A Fault-Tolerant Sequencer for Timed Asynchronous Systems*

Roberto Baldoni, Carlo Marchetti, and Sara Tucci Piergiovanni

Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza",
Via Salaria 113, 00198 Roma, Italia.
{baldoni,marchet,tucci}@dis.uniroma1.it

Abstract. In this paper we present the specification of a sequencer service that allows independent processes to get a sequence number that can be used to label successive operations (e.g. to allow a set of independent and concurrent processes to get a total order on their operations). Moreover, we provide an implementation of the sequencer service in a specific partially synchronous distributed system, namely the timed asynchronous model. As an example, if a sequencer is used by a software replication scheme then we get the advantage to deploy server replicas across an asynchronous distributed system such as the Internet.

1 Introduction

Distributed agreement among processes is one of the fundamental building blocks for the solution of many important problems in asynchronous distributed systems, e.g. mutual exclusion[9] and replication[10,3,8]. As an example, in the context of software replication replicas have to run a distributed agreement protocol in order to maintain replica consistency. In particular, in the case of active replication[10] the agreement problem reduces to the total order multicast problem and in the case of passive replication[3] to the view synchronous multicast problem[8]. In both cases these problems are not solvable in asynchronous distributed systems prone to process crash failures due to FLP impossibility result[7]. As a consequence, to solve these problems replicas have to be deployed over a partially synchronous distributed system i.e., an asynchronous distributed system which enjoys some timing assumption (e.g., known, or eventually known, bounds on message transfer delay and on relative process speeds). Practically, when working on a partially synchronous system, replication techniques can benefit of group communication primitives and services implemented by group toolkits such as total order multicast, view synchronous multicast, group membership, state transfer, etc.. These primitives employ agreement protocols to ensure replica consistency. Let us remark that the partial synchronous system assumption makes impossible the deployment of server replicas in real asynchronous distributed systems such as the Internet.

* Work partially supported by MIUR (DAQUINCIS) and by AleniaMarconiSystems.

B. Monien and R. Feldmann (Eds.): Euro-Par 2002, LNCS 2400, pp. 578–588.

In this paper we present the specification of a sequencer service. The specification ensures (i) that independent client processes obtain distinct sequence numbers for each distinct request submitted to the sequencer and (ii) that the sequence numbers assigned are consecutive, i.e. the sequence of assigned numbers does not present "holes".

A sequencer service can then be exploited, for instance, in the context of active replication. A sequencer implementation can be logically placed between clients and server replicas, and can be used to piggyback sequence numbers onto client requests, before sending them out to server replicas. In this way replicas can independently order client requests without the need of executing distributed agreement protocols. This actually makes possible the deployment of replicas over an asynchronous distributed system (interested readers can refer to [2] for additional details about this sequencer based replication technique) as there is no need of additional timing assumptions on the system model underlying the server replicas.

As second contribution, in this paper we provide a fault-tolerant implementation of the sequencer service. This implementation adopts a passive replication scheme in which the sequencer replicas run over a specific partially synchronous system, namely the timed asynchronous system model[6].

Coming back to the active replication example, the sequencer can then be seen as the component that embeds the partial synchrony necessary to maintain consistency among a set of server replicas. Therefore server replicas do not need to run any distributed agreement protocol. This allows server replica deployment over a real asynchronous distributed system[2].

The remainder of this paper is organized as follows: Section 2 introduces the specification of the sequencer service. Section 3 presents the distributed system model. Section 4 details our implementation of the sequencer. Section 5 concludes the paper. Due to lack of space, a formal proof of the correctness of the implementation can be found in[1].

2 Specification of the Sequencer Service

A sequencer service receives requests from clients and assigns an integer positive sequence number, denoted $\#seq$, to each *distinct* request. Each client request has a unique identifier, denoted req_id, which is a pair $\langle cl\_id, \#cl\_seq \rangle$ where cl_id is the client identifier and $\#cl\_seq$ represents the sequence number of the requests issued by cl_id. As clients implement a simple retransmission mechanism to cope with possible sequencer implementation failures or network delays, the sequencer service maintains a state A composed by a set of assignments $\{a_1, \ldots a_{k-1}, a_k\}$ where each assignment a corresponds to a pair $\langle req\_id, \#seq \rangle$, in which $a.req\_id$ is a request identifier and $a.\#seq$ is the sequence number returned by the sequencer service to client $a.req\_id.cl\_id$. A sequencer service has to satisfy the following properties:

P1. Assignment Validity. If $a \in A$ then there exists a client c that issued a request identified by req_id and $req\_id = a.req\_id$.

