Timed Atomic Broadcast Resiliet to Multiple Timing Faults

Taisuke IZUMI　Akinori SAITOH　Toshimitsu MASUZAWA
Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, 560-8531, Japan
E-mail:{t-izumi, saitoh, masuzawa}@ist.osaka-u.ac.jp

Abstract　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. This paper proposes a Δ-timed atomic broadcast algorithm in a partially synchronous system where communication delay is bounded (in the range of $[d-u, d]$ where d and u are known constants). The proposed algorithm can tolerate f_c crash faults and $\lceil (n-f_c)/2 \rceil - 1$ timing-faults, where n is the number of processes in the system. Moreover, the algorithm has a distinct advantage of guaranteeing that timing-faulty processes also delivers the same messages in the same order as the correct processes.

We also investigate the maximum number of faulty processes that can be tolerated. We show that no timed atomic broadcast algorithm can tolerate f_c crash faults and f_t timing faults, if $\lceil (n-f_c) \rceil /2 \leq f_t$ holds. The impossibility result implies that the proposed algorithm achieves the maximum resilience to both crash and timing faults.

1 Introduction

Atomic broadcast[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].

Partial 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 partial 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). Since message delivery latency strongly affects performance of applications based on the atomic broadcast, the timed atomic broadcast is very important.

However, as seen in most works on efficiency analysis of real distributed systems, it is natural to model communication delay as a probabilistic variable. In other words, it is actually inevitable, even though with low probability, that some messages experience communication delay larger than the upper bound in the partially synchronous model. The overdelay may prevent algorithms designed in the partially synchronous model from working correctly in real distributed systems. Even when the algorithms work regardless of the overdelay, the overdelay may cause a significant slowdown of the entire system and the timeliness of the timed atomic broadcast may be violated. Therefore, robustness for such overdelay is strongly desired in atomic broadcast algorithms.

In this paper, we consider timing faults of processes that cause overdelay on messages sent by the faulty processes, and investigate the possibility of the timed atomic broadcast in the partially synchronous model with the timing faults.

Cristian et al.[3] proposed an atomic broadcast algorithm resilient to timing faults in partially synchronous model with arbitrary topology. The algorithm guarantees that correct processes deliver the same messages in the same order if all correct processes are connected without going through faulty processes. However it guarantees nothing for messages delivered in the timing-faulty processes: the timing-faulty processes may deliver different messages or deliver messages in different order from the correct processes. 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, the timing-faulty processes should deliver the same messages in the same order.

In this paper, we propose a novel timed atomic broadcast algorithm in the partially synchronous model with crash and timing faults. The algorithm uses the reliable broadcast as a communication primitive, and ensures the atomic broadcast property for

all non-crashed processes including the timing-faulty processes. The algorithm can tolerate f_c crash faults and $\lceil (n - f_c)/2 \rceil - 1$ timing faults, where n is the number of processes. Moreover, the algorithm guarantees constant delivery latency of messages if the broadcasters are correct.

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_c crash faults and f_t timing faults if $\lceil (n - f_c) \rceil / 2 \leq f_t$ holds. The impossibility result implies that our timed atomic broadcast algorithm achieves the maximum resilience to both crash faults and timing faults.

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

2 Preliminaries

2.1 Distributed System

We adopt a model of a distributed system based on I/O automata[11] where sites, channels and a distributed system itself are modeled as I/O automata. For lack of space, this paper omits details of the model. We consider a partially synchronous distributed message-passing system consisting of n processes. Any pair of processes can communicate each other by exchanging messages through the channel. All channels are FIFO and reliable: each channel correctly transfers messages in the order they are sent. The system is partially synchronous, that is, all the messages sent by non-timing-faulty processes have communication delays in the range of $[d - u, d]$ where d and u ($0 \leq u \leq d$) are some constants every process knows 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. 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". While it is "correct", the process works correctly and messages sent by the process have communication delays in the range of $[d - u, d]$. When a process crashes, it changes its own fault state to "crashed", and ceases to operate. Once the fault state is changed to "crashed", it remains "crashed" forever. A process with the fault state of "timing-faulty" works correctly, but messages sent by the process may have communication delay greater than d. Without loss of generality, we can assume that the fault state of a timing-faulty process is always "timing-faulty" from the beginning, because messages sent by the timing-faulty process can have communication delays in the range of $[d - u, d]$. Notice that we introduce the faulty state only to represent the system configuration, and thus, 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 and f_t on the number of processes that can be timing-faulty. In this paper, we make the following assumption on the number of faulty processes:

