A Nonblocking Approach for Reaching an Agreement on Request Total Orders *

Yun Wang
Key Lab of CNII, MOE
School of Computer Science and Engineering
Southeast University, Nanjing, China, 210096
yunwang@seu.edu.cn

Jie Wu
Dept. of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431, USA
jie@cse.fau.edu

Abstract

In distributed systems that use active replication to achieve robustness, it is important to efficiently enforce consistency among replicas. The nonblocking mode helps to speed up system execution. Unfortunately, this benefit comes at the expense of introducing decision conflicts when the replicas form a single logical token ring and client requests are processed in sequence following the ring. In order to reach an agreement regarding request total orders, this paper proposes a forward-confirmation (FC) approach to identify and solve decision conflicts when up to k successive replicas fail simultaneously. The FC approach can obtain consistent decisions among replicas. An implementation of the FC approach, namely, the queueing method, is proposed. Test results show that our protocol in the nonblocking mode outperforms the Totem protocol regarding delays and failure recovery.

Keywords*: Agreement, nonblocking, performance, replica consistency, total order.*

1 Introduction

As applications of distributed systems continue to grow, robustness becomes more and more important. Usually a critical application service is replicated. Replicas run concurrently on different computers. The active replication technique is an effective approach [6], [11] which gives all replicas the same role without any centralized control. In distributed systems that use active replication, failures are masked by a sufficiently large group of replicas that fail independently. Replicas execute a total ordering protocol which guarantees that requests issued by clients will be processed by all functioning replicas in the same order. Assuming that the behavior of a replica is deterministic, replicas will obtain the same ending state from the same starting state. The active replication technique is useful in many applications, such as CORBA fault tolerant service [10] and its applications [15], P2P file systems [18], globally distributed data repository [6] and real-time scheduling in fault-tolerant systems [21].

There are numerous literatures regarding total ordering protocols [7], especially using the logical token ring [1], [2], [12]. A logical token ring is composed of replicas. Normally, a ring has one token circulating in an unidirectional way. Replicas hold the token in turn. Only the token holder can make a decision. Achieving a consistent total order for client requests is non-trivial in distributed systems that are subject to various failures. To our best knowledge, existing total ordering protocols using the logical token ring are *blocking* in failure scenarios: when there is failure, all functioning replicas in the corresponding ring are paused until an agreement about the evolving ring and current decision is reached. In the *nonblocking* mode, a replica independently handles the failures as they occur. However, the benefit of nonblocking comes at the expense of introducing decision conflicts because a new decision might be made before the current decision is resolved. Hence, the challenges in the nonblocking mode include determining how to handle decision conflicts and how to maintain consistency in the decisions of the replicas in the presence of decision conflicts.

In the nonblocking mode, a client request is first broadcast. Then a message containing the decision is broadcast in terms of associating the request with a total-order value. A *decision* is an action that associates a client request with a total-order value. Due to decision conflicts, replicas need to confirm the decision locally. A *decision confirmation* is a commitment on the relationship between the request and the total-order value in the decision. A decision is not final before its confirmation. In the system's viewpoint, if all functioning replicas confirm the same decision, an agreement

*This work was supported in part by NSF grants ANI 0073736, EIA 0130806, CCR 0329741, CNS 0422762, CNS 0434533, and CNS 0531410 in USA, and NSFC 60273038, NCET-04-0478 by MOE, and Jiangsu STT Project in China. This work of Yun Wang was completed when visiting FAU.

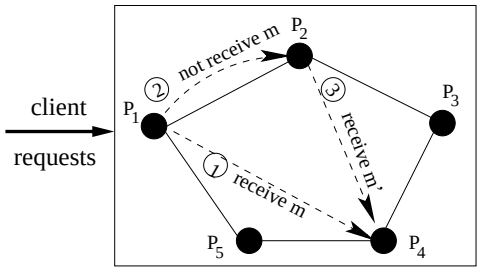


Figure 1. The conflicting decisions of m: $\langle\ p_1$, r_5, $1\ \rangle$ and m': $\langle\ p_2$, r_1, $1\rangle$. The order refers to the event sequence.

on the decision is reached. No explicit agreement operation exists in the nonblocking mode.