P2. Response Validity. If a client c delivers a reply $\#seq$, then $\exists a = \langle req\_id, \#seq \rangle \in A$.
P3. Bijection. $\forall a_i, a_j \in A : a_i.\#seq \neq a_j.\#seq \Leftrightarrow a_i.req\_id \neq a_j.req\_id$
P4. Consecutiveness. $\forall a_i \in A : (a_i.\#seq \geq 1) \wedge (a_i.\#seq > 1 \Rightarrow \exists a_j : a_j.\#seq = a_i.\#seq - 1)$
P5. Termination. If a client c issues a request, then, unless the client crashes, it eventually delivers a reply.

P1 expresses that the state of the sequencer does not contain "spurious" assignments. P2 states that the client cannot deliver a sequence number that has not already been assigned by the sequencer. The predicate "P1 and P2" implies that each client, delivering a sequence number, has previously issued a request. P3 states that there is a one-to-one correspondence between the set of req_id and the set A. P4 says that the sequence, starting from one, of numbers assigned by the sequencer has not "holes". P5 states that the service is live.

3 System Model

We consider a distributed system in which processes communicate by message passing. Processes can be of two types: clients and replicas. The latter form a set $\{r_1, \ldots, r_n\}$ of processes implementing the fault-tolerant sequencer. A client c communicates with replicas using *reliable asynchronous channels*. Replicas run over a *timed asynchronous model*[6].

Client Processes. A client process sends a request to the sequencer service and then waits for a sequence number. A client copes with replica failures using a simple retransmission mechanism. A client may fail by crashing.

Communication between clients and replicas is *asynchronous* and *reliable*. This communication is modelled by the following primitives: **A-send**(m, p) to send an unicast message m to process p; and **A-deliver**(m, p) to deliver a message m sent by process p.

To label a generic event with a sequence number, a client invokes the GetSeq() method. Such method blocks the client process until it receives an integer sequence number from a sequencer replica. In particular, the GetSeq() method assigns to the ongoing request a unique request identifier $req\_id = \langle cl\_id, \#cl\_seq \rangle$, then (i) it sends the request to a replica and (ii) sets a local timeout. Then, a result is returned by GetSeq() if the client receives a sequence number for the req_id request within the timeout expiration. Otherwise, another replica is selected (e.g. using a cyclic selection policy), and the request is sent again to the selected replica setting the relative timeout, until a reply is eventually delivered.
Replica Processes. Replicas have access to local hardware clock (which is not synchronized). Timeouts are defined for message transmission and scheduling delays. A performance failure occurs when an experienced delay is greater than the associated time-out. Replicas can also fail by crashing. A process is *timely* in a time interval $[s, t]$ iff during $[s, t]$ it neither crashes nor suffers a performance

failure. For simplicity, a process that fails by crashing cannot recover. A message whose transmission delay is lesser than the associated time-out is *timely*. A subset of replicas form a *stable* partition in $[s,t]$ if any pair of replicas belonging to the subset is timely and each message exchanged between the pair in $[s,t]$ is timely. Timed asynchronous communications are achieved through a *datagram service* which filters out non-timely messages to the above layer. Replicas communicate among them through the following primitives: **TA-send**(m, r_i) to send an unicast message m to process r_i; **TA-broadcast**(m) to broadcast m to all replicas including the sender of m; **TA-deliver**(m, r_j) is the upcall initiated by the datagram service to deliver a *timely* message m sent by process r_j.

We assume replicas implement the leader election service specified in[5]. This service ensures that: (i) at every physical time there exists at most one *leader*, a *leader* is a replica in which the *Leader?()* boolean function returns *true*; (ii) the leader election protocol underlying the *Leader?()* boolean function takes at least 2δ for a leader change; (iii) when a majority of replicas forms a stable partition in a time interval $[t, t + \Delta t]$ $(\Delta t \gg 2\delta)$, then it exists a replica r_i belonging to that majority that becomes leader in $[t, t + \Delta t]$.

