Adaptive timeliness of Consensus in Presence of Crash and Timing Faults

Taisuke Izumi\textsuperscript{a,*} Akinori Saitoh\textsuperscript{b} Toshimitsu Masuzawa\textsuperscript{a}

\textsuperscript{a}Graduate School of Information Science and Technology, Osaka University, 1-3 Machikaneyama, Toyonaka, Osaka, 560-8531, Japan. Tel: +81-6-6850-6584, Fax: +81-6-6850-6582.

\textsuperscript{b}Faculty of Environmental and Information Studies, Tottori University of Environmental Studies, 1-1-1, Wakabadai-Kita, Tottori, 689-1111, Japan.

Abstract

The $\Delta$-timed uniform consensus is a stronger variant of the traditional consensus and it satisfies the following additional property: Every correct process terminates its execution within a constant time $\Delta$ ($\Delta$-timeliness), and no two processes decide differently (Uniformity). In this paper, we consider the $\Delta$-timed uniform consensus problem in presence of $f_c$ crash processes and $f_t$ timing-faulty processes, and propose a $\Delta$-timed uniform consensus algorithm. The proposed algorithm is adaptive in the following sense: It solves the $\Delta$-timed uniform consensus when at least $f_t + 1$ correct processes exist in the system. If the system has less than $f_t + 1$ correct processes, the algorithm cannot solve the $\Delta$-timed uniform consensus. However, as long as $f_t + 1$ processes are non-crashed, the algorithm solves (non-timed) uniform consensus. We also investigate the maximum number of faulty processes that can be tolerated. We show that any $\Delta$-timed uniform consensus algorithm tolerating up to $f_t$ timing-faulty processes requires that the system has at least $f_t + 1$ correct processes. This impossibility result implies that the proposed algorithm attains the maximum resilience about the number of faulty processes. We also show that any $\Delta$-timed uniform consensus algorithm tolerating up to $f_t$ timing-faulty processes cannot solve the (non-timed) uniform consensus when the system has less than $f_t + 1$ non-crashed processes. This impossibility result implies that our algorithm attains the maximum adaptiveness.

Key words: distributed algorithm, timeliness property, consensus problem, crash fault, timing fault

* Corresponding author.

Email addresses: t-izumi@ist.osaka-u.ac.jp (Taisuke Izumi), saitoh@kankyo-u.ac.jp (Akinori Saitoh), masuzawa@ist.osaka-u.ac.jp (Toshimitsu Masuzawa).

Preprint submitted to Journal of Parallel and Distributed Computing22 February 2011
1 Introduction

The consensus problem is a fundamental and important problem for designing fault-tolerant distributed systems. In consensus algorithm, each process proposes a value, and all non-faulty processes have to agree on a common value that is proposed by a process. Despite of its applications (e.g., atomic broadcast [1–3], shared object [4,5], weak atomic commitment [6] and so on), it has no deterministic solution in asynchronous systems subject to only a single crash fault [7]. Thus, several consensus algorithms have been considered on systems with some assumptions [8–11,12–16]. Synchrony is one of the most commonly used assumptions for designing consensus algorithms. It is usually defined as the assumption that communication delay is bounded by some constant. This synchrony assumption also arouses an interest in the possibility of $\Delta$-timed consensus, where each correct process terminates its execution within a constant time $\Delta$ ($\Delta$-timeliness). The $\Delta$-timed consensus is not only particularly important for real-time systems, but also desirable for non-real-time systems since the execution time $\Delta$ strongly affects the overall performance.

However, because of the essential unpredictability of distributed systems, the synchrony assumption is occasionally violated in real systems, by overload of processes, message congestion, and so on. Such violation brings the timing-fault into synchronous systems. The timing-faults may prevent algorithms designed in the synchronous model from working correctly in real distributed systems. Even when the algorithms work regardless of the timing-faults, timing-faults may cause a significant slowdown of the entire system and the timeliness of the $\Delta$-timed consensus may be violated. Therefore, robustness for timing-faults is strongly desired in the $\Delta$-timed consensus. Moreover, robustness for timing-faults has another key advantage in real distributed systems. Many distributed systems have design parameters relevant to the upper bound of communication delay. For example, timeout detection is based on the parameter, and the parameter value is usually overestimated to avoid unnecessary timeout. However such overestimation sometimes deteriorates the overall performance of distributed systems. Robustness for timing-faults allows to estimate the upper bound more tightly. Thus, timing-fault tolerance can drastically improve the performance of timeout detection mechanisms.

In this paper, we consider process timing faults that cause overdelay on messages sent or received by the faulty processes, and investigate the possibility of the $\Delta$-timed uniform consensus. The uniform consensus is a stronger variant of the consensus such that faulty processes are disallowed to disagree (Uniformity). The system suffers both timing-faults and crashes, and has the respective upper bounds $f_c$ and $f_t$ for the numbers of crash faults and timing-faults. For the relevance of such a mixed-fault model, the following criticism
may occur: A Crash fault is a special case of timing-faults because a crash-faulty process can be regarded as the timing-faulty process that only sends the message with infinite communication delay, and thus, crash-faults and timing-faults can be unified. However, this unification hides the essential difference between crash-faults and timing-faults, which is that processes “eventually” receive messages from timing-faulty processes (as long as those messages are sent), whereas it never receives messages from crashed processes. Thus, the unification of crash faults and timing-faults weakens the ability of the model. In other words, there may be a case that \( f_c \) crash faults and \( f_t \) timing-faults can be tolerated, but \( f_t + f_c \) crash/timing-faults cannot be tolerated. Actually, this paper succeeds in showing the difference.

This paper presents a \( \Delta \)-timed uniform consensus algorithm tolerating \( f_c \) crash processes and \( f_t \) timing-faulty processes. The algorithm solves the \( \Delta \)-timed uniform consensus when at least \( f_t + 1 \) processes are correct (Condition C1). Moreover, the proposed algorithm has a remarkable property, called adaptiveness. Informally, the timed algorithm \( A \) is adaptive, if \( A \) is not only a \( \Delta \)-timed algorithm under the certain condition \( C \) about the number of faulty processes, but also a non-timed algorithm under some relaxed condition \( C' \). Precisely, the proposed algorithm realizes the \( \Delta \)-timed uniform consensus under the condition C1. If the condition C1 is violated, the algorithm cannot realize \( \Delta \)-timed uniform consensus. However, as long as the system has at least \( f_t + 1 \) non-crashed processes (Condition C2), the algorithm realizes the (non-timed) uniform consensus. In general, since the number of faulty processes dynamically changes, this adaptiveness is a favorable property.

