IEICE TRANS. INF. & SYST., VOL.E88–D, NO.1 JANUARY 2005

PAPER *Special Section on Foundations of Computer Science*

Timed Uniform Atomic Broadcast in Presence of Crash and Timing Faults

Taisuke IZUMI[†a)], ***Student Member and*** **Toshimitsu MASUZAWA**[†], ***Member***

SUMMARY Δ-Timed Atomic Broadcast is the broadcast ensuring that all correct processes deliver the same messages in the same order, and that delivery latency of any message broadcast by any correct process is some predetermined time Δ or less. In this paper, we propose a Δ-timed atomic broadcast algorithm in a synchronous system where communication delay is bounded by a known constant d and processes suffer both crash faults and timing faults. The proposed algorithm can tolerate f_c crash faults and f_t timing faults as long as at least $f_t + 1$ processes are correct, and its maximum delivery latency Δ is $(2f' + 7)d$ where f' is the actual number of (crash or timing) faulty processes. That is, the algorithm attains the *early-delivery* in the sense that its delivery latency depends on the actual number of faults rather than the maximum number of faults that the algorithm can tolerate. Moreover, the algorithm has a distinct advantage of guaranteeing that timing-faulty processes also deliver the same messages in the same order as the correct processes (Uniformity). We also investigate the maximum number of faulty processes that can be tolerated. We show that no Δ-timed atomic broadcast algorithm can tolerate f_t timing faults, if at most f_t processes are correct. The impossibility result implies that the proposed algorithm achieves the maximum fault-resilience with respect to the number of faulty processes.
key words: *timed atomic broadcast, fault tolerance, timing fault, crash fault*

1. Introduction

Atomic broadcast [3], [8] is a fundamental and effective communication primitive for designing fault-tolerant distributed systems. It ensures that all correct processes deliver the same messages in the same order. The atomic broadcast is widely used for preserving consistency of replicated data in many applications: distributed databases [12], shared objects [1], [9], [10], and so on. As corollary of the impossibility result on the consensus problem [6], it is proved that no deterministic algorithm can realize the atomic broadcast in an asynchronous message-passing system subject to only a single crash fault [4], [6]. Thus, several atomic broadcast algorithms have been proposed on the assumption of some synchrony [3], [7] or unreliable failure detection [2].

Synchrony is one of the most commonly used assumptions for designing atomic broadcast algorithms [5]. It assumes that communication delay between any pair of processes is bounded by some constant. The assumption of synchrony also arouses an interest in the possibility of timed atomic broadcast, where delivery latency of any broadcast message is bounded by some predetermined time Δ (Δ-timeliness). The Δ-timed atomic broadcast is not only particularly important for real-time systems, but also desirable for non-real-time systems because message delivery latency strongly affects the overall performance of distributed systems.

In real distributed systems, however, because of the inherent unpredictability of distributed systems, the synchrony assumption is occasionally violated, 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 Δ-timed atomic broadcast may be violated. Therefore, robustness for timing-faults is strongly desired in the Δ-timed atomic broadcast. 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 degenerates the overall performance of distributed systems. Robustness for timing-faults allows to estimate the upper bound more tightly, it can actively improve the performance.

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 Δ-timed atomic broadcast. The system suffers both timing-faults and crashes. A crash fault can be regarded as a special case of timing-faults such that all the messages sent by timing-faulty processes experience infinite communication delay. However, this paper definitely distinguishes crash faults from timing faults because the unification of those two fault models hides the essential difference between them: processes "eventually" receive messages from timing-faulty processes (once those messages are sent), whereas they never receive messages from crashed processes. In other words, there may be a case that f_c crash faults and f_t timing-faults can be tolerated, but $f_t + f_c$ timing-faults cannot be tolerated. Actually, this paper succeeds in showing the difference.

This paper presents a novel timed atomic broadcast algorithm in the synchronous model with crash and timing faults. The algorithm can tolerate f_c crash faults and f_t tim-

Manuscript received March 21, 2004.
Manuscript revised June 23, 2004.
[†]The authors are with the Graduate School of Information Science and Technology, Osaka University, Toyonaka-shi, 560–8531 Japan.
a) E-mail: t-izumi@ist.osaka-u.ac.jp

Copyright © 2005 The Institute of Electronics, Information and Communication Engineers

ing faults as long as at least $f_t + 1$ processes are correct, and its maximum delivery latency Δ is $(2f' + 7)d$ where f' is the actual number of (crash or timing) faulty processes. This implies that the algorithm attains the *early-delivery*, that is, the delivery latency does not depend on the maximum number of faulty processes that the algorithm can tolerate, but depends only on the actual number of faulty processes. In real operation of distributed systems, the actual number of faults is much smaller than the maximum number of faults that can be tolerated. Hence, the early-delivery property effectively affects the performance of the algorithm. Moreover, the algorithm has a distinct advantage of guaranteeing that timing-faulty processes also deliver the same messages in the same order as the correct processes (Uniformity). Since the overdelay of messages are often caused by transient overload of processes or the network, the timing fault should be considered as a transient fault. This implies that the timing-faulty processes recover from the faults and rejoin the application (possibly without detecting the faults). To ensure consistent recovery from the timing faults, it is strongly desired to guarantee Uniformity. The timing-fault tolerance of the atomic broadcast is first introduced in [3]. However, it does not consider the uniformity. To the best of our knowledge, this is the first paper that considers both uniformity and Δ-timeliness in presense of timing-faults.

We also consider the upper bound on the number of faulty processes that timed atomic broadcast algorithms can tolerate. We show that no algorithm can tolerate f_t timing faults if the number of correct processes is f_t or less. The impossibility result implies that our timed atomic broadcast algorithm achieves the maximum fault-resilience with respect to the numbers of both crash faulty and timing faulty processes.