Regarding the scenario illustrated in Figure 1, 5 replicas $p_1, \cdots, p_5$ in sequence, form a logical token ring. 10 client requests of $r_1, \cdots, r_{10}$ are broadcast. Assume that replicas receive the client requests in arbitrary orders. The current token-holding replica can make a decision, usually represented by a tuple $\langle$*replicaID, requestID, TONum*$\rangle$, where the parameters stand for decision-making replica, client request, and total-order value, respectively. For simplicity, we assume that one decision is associated with only one client request. The replica randomly chooses a received and unordered client request and computes the total-order value in its decision (Section 2.2). Suppose p_1 currently holds the token, and broadcasts its decision m of $\langle p_1, r_5, 1\rangle$, but p_1's immediate successor p_2 does not receive m due to p_1's failure. p_2 will observe a timeout exception, and regard this as a token loss. Hence, it generates a new token independently. Since p_2 does not receive m, it still associates the total-order value 1 in its decision m' of $\langle p_2, r_1, 1\rangle$ and broadcasts m'. m and m' are in conflict because they break the one-to-one correspondence between the client requests and the total-order values.

In this paper, we propose a forward-confirmation (FC) approach to resolve decision conflicts in the nonblocking mode. Using the FC approach, a replica locally confirms a decision made by p_i based on a subsequent decision made by $p_i's$ successor, namely the first conflict-free decision (Section 2.3). The first subsequent conflict-free decision increases the total-order value by one correct decision among the conflicting decisions. The FC approach needs to (1) identify a decision conflict; (2) determine the number of decisions in conflict; and (3) confirm one decision among the conflicting decisions. It is difficult to guarantee consistency among replicas for the following two reasons. First, a replica may locally receive parts of some conflicting decisions due to failures. In Figure 1, p_4 finds m and m' in conflict, but p_2 does not notice it. Second, it is an independent operation when a replica confirms a decision. In the example, p_2 and p_4 confirm m or m' independently. We prove that all functioning replicas confirm the same decisions using the FC approach, and based on the confirmed decisions, a consistent total order of client requests is obtained.

In our previous work [22], we proposed a nonblocking total ordering protocol under the assumption that no two successive replicas fail simultaneously. The assumption is relatively strong in practice. We relax this in the FC approach and address a more general problem: replicas reach an agreement on the total-order values of client requests when up to $k(0 \leq k \leq \lfloor n/2 \rfloor - 1)$ successive replicas fail simultaneously. Note that the FC approach does not affect the other parts of the protocol presented in [22], including token, request, and membership management. Regarding the issues of handling multiple tokens, ring (re)construction and group membership, please refer to [22].

Our contributions in this paper can be summarized as follows: (1) we introduce the concept of decision conflicts into total ordering protocols in a nonblocking mode; (2) we present the FC approach to handle decision conflicts when up to k successive replicas fail simultaneously in a ring and yet are still regarded as eligible to satisfy consistency; (3) we propose an FC implementation, and performance results show that the nonblocking mode helps to speed up system execution.

The remainder of the paper is organized as follows. Section 2 describes the model. The FC approach and its correctness proof are presented in Section 3. An implementation and performance analysis are given in Section 4. Section 5 reviews the related work. Section 6 concludes this paper.

2 Model

2.1 The Model and Assumptions

The distributed system under study comprises n replicas of $p_1, p_2, \cdots, p_n$. Each replica resides on a different computer. Replicas communicate with each other only via message exchange. There is no upper bound of time for message transmission or action execution.

Replicas form a logical token ring. Normally, only one token circulates in the ring in an unidirectional way. Only the current token holder can make a decision. After the current token holder broadcasts its decision, it automatically releases the token. Its immediate successor obtains the token if the message containing the decision is received. To simplify the discussion, we assume the following: (1) the replicas are fully-connected although we require that the replicas are inter-reachable; (2) each time, a replica can only send one message; (3) a message can contain only one decision; (4) a decision can randomly associate at most one

received and unordered client request. Initially, replicas begin to function, and have the same ring view, which covers all the replicas, and the same ring version, which is 1. Each replica knows its position in the ring. Usually, the replica with the smallest ID initially holds the token.

In our model, we assume that replicas receive the same set of client requests in arbitrary orders. If a replica does not receive a message from its immediate predecessor, the immediate predecessor can be regarded as a failure regardless of the underlying reasons, such as replica crash, message loss, or a slow link. We treat this as replica failure. We assume that at least $\lfloor n/2 \rfloor + 1$ replicas do not fail. A system can mask up to k ($k \leq \lfloor n/2 \rfloor - 1$) successive replicas' failures simultaneously. We do not consider network partition in the paper. Please refer to [13] for various solutions regarding the topic. Replicas conform to the fail-stop model, i.e., a failed replica will not take part in computation.

Each replica is equipped with a queue and maintains a waiting timer. Each replica is also equipped with an FC implementation and runs the FC approach locally.

2.2 Decision and Confirmation

A decision is an action to associate a client request with a total-order value. In a fault-free environment, a *decision* is represented by ⟨*replicaID, requestID, TONum*⟩, where the parameters stand for decision-making replica, client request and total-order value, respectively. Each time, a replica can only make one decision.