We also investigate the necessary condition on the number of faulty processes where the \( \Delta \)-timed/non-timed uniform consensus can be realized. We show that no \( \Delta \)-timed uniform consensus algorithm can tolerate \( f_t \) timing-faulty processes if the condition C1 does not holds. Moreover, we show that any \( \Delta \)-timed algorithm tolerating up to \( f_t \) timing-faulty processes cannot solve the (non-timed) uniform consensus if the condition C2 is violated. These results imply that our algorithm attains the maximum resilience and adaptiveness.

The roadmap of this paper is as follows: In section 2, we mention previous works related to this paper. In section 3, we present definitions of the system model, the \( \Delta \)-timed uniform consensus problem, and the adaptive timed algorithm. Section 4 provides the adaptive \( \Delta \)-timed uniform consensus algorithm. We show the impossibility results in Section 5, and conclude this paper in Section 6.
2 Related Work

Since there are many literatures studying about the consensus problem in synchronous or asynchronous systems, we cite here only the closely related results. The $\Delta$-timed consensus is implicitly studied as the consensus in synchronous systems [8,9,11]. Some papers have described consensus algorithms in synchronous systems subject to a certain class of faults [9,11]. The brief survey of the synchronous consensus is given by Raynal [17]. However, in those works, there are few papers investigating timing-faults: Lima et al. assume the model where concurrent messages are scheduled by assigned priorities and the message with highest priority is guaranteed to be timely, and propose a timed consensus algorithm in that model [18]. Aguilera et al. propose “expiring link” technique [19], which is a generic way to reduce timing faults to omission faults. Combining this technique and existing omission-fault-tolerant algorithms [20,21,11], timing-fault-tolerant algorithms can be constructed. The series of papers by Casimiro et al. propose a generic scheme of handling timing-faults in the environment where the timing-faults are detectable [22,23]. However, all of the above solutions do not consider the adaptiveness. The adaptive timeliness property proposed in this paper definitely differentiates our results from existing ones. In addition, some of them cannot achieve the optimal fault resilience that is presented in this paper.

The key idea of our algorithm is mainly derived from consensus algorithms using the strong failure detector [1,12–14]. Additionally, the failure suspicion scheme proposed in this paper is inspired by the idea of Gafni and Mostefaoui et al. [24,25] (but different in many other points, especially, concerning time bound or not).

Adaptiveness concerning timeliness is first introduced in this paper. Other types of adaptiveness are investigated in several works. For example, Paxos algorithm [26,27], an asynchronous consensus algorithm, always ensures the safety property, and it ensures the liveness property in good conditions.

From another aspect, our model can be regarded as an intermediate model between asynchronous systems and synchronous ones. Some papers present such intermediate models and address the solvability of the non-timed consensus. For example, the partially synchronous systems [10] and the timed asynchronous systems [28] are proposed. Moreover, Aguilera et al. proposed several weaker synchronous system models where the consensus problem can be solved [29,30]. One of these models is strictly weaker than our model. Nevertheless, as we mentioned, all the above results only consider the non-timed consensus, and the $\Delta$-timeliness property is out of their interests.
3 Preliminaries

3.1 Distributed System

We consider a synchronous distributed message-passing system consisting of \( n \) processes \( P = \{p_0, p_1, p_2, \ldots, p_{n-1}\} \), in which any pair of processes can communicate with each other by exchanging messages. All channels are reliable: each channel correctly transfers messages. Processes can crash and can become timing-faulty. The system is synchronous in the sense that all the messages transmitted between non-timing-faulty processes have communication delays of \( d \) or less. We assume every process knows the constant \( d \) a priori. In addition, we assume that each process has a timer and can set the timer to raise an alarm after the preset time interval. In the following subsections, we describe the behavior of the system in more details.

3.1.1 Processes

A process is modeled as a state machine, and changes its own state when an event occurs. Local processing time is negligible, that is, state transition occurs instantaneously. Processes are subject to crash and timing faults. These faults are modeled as particular states of processes. In the system model, a state of a process is defined as a pair of \((s, f)\), where \(s\) is the state of process itself, and \(f\) is the fault state. The fault state can be “correct”, “crashed”, or “timing-faulty”. According to its fault state, each process works as follows.

- When the fault state is “correct” or “timing-faulty”, the process works correctly according to its state transition function. However, a correct process and a timing-faulty process have difference in communication delay: the delay of message transfer from or to a timing-faulty process may exceed the upper bound \( d \). The difference is formally specified later. The fault state can change to “crashed” on the occurrence of a crash event (this implies the process crashes). The crash event can occur at any time.
- When the fault state is “crashed”, the process makes no operation. Once the fault state becomes “crashed”, it remains “crashed” forever.

Without loss of generality, we can assume that the fault state never changes from “correct” to “timing-faulty”. In other words, we regard the process that suffers from the timing-fault as being “timing-faulty” from the beginning. Therefore, each process has one of following fault patterns.

1. Fault state is always correct.
2. Fault state is initially correct, and changes to crashed during execution.
3. Fault state is always timing-faulty.
(4) Fault state is initially timing-faulty, and changes to crashed during execution.
(5) Fault state is always crashed.

Afterward, we say that process $p_i$ is correct if $p_i$ has fault pattern 1. We also say that process $p_i$ is non-timing-faulty if $p_i$ has fault pattern 1 or 2. Intuitively, a correct process is one that is correct and never crashes, and a non-timing-faulty process is one that is correct but possibly crash in the future. Moreover, we say that process $p_i$ is timing-faulty if $p_i$ has fault pattern 3 or 4. We give a transition diagram of fault state in Figure 1. Notice that we introduce the faulty state only to represent the system configuration, and we assume processes are unaware of their fault states: the same state transition can occur, whether its fault state is “correct” or “timing-faulty”.

There are upper bounds $f_c$ on the number of processes that can crash (that is, the number of processes whose fault patterns are pattern 2, 4 or 5) and $f_t$ on the number of processes that are timing-faulty (the processes whose fault patterns are pattern 3 or 4). We assume that every process knows the values of $f_c$ and $f_t$ a priori. Notice that a process with fault pattern 4 is counted as both a timing-faulty process and a crashed process. For simplicity, the constant $f$ denotes $\min\{f_t + f_c, n - 1\}$.