At the end of this section, it is worth while to touch upon the relation between this work and our previous work. In our previous work [11], as a closely related result, we proposed a timed uniform consensus algorithm tolerating crash and timing-faults. It is well known that a uniform atomic broadcast algorithm can be constructed from a uniform consensus algorithm in asynchronous systems (and thus in synchronous systems) [2]. However, it is not the case for the timed uniform atomic broadcast that can tolerate timing-faults. Actually, our construction of a timed uniform atomic broadcast algorithm requires a uniform consensus algorithm that satisfies a stronger validity condition than our previous uniform consensus algorithm. Thus, we cannot construct a timed uniform atomic broadcast algorithm from the uniform consensus algorithm proposed in [2]. We will discuss more details at the end of Sect. 3.

The paper is organized as follows. After introducing the model and definition of the timed atomic broadcast in Sect. 2, we propose the timed atomic broadcast algorithm in Sect. 3. We show the impossibility result in Sect. 4. Finally, we conclude this paper in Sect. 5.

2. Preliminaries

2.1 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 each other by directly exchanging messages. All channels are reliable: each channel correctly transfers messages without loss or duplication. 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 within d. 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 detail.

2.1.1 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 in an instant. Processes are subject to crash and timing faults. These faults are modeled as particular states of processes. Actually, a state of a process is defined as the pair of (s, f), where s is the system state 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 has 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. 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". Afterward, to clarify, we say that the process p_i is correct when p_i is not timing-faulty and never crash, say that the process p_i is non-timing-faulty when p_i is not timing-faulty but may possibly crash in the future, and say that the process p_i is non-crashed when p_i never crashes (even in the future).

There are upper bounds f_c on the number of processes that can crash and f_t on the number of processes that can be timing-faulty: at most f_c processes can experience the state "crashed", and at most f_t processes can experience the state "timing-faulty." We assume that every process knows the values of f_c and f_t a priori. We also denote $f_c'(\leq f_c)$ and $f_t'(\leq f_t)$ to be the actual number of crash or timing-faulty processes respectively. No process can know the value of f_c' and f_t'. Notice that a process is counted as both a timing-faulty process and a crashed process when the process is initially timing-faulty and then crashes. For short, we denote the constant f to be $\min\{f_t + f_c, n-1\}$ and f' to be $f_t' + f_c'$.

For the number of faulty processes, we assume as follows:

Assumption 1 At least $f_t + 1$ processes are correct.

Remark: Since the atomic broadcast algorithm proposed in this paper attains the early-delivery, its performance depends on f' rather than f. This implies that overestimation of f_t and f_c does not cause performance degradation. Thus, practically, users have only to set the parameters f_t and f_c of the algorithm to be maximal under the restriction that Assumption 1 holds.

2.1.2 Timer

Each process has a *timer*. The timer can be set by a primitive operation $Timerset_*(time, id)$ to raise an event $Alarm_*(id)$ identified by *id* after the interval *time*. 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).

2.1.3 Communication

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

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.

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.

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. Notice that we do not assume messages are received in FIFO order.

2.1.4 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. Since we assume synchrony, 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 (a) e' is p_j's event receiving m, (b) p_i's crash event, or (c) 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 (a) e' is p_j's event receiving m, (b) p_i's crash event, or (c) p_j's crash event.
4. If (e, t) is an event setting a timer for an interval τ at a non-timing-faulty process p_i, then there exists the event (e', t') such that (a) e' is p_i's alarm event and $t' = t + \tau$ holds, or (b) 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 τ at a process p_i, then there exists the event (e', t') such that (a) e' is p_i's alarm event and $t \leq t'$ holds, or (b) e' is p_i's crash event and $t \leq t'$ holds.
6. If (e, t) is an internal or send event at a 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 raises 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.

2.2 Δ-Timed Uniform Atomic Broadcast

Uniform Atomic broadcast is the broadcast ensuring that all non-crashed processes deliver the same set of messages in the same order. The set of messages includes all messages broadcast by processes and no spurious messages. The Δ-timed uniform atomic broadcast is defined to be the atomic broadcast with an additional property, Δ-timeliness.

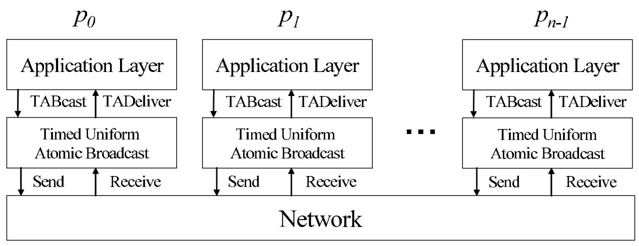


Fig. 1 Architecture of the uniform atomic broadcast.

The Δ-timeliness guarantees that delivery latency between two non-timing-faulty processes is bounded by some constant Δ. The atomic broadcast algorithm provides two interfaces, *TABcast*$_i(m)$ and *TADeliver*$_i(m)$, to an upper application layer. The event *TABcast*$_i(m)$ is invoked by the upper application to broadcast a message m by the Δ-timed uniform atomic broadcast, and *TADeliver*$_i(m)$ is invoked by the Δ-timed uniform atomic broadcast algorithm to deliver a message m. The algorithm is implemented by using *Send* and *Receive* primitives that are defined previously, and deployed, as a module, to each process. Figure 1 illustrates the architecture of the uniform timed atomic broadcast. Formally, the Δ-timed uniform atomic broadcast is the broadcast satisfying the following specifications:

Validity: If a process p_i broadcasts a message m and does not crash, then the process p_i eventually delivers m, or become crashed eventually.
Uniform Integrity: For any message m, every process delivers m at most once, only if some process broadcasts m.
Uniform Agreement: If a process delivers a message m, then each process eventually delivers m, or becomes crashed eventually.
Uniform Total Order: Let processes p_i and p_j both deliver messages m and m'. Then, p_i delivers m before m' if and only if p_j delivers m before m'.
Nonfaulty Δ-Validity: If a non-timing-faulty process p_i broadcasts a message m at t and does not crash by $t+\Delta$, then the process p_i delivers m at $t + \Delta$ or earlier.
Δ-Agreement: If a non-timing-faulty process broadcasts a message m at t and a process delivers m, then each non-timing-faulty process delivers m at $t + \Delta$ or earlier, or is crashed at $t + \Delta$.

We call the messages to be broadcast by the Δ-timed uniform atomic broadcast algorithm "ABcast message" to distinguish them from the messages sent by *Send* primitive.

