A total ordering protocol using a dynamic token-passing scheme

1997 Distrib. Syst. Engng. 4 87

(http://iopscience.iop.org/0967-1846/4/2/003)

Distrib. Syst. Engng **4** (1997) 87–95. Printed in the UK PII: S0967-1846(97)83082-3

A total ordering protocol using a dynamic token-passing scheme

Jongsung Kim† and Cheeha Kim‡

Department of Computer Science and Engineering/PIRL, Pohang University of Science and Technology, San 31, Hyoja-Dong, Pohang, 790-784 Korea

Received 19 January 1996, in final form 16 January 1997

Abstract. Solutions to the total ordering problem can be used to maintain consistency in distributed system applications such as replicated databases. We propose a total ordering protocol based on a dynamic token-passing scheme which determines the next token holder dynamically, not in predetermined order. The proposed protocol provides fast stability time, uses a small buffer, and distributes evenly the load of ordering messages to accomplish a total message ordering. We present simulation results to illustrate that the performance of the proposed protocol is superior to that of existing token-based total ordering protocols.

1. Introduction

In distributed systems, data are replicated on several autonomous hosts in order to increase availability and reliability. To maintain consistency on such systems, the updates for data must be performed in the same order at all processes managing the replicated data, despite random communication delays and failures. Note that each update is multicast to all managing processes. A total ordering protocol ensures that the updates are delivered to all processes in the same order, and so relieves the application programmers from the burden of dealing with the difficult issue of maintaining consistent replica. It is well known that total ordering is one of the fundamental services in the communication subsystem for a distributed system and is a perfectly reasonable technology to use side-by-side with transactional systems, even if it is not a complete solution to replicated transactional data and control/communication applications [6, 8, 22]. So, total ordering has been adopted in many recent communication subsystems for distributed systems such as ISIS [5], Ameoba [14, 13], Transis [2], Totem [1, 3], Consul [17]. There are many other applications which require the ordering of multicast messages, such as the coordination of cooperating processors, distributed load balancing mechanisms, distributed transaction management, and so on.

There are several total ordering protocols including the Totem protocol, the CM protocol [7], the Pinwheel protocol [10], the protocol implemented in the Ameoba operating system [13, 14], the ABCAST protocol in the ISIS system [5], trans and total protocol [16], Psync [20]. These protocols are classified into three classes according to the scheme of ordering the multicast messages [10]: sequencer-based [4, 5, 12, 14, 13, 19], token-based [3, 7, 10, 15, 21, 24], and history-based [2, 16, 20]. In the sequencer-based protocols, a unique server, called the sequencer, is responsible for making a total ordering on all messages passing in the system. In the token-based protocols, there is a cyclic order of processes in which a single token circulates. A process holding the token plays the role of the sequencer in the sequencer-based protocols. In the communication history-based protocols, a partial ordering of multicast messages is constructed in a distributed manner and then converted into a total ordering. History-based protocols require a rather large buffer space when all the processes are not equally active. In the sequencer-based protocols, the sequencer is overloaded and its failure is catastrophic. But the token-based protocols provide comparable performance and suffer least when transient failures affect the communication among group members [9]§. Other protocols in [11, 12] deal with the total ordering on the overlapping groups, when the system is composed of multiple groups and the groups may be overlapped, i.e. processes may belong to one or more groups.

In this paper, we propose another token-based total ordering protocol which is more efficient than the others. In token-based total ordering protocols, a process requests its local total ordering module (LTM) to multicast an update U and the LTM passes it to the underlying transport module for multicasting to the LTMs in sites where the replicated database is maintained. Upon receiving U, the token holder, i.e. the LTM holding token, determines and multicasts U's delivery order to the other LTMs. All LTMs deliver U to local processes in the delivery order and store it in the local buffer for retransmission until U is *stable*, that is, U has been delivered to all processes. Stable messages (we will use update and

† E-mail address: jskim@turing.postech.ac.kr
‡ E-mail address: chkim@vision.postech.ac.kr

§ A comparative study of sequencer- and token-based protocols is exhaustively conducted in [9].

message exchangeably) are immediately removed from the local buffers. To evaluate the performance of total ordering protocols, we consider the following performance metrics: average delivery time defined as the average interval from the moment a process entrusts multicasting an update U to the moment U is delivered to every other process, average stability time, average number of messages required to accomplish the total ordering for an update, and how evenly the load generated to provide the total ordering service is distributed over LTMs as in [10]. Most of the existing total ordering protocols perform comparably in these metrics except average stability time [10, 9]. Average stability time is defined as the interval from the moment a LTM receives the request of multicasting an update U from a process to the moment all LTMs learn that U is *stable*. The stability time is important for distributed systems since it measures how fast all processes achieve agreement on the communication history representing the status of each process. The stability time is also used to measure how much buffer space a LTM needs. In this paper, we propose an efficient token-based total ordering protocol, called the *Dynamic Token-Passing* (DTP) protocol, which a LTM carries out. While the token rotates among group members in the predetermined order in existing token-based protocols, in the DTP protocol, the next token holder is dynamically determined at run-time, resulting in reducing the stability time significantly.

This paper is structured as follows. We describe our system model in section 2. Section 3 describes the dynamic token-passing scheme. Section 4 describes the DTP protocol in detail and presents the formal correctness proof. Section 5 discusses the performance of the proposed protocol compared with that of other existing token-based protocols. Finally, section 6 gives some concluding remarks.

2. System model

We assume that a distributed system consists of n processes communicating with each other by exchanging messages. There is neither shared memory nor a common physical clock. Each process is denoted by P_{ID} where ID is a unique *ID* in the range of $[1, n]$. We do not make any assumption about the communication network topology: it can be either a broadcast channel or a set of point-to-point channels.

P_i requests its local LTM_i to multicast an update U. LTM_i generates a message consisting of U and corresponding control information and passes it to the underlying transport module for multicasting. The transport module provides a datagram communication service, such as UDP with omission/performance failure semantics. That is, the transport module is responsible for multicasting the message, but it does not guarantee reliable message transmission or any bound on the transmission delay. Every LTM_j *receives* messages from its local transport module and *delivers* them to P_j in the same order by assistance of the token holder. The messages created for total ordering by the token holder are also multicast.