Note that the leader election service cannot guarantee that when a replica becomes leader it stays within the stable partition for the duration of its leadership (e.g. the leader could crash or send non-timely messages to other replicas).

In order to cope with *asynchronous interactions* between clients and replicas, to ensure the *liveness* of our sequencer protocol, we introduce the following assumption, i.e.:

eventual global stabilization: there exists a time[1] t and a set $\mathcal{S} \subseteq \{r_1, ...r_n\}$: $|\mathcal{S}| \geq \lceil \frac{n+1}{2} \rceil$ such that $\forall t' \geq t$, $\mathcal{S}$ is a *stable* partition.

The eventual global stabilization assumption implies (i) only a minority of replicas can crash[2] and (ii) there will eventually exist a leader replica $l_s \in \mathcal{S}$.

4 The Sequencer Protocol

In this section we present a fault-tolerant implementation of the sequencer service. A primary-backup replication scheme is adopted [3,8]. In this scheme a particular replica, the *primary*, handles all the requests coming from clients. Other replicas are called *backups*. When a primary receives a client request, it processes the request, updates the backups and then sends the reply to the client. In our implementation the backup update lies on an update primitive (denoted WriteMaj()) that successfully returns if it *timely* updates at least a *majority* of replicas. This implies that inconsistencies can arise in some replica state.

If the primary fails, then the election of a new primary is needed. The primary election lies on: (i) the availability of the leader election service running among replicas (see Section 3). Leadership is a necessary condition to become primary

[1] Time t is not a priori known.

[2] Note that at any given time t' (with $t' < t$) any number of replicas can simultaneously suffer a performance failure.

and then to stay as the primary; (ii) a "reconciliation" procedure, namely "*computing_sequencer_state*" procedure, (*css* in the rest of the paper) that allows a newly elected leader to remove possible inconsistencies from its state before becoming a primary. These inconsistencies, if kept in the primary state, could violate the properties defined in Section 2. Hence a newly elected leader, before becoming a primary, will read at least a majority of replica states, (this is done by a ReadMaj() primitive during the *css* procedure). This allows a leader to have in its state all successfully updates done by previous primaries. Then the leader removes from its state all possible inconsistencies caused by unsuccessful primary updates.

4.1 Protocol Data Structures

A replica r_i endows: (1) a boolean variable called *primary*, which is set according to the role (either primary or backup) played by the replica at a given time; (2) an integer variable called *seq*, used to assign sequence numbers when r_i acts as a primary; (3) a *state* consisting of a pair $\langle TA, epoch \rangle$ where TA is a set $\{ta_1, ...ta_k\}$ of *tentative assignments* and *epoch* is an integer variable. *state.epoch* represents a value associated with the last primary seen by r_i. When r_i becomes primary, *state.epoch* is greater than any *epoch* value associated with previous primaries. *state.epoch* is set when a replica becomes primary and it does not change during all the time a replica is the primary. A *tentative assignment ta* is a triple $\langle req\_id, \#seq, \#epoch \rangle$ where $ta.\#seq$ is the sequence number assigned to the request $ta.req\_id$ and $ta.\#epoch$ is the epoch of the primary that executed ta [3].

The set $state.TA$ is ordered by $TA.\#seq$ field and ties are broken using $TA.\#epoch$ field. We introduce $last(state.TA)$ operation that returns the tentative assignment with greatest epoch number among those (if any) with greatest sequence number. If $state.TA$ is empty, then $last(state.TA)$ returns *null*.

4.2 Basic Primitives and Definitions

In this section we present the basic primitives used to update and to read replica states. Due to lack of space the pseudo-code of such primitives can be found in[1].
UpdateMaj(). Accepts as input parameter m, which can be either a tentative assignment ta or an epoch e, and returns as output parameter a boolean value b. Upon invocation, WriteMaj() executes **TA-broadcast**(m). Every replica r_i sends an acknowledgement upon the delivery of m. The method returns $\top$ (i.e., it *successfully* returns) if (i) the invoker receives at least a majority of *timely* acknowledgements, hence m is put into the replica state according to its type, and (ii) if the invoker is still the leader at the end of the invocation.

[3] Epoch numbers are handled by primaries to label their tentative assignments and by leaders to remove inconsistencies during the *css* procedure.