3. Δ-Timed Uniform Atomic Broadcast Algorithm

3.1 Overview

In general, there are three problems in implementing the Δ-timed uniform atomic broadcast: reliable delivery (all processes deliver the same set of ABcast messages unless they crash), totally-ordered delivery (all processes deliver the messages in the same order) and Δ-timeliness. Informally, the algorithm resolves these three problems as follows: The algorithm divides its execution into synchronous rounds. Each process assigns each received ABcast messages to a round. All ABcast messages are delivered in order of the assigned rounds. Two ABcast messages assigned to the same round are delivered in ascending order of their broadcaster's ID. If the two messages have the same broadcaster's IDs, the one broadcast earlier is delivered first. Clearly, if all processes assign a same set of ABcast messages to a common round, reliable delivery and totally-ordered delivery is guaranteed.

In order to assign an ABcast message to a common round, the algorithm uses the consensus. At the end of each round, each process executes the consensus algorithm to agree on the set of messages assigned to the round. However, to ensure the Δ-timeliness, the consensus algorithm must complete its execution within constant time, regardless of presence of faulty processes. Moreover, to ensure timing-faulty processes also deliver ABcast messages in the same order as correct processes, the timing-faulty processes have to participate in the consensus. Therefore, we cannot use existing consensus algorithms. To resolve these problems, we propose a novel consensus algorithm, Δ-timed consensus. The Δ-timed consensus algorithm has the following two properties distinct from existing consensus algorithms: (1) When a non-timing-faulty process proposes a value at t, it decides a value at $t + \Delta$ or earlier unless it crashes by $t + \Delta$. That is, completion of the algorithm by non-timing-faulty processes is not delayed by timing-faulty processes. (2) After execution of the algorithm, all non-crashed processes, including the timing-faulty processes, have a common decision.

The Δ-timed uniform atomic broadcast algorithm consists of three parts, the round synchronization algorithm Sync, Δ-timed consensus Tconsensus and the main algorithm TABcast. The algorithm Sync divides an execution into synchronous rounds. Since the algorithms Tconsensus and TABcast are based on the synchronous rounds, we first introduce the round synchronization algorithm Sync in the next subsection, and later introduce the other two algorithms.

3.2 Round Synchronization Algorithm Sync

3.2.1 Specification

The objective of the round synchronization algorithm is that each process synchronously repeats invocation of the end-of-round events. The algorithm provides two primitives, *StartSync*$_i$ and *EOR*$_i(r)$. The event *StartSync*$_i$ and *EOR*$_i(r)$ are respectively invoked by process p_i to initiate the round synchronization and to report the end of the rth round. Formally, the round synchronization algorithm satisfies the following specifications:

Timed Initiation: If a non-timing-faulty process p_i invokes the *StartSync*$_i$ at t and does not crash by $t + d$, there is some constant δ such that every non-timing-faulty process p_k invokes $EOR_k(0)$ or becomes crashed by $t + \delta$.

Eventual Initiation: If a timing-faulty process p_i invokes the $StartSync_i$ at t and does not crash, every non-timing-faulty process p_k invokes $EOR_k(0)$ or becomes crashed eventually.

Liveness: If a non-timing-faulty process p_i invokes $EOR_i(r)$ at t, it invokes $EOR_i(r + 1)$ or crashes after t.

Integrity: For any positive integer r, every process p_i invokes $EOR_i(r)$ at most once only if $EOR_i(r - 1)$ has already occurred.

Agreement: If a non-timing-faulty process p_i invokes $EOR_i(0)$, then each non-timing-faulty process p_k invokes $EOR_k(0)$ or becomes crashed eventually.

Synchrony: If a non-timing-faulty process p_i invokes $EOR_i(r)$ at t_i and a non-timing-faulty process p_k invokes $EOR_k(r + 1)$ at t_k, then $t_k - t_i \geq d$ holds.

The synchrony property implies that the length of each round is sufficiently long so that if a non-timing-faulty process sends a message to another non-timing-faulty process at the beginning of a round, the message can be received by the end of the round.

3.2.2 Algorithm

The key of the round synchronization algorithm Sync is to synchronize the occurrence of $EOR_*(0)$ at all non-timing-faulty processes. If $EOR_*(r)$ are invoked synchronously, it is easy to invoke $EOR_*(r + 1)$ synchronously: On occurrence of $EOR_i(r)$, each process p_i sets a timer for some predefined time ($2d$ in the proposed algorithm), and it invokes $EOR_i(r + 1)$ when the alarm raises. Then, each non-timing-faulty process invokes $EOR_i(r + 1)$ with the same difference on timing as that of the $EOR_i(r)$. Thus, in what follows, we only describe how to synchronize $EOR_i(0)$.

In our algorithm, the process p_i invoking $StartSync_i$ broadcasts an invocation message. When a process receives the invocation message first, it broadcasts the invocation message and invokes $EOR_*(0)$ after d. Figure 2 presents the program code of Sync in 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.

```
1:  variable
2:  activated_i : init FALSE
3:  round_i : init 0

4:  transition function of process p_i

5:  upon StartSync_i do :
6:      if activated_i = FALSE then
7:          send_i(invocation) to all processes (including p_i)
8:      endif

9:  upon receive_i(invocation) do :
10:     if activated_i = FALSE then
11:         activated_i ← TRUE
12:         send_i(invocation) to all processes
13:         Timerset_i(d, next)
14:     endif

15: upon Alarm_i(next) do :
16:     EOR_i(round_i)
17:     round_i ← round_i + 1
18:     Timerset_i(2d, next)
```

Fig. 2 Algorithm Sync.