The current token holder always makes a decision even if no client requests are available. The reason for this is that the current token holder may not have unordered client requests, but it is still necessary to reflect the ring's operational status via token circulation. Thusly, the $requestID$ is set to $NULL$.

In faulty environments, the failed replica should be removed from the ring. This incurs a ring evolution. We should include this information in the decision. We use a ring version and a ring view to represent the ring evolution information. Ring version is an assigned value that is increased by 1 once a ring evolution occurs. The ring view is the set of current replicas. Hence, a decision can be represented by ⟨*replicaID, requestID, TONum, rversion, rview*⟩, where $rversion$ and $rview$ stand for ring version and ring view, respectively.

Due to decision conflicts (Section 2.3), a decision is not final before its confirmation. A decision confirmation is a commitment of the relationship between the request and the total-order value in the decision. Once a decision is confirmed by all functioning replicas using the FC approach, the order of the client request in the decision is agreed upon. Once the orders of all client requests are agreed upon, the total-ordered sequence of client requests is formed by sorting client requests in ascending order of their associated total-order values. Due to space limitation, we can only briefly specify the method we use to set a total-order value in a decision.

Usually, the total-order value starts at 1. Therefore, the very first decision in a ring includes the total-order value 1. Thereafter, the current token holder sets a total-order value of 1 larger than the maximum total-order value in decisions it has received. Thusly, the total-order values in the decisions made by successive replicas are in ascending order.

We now consider the example in Figure 1 in a fault-free environment. Suppose p_1 is the current token holder. It makes a decision of ⟨p_1, r_4, 1⟩ and broadcasts it, and all of the replicas receive it. Hence, p_1's immediate successive replica p_2 automatically obtains the token. Because the maximum total-order value in decisions p_2 receives is 1, p_2 sets 2 to $TONum$ in its decision. A decision of ⟨p_2, r_3, 2⟩ is made and broadcast. This process continues until 10 client requests associate total-order values in the decisions. Each replica receives the decisions, confirms the decisions, and sorts the client requests according to their associated total-order values in ascending order. Therefore, replicas locally obtain a total-ordered sequence of client requests beginning with r_4 and r_3 and so on, no matter what the receiving orders of the client requests are. The local sequence of client requests on each replica is consistent among replicas.

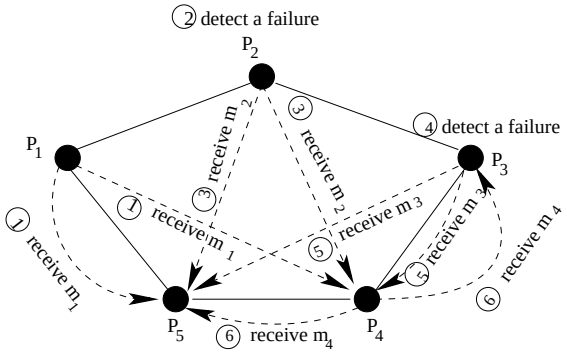
2.3 Conflicted Decision

In a token ring, the current token holder p_i normally broadcasts its decision and automatically releases the token by sending a message containing the decision. Once p_i releases the token, it sets a timer locally. It expects to re-obtain the token before the timer expires. p_{i+1} receives the decision made by p_i and automatically obtains the token. However, if p_{i+1} does not receive the message containing the decision and notices a local timeout exception, p_{i+1} regards it as a token loss and generates a new token. In the nonblocking mode, decision conflicts may occur. As shown in Figure 1, m and m' are conflicting decisions that mistakenly share the same total-order value 1 in different decisions.

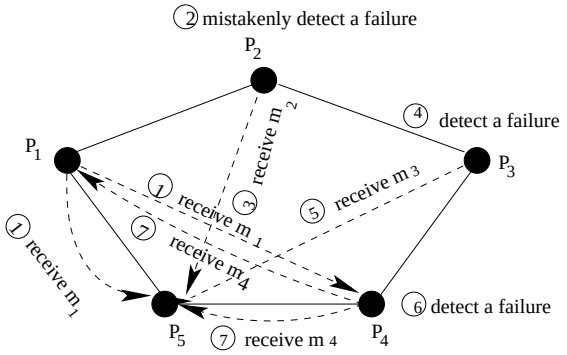
Definition 1. ***(Conflict)*** *For any decisions m and m', m' and m are conflicting decisions if and only if the following expression holds:*

$$m.replicaID \neq m'.replicaID \wedge m.TONum = m'.TONum.$$

In order to maintain the total order property, only one of the conflicting decisions could be confirmed by replicas using the FC approach discussed in Section 3.