### 3.1.2 Timer

Each process has a timer. A Timer can be set for some interval by a primitive operation $\text{Timerset}$ to raise an event $\text{Alarm}$ after the interval. The timer of each non-timing-faulty process raises the alarm event at the exact time, unless the process crashes. However, the timer of each timing-faulty process can raise up the alarm at wrong time (but it necessarily raises up the alarm event unless it crashes).
3.1.3 Communication

Processes can communicate with each other by exchanging messages. A message can be sent by a primitive operation \textit{Send} and received by a primitive operation \textit{Receive}. We assume that the message passing is reliable as follows:

\textbf{Nonfaulty-Liveness} : If a non-timing-faulty process \( p_i \) sends a message \( m \) to a non-timing-faulty process \( p_k \) at \( t \) and both of them does not crash by \( t + d \), then \( p_k \) receives \( m \) at \( t + d \) or earlier.

\textbf{Faulty-Liveness} : If a process \( p_i \) sends a message \( m \) to a process \( p_k \) and does not crash, then \( p_k \) eventually receives \( m \) or eventually crashes.

\textbf{Uniform Integrity} : A message \( m \) can be received at most once, only if it is previously sent by some process.

As we mentioned in the previous subsection, this specification also defines the essential difference between a non-timing-faulty process and a timing-faulty process.

3.1.4 Configuration and Execution

A system configuration is represented by all processes’ states, messages under transmission, and a set of alarms which have been set but have not gone off. An execution of a distributed system is an alternative sequence of configurations and events \( E = c_0, e_1, c_1, e_2, c_2 \cdots \) such that occurrence of event \( e_i \) changes the configuration from \( c_{i-1} \) to \( c_i \). Each primitive \textit{Timerset}, \textit{Alarm}, \textit{Send} or \textit{Receive} is one of events. All events other than \textit{Alarm} and \textit{Receive} are called \textit{internal events}. In this paper, we deal with a timed execution \( E = c_0, (e_1, t_1), c_1, (e_2, t_2), \cdots, c_k, (e_{k+1}, t_{k+1}), \cdots \) where each event \( e_i \) is associated with global time \( t_i \) when the event occurs. The timed execution we consider satisfies the following conditions:

1. The times assigned to events are non-decreasing, that is \( t_{k-1} \leq t_k \) holds for any \( k \).
2. If \((e, t)\) is an event sending a message \( m \) from a non-timing-faulty process \( p_i \) to a non-timing-faulty process \( p_j \), then there exists an event \((e', t')\) such that \( t \leq t' \leq t + d \) holds and \( e' \) is \( p_j \)'s event receiving \( m \), \( p_i \)'s crash event, or \( p_j \)'s crash event.
3. If \((e, t)\) is an event sending a message \( m \) from a process \( p_i \) to a process \( p_j \), there exists event \((e', t')\) such that \( t \leq t' \) holds and \( e' \) is \( p_j \)'s event receiving \( m \), \( p_i \)'s crash event, or \( p_j \)'s crash event.
4. If \((e, t)\) is an event setting a timer for an interval \( \tau \) at a non-timing-faulty process \( p_i \), then there exists an event \((e', t')\) such that \( e' \) is \( p_i \)'s alarm event and \( t' = t + \tau \) holds, or \( e' \) is \( p_i \)'s crash event and \( t \leq t' \leq t + \tau \) holds.
5. If \((e, t)\) is an event setting a timer for an interval \( \tau \) at a process \( p_i \), then
there exists an event \((e', t')\) such that \(e'\) is \(p_i\)'s alarm event and \(t \leq t'\) holds, or \(e'\) is \(p_i\)'s crash event and \(t \leq t'\) holds.

(6) If \((e, t)\) is an internal, timer set or send event that is not the first event at correct process \(p_i\), then there exists a preceding event \((e', t')\) at \(p_i\) such that \(t = t'\) holds.

Conditions 2 and 3 imply that all messages are eventually received unless their sender or receiver crash. Condition 2 also implies that any message exchanged between non-timing-faulty processes experiences delay of at most \(d\). Conditions 4 and 5 respectively imply that the timer of each non-timing-faulty process works correctly, and the timer of each timing-faulty process can raise up the alarm event at any time. Condition 6 implies that processing time of local computation is negligible, that is, several internal and send events can be executed in an instant.

**NOTICE** Several previous works define the process timing fault as the fault causing a slowdown in local processing of a process, whereas we define the timing fault as the fault causing a slowdown in message delay and the timer. Clearly, our “timing-faulty” concept essentially involves the slowdown in local processing, and thus, the algorithm proposed in this paper can be easily modified to work correctly in the environment where local processing of timing-faulty processes is slowed down.

### 3.2 \(\Delta\)-timed Uniform Consensus

In a consensus algorithm, each correct process initially proposes a value, and eventually chooses a decision value from the values proposed by processes so that all processes decide the same value. The uniform consensus is a stronger variant of the consensus. It disallows faulty processes to disagree on the decided value. The uniform consensus algorithm provides two primitives, \(\text{propose}_i(v)\) and \(\text{decide}_i(v)\), to the upper application layer. The event \(\text{propose}_i(v)\) is invoked by the upper application to propose the value \(v\), and \(\text{decide}_i(v)\) is invoked by the algorithm to notify the decision value \(v\). More precisely, the uniform consensus is specified as follows:

**Termination** : If a process \(p_i\) invokes \(\text{propose}_i(v)\), it eventually invokes \(\text{decide}_i(v')\) for some value \(v'\) or eventually crashes.

**Uniform Agreement** : If \(\text{decide}_i(v_i)\) and \(\text{decide}_j(v_j)\) are invoked, \(v_i = v_j\) holds.

**Uniform Validity** : If a process \(p_i\) invokes \(\text{decide}_i(v)\), then \(\text{propose}_j(v)\) is previously invoked by a process \(p_j\)

The \(\Delta\)-timed uniform consensus is the uniform consensus ensuring any non-timing-faulty process \(p_i\) invokes \(\text{decide}_i(v')\) within \(\Delta\) time after the invocation
of $\text{propose}_i(v)$ unless it crashes. Formally, in addition to the Termination, Uniform Agreement, and Uniform Validity, the $\Delta$-timed uniform consensus satisfies the following specification:

$\Delta$-timed Termination: If a non-timing-faulty process $p_i$ invokes $\text{propose}_i(v)$ at $t$, it invokes $\text{decide}_i(v')$ at $t + \Delta$ or earlier, or crashes by $t + \Delta$.

