アルゴリズム 32－3
（1993. 3. 18）

分散システムにおけるＦＩＦＯキューの線形化可能な実現

井上美智子　増澤利光　都倉信樹
大阪大学基礎工学部情報工学科

線形化可能性（linearizability）とは，分散環境で実現した共有オブジェクトの正当性条件の一つである．本稿では，分散環境におけるＦＩＦＯキューの線形化可能な実現のコストに関するいくつかの上界，下界を示す．コストは，ＦＩＦＯキューに対する操作エンキュー（enqueue），デキュー（dequeue）それぞれの最悪時の応答時間 E_{res}，D_{res} で評価する．本稿では，プロセッサ間のメッセージ伝送遅延がすべて $[d-u, d]$（d,u は $0 \le u \le d$ を満たす定数）の範囲である場合，(1)$E_{res} = u$，$D_{res} = 2d$ である線形化可能な実現を示す．また，(2)$u/2 \le E_{res} < u$ のとき，$D_{res} + 2E_{res} \ge 2d$ が成り立つ，(3) プロセッサ数が $2m-1$ 以上のとき，$E_{res} \ge u\frac{m-1}{m}$ が成り立つことを示す．

Efficient Linearizable Implementation of FIFO Queues

Michiko Inoue　Toshimitsu Masuzawa　Nobuki Tokura
Faculty of Engineering Science,Osaka University

The cost of implementing FIFO queues in a distributed system is studied under a consistency condition linearizability. The cost is measured by the worst-case response times: E_{res} for enqueue operation and D_{res} for dequeue operation. We show the following results on the assumption that all message delays are in the range $[d-u, d]$ for some constants d and u $(0 \le u \le d)$. (1) There exists a linearizable implementation of FIFO queues with E_{res} and D_{res} are u and $2d$ respectively. (2) $D_{res} + 2E_{res} \ge 2d$ holds in the case where $u/2 \le E_{res} < u$. (3) $E_{res} \ge u\frac{m-1}{m}$ holds in the case where the number of processes is more than or equal to $2m-1$.

1 Introduction

How to provide a logically shared memory model in a distributed system is a fundamental problem in concurrent computing. The shared memory model must allow user process to have access to memory concurrently, that is, each access takes some duration and different processes can have accesses to memory with overlapping their durations. Overlapping of memory accesses introduces the problems of correctness of the system. It becomes more complicated in the case where implementations employ multiple copies of a single memory object to enhance performance. A consistency mechanism guarantees some consistency condition for the system behaivor. *Sequential consistency*([1]) and *linearizability*([2]) are proposed as consistency conditions. Sequential consistency guarantees that the result of any execution is same as that of some sequential execution. When this sequential order preserves the global ordering of non-overlapping operations, this consistency condition is called linearizability.

We consider linearizability which is stronger condition than sequential consistency. Unlike sequential consistency, linealizability is a local property: a system is linearizable if each individual object is linearizable. Locality allows concurrent systems to be designed and constructed in a modular fashion; linearizable objects can be implemented, verified, and executed independently. Linearizability is also a non-blocking property: a pending call of a totally-defined operation is never required to wait for another pending call to complete. Non-blocking implies that linearizability is an appropriate condition for system for which real-time response is important.

Several author have investigated a read/write object as a shared memory object([3],[4],[5],[6]). But read/write objects are weaker type of object in the sense that many types of object cannot be implemented using read/write objects([7]). Many multiprocessor systems now support more powerful objects, e.g., FIFO queues, stacks, the objects that support test-and-set, or fetch-and -add([8]). Attiya([5]) investigated FIFO queues and stacks on the assumption that all message delays are in the range $[d-u, d]$ for some constants d and u, $0 \leq u \leq d$. . The cost is measured by the worst-case response times: E_{res} for enqueue operation and D_{res} for dequeue operation. In [5], it is shown for linearizable implementations that D_{res} is at least d, and that E_{res} is at least $u/2$ in the case where the number of processes is more than or equal to 3. Attiya also presents a sequentially consistent implementaion with E_{res} and D_{res} are at most 0 and $2d$, respectively.

In this paper, we show the following results for linearizable implementations: (1) There exists a linearizable implementation of FIFO queues with E_{res} and D_{res} are u and $2d$ respectively. (2) $D_{res} + 2E_{res} \geq 2d$ in the case where $u/2 \leq E_{res} < u$. (3) $E_{res} \geq u\frac{m-1}{m}$ in the case where the number of processes is more than or equal to $2m-1$. The last result generalized Attiya's result([5]).

In the implementation presented here, each process has a local copy of every FIFO queue. And the process stopping excecution can be distinguished, since all message delays are in some range. Therefore, it seems that the implementation is easy to improved to have fault-tolerancy.