Assumption 1 $f_t \leq \lceil (n - f_c)/2 \rceil - 1$

A system configuration is represented by all processes' states, all channels' states (i.e. messages under transmission on the channels) 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 partial 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 process p_j, then there exists an event (e', t') receiving M at process p_j such that $t + d - u \leq t' \leq t + d$, or p_j crashes by $t + d$. (3) If (e, t) is an event sending a message M from a timing-faulty process p_i to a process p_j, there exists event (e', t') receiving M at process p_j such that $t + d - u \leq t'$, or p_j eventually crashes. (4) If (e, t) is p_i's event setting its timer to τ, then there exists p_i's alarm event $(e', t + \tau)$, or p_i crashes by $t + \tau$. (5) If an internal or send event e is applicable at c_k, then there exists an event (e, t) such that $t = t_k$ (where t_k is the time assigned to the event e_k that changes the configuration to c_k).

The condition (2) and (3) implies that all messages are eventually received unless their destination processes crash. The condition (2) also implies that the delivery latency of messages sent by non-timing-faulty processes are in the range of $[d - u, d]$. The condition (5) implies that processing times of local computations are negligible, that is, several internal and send events can be executed in an instant.

2.2 Uniform FIFO Reliable Broadcast

Processes can use uniform FIFO reliable broadcast as a communication primitive. The uniform reliable broadcast guarantees that all non-crashed (may be timing-faulty) processes deliver a message if and only if the message is broadcast by a process. Formally, we define the uniform FIFO reliable broadcast in partially synchronous environment to be the broadcast satisfying the following specifications:

Nonfaulty Liveness : If a non-timing-faulty process broadcasts a message m at t, then each process delivers m at t' such that $t + d - u \leq t' \leq d$, or is crashed at $t + d$.

Faulty Liveness : If a timing-faulty process broadcasts a message m, then each process eventually delivers m at t' such that $t + d - u \leq t'$, or becomes crashed eventually.

Uniform Integrity : For any message m, every process delivers m at most once, only if some process broadcasts m.

Uniform FIFO Order : If a process broadcasts a message m before it broadcasts a message m', then no process delivers m' before m.

2.3 Δ-Timed Atomic Broadcast

Atomic broadcast is the broadcast ensuring that all non-crashed processes delver the same set of messages in the same order, which includes all messages broadcast by processes and no spurious messages. We define the Δ-timed atomic broadcast to be the atomic broadcast with an additional property, Δ-timeliness. The Δ-timeliness guarantees that delivery latency is bounded by some constant Δ. Formally, The timed atomic broadcast is the broadcast satisfying the following specifications:

Nonfaulty Δ-Liveness : If a non-timing-faulty process broadcasts a message m at t, then each process delivers m by $t + \Delta$ or is crashed at $t + \Delta$.

Faulty Liveness : If a timing-faulty process broadcasts a message m at t and, then each process eventually delivers m, or becomes crashed eventually.

Uniform Integrity : For any message m, every process delivers m at most once, only if some process broadcasts m.

Uniform Total Order : If 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'.

To distinguish the messages to be broadcast by the Δ-timed atomic broadcast algorithm from the messages the algorithm, we call the messages to be broadcast "ABcast message".

3 Δ-timed Atomic Broadcast Algorithm

3.1 Overview

In this section, we present the Δ-timed atomic broadcast algorithm using the uniform FIFO reliable broadcast. The 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 an ABcast message m by the Δ-timed atomic broadcast, and $TADeliver_i(m)$ is invoked by the Δ-timed atomic broadcast algorithm to deliver an ABcast message m.

