PAPER

# Wait-Free Linearizable Distributed Shared Memory

**Sen MORIYA**[†*], ***Regular Member***, **Katsuro SUDA**[†**], ***Nonmember***, **Michiko INOUE**[†], ***Member***, **Toshimitsu MASUZAWA**[†], ***and*** **Hideo FUJIWARA**[†], ***Regular Members***

**SUMMARY** We consider a wait-free linearizable implementation of shared objects on a distributed message-passing system. We assume that the system provides each process with a local clock that runs at the same speed as global time and that all message delays are in the range $[d-u,d]$ where $d$ and $u$ $(0<u\leq d)$ are constants known to every process. We present four wait-free linearizable implementations of read/write registers on reliable and unreliable broadcast models. We also present two wait-free linearizable implementations of general objects on a reliable broadcast model. The efficiency of an implementation is measured by the worst-case response time for each operation of the implemented object. Response times of our wait-free implementations of read/write registers on a reliable broadcast model is better than a previously known implementation in which wait-freedom is not taken into account.

***key words:*** *synchronous message-passing system, distributed shared memory, linearizability, wait-freedom*

## 1. Introduction

How to provide logically shared objects in a distributed system is a fundamental problem on concurrent computing. A distributed system has good scalability while it needs low-level or complex control to shared data through message-passing paradigm. Logically shared objects greatly simplifies a design of a user program owing to its simple and general computing paradigm. A *distributed shared memory* consisting of such shared objects aims at providing useful and scalable programming environment for high-performance computing using multiple processors.

We implement logical shared objects which are used by multiple application processes concurrently. The implemented shared objects should provide some consistency for concurrent accesses. We consider *linearizable* implementations [1] of shared objects on a distributed message passing system. Informally, linearizability guarantees that operations to the implemented objects seem to be executed sequentially in some total order, and, for two operations such that one operation starts after the other operation completed, this total order preserves the real-time order on them. It has some good properties, such as *locality* and *nonblocking*. Locality means that a system is linearizable if each individual object is linearizable. Locality allows a concurrent system to be designed and constructed in a modular fashion; each of linearizable objects can be implemented, verified and executed independently. Nonblocking property means that a pending operation is never required to wait for another pending operation to complete. Nonblocking implies that linearizability is an appropriate condition for a system where real-time response is important.

An implementation of objects is said to be *wait-free* if any operations of the implemented objects are completed in finite time regardless of other processes' behavior [2]. We consider a wait-free linearizable implementation, which tolerates crash faults of any number of processes. James et al. showed that there are no wait-free linearizable implementations of read/write registers on a fully asynchronous system [3]. In this paper, we assume that all message delays in the system are in the range $[d-u,d]$ for some constants $d$ and $u(0<u\leq d)$, and these constants are known to every process. We also assume that the system provides each process with a local clock that runs at the same speed as global time. We consider two kinds of models on message exchange, a *reliable broadcast* and an *unreliable broadcast*. These two models differ in a guarantee on a case where a process crashes during its broadcast. A reliable broadcast model guarantees that every correct process receives a broadcasted message. In an unreliable broadcast model, if a process crashes during its broadcast, the message is not guaranteed to be sent to all correct processes. In such a case, some correct processes receive the message while the other correct processes may not receive it. We consider two kinds of models also on local clocks, *asynchronous clocks* and *u-synchronous clocks*. In a $u$-synchronous clock model, the difference between any pair of two local clock values is at most $u$. In an asynchronous clock model, we make no assumptions on such a difference. The efficiency of an implementation is measured by the worst-case response time $res\_time(op)$ for each operation $op$ of the implemented objects.

Several authors have investigated linearizable implementations of shared objects on a system in which no

Manuscript received October 4, 1999.

Manuscript revised March 16, 2000.

[†]The authors are with the Graduate School of Information Science, Nara Institute of Science and Technology(NAIST), Ikoma-shi, 630–0101 Japan.

[*]Presently, with NTT Communication Science Laboratories.

[**]Presently, with NTT Software Corporation.

**Table 1** Previous known results on linearizable implementations.

read/write registers

| bound | clocks | | *res_time* (*write*) | *res_time* (*read*) |
|---|---|---|---|---|
| upper | $u$-synchronous | [4] | $4u + \alpha m$ | $4u+ (1-\alpha)m^*$ |
| | | | $(0 < \alpha \leq 1,\ m = d - u)$ | |
| lower | asynchronous | [5] | $u/2$ | |
| | | [4] | The sum is $d + u/2$. | |
| | | | | $u/2$ |

FIFO queues

| | | | *res_time* (*dequeue*) | *res_time* (*enqueue*) |
|---|---|---|---|---|
| lower | asynchronous | [6] | $d + u/2$ if $u \leq (2/3)d$ | $u(n-1)/n$ |

general objects

| | | | *res_time* ($op_a$) | *res_time* ($op_v$) |
|---|---|---|---|---|
| upper | asynchronous | [6] | $u$ | $2d$ |

all message delays in $[d-u, d]$

$n$ : the number of processes

$^*$The bibliography [4] shows that the implementation achieves $res\_time(write) = 4u + \alpha(d-u)$ and $res\_time(read) = 4u + (1-\alpha)(d-u) + b$ where $b$ is an arbitrarily small constant implying the length of an interval for a broadcast. We ignore the length here and regard the small constant $b$ as zero.

**Table 2** Wait-free linearizable implementations in this paper.

read/write registers

| broadcast | clocks | *res_time* (*write*) | *res_time* (*read*) |
|---|---|---|---|
| reliable | asynchronous | $d$ | $u$ |
| | $u$-synchronous | $u + \alpha m'$ | $u + (1-\alpha)m'$ |
| | | $(0 \leq \alpha \leq 1,$ $m' = \max\{d-2u, 0\})$ | |
| unreliable | asynchronous | $d$ | $d$ |
| | $u$-synchronous | $u$ | $d$ |

general objects

| | | *res_time* ($op_a$) | *res_time* ($op_v$) |
|---|---|---|---|
| reliable | asynchronous | $u$ | $2d$ |
| | $u$-synchronous | $u$ | $d + u$ |

all message delays in $[d-u, d]$

processes crash and all message delays are in the range $[d-u, d]$. Previous results are shown in Table 1. Attiya et al. [5] and Mavronicolas et al. [4] showed lower bound results about any implementations of read/write registers on an asynchronous clock model. Mavronicolas et al. also presented an implementation of read/write registers on a $u$-synchronous clock model [4]. Inoue et al. presented an implementation of general objects such that $res\_time(op_a) = u$ and $res\_time(op_v) = 2d$ on an asynchronous clock model, where $op_a$ is any operation returning a unique response, called to be *ack-type*, and $op_v$ is any operation that is not ack-type, called to be *val-type*[6]. They also showed lower bound results about any implementations of FIFO queues on an asynchronous clock model [6].