2 Model

A *distributed system* consists of *processors*. For specifying system behavior, a (system-wide) global clock is used. Remark that the global clock is introduced only for specifying system behavior and no processor in the system can have access to the global clock. Each processor has a local clock that runs at the same rate as a global clock, and on each processor multiple processes, each of which executes some program, run concurrently. That is, a distributed system $D = (\{P_1, P_2, \cdots, P_n\}, GCK)$, where P_i is a processor and GCK is a global clock. Each processor $P_i = (\{p_{i_1}, p_{i_2}, \cdots, p_{i_m}\}, LCK_i)$, where p_{i_j} is a process and LCK_i is a local clock of P_i. Times defined by GCK and LCK_i are called a global time and a local time of P_i, respectively. For any local time defined by any LCK_i, if a global time is T at a local time t, a global time is $T+1$ at a local time $t+1$.

We consider the following three processes for each processor P(Fig1): (1)an *application process* a_P that executes some application program, (2)a *timer* process tm_P that informs other process in P of current local time T if the timer is set at the past clock time $T-t$ $(t \geq 0)$, and (3)a *memory consistent system*(mcs) process mcs_P (mentioned later).

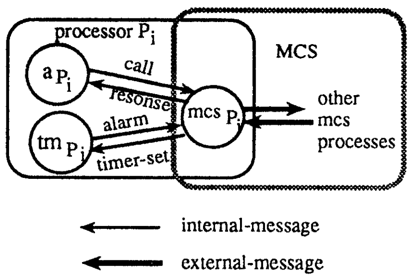


Fig1. memory consitent System.

The process communicates with other processes by only exchanging messages. A message M is exchanged by using two events, *send-event* and *receive-event*. An event is identified by the process at which the event ocuurs, the type of the event (send or receive), and the associated message. A sender process of M generates a *send-event* of M to send M, and *receive−event* of M occurs at receiver-process when M arrives. There are the following two kinds of messages: (1)*internal-messages* exchanged between the processes on the same processor, and (2)*external-messages* are exchanged between the processes on the differnt processors. A *delay* of a message is the time elapsed between sending and the corresponding receipt, that is, if a message M is sent at global time T and is received at global time T', the delay of M is $T' - T$. We assume that delays of internal-messages are all 0, and the delays of external-messeges are in the range $[d-u, d]$ for some known constants u and d, such that $0 \le u \le d$.

Consider *shared objects*, or just *object*, to which multiple application processes on different prosessors can have accesses concurrntly. The object is defined by a unique *name* and a set of *operations* that provide the means to manipulate the object. A set of objects OBJ is provided by a MCS (memory consistent system) that consists of mcs processes(Fig1). An application process a_P manipulates an object O in OBJ as follows: (1)a_P calls some operation op of O by sending some internal-message to mcs_P. (2)mcs_P responds the call by sending some internal-message to a_P. An operation is defined by an *operation-name*, a sequence of *arguments*, a *response-name* and a *return-type*. An application process a_P calls an operation by sending an internal-message $op(Q,args)$, where Q is an object, op is an operation-name, and $args$ is a sequence of arguments. A mcs process mcs_P responds for a call of an operation op by sending an internal-message $res(Q,v)$, where Q is an object, res is a response-name of op, and v is a value of return-type of op. We say MCS *implements* a set of objects OBJ, or the MCS is an *implementation* of OBJ, if MCS consists of $mcs_{P_1}, mcs_{P_2}, \cdots, mcs_{P_n}$, such that, mcs_{P_i} responds for a call of any operation of any object in OBJ from a_P.

Each mcs process mcs_P uses some kinds of external-messages and the following four kinds of internal-messages: (1)*call-messages* sent by a_P to call an operation, (2)*response-messages* sent by mcs_P to respond to a_P, (3)*timer-set-messages* sent by mcs_P to tm_P to set a timer for the future local time, and (4)*alarm-messages* sent by tm_P to inform mcs_P of current local time for which a timer is set. Associated with the kinds of messages, a set of event occured at the mcs process is classified the following six kinds: (1)*calls* which are receive-events of call-messages, (2)*responses* which are send-events of response-messages, (3)*timer-sets* which are send-events of timer-set-messages, (4)*alarms* which are receive-events of alarm-messages, (5)*external-sends* which are send-events of external-messages, and (6)*external-receives* which are receive-events of external-messages.

The mcs process is modeled as a finite state machine (Q, I, O, s). where,

- Q is a set of finite states, including an initial state q_0.
- I is a set of receive-events occured at the mcs process.
- O is a set of send-events generated by the mcs process.
- s is a transition function given by a set of 4-tuples (s, ie, soe, s') where s and s' are states, ie is a receive-event, and soe is a (possibly empty) sequence of send-events. Such a 4-tuple is called a *step*.

A step (s, ie, soe, s') means that when a receive-event ie is occured at the process in state s, the process generates send-events in soe and becomes state s'.

For the mcs process p, a *process history* h_p is a finite sequence of pairs of a step and a local time (st_0, t_0), (st_1, t_1), $\cdots$, (st_n, t_n), where st_i is a step and t_i is a local time at which step st_i is done. Note that any prefix of a process history is a process history. Letting $st_i = (s_i, ie_i, soe_i, s'_i)$, s_0 is an initial state and $s'_i = s_{i+1}$. A response *matches* a call if processes at which they occure agree and a response-name of the response and an operation-name of a call are of same operation. A call is *pending* if no matching response follows the call in process history. The process history h_p of mcs process p is *well-formed*, if it satisfies the following two conditions: (1)In any prefix of the process history, at most one call is pending. (2)An alarm is occured at local time t iff the corresponding timer-set is generated before t, that is, $((s_0,ie,soe_0,s'_0),t)$ appears in h_p such that ie is an alarm iff $((s_1,ie_1,soe_1,s'_1),t')$ appears in h_p such that soe_1 includes the timer-set for t and $t' \le t$.