ReadMaj(). Does not take input parameters and returns as output parameter a pair $\langle b, maj\_state \rangle$ where b is a boolean value and maj_state is a state as defined in Section 4.1. Upon invocation, ReadMaj() executes a **TA-broadcast**. Every replica r_i sends its state as reply. If ReadMaj() receives at least a majority of *timely* replies, then it computes the union $maj\_state.TA$ of the tentative assignments contained in the just received states and sets $maj\_state.epoch$ to the maximum among the epochs contained in the just received states. If the invoker is still leader it returns $\langle \top, maj\_state \rangle$, otherwise $\langle \bot, - \rangle$.

Definitive Assignment: a tentative assignment ta is a *definitive* assignment *iff* exists a primary p such that p executed WriteMaj(ta)$= \top$.
Non-definitive Assignment: a tentative assignment which is not definitive.
Therefore, a definitive assignment is a tentative one. The viceversa is not necessarily true. Non-definitive assignments are actually inconsistencies due to unsuccessful WriteMaj() executions.

4.3 Protocol Description

Let us present in this section a preliminary explanation of the sequencer protocol and two introductory examples before showing the replica pseudo-code.
Primary Failure-Free Behaviour. A primary upon receiving a client request first checks if a sequence number was already assigned to the request, otherwise (i) it creates a new tentative assignment ta embedding the request identifier and a sequence number consecutive to the one associated with the last request, (ii) invokes WriteMaj(ta) to update the backups and (iii) if WriteMaj(ta) successfully returns, it sends back the sequence number to the client as ta is a definitive assignment.
Change of Primary. There are three events that cause a primary replica r_i to lose the primaryship: (i) r_i fails by crashing or (ii) WriteMaj(ta) returns $\bot$ (WriteMaj(ta) *could* have notified ta to less than a majority of replicas) or (iii) there is a leadership loss of r_i (i.e., the *Leader*?() value becomes false in r_i). If any of these events occurs, the protocol waits that a new leader is elected by the underlying leader election service. Then the *css* procedure is executed by the new leader before starting serving requests as primary.
The css *Procedure.* The first action performed by a newly elected leader r_i is to invoke ReadMaj(). If ReadMaj() returns $\langle \bot, - \rangle$ and r_i is always the leader, r_i will execute again ReadMaj(). If r_i is no longer leader, the following leader will execute ReadMaj() till this primitive will be successfully executed.

Once the union of the states of a majority of replicas, denoted maj_state, has been fetched by ReadMaj(), the *css* procedure has three main goals. The first goal is to transform the tentative assignment $last(maj\_state.TA)$ in a definitive assignment *on behalf of a previous primary* that issued WriteMaj($last(maj\_state.TA)$). There is no way in fact for r_i to know if that WriteMaj() was successfully executed by the previous primary. The second goal is to remove from $maj\_state.TA$ all non-definitive assignments. Non-definitive assignments are filtered out using the epoch field of tentative assign-

ments. More specifically, the implementation enforces bijection (Section 2) guaranteeing that when *there are multiple assignments with the same sequence number, the one with the greatest epoch number is a definitive assignment.* The third goal is to impose a primary epoch number e by using WRITEMAJ(). e is greater than the one returned by READMAJ() in *maj_state.epoch* and greater than all epoch numbers associated to previous primaries. If r_i successfully executed all previous points it starts serving requests as primary.

In the following we introduce two examples which point out how the previous actions removes inconsistencies from a primary state.

Example 1: Avoiding inconsistencies by redoing the last tentative assignment. The example is shown in Fig.1. Primary r_1 accepts a client request req_id_1, creates a tentative assignment $ta_1 = \langle req\_id_1, 1, 1\rangle$, performs WRITEMAJ($ta_1$)$= \top$ (i.e. ta_1 is a definitive assignment) and sends the result $\langle 1, req\_id_1\rangle$ to the client. Then r_1 receives a new request req_id_2, invokes WRITEMAJ($ta_2 = \langle req\_id_2, 2, 1\rangle$) and crashes during the invocation. Before crashing it updated only r_3. The next leader r_2 enters the *css* procedure: READMAJ() returns in $maj\_state.TA$ the union of r_2 and r_3 states (i.e., $\{ta_1, ta_2\}$) and in $maj\_state.epoch$ the epoch of the previous primary r_1 (i.e., 1). Therefore, as $last(maj\_state.TA)$ returns ta_2, r_2 executes WRITEMAJ(ta_2)$= \top$ on behalf of the previous primary (r_2 cannot know if ta_2 is definitive or not). Then r_2 executes WRITEMAJ($maj\_state.epoch + 1$) and it ends the *css* procedure. When r_2 receives the req_id_2, it finds ta_2 in its state then sends $\langle 2, req\_id_2\rangle$ to the client.