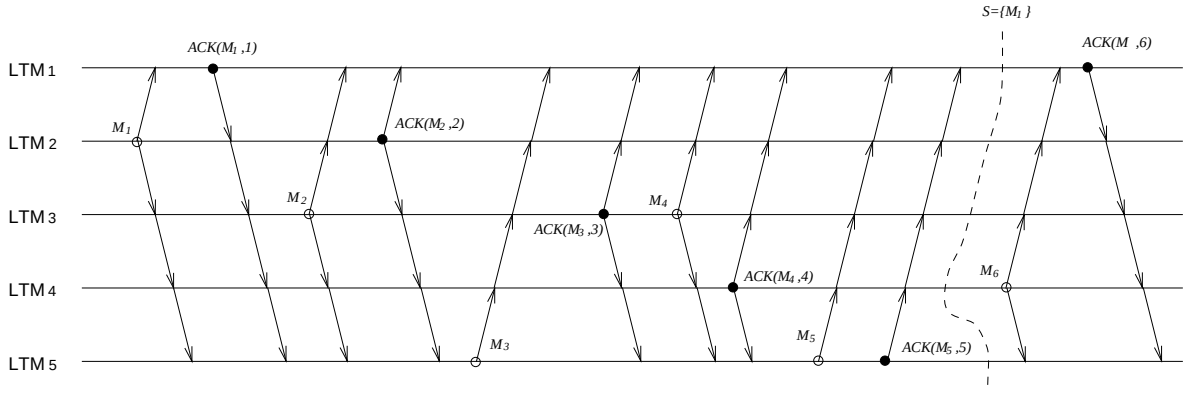
We assume that there is a group membership protocol [3, 5, 18, 20, 23]. Every LTM agrees on the initial group members with the assistance of the group membership protocol. The group membership protocol deals with group membership changes due to failures such as processor failures and network partitions. A message can be lost because of transmission errors, buffer overflow and internetworking units such as bridges, routers and gateways. A failure occurs when LTM fails to communicate with another LTM after A attempts, and causes the membership to be redefined by the group membership protocol. To concentrate on the total ordering problem, we assume that no group membership change occurs. In other words, that every LTM will receive all messages eventually from its local transport module.

3. Dynamic token-passing scheme

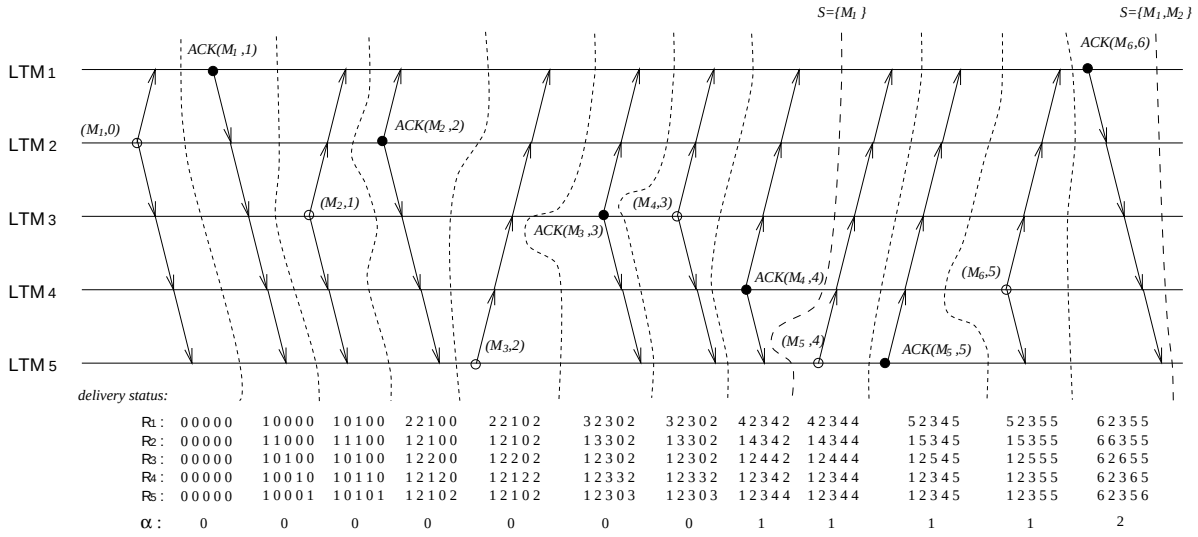
In the CM protocol [7], the *token* is circulated among LTMs along the predefined path. When LTM_i receives a request to multicast U from P_i, it multicasts the message $MSG(U, U_{id})$. U_{id} is the identifier of U consisting of two fields (i, l) where i identifies the sender of U and l is a local sequence number (lsn) assigned by LTM_i. The message $MSG(U, (i, l))$ is the lth message LTM_i multicasts. Let LTM_t denote the LTM holding the token. LTM_t acknowledges $MSG(U, U_{id})$ from LTM_i by multicasting $ACK(t, U_{id}, g)$ where t is the sender's ID and g is the unique global sequence number (gsn) assigned by LTM_t. And each $LTM_j, 1 \leq j \leq n$, delivers messages in an orderly way (actually, updates excluding control information) to P_j according to their gsns. LTM_t is allowed to acknowledge a new message after receiving all the messages already acknowledged. After $MSG(U, U_{id})$ and L subsequent messages are acknowledged, it is certain that $L+1$ processes have obtained U. Therefore U becomes stable after the message $MSG(U, U_{id})$ and $n-1$ subsequent messages are acknowledged.