(a) The conflicting decisions of m_1, m_2 and m_3 when p_1 and p_2 fail. m_1 is $\langle p_1, r_7, 1, 1, \{p_1, p_2, p_3, p_4, p_5\}\rangle$. m_2 is $\langle p_2, r_1, 1, 2, \{p_2, p_3, p_4, p_5\}\rangle$. m_3 is $\langle p_3, r_3, 1, 2, \{p_3, p_4, p_5\}\rangle$. m_4 is $\langle p_4, r_6, 2, 2, NULL\rangle$.



(b) The conflicting decisions of m_1, m_2 and m_3 when p_2 and p_3 fail. m_1, m_2 and m_3 are the same as in (a). m_4 is $\langle p_4, r_6, 2, 2, \{p_1, p_4, p_5\}\rangle$.

Figure 2. The scenarios of conflicting decisions when $k = 2$.

Definition 2. ***(Conflict-freedom)*** *For any decisions m and m', m' is a conflict-free decision to m if and only if the following expression holds:*

$$m.replicaID \neq m'.replicaID \wedge m'.TONum > m.TONum.$$

Consider the scenarios of conflicting decisions when $k = 2$ in Figure 2. Suppose that p_1 and p_2 fail in Figure 2(a). When p_1 holds the token, it makes a decision m_1 of $\langle p_1, r_7, 1, 1, \{p_1, p_2, p_3, p_4, p_5\}\rangle$ and broadcasts m_1. p_2 does not receive m_1; it observes a timeout exception, and regards it as a token loss. p_2 generates a new token, and regards its immediate predecessor as a failure. Therefore, the ring evolves. p_2 makes a decision m_2 of $\langle p_2, r_1, 1, 2, \{p_2, p_3, p_4, p_5\}\rangle$ and broadcasts m_2. Again, p_3 does not receive m_2; it finds a timeout exception, and regards it as a token loss. p_3 generates a new token. It does not receive m_1, either. Thusly, p_3 still sets $TONum$ to 1, and makes a decision m_3 of $\langle p_3, r_3, 1, 2, \{p_3, p_4, p_5\}\rangle$. m_1, m_2 and m_3 are conflicting decisions. As shown in Figure 2(b), if p_2 and p_3 fail, and if the failing replica p_2 mistakenly regards p_1 as failed, m_1, m_2 and m_3, which are made by p_1, p_2 and p_3 respectively, are conflicting decisions, too. We observe that in these two scenarios, m_4 is almost the same except for its ring view.

3 The FC Approach

3.1 The Overview

In order to achieve the system objective, i.e., replicas reach an agreement on the total-order values of client requests when up to k successive replicas fail simultaneously, the FC approach is proposed. In the nonblocking mode, there is no explicit agreement operation for a decision. Replicas confirm the decisions independently and do not exchange such confirmation results.

The main idea of the FC approach is to let a replica confirm a decision by the decision's first subsequent conflict-free decision. Consider that a replica tries to confirm m_1. It finds that a decision cfd is the first subsequent conflict-free decision to m_1. Because cfd is conflict-free to m_1, $cfd.TONum > m_1.TONum$. Further, because cfd is the first subsequent conflict-free decision to m_1, based on the rules to set total-order values in decisions, $cfd.TONum = m_1.TONum+1$. The reason behind progressing the total-order value in cfd is that $cfd.replicaID$ receives at least one of the conflicting decisions. The FC approach needs to trace which conflicting decision is confirmed by $cfd.replicaID$, and let the host replica confirm the same conflicting decision.

In fault-free environments, a replica confirms a decision m made by p_i by the decision m' made by p_i's immediate successor p_{i+1}. This is because p_{i+1} always receives m, and sets $m'.TONum$ to $m.TONum + 1$. Thusly, m' is always the first subsequent conflict-free decision to m. It is more complicated in faulty environments. In Figure 2(a), p_5 tries to confirm one conflicting decision among m_1, m_2 and m_3. Based on the assumption that only p_1 and p_2 fail, p_3 does not fail and hence p_4 receives m_3. Then, p_4 automatically obtains the token. It makes a decision m_4 of $\langle p_4, r_6, 2, 2, NULL\rangle$. m_4 is the first subsequent conflict-free decision to m_1. p_5 receives m_4, and using the FC approach, it confirms m_3. A tricky situation is that although $k = 2$ holds in both Figure 2(a) and Figure 2(b), and m_1, m_2 and m_3 are conflicting decisions, p_5 confirms m_3 in Figure 2(a) while confirming m_1 in Figure 2(b) (details are discussed in Section 3.2).

