication network (e.g., at the application layer of the OSI ISO reference model). The problem of reaching a consensus among remote processes and the problem of distributed termination detection are two of the most fundamental problems in distributed computing. The Consensus Problem which is to achieve an agreement among processes, initially has an input of 0 or 1 for each process. Eventually a common value has to be agreed by all nonfaulty processes. An algorithm that solves the consensus problem is called a consensus protocol. The Termination Detection Problem monitors a distributed system, often called the basic or underlying system, and determines whether the underlying system has no possible future changes. Each process of the underlying system switches between two possible states: active and idle. Active processes may send or receive messages at will. Idle processes can receive but cannot send messages. If an idle process receives a message, it becomes active immediately. The underlying system is said to be terminated if i) all processes are idle and ii) there are no messages in transit. The problem of termination detection is to determine whether the underlying system has terminated. A distributed algorithm that solves the termination detection problem is called a termination detector. A termination detector itself may be regarded as a distributed system whose function is to determine whether another distributed system, the underlying system, is terminated.

The consensus problem is particularly interesting in the presence of faulty processes. In a surprising result [8], Fischer et al. showed that in a totally asynchronous system, the consensus problem cannot be solved even in the presence of a single fault. This is true even if the type of failure considered is the most benign one, the fail-stop fault, which a process fails by suddenly stopping functioning. A protocol is said to be \( k \)-resilient or \( k \)-solves a problem if it works correctly even in the face of \( k \) faulty processes. Thus, in a totally asynchronous system, no 1-resilient consensus protocol is possible. On the other hand, Dolev et al. [7] identified two "minimal" cases in which the consensus problem can be \( n \)-solved without using broadcast transmission. They are
(1) synchronous processes and synchronous communication;
(2) synchronous processes and synchronous message order.

These two conditions are each said to be "minimal" because any weakening of either condition by changing one parameter from synchronous to asynchronous is sufficient to prove that there is no $k$-resilient protocol where $k$ is either 1 or 2.

Each of the above two conditions implies that process failure is detectable. However, that fault can be detected within a finite amount of time (or fault detectability) does not necessarily imply any of these conditions. That is, fault detectability is weaker than them. It is thus natural to make this weaker assumption and ask whether fault detectability is sufficient for the existence of an $n$-resilient consensus protocol. In the first part of this paper, we answer this question in the affirmative.

The second part of this paper studies the relationship between consensus and termination detection. Intuitively, termination detection seems to be a special kind of consensus problem, and we wonder whether similar possibility/impossibility results hold for termination detection. In this paper, we show that termination detection is actually harder than the consensus problem.

In view of this result, it is not surprising to notice that most termination detectors available in the literature (e.g., [2, 4, 5, 6, 9, 10, 11, 12, 14, 15, 16]) are not even 1-resilient. When a termination detector is applied to a fault-tolerant underlying system, it is desirable that the former also be fault tolerant. Recently, termination detection was claimed to be 1-solvable [17] if the underlying communication network satisfies the following conditions: a) all messages sent from a process to another process are delivered without errors, with a finite but arbitrary delay, and in a FIFO fashion; b) should a processor fail, each nonfaulty process knows of the failure within a finite amount of time; c) a message sent to a failed processor is returned to the sender; d) each processor is able to send two or more messages atomically.

As a by-product of the aforementioned "termination detection harder than consensus" result, we will show that the above conditions (a, b, c, d) are actually not sufficient for the existence of a 1-resilient termination detector. The protocol described in [17] needs a stronger condition in order to work correctly.

2 Problem Model

A totally asynchronous distributed system consists of $n$ processes, $P = \{p_1, p_2, \ldots, p_n\}$, running on top of a communication network (e.g., at the application layer of the OSI ISO reference model). All messages sent from a process to another process are correctly delivered with a finite but arbitrary delay. The relative speed between processes is not known a priori.