The Pinwheel protocol is mostly similar to the CM protocol, but it adopts additionally the *positive notification scheme* of the message delivery status in order to reduce the stability time. Let $hgsn_i$ denote gsn of the message which LTM_i has delivered to P_i most recently. $hgsn_i$ represents the message delivery status of LTM_i. Each LTM_i piggybacks $hgsn_i$ in messages it sends out, and maintains $hgsn_j$ it has received from LTM_j. Note that the gsn assigned to the message by LTM_t is equal to $hgsn_t$. Let $R_i[k]$ denote $hgsn_k$ which LTM_i knows at present. The stability time of the message with gsn of g is recognized by LTM_i when $g \leq min(\{R_i[k] \mid 1 \leq k \leq n\})$. In virtue of the positive notification scheme, LTM_i can determine whether a message is stable in a shorter time than the CM protocol.

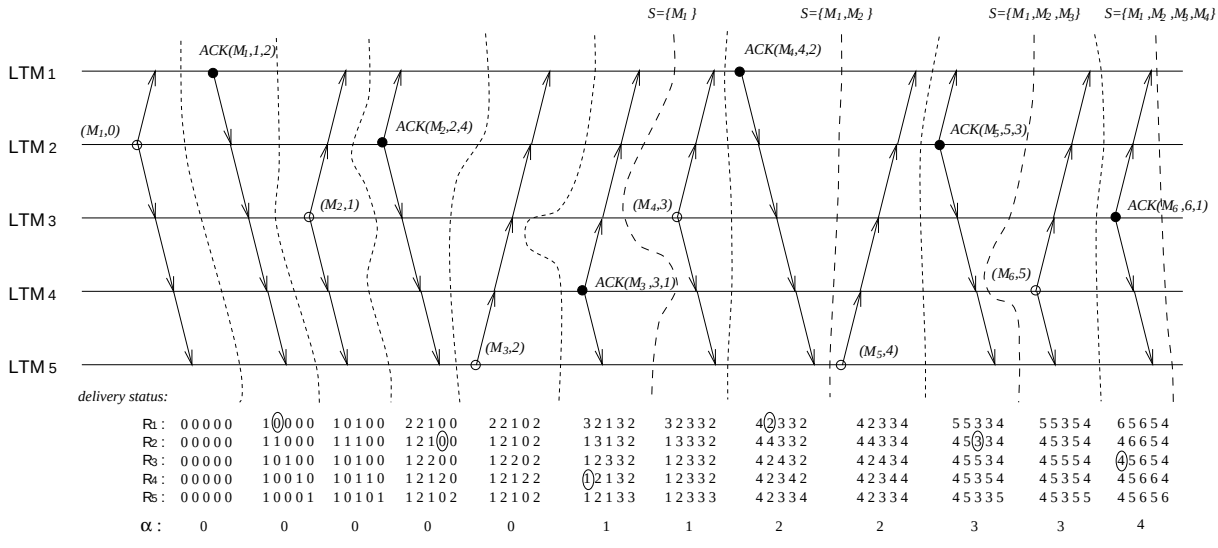
We propose the scheme of passing the token to the least active LTM. This scheme reduces the stability time further, since an inactive LTM can notify its own $hgsn$ in a rather short time. In the Pinwheel protocol, a LTM has to wait to notify its delivery status until it has a message to send or it becomes the token holder. To give an inactive LTM a chance to notify its $hgsn$, the token



(a)



(b)



(c)

Figure 1. Stability time: (a) CM protocol, (b) Pinwheel protocol, (c) proposed protocol.

will be passed to the least active LTM. Then this new token holder can notify its $hgsn$ using the ACK message. LTM_t can determine the least active LTM_m such that $R_t[m] = min(\{R_t[k] \mid 1 \leq k \leq n\})$. We call this scheme the *dynamic token-passing scheme*. This scheme allows inactive LTMs to notify their own delivery status in a rather short time. As a result, the stability time can be significantly shortened.

The space–time diagrams shown in figure 1 describe examples of these three protocols performed on a group consisting of five members for the same sequence of message transmissions. For simplicity, in figure 1, M_z and $ACK(M_z, g)$ denote $MSG(U, U_{id})$ and $ACK(t, U_{id}, g)$, respectively, where M_z represents the zth message sent in the system. S denotes a set of stable messages. It is assumed that initially the token holder is LTM_1. In figure 1(a), M_1 becomes stable when M_1 and four subsequent messages are acknowledged, i.e. after the fifth message has been acknowledged. In figure 1(b), h in (M_z, h) sent by LTM_i represents the message delivery status of LTM_i, i.e. $hgsn_i$. Due to the positive notification scheme, M_1 becomes stable after the fourth message has been acknowledged. In figure 1(c), m in $ACK(M_z, g, m)$ is the next token holder's ID selected by LTM_t and the circled number in $R_t[1 \ldots n]$ of LTM_t represents $min(\{R_t[k] \mid 1 \leq k \leq n\})$. Figure 1(c) illustrates the effectiveness of the dynamic token-passing scheme. M_1 becomes stable after the acknowledgement of the third message. Note that when the sixth message has been acknowledged in figure 1(c), four messages become stable. This is a significant improvement.

4. The total ordering protocol

4.1. The data structures

LTM_i has two state variables as follows:

- $S.Token_i$: this variable represents whether LTM_i holds the token or not. The state is one of the following:
  - *HOLD*: LTM_i holds the token.
  - *TRST*: after passing the token to the next token holder, LTM_i waits for the acknowledgment of the token transfer.
  - *EMPT*: LTM_i does not hold the token.
- $S.ACK_i$: this variable represents whether LTM_i has received ACK for the message it sent most recently. The state is one of the following:
  - *DONE*: LTM_i received ACK of the message it sent most recently.
  - *WAIT*: LTM_i is waiting for ACK of the message it sent most recently.

Initially, $S.Token_1$ of LTM_1 is *HOLD* and *EMPT* for the others. At any moment, there exists at most one LTM, denoted by LTM_t, whose $S.Token_t$ is *HOLD* and LTM_t is responsible for ordering a new message. $S.ACK_i$ is initially *DONE*. LTM_i is allowed to send another message only when its own $S.ACK_i$ is *DONE*. When a new message is multicast, $S.ACK_i$ is set to *WAIT*, and when it is acknowledged, $S.ACK_i$ is reset to *DONE*. The message flow is controlled by $S.ACK_i$.