In general, there is three main difficulties in implementing the Δ-timed atomic broadcast: reliable delivery (all processes deliver the same set of ABcast messages), totally-ordered delivery (all processes deliver the messages in the same order) and Δ-timeliness. Since we assume that processes can use the uniform FIFO reliable broadcast, it is easy to achieve the reliable delivery. Thus, the remaining problems to be resolved are the totally-ordered delivery and the Δ-timeliness. In the rest of this subsection, we briefly describe the idea for resolving these two problems.

Informally, the algorithm achieves the totally-ordered delivery 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 that are assigned to the same round are delivered in ascending order of their broadcaster's ID. If the two messages have the same broadcaster's ID, the one broadcast earlier is delivered first. Clearly, if all processes assign each ABcast message to a common round, 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, even in 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 problem, 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 process proposes a value at t, it decides a value at $t+\Delta$ or earlier (even if the process is timing-faulty). That is, the running time of the algorithm is not affected by overdelayed messages. (2) After execution of the algorithm, all non-crashed processes, including the timing-faulty processes, have a common decision.

The 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

3.2.1 Specification

Roughly speaking, the objective of the round synchronization algorithm is that each process periodically invokes the synchronous end-of-round event. The algorithm provides two primitives, $StartSync_i$ and $EOR_i(r)$. The event $StartSync_i$ and $EOR_i(r)$ are respectively invoked by the process p_i to initiates 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, there is some constant δ such that every process p_k invokes $EOR_k(0)$ or becomes crashed at $t+\delta$.

Eventual Initiation : If a timing-faulty process p_i invokes the $StartSync_i$ at t, every process p_k invokes $EOR_k(0)$ or becomes crashed eventually.

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

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

Synchrony : If a process p_i invokes $EOR_i(r)$ at t_i and a 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 any message sent by a non-timing-faulty process at the beginning of a round can be received by all non-crashed processes by the end of the round.

3.2.2 Algorithm

The key of the round synchronization algorithm Sync is to synchronize the occurrence of $EOR_*(0)$. 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, and it invokes $EOR_i(r+1)$ when the alarm raises. Then, each process invokes the $EOR_i(r+1)$ with the same timing difference as that of the rth. 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. Each process invokes $EOR_*(0)$ when it receives the invocation message first. Clearly, if p_i is non-timing-faulty, invocations of $EOR_*(0)$ are synchronized with difference of u or less. However, if p_i is timing-faulty, processes may receive the invocation messages with unbounded difference of receiving timing. Thus, the algorithm cannot attain the synchronization of $EOR_*(0)$ only by using the uniform FIFO reliable broadcast of the invocation messages. To resolve this problem, the algorithm relays an invocation message through distinct f_t+1 processes (including the process invoking $StartSync_*$). Then, a transmission path corresponds to a permutation of f_t+1 processes starting from p_i. When $StartSync_i$ is invoked, p_i tries to transmit the invocation message through all possible paths. However, the number of all possible paths is exponential to f_t and thus the message complexity is unacceptable. To reduce the message complexity, the algorithm stops the transmission along a path by some condition. In fact, our algorithm uses at most $n^2 f_t$ messages for one initiation.

Figure 1 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. The invocation messages is the pair (cnt, dom) where cnt is the number of times of relays and dom is the set of processes that have relayed this invocation message*. The invocation message $(k, *)$ is called a k-level invocation message. The process p_i that invokes $StartSync_i$ broadcasts the 0-level invocation message $(0, \{p_i\})$. When a process p_j receives a k-level invocation message $M = (k, dom)$, p_j broadcasts the $k+1$-level invocation message $(k+1, dom \cup \{p_j\})$ if (1) p_j is not in dom and (2) p_j has not yet broadcast the k'-level invocation message such that $k' \geq k+1$. The condition (1) is to prevent the path from forming a cycle. The condition (2) is to stop an unnecessary relay (in the