We only consider the most benign fail-stop process failure, upon which a process fails by suddenly stopping to function. Further assume a failed process does not restart or partition the network.

A protocol is a distributed algorithm running on $n$ processes. An execution of a protocol $\alpha$ is the sequence of interleaved operations executed by the $n$ processes that are running $\alpha$. A protocol is $k$-resilient or $k$-solves a problem if it works correctly in every execution even in the face of $k$ faulty processes.

Throughout the paper, CON stands for the "non-trivial" consensus problem ("non-trivial" means that the processes cannot always decide on 0 or always decide on 1), and TD for the termination detection problem. The $CON^k$ problem requires every nonfaulty process eventually decides on the same value in a system with at most $k$ faulty processes. The $TD^k$ problem, assuming at most $k$ possible process failures, requires every nonfaulty process to decide on 1 in a finite amount of time after the underlying system is terminated; here, in the presence of faulty processes, the underlying system is said to be terminated if every nonfaulty process is idle and no basic messages, which are sent by the underlying system, addressed to a nonfaulty process are still in transit.
A1: (Upon receiving Decide(v) from p_j)
   if (CON_i = \perp) and (j \notin F_i)
   then CON_i := v;

A2: (Upon detecting process p_i's failure)
   F_i := F_i \cup \{x\};  (* P = \{1..n\}. *)
   if i = \min(P - F_i) then leader_i := true;
   else send Check(F_i, CON_i) to \min(P - F_i);

A3: (Upon receiving Check(F, v) from p_j)
   FaultSet_i[j] := F_i;
   if (CON_i = \perp) and (v \neq \perp)
   then CON_i := v;

A4: (When (leader_i = true) and (\{1, \ldots, i-1\} \subseteq \text{FaultSet}_i[j] for all j \in P - F_i - \{i\}))
   if CON_i = \perp then
     CON_i := x_i;
     (* x_i is the input register of p_i. *)
     for j \notin F_i do
       send Decide(CON_i) to p_j;
   endif

Figure 1: Protocol CONSENSUS for process p_i, 1 \leq i \leq n.

3 An \textit{N}-Resilient Consensus Protocol

As mentioned in Section 1, the assumption of fault detectability is weaker than the two minimal cases described in [7]. In this section, we show fault detectability is indeed sufficient for the existence of an \textit{N}-resilient consensus protocol.

Assume process failures are detectable within a finite amount of time. We design an \textit{N}-resilient protocol for the consensus problem. The protocol, referred to as Protocol CONSENSUS, is presented in Figure 1 as four event-driven actions. Each action is triggered when its "guarded" condition is satisfied.

Each process has a unique ID number, which, without loss of generality, is assumed to be between 1 and \textit{n}. At any time, the nonfaulty process with smallest ID is defined to be the leader. Each process \textit{i} maintains the following four variables. Their initial values are either \perp (undefined), false, or \emptyset (empty set), if no explicit specification is described.

1. \textit{CON}_i is the output decision register indicating the decision made by \textit{p}_i.

2. leader_i is a boolean variable indicating whether \textit{p}_i is the process with smallest ID among all nonfaulty processes, so as to qualify for making a final decision. Initially, only leader_1 is true.

3. \textit{F}_i records the set of faults known to \textit{p}_i.

4. FaultSet_i is an integer array, where FaultSet_i[j] indicates the faults detected by \textit{p}_j that are known to \textit{p}_i.

The protocol employs two kinds of messages: Decide(v) and Check(F, v). A Decide(v) message is used for the leader to inform other processes about the value \textit{v} it has decided on. A Check(F, v) message is used for a process to inform the leader the faults \textit{F} it has detected and the value \textit{v} it has reached.