In this paper, we consider both the $\Delta$-timed uniform consensus and the (non-timed) uniform consensus.

3.3 Adaptive Timed Algorithm

In this subsection, we introduce the adaptiveness of a timed algorithm. We assume that the considered problem has both the non-timed specification and the $\Delta$-timed specification. Let $A$ be a $\Delta$-timed algorithm under some condition $C$, that is, $A$ solves a problem in $\Delta$ time when the condition $C$ is satisfied. Then, the $\Delta$-timed algorithm $A$ is adaptive, if $A$ solves the problem but does not guarantee the $\Delta$-timeliness under some relaxed condition $C'$. Typically, $C$ and $C'$ are conditions on the number of faulty processes. In this paper, the following two conditions are considered:

**Condition C1** At least $f_i + 1$ correct processes exist.

**Condition C2** At least $f_i + 1$ processes never crash.

It is obvious that Condition C2 is a relaxed condition of C1, that is, Condition C2 holds if Condition C1 holds.

4 Adaptive $\Delta$-timed Uniform Consensus Algorithm

In this section, we present the $\Delta$-timed uniform consensus algorithm ATC. It realizes the $\Delta$-timed uniform consensus when condition C1 is satisfied and realizes the non-timed uniform consensus when condition C2 is satisfied. This implies the algorithm is an adaptive $\Delta$-timed uniform consensus algorithm.

4.1 Overview

The algorithm ATC is based on the rotating coordinator (RC) paradigm [31,13,12]. The typical framework of the RC-based consensus algorithms is as follows: Each process proceeds in consecutive rounds, and maintains the
current estimation of the decision value. Each round $r\ (r = 0, 1, 2, \cdots)$ is managed by the coordinator $p_{r \mod n}$. At the round $r$, the algorithm tries to update the current estimations of all non-crashed processes by the coordinator’s estimation. This trial may fail, however, the algorithm guarantees that the trial necessarily succeeds at some round. The algorithm ATC also follows this framework. However, in the way how each estimation is updated, the algorithm ATC is crucially different from the previous RC-based algorithms: In the previous algorithms, at each round, the coordinator actively tries to impose its current estimate on other processes by broadcasting its current estimate. By contrast, in the algorithm ATC, the coordinator is passive. Each process queries the coordinator about its estimation. The coordinator only replies in answer to those queries. This modification is introduced to guarantee the $\Delta$-timed Termination: In the previous active coordination scheme, a round can be unboundedly long, if the coordinator is timing-faulty. On the other hand, our passive coordination scheme allows non-timing-faulty processes to avoid waiting for the timing-faulty coordinator. If a non-timing-faulty process cannot receive the reply from the coordinator within a constant time, it can regard the trial failed and proceed to the next round.

The simplest method to transmit queries and replies is the round-trip communication, that is, each process directly sends the query to the coordinator, and the coordinator directly sends back the reply upon reception of the query. However, this method has a critical problem: a timing-faulty process may cause the timeout at every round and never succeeds in receiving the reply, and thus, the process may fail to have the same decision value as other processes. This violates the uniform agreement property.

In order to resolve this problem, we introduce a novel communication service, *Round-trip Communication with Fault Suspicion* (RCFS) to transmit the queries and the replies. By RCFS, a non-crashed process eventually receives the reply or suspects fault of the coordinator. The important characteristic of RCFS is that the coordinator is never suspected if it is correct. This guarantees that all the non-crashed processes eventually receive the reply at the round where the coordinator is correct.

In the following subsection, we introduce the RCFS algorithm RCFS. When the condition C2 holds, the algorithm RCFS realizes the RCFS. In addition, the algorithm RCFS is adaptive. To be more precise, when the condition C1 is satisfied, the algorithm RCFS realizes the $\Delta$-timed RCFS, that is, a non-timing-faulty process completes the execution (or crashes) within a constant time from the activation of the algorithm. The adaptiveness of the $\Delta$-timed consensus algorithm ATC is essentially derived from this adaptiveness of RCFS.
4.2 Round-trip Communication with Fault Suspicion

Specification

The RCFS service provides four primitives, \( \text{Call}_i(p_j, v) \), \( \text{Called}_j(p_i, v) \), \( \text{Respond}_j(p_i, v') \) and \( \text{Responded}_i(p_j, v'') \). The primitive \( \text{Call}_i(p_j, v) \) is invoked by process \( p_i \) to activate the RCFS algorithm, where \( p_i \) tries to send message \( v \) to \( p_j \). If the algorithm succeeds in the transmission of \( v \), \( \text{Called}_j(p_i, v) \) occurs on \( p_j \) (if fails, it never occurs). Then, \( p_j \) invokes \( \text{Respond}_j(p_i, v') \), where \( v' \) is the reply message. The primitive \( \text{Responded}_i(p_j, v'') \) occurs on \( p_i \) to deliver the reply message or notify the suspicion. If \( p_j \) is suspected, then \( v'' \) is SUSPECTED. Otherwise, \( v'' \) is the reply message. Notice that, regardless of success or fail of the transmission, \( \text{Responded}_i(p_j, v'') \) eventually occurs unless \( p_i \) crashes. Formally, the RCFS algorithm satisfies the following specifications:

1. **Response-Liveness**: If a process \( p_i \) invokes \( \text{Call}_i(p_j, v) \), then \( \text{Responded}_i(p_j, v'') \) eventually occurs unless \( p_i \) crashes.
2. **Call-Liveness**: If a process \( p_i \) invokes \( \text{Call}_i(p_j, v) \) and \( p_j \) is correct, then \( \text{Called}_j(p_i, v) \) eventually occurs.
3. **Call-Integrity**: \( \text{Called}_j(p_i, v) \) occurs only when \( \text{Call}_i(p_j, v) \) occurs.
4. **Response-Integrity**: \( \text{Responded}_i(p_j, v) \) occurs for \( v \neq \text{SUSPECTED} \), only when \( p_j \) invokes \( \text{Respond}_j(p_i, v') \).
5. **Correct-Suspicion**: \( \text{Responded}_i(p_j, \text{SUSPECTED}) \) occurs only when \( p_j \) is timing-faulty or crashed.
6. **At-Once-Invocation**: \( \text{Called}_j(p_i, \ast), \text{Respond}_j(p_i, \ast), \) and \( \text{Responded}_i(p_j, \ast) \) respectively occur at most once for one invocation of \( \text{Call}_i(p_j, \ast) \).