*Since $cnt = |dom| - 1$ always holds, cnt is redundant in practice. We introduce cnt only for ease of explanation.

```
variable
    rcnt : init 0
    round: init 0
    activated: init FALSE
transition function of process p_i

when StartSync_i occurs :
    Bcast_i(0, {p_i})

when Deliver_i(cnt, dom) occurs :
    if cnt = f_t and activated = FALSE then
        EOR_i(round)
        round ← round + 1
        activated = TRUE
        Timerset_i((f_t + 1)u + d, next)
    elseif p_i ∉ dom and rcnt ≤ cnt then
        Bcast_i(cnt + 1, dom ∪ {p_i})
        rcnt ← cnt + 1
    endif

when Alarm_i(next) occurs :
        EOR_i(round)
        round ← round + 1
        Timerset_i((f_t + 1)u + d, next)
```

Figure 1: Algorithm Sync

```
variable
    r : init 0 /* current round */
    buffer^[0..f_t] : init ∅

transition function of process p_i

when Propose_i(val) occurs :
    Bcast(0, {(val, p_i, {p_i})})

when EOR_i(*) occurs:
    if r = f_t then
        decide_i(p_i,intersection of value in buffer^r)
    else
        Bcast_i(r, buffer^r)
        r ← r + 1
    endif

when Deliver_i(c, buf) occurs :
    for all (val, pid, dom) in buf do
        if (val, pid, *) is not in buffer^c then
            if (p_i ∉ dom and r ≤ c) or c = f_t then
                buffer^c := buffer^c ∪ {(value, pid, dom ∪ p_i)}
            endif
        endif
    endfor
```

Figure 2: Algorithm TConsensus

correctness proof, we explain why it is unnecessary). Each process p_i invokes $EOR_i(0)$ when it receives the f_t-level invocation message first. When $EOR_i(r)$ occurs at t, the process p_i sets the timer to raise up at $(f_t+1)u+d$ later, which is the time when p_i invokes $EOR_i(r+1)$. For Sync, the following theorem holds (for lack of space, we omit the correctness proof).

Theorem 1 The algorithm Sync realizes the round synchronization,

About the length of each round, we propose the next corollary, which is easily obtained from theorem 1.

Corollary 1 For any $r > 0$, the length of round r is $(f_t + 1)u + d$, and the length of 0-th round is at most $(f_t + 1)d$ if a initiator is non-timing-faulty.

3.3 Δ-Timed Consensus Algorithm

3.3.1 Specification

The Δ-timed consensus is the consensus algorithm that completes its execution within constant Δ 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 *Delta*-timed atomic broadcast algorithm). We define the Δ-timed consensus algorithm to be the algorithm satisfying the following specification:

Timed Termination If $propose_i(V)$ occurs at t, the process p_i invokes $decide_i(*)$ or becomes crashed by $t + \Delta$.

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

Validity If every proposed value V_i contains a common element v, the decision value by every process contains v.

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 two assumptions:

Assumption 2 All processes invoke $propose_*(*)$ at the beginning of a same round.

Assumption 3 The length of each round is within a constant.

In the following discussion, let r_0 be the round that all processes invoke $propose_*(*)$.

3.3.2 Algorithm

In this subsection, we briefly describe the idea of the Δ-timed consensus algorithm Tconsensus. In the algorithm, each process gathers the proposed value from all processes. The decision value that the algorithm returns is the intersection of the all gathered values. To guarantee Uniform Agreement and Timed Termination, all non-crashed processes must gather the same set of values within constant time. The values proposed by non-timing-faulty processes can be

gathered within constant time. On the other hand, the values proposed by timing-faulty processes cannot be always gathered within constant time. Therefore, each process has to ignore the values proposed by timing-faulty processes, if the values are not gathered to all non-crashed processes. This is the main problem we have to consider.