A *system history*, or just *history*, H consists of two sets. One is a set of *difference times* $\delta_p(H)$ of all process p in the system. The difference time $\delta_p(H)$ is p's local time minus a global time. Another is a set of well-formed process histories of all processes in the system. The process histories satisfy the following condition: An external-message M from q is received at global time T in p's process history iff the corresponding send-event occures before global time T in q's process history. A history is *admissible* if the delay of every external-message is in the range $[d-u, d]$.

In this paper, we consider FIFO queues over some domain V as objects. There are two kinds of operations of FIFO queue Q: (1)$Enq(v)/Ack()$ where Enq, v, Ack are an operation-name, an argument, a response-name, respectively, and $v \in V$. It has no return value. Enq means to insert v to Q. (2)$Deq()/Ret(V')$ where Deq, Ret, V' are an operation-name, a response-name, a return-type, respectively, and $V' = V \cup \{\perp\}$ where $\perp$ is a special value and $\perp \notin V$. It has no argument. Deq means to return the value inserted to Q first, and remove the value from Q. A special value '$\perp$' is returned if Q is empty. Denote by $Enq_p(Q,v)$, $Ack_p(Q)$, $Deq_p(Q)$, and $Ret_p(Q,v)$, a call and a response of Enq of Q and a call and a response of Deq of Q, respectively, such that they occures at the mcs process p.

An *object history* is a finite sequence of events consisting of calls and responses. For the object Q, a *one-object history* h_Q is an object history consisting of events of Q. A one-object history h_Q is *sequential* if:

(1)It is an alternating sequence of calls and responses.

(2)Its first event is a call, and its last event is a response.

(2)Each call is immediately followed by a matching response.

Note that the length of any sequential history is even.

For any sequential one-object history h_Q, a *queue-state* of h_Q is a list of values in V. Denote by S_τ a queue-state for a sequential one-object history τ. We use the following three functions for a queue-state q and a value v: (1)ins(q,v) returns a queue-state made by inserting v to the end of q. (2)rest(q) returns a queue-state made by removing the first element of q if q is not empty, othewise returns q. (3)first(q) returns the value of the first element of q if q is not empty, otherwise returns '$\perp$'. A *sequential specification* defines a behaivor of an object, that is, it defines a set of possible sequential one-object histories. The queue-state and the sequential specification are defined recurcively as follows. For an empty one-object history τ_0, S_{τ_0} is empty and τ_0 is in the sequential specification. Let τ_{2i} be a sequential one-object history of length $2i$ in the sequential specification. For the one-object history $\tau_{2i+2}=\tau_{2i}\circ<Enq_p(Q,v),Ack_p(Q)>$, $S_{\tau_{2i+2}}=ins(S_{\tau_{2i}},v)$ and τ_{2i+2} is in the sequential specification, where $\circ$ is a concatenation operator on sequences and $<\cdots>$ represents a sequence of events. For the one-object history $\tau_{2i+2}=\tau_{2i}\circ<Deq_p(Q),Ret_p(Q,v)>$, $S_{\tau_{2i+2}}= rest(S_{\tau_{2i}})$ and τ_{2i+2} is in the sequential specification iff $v=first(S_{\tau_{2i}})$.

For a set of processes and a set of objects, the object history τ is *legal*, if, for each object Q, the subsequence of τ consisting of events of Q is in the Q's sequential specification.

For object history τ and process p, $\tau|p$ is the subsequence of τ consisting events occured at p. For history H and process p, $ops_p(H)$ is the subsequence of H consisting of only call and response events occured at p.

Definition 1 *An admissible history H is linearizable if there exist some admissible history H' and some legal object history τ, such that H is extended to H' by adding some (possibly zero) response events, $ops_p(H') = \tau|p$ for each process p, and if the response of operation op_1 occured before the call of operation op_2 in H', then the response of op_1 precedes the call of op_2 in τ.*

An mcs is a linearizable implementaion of a set of objects if any admissible history of the mcs is linearizable.

A response time of operation op is the time elapsed between the call and the corresponding response event of op. For convenience, the response time of op is 0 if the call op is pending. The efficiency of an implementation can be measured by the worst-case response times for operations on the object. Given a particular MCS and a FIFO queue Q implemented by it, we denote by $E_{res}(Q)$ the maximum response time of Enq operations on Q and by $D_{res}(Q)$ the maximum response time of Deq operations on Q, over all admissible history. Denote by E_{res} the maximum of $E_{res}(Q)$, and by D_{res} the maximum of $D_{res}(Q)$, over all objects Q implemented by the mcs.

3 Upper Bounds

In this chapter, we show that there exists a linearizable implementation of FIFO queues with $E_{res} = u$ and $D_{res} = 2d$. Linearizability is a local property, that is a system is linearizable if and only if each individual object is linearizable. Therefore, it suffices to present a linearizable implementation of one FIFO queue Q with $E_{res} = u$ and $D_{res} = 2d$.