In this paper, we propose wait-free linearizable implementations. First, we present four wait-free linearizable implementations of read/write registers. Two of them are implementations using reliable broadcasts on an asynchronous and $u$-synchronous clock models. The other two are implementations on an unreliable broadcast model, on an asynchronous and $u$-synchronous clock models. Furthermore, we present two wait-free linearizable implementations of general objects on an asynchronous and $u$-synchronous clock models using reliable broadcasts. The implementation of general objects on an asynchronous clock model is based on an implementation in [6], which is not wait-free. We show our results in Table 2. All of them are wait-free, that is, they tolerate crash faults of any number of processes. Moreover, the response time of our implementation on a $u$-synchronous clock model using reliable broadcasts is more efficient than previous known implementation on a $u$-synchronous clock model in [4].

## 2. Definitions

### 2.1 System

A distributed message-passing system consists of multiple processes and a communication network. A process communicates with any other processes by exchanging messages through the network. No messages are omitted in exchange of messages. All message delays are in the range $[d-u, d]$ for some constants $d$ and $u$ $(0 < u \leq d)$ where every process knows these $d$ and $u$. Each process has a local clock that runs at the same rate as global time[†]. The process obtains local time from its local clock. The process has a timer based on its local clock, and it can set an alarm by the timer. We assume the difference between any pair of local clock values in a system is at most $\epsilon$ for some constant $\epsilon$. Such a model is called *$\epsilon$-synchronous clock* model. If $\epsilon$ is the infinity, the model is called *asynchronous clock* model.

We assume that a process may crash. After a process crashes, it ceases to operate. We consider two kinds of models about exchange of messages, a *reliable broadcast model* and an *unreliable broadcast model*. In both models, processes communicate with each other by sending and receiving messages. A reliable broadcast means that every broadcasted message is guaranteed to be received by all correct processes. Therefore, in a reliable broadcast model, a broadcast is modeled as one event where the process sends a message to all processes atomically. On the other hand, a broadcast is implemented by sending a message sequentially to all processes in an unreliable broadcast model. In this model, if a process gets faulty during its broadcast, some correct processes receives a message while the other correct processes do not receive it.

A process $p$ is modeled as a state machine. Its

[†]We use system-wide global time to specify system behavior. Note that the global time is introduced only for specification and no processes can use it.

state changes when some event occurs at $p$. A *system configuration* (or we call just *configuration*) is defined as all process states, a set $\mathcal{N}$ of in-transit messages and sets $\mathcal{A}_p$ of alarms which have been set and have not gone off at each process $p$. An in-transit message is a triple $(M, s, r)$ where $M$ is a message, $s$ is the sender, and $r$ is the destination. An alarm is a pair $(K, t)$ where $K$ is the type of an alarm and $t$ is the local time for the alarm to go off. Each process has the following events.

- Communication events: events in which a process sends or receives a message. In a reliable broadcast model, broadcast events $BroadCast(p, M)$ and receive events $Receive(p, q, M)$ can occur. In an unreliable broadcast model, send events $Send(p, q, M)$ and receive events $Receive(p, q, M)$ can occur.

  $Send(p, q, M)$ : Process $p$ sends a message $M$ to process $q$. A triple $(M, p, q)$ is added to $\mathcal{N}$.

  $BroadCast(p, M)$ : Process $p$ broadcasts a message $M$, that is, it sends $M$ to all processes[†]. For each process $q$, $(M, p, q)$ is added to $\mathcal{N}$.

  $Receive(p, q, M)$ : Process $p$ receives a message $M$ from process $q$. A triple $(M, q, p)$ is removed from $\mathcal{N}$.

- Time events: events about the local clock.

  $TimerSet(p, \bar{t}, K)$ : Process $p$ sets its timer of type $K$ to go off after $\bar{t}$. When an event $TimerSet(p, \bar{t}, K)$ occurs at local time $t$, a pair $(K, t + \bar{t})$ is added to $\mathcal{A}_p$.

  $Alarm(p, K)$ : An alarm of type $K$ occurs at process $p$. When an event $Alarm(p, K)$ occurs at local time $t$, a pair $(K, t)$ is removed from $\mathcal{A}_p$.

  $ReadClock(p, s)$ : Process $p$ obtains the clock value $s$ from its local clock.

- $Stop(p)$ : Process $p$ crashes. In this event, $p$'s state changes to *fault state* and $p$ ceases to operate.
- A process communicates also with the outside of the system, which we call *environment*. We describe events about communication between a process and the environment later.

The receive, alarm and stop events are *input* events, which arise out of the process's control.

The *system history* (or we call just *history*) is defined as a finite or infinite alternating sequence of configurations and occurrences of events, $H = c_0, (e_1, t_1), c_1, \cdots, c_k, (e_{k+1}, t_{k+1}), c_{k+1}, \cdots$, where each $c_k (k \geq 0)$ is a configuration, $(e_k, t_k)(k \geq 1)$ is an occurrence of an event, $e_k$ is an event and $t_k$ is global time when $e_k$ occurs. Each $t_k$ is denoted by $time(e_k)$. A process $p$'s state of a configuration $c_k$ is a projection of $c_k$ to $p$, denoted by $c_k|p$. The first configuration $c_0$ is called an initial configuration, in which all processes are in the initial state, and $\mathcal{N}$ and $\mathcal{A}_p$ for any process $p$ are empty. For each $k$, $t_k \leq t_{k+1}$ holds. A history $H$ implies that, for each $k (k \geq 0)$, an event $e_{k+1}$ occurs at some process $p$ at $t_{k+1}$ in a configuration $c_k$, and $p$'s state changes from $c_k|p$ to $c_{k+1}|p$ (and $\mathcal{N}$ or $\mathcal{A}_p$ also may change). To be simplified, all events in a history are distinct. We assume the following conditions on any history $H$.

- If $\mathcal{A}_p$ contains a pair $(K, t)$, $Alarm(p, K)$ occurs or $p$ is in a fault state at local time $t$. Conversely, $Alarm(p, K)$ occurs at $t$, only if $(K, t)$ is in $\mathcal{A}_p$.
- If a triple $(M, p, q)$ is added to $\mathcal{N}$ at $T$, $Receive(q, p, M)$ occurs in $[T + d - u, T + d]$ or $q$ is a fault state at $T + d$. Only if $\mathcal{N}$ contains a triple $(M, p, q)$, a receive event $Receive(q, p, M)$ occurs.