3.2.3 Correctness

Theorem 1 The algorithm Sync realizes the round synchronization,

Proof (1) Liveness and Integrity: These clearly hold. **(2) Timed Initiation**: If a non-timing-faulty process p_i broadcasts the invocation messages at t (that is, p_i invokes $StartSync_i$ at t) and does not crash by $t + d$, every non-timing-faulty process p_j receives the invocation message by $t + d$ and invokes $Timerset(d, next)$ (line 13). This implies that p_j invokes $EOR_j(0)$ by $t + 2d$ unless it crashes (line 16). Therefore Timed Initiation clearly holds. **(3) Eventual Initiation**: If a timing-faulty process p_i broadcasts the invocation message and does not crash, every non-timing-faulty process eventually receives the invocation message and invokes $EOR_*(0)$. Thus, the eventual initiation holds. **(4) Agreement and Synchrony**: Let p_i be the non-timing-faulty process that invokes $EOR_i(0)$ earliest, and t be the time when p_i invokes $EOR_i(0)$ (line 16). Then, since $Alarm_i(next)$ occurs at t (line 15), p_i invokes $Timerset_i(d, next)$ at $t - d$. This implies that p_i broadcasts the invocation message at $t - d$ (line 12). Since p_i does not crash by t, each non-timing-faulty process receives the invocation message by t and invokes $EOR_*(0)$ at $t + d$ or earlier (Agreement). Thus the difference of timings when each process invokes $EOR_*(0)$ is at most d. This also holds for $EOR_*(r)$ of any r. Since $EOR_k(r + 1)$ occurs exactly $2d$ later after $EOR_k(r)$ at each process p_k (line 15–18), Synchrony property holds. □

Concerning the length of each round, the following corollary obviously holds.

Corollary 1 (1) If a non-timing-faulty process p_i invokes $StartSync_i$ at t, then $EOR_k(0)$ is invoked at $t + 2d$ or earlier at any non-timing-faulty process p_k. (2) When a non-timing-faulty process p_i invokes $EOR_i(r)$ ($r \geq 0$) at t, then $EOR_i(r + 1)$ is invoked at $t + 2d$ unless it crashes by $t + 2d$.

3.3 Δ-Timed Consensus Algorithm Tconsensus

3.3.1 Specification

The Δ-timed consensus is the consensus algorithm such that each non-timing-faulty process completes its execution within constant Δ time even in presence of faulty processes. Moreover, every timing-faulty process reaches the same decision as correct processes. The Δ-timed consensus algorithm provides two primitives, $propose_i(V)$ and $decide_i(V)$.

The events $propose_i(V)$ and $decide_i(V)$ are respectively invoked by process p_i to propose the value V and to return the decision value V. Notice that V is a set (of ABcast messages in our Δ-timed atomic broadcast algorithm). We define the Δ-timed consensus algorithm to be the algorithm satisfying the following specification:

Δ-Timed Termination: If a non-timing-faulty process p_i invokes $propose_i(V)$ at t, p_i invokes $decide_i(*)$ or becomes crashed by $t + \Delta$.

Eventual Termination: If a process p_i invokes $propose_i(V)$ at t, p_i invokes $decide_i(*)$ or becomes crashed eventually.

Uniform Agreement: If $decide_i(V_i)$ and $decide_j(V_j)$ occur, $V_i = V_j$ holds.

Validity: If a non-timing-faulty process p_i invokes $propose_i$ (V_i) and $decide_i(V_i')$, then $V_i \subseteq V_i'$ holds.

Since our algorithm works on synchronized rounds realized by Sync, we assume that a process (or the subset of processes in the system) invokes $StartSync_*$ in advance. Moreover, the algorithm requires the following assumption:

Assumption 2 All non-crashed processes invoke $propose$ $_*(*)$ at the beginning of a common round $r (r \neq 0)$.

In the following discussion, we assume that the all non-crashed processes invoke $propose_*(*)$ at the beginning of round 1.

```
1:  variable
2:  $r_i$ : init 1
3:  $gatd_i$ : init FALSE
4:  $vals_i, sent_i$ : init $\emptyset$
5:  $dec_i$ : multiset init $\emptyset$
6:  $rcvd_i[1..f+1], msgs_i[1..f+1]$ : init $(\emptyset, \emptyset, \cdots, \emptyset)$
7:  $sus_i$ : init $\emptyset$

8:  transition function of process $p_i$

9:  upon $propose_i(V)$ do :
10:     $vals_i \leftarrow vals_i \cup V$
11:     $send_i(vals_i, r_i)$ to all processes

12: upon $receive_i(vals, r)$ from $p_k$ do :
13:     $msgs_i[r] \leftarrow msgs_i[r] \cup \{(vals, p_k)\}$
14:     $rcvd_i[r] \leftarrow rcvd_i[r] \cup \{p_k\}$

15: upon $EOR_i(*)$ do :
16:     if $gatd_i$ = FALSE then
17:         foreach $(vals, p_k) \in msgs_i[r_i]$ do
18:             if $p_k \notin sus_i$ then
19:                 $vals_i \leftarrow vals_i \cup vals$                /* gathering values */
20:             endif
21:         endfor
22:         $sus_i \leftarrow sus_i \cup (P - rcvd[r_i])$
23:         $r_i \leftarrow r_i + 1$                         /* the beginning of next round */
24:         if $|sus_i| + 1 < r_i$ then
25:             $gatd_i$ = TRUE                                /* the end of first phase */
26:             $send_i(vals_i$, EST) to all processes
27:         else
28:             $send_i(vals_i - sent_i, r_i)$ to all processes
29:             $sent_i \leftarrow vals_i$
30:         endif
31:     endif

32: upon $receive_i(est$, EST) do :
33:     $dec_i \leftarrow dec_i \cup \{est\}$
34:     if $est$ appears $f_i + 1$ times in $dec_i$ then
35:         $decide_i(est)$
36:     endif
```

Fig. 3 Algorithm Tconsensus.

3.3.2 Algorithm

In this subsection, we briefly describe the idea of the Δ-timed consensus algorithm Tconsensus (Fig. 3). The algorithm consists of two phases. The objective of the first phase is that all non-timing-faulty processes estimate a common decision value. In the first phase, each process tries to gather the proposed values from all processes and determines the union (set) of the gathered values to be the estimation. The algorithm guarantees that all non-timing-faulty processes gather the same set of proposed values, and thus they estimate the common decision value. However, at the end of the first phase, a timing-faulty process does not necessarily have the same estimation as non-timing-faulty processes. In the second phase, the algorithm guarantees that timing-faulty processes decide the same value as non-timing-faulty processes.