In this implementation, each process p keeps a local copy of Q and updates (either enqueues or dequeues) the copy in the common order to all processes. For simplicity, we assume FIFO channels, that is , messages sent along the same channel are deliverd in the FIFO order. Note that FIFO channels are easily implemented by using serial numbers.

To explain the idea of this implementation, consider the case where the response times for all operations are u. Let H be a linearizable and admissible history. And, let op_1 be any operation called by any process p_1 at global time t_1, and op_2 be an any operation called by any process p_2 at global time t_2 such that $t_2 > t_1+u$. There exist some history H' to which H is extended, and some legal object history τ, such that, $ops_p(H') = \tau|p$ for each process p, and if the response of operation op occured earlier than the call of operation op' in H', then the response of op precedes the call of op' in τ. In τ, the response of op_1 precedes the call of op_2. If every process sends some kind of messages to all processes at calling, such a message of op_1 is received at p_2 ealier than $t_2 + d - u$ $(> t_1 + d)$, and such a message of op_2 is received at p_1 later than $t_1 + d - u$ $(< t_2 + d - u)$. Each process, called an operation at t, checks the messeges that were received ealier than $t + d - u$, and regard corresponding operations as the preceding operations to its own operation. Because operations regarded as preceding are called at least ealier than $d-u$ after its receipt $(< t)$, this ordering includes no cycle. By gathering such ordering information for every operation, the partial order on operations is obtained. In our implementaion, every process handles local copies of FIFO queues in some total order that includes this partial order. This total order suffices for linearizability in the case where every response time is at least u.

To decide this total order, each process p uses two directed acyclic graphs, a partial order graph POG_p, and a total order graph TOG_p. Each respects a partial order on call events. That is, its node set consists of call events and each order graph G induces a partial order $\prec_G$ on call events: $e_1 \prec_G e_2$ if e_1 is reachable from e_2 on G. Events unrelated by $\prec_G$ are said to be *concurrent*. Particularly, $G_{p,t}$ denotes a partial graph G_p at global time t. Each process p also uses a serial number for its call event.

A graph $POG_{p,t}$ presents a partial order that is regarded by p at global time t. Initially, POG_p has no events, that is, POG_p is an empty graph. When a call event e_1 occurs at q, q sends an *update* message to all processes (including q) with the serial number of the call event. Let t be a global time at which e_1 occures. At $t+d-u$, q checks *update* messages that were arrived earlier than $t+d-u$, and sends *report* message to all processes in order to inform them that the events contained in this message precede e_1 (it suffices to inform the maximum serial numbers received from each process). We call this message the *report* message of e_1. When p receives a *report* message of e_1 from q, p extends POG_p by (1)adding the events contained in the message, if necessary, and (2)adding the edges from e_1 to the events contained in this message and the edge from e_1 to q's immediately privious event. Let m be a *report* message of an event e. Denote by $C(m)$, $E(m)$ and $rm(e)$ a set of events contained in m, a set of events $C(m) \cup \{e\}$, and a *report* message of e, and denote by $R_p(t)$ a set of message arrived at p at global time $t'(\leq t)$, A graph $POG_{p,t}$ consists of the following two compornents: (1) a set of nodes $\{e|e \in E(m)$, and $m \in R_p(t)\}$ and (2) a set of directed edge $\{(e,e')|\ e' \in C(rm(e))$, and $rm(e) \in R_p(t)\}$.

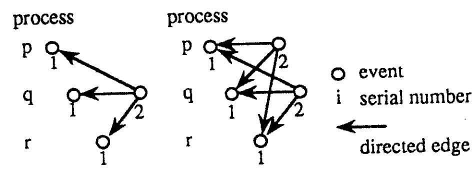
Let t' be a global time at $2d$ after the call of p's own event e_p. At t', $TOG_{p,t'}$ is created from $POG_{p,t'}$. Its node set consists of the events whose *report* messages have arrived at p, and, for any two events in this set, if the edge between them exists on $POG_{p,t'}$, $TOG_{p,t'}$ has same egde. And, any two concurrent events on $\prec_{POG_{p,t'}}$ are orderd using process id[1]. Denote by $Id(e)$ an id of process which

[1] every process has process a unique id, and there exists some total order on ids.

calls a event e. A graph $TOG_{p,t'}$ consists of the following two compornents: (1) a set of nodes $N = \{e | rm(e) \in R_p(t)\}$ and (2) a set of directed edge $\{(e, e') | \ e, e' \in N$, and $e' \in C(rm(e))\} \cup \{(e, e') | \ e, e' \in N, Id(e) < Id(e')$, and e, e' are concurrent on $\prec_{POG_{p,t'}}$ $\}$. Then, p applies associated operations that have not applied yet to copy of Q in the order of $TOG_{p,t'}$ until the operation associated with e_p is applied.

In the following two examples, denote every call event as an orderd pair (process id, serial number).