In addition, LTM_i maintains two array variables and one buffer as follows:

- $N_i[k]$: the expected lsn of the next message from LTM_k $(1 \leq k \leq n)$.
- $R_i[k]$: $hgsn_k$ which LTM_i knows currently $(1 \leq k \leq n)$.
- BUF_i: this buffer contains the messages which are not stable locally yet.

Initially, $N_i[k] = 1$ and $R_i[k] = 0$ for all k. $N_i[i]$ contains lsn for the next message of LTM_i. $R_i[j]$ is used for two purposes: first, it is used to determine stable messages in BUF_i, and second, it is used to determine the next token holder if LTM_i is the token holder. $R_i[i]$ contains $hgsn_i$ representing the message delivery status of LTM_i.

4.2. DTP protocol

The DTP protocol uses three message types: MSG, ACK and REQ. MSG consists of an update U, U_{id}=(MSG's sender ID, lsn) and $hgsn$ of the sender. ACK consists of ACK's sender ID, U_{id}, gsn, and the next token holder's ID. REQ consists of REQ's sender ID, $hgsn$ of the sender and the lost message list, $msglst$ which contains a set of ('msg',U_{id})s or ('ack',gsn)s.

There are three kinds of events associated with the DTP protocol. They are update arrivals, message receptions and timeouts. At the occurrence of an event, the corresponding procedure is executed.

Update arrival Upon receiving the transmission request of an update U from P_i, LTM_i checks whether $S.ACK_i$ is *WAIT*. If so, multicasting U is deferred until $S.ACK_i$ has been *DONE*. Otherwise if $S.ACK_i$ is *DONE*, it constructs and multicasts $MSG(U, (i, N_i[i]), R_i[i])$. Then LTM_i sets $S.ACK_i$ to *WAIT*, increments $N_i[i]$, and saves the message in BUF_i until the message is stable.

If LTM_i holds the token, it must recover all of the lost messages before acknowledging a new message. LTM_i increments $R_i[i]$ by one and assigns gsn $(= R_i[i])$ to the message. It selects the next token holder LTM_m such that $R_i[m] = min(\{R_i[k] \mid 1 \leq k \leq n\})$. To LTM_i, LTM_m seems the least active. LTM_i multicasts $ACK(i, U_{id}, R_i[i], m)$ and puts it into BUF_i. LTM_i also delivers U to P_i and sets $S.Token_i$ to *TRST*. Since the MSG has been acknowledged, LTM_i resets $S.ACK_i$ to *DONE*.

Receiving MSG message When LTM_i receives $MSG(U, (j, l), h)$, if $l = N_i[j]$, LTM_i increments $N_i[j]$, changes $R_i[j]$ to h, puts the message in BUF_i, and waits for gsn of the message. If LTM_i holds the token, it performs for the MSG the procedure multicasting an ACK message as mentioned above. If $l > N_i[j]$, it recognizes that it lost messages from LTM_j, constructs and multicasts $REQ(i, R_i[i], msglst)$, and waits for the response from the token holder. If $l < N_i[j]$, it discards the message. But if LTM_i holds the token, LTM_i assumes that LTM_j lost the ACK message related to the MSG, and retransmits to LTM_j the ACK message with $U_{id} = (j, l)$ in BUF_i.

Receiving ACK message Upon receiving $ACK(t, U_{id}, g, m)$, LTM_i puts ACK into BUF_i if $g = R_i[i] + 1$. Then, it updates both $R_i[t]$ and $R_i[i]$ to g, and delivers immediately U with U_{id} in BUF_i to P_i. If U is an update from P_i, i.e. $U_{id} = (i, N_i[i] - 1)$, LTM_i resets $S.ACK_i$ to $DONE$. But if it fails to find such a U in BUF_i, it requests the token holder to retransmit the lost message by multicasting $REQ(i, R_i[i], \{(\text{'}msg\text{'}, U_{id})\})$. After multicasting a REQ message, LTM_i multicasts the REQ message at every timeout until it receives the requested message. If $g < R_i[i] + 1$, it discards the message. If $g > R_i[i] + 1$, it recognizes that it lost ACKs, constructs $REQ(i, R_i[i], msglst)$, and requests the token holder to retransmit them by multicasting it. When $S.Token_i = TRST$, if $g \geq R_i[i] + 1$ or $i = m$, LTM_i assumes that the token has been successfully passed to the next token holder, and changes $S.Token_i$ from *TRST* to *EMPT*.

If $i = m$, then LTM_i recognizes that it has become the token holder and sets $S.Token_i$ to *HOLD*. After recovering all lost messages if any, LTM_i checks if BUF_i contains an unacknowledged MSG whose acknowledgment is not in BUF_i. If so, LTM_i sends an ACK message for the MSG. Otherwise, LTM_i waits for the arrival of a new $MSG(U', (k, s), h')$ such that $s = N_i[k]$ during a maximum silence period T.

Receiving REQ message Upon receiving $REQ(j, h, msglst)$, LTM_i, if $S.Token_i$ = *HOLD*, sends to LTM_j the requested message(s) in $msglst$ as a point-to-point message. If $S.Token_i$ = *EMPT*, it discards the REQ message. If $S.Token_i$ = *TRST*, LTM_i checks if the request message is multicast by its next token holder, i.e. if $m = j$ in $ACK(i, U_{id}, R_i[i], m)$ stored in BUF_i. If so, LTM_i assumes that its next token holder LTM_j requests LTM_i to retransmit lost messages, and retransmits the requested messages to LTM_j as a point-to-point message.

Timeout After multicasting a MSG, LTM_i multicasts the same MSG at every timeout until it receives an acknowledgment for the message. After multicasting a $ACK(i, U_{id}, g, m)$ message, in order to ensure that the token is successfully passed to the next token holder LTM_m, LTM_i also multicasts the same ACK message at every timeout until it receives $ACK(x, U'_{id}, g', y)$ such that $R_i[i] + 1 \leq g'$ or $x = m$.