The RCFS algorithm requires the following assumption to the application layer:

1. **Immediate Response**: If \( \text{Called}_j(p_i, v) \) occurs at time \( t \) and process \( p_j \) is non-timing-faulty, then \( p_j \) invokes \( \text{Respond}_j(p_i, v') \) at \( t \), or crashes at \( t \).

Moreover, we define the timed variant of RCFS. The \( \Delta \)-timed RCFS satisfies the following property in addition to Faulty-Liveness, At-Once-Invocation, and the three agreement properties.

1. **\( \Delta \)-Timed-Response-Liveness**: If a non-timing-faulty process \( p_i \) invokes \( \text{Call}_i(p_j, v) \) at time \( t \) and does not crash by \( t + \Delta \), then \( p_i \) invokes \( \text{Responded}_i(p_j, v') \) at \( t + \Delta \) or earlier.

In what follows, we call the process invoking \( \text{Call}_i(p_j, v) \) the **caller**, and then call the process \( p_j \) the **responder**.
Algorithm RCFS

Figure 2 presents algorithm RCFS in the event driven style: Each transition is represented by a triggering event followed by its handler. If two triggering events occur at the same time, the transition preceding in the description is executed first. Notice that the caller and the responder respectively have the additional transition functions (respectively lines 6-12 and 13-19).

The key idea behind our implementation is to multiplex the simple round-trip communication. More precisely, when the caller process \( p_i \) invokes \( \text{Call}_i(p_j, v) \), it first sends the message “call1” containing \( v \) to all processes. When each process \( p_k \) receives the message call1 (in what follows, we call those processes relayers *), \( p_k \) tries the round-trip communication to \( p_j \): The relayer \( p_k \) sends the message “call2” containing \( v \) to \( p_j \). When responder \( p_j \) receives the message call2 first, it invokes \( \text{Called}_j(p_i, v) \) and immediately invokes \( \text{Respond}_j(p_i, v') \). When \( \text{Respond}_j(p_i, v') \) occurs, \( p_j \) stores the value of \( v' \) into the variable \( \text{reply} \), and sends the message “response1”. Hereafter, the responder only sends the response1 messages in answer to the call2 messages. If relayer \( p_k \) receives the message response1 within 2\( d \) time after sending call2, it sends the response2 message to the caller \( p_i \). Otherwise (that is, if timeout occurs), \( p_k \) sends the SUSPECTED message to \( p_i \). The caller \( p_i \) invokes \( \text{Responded}_i(p_j, v') \) when (1) \( p_i \) receives one response2 message, or (2) receives \( f_t + 1 \) SUSPECTED messages. Intuitively, the correctness of the algorithm is understood as follows: When the caller receives \( f_t + 1 \) SUSPECTED messages, the caller receives at least one SUSPECTED message from a non-timing-faulty relayer. However, if the responder \( p_j \) is correct, the non-timing-faulty relayer never suspects \( p_j \). This implies that the correct responder \( p_j \) is never suspected. To guarantee the liveness properties, this algorithm clearly requires at least \( f_t + 1 \) non-crashed processes. Moreover, if the system has \( f_t + 1 \) correct processes, the non-timing-faulty caller receives \( f_t + 1 \) response2 messages within 4\( d \) time from the occurrence of \( \text{Call}_i(*,*) \). Therefore, the following theorems hold.

**Theorem 1** The algorithm RCFS realizes the RCFS when Condition C2 is satisfied.

**Proof.** Let \( p_i \) and \( p_j \) be a caller and its corresponding responder respectively. (1) **Response-Liveness:** From the condition C2, at least \( f_t + 1 \) non-crashed processes exist. Each of them receives a call1 message and eventually either receives a response1 message or invokes \( \text{Alarm}_i(\text{timeout2}) \). In both cases, they send response2 messages to \( p_i \). This implies that \( p_i \) eventually receives at least \( f_t + 1 \) response2 messages. That is, \( p_i \) eventually invokes \( \text{Responded}_i(p_j, v') \).

* Notice that the caller and the responder are also relayers.

† In Figure 2, the SUSPECTED message is described as the response2 message whose carrying value is SUSPECTED.
(2) **Call-Liveness**: Since \( p_j \) is correct, if a caller \( p_i \) invokes \( Call_i(p_j, v) \), \( p_j \) eventually receives a call1 message, and sends a call2 message. Therefore, it also eventually receives call2 message with \( v \) from \( p_j \). This implies that \( p_j \) eventually invokes \( Called_j(p_i, v) \). (3) **Call-Integrity, Response-Integrity, and At-Once-Invocation**: They clearly hold. (4) **Correct-Suspicion**: From Response-Liveness and Response-Integrity, if a responder invokes \( Respond_j(p_i, v) \), \( p_i \) invokes \( Responded_i(p_j, v') \) for either \( v' = v \) or \( v' = \text{SUSPECTED} \). Therefore, we prove this property by showing that \( v' \neq \text{SUSPECTED} \) if \( p_j \) is correct. Suppose for contradiction that the value of \( v' \) is \( \text{SUSPECTED} \). Then, the process \( p_i \) receives \( f + 1 \) \( \text{SUSPECTED} \) messages. This implies that in the senders of those messages, there exists at least one non-timing-faulty process because at most \( f \) processes are timing-faulty. Let \( p_k \) be this correct process. Since \( p_j \) and \( p_k \) are correct, the call2 message from \( p_k \) and the response1 message from \( p_j \) is transferred within \( d \) time units. Therefore, from Immediate Response assumption, \( p_k \) necessarily receives the call2 message before the occurrence of \( Alarm_k(\text{timeout2}) \). This contradicts the fact that \( p_k \) sends a \( \text{SUSPECTED} \) message. 

**Theorem 2.** The algorithm \( \text{RCFS} \) realizes the \( 4d \)-timed \( \text{RCFS} \) when Condition C1 is satisfied.