## 2.2 Implementation of an Object

We define a *deterministic shared object* (we call just *object* in the following). An object is a data structure to which multiple processes can access concurrently. An object has a unique *name* and a *type*. The type is a tuple $(OP, RES, Q, q_0, \delta)$, where $OP$ is a set of operations, $RES$ is a set of responses, $Q$ is a set of states, $q_0$ is an initial state, and $\delta : Q \times OP \rightarrow Q \times RES$ is a function called *sequential specification*. The sequential specification defines a behavior of the object when operations are applied sequentially: if an operation *op* is applied to the object in a state $s$, the object changes its state to $s'$ and returns the response *res* where $\delta(s, op) = (s', res)$ holds. Such an object is called to be *deterministic*, since the sequential specification is a function. If an operation *op* always returns a unique response, that is $|\{res | \exists s, s' [\delta(s, op) = (s', res)]\}| = 1$, *op* is called to be *ack-type*. An operation is called to be *val-type*, if it is not ack-type. In the following, $op_a$ denotes any ack-type operation and $op_v$ denotes any val-type operation.

Next, we define an *implementation* of an object $O$ of type $(OP, RES, Q, q_0, \delta)$. We implement a virtual shared object which is used concurrently by environment. Figure 1 illustrates the implementation. An object is implemented by a set of processes $\{p_1, p_2, \cdots, p_n\}$. A subscript $i$ of each $p_i$ is the process identifier. Environment can access an object by communicating with a process $p_i$. Communication between environment and a process $p_i$ is modeled as the following events.

- $Invoke(p_i, op)$ : Environment calls $p_i$ to apply an operation $op (\in OP)$ to the object $O$.
- $Response(p_i, res)$ : Process $p_i$ returns a response $res (\in RES)$ for an invocation to environment.

The invoke event is an input event. We assume the

[†]For convenience, we assume that a process sends a message to all processes including itself by a broadcast.

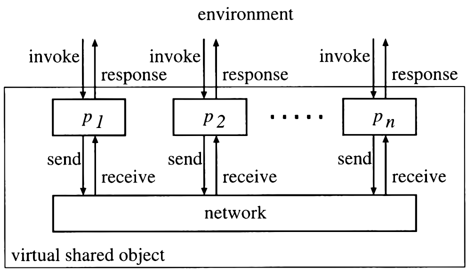


**Fig. 1** Implementation of a shared object.

following condition about communication between environment and $p_i$ on any history $H$.

- Once environment invokes an operation to a process $p_i$, it does not invoke the next operation to $p_i$ until $p_i$ returns a response for the former invocation.

To return consistent responses, the processes exchange messages with each other.

For each $p_i$, we consider a sequence obtained by restricting a history to $p_i$'s invoke and response events. In an implementation, a process returns a response if and only if an operation is invoked to the process. Therefore, the obtained sequence should be an alternating sequence $Inv_1, Res_1, Inv_2, Res_2, \cdots$ where $Inv_k$ is an invoke event and $Res_k$ is a response event for each $k(k \geq 1)$. For each invoke event $Inv_k$, the next event $Res_k$ is called to be a *corresponding* response event. A pair of events $(Inv_k, Res_k)$ is called an *operation execution.* An invoke event that has no corresponding response events is said to be *pending*. If an invoke event of an operation is not pending, it is said to be *completed*.

The implemented object should provide some consistency for concurrent accesses. We adopt *linearizability* as a consistency condition of an implementation of an object. Herlihy et al. showed a *local* property of linearizability [1]. The locality means that an implementation of multiple shared objects is linearizable if and only if each object is implemented linearizably. This says that objects can be implemented and verified independently. In this paper, we consider an implementation of one object, and we define an implementation of only one object. From locality, we can implement multiple objects from our implementations of one object.

Now we define linearizability and wait-freedom. We consider a sequence of operation executions $\tau = (Inv_1, Res_1), (Inv_2, Res_2), \cdots$. For each $k(k \geq 1)$, let $Inv_k$ and $Res_k$ be $Invoke(p_{i_k}, op_k)$ and $Response(p_{i_k}, res_k)$ respectively. For an object $O$ of type $(OP, RES, Q, q_0, \delta)$, if there exists a sequence $\theta = q_0, q_1, \cdots$ of states of $O$, where $\delta(q_{k-1}, op_k) = (q_k, res_k)$ holds for each $k \geq 1$, $\tau$ is said to be *legal*. In a system history $H$, if $time(Res_k) < time(Inv_l)$ holds for two operation executions $op_k = (Inv_k, Res_k)$ and $op_l = (Inv_l, Res_l)$, we say that $op_k$ precedes $op_l$ and $op_l$ succeeds $op_k$, denoted by $op_k \xrightarrow{H} op_l$. A sequence obtained by restricting a history $H$ to completed invoke and response events is denoted by $complete(H)$.

**Definition 1:** A history $H$ is said to be linearizable, if there exists a history $H'$ that satisfies the followings.

- The history $H'$ is obtained from $H$ by adding corresponding response events for some (possibly empty) pending invoke events.
- There exists a legal sequence $\tau$ consisting of all operation executions in $complete(H')$ such that, for any operation executions $op_1$ and $op_2$ satisfying $op_1 \xrightarrow{complete(H')} op_2$, $op_1$ precedes $op_2$ in $\tau$. □

**Definition 2:** For an implementation $I$, if any possible system history $H$ is linearizable, the implementation $I$ is said to be linearizable. □

**Definition 3:** A history $H$ is said to be *wait-free*, if any invoke event $Inv$ in a history $H$ satisfies one of the followings.

- There exists a corresponding response event.
- For the process $p_i$ in which $Inv$ occurs, $Stop(p_i)$ occurs after $Inv$. □

**Definition 4:** For an implementation $I$, if any possible system history $H$ is wait-free, the implementation $I$ is said to be wait-free. □

The efficiency of an implementation $I$ is measured by the worst-case response times of operation executions. For an operation execution $(Inv, Res)$, we define the response time as $time(Res) - time(Inv)$. Let $OPE(H)$ denote a set of operation executions that appear in a history $H$. For an operation execution $ope = (Inv, Res)$, let $ope.op$ denote an operation invoked in $Inv$. For an implementation $I$ of an object $O$ of type $(OP, RES, Q, q_0, \delta)$, we define the worst-case response time of $op$, denoted by $res\_time(op)$, as $\max\{res\_time(ope) | ope \in OPE(H), ope.op = op, H \text{ is a history of } I\}$.

## 3. Read/Write Registers

In this section, we present four implementations of a read/write register. We show the type of a read/write register on a domain $D$ in Fig. 2. The efficiency of a read/write register is measured by $res\_time(read)$ and $res\_time(write)$ where $res\_time(write) = \max\{res\_time(write(v))\}$. Two implementations described in the first subsection use reliable broadcasts, and two implementations described in the second subsection does not use reliable broadcasts.

In all implementations, each process keeps a local copy of a read/write register. When a write operation is invoked at a process, the process assigns a timestamp

```
type of read/write register $(OP, RES, Q, q_0, \delta)$
    $OP = \{write(v) | v \in D\} \cup \{read\}$
    $RES = \{\text{ack}\} \cup D$
    $Q = D$
    $\forall v, v' \quad \delta(v, write(v')) = (v', \text{ack})$
    $\forall v \quad \delta(v, read) = (v, v)$
```

**Fig. 2** Type of a read/write register on a domain $D$.

to the write operation and broadcasts an update message that contains the written value and the timestamp of the write operation. A process updates its local copy according to received update messages. The update depends on a timestamp assigned to the write operation. In a read operation, the value of its local copy at some time during the operation is returned.