Example 1 Figure2(a) shows POG_p consists of $(p, 1)$, $(q, 1)$, $(q, 2)$, and $(r, 1)$. When p receives *report*(2, {1,1,1}) from p, event $(p, 2)$ and edges from $(p, 2)$ to $(p, 1)$, $(q, 1)$, and $(r, 1)$ are added(Fig2(b)). □



(a) before report message of (p,2) is received.

(b) after report message of (p,2) is received.

Fig2. Update of POG.

The details of implementaion are shown in Fig.3. A response for *Enq* is generated at u after its call. A response for *Deq* is generated when the *Deq* is applied to a copy of a FIFO queue.

To show that the implementation is correct, fix any admissible history H, and let H' be some admissibe history, to which H is extended, such that the responses for all pending calls are returned and no new call occurs. Such history exists because the response for every call is returned at most $2d$ after its call.

Lemma 1 *In H', let t be a global time at $2d$ after a call of any event e. For every process p and $\forall t' > t$, $TOG_{p,t'}$ induces a same total order as $TOG_{p,t}$ on a set of events that precede e on $\prec_{TOG_{p,t}}$. And this order is common to all processes.*

proof: By implementaion, POG of all processes are created from common information. Since H' consists of the process histories each of which is finite, after sufficiently long time, for every process p, $\prec_{POG_p}$ and $\prec_{TOG_p}$ is common to all processes. So, it suffices to show that, for every p, $\prec_{TOG_p}$ on a set of events that precede e on $\prec_{TOG_{p,t}}$ does not change after t.

Suppose opposite. Let e_1, e_2, ..., and $e_n = e$ be the events such that $e_i \prec_{TOG_{p,t}} e_{i+1}$. At some time $t' > t$, at least one of three cases occures: (1) for some $e_i (0 \leq i < n)$, $e \prec_{TOG_{p,t'}} e_i$, (2) for some e_0(not e_i), $e_0 \prec_{TOG_{p,t'}} e$, (3) $e_i \prec_{TOG_{p,t'}} e_j$ $(1 \leq j < i < n)$.

(1) for some $e_i (0 \leq i < n)$, $e \prec_{TOG_{p,t'}} e_i$.

The data types:

update : record with fields

- sn : integer;
- op : name of operation;
- obj : name of an object;
- val : value;
- id : process id;
- time : time;

The components of state of process p

- *val* : value; *sn* : integer, initially 0(serial number);
- *update-buffer* : set of update, initially empty;
- *receipt*[] : array of integer (array of serial number);
- POG_p : partial order graph;
- TOG_p : total order graph;
- copy of every objest Q, initially empty queue;

The transition function of process p:

$Enq_p(Q, v)$

- send *update*(sn, "*Enq*",Q, v) to all processes
- $val := v$
- $sn := sn + 1$
- generate *timer-set*(u, "generate-Ack")
- generate *timer-set*($d - u$, "check-update(sn)")

$Deq_p(Q)$

- send *update*(sn, "*Deq*", Q, $\perp$) to all processes
- $sn := sn + 1$
- generate *timer-set*($d - u$, "check-update(sn)")
- generate *timer-set*($2d$, "generate-Ret")

receive *update*(sn, op, Q, v) from q at local time t

- add $(sn, op, Q, \perp, q, t)$ to *update-buffer*

receive *report*(sn, *receipt*) from q

- update POG_p using the information of *report*

alarm("generate-Ack")

- generate $Ack_p(val)$

alarm("check-update(n)") at local time t

- for all process p do
  - *receipt*[p] := max(sn | (sn, op, Q, v, q, t') $\in$ *update-buffer*, $t' < t$)
  - send *report*(n, *receipt*) to all processes

alarm("generate-Ret")

- generate TOG_p from POG_p
- **repeat**
- take next unhandled $E = (sn, op, Q, v, q, t)$ in the order TOG_p
- handle E to local copy of Q
- if $E.op$ ="*Deq*" then get return value v
- if $E.op$ ="*Deq*" and $E.id = p$ then generate $Ret_p(sn, v)$
- until $E.op$ ="*Deq*" and $E.id = p$

Fig.3 An Implementation of FIFO queues.

Since e is not reachable from e_i on $POG_{p,t}$, *report* message of e_i that was received by p earlier than t informed that e's *update* message was not received earlier than $d-u$ after a call of e_i This implies that call of e_i occured earlier than u after a call of e, that is, it occured before $t-2d+u$. Since e is reachable from e_i on $POG_{p,t'}$, some e_0 exists such that its *report* message arrives at p after t, makes e_0 reachable from e_i, and makes e reachable from e_0. A call of e_0 occured before a call of e_i because e_0 is reachable from e_i on $POG_{p,t'}$, that is, it occured before $t-2d+u$. This implies that p received the *repoprt* message of e_0 before t. A contradiction.

(2) for some event e_0(not e_i),$e_0 \prec_{TOG_{p,t'}} e$.

(a)Case where p received e_0's *report* message later than t : Letting q be the process that called e_0, q called e_0 later than $t-2d+u$, and received e's *update* message not later than $t-d$. That is, q received the *update* message earlier than $d-u$ after e_0's call. Thus, *report* message of e_0 informed that the *update* message of e was received ealier than $d-u$ after its call. Once e_0 belongs to TOG_p, it always holds that $e \prec_{TOG_p} e_0$. A contradiction.