If LTM_i is the token holder and there is no new MSG in BUF_i, LTM_i waits for the arrival of a new MSG during a maximum silence period T. If no new MSG arrives in T, LTM_i multicasts $ACK(i, NULL, R_i[i], m)$. In other words, LTM_i passes the token without ordering any message in order to bound the stability time. When LTM_i receives $ACK(j, NULL, g, m)$, if $g < R_i[i]$, LTM_i discards the ACK message. Otherwise, it updates $R_i[j]$ to g. If $g > R_i[i]$, LTM_i recognizes that it lost ACKs and requests the token holder to retransmit the lost messages by multicasting a REQ message.

So far, we have described procedures invoked at the occurrence of an event. In the following, we point out the important operations involved in the procedures and some remarks.

Removing stable messages Whenever LTM_i receives the delivery status h from LTM_j, it checks if $h > R_i[j]$. If so, it updates $R_i[j]$ to h and removes the stable MSGs whose gsn is less than or equal to $min(\{R_i[k] \mid 1 \leq k \leq n\})$ and ACKs associated with them from BUF_i.

Flow control and buffer space The flow control scheme of the DTP protocol is identical to that of the CM protocol. Each LTM is allowed to have only one outstanding message at a time. So, LTM_i may store up to n unacknowledged MSGs in its BUF. And acknowledged messages must stay in BUF_i until they become stable. In the worst case, MSG becomes stable after its subsequent $n - 1$ MSGs are acknowledged. Therefore, BUF_i holds $2n - 1$ MSG and $n - 1$ ACK messages at most.

Adjusting degree of fault tolerance If the failures of F sites in a system do not prevent the system from maintaining the consistent state, the system is *F-resilient* to *faults*. This system is referred to as an *F-resilient* system. F represents the reliability degree of the system and will be variable according to the requirement for the fault tolerance of applications. Given F, the *F-resilient* system can be built as follows. LTM_i defers the delivery U until $|\{R_i[k] \mid R_i[k] \geq g\}| \geq F + 1$ where g is the gsn of U. When the condition is satisfied, it is certain that $F + 1$ processes have obtained U. At this time, LTM_i delivers U to P_i. Then as long as F or fewer sites fails, all lost messages can be recovered. The delivery service of the DTP protocol is identical to the agreed delivery service of the Totem protocol. When F is $n - 1$, the total ordering service of the DTP protocol represents the safe delivery service of the Totem protocol [3].

4.3. Correctness

Lemma 1. The token holder acknowledges a new message which has never been acknowledged.

Proof. Let $MSG_{u,v}$ denote a MSG whose identifier of update is (u, v) and $ACK_{u,v}$ denote an acknowledgment for $MSG_{u,v}$. Suppose that LTM_i is a token holder. LTM_i acknowledges one of the following messages after recovering lost messages if any:

(1) $MSG_{k,s}$ such that $s = N_i[k]$, to be received by LTM_i and

(2) $MSG_{k,s}$ such that $MSG_{k,s} \in BUF_i$ and $ACK_{k,s} \notin BUF_i$.

We prove by contradiction that the messages are new messages.

(i) Suppose that $MSG_{k,s}$ is a message which has been acknowledged. Then it is certain that LTM_i has received $MSG_{k,s}$ since it has no lost message. Initially, $N_k[l] = 1$ for $1 \leq k, l \leq n$. LTM_k increments $N_k[k]$ by one whenever it sends $MSG_{k,u}$ where $u = N_k[k]$. LTM_i increments $N_i[k]$ by one whenever it receives $MSG_{k,u}$ where $u = N_i[k]$. Therefore, $s = N_i[k]$

implies that LTM_i has never received $MSG_{k,s}$. This shows contradiction.

(ii) Suppose that $MSG_{k,s}$ is a message which has been acknowledged. Then it is certain that LTM_i has received both $MSG_{k,s}$ and $ACK_{k,s}$ and stored them in BUF_i since it has no lost messages. And $s < N_i[k]$ since LTM_i increments $N_i[k]$ when it receives $MSG_{k,s}$. If $MSG_{k,s}$ is a stable message, LTM_i will remove $MSG_{k,s}$ from BUF_i. Even when it receives $MSG_{k,s}$ again, where $s < N_i[k]$, LTM_i will discard $MSG_{k,s}$. Therefore, $MSG_{k,s} \notin BUF_i$. Otherwise if $MSG_{k,s}$ is not a stable message, $ACK_{k,s}$ should be in BUF_i since LTM_i removes only stable $MSG_{u,v}$s and $ACK_{u,v}$s from BUF_i. This shows contradiction. □

Theorem 1. **(Safety)** LTM_i, $1 \leq i \leq n$, achieves total ordering of the messages.

Proof. MSGs are multicast to group members and acknowledged with gsn by the token holder. $LTM_k, 1 \leq k \leq n$, delivers them to P_k in the increasing order of gsn. Let MSG^x denote the xth MSG acknowledged by token holder and ACK^k denote ACK related to MSG^x. We show that MSG^x is acknowledged with a unique gsn x by mathematical induction on $x \geq 1$.

Base step: Initially, $R_k[l] = 0$ for $1 \leq k, l \leq n$ and a token holder is LTM_1. Upon receiving MSG^1, LTM_1 increments $R_1[1]$ and acknowledges MSG^1 by multicasting ACK^1 with a unique gsn $R_1[1]$ where $R_1[1] = 1$.