We describe each program code by event-driven form for input events. A series of each event and the succeeding internal changes of the state is atomic, that is, the process does not crash during the series. If multiple input events occur at the same time, they are handled in an order such that they appear in the described code except a stop event.

### 3.1 Implementations Using Reliable Broadcasts

In this subsection, we show two implementations using reliable broadcasts. The first one is an implementation on an asynchronous clock model and the other is on a $u$-synchronous clock model.

#### 3.1.1 Asynchronous Clocks

The first implementation, which we call $register_{RB-AC}$ (for "reliable broadcast, asynchronous clocks"), provides a read operation with response time $u$ and a write operation with response time $d$. The program code for $p_i$ is given in Fig. 3.

First we assume that no faults occur and later consider the case where any number of processes crash. In a write operation, the process broadcasts an update message first. When a process receives the update message, it updates its local copy according to the message. The write operation is completed by returning ack after $d$ since its invocation. In a read operation, the value of its local copy at invoked time is returned. After $u$ since its invocation, the process returns the value.

In this implementation, a process uses an integer *count* as a timestamp. The integer *count* increases by one when a write operation is invoked at the process. If the timestamp contained in a received update message is greater than the process's *count*, the process sets its *count* to the timestamp. Since any message delay is not greater than $d$, a write operation execution $W_2$ succeeding another one $W_1$ is assigned a greater timestamp than $W_1$'s timestamp.

A process sets its local copy to the written value contained in each update message in order of its timestamp (breaking tie by process identifiers). However, there is a case where a process receives an update message after it updates its local copy according to some update message with a greater timestamp. In this case, the process considers that such an update message has already been handled and the value is overwritten by the write operation with a greater timestamp. Thus the process ignores such an update message. An update message broadcasted at time $t$ of a write operation $W$ is received in $[t+d-u, t+d]$. Therefore, at each process, the update message for $W$ is handled in this interval, or it is ignored. For two read operations executions $R$ and $R'$ such that $R'$ precedes $R$, it is guaranteed that $R$ returns the value written by the write operation with timestamp greater than or equal to $R'$.

```
data type
timestamp=(integer, process identifier);

variables
$count$, type integer, init 0;
$res\_val$, type value of the register ;
$last\_up\_ts$, type timestamp, init $(0, 0)$;
$local\_copy$, type value of the register, init initial value of
    the register;

transition functions of process $p_i$
$Invoke(p_i, write(v))$ :
    $count := count + 1$;
    $BroadCast(p_i, \text{update}(v, (count, i)))$; /* update message */
    $TimerSet(p_i, d, \text{WRITE})$;

$Invoke(p_i, read)$ :
    $res\_val := local\_copy$;
    $TimerSet(p_i, u, \text{READ})$;

$Receive(p_i, p_j, \text{update}(v, (recvd\_count, recvd\_uid)))$ :
    $count := \max(count, recvd\_count)$;
    if $last\_up\_ts <^{\dagger} (recvd\_count, recvd\_uid)$
        then $local\_copy := v$; $last\_up\_ts := (recvd\_count, recvd\_uid)$;

$Alarm(p_i, \text{WRITE})$ :
    $Response(p_i, \text{ack})$;

$Alarm(p_i, \text{READ})$ :
    $Response(p_i, res\_val)$;

$Stop(p_i)$ :
    No events can happen after this event.
```

†A relation $(a_1, b_1) < (a_2, b_2)$ means that $a_1 < a_2$, or $a_1 = a_2$ and $b_1 < b_2$.

**Fig. 3** Implementation $register_{RB-AC}$. (The code for $p_i$.)

Next, we consider the case where any number of processes crash. If a process crashes during an operation, the operation is left pending. In implementation $register_{RB-AC}$, only a write operation influences the other processes. In a reliable broadcast model, if and only if some process broadcasts an update message in a write operation execution, all correct processes receive the update message. This does not depend on whether the write operation is completed. Therefore, implementation $register_{RB-AC}$ works correctly in a case where some processes crash.

Now we prove that $register_{RB-AC}$ is a wait-free linearizable implementation of a read/write register. We show that any possible history $H$ in $register_{RB-AC}$

is linearizable and wait-free. A pending invoke event of a write operation $W$ is said to be *valid* if some response event of a read operation execution returns the value written by $W$. Let $H'$ be a history in which a response event corresponding to each valid pending event $e$ in $H$ is added at $time(e) + d$.

**Lemma 1:** Let $Inv$ be an invoke event of any write operation execution $W$ in a history $complete(H')$. Let $M$ be the update message of $W$. If a process $p_i$ updates its local copy according to $M$, it is done in $[time(Inv)+d-u, time(Inv)+d]$. □

For a write operation execution $W$, let $ts(W)$ denote a pair of $W$'s timestamp and the process identifier.

**Lemma 2:** For write operation executions $W_1$ and $W_2$ in $complete(H')$, if $W_1 \overset{complete(H')}{\longrightarrow} W_2$, then $ts(W_1) < ts(W_2)$ holds. □

For a read operation $R$, let $Write(R)$ be the write operation execution whose written value $R$ returns.

**Lemma 3:** Let $R_1$ and $R_2$ be read operation executions in $complete(H')$ satisfying $R_1 \overset{complete(H')}{\longrightarrow} R_2$. Then, $ts(Write(R_1)) < ts(Write(R_2))$ or $Write(R_1) = Write(R_2)$ holds. □

**Theorem 4:** The implementation $register_{RB-AC}$ is a wait-free linearizable implementation of a read/write register which achieves $res\_time(write) = d$ and $res\_time\ (read) = u$ on an asynchronous clock model using reliable broadcasts.

*Proof.* It is trivial that the implementation is wait-free and achieves $res\_time(write) = d$ and $res\_time(read) = u$. We show the implementation is linearizable in the rest of the proof.

We construct a legal sequence $\tau$ that consists of all operation executions in $complete(H')$, and show that, for any operation executions $op_1$ and $op_2$, $op_1$ precedes $op_2$ in $\tau$ if $op_1 \overset{complete(H')}{\longrightarrow} op_2$. First, we assume that a sequence $\tau$ begins with a virtual write operation $W_0$ that writes the initial value, and arrange all write operation executions in $complete(H')$ after $W_0$ in order of their timestamps. Next, we put each read operation $R$ between write operation executions in order of its invocation time by the following way to accomplish constructing a sequence $\tau$. Let $W_k$ be $Write(R)$ and $W_{k+1}$ be the write operation execution that we have arranged next to $W_k$. We put $R$ immediately before $W_{k+1}$.