The function of the FC approach has two tasks. In the first task, a replica checks every message containing a decision it receives until it finds the first subsequent conflict-free decision cfd. The replica stores all conflicting deci-

Algorithm 1 FC

Require: m_1: $\langle p_i, req, val, ringver, ringview \rangle$.
Ensure: A decision is confirmed.
1: Initialization: $queue \leftarrow NULL$; $queue[1] \leftarrow m_1$; $z \leftarrow 1$; $cfd \leftarrow NULL$;
Task 1: (Find cfd for m_1)
2: **while** (True) **do**
3: Receive a decision m made by p_i's j^{th} ($j \leq (k+1)$) successor
4: **if** ($m.TONum$ is val) **then**
5: $queue[j+1] \leftarrow m$
6: **else if** ($queue[j].TONum$ is val and $m.TONum$ is (val+1)) **then**
7: $cfd \leftarrow m$; $z \leftarrow j$; break;
8: **end if**
9: **end while**
Task 2: (Confirm a decision in $queue$ based on cfd)
10: **if** (z is 1) **then**
11: Confirm $queue[1]$
12: **else if** ($cfd.rview$ is $NULL$) **then**
13: Confirm $queue[z]$; Discard $queue[1]$ to $queue[z-1]$;
14: **else**
15: Confirm $queue[1]$; Discard $queue[2]$ to $queue[z]$;
16: **end if**

sions it observes during the processing. In the second task, the replica confirms one conflicting decision based on cfd by deduction. The FC approach guarantees that (1) for a set of conflicting decisions, only one decision is confirmed by each replica independently; and (2) replicas confirm the same decision even if some replicas suffer from conflicts while the others do not. Regarding confirmed decisions, a one-to-one correspondence is set up between client requests and total-order values.

3.2 The FC Algorithm

In this section, we first discuss the FC algorithm (Algorithm 3.1) in general. Then we describe the algorithm in two cases, i.e., when exactly k successive replicas fail simultaneously and when up to k successive replicas fail simultaneously. Suppose that a decision m_1 of $\langle p_i, req, val, ringver, ringview \rangle$ is under consideration. The parameters in m_1 mean that replica p_i associates a total-order value val to a client request req when the ring version is $ringver$ and the ring view is $ringview$. cfd is the first subsequent conflict-free decision to m_1. In the algorithm, we use two additional variables. One is $queue$, which is a variable-length array to temporarily store m_1's conflicting decisions. The other is z. It is an integer, and is used for recording the index in the queue of the conflicting decision which is exactly before cfd.

In the initialization, $queue$ is emptied. m_1 is inserted into $queue$ with the index of 1. z is set to 1. cfd is $NULL$.

Task 1: Find cfd for m_1 (lines 2-9).

A replica checks every message containing a decision m it receives. If m is made by p_i's j^{th} successor and $j \leq (k+1)$, the replica goes on to further check on m. Otherwise, the replica just ignores m because m is not related to m_1's confirmation. If $m.TONum$ equals val, according to the definition of conflict, m and m_1 are conflicting decisions. m is inserted to $queue$ with the index of $j+1$. Otherwise, if $m.TONum$ equals $val+1$, and further if the decision just before m is a conflicting decision to m_1, m is the first subsequent conflict-free decision to m_1. Therefore, cfd is m made by p_{i+z}, and z is set to j.

Task 2: Confirm a decision in $queue$ based on cfd (lines 10-16).

If $z = 1$, it means only that m_1 is in $queue$, and that m_2 is the cfd. There is no decision conflict. Hence, m_1 is confirmed. Otherwise, the further processing depends on $cfd.rview$.

1. $cfd.rview$ is $NULL$. This means that p_{i+z} regards its immediate predecessor p_{i+z-1} as normal because p_{i+z} receives m_z, and no ring evolution occurs. Therefore, the $z-1$ successive replicas from p_i to p_{i+z-2} fail. Only m_z is confirmed, and all the other conflicting decisions, namely $m_1, \cdots, m_{z-1}$, which are in $queue$ with the index from 1 to $z-1$, are discarded.

2. $cfd.rview$ is not $NULL$. This means that p_{i+z} regards its immediate predecessor p_{i+z-1} as a failure. It generates a new token and makes a new decision including the evolving ring information. Because simultaneously failed replicas are successive, and only one of the replicas making conflicting decisions succeeds, the only possible situation is that the replica making the first conflicting decision m_1 does not fail. $z-1$ successive replicas from p_{i+1} to p_{i+z-1} fail. p_i does not fail. p_{i+z} receives m_1. Therefore, only m_1 is confirmed and all the other conflicting decisions, namely $m_2, \cdots, m_z$, which are in $queue$ with the index from 2 to z, are discarded.

We describe the FC algorithm in the following cases.

Exactly k Successive Replicas Fail Simultaneously

In Task 1, m_{k+2}, made by p_i's $(k+1)^{th}$ successive successor, is cfd .