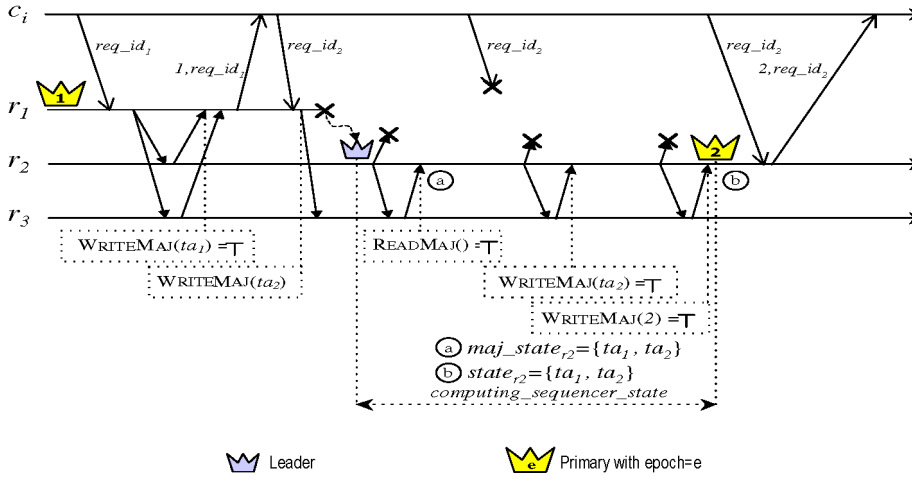

Fig. 1. Example of a Run of the Sequencer Protocol

Example 2: Avoiding inconsistencies by filtering out non-definitive assignments. The example is shown in Fig. 2. Primary r_1 successfully serves req_id_1. Then, upon the arrival of req_id_2, it invokes WRITEMAJ(), exhibits a performance failure and updates only replica r_3 (ta_2 is a non-definitive assignment). Then r_1 loses its primaryship and another leader (r_2) is elected. r_2 executes READMAJ() which

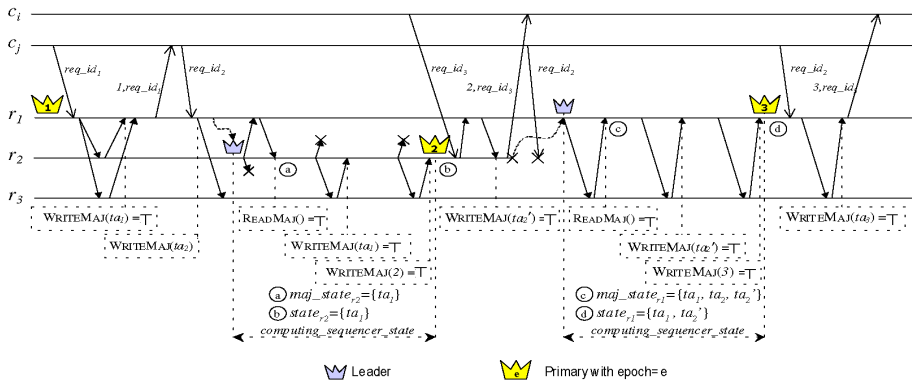

Fig. 2. Example of a Run of the Sequencer Protocol

returns in maj_state the union of r_1 and r_2 states (i.e., $\{ta_1\}$). Then r_2 executes WRITEMAJ(ta_1)$= \top$ and imposes its epoch. Upon the arrival of a new request req_id_3, primary r_2 successfully executes WRITEMAJ($ta'_2 = \langle req\_id_3, 2, 2\rangle$) (i.e. ta'_2 is a definitive assignment) and sends back the result $\langle 2, req\_id_3\rangle$ to the client.