In A1, when \textit{p}_i receives a Decide(v) message from \textit{p}_j, if \textit{i} according to \textit{p}_i's knowledge \textit{p}_j is not faulty and ii) \textit{p}_i has not yet decided on a value for \textit{CON}_i, then \textit{p}_i makes the same decision as \textit{p}_j. In A2, upon detecting a process's failure, if process \textit{p}_i is the one with smallest ID among nonfaulty processes, \textit{p}_i becomes the leader. Otherwise, \textit{p}_i reports to the process with ID equal to \min(P - F_i) which is likely to be the current leader. It is not too hard to see that at any time there is at most one nonfaulty process with leader_i set to true. Only a leader may receive a Check(F, v) message. Upon receiving such a message in A3, the leader records the detected faults \textit{F} and the publicized value \textit{v}. A4 indicates how and when the leader decides on and publicizes a value. After a process \textit{i} has recognized itself as the leader (i.e., processes 1, \ldots, \textit{i}-1 have been detected by \textit{p}_i to be faulty), if the failures of processes 1, \ldots, \textit{i}-1 are also known by every other nonfaulty process, then \textit{p}_i decides on a
value and publicizes it. The correctness of Protocol CONSENSUS is established in the following.

**Theorem 1** Protocol CONSENSUS solves the consensus problem even in the presence of any number of process failures.

**Proof.** It is sufficient to establish the progress and safety properties for the protocol: 1) every nonfaulty process eventually decides (i.e., assigns a value to $CON_i$), and 2) the decision values reached by nonfaulty processes are all equal.

To prove the first property, consider any execution of the protocol in which there are exactly $f$ processes fail, where $0 \leq f \leq n$. After the $f$ failures have all occurred, let $p_i$ be the surviving process with smallest ID number. By the assumption of fault detectability, every nonfaulty process eventually detects all the failures and recognizes $p_i$ as the leader. At the time, if every nonfaulty process has decided then we are done. Otherwise, suppose there is a nonfaulty process that has not yet decided. On detecting the last failure, each process reports to $p_i$ via a Check($F, v$) message, with $F$ containing the ID numbers of all failures (A2). After all these reports are received, $p_i$ executes A4 and publicizes its decision with Decide messages. By A1, every nonfaulty process will at latest decide upon receiving such a message. This establishes the progress property.

To prove the safety property, consider any execution of the protocol. At any time $t$ during the execution, a Decide message from $p_i$ to $p_j$ that has not been received or is being received is said to be valid if $p_j$ is nonfaulty and $i \notin F_j$. Also, call a not-yet-received Check($F, v$) message valid if $v \neq \bot$ and the destination is a nonfaulty process that has not decided. We establish the safety property by proving the following proposition: At any time $t$, all decided nonfaulty processes, let it be set $X$, as well as all valid Decide and Check messages, let it be set $Y$, carry the same decision value.

Only the following five events may possibly affect the set $X$ or $Y$ and thereby change the validity of the proposition: E1) when a (nonfaulty) process decides, E2) when a (valid) Decide message is sent, E3) when a (valid) Check message is sent, E4) when a (valid) Decide message is nullified; i.e., when a fault is detected or the destination process crashes, E5) when a (valid) Check message is nullified; i.e., when the destination process crashes or decides.

Initially, at time $0$, the proposition is obviously true, as no process has decided and no Decide or Check messages are in transit. Assume that immediately before an event $E$ the proposition holds. We show that the proposition holds after event $E$.

Let $d$ be the common value referred to in the proposition. Let $d$ be $\bot$ if $X = Y = \emptyset$.

E4 and E5 only reduce $X$ or $Y$ and obviously do not affect the proposition.

When E2 or E3 occurs, the decision value carried in the Decide or Check message must be $d$ and the proposition holds. Since the value they carried is from a value decided by a member in $X$. E1 may occur only in A1, A3, or A4. The first two cases obviously do not falsify the proposition as the process decides on the value carried by a member in $Y$.