Induction step: Suppose MSG^x is acknowledged with a unique gsn x for all x such that $x \leq m$. Let LTM_j be ACK^{x-1}'s sender and LTM_i be the next token holder of LTM_j. We consider the case of $x = m + 1$. LTM_i increments $R_i[i]$ by one whenever it receives ACK^k or it as token holder sends ACK^k where $k \geq 1$, and stores ACK^k in BUF_i. Upon receiving ACK^{x-1}, LTM_i can find lost message ACK^k where $R_i[i] + 1 \leq k < x - 1$ and related MSG^k such that $MSG^k \notin BUF_i$. So, if LTM_i detects the lost messages, it constructs a list $msglst$ of the lost messages and requests LTM_j to retransmit them by sending $REQ(i, hgsn_i, msglst)$. Note that LTM_j multicasts ACK^{x-1} after recovering lost messages and BUF_j will contain unstable messages MSG^v and ACK^v where $min(\{R_j[l] \mid 1 \leq l \leq n\}) < v \leq x - 1$. Since LTM_j maintains in $R_j[i]$ $hgsn_i$, i.e. $R_i[i]$, in messages from LTM_i, $R_i[i] \geq min(\{R_j[l] \mid 1 \leq l \leq n\})$. Therefore, BUF_j will contain all of the requested messages in $msglst$. Upon receiving the REQ message, LTM_j will repond to the request from its next token holder LTM_i by retransmitting the requested messages. After the lost messages have been recovered, $R_i[i]$ will be $x - 1$. Now, LTM_i acknowledges MSG^x. It follows from lemma 1 that MSG^x is a new message which has not been acknowledged. LTM_i increments $R_i[i]$ and sends ACK^x with a unique gsn $R_i[i]$ where $R_i[i] = x$. □

Theorem 2. **(Liveness)** Every message is eventually delivered.

From theorem 1, LTM_i will get the messages and related acknowledgments with unique gsn and deliver immediately the messages in increasing order of gsn.

5. Protocol performance

We present the simulation results for the performance comparison between the proposed DTP protocol, the CM protocol, the Pinwheel protocol and the Totem. The lost message detection and retransmission scheme of the DTP protocol is similar to those of above protocols. So, we have simulated in the absence of failures the behaviour of these protocols on an Ethernet by sending ten thousand

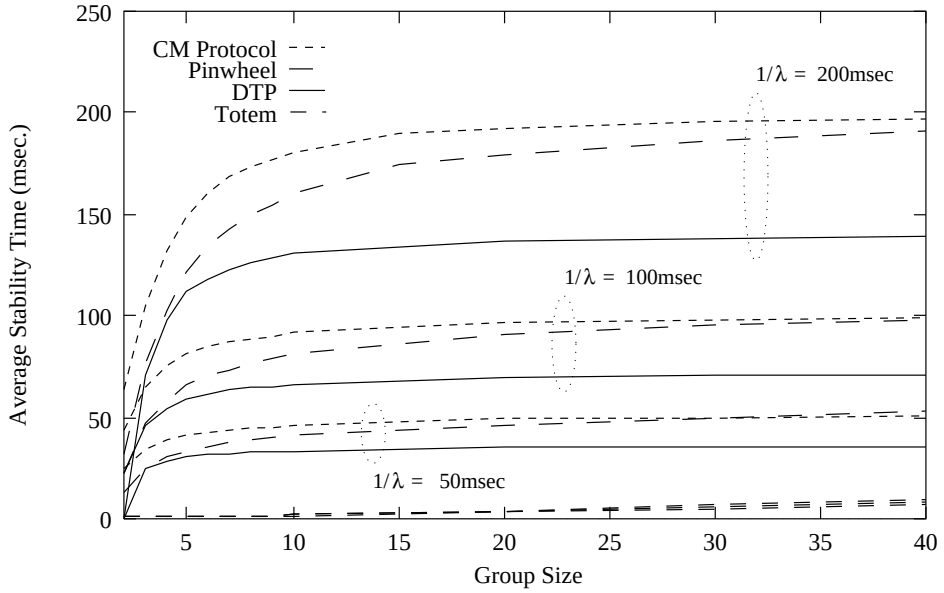


Figure 2. Comparison for average stability time.

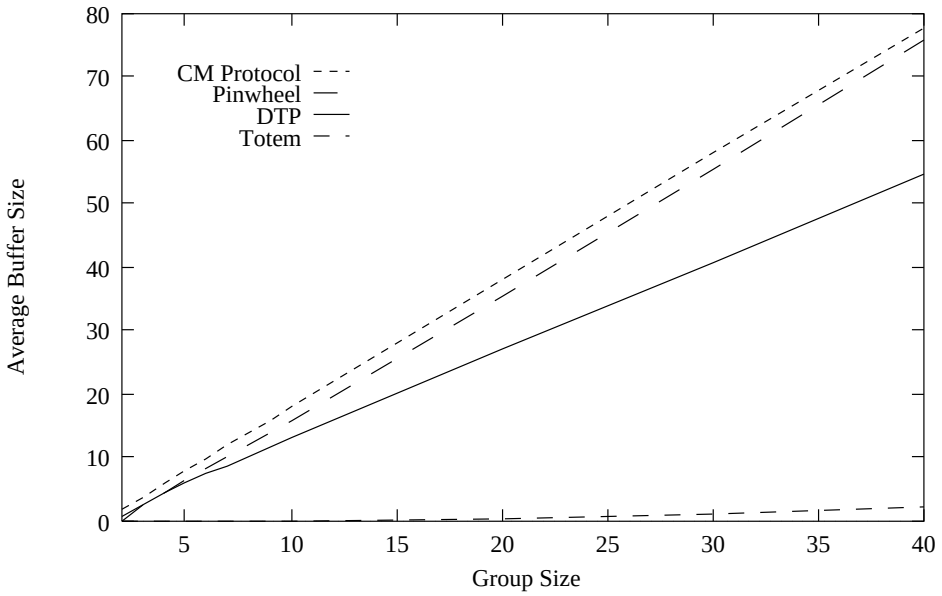


Figure 3. Comparison for average buffer size: $1/\lambda$ = 100 ms.

500 byte long messages. We assume that the CPU time needed to process a message is zero [9]. We also assume that update arrivals at every LTM_i form an independent Poisson process with the same rate λ. A silence period (T) of 100 ms is used.