Like the discussion in Section 3.1, if $k = 0$, cfd is m_2 made by p_{i+1}. Now $k > 0$ is under consideration. If p_{i+1} does not receive m_1, consequently, p_{i+1} generates a new token and makes a new decision m_2 of $\langle p_{i+1}, req_j, val, ringver+1, ringview-\{p_i\} \rangle$. Then p_{i+1} broadcasts m_2 and fails. p_{i+2} does not receive m_1 or m_2. It then assumes there has been a token loss and generates a new token. Further, it makes a new decision m_3 of $\langle p_{i+2}, req_s, val, ringver+1,$

$ringview$-$\{p_i, p_{i+1}\}\rangle$. It broadcasts m_3 and fails. All k decisions made by p_i's k successive successors associate the same total-order value val. The k decisions, i.e., m_2, $\cdots$, m_{k+1}, are conflicting decisions to m_1. The k decisions plus m_1 are in conflict. Furthermore, p_{i+k+1}, p_i's $(k+1)^{th}$ successive successor, does not view all of its $(k+1)$ successive predecessor's failures simultaneously because exactly k successive replicas fail simultaneously. It receives at least one of the conflicting decisions. Hence, m_{k+2} associates a total-order value $val+1$. Therefore, cfd is m_{k+2}.

A replica may receive parts of the conflicting decisions because some of the conflicting decisions may be lost, or some of the failed replicas do not make any decisions before they fail. No matter which case, m_{k+2} is always necessary.

In Task 2, a replica confirms one decision among the conflicting decisions based on m_{k+2}. If $k=0$, no decision conflict exists. m_1 is confirmed. Otherwise, if $m_{k+2}.rview$ is $NULL$, only m_{k+1} is confirmed. If $m_{k+2}.rview$ is not $NULL$, only m_1 is confirmed.

We now reconsider the example when $k=2$ in Figure 2. In the situation of Figure 2(a), p_1 and p_2 fail simultaneously. p_4 and p_5 try to confirm m_1 locally. p_4 and p_5 receive m_1, m_2 and m_3. According to Task 1, p_4 and p_5 know that cfd is a decision made by p_1's 3^{rd} successor p_4. p_4 receives m_3 and automatically obtains the token, then makes a decision m_4 of $\langle p_4, r_6, 2, 2, NULL\rangle$. Therefore, cfd is m_4. According to Task 2, because $m_4.rview$ is $NULL$, p_4 and p_5 confirm m_3 and discard both m_1 and m_2. In the situation of Figure 2(b), p_4 and p_5 try to confirm m_1, too. p_5 receives conflicting decisions m_1, m_2 and m_3. p_4 receives m_1. Both of the replicas know that cfd is m_4 and wait for it. p_4 does not receive m_3, observes a timeout exception, and generates a new token. Because p_4 receives m_1 with $m_1.TONum=1$, p_4 sets 2 to $m_4.TONum$. Because p_4 regards p_3 as a failure and $k=2$, the only possible situation is that p_3 and p_2 fail simultaneously. Hence, the ring evolves. p_4 makes a decision m_4 of $\langle p_4, r_6, 2, 2, \{p_1, p_4, p_5\}\rangle$. p_4 and p_5 receive m_4. Again according to Task 2, because $m_4.rview$ is not $NULL$, p_4 and p_5 confirm m_1 and discard m_2 and m_3 if received. So, in the two scenarios when $k=2$, m_3 is confirmed in Figure 2(a) while m_1 is confirmed in Figure 2(b).

Up To k Successive Replicas Fail Simultaneously

If up to k successive replicas fail simultaneously, m_{k+2} made by p_{i+k+1} is possibly not the first conflict-free decision to m_1 made by p_i. For instance in Figure 2(a), suppose $k=2$ and actually only 1 replica p_1 fails. cfd is m_3 rather than m_4. The first subsequent conflict-free decision appears earlier because a system may suffer from a number of failures less than k.

The processing is exactly the same as that shown in Algorithm 3.1. In Task 1, cfd is m_{z+1}. In Task 2, if $z=1$, no conflict decision exists, and m_1 is confirmed. Otherwise, if $cfd.rview$ is $NULL$, only m_z is confirmed. All the other conflicting decisions, i.e., m_1, $\cdots$, m_{z-1}, are discarded. If $cfd.rview$ is not $NULL$, only m_1 is confirmed. All the other conflicting decisions, i.e., $m_2, \cdots, m_z$, are discarded.

Now we revisit the example in Figure 2. In the situation of Figure 2(a), as we discussed, p_4 and p_5 confirm m_3. For p_3, it receives m_3 as its first decision without receiving m_1 and m_2. Thusly, p_3 tries to confirm m_3 rather than m_1 or m_2 because it does not even know of these decisions' existence. Then p_3 receives m_4. It is the cfd to m_3. There is no decision conflict. According to Task 2, p_3 confirms m_3. Until now, all functioning replicas in Figure 2(a) confirm m_3 among m_1, m_2 and m_3. The processing of m_1's confirmation is similar in Figure 2(b).