**PROOF.** Since the Condition C1 is stronger than the condition C2, Theorem 1 also holds in this case. Thus, we only have to prove \( 4d \)-Timed-Response-Liveness property. Let \( t \) be the time when the caller \( p_i \) invokes \( Call_i(p_j, v) \). Then, each correct relayer receives a call1 message by \( t + d \) because we assume that \( p_i \) is correct. Moreover, from the condition C2, the number of such correct relayers is at least \( f + 1 \). Then, those correct relayers send response2 messages by \( t + 3d \), and thus the caller receives at least \( f + 1 \) response2 messages by \( t + 4d \). This implies that the caller \( p_i \) invokes \( Responded_i(p_j, v') \) by \( t + 4d \). 

4.3 **Algorithm ATC**

Using the RCFS service, we realize the \( \Delta \)-timed uniform consensus algorithm \( \text{ATC} \) (Figure 3). The execution of each process consists of consecutive rounds, numbered by \( 0, 1, 2, \ldots, f \). Each process \( p_r \) acts as the coordinator in the round \( r \). The round number of the round each process \( p_i \) is executing is maintained by the local variable \( r_i \). The local variable \( est_i[r] \) represents \( p_i \)'s estimation of decision value at the end of round \( r \). For simplicity, \( est_i[-1] \) represents the value proposed by \( p_i \).

In the algorithm \( \text{ATC} \), each process works as follows:
1: : variable
2: : timeout, init FALSE
3: : reply : init ⊥
4: : proc : init ⊥
5: : ci : init 0

6: : transition function of caller process pi
7: : upon Calli(pj, v) do :
8: : sendi(call1, pi, pj, v) to all processes (including pi itself)
9: : upon Receivei(response2, pj, pi, v) from pk do :
10: : ci ← ci + 1
11: : if v ≠ SUSPECTED then Respondi(pj, v) ; exit
12: : else ci ≥ fi + 1 then Respondi(pj, SUSPECTED) endif

13: : transition function of responder process pj
14: : upon Receivej(call2, pi, pj, v) from pk do (pk may be pj itself):
15: : if reply = ⊥ then proc ← pk ; Calledj(pi, v)
16: : else sendj(response1, pi, pj, reply) to pk endif

17: : upon Respondj(pi, v) do :
18: : reply ← v
19: : sendj(response1, pj, pi, v) to proc

20: : transition function of each process pk (including the caller and the responder)
21: : upon Receivek(call1, pi, pj, v) from pi do :
22: : sendk(call2, pi, pj, v) to pj
23: : Timersetk(2d, timeout2)

24: : upon Receivek(response1, pi, pj, v) from pj do :
25: : if timeoutk ≠ TRUE then sendk(response2, pi, pj, v) to pk endif

26: : upon Alarmk(timeout2) do :
27: : timeoutk ← TRUE ; sendk(response2, pi, SUSPECTED) to pi

Fig. 2. Algorithm RCFS

- **Coordinator**: The primary work of the coordinator pi is to invoke Respondi(∗, esti[i]) for the occurrence of Calledi(∗, v). Notice that Calledi(∗, v) can occur even when pi is not at the round i, because each process can go on each round asynchronously. When Calledi(∗, v) occurs, the following two cases are considered: (1) the variable esti[i] stores some value, or (2) the variable esti[i] is empty. In the case of (1), the process pi simply invokes Respondi(∗, esti[i]). On the other hand, in the case of (2), the estimation of the round i is not available, and thus, the process pi adopts the value v as the estimation of the round i. Once the variable esti[i] is written, it is never overwritten. At the beginning of the round i, the coordinator pi writes the value of esti[i − 1] into esti[i], if the variable esti[i] is empty.

- **Non-coordinator**: At the beginning of round r ≠ i, each process pi invokes Calli(pj, esti[r − 1]) to know the coordinator’s estimation. When Respondi(pj, v) occurs, pi finishes the round r and writes into esti[r] as follows. If v = SUSPECTED, then pi’s estimation does not change, and thus, the value of esti[r − 1] is written into esti[r]. If v ≠ SUSPECTED, the value of v is written into esti[r].
PROOF. The proof is by induction of \( r' \). (Basis) In the case of \( r' = r \), since \( p_r \) is correct and the condition C2 is satisfied, Theorem 1 holds. Thus, from Response-Liveness and Correct-Suspicion, the process \( p_i \) and \( p_j \) respectively invoke \( \text{Response}(p_r, v_i) \) and \( \text{Response}(p_r, v_j) \) such that \( v_i = v_j \neq \bot \), that is, \( \text{est}_i[r'] = \text{est}_j[r'] \neq \bot \) holds. This implies that both \( \text{est}_i[r'] \) and \( \text{est}_j[r'] \) have a same proposed value because for any \( r' \) and \( p_i \), \( \text{est}_i[r'] \) stores only one of proposals or \( \bot \). (Inductive step) We assume as the induction hypothesis that the lemma holds for any \( r' \) such that \( r \leq r' < k \) and consider the case of \( r' = k \). By the induction hypothesis, all variables \( \text{est}_s[k - 1] \) have the same value \( v(\neq \bot) \) that is one of proposed values. Thus, we only have to prove that for any \( p_k \), the variable \( \text{est}_h[k] \) has the value that some process \( p_j \) stores in \( \text{est}_j[k - 1] (= v) \). If \( p_h = p_k \), that is \( p_k \) is the coordinator of round \( k \), then \( \text{est}_h[k] \) is written by the value of \( \text{est}_h[k - 1] \), or the value of \( \text{est}_s[k - 1] \). If \( p_h \) is not the coordinator, then \( \text{est}_h[k] \) is written by the value of \( \text{est}_h[k - 1] \), or the value of \( \text{est}_s[k - 1] \). In any case, \( \text{est}_h[k] = \text{est}_s[k - 1] = v(\neq \bot) \) holds. Therefore, this lemma holds for every \( r' \) such that \( r' \geq r \). \( \square \)
Theorem 4 The algorithm ATC realizes the uniform consensus when condition C2 is satisfied.

