Fault-Tolerant Sequencer: Specification and an Implementation*

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

The synchronization among thin, independent and concurrent processes in an open distributed system is a fundamental issue in current architectures (e.g. middlewares, three-tier architectures etc.). "Independent process" means no message has to be exchanged among the processes to synchronize themselves and "open" means that the number of processes that require to synchronize changes along the time. 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 these labelled operations). Moreover, we propose an implementation of the sequencer service in the timed asynchronous model along with its correctness proof.

1 Introduction

Since the middle of 80s, the abstraction of process group has been of primary importance in designing fault-tolerant distributed applications [3]. A group is a set of cooperating processes able to order events (such as message receipts, failures, recoveries etc.) identically [3] by using specific group operations and mechanisms such as broadcast communication primitives, views operations, state transfer, failure detection mechanisms etc. ([4]). This approach, which actually *tightly couples* the processes of a group, has shown to work quite well on-the-field when facing small and closed groups [2].

Recently, distributed systems are moving towards *open* distributed architectures in which *loosely coupled* and *independent* entities cooperate in order to meet a common goal. Success factors of such architectures are application scalability, mainteinability and (sometimes) portability as well as the simplification of application development. *Three-tier architectures* (e.g. [1, 12, 7, 15]) and open *middleware platforms* (e.g. [19]) are examples of such open distributed systems.

A three-tier architecture separates the presentation logic (implemented by thin clients), the application logic (implemented by the mid-tier), and the application state (stored into the back-end servers) of a distributed application. The mid-tier actually orchestrates the back-end server accesses on behalf of clients. Three-tier architectures are open in the sense that they allow mid-tier to interact with a variety of clients and back-end servers which is not necessarily a priori known. Moreover, clients and back-end servers are usually independent i.e., clients (resp.

*Technical Report #27.01, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", November 2001

back-end entities) do not exchange messages among them, and are loosely coupled, i.e. clients and servers embed only rudimentary failure detection mechanisms based on time-outs and retransmissions to detect the crash of cooperating entities.

A middleware platform such as CORBA [19] basically allows an a priori unknown number of heterogeneous objects to cooperate through a standard "object bus" (the Object Request Broker - ORB - component). Moreover, specific operations that could be used by a wide range of distributed applications (such as naming, many-to-many communication, security etc) are provided by standard services.

Independently from the nature of a distributed architecture, a basic issue like the *synchronization* among processes spread over the computer network has to be faced. Synchronization in the presence of failures is the fundamental building block for the solution of many important problems in distributed systems such as mutual exclusion ([20]) and replication ([13]), just to name a few. Synchronization in mutual exclusion is needed to get a total order on critical section accesses while in replication to get the same total order of updates at each replica[1].

In the context of a closed group, the solutions proposed for this class of problems are mainly either fully distributed (each process runs the same protocol) [3, 6] or token-based [16]. These solutions do not fit well open architectures, where (i) the number of cooperating processes could change along the time, (ii) cooperating processes are not supposed to implement a common distributed protocol and (iii) such processes could not directly communicate (this implies that cooperating processes could even not know the existence of each other). These reasons make appealing the service approach for synchronization in such architectures. Therefore, in some sense, in open architectures the notion of service *replaces* the notion of process group.

In this paper we first present the specification of a sequencer service that allows thin client processes which implement a rudimentary time-out based retransmission mechanism to get a sequence number that can be used to label their operations. Such specification allows different processes to get a different sequence number for each distinct operation despite, for example, multiple receipts of the same request by the sequencer. Moreover, the sequence number associated by the sequencer to each request are consecutive[2].

Then, we propose a fault-tolerant implementation of the sequencer service. Such implementation adopts a primary-backups (passive) replication scheme where replicas interact with clients through asynchronous channels and among them through a timed asynchronous datagram service [8]. This model captures the interesting application scenario in which replicas are on a LAN while clients are thin applications (e.g. browsers) spread over the Internet. Finally, we prove the correctness of our implementation with respect to the sequencer specification.

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 presents the formal correctness proof of our implementation. Finally, Section 6 concludes the paper.

[1] In an asynchronous distributed system where processes can crash this problem cannot be solved (unless a minimum degree of synchrony is added [6] to the system) as it is equivalent to solve the consensus problem [11].

[2] In the context of the replication problem, a client could get its number from the sequencer and then send its update request (labelled with the sequence number) to each replica. A total order on updates at each replica can be then easily achieved thanks to this sequence number consecutiveness property.

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 identifier of the client 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 $\{a_1, a_2 \dots a_{k-1}, a_k\}$ where each assignment a corresponds to a pair $\langle req\_id, \#seq \rangle$, in which $a.req\_id$ is a client 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 five 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 identified by req_id, then, unless the client crashes, it eventually delivers a reply $\#seq$.

Properties from (P1) to (P4) define the safety of the sequencer service while (P5) defines its liveness.

More specifically, (P1) expresses that the state of the sequencer does not contain "spurious" assignments. i.e., each assignment has been executed after having received a request from some client. (P2) states that the client cannot deliver a sequence number that has not been assigned by the sequencer to a req_id. The predicate "(P1) and (P2)" implies that each client delivering a sequence number has previously issued a request.

Property (P3) states that there is an one-to-one correspondence between the set of req_id and the elements of the set A. i.e., the sequencer has to assign a different sequence number to each distinct client request. Property (P4) says that numbers assigned by the sequencer to client requests do not have "holes" in a sequence starting from one. Property (P5) expresses the fact 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 $\{r_1, \dots, r_n\}$ of processes implementing the fault-tolerant sequencer. A client c runs in an asynchronous distributed system and communicates only with replicas using *reliable asynchronous channels*. Replicas exchange messages among them by using a *timed asynchronous datagram service*([10, 8]).

3.1 Client Processes

A client process sends a request to the sequencer service and then waits for a sequence number. A client performs (unreliable) failure detection of replicas using only local timeouts and cope with replica failures using a simple retransmission mechanism. A client may fail by crashing.

Communication between clients and replicas is *asynchronous* and *reliable*. Therefore, (i) there is no bound on message transfer delay and process speeds (asynchrony) and (ii) messages exchanged between two non-crashing processes are eventually delivered (reliability). More specifically, clients and replicas use the following communication primitives to exchange messages:

- **A-send**(m, p): to send an unicast message m to process p;
- **A-deliver**(m, p): to deliver a message m sent by process p.