Figure 2 shows the average stability time. The results show that for three cases where $1/\lambda = 50$, 100 and 200 ms, the DTP protocol is superior to the others except the Totem. The stability time of the DTP protocol is bounded to about two-thirds of $1/\lambda$ when the group size increases. But the stability times of the CM protocol and the Pinwheel protocol approach nearly $1/\lambda$ as the group size increases. In virtue of the positive notification of delivery status, the Pinwheel protocol can achieve shorter stability time than that of the CM protocol. The stability time of the Pinwheel protocol is further shortened by using the dynamic token-passing scheme as in the DTP protocol. The Pinwheel protocol allows LTMs to send up to m multicast messages without being acknowledged, where m is determined by its own flow control mechanism, and the token holder to acknowledge one or more multicast messages. When the message transmission rate at the LTM is very high, this scheme may help to shorten the average delivery time of the messages, but it extends the stability time as seen in the case where group size is 40 and $1/\lambda$ is 50 ms.

To illustrate the simulation result of the Totem, we briefly describe how the Totem achieves the total ordering. The token with a sequence number field, called *seq*, circulates among LTMs as a point-to-point message. On receipt of the token, LTM multicasts more than one message, subject to the constraints imposed by the flow control mechanism, and transmits the token with its own message delivery information to the next LTM. For each message, it increments the *seq* field of the token and sets the sequence number of the message to this value. The total ordering is achieved by using a single sequence of message sequence numbers and the stability of messages is determined by using the message delivery information in the token†. If LTM has no messages to multicast, it immediately passes the token without waiting for a new update arrival. Therefore when no LTM has a message to multicast, the token continues to circulate. This scheme decreases the average stability time of the Totem as in figure 2.

The simulation results for the average buffer size are shown in figure 3. The DTP protocol requires small buffer size compared to the others except the Totem. The difference increases as the group size increases. In the Totem, the continuous circulation of the token causes the average buffer size to decrease.

As shown in figure 4, the average delivery time of the DTP protocol is comparable with the other protocols except the Totem. In the Totem, since a LTM defers multicasting updates until it has hold of the token, the average delivery time increases when the group size grows.

The simulation results of the average number of overhead messages required to accomplish the total ordering of messages are shown in figure 5. The average number of overhead messages per message is comparable with the other protocols except the Totem, i.e. about 1 message per message. In the Totem, due to the continuous token circulation, the number of overhead messages increases when the group size is small, and decreases when the group size grows, equivalently the total message arrival rate increases. So, the Totem is too expensive in terms of the average number of overhead messages per message, causing a wastage of network bandwidth.

Typical history-based and sequencer-based protocols are Transis and ISIS's ABCAST, respectively. Transis constructs a partial ordering of multicast messages from all processes and then converts into a total ordering. So, when

† In Totem, the stability time is approximately two token rotation times [3].

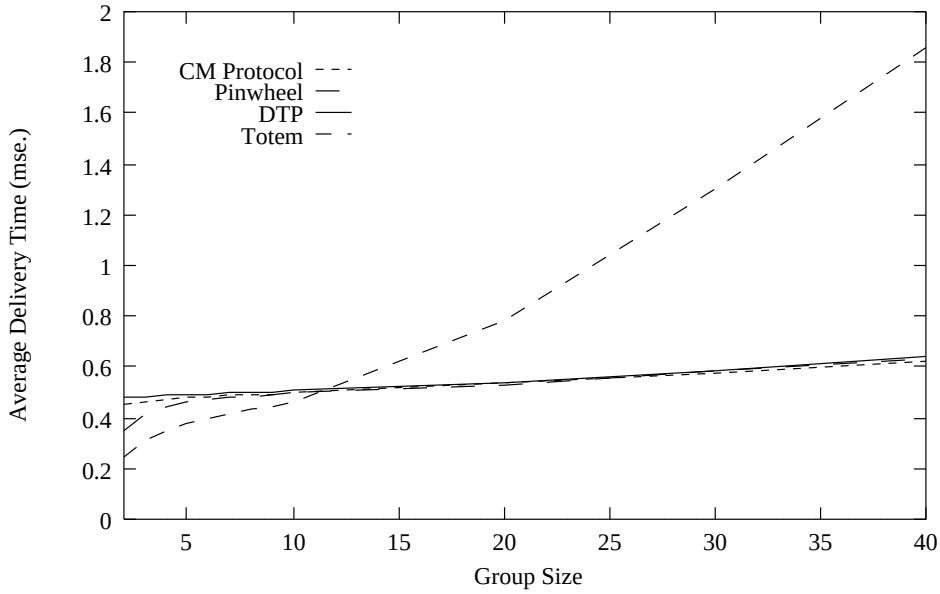


Figure 4. Comparison for average delivery time: $1/\lambda$ = 100 ms.

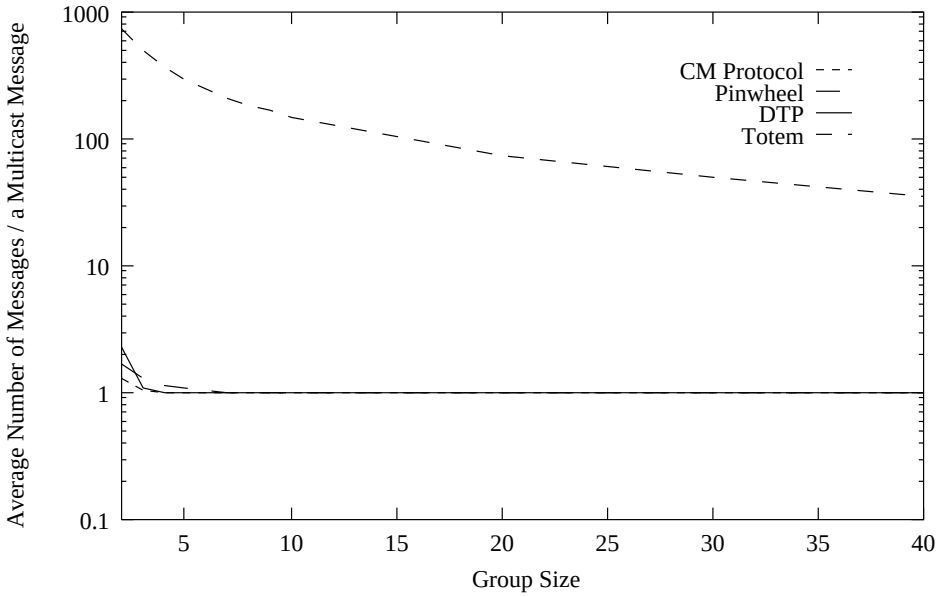


Figure 5. Comparison for average number of messages per multicast message: $1/\lambda$ = 100 ms.

all the processes are not equally active, Transis requires a rather large buffer space and long average delivery and stability time compared to others presented in the simulation results. The performance comparison between the ABCAST and Pinwheel protocols shows that the Pinwheel protocol has better performance than ABCAST [10], so we eliminate ABCAST in the comparison.