Note that r_1 and r_3 contain two distinct assignments (ta_2 and ta'_2) with same sequence number and different epoch numbers ($ta_2.\#epoch = 1$ and $ta'_2.\#epoch = 2$). However the $maj\_state.TA$ of a successive leader r_i (r_1 in Figure 2) includes the definitive assignment ta'_2 (as it contained in a majority of replicas). If ta_2 is also a member of $maj\_state.TA$, r_i is able to filter ta_2 out from $maj\_state.TA$ as $ta_2.\#epoch = 1 < ta'_2.\#epoch = 2$. After filtering, the state of the primary r_1 is composed only by definitive assignments. Note that without performing such filtering the bijection would result violated, as the state of a primary could contain two assignments with same sequence number.

Then, when r_1 receives the request req_id_2 it performs WRITEMAJ($ta_3 = \langle req\_id_2, 3, 3\rangle$) and if it successfully returns, r_1 sends $\langle 3, req\_id_2\rangle$ to the client.

4.4 Behaviour of Each Replica

The protocol executed by r_i consists in an infinite loop where three types of events can occur (see Figure 3):

(1) receipt of a client request when r_i acts as a primary (line 6); (2) receipt of a "no leadership" notification from the leader election service (line 14); (3) receipt of a "leadership" notification from the leader election service when r_i is not primary (line 16).

Receipt of a client request req_id when r_i acts as a primary. r_i first checks if the client request has been already served (line 7). In the affirmative, r_i returns to the client the global sequence number previously assigned to the request (line 8). Otherwise, r_i (i) increases by 1 the seq variable (line 9) and (ii) creates a tentative assignment ta such that $ta.\#seq = seq$; $ta.req\_id = req\_id$; $ta.\#epoch = state.epoch$ (line 10). Then r_i executes WRITEMAJ(ta) (line 11). If it successfully

```
Class Sequencer
 1  Tentative Assignment ta;
 2  State state := (∅, 0);
 3  boolean primary := ⊥; connected := ⊥;
 4  Integer seq := 0;
 5  loop
 6    when ((A-deliver ["GetSeq", req_id] from c) and primary) do
 7      if (∃ta' ∈ state.TA : ta'.req_id = req_id)
 8        then A-send ["Seq", ta'.#seq, req_id] to c;
 9        else seq := seq + 1;
10             ta.#seq := seq; ta.req_id := req_id; ta.#epoch := state.epoch;
11             if (WriteMaj (ta))
12               then A-send ["Seq", seq, req_id] to c;
13               else primary := ⊥;
14    when (not Leader?()) do
15      primary := ⊥;
16    when ((Leader?()) and (not primary)) do
17      (connected, maj_state) := ReadMaj ();   % computing_sequencer_state %
18      if (connected)
19        then ta := last(maj_state.TA);
20             if (ta ≠ null)
21               then connected := WriteMaj (ta);
22                    if (connected)
23                      then for each ta_j, ta_ℓ ∈ maj_state.TA :
24                                (ta_j.#seq = ta_ℓ.#seq) and (ta_j.#epoch > ta_ℓ.#epoch)
25                                do maj_state.TA := maj_state.TA − {ta_ℓ};
26                           state.TA := maj_state.TA; seq := last(state.TA).#seq;
27             if (WriteMaj (maj_state.epoch + 1) and connected)
28               then primary := ⊤;
29  end loop
```

Fig. 3. The Sequencer Protocol Pseudo-code Executed by r_i

returns, ta becomes a definitive assignment and the result is sent to the client (line 12). Otherwise, the primary sets $primary = \perp$ (line 13) as WriteMaj(ta) failed and r_i stops serving client requests.

Receipt of a "leadership" notification when r_i is not primary. A *css* procedure (lines 17-28) is started by r_i to become primary. As described in the previous section, r_i has to successfully complete the following four actions to become primary: (1) r_i invokes ReadMaj()(line 17). If the invocation is successful it timely returns a majority state in the maj_state variable[4]. (2) r_i extracts the last assignment ta from $maj\_state.TA$ (line 19) and invokes WriteMaj(ta) (line 21) to make definitive the last assignment of $maj\_state.TA$ (see the examples in the previous section). (3) r_i eliminates from $maj\_state.TA$ any assignment ta_ℓ such that it exists another assignment ta_j having the same sequence number of ta_ℓ but greater epoch number (lines 23-25). The presence of such a ta_j in maj_state implies that ta_ℓ is not definitive. This can be intuitively justified by noting that if an assignment ta_j performed by a primary p_k is definitive, no following primary will try to execute another assignment with the same sequence number. After the filtering, $state.TA$ is set to $maj\_state.TA$ and seq to $last(state.TA).\#seq$ as this is the last executed definitive assignment (line 26).