The client pseudo-code is shown in Figure 1. To label a generic event with a sequence number generated by the sequencer service, a client invokes the GetSeq() method (line 3). Such method blocks the client process and invokes the sequencer replicas. Once an integer sequence number has been received from a replica, the GetSeq() method returns it as output parameter. In particular, the GetSeq() method first assigns to the ongoing request a unique request identifier $req\_id = \langle cl\_id, \#cl\_seq \rangle$ (line 7-8), and then enters a loop (line 9). Within the loop, the client (i) sends the request to a replica (line 10) and (ii) sets a local timeout (line 11). Then, a result is returned by GetSeq() if the client process receives within the timeout period a sequence number for the *req_id* request (line 14). Otherwise another replica is selected (line 15) and the request is sent again towards such a replica (line 12).

```
Class Client
 1  rlist := ⟨r_1, ..., r_n⟩;
 2  Integer #cl_seq := 0;
 3  Integer GetSeq()
 4    begin
 5      Integer i := 0;
 6      Request req_id;
 7      #cl_seq := #cl_seq + 1;
 8      req_id :=< cl_id, #cl_seq >;
 9      loop
10        A-send ["getSeq", req_id] to rlist[i];
11        t.setTimeout := period;
12        wait until ((A-deliver ["Seq", seq, req_id] from r ∈ rlist) or (t.expired()))
13        if (not t.expired())
14          then return (seq);
15          else  i := (i + 1) mod |rlist|;
16      end loop
17    end
```

Figure 1: Protocol Executed by a Client c

3.2 Replica Processes

In this section we outline the main features of the Timed Asynchronous Model and of the leader election service designed in this system model. Interested readers can refer to [10, 9] for further details.

Each replica r_i has access to a hardware clock with bounded drift rate with respect to other replicas' clocks. Replicas can fail by crashing. However they can also become "slow" with respect to their specification: a time-out σ is introduced to define a replica performance failure. A replica with a scheduling delay greater than σ suffers a performance failure. 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.

Communications among replicas occur through channels that are subject to two kind of failures: a message can be omitted (dropped) or can be delivered after a given timeout δ (performance failure). A message whose transmission delay is at most δ is *timely*. Two replicas are *connected* in a time interval $[s,t]$ iff they are *timely* in $[s,t]$ and each message exchanged between the two replicas in $[s,t]$ it timely. A subset of replicas form a *stable* partition in $[s,t]$ if any pair of replicas belonging to the subset is connected. Timed asynchronous communications are achieved through a *datagram service* ([8]) which filters out non-timely messages to the above layer. In the following we assume replicas communicate 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): 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 by Cristian and Fetzer in [9]. The leader election service ensures that:

- at every physical time there exists at most one *leader*, a *leader* is a replica in which the *Leader?()* boolean function returns *true*;
- the leader election protocol underlying the *Leader?()* boolean function takes at least 2δ for a leader change;
- when a majority of replicas forms a stable partition in a time interval $[t, t+\Delta t]$ ($\Delta t >> 2\delta$) , 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 connected to all other replicas of its stable partition for the duration of its leadership.

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 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[3] 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 (or passive replication) scheme is adopted [5, 13]. In this scheme, a particular replica, the *primary*, handles all the requests coming from clients. At most one replica can be the primary at any physical time t. Other replicas (the *backups*) do not interact with clients and exchange messages only with the primary. When a primary receives a client request, it assigns a sequence number to the request, updates the backups by using an update primitive and then sends the reply with the sequence number to the client.

Backup failures are transparent to clients while, when a primary fails (either by crashing or by a performance failure), a main problem has to be addressed: *the election of a new primary whose internal state verifies the sequencer specification properties described in Section* 2.

In our implementation, the election of a primary lies on:

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

1. The availability of the leader election service running among the replicas (see Section 3). To be a leader is a necessary condition firstly to have the chance to become the primary and, secondly, to stay as the primary.

2. A "reconciliation" procedure (namely "*computing_sequencer_state*" procedure) 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 sequencer service properties defined in Section 2.

In our implementation we exploit the *computing_sequencer_state* procedure, in order to enhance performance of the update primitives during failure-free runs. More specifically, an update primitive (denoted WriteMaj()) issued by a primary successfully returns if it timely updates at least a *majority* of replicas. As a consequence during the reconciliation procedure a newly elected leader, before becoming a primary, has to read at least a majority of states of other replicas (this is done by a ReadMaj() primitive). This allows a leader to have a state containing all the successfully updates done by previous primaries. Then the leader removes from such state all possible inconsistencies caused by unsuccessful primary updates.

In the next subsections, we first introduce the data structures local at each replica. Then we present the two basic primitives used for the interaction among replicas: WriteMaj() and ReadMaj() functions. The first is used by a primary to update its backups. The second is used by a newly elected leader to start the *computing_sequencer_state* procedure fetching the local states of a majority of replicas.

4.1 Protocol Data Structures

Each replica r_i endows:

- *primary* boolean variable, which is set according to the role (either primary or backup) played by the replica at a given time;
- *seq* integer variable, which represents the sequence number assigned to a client request when r_i acts as a primary;
- *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 the current primary, *epoch* has to be greater than any *epoch* value associated with previous primary. *state.epoch* is set when a replica becomes primary and it does not change during all the time a replica is the primary[4].
- A *tentative assignment ta* is a triple $\langle req\_id, \#seq, \#epoch \rangle$ where $ta.\#seq$ is the sequence number assigned to the client request $ta.req\_id$ and $ta.\#epoch$ is the epoch number of the primary that executed the assignment ta.

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

4.2 Basic Primitives and Definitions

The pseudo-codes of the WriteMaj() and the ReadMaj() functions are respectively shown in Figures 2 and 3. Moreover, Figure 4 shows the pseudo-code of the Listener() thread handling message receipts at each replica.