(b)Case where p received e_0's *report* message not later than t: e_0 is not reachable from e on $TOG_{p,t}$ and e_0 is reachable from e on $TOG_{p,t'}$. There exists some e' such that its *report* message arrives at p after t, makes e_0 reachable from e', and makes e' reachable from e on POG_p. Because e' is reachable from e on $POG_{p,t'}$, e''s call occured earlier than $t-2d+u$, and e''s *report* message arrives at p before t. A contradiction.

(3) $e_i \prec_{TOG_{p,t'}} e_j$ $(1 \le j < i < n)$.

On $POG_{p,t}$, e_i is not reachable from e_j while reachable on $POG_{p,t'}$. On the path from e_j to e_i on $POG_{p,t'}$, if some event e' ($\neq e_k(1 \le k < n)$) exists, $e' \prec_{TOG_{p,t'}} e_j$. But $e \prec_{TOG_{p,t'}} e'$ and $e_j \prec_{TOG_{p,t'}} e$ by (1) and (2), therefore $e_j \prec_{TOG_{p,t'}} e'$, a contradiction. Thus, all events on this path are e_k $(1 \le k < n)$. Since every e_k's *report* messages have arrived at p not later than t, no edge between e_ks' are added to POG_p after t. Therefore, e_i can not be reachable from e_j on POG_p after t. A contradiction. ■

By lemma1, after sufficiently long time, for every process p, TOG_p induces common total order to all processes. In the followings, we denote by $\prec$ this total order.

Lemma 2 *For any two call events e_1 and e_2 in H', if the response for e_1 occuerd before the call of e_2, then $e_1 \prec e_2$ holds.*

proof: Let t, t' and t'' be global times at which e_1's call, e_1's response and e_2's call occured. By implementation, $t' \ge t+u$, and e_1's *update* message arrived at the process that called e_2 not later than $t+d$. It follows from $t+d < t''+d-u$ that e_1's *update* message arrived earlier than $d-u$ after e_2's call. Each process that receives e_2's *report* message adds the edge from e_2 to e_1 to its POG. Therefore, $e_1 \prec e_2$ holds. ■

Theorem 1 *There exists a linearizabe implementation of FIFO queues with $E_{res} = u$ and $D_{res} = 2d$.*

We show the implementaion of Fig.3 is a linearizable implementation of FIFO queue Q with $E_{res} = u$ and $D_{res} = 2d$. It is clear that $E_{res} = u$ and $D_{res} = 2d$.

To show that the implementaion is linearizable, it suffices H is linearizable. Let τ be an object history in which all call events appear in the total order $\prec$ on call events in H'.

For a call e_i of any operation op_i of Q, all operations whose calls precede e_i in τ are applied to Q before applying op_i. Therefore, the response of op_i can be created so that sequence up to the response of e_i is in the sequential specification of FIFO queue. Therefore, τ is legal.

For every process p, $ops_p(H') = \tau|p$ holds, since p updates Q in the same order τ. By lemma2, for any two call events e_1 and e_2 in H', if the response of e_1 occuerd before the call of e_2, then $e_1 \prec e_2$ holds, that is, e_1 precedes e_2 in τ.

By the above, there exist some history H' to which H is extended and some legal object history τ, such that, $ops_p(H') = \tau|p$ for each process p, and if the response of operation op_1 occuerd before the call of operation op_2 in H', then the response of op_1 precedes the call of op_2 in τ. Any admissible history H of the implementaion is linearizable. ■

4 Lower Bounds

We show the following two lower bounds on the worst-case response time: (1) $D_{res} + 2E_{res} \ge 2d$ in the case where $u/2 \le E_{res} < u$. (2)$E_{res} \ge u\frac{m-1}{m}$ in the case where the number of processes is more than or equal to $2m-1$.

Theorem 2 *For the memory-consistency system that is a linearizable implementation of FIFO queues, $D_{res}+2E_{res} \ge 2d$ in the case where $u/2 \le E_{res} < u$.*

proof: Assume opposite. Let $E_{res} = T$ and $D_{res} = 2d - 2T - \epsilon_0$ for some $\epsilon_0 > 0$. There exist some ϵ_1, ϵ_2 and ϵ_3 such that $\epsilon_1 > 0$, $\epsilon_2 > 0$, $\epsilon_3 > 0$, $\epsilon_1 + \epsilon_2 < \epsilon_3 < \epsilon_0$ and $\epsilon_3 < u - T$. Let p and q be two processes that can execute Enq operations to some FIFO queue Q and r be a process that can execute Deq operations to Q. Consider following two histories. In both histories, mcs starts with empty queue Q at global time 0.

history1 There exists some admissible history such that $Enq_p(Q,1)$ and $Enq_q(Q,2)$ occur at global time 0, and $Deq_r(Q)$ occurs at global time $T+\epsilon_1$ (Fig4(a)). The difference times of both processes are 0. The message delays are d. Since, both responsses for $Enq_p(Q,1)$ and $Enq_q(Q,2)$ occur earlier than $T+\epsilon_1$, the response for $Deq_r(Q)$ has the return value 1 or 2. Without loss of generality, we can assume that $Ret_r(Q,2)$ occures.