The Δ-timed consensus algorithm Tconsensus is given in Figure 2. The idea for Tconsensus is similar to Sync: the algorithm transmits a proposed value by relaying through the distinct $f_t + 1$ processes in synchronous fashion. The message is the triple (val, p_i, dom) where val is the proposed value, p_i is the proposer's ID and dom is the set of process IDs that have already relayed this message. At the beginning of r_0, every process p_i broadcasts the message $(V_i, p_i, \{p_i\})$ where V_i is the value that p_i proposes. If a process p_i receives the messages $M = (val, p_j, dom)$ at round r, p_i records it to $buffer^r$, which is the variable storing the messages to be relayed at the next round, unless p_i is in dom, M is sent at round $r-2$ or before, or $(val, p_j, *)$ is in $buffer^r$ already. At the beginning of round $r+1 (0 \leq r \leq f_t - 1)$, each process broadcasts its $buffer^r$ with the current round number (that is $r+1$) [†] . At the end of round $r_0 + f_t$, each process returns a decision value by intersection the all proposed values stored in $buffer^{f_t}$. Due to lack of space, we omit the correctness proof of Tconsensus.

Theorem 2 Under the Assumptions 2 and 3, the algorithm Tconsensus realizes the Δ-timed consensus, and completes the execution in exactly $f_t + 1$ rounds.

3.4 Δ-timed Atomic Broadcast

3.4.1 Algorithm

Using Tconsensus and Sync, we realize the Δ-timed atomic broadcast algorithm TABcast. In the previous section, we have already stated the idea of TABcast Atomic Broadcast. 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 atomic broadcast, p_i invokes the $StartSync_i$ to start Sync if p_i has not yet recognized that Sync has already been executed. Each process maintains the local variable $sync$ to know whether Sync is running or not. The variable $sync$ is initially FALSE, and changes to TRUE when $StartSync_*$ or $EOR_*(0)$ occur.

[†] At the beginning of each round, each process broadcasts one "packed message" containing all messages to be broadcast at the round.

In our algorithm, every ABcast message has a distinct identifier. An identifier is a pair of broadcaster's ID and a serial number. Each process maintains its own serial number in a local variable sn. When $TABcast_i(m)$ occurs, the process broadcasts $M = (m, p_i, sn)$ by the uniform FIFO reliable broadcast. 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. Then, two or more Tconsensus algorithm may be concurrently executed. However, in our algorithm, each execution is distinguished from each other by its round number, and is executed independently. The value a process p_i proposes is the set of ABcast messages that are received by p_i but not yet delivered (that is $Received - Delivered$). When $decide_i(V_i)$ occurs, the process p_i removes the messages that have already delivered by p_i from V_i and delivers the rest in order of their broadcaster's ID. If two or more messages have the same broadcaster's ID, they are ordered by their serial numbers.

For lack of space, we omit the correctness proof of TABcast and only investigate the delivery latency Δ. Let a non-timing-faulty process p_i invoke $TABcast_i(m)$ at t in a round r. Since all processes receive m at round r or $r+1$, the ABcast message m is assigned to the round r or $r+1$ because of Validity of the Δ-timed consensus. From Theorem 2, Tconsensus completes each execution in exactly $f_t + 1$ rounds, and thus, m is delivered by the end of round $r + f_t + 2$. Then, we consider the two case. (1) In the case of $r > 0$, m's delivery latency is at most $((f_t + 1)u + d)(f_t + 2)$ because the length of each round is $(f_t + 1)u + d$ from Corollary 1. (2) In case of $r = 0$, since a non-timing-faulty process p_i invokes $StartSync_i$ at t or earlier, the length of the r-th round is at most $(f_t + 1)d$ from Corollary 1. Therefore, m's delivery latency is at most $(f_t + 1)d + ((f_t + 1)u + d)(f_t + 1)$.

From this observation, the following theorem holds;

Theorem 3 The algorithm TABcast realizes $(f_t + 1)d + (f_t + 2)(d + (f_t + 1)u)$-timed atomic broadcast.