As we discussed, the FC approach can determine a definite set of conflicting decisions to any decision m_1, and therefore finds cfd. The FC approach further traces the decision confirmed by $cfd.replicaID$ and allows the host replica to confirm the same decision. All functioning replicas locally confirm the same decision among conflicting decisions using the FC approach, and then they achieve a consistent total order of client requests based on all confirmed decisions.

4 Implementation and Analysis

If up to k successive replicas fail simultaneously, up to $k+1$ decisions are in conflict. These $k+1$ decisions are temporarily stored in a local queue. The queue length represents how many conflicting decisions there are, and also reflects how long a replica has to wait before launching the FC approach. A straightforward way to implement the FC approach is to set up a queue with the maximum length of $k+1$. The FC implementation with the queueing method is presented in this section. Further, the performance issue is analyzed.

4.1 The Queueing Method

Each replica is equipped with a local queue with the maximum length of $k+1$ if up to k successive replicas fail simultaneously. The worst case work flow in the queueing method is described as follows. If p_i tries to confirm m_1, it inserts m_1 to its queue as the first element. p_i possibly receives $k+1$ conflicting decisions to m_1, and inserts them to the queue. As soon as p_i receives the first subsequent conflict-free decision m_{k+2}, it locally launches the FC approach, and confirms either m_1 or m_{k+1} based on m_{k+2}. Then p_i empties the queue, inserts m_{k+2}, and starts to confirm m_{k+2}.

It is not necessary for p_i to use the maximum length of the queue because the number of failed successive replicas may be less than k in an FC processing. Hence, fewer than

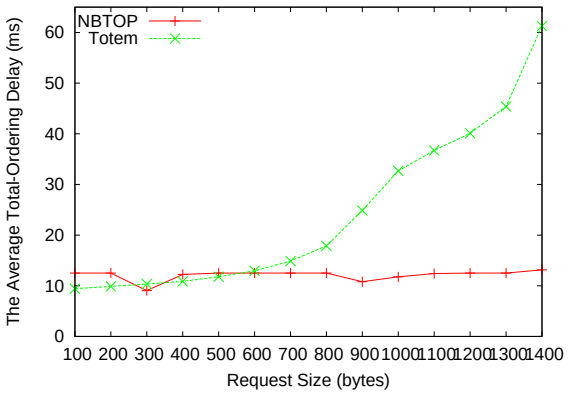


Figure 3. The relationship between request size and request total-ordering delay.

$k + 1$ conflicting decisions are inserted to the queue. The rest of the procedure remains the same.

4.2 Performance Comparison

The FC approach is added to the previous protocol in [22] called NBTOP, which is implemented with Visual C++ 6.0. We compare the performance of NBTOP with the Totem protocol under the same circumstances. A Spread version 3.17.4 of Totem is used which is available at http://www.spread.org. The test environment is composed of 5 PCs (Windows XP, Intel 1.6GHz Pentium 4, 1GB of RAM, connecting to 100Mbps Switched Ethernet). The machines compose a logical ring.

Figure 3 demonstrates the relationship between request size and request total ordering delay. It shows that the average total ordering delay in NBTOP and Totem is quite even - around $12ms$ - if the request size is less than 600 bytes. Regarding the NBTOP, the average total ordering delay remains stable with the growth of the request size. The delay is $13.134ms$ when the request size is 1400 bytes. Comparatively, the average total ordering delay in the Totem protocol goes up drastically. The delay is $61.278ms$ with the request size of 1400 bytes.

Further, we observe that a replica using NBTOP takes, on average, $11ms$ to detect and recover from its immediate predecessor replica failure, and $11.455ms$ from 2 successive replicas' simultaneous failures if the waiting timer is set to 100ms. These are better results than when using the Totem protocol of $40ms$ with the same interval of Token Loss timeout.

5 Related Work

Total-ordering protocols. Several literatures addressed the issue of message total ordering with the logical token ring. In Totem developed by UCSB, a symmetrical fault tolerant protocol based on the logical token ring was implemented, including total ordering protocol with single ring and multiple rings respectively [1], [2]. The functioning replicas had to stay in the *Recovery* state before they were switched to the *Operational* state if some faults occurred and all functioning replicas needed to reach an explicit agreement on their decisions about the ring before any new decisions were made. [12] proposed a total ordering protocol with a logical token ring. A decision needed to wait for ACKs from all the other replicas in the presence of failures. Total order is useful in many applications. [3] proposed a method to sort a directed graph geographically in a total order for processing line segments.