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

WriteMaj(). The WRITEMAJ() function (Figure 2) takes as input argument m and returns a boolean b. m can be either a tentative assignment ta or an epoch e. In both cases, upon invocation, the WRITEMAJ() function first checks if the replica is the leader, then it executes **TA-broadcast**(m) and then sets a timer of duration [5] $T = 2\delta(1+\rho)$ to count the number of timely received acknowledgement messages (lines 5-10). Each replica sends an acknowledgement upon the delivery of m (see Figure 4, line 8). When the timer expires (line 11) the function checks if a majority of timely acknowledgments has been received (line 12). In the affirmative, m is put into the replica state according to its type (line 15-16), then the function returns $\top$ (i.e. it *successfully* returns) if the replica is still the leader at the end of the invocation (line 17).

```
1  BOOLEAN WRITEMAJ(MSG msg)
2    begin
3      BOOLEAN succeeded := ⊥;
4      INTEGER i := 0;
5      if (Leader?())
6        then TA-broadcast (["Write", msg]);
7             alarmclock.setalarm(T); % T = H() + 2δ(1 + ρ) %
8             loop
9               when (TA-deliver (["Ack"](sender))) do
10                i := i + 1;
11              when (alarmclock.wake(T)) do
12                if (i ≥ ⌈(n+1)/2⌉)
13                  then succeeded := ⊤;
14                       if (msg is Assignment)
15                         then state.TA := state.TA ⋃ msg;
16                         else  state.epoch := msg;
17                return (succeeded and Leader?())
18            end loop
19   end
```

Figure 2: The WRITEMAJ() Function Pseudo-code Executed by r_i

Let us finally present the following observations that will be used in Section 5 to show the correctness of our protocol:

Observation 1. *Let ta be a tentative assignment, if r_i successfully executes* WRITEMAJ(ta) *then* $\exists maj : maj \subseteq \{r_1, ..., r_n\}, |maj| \geq \lceil \frac{n+1}{2} \rceil,\ r_i \in maj,\ r_j \in maj \Rightarrow ta \in state_{r_j}.TA$.

Observation 2. *Let ta be a tentative assignment, if r_i executes without success* WRITEMAJ(ta) *then* $ta \notin state_{r_i}.TA$.

Observation 3. *Let e be an epoch number, if r_i successfully executes* WRITEMAJ(e) *then* $\exists maj : maj \subseteq \{r_1, ..., r_n\}, |maj| \geq \lceil \frac{n+1}{2} \rceil,\ r_i \in maj,\ r_j \in maj \Rightarrow state_{r_j}.epoch = e$.

Such properties trivially follow from the WRITEMAJ() function and LISTENER() thread pseudo-codes (Figure 2 and Figure 4).

Definitive and Non-definitive Assignments. We are now in the position to introduce the notion of *definitive assignment* that will be used in the rest of the paper:

[5] 2δ is the maximum round-trip time for timely messages in timed asynchronous model and $2\delta(1+\rho)$ is the timeout to set in a replica with the clock with maximum positive drift (ρ) to measure a 2δ time duration.

Definition 1. *A tentative assignment ta is a* definitive *assignment* iff *exists a primary p such that p executed* WriteMaj(ta)$= \top$.

Therefore, a definitive assignment is a tentative one. The viceversa is not necessarily true. A tentative assignment which is not definitive is called *non-definitive*.

Non-definitive assignments are actually inconsistencies due to unsuccessful WriteMaj() executions. If the state of a primary would contain non-definitive assignments, it could violate the bijection property (Section 2). However, by filtering out non-definitive assignments during the *computing_sequencer_state* procedure, we let the state of a primary contain only definitive assignments (Lemma 7 in Section 5). Thus we enforce the bijection property only on definitive assignments (Theorem 3 in Section 5).

ReadMaj( ). This function does not have input arguments and returns as output parameter a pair $\langle b, s\rangle$ where b is a boolean value and s is a state as defined in Section 4.1. If ReadMaj() returns $b = \top$, then (i) $s.TA$ contains the union of the tentative assignments contained in the states of a majority of replicas, denoted *maj_state* and (ii) *s.epoch* equals the greatest epoch number contained in states of the replicas belonging to *maj_state*.

As shown in Figure 3, this function executes a **TA-brodcast**() (line 6) which causes the Listener() thread running in every replica r_i to send its replica state ($state_i$) as the reply to the broadcast (Figure 4, line 9). After executed the broadcast, the ReadMaj() function sets a timeout (line 7) to count timely replies. Then it enters a loop where handles two types of events, i.e. the arrival of a timely reply and the elapsing of the timeout. In the first case (lines 9-12) the function (i) merges the tentative assignments contained in the just received state ($state_{sender}$) with the one contained in the $maj\_state.TA$ variable (initially empty), (ii) sets the *maj_state.epoch* value to the maximum between the current *maj_state.epoch* value and $state_{sender}.epoch$ and (iii) increases the counter of the timely reply.

In the second case (i.e., when the timeout elapses, line 13), the function checks if at least a majority of timely replies has been received (line 14). In the affirmative, if the replica is still the leader it returns a pair $\langle b, s\rangle$ with $b = \top$ and $s = maj\_state$ (line 17).

Let us introduce the following observations that will be used in Section 5 to show the correctness of our protocol:

Observation 4. *If r_i successfully executes* ReadMaj(), *then maj_state.TA contains all the definitive assignments previously executed.*

Observation 5. *If r_i successfully executes* ReadMaj(), *then maj_state.epoch* $\geq max\{e' :$ *some replica executed* WriteMaj(e') $= \top\}$.

Such properties trivially follow from the ReadMaj() function and Listener() thread pseudo-codes (Figure 3 and Figure 4) and from Definition 1.

4.3 Introductory Examples and Descriptions

Let us present in this section two introductory examples and a preliminary explanation of the sequencer protocol before getting through the pseudo-code executed by each replica (shown in Figure 7).