4 Impossibility Result

In this section, we consider the maximum number of faulty-processes that a timed atomic broadcast algorithm can tolerate. Let f_c and f_t be the maximum numbers of crashed processes and timing-faulty processes respectively. We prove that no Δ-timed atomic broadcast can be designed if $\lceil (n - f_c) \rceil / 2 \leq f_t$

```
variable
    sn : init 0
    sync : init FALSE
    DeliverList, Received, Delivered : init ∅

transition function s of process p_i
when TABcast_i(m) occurs :
    if synced = FALSE then StartSync_i
    sn ← sn + 1
    M ← (m, p_i, sn)
    Bcast_i(M)

when Deliver_i(M) occurs :
    Received ← Received ∪ M

when Decide_i^r(D) occurs :
    DeliverList ← D − Delivered
    TADeliver all messages in DeliverList in some determin-
istic order
    /* It is ordered firstly by Broadcaster's ID, and secondary
by sn. */
    Delivered ← Delivered ∪ DeliverList

when EOR_i(r) occurs :
    if synced = FALSE then synced = TRUE
    propose_i^r(Received − Delvered)
```

Figure 3: Algorithm TABcast

holds. In the following proof, we use the technique of *shifting* [1], which changes the timing of occurrence of events appeared in an execution. For a timed execution E, a set of processes P, and a constant α, we define $shift(E, P, \alpha)$ to be the execution obtained from E as follows: (1)Every event (e, t) of E occurring at each process in P is replaced by $(e, t + \alpha)$, (2)other events of E are unchanged, and (3)those events are ordered in increasing order of the occurrence times. Notice that all shifted executions are not possible executions. However, for an timed execution E of an algorithm AL, $E_s = shift(E, P, \alpha)$ is possible execution of AL if delays of all messages broadcast by non-timing-faulty processes are in the range of $[d - u, d]$.

Theorem 4 For any Δ, no Δ-timed atomic broadcast algorithm can tolerate f_t timing-faulty processes and f_c crashed processes if $\lceil (n - f_c) \rceil / 2 \leq f_t$ holds.

Proof Suppose for contradiction that AL is a Δ-timed atomic broadcast algorithm that works correctly in case of $\lceil (n - f_c) \rceil / 2 \leq f_t$. We consider the execution E defined as follows (Fig. 4(1)): (1) f_c processes are crashed from the beginning of the execution. (2) Delay of each message is (2a) d if the sender is correct, (2b) d if both of the sender and the receiver are timing-faulty, and (2c)$\Delta + \epsilon (\epsilon > 0)$ if the sender is timing-faulty and the receiver is correct. (3) $TABcast_i(m)$ occurs at t, where p_i is a timing-faulty process. Clearly, E is a possible execution of AL. First, we show $TADeliver(p_i, p_i, m)$ occurs after $t + \Delta + \epsilon$ in E. Assume for contradiction that $TADeliver_i(m)$ occurs at $t' (\leq t + \Delta + \epsilon)$. Consider E' obtained from E by replacing the delays $\Delta + \epsilon$ of (2c) with $2\Delta + 3\epsilon$. Process p_i cannot recognize the difference between E and E' before t', and hence $TADeliver_i(m)$ occurs at t' also in E'. Moreover,we extend E' to E'' by adding an event $(TABcast_j(m'), t + \Delta + 2\epsilon)$, where p_j is a correct process (Fig. 4(2)). Since E is an execution of AL, E' and E'' are also possible executions of AL. Then, from the specification of the Δ-timed atomic broadcast, m' is delivered by all non-crashed processes at $t + 2\Delta + 2\epsilon$ or earlier. However, all correct processes cannot deliver m until $t + 2\Delta + 3\epsilon$ because they are unaware of the occurrence of the $TABcast_i(m)$ at that time. On the other hand, p_i delivers m at t', that is the time before m' is delivered. This violates Total Order of the Δ-timed atomic broadcast. A contradiction. Thus, $TADeliver_i(m)$ occurs after $t + \Delta + \epsilon$ in E.