history2 $Enq_p(Q,1)$ occurs at global time are 0, $Enq_q(Q,2)$ occurs at global time $T+\epsilon_2$, and $Deq_r(Q)$ occurs at global time $2T-u+\epsilon_3$ (Fig4(b)). The difference times are 0 for p, $T_q = -(T+\epsilon_2)$ for q, and $T_r = u - T + \epsilon_1 - \epsilon_3$ for r. That is, all call events

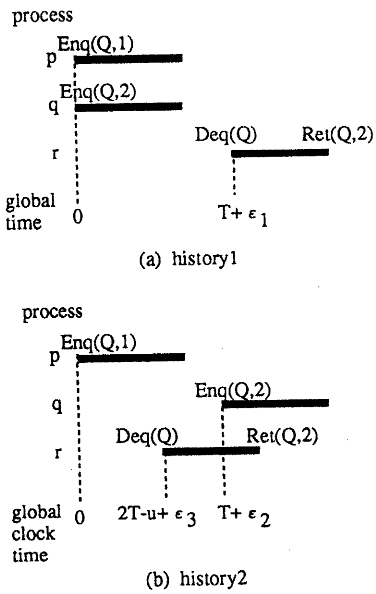


Fig4. history1 and history2 in proof of theorem2

occure at the same clock times at each process as history1. The message delays are $d-T_r$ from p to r, $d+T_q-T_r$ from q to r, and d for all other orderd pairs. That is, message-receive events for messages to r occur at the same local times at r as history1. Since $d-T_r$ and $d+T_q-T_r$ are both in the range $[d-u,d]$, history2 is admissible.

In history2, $Ack_p(Q)$ occurs earlier than $Enq_q(Q,2)$, and $Deq_r(Q)$ is the only dequeue operation for Q. The returned value of $Deq_r(Q)$ is either 1 or $\perp$.

In the both histories, p and q receive no message not later than global time d. The messages sent earlier than global time d from p or q are same as hitory1, and are received at the same local time to r. During receiving these messages, r has done same steps as history1. If r generates the response for $Deq_r(Q)$, its return value must be 2, that is a contaradiction. r must generate the response after this interval, that is after $d+d-T_r=2d-u+T-\epsilon_1+\epsilon_3>2d-u$ and after $d+d+T_q-T_r=2d-u+\epsilon_3-(\epsilon_1+\epsilon_2)>2d-u$. By assumption, r generates the response until $2T-u+\epsilon_3+D_{res}=2d-u+\epsilon_3-\epsilon_0<2d-u$, that is a contradiction. ■

We then show the second result for lower bounds. The case where the number of processes is more than or equal to 3 is proved in [5] . Our result generalizes this result.

We prove the second result using the technique of *shifting* [5] , that is used to change the timing and the ordering of events in the system while preserving the local views of the processes.

H' is the shifting history of the history H by timing t for process p, if the only difference of H and H' is that $\delta_p(H')=\delta_p(H)+t$. In the shifting history, the global times at which the events occure at p are changed while p's process history isn't changed. The shifting changes the message delays so that the delay of any message to p decreases by t, and the delay of any message from p increases by t. For any integer n, if H_1 is the shifting history of H by t_1 for p_1, and H_i is the shifting history of H_{i-1} by t_i for p_i for $1<i\leq n$, we briefly say that H_n is shifting history of H by t_i for p_i for $1\leq i\leq n$.

Theorem 3 *In the case where the number of processes is more than or equal to $2m-1$, $E_{res}\geq u\frac{m-1}{m}$ holds.*

proof: Let $p_0,p_1,\ldots,p_{m-1}$ and $q_0,q_1,\ldots,q_{m-2}$ be $2m-1$ processes that access to some FIFO queue Q. Initially, Q is empty. Assume $E_{res}=T<u\frac{m-1}{m}$. Consider following two histories.

history1 $Enq_{p_i}(Q,v_i)$ occurs at global time $u\frac{i}{m}$ for $0\leq i\leq m-1$, and $Deq_{q_j}(Q)$ occurs later than $u\frac{m-1}{m}+T$ for $0\leq j\leq m-2$ (Fig5(a)). The message delays are d from p_i to p_j $(i<j)$, $d-u$ from p_i to p_j $(i>j)$, $d-u\frac{i}{m}$ from p_i to q_j, and $d-u\frac{m-i}{m}$ from q_j to p_i. Since all message delays are in the range $[d-u,d]$, this history is admissible.

In history1, every *Ack* occures not later than $u\frac{m-1}{m}+T$, that is, every *Ack* occurs earlier than any *Deq*. Thus, $m-1$ *Deq*s return $m-1$ values of m values that are enqueued by previous *Enq*s. Let v_k be a value that is not dequeued. It is not v_0, that is $1\leq k\leq m-1$, because Ack_{p_0} occured not later than T and $Enq_{p_{m-1}}$ occurs at $u\frac{m-1}{m}$ $(>T)$.

history2 It is the shifting history of history1 by $u\frac{m-k}{m}$ for $p_0,p_1,\ldots,p_{k-1}$ and by $-u\frac{k}{m}$ for $p_k,p_{k+1},\ldots,p_{m-1}$ (Fig5(b)). The message delays become $d-u$ from p_i to p_j $(i<k,j\geq k)$, d from p_i to p_j $(i\geq k,j<k)$, $d-u\frac{i+m-k}{m}$ from p_i to q_j $(i<k)$, and $d-u\frac{i-k}{m}$ from p_i to q_j $(i\geq k)$, and all other delays are unchanged. All message delays are in the range $[d-u,d]$, so history2 is admissible.