Primary Failure-Free Behaviour. A primary upon receiving a client request first checks if an assignment was already associated to the request, otherwise (i) it creates a new tentative assignment *ta* embedding the request identifier and a sequence number consecutive to one associated to the last served 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.

```
1  <Boolean,State> ReadMaj()
2    begin
3      Boolean succeeded := ⊥;
4      Integer i := 0;
5      State maj_state := (∅ 0);
6      TA-broadcast ([“Read′′]);
7      alarmclock.setalarm(T); % T = H() + 2δ(1 + ρ) %
8      loop
9        when (TA-deliver ([“State”, state_sender](sender))) do
10         maj_state.TA := maj_state.TA ⋃ state_sender.TA;
11         maj_state.epoch := max(maj_state.epoch, state_sender.epoch);
12         i := i + 1;
13       when (alarmclock.wake(T)) do
14         if (i ≥ ⌈(n+1)/2⌉)
15           then succeeded := ⊤;
16         return (succeeded and Leader?(), maj_state)
17     end loop
18   end
```

Figure 3: The ReadMaj() Function Pseudo-code Executed by r_i

```
1  Thread Listener()
2    begin
3      when (TA-deliver ([typemsg, recmsg](sender))) do
4        case typemsg
5        {“Write′′} : if (recmsg is Assignment and r_i ≠ sender)
6                       then state.TA := state.TA ⋃ recmsg;
7                       else state.epoch := recmsg;
8                    TA-send ([“Ack” ]) to sender;
9        {“Read′′} : TA-send ([“State”, state]) to sender;
10       end case
11   end
```

Figure 4: The Listener() Thread Pseudo-code Executed by r_i

Change of Primary. There are three events that cause a primary replica r_i to loose the primaryship:

1. r_i fails by crashing or
2. WriteMaj(ta) returns $\perp$ (WriteMaj(ta) *could* have notified ta to less than a majority of replicas) or
3. there is a loss of leadership of r_i (i.e., the *Leader*?() value becomes false in r_i),

In the last two cases, r_i sets its boolean flag *primary* to false. If any of these events occurs, the protocol waits that a new leader is elected by the underlying leader election service. Then the “*computing_sequencer_state*” procedure is carried out by the new leader to let *state* verify the sequencer properties (Section 2) before starting serving client requests as primary.

The “*computing_sequencer_state*” procedure. The first action performed by a newly elected leader r_i is to invoke ReadMaj(). If ReadMaj() returns false 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 backup replicas, denoted *maj_state*, has been fetched by READMAJ(), the *computing_sequencer_state* procedure executed by r_i has three main goals:

- to transform the tentative assignment $last(maj\_state.TA)$ in a definitive assignment *on behalf of a previous primary* that issued WRITEMAJ($last(maj\_state.TA)$), as there is no way for r_i to know if that WRITEMAJ() was executed with success by the previous primary.
- to remove from $maj\_state.TA$ all non-definitive assignments. Non-definitive assignments are filtered out using the epoch field of a tentative assignment. More specifically, our sequencer implementation enforces the bijection property (Section 2) by guaranteeing that when *there are multiple assignments with the same sequence number, the one with the greatest epoch number is a definitive assignment* (see Lemma 3 in Section 5). The filter is shown in Figure 7 from line 23 to line 25.
- to impose a primary epoch number e by using a WRITEMAJ() function. e is greater than the one returned by READMAJ() in *maj_state.epoch*. From Observation 5, it also follows that e is greater than all previous epoch numbers associated to primaries.

If r_i executed with success all previous points it sets *state* to *maj_state* and starts serving client requests as primary.

In the following we introduce two examples which point out how the previous actions removes inconsistencies (i.e., non-definitive assignments) from a primary state during the *computing_sequencer_state* procedure.

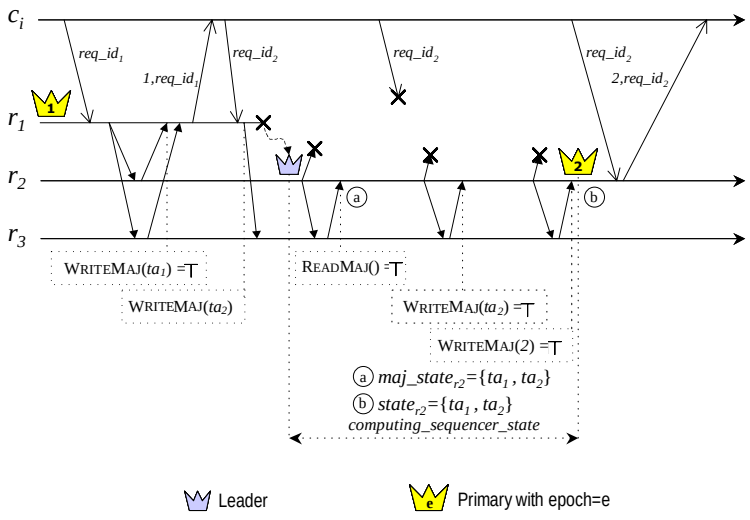

Figure 5: Example of a Run of the Sequencer Protocol

Example 1: Avoiding inconsistencies by redoing the last tentative assignment. Figure 5 shows a protocol run in which a primary replica r_1 starts serving client requests. In particular 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 finally sends back the result $\langle 1, req\_id_1\rangle$ to the client. Then r_1 receives a new client request $req\_id_2 \neq req\_id_1$, invokes WRITEMAJ($ta_2 = \langle req\_id_2, 2, 1\rangle$) and crashes during the invocation. Before crashing it updated only replica r_3. The next leader r_2 enters the sequencer state computation: it executes READMAJ(), which returns in $maj\_state.TA$ the union of states of r_2 and r_3 (i.e., $\{ta_1, ta_2\}$) and in $maj\_state.epoch$ the epoch number 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). r_2 then executes WRITEMAJ($maj\_state.epoch + 1$) to notify its epoch number as last action of the "*computing_sequencer_state*" procedure. Finally, when r_2 receives the request req_id_2, it finds ta_2 in its state and immediately returns $ta_2.\#seq$ to the client.

Example 2: Avoiding inconsistencies by filtering out non-definitive assignments. The example is shown in Figure 6. Primary r_1 successfully serves request req_id_1. Then, upon the arrival of a new request req_id_2, it invokes WRITEMAJ(), exhibits a performance failure and updates only replica r_3 (ta_2 is a non-definitive assignment). As a consequence r_i looses its primaryship and another leader r_2 is elected. r_2 executes READMAJ() which returns in maj_state the union of assignments belonging to r_1 and r_2 states (i.e., $\{ta_1\}$). Then r_2 ends its reconciliation procedure by executing WRITEMAJ(ta_1)= $\top$ and by notifying its epoch.