In the case of A4, we claim that before event $E$ no nonfaulty process has decided and no valid Check or Decide message is in transit (i.e., $X = Y = \emptyset$). To see this, suppose that a process $j > i$ with $j \notin F_i$ has decided. (Recall that $p_i$ is the leader and all processes $p_x$ with $x < i$ are faulty.) The decision value of $p_j$ must come from a then-valid Decide message sent by some $p_x$, $1 \leq x < i$. Let $p_j$ decided at time $t_1$. We know $x \notin F_j$ at time $t_1$ by A1. Let $t_2$ be the time at which $p_j$ sent a Check message with $x \in F_j$ to $p_i$, which did happen because $p_i$ had received a Check($F, v$) message from $p_j$ that included $\{1, \ldots, i-1\}$ in the $F$ set. Clearly, $t_2 > t_1$. That means the decision value $v$ carried in the Check message was not $\bot$. So, $p_i$ would have made its decision upon receiving that Check message in A3, a contradiction. So there is no such decided nonfaulty process $p_j$ at the time $p_i$ decides on its $x_i$. This in turn implies that no valid Check messages are in transit. From the guarded condition of A4, there are no valid Decide messages in transit because all previous valid Decide
messages have been nullified. Therefore, \( d \) equals 1 before \( A4 \) and after \( A4 \) \( d \) becomes \( z \); and the proposition still holds.

\[ \square \]

4 Necessary/Sufficient Conditions for TD

Termination detection was recently claimed to be 1-solvable [17] if the underlying communication network satisfies the following conditions: a) all messages sent from a process to another process are delivered without errors, with a finite but arbitrary delay, and in a FIFO fashion; b) should a processor fail, each nonfaulty process knows of the failure within a finite amount of time; c) a message sent to a failed processor is returned to the sender; d) each processor is able to send two or more messages atomically.

Now, as shown in the last section, the consensus problem is \( n \)-solvable if the underlying network satisfies conditions a and b. In view that termination detection is nothing but reaching a special kind of consensus, one might expect termination detection to be \( n \)-solvable or at least 1-solvable under conditions a and b (without c and d). Unfortunately, that is not the case. As explained in the following, termination detection is much harder than consensus, so hard that even the above four conditions together are not sufficient. As a result, the protocol described in [17] needs a stronger condition in order to work correctly.

Assume protocol \( \beta \) 1-solves TD under the mentioned conditions. Construct an underlying computation \( c \) in which process \( p_i \) fails at time \( t_i \), and the system terminates at time \( t' > t_i \). As protocol \( \beta \) solves \( T D^1 \), it eventually detects the termination at some time \( t > t' \). Construct another computation \( c' \) which is identical to \( c \) except that in \( c' \) \( p_i \) sends a basic message \( m \) to \( p_j \) immediately before it fails and the message arrives at its destination far after \( t \).

To see why the aforementioned four conditions are not sufficient to solve the problem of termination detection, we observe that under conditions a–d no other process except the faulty \( p_i \) has the capability of distinguishing \( c \) from \( c' \) (i.e., the knowledge concerning the existence of message \( m \)), until message \( m \) has arrived at its destination. One may even try to use the combination of conditions a) and c) to check the existence of \( m \). For example, after another \( p_j \) detects \( p_i \) to be faulty, \( p_j \) sends an \( m' \) and waits it to bounce back. However, a close examination reveals that the ordering between a sent message (i.e., \( m \)) and a bounced back message (i.e., \( m' \)) from a faulty process is not specified by the FIFO assumption. In fact, [17] used these conditions and has presumably implied the ordering between a sent message and a bounced back message.

Since protocol \( \beta \) declares \( c \) to be terminated at time \( t \), as no process other than the faulty one knows of the existence of \( m \), it will, by mistake, declare \( c' \) to be terminated at the same time. Thus, conditions a–d are not sufficient to make protocol \( \beta \) correct.