The DTP protocol has been implemented in the C programming language on a network of two Sun Sparc20s and one Sun Sparc10 workstation connected by an Ethernet. The implementation uses the UDP broadcast interface in the Unix operating system SunOS 4.1.3. We have considered possible test cases where messages are generated probabilistically. In all test cases, the DTP protocol works successfully as desired. When we have each node ready to broadcast messages at all times, we have observed that about 1000 thousand byte long messages can be handled per second for ordering.

6. Concluding remarks

We have proposed a new token-based total ordering protocol, called the DTP protocol. The DTP protocol uses a new token-passing scheme. The existing token-based protocols construct the logical ring for group members and the logical ring determines the passing order of the token.

In the proposed token-passing scheme, the token holder is dynamically determined in the run-time such that the most inactive member performs the role of the token holder. The proposed token-passing scheme is very effective in the sense of reducing the average stability time and the average buffer size. It allows inactive members to announce their message delivery status rather frequently. The proposed protocol has a substantial performance advantage, particularly with large group sizes. In the DTP protocol the processing load for ordering messages is more effectively distributed among LTMs than in the other protocols such that the most inactive group member is in charge.

References

[1] Agarwal D A, Melliar-Smith P M and Moser L E 1992 Totem: a protocol for message ordering in a wide-area network *Proc. 1st Int. Conf. on Computer Communication and Networks* pp 1–5

[2] Amir Y, Dolev D, Kramer S and Malki D 1992 Transis: a communication sub-system for high availability *Proc. 22nd Int. Symp. on Fault-Tolerant Computing* (Los Alamitos, CA: IEEE Computer Society Press) pp 76–84

[3] Amir Y, Moser L E, Melliar-Smith P M, Agarwal D A and Ciarfella P 1995 The Totem single-ring ordering and membership protocol *ACM Trans. Comput. Syst.* **13** 311–42

[4] Armstrong S, Freier A and Marzullo K 1992 Multicast transport protocol *Request for Comments RFC-1301* IETF

[5] Birman K, Schiper A and Stephenson P 1991 Lightweight causal and atomic group multicast *ACM Trans. Comput. Syst.* **9** 272–314

[6] Birman K 1994 A response to Cheriton and Skeen's criticism of causal and totally ordered communication *ACM Operating Syst. Rev.* **28** (1) 11–21

[7] Chang J and Maxemchuk N F 1984 Reliable broadcast protocols *ACM Trans. Comput. Syst.* **2** 251–73

[8] Cheriton D R and Skeen D 1993 Understanding the limitations of causally and totally ordered communication *Proc. 14th ACM Symp. on Operating Systems Principles*

[9] Cristian F, Beijer R and Mishra S 1994 A performance comparison of asynchronous atomic broadcast protocols *Distrib. Syst. Engng* **1** 177–201

[10] Cristian F and Mishra S 1995 The Pinwheel asynchronous atomic broadcast protocols *Proc. 2nd Int. Symp. on Autonomous Decentralized Systems*

[11] Jia X 1995 A total ordering multicast protocol using propagation trees *IEEE Trans. Parallel Distrib. Syst.* **6** 617–27

[12] Garcia-Molina H and Spauster A 1991 Ordered and reliable multicast communication *ACM Trans. Comput. Syst.* **9** 241–71

[13] Kaashoek M F and Tanenbaum A S 1993 Efficient reliable group communication for distributed systems *Technical Report IR-295* Department of Math and Computer Science, Vrije Universiteit, Amsterdam

[14] Kaashoek M F, Tanenbaum A S, Hummel S F and Bal H E 1989 An efficient reliable broadcast protocol *Operating Syst. Rev.* **23** (4) 5–19

[15] Kim J, Kim C and Seo I 1993 Multicast message ordering on Ethernet-like networks *Proc. 8th Joint Workshop on Computer Communications (Taipei)* pp B1-2-1–B1-2-5

[16] Melliar-Smith P M, Moser L E and Agrawala V 1990 Broadcast protocols for distributed systems *IEEE Trans. Parallel Distrib. Syst.* **1** (1) 17–25

[17] Mishra S, Peterson L L and Schlichting R D 1993 Modularity in the design and implementation of Consul *Proc. 1st IEEE Symp. on Autonomous Decentralized Systems (Kawasaki)*

[18] Moser L E, Amir Y, Melliar-Smith P M and Agarwal D A 1994 Extended virtual synchrony *Proc. 14th Int. Conf. on Distributed Computing Systems* (Los Alamitos, CA: IEEE Computer Society Press) pp 56–65

[19] Oestreicher D 1991 A simple reliable globally-ordered broadcast service *Operating Syst. Rev.* **25** (4) 66–76

[20] Peterson L L, Buchholz N C and Schlichting R D 1989 Preserving and using context information in interprocess communication *ACM Trans. Comput. Syst.* **7** 217–46

[21] Rajagopalan B and McKinley P K 1989 A token-based protocol for reliable, ordered multicast communication *Proc. 8th IEEE Symp. on Reliable Distributed Systems* pp 84–93

[22] Renesse R 1994 Why bother with CATOCS? *ACM Operating Syst. Rev.* **28** (1) 22–7

[23] Schiper A and Sandoz A 1993 Uniform reliable multicast in a virtually synchronous environment *Proc. 13th Int. Conf. on Distributed Computing Systems* (Los Alamitos, CA: IEEE Computer Society Press) pp 561–8

[24] Montgomery T 1994 Design, implementation, and verification of the reliable multicast protocol *Master Thesis* Electrical Engineering Department, West Virginia University