Upon the arrival of a new request req_id_3, primary r_2 executes WRITEMAJ($ta'_2 = \langle req\_id_3, 2, 2\rangle$) with success (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 (i.e., ta_2 and ta'_2) with a same sequence number and different epoch numbers ($ta_2.\#epoch = 1$ and $ta'_2.\#epoch = 2$). The $maj\_state.TA$ of a successive leader r_i (r_1 in Figure 6) includes, from Observation 4, the definitive assignment ta'_2. 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 $ta_2.\#epoch < ta'_2.\#epoch = 2$. After the filtering, the state of the primary r_1 is composed only by definitive assignments. Note that without performing such filtering the bijection property would result violated, as the state of a primary could contain two assignments with a same sequence number.

Then, when r_1 receives the client request req_id_2 (due to the client retransmission mechanism) previously associated to ta_2, it performs WRITEMAJ($ta_3 = \langle req\_id_2, 3, 3\rangle$) and if it returns with success, r_1 returns the sequence number 3 to the client.

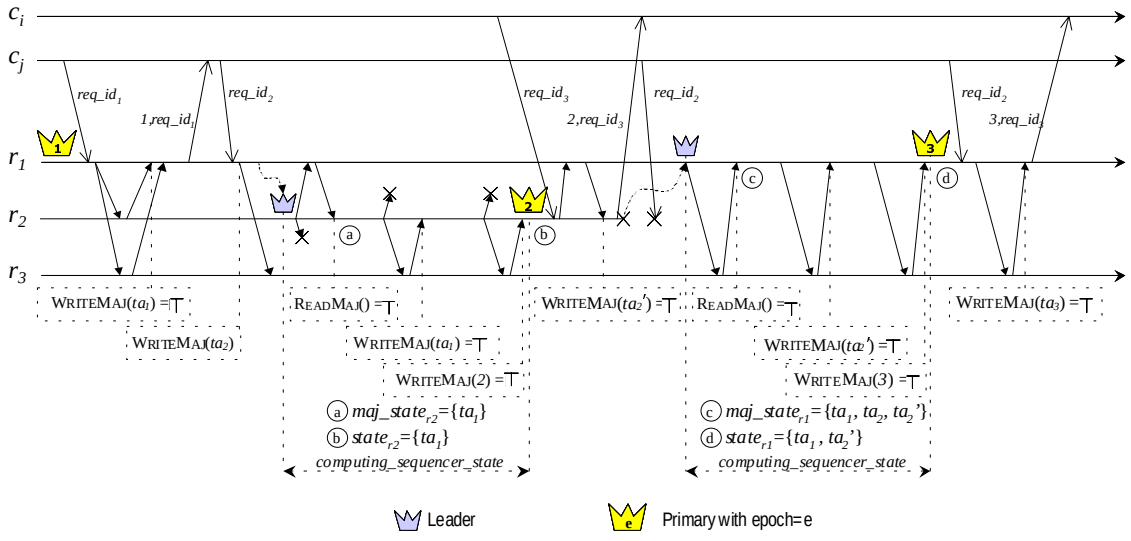

Figure 6: Example of a Run of the Sequencer Protocol

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 7):

1. Receipt of a client request when r_i acts as a primary (line 6);

```
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
```

Figure 7: The Sequencer Protocol Pseudo-code Executed by r_i

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 is a retransmission of an already served request (line 7). In the affirmative, p_i simply returns to the client the global sequence number previously assigned to the requests (line 8). Otherwise, p_i (i) increases by 1 the *seq* variable (line 9) and (ii) generates a tentative assignment *ta* such that $ta.\#seq = seq; ta.req\_id = req\_id; ta.\#epoch := state.epoch$ (line 10). Then p_i executes WRITEMAJ(ta) (line 11). If it successfully returns *ta* becomes a definitive assignment and the result is sent back to the client (line 12). Otherwise, the primary sets $primary = \bot$ (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 *computing_sequencer_state* procedure (lines 16-29) is started by r_i to become primary. As described in the previous section, r_i has to execute with success all the following four actions to become a primary:

A1. r_i invokes the READMAJ() function (line 18). If the invocation is successful it timely returns a majority state in the *maj_state* variable[6].

A2. 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).
A3. 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).
A4. 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 *computing_sequencer_state* procedure, it restarts to execute the procedure.

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 $\perp$, see Figure 2, line 17 and Figure 3 line 16). As a consequence if r_i was serving a request and executing statement 11, it sets *primary* to $\perp$ (line 13) upon a leadership loss.

5 Correctness Proof

In this section we show that our sequencer fault-tolerant implementation satisfies the sequencer properties defined in Section 2. Let us remark that the state of the sequencer service A corresponds to the set of tentative assignments, denoted $state_{p_i}.TA$, contained in the state of the current primary p_i. We first introduce some preliminary lemmas and then we prove the sequencer service properties.

5.1 Preliminary Lemmas

Definition 2 (Primary Sequence $\mathcal{P}$). *The primary sequence $\mathcal{P} = \langle p_1, ...p_k \rangle$ [7] is a sequence of replica identifiers r_i where p_i represents the i-th replica executing statement 28.*

Lemma 1.
If $ta \in state_{r_j}.TA$
then *(i) it exists c that issued a request identified by ta.req_id and (ii) it exists a primary $p_i \in \mathcal{P}$ which executed* WriteMaj(ta) *at statement 11.*

Proof. The existence of a tentative assignment $ta \in state_{r_j}.TA$ implies that a WriteMaj(ta) has been executed either at line 11 by a primary p_i or at line 21 by a leader not yet primary. The second case implies that ta is already in a state of some replica, therefore there was a previous primary p_i that issued a WriteMaj(ta) at line 11. Such statement can be executed only after the receipt of a client request al line 6. □

The following lemma shows that when r_i becomes the current primary, its *state.epoch* is greater than any *epoch* value associated to previous primary.

[6] Due to the time taken by the the leader election protocol [9] (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().
[7] the sequence is finite by global stabilization assumption.