The algorithmic scheme of the first phase derives from the existing early-deciding consensus algorithm tolerating send/receive omission faults [13], [14]: The algorithm treats overdelayed messages as lost messages, and neglects them. The sender p_i attaches its current round number r to each message m, and receiver p_k stores m to its local variable $msgs_k[r]$. At the end of p_k's round r, p_k processes the messages in $msgs_k[r]$. If m is received after the end of p_k's round r, p_k ignores m.

The first phase consists of at most $f' + 1$ consecutive rounds. Each process p_i maintains the two variables $vals_i$ and $sent_i$, which respectively stand for the set of values that p_i has already gathered and the set of values that p_i has already sent. In each round, each process sends the value $vals_i - sent_i$, which represents the values newly gathered in the immediately previous round, to all processes (line 28). At the first round, each process sends its own proposal. In addition to this gathering scheme, each process also suspects faulty processes. When a process p_i does not receive the message from p_k, then the process p_i suspects p_k to be faulty and subsequently ignores all messages that p_i receives from p_k (line 22). This suspicion may be faulty if p_i is timing-faulty. However, it is guaranteed that a non-timing-faulty process never suspects any other non-timing-faulty process. The set of suspected processes is maintained in the local variable sus_i. Notice that the variable sus_i is accumulative, that is, once a process p_k is added to sus_i, it is never removed.

Each process p_i terminates the first phase when it begins the round r such that $r > |sus_i| + 1$ (line 24). Actually, p_i can gather no additional value in the last round $(|sus_i| + 1)$. However, to broadcast the values p_i gathered in round $|sus_i|$, p_i executes the extra round. When process p_i terminates the first phase, it changes the value of $gatd_i$ to TRUE (line 25).

In the second phase, each process sends the union of the all gathered values, as its estimation, to all processes. When each process receives the same estimation value V

$f_t + 1$ times, it invokes $decide_i(V)$ (line 32–36). As we mentioned, at the end of the first phase, all non-timing-faulty processes reach the same estimation, but timing-faulty processes can have different estimations. Therefore, if a process receives the same estimation value V $f_t + 1$ times, then V is the value estimated by non-timing-faulty processes because at most f_t processes have different estimations.

3.3.3 Correctness

In this subsection, we prove the correctness of the algorithm Tconsensus. For the proof, we define the following notations and terms: Let $vals_i(r)$ and $sus_i(r)$ respectively be the values of $vals_i$ and sus_i at the beginning of round r, and r_i^d be the value of r_i when the value of $gatd_i$ changes to TRUE. If $gatd_i$ never changes to TRUE, the value of r_i^d is undefined. We call a non-timing-faulty process p_i a *correct estimator* if r_i^d is defined. For a message $m = (V, *)$, we say "message m contains v" if $v \in V$ holds. Moreover, we say "process p_i gathers the value v in the round r" (1) when $v \notin vals_i(r)$ and $v \in vals_i(r + 1)$ holds for $r \geq 1$, or (2) when $r = 0$ holds and p_i proposes the value that contains v.

Lemma 1 If a process p_i gathers a value v in round $r(\geq 1)$, there exists a process p_k that gathers the value v in round $r - 1$.

Proof (**Case1**) For $r = 1$: The lemma clearly holds (the process that proposes V containing v is p_k). (**Case2**) When $r > 1$ holds: Since p_i gathers v in round r by executing line 19, p_i receives a message m containing v from some process $p_j \notin sus_i(r)$. In what follows, we prove that p_j gathers the value v at round $r - 1$. Suppose for contradiction that p_j gathers the value v at round r' $(r' < r - 1)$. Then, since the process p_j executes the first phase at round r (thus, also at round $r' + 1$), it sends a message m' containing v to p_i by executing line 28 at the beginning of round $r' + 1$. Moreover, it follows from $p_j \notin sus_i(r)$ that $p_j \notin sus_i(r' + 1)$ holds. These imply that p_i gathers v at round $r' + 1 < r$ because $v \in vals_i$ when p_i execute the line 17-21 in round $r' + 1$. That is a contradiction. □

Lemma 2 If a process p_i gathers a value v at round $r(\geq 1)$, then, at the beginning of round $r'(1 \leq r' \leq r)$, at least r' processes have already gathered the value v.

Proof By applying Lemma 1 recursively, if p_i gathers v in round r, there exists a sequence of distinct processes $S = p'_0, p'_1, \cdots, p'_{r-1}$ such that p'_j gathers v at round j. Thus, at least r' processes have already gathered v at the beginning of round r'. □

Lemma 3 Any process p_i gathers no value at round $r_i^d - 1$.

Proof Assume for contradiction that a process p_i gathers a value v in round $r_i^d - 1$. We consider the same sequence $S = p'_0, p'_1, \cdots, p'_{r_i^d-2}$ as proof of Lemma 2. Then, for any k $(0 \leq k \leq r_i^d - 3)$, p'_k sends a message m containing v to all processes (by executing line 28) at the beginning of round $k + 1$. On the other hand, p_i never gathers the value v by round $r_i^d - 2(\geq k + 1)$. This implies that p_i does not receive m at round $k + 1$ and adds p'_k to sus_i at round $k + 1$. Thus, $r_i^d - 2 \leq |sus_i(r_i^d - 1)|$ holds, that is, $r_i \leq |sus_i(r_i)| + 1$ holds when p_i executes line 23 at the round $r_i^d - 1$. This contradicts the fact that the value of $gatd_i$ changes to TRUE in round $r_i^d - 1$. □

Lemma 4 Let p_i and p_k be correct estimators. If $v \in vals_i(r_i^d)$ holds, $v \in vals_k(r_k^d)$ holds.