As a result, the above conditions are not sufficient because they are not strong enough for one to tell whether the channel from a faulty process to a nonfaulty process is empty. In a previous paper [13], we developed an \((n-1)\)-resilient termination detector that works correctly if the underlying communication network satisfies these conditions:

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

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

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

**S4: Undeliverable Messages.** A message which is not a return-flush and whose destination process has failed is said to be undeliverable. The network discards all undeliverable messages.
Some comments about these conditions are in order. First, the concept of return-flush is inspired by that of forward-flush [1]: If a message \( m \) is sent as a forward flush, every message sent before it to the same destination is guaranteed to be received before \( m \). The return-flush primitive can be implemented using the technique of [1]. Second, S4 is not crucial. If the underlying network chooses to return all undeliverable messages back to their senders, our algorithm will still work well. However, in that case, a return-flush \( m \) from \( p \) to \( q \) must be implemented in a way such that it can "flush" the channel between \( p \) and \( q \) in both directions.

S1 is perhaps an undesirable condition, being difficult to achieve. Unfortunately, as proved in the following, it is a necessary condition for termination detection to be 1-solvable.

**Theorem 2** Assume that message delay is arbitrary. Then, \( TD^1 \) is solvable only if faults are detectable.

**Proof.** The proof is by contradiction. Assume faults are not detectable but there is a protocol, say \( \beta \), that solves \( TD^1 \). Construct a computation \( c \) in which an active process \( p_i \) fails at time \( t_i \) and the computation terminates at \( t' \) after \( t_i \). As protocol \( \beta \) solves \( TD^1 \), it eventually detects the termination at some time \( t > t' \).

Construct another computation \( c' \) that is identical to \( c \) except that \( p_i \) does not fail at \( t_i \) but stays active all the time after \( t_i \), even after \( t \). Let \( p_i \) send no basic message after \( t_i \), and let all control messages sent by \( p_i \) after \( t_i \), if any, be delayed until after time \( t \).

As faults are not detectable, the protocol cannot distinguish between \( c \) and \( c' \) and, therefore, will declare \( c' \) to be terminated at time \( t \), which, of course, is not correct.

Not only fault detectability is necessary for fault-tolerant termination detection, but also is return flush. We do not mean the exact mechanism as stated in S3, but what is achieved by that mechanism: if process \( p \) knows process \( q \) has failed, then there will be a time \( t \) at which \( p \) knows that there are no messages in transit in the channel from \( q \) to \( p \). Let such a mechanism be called as *message fail-flush*. We prove in the following theorem that message fail-flush is necessary for termination detection to be solvable.

**Theorem 3** \( TD^1 \) is solvable only if the system provides message fail-flush capability.

**Proof.** Assume protocol \( \beta \) 1-solves \( TD \) without the fail-flush capability. That is, when a process \( p_i \) fails, process \( p_j \) has no way to know when the channel from \( p_i \) to \( p_j \) is empty.

Construct an underlying computation \( c \) in which process \( p_i \) fails at time \( t_i \), and the system terminates at time \( t' \). As protocol \( \beta \) solves \( TD^1 \), it eventually detects the termination at some time \( t > t' \). Construct another computation \( c' \) which is identical to \( c \) except that in \( c' \), \( p_i \) sends a basic message \( m \) to \( p_j \) immediately before it fails and the message arrives at its destination far after \( t \).

The existence of \( m \) was known to only \( p_j \), which unfortunately has died; all other processes will not know nothing about it until after time \( t \) when the message eventually arrives at its destination. Until then, protocol \( \beta \) cannot distinguish between \( c' \) and \( c \). Since protocol \( \beta \) declares \( c \) to be terminated at time \( t \), it will, by mistake, declare \( c' \) to be terminated at the same time. The protocol is thus incorrect.

\[
\square
\]

5 TD Is Harder than Consensus

The last two sections showed the existence of an environment in which consensus is \( n \)-solvable but termination is not even 1-solvable. That does not necessarily mean the latter is harder than the former unless one can show that under any condition, termination detection is never easier than consensus. In this section, we show that is indeed the case, in the sense that consensus can be reduced to termination detection and, therefore, consensus is \( k \)-solvable whenever termination detection is \( k \)-solvable.