Lemma 2. *Let* $|\mathcal{P}| = k, k > 1$ then

$$\forall p_i, p_j \in \mathcal{P}, i < j \Leftrightarrow state_{p_i}.epoch < state_{p_j}.epoch$$

Proof.

1. ($\Rightarrow$) By contradiction. Suppose $i < j$ and $state_{p_i}.epoch \geq state_{p_j}.epoch$. As p_i is a primary, it executed statement 27 and set $state_{p_i}.epoch$ (Observation 3). Therefore p_j before becoming primary invokes the ReadMaj() function, which returns $\top$ as boolean output parameter and in $maj\_state.epoch$ the maximum epoch number set by a replica that successfully executed statement 27 in the past (Observation 5). Hence, when p_j executes statement 27, we have $state_{p_j}.epoch > state_{p_j}.epoch$ that contradicts the initial assumption.

2. ($\Leftarrow$) By contradiction. Suppose $state_{p_i}.epoch < state_{p_j}.epoch$ and $i \geq j$. We distinguish the following two cases:

   - $(i = j)$. In this case we get the following absurd: $state_{p_i}.epoch < state_{p_i}.epoch$.
   - $(i > j)$. Exchanging i with j we fall in case 1.

□

The following lemma ensures that each definitive assignments is associated to a unique sequence number.

Lemma 3. *Let ta be a definitive assignment:*

$$\nexists ta' \in state_{r_i}.TA : ta'.seq = ta.seq \wedge ta'.\#epoch > ta.\#epoch$$

Proof. Suppose ta is a definitive assignment and $\exists ta' \in state_{r_k}.TA : ta'.seq = ta.seq \wedge ta'.\#epoch > ta.\#epoch$. As ta is definitive and from Lemma 1, there exist two primaries p_i, p_j that respectively executed WriteMaj(ta)$= \top$ and WriteMaj(ta'). As $ta'.\#epoch > ta.\#epoch$ from Lemma 2, it follows $j > i$. Without loss of generality, we assume $\nexists p_{j'} \in \mathcal{P}$ with $i < j' < j$ which executed WriteMaj(ta''), and $ta''.\#epoch > ta.\#epoch \wedge ta''.\#seq = ta.\#seq$. Then when p_j executes statement 17, $ta \in maj\_state_{p_j}.TA$ (Observation 4), $last(maj\_state.TA).\#seq \geq ta.\#seq$. This implies $seq_{p_j} \geq ta.\#seq$ at line 26. Hence, for each $\overline{ta}$ such that p_j executes WriteMaj($\overline{ta}$) at statement 11, it follows $\overline{ta}.\#seq > ta.\#seq$ as seq_{p_j} is incremented by 1 at statement 9. When $\overline{ta} = ta'$, we get the following absurd $ta'.\#seq > ta'.\#seq$ that contradicts the initial assumption. □

The following lemma shows that a definitive assignment will never be removed from the state of a primary.

Lemma 4. *Let* $p_i \in \mathcal{P}$ *be a primary that executes* WriteMaj(ta)$=\top$*:*

$$\forall p_j \in \mathcal{P} \wedge j > i \Rightarrow ta \in state_{p_j}.TA \text{ before } p_j \text{ starts serving requests}$$

Proof. By contradiction. $\exists p_i, p_j, j > i : p_i$ executes WriteMaj(ta)$=\top \wedge ta \notin state_{p_j}.TA$ before p_j starts serving request (it means $ta \notin state_{p_j}.TA$ when p_j ends the *computing_sequencer_state* procedure). As each primary $p_k, k > i$, executes the *computing_sequencer_state* procedure, it will execute statement 17. From the Observation 4, it follows that $ta \in maj\_state.TA$. Lemma 3 implies ta cannot be eliminated at statement 25. Hence $ta \in state_{p_k}.TA$ at statement 26. This holds also for $k = j$ contradicting the initial hypothesis. □

The following lemma says that a primary that attempts to perform an assignment after a client request assigns to the request the greatest sequence number.

Lemma 5. If p_i *executes* WriteMaj(ta) *at statement 11,* then *for each definitive assignment* ta'*:* $ta' \in state_{r_k}.TA \Rightarrow ta'.\#seq < ta.\#seq$

Proof. By contradiction. Suppose p_i executes WRITEMAJ(ta) at statement 11 and $\exists ta' \in state_{r_k}.TA : ta'.\#seq \geq ta.\#seq$ and ta' is definitive. As ta' is definitive, there exists a primary p_j that executed WRITEMAJ(ta')=$\top$. We have two cases:

- $(i = j)$. As p_i increases the *seq* variable (line 9) before invoking the WRITEMAJ() function at statement 11, it follows $ta.\#seq > ta'.\#seq$ contradicting the initial hypothesis.
- $(j < i)$. From Lemma 4 when p_i executes line 17, $ta' \in maj\_state_{p_i}.TA$. Hence $last(maj\_state_{p_i}.TA).\#seq \geq ta'.\#seq$ (line 19) and $seq_{p_i} \geq ta'.\#seq$ at line 26. As a primary increases the *seq* variable before serving a client request, when p_i executes WRITEMAJ(ta) at statement 11, we have $ta.\#seq > ta'.\#seq$ contradicting the initial hypothesis.

□

The following lemma says that when a primary attempts to assign a sequence number to a client, the previous number has been already assigned.

Lemma 6.

If p_i *executes* WRITEMAJ(ta) *at statement 11 and* $ta.\#seq > 1$
then *there exists a definitive assignment* $ta' \in state_{p_i}.TA : ta'.\#seq = ta.\#seq - 1$.