In history2, Enq_{p_k} occurs at global time 0, its Ack_{p_k} occurs not later than T, and $Enq_{p_{k-1}}$ occurs at real time $u\frac{m-1}{m}$ $(>T)$. As history1, $m-1$ *Deq*s return $m-1$ values of m values that are enqueued by m *Enq*s in history2. Since Ack_{p_k} occured not later than T and $Enq_{p_{k-1}}$ occurs at $u\frac{m-1}{m}$ $(>T)$, v_k must be included these $m-1$ values. But no *Ret* returns v_k because every process histories are same as history1. A contradiction.

5 conclusion

We show three result for the cost of linearizable implementation of virtual shared FIFO queues in the distributed multiprocessor system. Attiya shows that the result for FIFO queues can be extended to apply for *stacks*([5]). Our results apply for stacks with replacing *Enq* and *Deq* by *Push* and *Pop* respectively.

Moreover, the implementation of FIFO queues presented here can be extended to apply any object that have only total operations that are defined for any object states in its sequential specification. In the implementation, when the response for *Deq* is returned, the operation sequence up to the *Deq* is fixed. In the same manner, at $2d$ after call of any operation, the operation sequence up to the operation can be fixed. Each process can generate the response for own call at $2d$ after the call, by updating local copies of objects in the order of this fixed operation sequence. Therefore, there exists a linearizable implementation of any object that have only total operations, such that the worst-case response time of any operation is $2d$.

References

[1] L. Lamport: "How to make a multiprocessor computer that correctly executes multiproess programs", IEEE Trans. on Computers, **C-28**, 9, pp. 690–691 (1979).

[2] M. Herlihy and J. Wing: "Linearizability: A correctness condition for concurrent objects", ACM TOPLAS, **12**, 3, pp. 463–492 (1990).

[3] Y. Afek, G. Brown and M. Merritt: "A lazy cach algorithm", Proc. 1st ACM Symp. on Parallel Algorithms and Architectures, pp. 209–222 (1989).

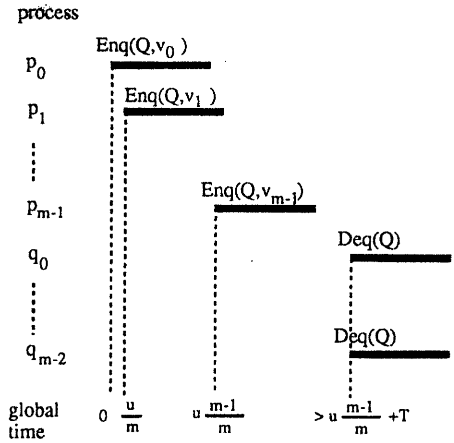
[4] H. Attiya and J. Welch: "Sequential consistency versus linearizability", Proc. of the 3rd ACM Symp. on Parallel Algorithms and Architectures, pp. 305–315 (1991).

[5] H. Attiya: "Implementing FIFO Queues and Stacks", Proc. of the 5th Int. Workshop on Distributed Algorithms (LNCS 579), pp. 80–94 (1991).

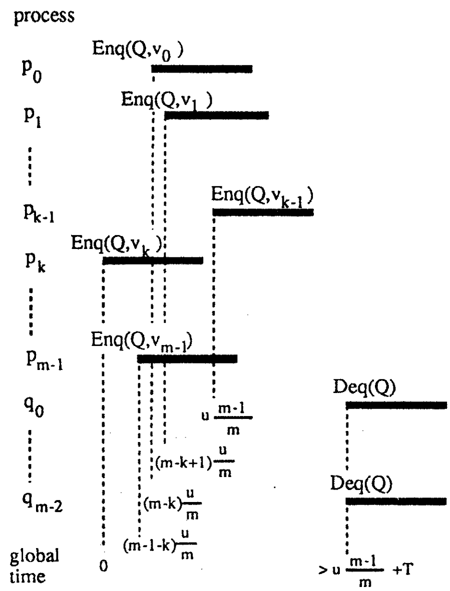
[6] M. Mavronicolas and D. Roth: "Efficient, strongly consistent implementaions of shared memory", Proc. of the 6th Int. Workshop on Distributed Algorithms (LNCS 647), pp. 346–361 (1992).

[7] M. Herlihy: "Wait-free synchronization", ACM TOPLAS, **11**, 1, pp. 124–149 (1991).

[8] W. Brantley, K. McAuliffffe and J. Weiss: "Rp3 processos-memory element", Proc. Int. Conf. on Parallel Processing, pp. 782–789 (1985).



(a) history1



(a) history2

Fig5. history1 and history2 in proof of theorem3.