Now we show that for any operation executions $op_1$ and $op_2$, if $op_1 \overset{complete(H')}{\longrightarrow} op_2$, $op_1$ precedes $op_2$ in $\tau$.

(i) $op_1$ and $op_2$ are write operation executions $W_1$ and $W_2$: From Lemma 2, if $W_1 \overset{complete(H')}{\longrightarrow} W_2$, then $ts(W_1) < ts(W_2)$ holds. From the rule to construct $\tau$, $W_1$ precedes $W_2$ in $\tau$.

(ii) $op_1$ and $op_2$ are read operation executions $R_1$ and $R_2$: Let $W_1$ and $W_2$ be $Write(R_1)$ and $Write(R_2)$ respectively. From Lemma 3, $ts(W_1) < ts(W_2)$ or $W_1 = W_2$ holds. From the rule to construct $\tau$, $R_1$ precedes $R_2$ in $\tau$ in both cases.

(iii) $op_1$ is a write operation execution $W_1$ and $op_2$ is a read operation execution $R_1$: Let $W_1$ be $(Inv_{W_1}, Res_{W_1})$ and $R_1$ be $(Inv_{R_1}, Res_{R_1})$. Let $R_1$ be an operation at $p_i$. From $W_1 \overset{complete(H')}{\longrightarrow} R_1$, $time(Inv_{W_1}) + d = time(Res_{W_1}) < time(Inv_{R_1})$ holds. The process $p_i$ receives the update message for $W_1$ at $time(Inv_{W_1}) + d$ or before, and then returns the response for $R_1$ at $time(Res_{R_1}) = time(Inv_{R_1}) + u$. From the implementation, $p_i$'s *count* at $time(Res_{W_1})$ is greater than or equal to $ts(W_1)$ and it does not decrease. Therefore, $Write(R_1)$'s timestamp is greater than or equal to $ts(W_1)$. This implies that $R_1$ succeeds $W_1$ in $\tau$.

(iv) $op_1$ is a read operation execution $R_1$ and $op_2$ is a write operation execution $W_1$: Let $W_1$ be $(Inv_{W_1}, Res_{W_1})$ and $R_1$ be $(Inv_{R_1}, Res_{R_1})$. Let $W_2 = Write(R_1)$ be $(Inv_{W_2}, Res_{W_2})$. From Lemma 1, $W_2$ is invoked before $time(Inv_{R_1}) + u - d$. From $time(Inv_{W_1}) > time(Inv_{R_1}) + u$, $time(Inv_{W_1}) > time(Res_{W_2})$ holds. Since $W_2 \overset{complete(H')}{\longrightarrow} W_1$ holds, $R_1$ is put between $W_2$ and $W_1$ in $\tau$.

Finally, from the rule to construct $\tau$, all read operations return the value written by the latest write operation. Therefore, $\tau$ is legal. □

### 3.1.2 $u$-Synchronous Clocks

The next implementation, which we call $register_{RB-uC}$ (for "reliable broadcast, $u$-synchronous clocks"), provides a read operation with response time $u + (1-\alpha)\max\{d-2u, 0\}$ and a write operation with response time $u + \alpha \cdot \max\{d-2u, 0\}$ where $\alpha (0 \leq \alpha \leq 1)$ is a parameter. The program code for $p_i$ is given in Fig. 4.

Implementation $register_{RB-uC}$ is based on implementation $register_{RB-AC}$. The difference between $register_{RB-uC}$ and $register_{RB-AC}$ is that we use a local clock value as a timestamp in $register_{RB-uC}$ instead of *count* in $register_{RB-AC}$. In $u$-synchronous clock model, the difference between any pair of local clock values is at most $u$. Therefore, it is guaranteed that a preceeding write operation has a smaller timestamp if response time of a write operation is $u$ or more. For a write operation execution $W = (Inv_W, Res_W)$, any process $p_i$ updates its local copy according to $W$'s update message in $[time(Inv_W)+d-u, time(Inv_W)+d]$ or ignores the update message. Now let $R$ be a read operation execution and $W$ be $Write(R)$. For any read operation execution $R'$ preceeding $R$, it is guaranteed that $W$ has a greater timestamp than $Write(R')$ if response time of read operation is $u$ or more. Further-

```
constant
$|W| = u + \alpha \cdot \max\{d - 2u, 0\}$, $|R| = u + (1-\alpha)\max\{d - 2u, 0\}$

transition functions of process $p_i$
$Invoke(p_i, write(v))$ :
    $ReadClock(p_i, local\_cl)$;
    $BroadCast(p_i, \mathrm{update}(v, (local\_cl, i)))$; /* update message */
    $TimerSet(p_i, |W|, \mathrm{WRITE})$;

$Invoke(p_i, read)$ :
    $TimerSet(p_i, \min\{|R|, d-u\}, \mathrm{SET\_VAL})$;
    $TimerSet(p_i, |R|, \mathrm{READ})$;

$Receive(p_i, p_j, \mathrm{update}(v, (recvd\_cl, recvd\_uid)))$ :
    if $last\_up\_ts < (recvd\_cl, recvd\_uid)$
        then $local\_copy := v$; $last\_up\_ts := (recvd\_cl, recvd\_uid)$;

$Alarm(p_i, \mathrm{WRITE})$ :
    $Response(p_i, \mathrm{ack})$;

$Alarm(p_i, \mathrm{SET\_VAL})$ :
    $res\_val := local\_copy$;

$Alarm(p_i, \mathrm{READ})$ :
    $Response(p_i, res\_val)$;

$Stop(p_i)$ :
    No events can happen after this event.
```

**Fig. 4** Implementation $register_{RB-uC}$. (The code for $p_i$.)

more, the time when $res\_val$ is stored in a read operation guarantees that any write operation execution preceeding $R$ except $W$ has a smaller timestamp than $W$. For any write operation execution $W'$ preceeding $R$, it is guaranteed that $W$ has a greater timestamp than $W'$ if the sum of response time of read and write operations is $d$ or more.

Lemmas 1–3 also hold in $register_{RB-uC}$. From these lemmas, we can construct a legal sequence $\tau$ for any possible history $H$ like Theorem 4. Therefore, $register_{RB-uC}$ is a wait-free linearizable implementation of a read/write register.

**Theorem 5:** The implementation $register_{RB-uC}$ is a wait-free linearizable implementation of a read/write register which achieves $res\_time(write) = u + \alpha \cdot \max\{d - 2u, 0\}$ and $res\_time\ (read) = u + (1 - \alpha)\max\{d - 2u, 0\}$ $(0 \leq \alpha \leq 1)$ on a $u$-synchronous clock model using reliable broadcasts. □

## 3.2 Implementations on Unreliable Broadcast Model

In this subsection, we show two implementations on an unreliable broadcast model. The first one is on asynchronous clock model, and the other is on $u$-synchronous clock model. A message broadcasted in this model is not guaranteed to be received by all correct processes if the sender crashes during its broadcasting. A message which all correct processes do not receive is called to be *incompletely broadcasted*.