Proof. By contradiction. Suppose it does not exist definitive assignment $ta' \in state_{p_i}.TA$ such that $ta'.\#seq = ta.\#seq - 1$, and p_i executes WRITEMAJ(ta) at statement 11 with $ta.\#seq > 1$. We have two cases:

- (p_i executes statement 11 for the first time). Being primary, p_i completed *computing_sequencer_state* procedure (lines 17–29). When p_i executed statement 19, let $ta' = last(maj\_state)$. We have two cases:
  (a) $ta' = null$. When p_i executes statement 11, from line 4 and line 9 it follows that $ta.\#seq = 1$.
  (b) $ta' \neq null$. Then p_i executed statement 21-26, which imply (i) $ta' \in state_{p_i}.TA$ (from Observation 1), (ii) ta' is a definitive assignment and (iii) $seq_{p_i} = ta'.\#seq$. When p_i finally executes line 9, it increases seq_{p_i} by one, hence $ta'.\#seq = ta.\#seq - 1$. This contradicts the initial hypothesis.
- (p_i already executed statement 11 at least one time). In this case p_i has previously executed statement 11 with an assignment ta'. Therefore $ta' \in state_{p_i}.TA$ (Observation 1), and ta' is a definitive assignment. When p_i executes WRITEMAJ(ta) at statement 11, from line 9, it follows $ta.\#seq = ta'.\#seq + 1$ which contradicts the initial hypothesis.

□

Lemma 7.

If $ta \in state_{p_i}.TA$ then *ta is a definitive assignment.*

Proof. By contradiction. Suppose $ta \in state_{p_i}.TA$, ta is not a definitive assignment, i.e. $\nexists p_w \in \mathcal{P}$ that executed WRITEMAJ(ta)= $\top$. Without loss of generality, we assume that ta is the first non-definitive assignment among the assignment executed so far in the system. From Lemma 1, it must exist a primary $p_j, j \leq i$ that executed WRITEMAJ(ta)= $\bot$ at statement 11. We distinguish the following two cases:

- $(i = j)$. In this case, from Observation 2, it follows that $ta \notin state_{p_i}.TA$. Contradiction.
- $(j < i)$. Let $p_k, j < k \leq i$ be the first primary such that (a) $maj\_state_k.TA$ at line 17 contains *non-definitive* assignments and (b) $ta \in maj\_state_k.TA$. If ta belongs to $state_{p_k}.TA$, two conditions must hold:
  (i) $last(maj\_state_k.TA) \neq ta$ and (ii) $\nexists ta' \in maj\_state.TA : ta'.\#seq = ta.\#seq \wedge ta'.\#epoch > ta.\#epoch$
  We have two cases:

1. (no primary $p_h, j < h < k$ executes statement 11).

   In this case from Lemma 5 follows $last(maj\_state_k.TA) = ta$. This contradicts (i).

2. (some primary $p_h, j < h < k$ executes statement 11). In this case it exists a non-empty sequence of tentative assignments $s = \langle ta_1, ..., ta_m \rangle$ such that for each $ta_i \in s, ta_i \neq ta$ we have $ta_i.\#epoch > ta.\#epoch$ (Lemma 2). We have two cases:

   – (There not exists a $ta_i \in s$ such that WriteMaj(ta_i)= $\top$) as $maj\_state_h.TA$ contains only definitive assignments (from assumption (a)), from Lemma 6 it follows $last(maj\_state_h.TA) = ta' : ta'.\#seq = ta.\#seq - 1$, therefore when p_h executes WriteMaj(ta_i) we have $ta_i.\#seq = ta.\#seq$.
   If ta_i belongs to $maj\_state_k.TA$ then condition (ii) is no longer true. Otherwise, if $maj\_state_k.TA$ does not contain any of the tentative assignments of s, we fall in case 1.

   – (There exists at least one $ta_i \in s$ such that WriteMaj(ta_i)= $\top$) Let p_h be primary executing the first ta_i such that WriteMaj(ta_i)= $\top$ at statement 11. This implies $ta_i.\#seq = ta.\#seq$. From Lemma 4, it follows $ta_i \in maj\_state_k.TA$. Therefore (ii) is no longer true.

□

5.2 Theorems

Theorem 1 (P1). If $ta \in state_{p_i}.TA$ then *there exists a client c that issued a request identified by req_id and* $req\_id = ta.req\_id$.

Proof. From Lemma 1, it follows that $ta \in state_{p_i}.TA$ only if there exists $p_j (j \leq i)$ that executes WriteMaj(ta) at statement 11. The latter statement is executed by a primary only after the receipt of a client request (line 6) identified by req_id and after $ta.req\_id$ has been set to req_id at line 10. □

Theorem 2 (P2). If *a client c delivers a reply* $\#seq$, then *it exists a primary* $p_i \in \mathcal{P}$ *and a tentative assignment ta such that* $ta = \langle req\_id, \#seq \rangle \in state_{p_i}.TA$.

Proof. By contradiction. Suppose a client c delivers a reply $\#seq$ and $\nexists p_i \in \mathcal{P} : ta \in state_{p_i}.TA$ and $ta.\#seq = \#seq$. If c delivers a reply $\#seq$, from channel reliability assumption, it has been sent by a primary p_j that executed either statement 8 or statement 12. In both cases $ta \in state_{p_j}.TA$ (either ta already belongs to the p_j's state, statement 7, or from Observation 1 as p_j executed WriteMaj(ta)= $\top$ at statement 11). Therefore in both cases $ta.\#seq = \#seq$. This contradicts the initial assumption. □

Theorem 3 (P3). *Let* p_i *be the current primary*

$$\forall ta_i, ta_j \in state_{p_i}.TA : ta_i.\#seq \neq ta_j.\#seq \Leftrightarrow ta_i.req\_id \neq ta_j.req\_id$$

Proof. Let ta_i and ta_j be two assignments belonging to $state_{p_i}.TA$. From Lemma 7, follows that both ta_i and ta_j are definitive assignments and then two primaries exist, say $p_a, p_b \in \mathcal{P}$, such that p_a executed WriteMaj(ta_i)= $\top$ and p_b that executed WriteMaj(ta_j)= $\top$. Without loss of generality, we assume WriteMaj(ta_i)= $\top$ is executed before WriteMaj(ta_j)= $\top$.

- ($\Rightarrow$). By contradiction. Suppose $ta_i.\#seq \neq ta_j.\#seq$ and $ta_i.req\_id = ta_j.req\_id$. From Lemma 4 and from Observation 1 it follows $\forall p_c \in \mathcal{P}, c \geq a \Rightarrow ta_i \in state_{p_c}.TA$. Then each primary p_c receiving requests identified by $req\_id = ta_i.req\_id$ will execute statements 7 and 8, and will never execute statement 11. This implies that WriteMaj(ta_j) with $ta_j.\#seq \neq ta_i.\#seq$ and $ta_j.req\_id = ta_i.req\_id$ can never be executed. Contradiction.