[4] Due to the time taken by the the leader election protocol[5] (at least 2δ) to select a leader (see Section 3), it follows that any ReadMaj() function starts after the arrival of all the *timely* messages broadcast through any previous WriteMaj().

(4) r_i invokes WRITEMAJ($maj\_state.epoch+1$) at line 27 to impose its primary epoch number (greater than any previous primary). Then, r_i becomes primary (line 28).

If any of the above actions is not successfully executed by r_i, it will not become primary. Note that if r_i is still leader after the unsuccessful execution of the *css* procedure, it starts to execute it again.

Receipt of a "no leadership" notification. r_i sets the *primary* variable to $\perp$ (line 15). Note that a notification of "no leadership" imposes READMAJ() and WRITEMAJ() to fail (i.e. to return $\perp$). Consequently if r_i was serving a request and executing statement 11, it sets *primary* to $\perp$ (line 13).

Note that the proposed implementation adopts an optimistic approach[4]: it allows internal inconsistencies among the sequencer replica states as it requires only a majority of replicas to be updated at the end of each definitive assignment. In other words the implementation sacrifices update atomicity to achieve better performances in failure-free runs. The price to pay is in the *css* phase carried out at each primary change.

It can be shown that the proposed protocol (along with the simple client invocation semantic described in Section 3) satisfies the sequencer specification given in Section 2. A detailed proof of correctness is given in[1].

5 Conclusions

In this paper we presented the specification of a sequencer service that allows thin, independent clients to get a unique and consecutive sequence number to label successive operations. We have then shown a fault-tolerant sequencer implementation based on a primary-backup replication scheme that adopts a specific partially synchronous model, namely the timed asynchronous model.

The proposed implementation adopts an optimistic approach to increase performances in failure-free runs with respect to (possible) implementations using standard group communication primitives, e.g. total order multicast. This follows because the proposed implementation only requires a majority of replicas to receive primary updates.

The practical interest of a fault-tolerant implementation of a sequencer service lies in the fact that it can be used to synchronize processes running over an asynchronous distributed system. For example, in the context of software replication, the sequencer actually embeds the partial synchrony necessary to solve the problem of maintaining server replica consistency despite process failures. This also allows to free server replicas from running over a partially synchronous system, i.e. to deploy server replicas over an asynchronous system.

References

1. R. Baldoni, C. Marchetti, and S. Tucci-Piergiovanni. Fault Tolerant Sequencer: Specification and an Implementation. Technical Report 27.01, Dipartimento di Informatica e Sistemistica, Università di Roma " La Sapienza", november 2001.

2. R. Baldoni, C. Marchetti, and S. Tucci-Piergiovanni. Active Replication in Asynchronous Three-Tier Distributed System. Technical Report 05-02, Dipartimento di Informatica e Sistemistica, Università di Roma " La Sapienza", february 2002.
3. N. Budhiraja, F.B. Schneider, S. Toueg, and K. Marzullo. The Primary-Backup Approach. In S. Mullender, editor, *Distributed Systems*, pages 199–216. Addison Wesley, 1993.
4. X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 43–50, West Lafayette, IN, USA, October 1998.
5. C. Fetzer and F. Cristian. A Highly Available Local Leader Election Service. *IEEE Transactions on Software Engineering*, 25(5):603–618, 1999.
6. C. Fetzer and F. Cristian. The Timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.
7. M. Fischer, N. Lynch, and M. Patterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.
8. R. Guerraoui and A. Schiper. Software-Based Replication for Fault Tolerance. *IEEE Computer - Special Issue on Fault Tolerance*, 30:68–74, April 1997.
9. M. Raynal. *Algorithms for Mutual Exclusion*. MIT Press, 1986.
10. F.B. Schneider. Replication Management Using State-Machine Approach. In S. Mullender, editor, *Distributed Systems*, pages 169–198. Addison Wesley, 1993.