Consensus Problem. [9] gave an impressive result that any protocol for consensus problem in asynchronous distributed systems has the possibility of nontermination, even with only one faulty process. Chandra and Toueg gave a solution to the consensus problem with failure detectors, which at least satisfied $\Diamond S$ (the properties of both eventually strong completeness and eventually weak accuracy) requirements [4]. The communication problem in the consensus problem also attracts much research. [7] presented a survey and classification on total order broadcast and multicast algorithms. [5] showed different kinds of algorithms to achieve group communication. A secure ring group communication protocol was studied in [14]. [8] applied group communication to reach dynamic load balancing. [19] proposed new specifications of dynamic reliable broadcast, dynamic atomic broadcast and group membership.

Nonblocking approaches. There are numerous literatures on the nonblocking property and its applications. [17] presented the protocols solving the nonblocking atomic commitment problem. A nonblocking k-fold multicast network is studied in [23]. A nonblocking checkpointing mode was given in [16] to support concurrency in the execution of state saving and other simulation-specific operations. Nonblocking synchronization in concurrent programming was studied in [20]. The research results showed that nonblocking mode helps improve system effectiveness.

6 Conclusion

This paper presents the FC approach to achieve total-order values of client requests in a nonblocking mode with the logical token ring. It defines the decision conflict introduced by the nonblocking mode. The FC approach is presented in detail to handle decision conflicts when up to k

successive replicas fail simultaneously. A queueing method to implement the FC approach is proposed. Test results show that the nonblocking mode helps to improve the total ordering protocol's performance compared to Totem's. We will further study on how to improve delay performance in a large ring in order to make the system performance better.

References

[1] D. A. Agarwal, L. E. Moser, S. P. Melliar-Smith, and R. K. Budhia. The Totem multiple-ring ordering and topology maintenance protocol. *ACM Transactions on Computer Systems*, 16(2):93–132, 1998.

[2] Y. Amir, L. E. Moser, S. P. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, 1995.

[3] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, 47:1–25, 2007.

[4] J. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[5] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, 2001.

[6] H. Decher, B. L. Irun-Briz, M. R. Juan-Marin, J. E. Armendariz, and F. Munoz-Escoi. Wide-area replication support for global data repositories. In *Proceedings of 16th International Workshop on Database and Expert Systems Applications*, pages 1117–1121, 2005.

[7] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[8] S. Dolev, R. Segala, and A. Shvartsman. Dynamic load balancing with group communication. *Theoretical Computer Science*, 369:348–360, 2006.

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

[10] Object Management Group. Fault tolerant CORBA, common object request broker architecture, v.3.0, 2002.

[11] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, 1997.

[12] W. Jia, J. Cao, T. Cheung, and X. Jia. A multicast protocol based on a single logical ring using a virtual token and logical clocks. *The Computer Journal*, 42(3):402–420, 1999.

[13] J.Wu. *Distributed System Design*. CRC Press, 1998.

[14] K. Kihlstrom, L. E. Moser, and S. P. Melliar-Smith. The secure ring group communication system. *ACM Transactions on Information and System Security*, 4(4):371–406, 2001.

[15] P. Narasimhan, L. E. Moser, and S. P. Melliar-Smith. Lessons learned in building a fault-tolerant CORBA system. In *International Conference on Dependable Systems and Networks*, pages 39–44, 2002.

[16] F. Quaglia and A. Santoro. Nonblocking checkpointing for optimistic parallel simulation: description and an implementation. *IEEE Transactions on Parallel and Distributed systems*, 14(6):593–610, 2003.

[17] M. Raynal. Revisiting the non-blocking atomic commitment problem in distributed systems. In *Proceedings of 2nd IPPS IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 116–133, 1997.

[18] S. Sai and J. Carter. Flexible consistency for wide area peer replication. In *Proceedings of 25th IEEE International Conference on Distributed Computing Systems (ICDCS 2005)*, pages 199 – 208, 2005.

[19] A. Schiper. Dynamic group communication. *Distributed Computing*, 18(5):359–374, 2006.

[20] L. Wang and S. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *Proceedings of 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 61–71, 2005.

[21] Y. Wang, E. Anceaume, F. Brasileiro, F. Greve, and M. Hurfin. Solving the group priority inversion problem in a timed asynchronous system. *IEEE Transactions on Computers*, 51(8):900–915, 2002.

[22] Y. Wang and J. Wang. A non-blocking message total ordering protocol. *Science in China*, 2007. To appear.

[23] Y. Yang and J. Wang. Nonblocking k-fold multicast networks. *IEEE Transactions on Parallel and Distributed systems*, 14(2):131–141, 2003.