### 3.2.1 Asynchronous Clocks

The first implementation, which we call $register_{UB-AC}$

```
transition functions of process $p_i$
$Invoke(p_i, \mathrm{Write}(v))$ :
    $count := count + 1$;
    for $j = 1$ to $n$  /* broadcasting an original update message */
        do $Send(p_i, p_j, \mathrm{update}(v, (count, i)))$;
    $TimerSet(p_i, d, \mathrm{WRITE})$;

$Invoke(p_i, \mathrm{Read})$ :
    $res\_val := local\_copy$;
    for $j = 1$ to $n$  /* broadcasting an additional update message */
        do $Send(p_i, p_j, \mathrm{update}(res\_val, last\_up\_ts))$;
    $TimerSet(p_i, d, \mathrm{READ})$;

$Receive(p_i, \mathrm{update}(v, (recvd\_count, recvd\_uid)))$ :
    $count := \max(count, recvd\_count)$;
    if $last\_up\_ts < (recvd\_count, recvd\_uid)$
        then $local\_copy := v$; $last\_up\_ts := (recvd\_count, recvd\_uid)$;

$Alarm(p_i, \mathrm{WRITE})$ :
    $Response(p_i, \mathrm{ack})$;

$Alarm(p_i, \mathrm{READ})$ :
    $Response(p_i, res\_val)$;

$Stop(p_i)$ :
    No events can happen after this event.
```

**Fig. 5** Implementation $register_{UB-AC}$. (The code for $p_i$.)

(for "unreliable broadcast, asynchronous clocks") provides a read operation with response time $d$ and a write operations with response time $d$. The program code for $p_i$ is given in Fig. 5.

First, we consider to apply $register_{RB-AC}$ to an unreliable broadcast and asynchronous clock model. An incompletely broadcasted update message $M$ does not cause update of a local copy to all correct processes. This violates linearlizability as follows. A correct process that receives $M$ returns a value $v$ contained in $M$ for a read operation execution $R$ after receiving $M$. On the other hand, a correct process that does not receive $M$ cannot return $v$ for a read operation execution succeeding $R$.

In an implementation $register_{UB-AC}$, a process where a read operation execution $R$ occurs relays information about the update message that contained the returned value by sending an *additional* update message to the others. Every process never fails to know information about an update message that contains the value returned by $R$. Even if a correct process does not receive an original update message, it updates its local copy according to the additional update message.

Here we explain how a process relays such necessary information. In each read operation $R$, a process assigns the timestamp (containing the process identifier) of $Write(R)$. If $R$ returns the initial value, $(0, 0)$ is assigned. A process decides a returned value when the invocation occurred, and then the process broadcasts an additional update message which contains the returned value and $Write(R)$'s timestamp. This is a different point from $register_{RB-AC}$ or $register_{RB-uC}$. The process returns the value after $d$ since its invocation. If some process completes a read operation, it is guaranteed that all correct processes obtain information about the value returned in the operation. A process sets its local copy to the value contained in each original or additional update message in order of its timestamp. When a process receives an update message, only if the message has a greater timestamp than the latest update, it updates its local copy according to the message.

To prove that $register_{UB-AC}$ is a wait-free linearizable implementation of a read/write register, we consider any possible history $H$. For the history $H$, we construct a history $H'$ as follows. We consider any pending write operation execution $W$ whose value some read operation returns. Let $R$ be the first read operation that returns $W$'s value, and add $W$'s response event at the same time as $R$'s response event in a history $H'$. Let $ts(op)$ denote a pair of the timestamp and the process identifier assigned to an operation execution $op$ including a read operation execution. Note that $ts(Write(R)) = ts(R)$ holds for a read operation $R$. Let $op\_ts(op)$ be a pair of *count* and the process identifier at invoked time. Note that $ts(W) = op\_ts(W)$ holds for a write operation $W$. Then, the next lemma holds.

**Lemma 6:** For any operation executions $op_1$ and $op_2$, if $op_1 \overset{complete(H')}{\longrightarrow} op_2$ holds, $(ts(op_1), op\_ts(op_1)) < (ts(op_2), op\_ts(op_2))$ holds. □

**Theorem 7:** The implementation $register_{UB-AC}$ is a wait-free linearizable implementation of a read/write register which achieves $res\_time\ (write) = d$ and $res\_time\ (read) = d$ on an asynchronous clock and unreliable broadcast model.

*Proof.* It is trivial that the implementation is wait-free and achieves $res\_time(write) = d$ and $res\_time(read) = d$. We show the implementation is linearizable in the rest of the proof.

We construct a legal sequence $\tau$. First, we assume that there is a virtual write operation $W_0$ which writes the initial value such that $ts(W_0) = op\_ts(W_0) = (0, 0)$. Next, we arrange each operation $op$ in $complete(H')$ in order of $(ts(op), op\_ts(op))$ to accomplish constructing a sequence $\tau$. From Lemma 6, for any operation executions $op_1$ and $op_2$, $op_1$ precedes $op_2$ in $\tau$ if $op_1 \overset{complete(H')}{\longrightarrow} op_2$. For each read operation $R$, there is a write operation $W$ such that $W = Write(R)$ and $ts(R) = ts(W)$ hold. For such $R$ and $W$, $op\_ts(W) < op\_ts(R)$ holds. Furthermore, for any write operation $W_1$ assigned a greater timestamp, $ts(W_1) > ts(W) = ts(R)$ holds. For any write operation $W_2$ assigned a smaller timestamp, $ts(W_2) < ts(W)$ holds. Therefore, there are no write operations between $W$ and $R$, and $\tau$ is legal. □

### 3.2.2 $u$-Synchronous Clocks

Here we describe a brief outline of an implementation on a $u$-synchronous clock and unreliable broadcast model, called $register_{UB-uC}$. The implementation provides a write operation with response time $u$ and a read operation with response time $d$. In $register_{UB-uC}$, a process uses its local clock value as a timestamp instead of *count* in $register_{UB-AC}$. When a write operation is invoked, a process sends a message containing the written value and its timestamp to all processes. In a read operation, a process decides the returned value and sends a message containing the value and its timestamp to all processes in $d - u$ since the invocation. The read operation is completed by returning the value. Response times of both write and read operations in $register_{UB-AC}$ is $d$, while response time of a write operation in $register_{UB-uC}$ can be reduced to $u$. This is because, for any write operation executions $W_1$ and $W_2$, $W_2$ is assigned a greater timestamp than $W_1$ if $W_2$ is invoked after $u$ since an invocation of $W_1$.