**Theorem 4** \( CON^k \leq TD^k \) for any \( k, 1 \leq k \leq n \).
A1: (When there is an incoming message)

\[ \text{MsgCount}_i := \text{MsgCount}_i + 1; \]

consume the message;
randomly select
a (possibly empty) set \( Q_i \subset P \);
send a message to each process in \( Q_i \).

A2: (When \( \text{MsgCount}_i > 100 \))

send a message to each process in \( P \).

Figure 2: The underlying protocol \( \alpha \), for process \( p_i, 1 \leq i \leq n \).

Proof. Let \( \beta \) be any protocol that solves \( TD^k \). We shall construct a protocol \( \gamma \) that solves \( CON^k \) making use of \( \beta \). In particular, let \( P = \{p_1, p_2, \ldots, p_n\} \) be the set of \( n \) processes that want to reach a consensus. We shall let these processes run a distributed algorithm \( \alpha \) and simultaneously run \( \beta \) to decide if \( \alpha \) has terminated and to thereby reach a consensus.

The underlying system \( \alpha \) running on \( P \) consists of two event-driven actions. An action is enabled if the guarded condition is satisfied. A process is active if there are enabled actions, and idle otherwise. Each process \( p_i \) has a message counter \( \text{MsgCount}_i \) to record the number of messages \( p_i \) has received. Initially, every process has an incoming message (i.e., initially every process is active) and then keeps running the protocol \( \alpha \) in Figure 2.

The main idea of this protocol is that as long as \( \text{MsgCount}_i \leq 100 \), \( p_i \) can be either active or idle. However, should \( \text{MsgCount}_i \) ever reach the value 101, \( p_i \) will stay active forever and keep sending messages to other processes. Protocol \( \alpha \) does not take any input and whether it will terminate or not is nondeterministic. In each execution of \( \alpha \), there are only two possible outcomes: either the execution terminates with \( \text{MsgCount}_i \leq 100 \) for all non-faulty processes \( p_i \), or the execution will not terminate.

This property provides an easy way for all the nonfaulty processes to reach a consensus. One simply runs the termination detector \( \beta \) to check

1. Run \( \beta \) with \( \alpha \) as the underlying system until \( TD_i = 1 \) or \( \text{MsgCount}_i > 100 \).

2. If \( TD_i = 1 \) then let \( CON_i := 1 \); otherwise, let \( CON_i := 0 \).

Figure 3: Protocol \( \gamma \) for process \( p_i, 1 \leq i \leq n \):

whether \( \alpha \) is terminated. Protocol \( \gamma \) is presented in Figure 3.

In the protocol, \( TD_i \) is the output register of \( p_i \) running \( \beta \) (\( TD_i = 1 \) iff \( p_i \) knows the underlying system—in this case, \( \alpha \)—is terminated), and \( CON_i \) is the output register of process \( p_i \) running protocol \( \gamma \).

It is obvious that Protocol \( \gamma \) is a \( k \)-resilient consensus protocol as long as \( \beta \) is a \( k \)-resilient termination detector.

\[ \Box \]

6 Conclusions

It has been believed in the literature that as consensus is impossible in the presence of faulty processes, so is termination detection. In this paper, we confirmed this belief and showed that termination detection is actually harder than consensus.

We also showed that the ability to detect faulty processes plays an important role in both consensus and termination detection problems. This feature is sufficient for the consensus problem to be \( n \)-solvable, and it is weaker than previously known conditions.

While fault detectability is necessary for termination detection to be \( 1 \)-solvable, it alone is not sufficient. The return-flush condition is also important for termination detection to be \( 1 \)-solvable.

Recently, a stronger version of the consensus problem, called agreement consensus, has been shown to be harder than the regular consensus problem \([3]\). It would be interesting to investigate this problem's relationship with termination detection, as well as its relationship with fault detectability.
References