PROOF. (1) Termination: Since the system has at least \( f_t + 1 \) non-crashed processes, the RCFS algorithm works correctly. Thus, from Response-Liveness property of RCFS, for any invocation of \( \text{Call}_a(\ast, \ast) \), the corresponding \( \text{Responded}_a(\ast, \ast) \) eventually occurs. This implies that each process eventually invokes \( \text{decide}_a(\ast) \) unless it crashes. (2) Validity and Uniform Agreement: Since the execution of the algorithm consists of \( f + 1 \) rounds, there is a round \( r(\leq f) \) whose coordinator \( p_r \) is correct. Then, from Lemma 3, every \( \text{est}_a[f] \) has a same proposed value. Therefore, the validity condition and the uniform agreement condition holds. \( \Box \)

Theorem 5 The algorithm ATC realizes the \( 4d(f + 1) \)-timed uniform consensus when condition C1 is satisfied.

PROOF. The proof of Termination, Validity and Uniform Agreement is same as Theorem 4. We have only to prove the \( \Delta \)-timed Termination. From Theorem 2, the non-timing-faulty process \( p_i \) executes each round within \( 4d \). The execution of \( p_i \) consists of \( f + 1 \) rounds, and thus \( p_i \) finishes its execution within \( 4d(f + 1) \). The \( \Delta \)-timed Termination clearly holds. \( \Box \)

Optimization

In the algorithm ATC, each process executes exactly \( f + 1 \) rounds. However, if a process \( p_i \) invokes \( \text{Called}_i(p_r, v) \) at an earlier round \( r' \) than \( i \), then no longer \( p_i \) has to execute the rounds between \( r' \) and \( i \) because the value \( v \) is stored to \( \text{est}_i[i] \). This implies that \( p_i \) can skip its own round from \( r' \) to \( i + 1 \). Then, the execution time can be reduced.

The number of messages transmitted by the algorithm is at most \( 4n^2(f + 1) \) because the algorithm RCFS requires \( 4n \) messages for one invocation of \( \text{Call}_a(\ast, \ast) \), and each process invokes \( \text{Call}_a(\ast, \ast) \) \( f + 1 \) times. This message complexity of the algorithm also can be reduced. In the algorithm ATC, all processes query to \( p_r \) when their own round is \( r \), and thus each process may have to work as relayers \( n \) times. However, actually, it is sufficient that a relayer \( p_i \) sends the call2 message to \( p_r \) only once when \( p_i \) receives the call1 message first. For the call1 messages \( p_i \) receives later, \( p_i \) has only to reply the response1 of the first time immediately. If we adopts this scheme, \( 4n + 2n(n - 1) \) messages are transmitted in each round. Therefore, the message complexity is reduced to \( 2n(n + 1)(f + 1) \).
5 Impossibility Results

In this section, we consider the necessary condition so that the $\Delta$-timed consensus can be solved. We show that any $\Delta$-timed uniform consensus algorithm tolerating $f_t$ timing-faulty processes requires that $f_t+1$ correct processes exist. Moreover, we also show that if an $\Delta$-timed uniform consensus algorithm tolerates $f_t$ timing-faulty processes (with some condition), it cannot solve the (non-timed) uniform consensus when exactly $f_t$ processes are non-crashed (even if all non-crashed processes are correct). These impossibly results imply that the algorithm ATC is optimal with respect to the number of faulty processes that can be tolerated, and thus, attains the maximum adaptiveness.

For the proof, we define the execution $E_\Delta(P_a, P_b, v_a, v_b, v_c)$, as the execution satisfying the following conditions: (1) All processes in $P_a$ and $P_b$ are respectively correct and timing-faulty (Notice that $P_a$ and $P_b$ are disjoint). Other processes (denoted by $P_c$) are initially crashed or timing-faulty. (2) Communication delay between any pair $p_i$ and $p_j$ is $(2a)\ d$ if $p_i, p_j \in P_x$ for each $x(= a, b, c)$, or $(2b)\ \Delta + \epsilon$ ($\epsilon > 0$). (3) Each process $p_i \in P_x$ invokes $propose_i(v_x)$ at time 0 unless $p_i$ is initially crashed.

**Lemma 6** No $\Delta$-timed uniform consensus algorithm tolerates $f_t$ timing-faulty processes if $2f_t \geq n$ holds.

**PROOF.** Suppose for contradiction that algorithm $\mathcal{A}$ solves the $\Delta$-timed uniform consensus even when $f_t(\geq n/2)$ processes are timing-faulty. Let $f_0 = \lceil n/2 \rceil$ and $f_1 = \lfloor n/2 \rfloor$. We define two disjoint sets (of processes) $P_0$ and $P_1$ such that $|P_0| = f_0$ and $|P_1| = f_1$. Notice that $P_0 \cup P_1 = P$ holds. For $P_0$ and $P_1$, we define three executions $E_0 = E_\Delta(P_0, P_1, v, v, v)$, $E_1 = E_\Delta(P_1, P_0, v', v', v')$, and $E_2 = E_\Delta(P_0, P_1, v, v', v')$ ($v \neq v'$) (Figure 4). Since $2f_t \geq n$ holds, $f_0 \leq f_t$ and $f_1 \leq f_t$ holds. Thus, in each executions $E_0$, $E_1$, and $E_2$, the algorithm $\mathcal{A}$ solves the $\Delta$-timed uniform consensus. From the uniform validity condition and the $\Delta$-timed termination condition, each process in $P_0$ invokes $decide_a(v)$ in $E_0$ at $\Delta$ or earlier, and each process in $P_1$ invokes $decide_a(v')$ in $E_1$ at $\Delta$ or earlier. This implies that in the execution $E_2$, each process in $P_0$ invokes $decide_a(v)$, and each process in $P_1$ invokes $decide_a(v')$ because each process in $P_0$ (or $P_1$) cannot distinguish the execution $E_2$ from $E_0$ (or $E_1$ respectively). Thus, the execution $E_2$ violates Uniform Agreement condition. That is contradiction. □

**Theorem 7** Let algorithm $\mathcal{A}$ be any $\Delta$-timed uniform consensus algorithm tolerating $f_t$ timing-faulty processes. The algorithm $\mathcal{A}$ cannot solve the $\Delta$-timed uniform consensus when exactly $k$ ($k \leq f_t$) processes are correct.