Proof Let r be the round when p_i gathers a value v. We consider the following two cases. (**Case1**) When $r < r_k^d - 1$ holds: From Lemma 3, $r < r_i^d - 1$ holds. This implies that p_i does not crash in round $r + 1$ or earlier, and thus, at the beginning of round $r + 1(\leq r_k^d - 1)$, p_i sends the message m containing v to p_k. Since the processes p_k and p_i are not timing-faulty, p_k never suspects p_i in round $r + 1$ or earlier. Hence, p_k receives m in round $r + 1$. This implies that $v \in vals_k(r + 2) \subseteq vals_k(r_k^d)$ holds. (**Case2**) When $r \geq r_k^d - 1$ holds: In each round, the process p_k receives messages from all processes except for at most $r_k^d - 2$ processes in round $r_k^d - 1$ because $|sus_k(r_k^d)| < r_k^d - 1$ holds. On the other hand, from Lemma 2, at least $r_k^d - 1$ processes have already gathered the value v at the beginning of round $r_k^d - 1$, and send the messages containing v in the round $r_k^d - 1$ or earlier. This implies that p_k receives at least one message containing v by round $r_k^d - 1$. Then, $v \in vals_k(r_k^d)$ holds. □

From this lemma, the following corollary clearly holds:

Corollary 2 If non-timing-faulty processes p_1 and p_2 respectively send a message (est_1, EST) and (est_2, EST), then $est_1 = est_2$ holds.

Theorem 2 The algorithm Tconsensus realizes the $2d(f' + 2)$-timed consensus.

Proof **(1) Δ-Timed Termination and Eventual Termination**: Let p_i be a non-timing-faulty process. Since the cardinality of $sus_i(*)$ is at most f', at the beginning of round $f' + 2$ or earlier, $|sus_i| + 1 < r_i$ holds. Thus, the process p_i terminates its first phase within $f' + 1$ rounds and sends the EST message to all processes. Then, from Corollary 2 and Assumption 1, at least $f_t + 1$ correct processes necessarily send the same EST message at the beginning of the round $f' + 2$ or earlier (line 26). Therefore, all non-timing-faulty processes receive the same estimation value at least $f_t + 1$ times by the end of the round $f' + 2$. Since it takes $2d$ that a non-timing-faulty process executes each round from Corollary 1 and Assumption 2, $2d(f' + 2)$-Timed Termination holds. Moreover, all timing-faulty (and non-crashed) processes also receive the same estimation value at least $f_t + 1$ times eventually. Therefore, Eventual termination holds. **(2) Uniform Agreement**: Let v_i and v_k be the decision values of p_i and p_k respectively. Then, p_i receives $f_t + 1$ EST messages (v_i, EST) and p_k receives $f_t + 1$ EST messages (v_k, EST) (line 33–35). Thus p_i and p_k receive at least one message from a non-timing-faulty process. From Corollary 2, all the EST messages sent by non-timing-faulty processes contain a common estimation value. This implies that $v_i = v_k$ holds. **(3) Validity**: If a non-timing-faulty process p_i proposes a value V, then all the estimations by non-timing-faulty processes include V because p_i is never suspected by non-timing-faulty processes and thus V is gathered by all non-timing-faulty processes. Since the decision value is the estimation by non-timing-faulty processes, Validity clearly holds. □

3.4 Δ-Timed Uniform Atomic Broadcast TABcast

3.4.1 Algorithm

Using Tconsensus and Sync, we realize the Δ-timed uniform atomic broadcast algorithm TABcast. In the previous section, we have briefly showed the idea of TABcast. We describe the detailed behavior of the algorithm in this subsection. The algorithm TABcast executes the algorithm Sync in advance: At the first time when a process p_i broadcasts an ABcast message by the Δ-timed uniform atomic broadcast, p_i invokes the $StartSync_i$ to start Sync if p_i has not yet recognized that Sync has already been started (line 7). Each process maintains the local variable *synced* to represent whether Sync is running or not. The variable *synced* is initially FALSE, and changes to TRUE when $StartSync_*$ or $EOR_*(0)$ occur.

In our algorithm, every ABcast message has a distinct identifier. An identifier is a pair of broadcaster's ID and a serial number in the broadcaster. Each process maintains its own serial number in a local variable *sn*. When $TABcast_i(m)$ occurs, the process broadcasts $M = (m, p_i, sn)$. When a process receives the message M, it records M to a local variable *Received*. When the process delivers the ABcast message m, it appends M to a local variable *Delivered*.

At the end of each round r, the algorithm executes Tconsensus. Since the execution of Tconsensus takes several rounds, two or more Tconsensus algorithm may be concurrently executed. Then, each execution is distinguished from each other by its round number, and is executed independently. In Fig. 4, $propose_i^r(V)$ and $decide_i^r(D)$ respectively denote p_i's propose and decide events of Tconsensus initiated at the end of round r. The value a process p_i proposes in round r is the set of ABcast messages that are received by p_i but not yet delivered (that is $Received - Delivered$) at the end of round $r - 1$ (line 23). When $decide_i^r(D)$ occurs and p_i terminates all Tconsensus executions corresponding to the rounds smaller than r, the process p_i removes the messages that have already delivered by p_i from D and delivers the remaining messages in order of their broadcaster's ID. If two or more messages

```
 1: variable
 2:   $sn_i, decd_i$ : init 0
 3:   $synced_i$ : init FALSE
 4:   $DeliverList_i, Received_i, Delivered_i$ : init $\emptyset$

 5: transition function s of process $p_i$

 6: upon $TABcast_i(m)$ do :
 7:     if $synced_i$ = FALSE then $StartSync_i$ endif
 8:     $send_i(m, p_i, sn_i)$ to all processes
 9:     $sn_i \leftarrow sn_i + 1$

10: upon $Receive_i(m, p_j, sn)$ from $p_k$ do :
11:     $Received_i \leftarrow Received_i \cup \{(m, p_j, sn)\}$

12: upon $decide_i^r(D)$ do :
13:     wait until $decd_i = r$ holds
14:     $DeliverList_i \leftarrow D - Delivered_i$
15:     TADeliver all messages in $DeliverList_i$
16:         in some deterministic order
17:         (It is ordered firstly by Broadcaster's ID,
18:         and secondary by $sn$.)
19:     $Delivered_i \leftarrow Delivered_i \cup DeliverList_i$
20:     $decd_i \leftarrow decd_i + 1$

21: upon $EOR_i(r)$ do :
22:     if $synced$ = FALSE then $synced$ = TRUE
23:     $propose_i^r(Received_i - Delvered_i)$
```

Fig. 4 Algorithm TABcast.

have the same broadcaster's ID, they are ordered by their serial numbers (line 12–20).