**Theorem 8:** The implementation $register_{UB-uC}$ is a wait-free linearizable implementation of a read/write register which achieves $res\_time\ (write) = u$ and $res\_time\ (read) = d$ on a $u$-synchronous clock and unreliable broadcast model. □

## 4. General Objects Using Reliable Broadcasts

In this section, we present two implementations of a general deterministic object using reliable broadcasts. One is on an asynchronous clock model, and the other is on a $u$-synchronous clock model. In our implementations of a general object, each process keeps a local copy of the implemented object, and applies invoked operations to it sequentially in some common order to all processes. Note that an implementation of a general objects reflects every supported operation.

### 4.1 Asynchronous Clocks

We previously presented a linearizable implementation of a general object on an asynchronous clock model, where we achieved $res\_time(op_a) = u$ and $res\_time(op_v) = 2d$[6]. Here we call that implementation $general_{IMT}$. Implementation $general_{IMT}$ did not assume any process fault, and actually some process fault causes violation of linearizability. The implementation is not wait-free in a sense that it does not tolerate a crash fault of a process. In this subsection, we slightly modify $general_{IMT}$ so as to guarantee wait-freedom in the case where a reliable broadcast is available. First, we explain $general_{IMT}$, and then mention the modification to produce a wait-free linearizable implementation, which we call $general_{RB-AC}$.

In implementation $general_{IMT}$, any val-type operation needs $2d$ since its invocation to obtain its response value, and any operation needs $u$ since its invocation to return a response that guarantees linearizability. As described before, each process applies invoked operations to the implemented object sequentially in some common total order to all processes. To decide the common total order, each process first decides a common partial order $\mathcal{PO}$ on occurrences of invoked operations, and then locally extends it to a total order $\mathcal{TO}$ by common rules to all processes. Here we just explain how to decide $\mathcal{PO}$ and $\mathcal{TO}$.

The partial order $\mathcal{PO}$ is decided as follows. Every process uses two kinds of messages, an *update* message and a *report* message. When an operation $op_1$ is invoked at a process $p_i$, the process $p_i$ broadcasts an update message to inform the other processes of the invocation. Let $t_1$ be the invoked time of $op_1$. If $p_i$ receives an update message of an operation $op_2$ from $p_j$ at $t$, the message was sent by $p_j$ at $t_2$ in the interval $[t-d, t-(d-u)]$. If $t < t_1 + d - u$ holds, it implies $t_2 < t_1$. In this case, $p_i$ considers $(op_2, op_1) \in \mathcal{PO}$. At $t_1 + d - u$, $p_i$ checks all occurrences of operations $op$ such that $(op, op_1) \in \mathcal{PO}$ (for reflexivility, consider $(op_1, op_1) \in \mathcal{PO}$). Then, $p_i$ informs all processes of this relation by broadcasting a report message which contains a set of occurrences of operations whose update message were received by $p_i$ before $t_1 + d - u$. When $p_i$ receives a report message, it augments $\mathcal{PO}$ with the relation informed by the report message and then taking its transitive closure.

The process $p_i$ returns a response for an operation $op_1$ as follows. If $op_1$ is ack-type, $p_i$ returns the response for $op_1$ after $u$ since the invocation. If $op_1$ is val-type, after $2d$ since an invocation of $op_1$, $p_i$ sets a total order $\mathcal{TO}$ on occurrences of operations whose report messages have been received by $p_i$. This total order $\mathcal{TO}$ is extended from $\mathcal{PO}$ by ordering unordered pairs by process identifiers. Then, $p_i$ applies the operations to its local copy in this order up to $op_1$. At that time, $p_i$ knows a response value for $op_1$, and returns a response for $op_1$.

Now we modify this implementation for wait-freedom. Only the problem is the case where some process crashes soon after some ack-type operation $op_1$ completed in the process. Once an ack-type operation is invoked at some process, the operation completes after $u$ since the invocation and the process broadcasts the corresponding report message after $d-u$ from the invocation. If $u < d - u$ holds and the process crashes after $op_1$ completes but before it broadcasts the message, any other process is not informed of the precedence relation about this operation. In this case, when some process $p_i$ attempts to obtain a response value of its operation, $p_i$ may apply $op_1$ to its local copy prior to some operation $op_2$ that is really prior to $op_1$ in a history. Such violation can be avoided as follows. If $op_2$ precedes $op_1$ in a history as in Fig. 6, every process including $p_i$ receives an update message of $op_2$ prior to an update message of $op_1$. In the modified implementation, processes broadcast such receipt orders in their report messages. When a process applies operations to its local copy, if some report message brings that an update message of $op_2$ is received before one of $op_1$ and no report message brings the reverse, the process applies $op_2$ prior to $op_1$. This modification can achieve wait-freedom without additional response time.

**Theorem 9:** The implementation $general_{RB-AC}$ is a wait-free linearizable implementation of any deterministic object which achieves $res\_time(op_a) = u$ and $res\_time(op_v) = 2d$ on an asynchronous clock model using reliable broadcasts, where $op_a$ is any ack-type operation and $op_v$ is any val-type operation. □

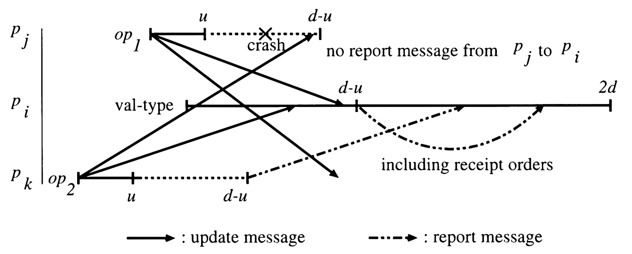


**Fig. 6** Case where a process crashes before broadcasting a report message.

```
variables
$op\_ts$, type timestamp ;
$local\_copy$, type value of theobject, init initial value of the
    object;
$update\_buffer$, init empty;

transition functions of process $p_i$
$Invoke(p_i, op)$ :
    $ReadClock(p_i, local\_cl)$;
    $BroadCast(p_i, \text{update}(op, (local\_cl, i)))$; /* update message */
    if $op$ is ack-type then $TimerSet(p_i, u, \text{ack})$;
    else /* $op$ is val-type */
        $op\_ts := (local\_cl, i)$; $TimerSet(p_i, d+u, \text{val})$;

$Receive(p_i, p_j, \text{update}(v, (recvd\_cl, recvd\_uid)))$ :
    $update\_buffer := update\_buffer \cup (op, (recvd\_cl, recvd\_uid))$

$Alarm(p_i, \text{ack})$ :
    $Response(p_i, res)$ where $res$ is a unique response value for
        current $op$;

$Alarm(p_i, \text{val})$ :
    while $op\_ts \geq \min\{ts | (op, ts) \in update\_buffer\}$ do
        $smallest := (op, ts)$ where $ts$ is the smallest in $update\_buffer$;
        apply $smallest$ to $local\_copy$;
        $update\_buffer := update\_buffer - \{smallest\}$;
    $Response(p_i, res)$;

$Stop(p_i)$ :
    No events can happen after this event.
```

**Fig. 7** Implementation $general_{RB-uC}$. (The code for $p_i$.)

## 4.2 $u$-Synchronous Clocks

Next, we propose an implementation of a general object on $u$-synchronous clock model, which we call $general_{RB-uC}$. It achieves $res\_time(op_a) = u$ and $res\_time(op_v) = d + u$. The program code for $p_i$ is given in Fig. 7.