**PROOF.** From Lemma 6, we only have to prove the case of $f_t \leq \lceil n/2 \rceil - 1$.  

17
Suppose for contradiction that the algorithm $A$ solves the $\Delta$-timed uniform consensus when exactly $k(\leq f_t)$ processes are correct. Since $k \leq f_t = \lceil n/2 \rceil - 1$, we can define two disjoint sets $P_0$ and $P_1$ such that $|P_0| = |P_1| = k$ holds. For $P_0$ and $P_1$, we can define three executions $E_0 = E_\Delta(P_0, P_1, v, v, v)$, $E_1 = E_\Delta(P_1, P_0, v', v', v')$, and $E_2 = E_\Delta(P_0, P_1, v, v, v)$ ($v \neq v'$). Then, each process in $P_2 (= P - (P_0 \cup P_1))$ can be either timing-faulty or crashed under the constraint that the number of timing-faulty processes does not exceed $f_t$. In $E_0$, $E_1$, and $E_2$, exactly $k$ processes are correct, and at most $f_t$ processes are timing-faulty. Therefore, the algorithm $A$ solves the $\Delta$-timed uniform consensus. Afterward, by the same token as the proof of Lemma 6, we can show that each process in $P_0$ or $P_1$ respectively decide $v$ or $v'$ in execution $E_2$, and thus Uniform Agreement is violated. This is contradiction.

**Theorem 8** Let $A$ be the $\Delta$-timed uniform consensus algorithm that tolerates $f_t$ timing-faulty processes if no crashed process exists. The algorithm $A$ cannot solve the (non-timed) uniform consensus when exactly $k$ $(\leq f_t)$ processes are correct and all other processes are crashed.

**Proof.** Suppose for contradiction that the algorithm $A$ solves solves the (non-timed) uniform consensus when $k$ processes are correct and all other processes are crashed. We define two disjoint sets $P_0$ and $P_1$ such that $|P_0| = f$ and $|P_1| = n - k$ holds. For $P_0$ and $P_1$, we define the following execution $E_0$: (1) Each process in $P_0$ is correct, and each process in $P_1$ is initially crashed. (2) Any communication delay between processes in $P_0$ is $d$. (3) Each process in $P_0$ invokes $\text{propose}_*(v)$ at time 0. Clearly, since $E_0$ is a possible execution of $A$, each process in $P_0$ invokes $\text{decide}_*(v)$ within finite time $\Delta'$. Then, letting $\Delta_m = \max\{\Delta, \Delta'\}$, we consider two executions $E_1 = E_{\Delta_m}(P_1, P_0, v', v', v')$ and $E_2 = E_{\Delta_m}(P_1, P_0, v', v, v)$. Since only $k(\leq f_t)$ processes are timing-faulty and no crashed process exists in $E_1$ and $E_2$, the algorithm $A$ solves the $\Delta$-timed uniform consensus. Afterward, by the same token as the proof of Lemma 6, we can show that each process in $P_0$ or $P_1$ respectively decide $v$ or $v'$ in execution $E_2$, and thus Uniform Agreement is violated. This is contradiction.

Theorem 7 implies that no $\Delta$-timed uniform consensus algorithm can tolerate $f_t$ timing-faulty processes if Condition C1 does not hold. Theorem 8 implies that any $\Delta$-timed uniform consensus algorithm that can tolerates $f_t$ timing-faulty processes with some stronger condition (no crashed process) than Condition C1 cannot solve the uniform consensus with some weaker condition (no timing-faulty process) than Condition C2. Therefore, from Theorem 7 and 8, we can conclude the algorithm ATC attains the maximum resilience and adaptiveness with respect to the number of faulty processes.
6 Concluding Remarks

We considered the \( \Delta \)-timed consensus resilient to \( f_c \) crashes and \( f_t \) timing faults. We presented a \( 4d(f_t + f_c) \)-timed consensus algorithm that tolerates \( f_c \) crashes and \( f_t \) timing faults when the system has at least \( f_t + 1 \) correct processes. This algorithm is adaptive: As long as \( f_t + 1 \) non-crashed processes exist in the system, the algorithm also realizes the (non-timed) uniform consensus, even when the system has only less than \( f_t + 1 \) correct processes. We showed that the \( \Delta \)-timed uniform consensus is unsolvable in the system where at most \( f_t \) correct processes exist. We also showed that any \( \Delta \)-timed uniform consensus algorithm tolerating up to \( f_t \) timing faults cannot solve the uniform consensus when at most \( f_t \) processes are non-crashed. These impossibility results imply that our \( \Delta \)-timed uniform consensus algorithm attains the maximum resilience and adaptiveness with respect to the number of faults.

Acknowledgment

The authors would like to thank Dr. Michiko Inoue, associate professor of Nara Institute of Science and Technology(NAIST) for her useful suggestions. This work is supported in part by a JSPS, Grant-in-Aid for Scientific Research((B)(2)15300017), and “The 21st Century Center of Excellence Program” of the Ministry of Education, Culture, Sports, Science and Technology, Japan.
References


[15] A. Schiper, Early consensus in an asynchronous system with a weak failure

[16] J. Yang, G. Neiger, E. Gafni, Structured derivations of consensus algorithms for
failure detectors, in: Proc. of 17th ACM Symposium on Principles of Distributed

of Pacific Rim International Symposium on Dependable Computing(PRDC),

[18] G. M. A. Lima, A. Burns, A timely distributed consensus solution in a
crash/omission-fault environment, in: Proc. of the work in progress session of

leader election, in: Proc. of 15th International Symposium on Distributed

[20] R. Bazzi, G. Neiger, Simplifying fault-tolerance: providing the abstraction of

systems with process omission failures, in: Proc. of the 16th annual ACM
symposium on Parallelism in algorithms and architectures(SPAA), 2004, pp.
302–310.

[22] A. Casimiro, P. Veríssimo, Generic timing fault tolerance using a timely
computing base, in: Proc. of the IEEE International Conference on Dependable
Systems and Networks(DSN), 2002, pp. 27–36.

[23] P. Veríssimo, A. Casimiro, The timely computing base model and architecture,

synchrony and asynchrony, in: Proceedings of the 17th annual ACM symposium

[25] A. Mostefaoui, E. Mourgaya, M. Raynal, Asynchronous implementation of
failure detectors, in: Proc. of the IEEE International Conference on Dependable
Systems and Networks(DSN), 2003, pp. 351–360.


[28] F. Cristian, C. Fetzer, The timed asynchronous distributed system model, IEEE

omega with weak reliability and synchrony assumptions, in: Proc. of 22th
annual ACM symposium on Principles of distributed computing(PODC), 2003,
pp. 306–314.