3.4.2 Correctness

We denote V_i^r to be p_i's decision value for the execution of TConsensus corresponding to the round r (If p_i does not invoke $decide_i^r(*)$, V_i^r is undefined). For a ABcast message m, we say "p_i assigns m to round r" if $V_i^{r'}$ is defined for any $r' \leq r$, $m \in V_i^r$ holds, and $m \notin V_i^{r'}$ holds for any $r' < r$.

Lemma 5 If a process p_i assigns m to a round r and another process p_j assigns m to a round r', then $r = r'$ holds.

Proof From the uniform agreement property, $m \in V_j^r$ holds and $m \notin V_j^k$ holds for any $k < r$. This implies that $r = r'$ holds. □

Theorem 3 The algorithm TABcast realizes the $(2f' + 7)d$-timed atomic broadcast.

Proof **(1) Validity**: Suppose for contradiction that p_i broadcasts m but does not deliver it. Since the process p_i does not crash, for any r, $decide_i^r(*)$ eventually occurs. Therefore, if p_i does not deliver m, it assigns m to no round. On the other hand, p_i eventually receives m. Letting r' be the round where p_i receives m, from the validity property of Δ-timed consensus, the message m is assigned to a round $r' + 1$ or an earlier round. This is contradiction. **(2) Uniform Integrity**: Clearly, each process never delivers ABcast message m that is not broadcast. Moreover, once a process p_i delivers m, it stores m in the variable $Delivered_i$ (line 19). Since the messages in *Delivered* are never delivered again, Uniform Integrity holds. **(3) Uniform Agreement**: From Lemma 5, this property clearly holds. **(4) Uniform**

Total Order: From Lemma 5, p_i and p_j assigns m_1 to a common round (and also assigned m_2 to a common round). Since messages assigned same round are delivered in order of their broadcaster's IDs and serial numbers (line 15–18), then p_i and p_j clearly deliver m_1 and m_2 in same order.
(5) Nonfaulty Δ-Validity and Δ-Agreement: Let a non-timing-faulty process p_i invoke $TABcast_i(m)$ at t. All correct processes receive m at $t + d$ or earlier. Thus, at $t + 3d$ or earlier, each correct process p_j invokes $propose^r_j(V_j)$ such that V_j contains m. Then, because of the validity property of the Δ-timed consensus, $m \in V^r_j$ holds for any p_j. This implies that m is assigned to the round r or an earlier round. Since the executions of every Tconsensus initiated in round r or earlier terminates within at most $f'+3$ rounds from Theorem 2, m is delivered at $t + 3d + (f' + 2)2d = t + (2f' + 7)d$ or earlier. □

3.5 Discussion

The algorithm TABcast is based on repeated execution of the consensus algorithm Tconsensus that determines the sets of messages to be delivered. Such consensus-based atomic broadcast was first introduced in Chandra-Toueg's atomic broadcast algorithm [2]. On the other hand, as we mentioned in the introduction, we already proposed a timed uniform consensus algorithm resilient to crash and timing faults [11]. However, the consensus algorithm in [11] cannot be used as a building block of the Δ-timed uniform atomic broadcast because it only guarantees a weaker validity condition such that the decision value is one of proposals. The following scenario clarifies the problem of the weaker validity condition in construction of the Δ-timed atomic broadcast algorithm: Consider a system consisting of three processes p_0, p_1 and p_2, where p_0 and p_1 are correct and p_2 is timing-faulty. Consider an execution where p_0 invokes $TABcast_0(m)$ at time 0, all messages received by p_2 experience delay larger than Δ, and the decision value for every execution of consensus is p_2's proposal. Since no message containing m is delivered at p_2 by time Δ, p_2's proposals never contain m by time Δ. Thus, the Δ-agreement condition of the Δ-timed uniform atomic broadcast is violated. This case actually occurs if the algorithm TABcast adopts the algorithm in [11] as a consensus module. Therefore, we had to newly design the Δ-timed consensus algorithm Tconsensus as a building block of the Δ-timed uniform atomic broadcast algorithm.

4. Impossibility Result

In this section, we prove that no Δ-timed uniform atomic broadcast algorithm can tolerate f_t timing-faults when the number of correct processes is f_t or less. This implies that our algorithm attains maximum fault-resilience with respect to the number of faulty processes.

For the proof, we define the execution $E_\Delta(P_0, P_1, m_0, m_1)$, as the execution satisfying the following conditions: (1) All processes in P_0 and P_1 are respectively always correct and timing-faulty (P_0 and P_1 are disjoint and nonempty). Other processes (denoted by P_2) are initially crashed. (2) Communication delay between any pair p_i and p_j is (2a) d if $p_i, p_j \in P_x$ for each $x(= 0, 1, 2)$, otherwise (2b) $\Delta + \epsilon$ ($\epsilon > 0$). (3) A process $p_i \in P_0$ invokes $TABcast_i(m_0)$ at time 0 and a process $p_j \in P_1$ invokes $TABcast_j(m_1)$ at time 0.

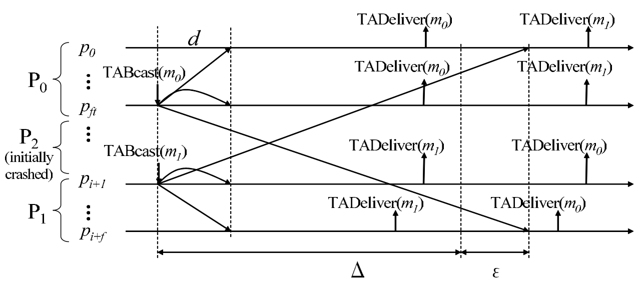


Fig. 5 Execution E_0 and E_1.

Theorem 4 Let algorithm $\mathcal{A}$ be a Δ-timed uniform atomic broadcast algorithm. If the algorithm $\mathcal{A}$ tolerates f_t timing-faulty processes when at least k processes are correct, then $f_t < k$ holds.