In this implementation, the common order to all processes is decided only by a timestamp assigned to each operation. When an operation *op* is invoked at $p_i$, $p_i$ assigns the value of its local clock as a timestamp to *op*, and broadcasts an update message with the timestamp. When a process receives an update message, it stores the information in its $update\_buffer$. Since the difference between any pair of local clock values is at most $u$ and message delays are at most $d$, the process does not receive an update message with smaller timestamp than an operation *op* after $d + u$ since the invocation of the operation *op*. Therefore, if *op* is val-type, $p_i$ can decide the total order of operations with smaller timestamp at that time and obtain its response value. And then, it returns the response. For an ack-type operation, the process need not obtain its returned value but need $u$ for linearizability. If a process crashes while an operation, the operation is left pending. In such a case, all correct processes receive the update message, or no processes receive it because of a reliable broadcast. Therefore, the implementation $general_{RB-uC}$ works correctly in the case where a process gets faulty.

**Theorem 10:** The implementation $general_{RB-uC}$ is a wait-free linearizable implementation of any deterministic object which achieves $res\_time(op_a) = u$ and $res\_time(op_v) = d + u$ on a $u$-synchronous clocks model using reliable broadcasts, where $op_a$ is any ack-type operation and $op_v$ is any val-type operation. □

## 5. Conclusions

In this paper, we have presented wait-free linearizable implementations shown in Table 2, which are four implementations of read/write registers and two implementations of general objects. In general, an implementation on an asynchronous clock model needs longer worst-case response times than an implementation on a $u$-synchronous clock model if the other conditions are the same. In an asynchronous clock model, if processes in the system execute a synchronization procedure (e.g. procedure `Synch`[7] for a reliable broadcast model) to make the difference between any pair of local clock values at most $u$, we can apply an implementation for a $u$-synchronous clock model. Taking costs of the synchronization procedure into consideration, implementations for a $u$-asynchronous clock model is more effective in the case where operations are invoked many times in an asynchronous clock model.

Some open problems are left. Some lower bound results as to worst-case response times in linearizable implementations were presented [4]–[6] . There are gaps between their results and our results. The other open problem is about linearizable implementations of general objects on an unreliable broadcast model. We can easily construct a wait-free linearizable implementation which provides operations with response time proportional to the number of processes. However, we do not know whether there exists a wait-free linearizable implementation which provides operations with shorter response time.

## Acknowledgement

We thank the members of Fujiwara Lab. for their useful suggestions. This work is supported in part by the Scientific Research Grant-in-Aid from Ministry of Education, Science and Culture of Japan ((B-2)10205218).

### References

[1] M. Herlihy and J. Wing, "Linearizability: A correctness condition for concurrent objects," ACM Trans. Programming Languages and Systems, vol.12, no.3, pp.463–492, 1990.

[2] M. Herlihy, "Wait-free synchronization," ACM Trans. Programming Languages and Systems, vol.13, no.1, pp.124–149, 1991.

[3] J. James and A. K. Singh, "Fault tolerance bounds for memory consistency," Proc. 11th Int. Workshop on Distributed Algorithms (LNCS1320), pp.200–214, 1997.

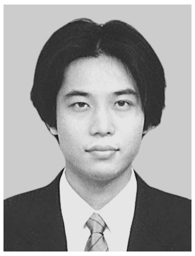
[4] M. Mavronicolas and D. Roth, "Efficient, strongly consistent implementations of shared memory," Proc. 6th Int. Workshop on Distributed Algorithms(LNCS647), pp.346–361, 1992.

[5] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," ACM Trans. Computer Systems, vol.12, no.2, pp.91–122, May 1994.

[6] 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.

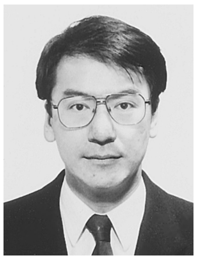
[7] M. Mavronicolas and D. Roth, "Sequential consistency and lineariability:read/write objects," Proc. 29th Annual Allerton Conf. on Communication, Control and Computing, pp.683–692, Oct. 1991.

**Sen Moriya** received the B.S. degree from Osaka University in 1995, and the M.E. and Ph.D. degrees from Graduate School of Information Science, Nara Institute of Science and Technology (NAIST) in 1997 and 2000. Since 2000, he has engaged in NTT Communication Science Laboratories. His research interests include distributed algorithms.



**Katsuro Suda** received the B.E. degree from Shinshu University in 1997, and the M.E. degree from Graduate School of Information Science, Nara Institute of Science and Technology (NAIST) in 1999. Since 1999, he has engaged in NTT Software Corporation.

**Michiko Inoue** received her B.E., M.E. and Ph.D. degrees from Osaka University in 1987, 1989, and 1995 respectively. She is an instructor of Graduate School of Information Science, Nara Institute of Science and Technology (NAIST). Her research interests include distributed algorithms, parallel algorithms, graph theory and design and test of digital systems. She is a member of IEEE, IPSJ, and JSAI.



**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 Education Center for Information Processing, Osaka University between 1987–1990, and had worked at Faculty of Engineering Science, Osaka University between 1990–1994. He is now an associate professor of Graduate School of Information Science, Nara Institute of Science and Technology (NAIST). 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.

**Hideo Fujiwara** received the B.E., M.E. and Ph.D. degrees in electronic engineering from Osaka University, Osaka, Japan, in 1969, 1971, and 1974, respectively. He was with Osaka University from 1974 to 1985 and Meiji University from 1985 to 1993, and joined Nara Institute of Science and Technology in 1993. In 1981 he was a Visiting Research Assistant Professor at the University of Waterloo, and in 1984 he was a Visiting Associate Professor at McGill University, Canada. Presently he is a Professor at the Graduate School of Information Science, Nara Institute of Science and Technology, Nara, Japan. His research interests are logic design, digital systems design and test, VLSI CAD and fault tolerant computing, including high-level/logic synthesis for testability, test synthesis, design for testability, built-in self-test, test pattern generation, parallel processing, and computational complexity. He is the author of Logic Testing and Design for Testability (MIT Press, 1985). He received the IECE Young Engineer Award in 1977, IEEE Computer Society Certificate of Appreciation Award in 1991, Okawa Prize for Publication in 1994, and IEEE Computer Society Meritorious Service Award in 1996. He is an advisory member of IEICE Trans. on Information and Systems and an editor of IEEE Trans. Computers, J. Electronic Testing, J. Circuits, Systems, J. VLSI Design and others. Dr. Fujiwara is a fellow of the IEEE as well as a member of the Information Processing Society of Japan.