Next we lead the main contradiction. Let P_t be the set of timing-faulty processes in E. We consider $E_s = shift(E, P_t, \Delta + \epsilon - d)$. In E_s, delay of messages are $\Delta + \epsilon$ if the sender is correct and the receiver is timing-faulty, and d otherwise. Notice that E_s may not be a possible execution of AL, because the delay of messages sent by correct processes to timing-faulty processes is $\Delta + \epsilon$ and may be larger than d. However, consider the execution E'_s (Fig. 4(3))obtained from E_s only by changing the fault states of processes as follows: Correct processes in E_s are changed to timing-faulty, and timing-faulty processes in E_s are changed to correct. It is clear that the execution E'_s is a possible execution of AL. Let f'_c and f'_t be the numbers of the crashed processes and the timing-faulty processes in E'_s. From assumption $\lceil (n - f_c) \rceil / 2 \leq f_t$, more than half of non-crashed processes in E are timing-faulty. This implies that $f'_c = f_c$ and $f'_t \leq f_t$ holds. Therefore, the algorithm AL must tolerate f'_c crashes and f'_t timing faults, that is, AL has to realizes Δ-timed atomic broadcast in E'_s. However, in E'_s, the ABcast message m sent by the correct process p_i has delivery latency of $\Delta + \epsilon$ or more at p_i. This contradicts to Nonfaulty Δ-Liveness of Δ-timed atomic broadcast. □

5 Concluding Remarks

We considered the Δ-Timed Atomic Broadcast resilient to crash and timing faults in partially synchronous environment ,where message delay is in the range of $[d - u, d]$, and timing faults. We presented a $(f_t + 1)d + (f_t + 2)(d + (f_t + 1)u)$-timed atomic broadcast algorithm that tolerates the f_c crashes and $\lceil (n - f_c)/2 \rceil - 1$ timing faults. The algorithm uses the uniform FIFO reliable broadcast as a commu-

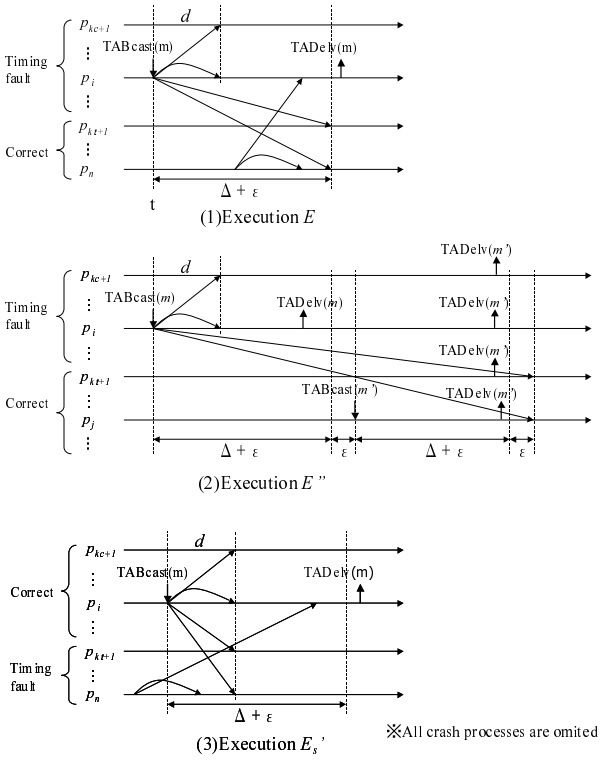

Figure 4: Execution E, E'' and E'_s

nication primitive and guarantees that the timing-faulty process also delivers the same messages in the same order as the correct processes. Moreover, we showed that there is no Δ-timed atomic broadcast algorithm if $\lceil (n - f_c) \rceil / 2 \leq f_t$ holds, where f_c and f_t are the numbers of crash processes and the timing-faulty processes respectively. This impossibility result implies that our algorithm attains the maximum resilience for both crash and timing 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

[1] H. Attiya and J. L. Welch. Sequential consistency versus linearizability. *ACM Trans. on Computer Systems*, 12(2):91–122, 1994.

[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2), 1996.

[3] F. Cristian, H. Aghali, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. *Information and Computation*, 118(1):158–179, 1995.

[4] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[5] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

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

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

[8] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 1993.

[9] T. Herman and T. Masuzawa. Stabilizing replicated search tree. In *Proc. of 15th International Symposium on Distributed Computing(DISC)*, pages 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*, E81-A(5), 1998.

[11] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[12] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proc. of International Conference on Parallel and Distributed Computing(Euro-Par)*, 1998.