Proof Suppose for contradiction $f_t \geq k$ holds. When f_t processes are actually timing-faulty, $2k \leq n$ clearly holds. Thus, we can define two disjoint sets P_0 and P_1 such that $|P_0| = f_t$, $|P_1| = k$. For P_0 and P_1, we can define two executions $E_0 = E_\Delta(P_0, P_1, m_0, m_1)$, $E_1 = E_\Delta(P_1, P_0, m_1, m_0)$ (Fig. 5). In E_0 and E_1, at least k processes are correct, and at most f_t processes are timing-faulty. Therefore, the algorithm $\mathcal{A}$ solves the Δ-timed atomic broadcast. From Nonfaulty Δ-Validity property and Δ-Agreement property each process in P_0 invokes $TADeliver_*(m_0)$ in E_0 at Δ or earlier, and each process in P_1 invokes $TADeliver_*(m_1)$ in E_1 at Δ or earlier. In addition, no process in P_1 invokes $TADeliver_*(m_0)$ in E_1 by $\Delta + \epsilon$ because processes in P_1 receive no message from processes in P_0 by $\Delta + \epsilon$. By the same reasoning, no process in P_0 invokes $TADeliver_*(m_1)$ in E_0 by $\Delta + \epsilon$. This implies that in the execution E_0 each process in P_0 first invokes $TADeliver_*(m_0)$ and then invokes $TADeliver_*(m_1)$, whereas each process in P_1 first invokes $TADeliver_*(m_1)$ because no process in P_1 can distinguish the execution E_1 from E_0. Thus, the execution E_0 violates Uniform Total Order condition. That is contradiction. □.

5. Concluding Remarks

We considered the Δ-timed atomic broadcast in a synchronous system where communication delay is bounded by a known constant d and processes suffer both crash faults and process timing faults. The proposed algorithm can tolerate f_c crash faults and f_t timing faults as long as at least $f_t + 1$ processes are correct, and its delivery latency Δ is $(2f' + 7)d$. This is early-delivery algorithm. Moreover, the algorithm guarantees that timing-faulty processes also delivers the same messages in the same order as the correct processes (Uniformity).

We also investigated the maximum number of faulty processes that can be tolerated. We show that no Δ-timed atomic broadcast algorithm can tolerate f_t timing faults, if at most f_t processes are correct. The impossibility result implies that the proposed algorithm achieves the maximum resilience to the number of faulty processes.

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

[1] H. Attiya and J.L. Welch, "Sequential consistency versus linearizability," ACM Trans. Comput. Syst., vol.12, no.2, pp.91–122, 1994.

[2] T.D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," J. ACM, vol.43, no.2, pp.225–267, 1996.

[3] F. Cristian, H. Aghali, R. Strong, and D. Dolev, "Atomic broadcast: From simple message diffusion to byzantine agreement," Inf. Comput., vol.118, no.1, pp.158–179, 1995.

[4] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," J. ACM, vol.34, no.1, pp.77–97, 1987.

[5] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," J. ACM, vol.35, no.2, pp.288–323, 1988.

[6] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of distributed consensus with one faulty process," J. ACM, vol.32, no.2, pp.374–382, 1985.

[7] A. Gopal, R. Strong, S. Toueg, and F. Cristian, "Early-delivery atomic broadcast," Proc. 19th annual ACM symposium on Principles of distributed computing (PODC), pp.297–309, 1990.

[8] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in Distributed Systems, ed. S. Mullender, ch.5, pp.97–145, Addison-Wesley, 1993.

[9] T. Herman and T. Masuzawa, "Stabilizing replicated search tree," Proc. 15th International Symposium on Distributed Computing (DISC), pp.315–329, 2001.

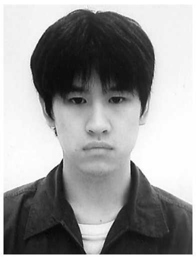
[10] M. Inoue, T. Masuzawa, and N. Tokura, "Efficient linearizable implementation of shared FIFO queues and general objects on a distributed system," IEICE Trans. Fundamentals, vol.E81-A, no.5, pp.768–775, May 1998.

[11] T. Izumi, A. Saitoh, and T. Masuzawa, "Timed uniform consensus resilient to crash and timing faults," Proc. IEEE International Conference on Dependable Systems and Networks (DSN), pp.243–252, July 2004.

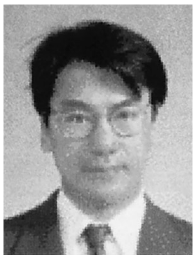
[12] F. Pedone, R. Guerraoui, and A. Schiper, "Exploiting atomic broadcast in replicated databases," Proc. International Conference on Parallel and Distributed Computing (Euro-Par), pp.513–520, 1998.

[13] K.J. Perry and S. Toueg, "Distributed agreement in the presence of processor and communication faults," IEEE Trans. Softw. Eng., vol.SE-12, no.3, pp.477–482, 1986.

[14] M.R. Philippe Raïpin Parvédy, "Optimal early stopping uniform consensus in synchronous systems with process omission failures," Proc. 16th annual ACM symposium on Parallelism in algorithms and architectures (SPAA), pp.302–310, June 2004.



Taisuke Izumi received the M.E. degrees in computer science from Osaka University in 2003. He is now a student of Graduate School of Information Science and Technology, Osaka University. His research interests include distributed algorithms. He is a student member of ACM and IEEE.



Toshimitsu Masuzawa received the B.E., M.E. and D.E. degrees in computer science from Osaka University in 1982, 1984 and 1987. He had worked at Osaka University during 1987–1994, and was an associate professor of Graduate School of Information Science, Nara Instituteof Science and Technology (NAIST) during 1994–2000. He is now a professor of Graduate School of Information Science and Technology, Osaka University. He was also a visiting associate professor of Department of Computer Science, Cornell University between 1993–1994. His research interests include distributed algorithms, parallel algorithms and graph theory. He is a member of ACM, IEEE, EATCS and the Information Processing Society of Japan.