- ($\Leftarrow$). By contradiction. Suppose $ta_i.\#seq = ta_j.\#seq$ and $ta_i.req\_id \neq ta_j.req\_id$. If p_a continues serving requests it cannot assign the same sequence number to other requests because it increases the *seq* variable (statement 10) before executing statement 11. Otherwise if p_a stops serving requests From Lemma 4 it follows $\forall p_c \in \mathcal{P}, c > a \Rightarrow ta_i \in state_{p_c}.TA$ before p_c starts serving requests. Hence when p_c executes statement 26, it holds $seq_{p_c} \geq ta_i.\#seq$. Therefore at statement 10 it holds $ta.\#seq > ta_i.\#seq$. Then, when p_b executes WriteMaj(ta_j), we get the following absurd $ta_j.\#seq > ta_j.\#seq$.

□

Theorem 4 (P4). *Let p_i be the current primary*

$$\forall ta_i \in state_{p_i}.TA : ta_i.\#seq \geq 1 \wedge ta_i.seq > 1 \Rightarrow \exists ta_j \in state_{p_i}.TA : ta_j.\#seq = ta_i.\#seq - 1$$

Proof. By contradiction. First let us assume that $\exists ta_i.\#seq < 1$. As at line 5 the *seq* variable is initialized to 0 and each primary before executing a tentative assignment (statement 10) executes an increment of *seq* (statement 9), there cannot exist a tentative assignment ta_i in a state of some replica with $ta_i.\#seq < 1$. Contradiction.

Then suppose that $\exists ta_i \in state_{p_i}.TA : ta_i.\#seq > 1$ and $\nexists ta_j \in state_{p_i}.TA : ta_j.\#seq = ta_i.\#seq - 1$. From Lemma 1 exists a primary $p_k, k \leq i$ that executed WriteMaj(ta_i) at statement 11. Lemma 6 implies that it exists a definitive assignment ta_j such that $ta_j.\#seq = ta_i.\#seq - 1$. If WriteMaj($ta_j$)=$\top$ has been executed by p_h with $h < i$ then from Lemma 4 follows that $ta_j \in state_{p_i}.TA$. Otherwise if $h = i$, $ta_j \in state_{p_i}.TA$ from Observation 1. In both cases we get a contradiction.

□

Theorem 5 (P5). If *a client c issues a request identified by req_id,* then*, unless c crashes it eventually delivers a reply #seq.*

Proof. By contradiction. Let us assume that c does not crash and invokes the GetSeq() method of class Client (Figure 1 at page 4) and never receives a reply. c eventually sends the request identified by *req_id* to every replica $r_i \in \{r_1, ..., r_n\}$. From the channel reliability assumption and the global stabilization assumption, it will eventually exist a primary p_k which will receive the *req_id* request, generate a reply *#seq* and send the reply back to the client. From channel reliability, the reply will eventually reach the client. Contradiction. □

6 Conclusions

In this paper we have presented the specification of a sequencer service which allows thin, independent clients to get a unique and consecutive sequence number in order to label successive operations. We have then shown a fault-tolerant sequencer implementation based on a primary-backup replication scheme which uses a timed asynchronous datagram service to communicate among replicas. The implementation shows good performance in failure free runs as only a majority of replicas needs to receive primary updates. The formal correctness proof of the implementation with respect to the specification has been also given.

The sequencer service allows to synchronize processes in open distributed architectures. It can be then used for getting a total order of operations coming from various (and unknown) clients. As a consequence, this service can be used as a basic building block for solving problems like mutual exclusion and replication in such open settings.

We plan to implement the presented fault-tolerant sequencer as a CORBA Service ([17]) in the context of our Interoperable Replication Logic project ([14]), whose main aim is to provide a Fault-Tolerant CORBA compliant ([18]) testbed platform for three-tier replication protocols ([1]). We are also working on a design of an implementation of a fault-tolerant sequencer in an asynchronous system with unreliable failure detectors ([6]).

References

[1] R. Baldoni and C. Marchetti, *Software replication in three-tiers architectures: is it a real challenge?*, Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'2001) (Bologna, Italy), November 2001, pp. 133–139.

[2] K. Birman, *The Process Group Approach to Reliable Distributed Computing*, Communications of the ACM **36** (1993), no. 12, 37–53.

[3] K. Birman and T. Joseph, *Reliable Communication in the Presence of Failures*, ACM Transactions on Computer Systems **5** (1987), no. 1, 47–76.

[4] K. Birman, A. Schiper, and P. Stephenson, *Lightweight Causal and Atomic Group Multicast*, ACM Transactions on Computer Systems **9** (1991), no. 3, 272–314.

[5] N. Budhiraja, F.B. Schneider, S. Toueg, and K. Marzullo, *The Primary-Backup Approach*, ch. 8, pp. 199–216, Addison Wesley, 1993.

[6] T. Chandra and S. Toueg, *Unreliable Failure Detectors for Reliable Distributed Systems*, Journal of the ACM (1996), 225–267.

[7] Pascal Felber, *Lightweight Fault Tolerance in CORBA*, Proceedings of the International Symposium on Distributed Objects and Applications (DOA'01) (Rome, Italy), September 2001, pp. 239–247.

[8] C. Fetzer and F. Cristian, *A Fail-aware Datagram Service*, IEEE Proceedings - Software Engineering **146** (1999), no. 2, 58–74.

[9] ———, *A Highly Available Local Leader Election Service*, IEEE Transactions on Software Engineering **25** (1999), no. 5, 603–618.

[10] ———, *The Timed Asynchronous Distributed System Model*, IEEE Transactions on Parallel and Distributed Systems **10** (1999), no. 6, 642–657.

[11] M. Fischer, N. Lynch, and M. Patterson, *Impossibility of Distributed Consensus with One Faulty Process*, Journal of the ACM **32** (1985), no. 2, 374–382.

[12] R. Guerraoui and S. Frolund, *Implementing E-Transactions with Asynchronous Replication*, IEEE Transactions on Parallel and Distributed Systems **12** (2001), no. 2, 133–146.

[13] R. Guerraoui and A. Shiper, *Software-Based Replication for Fault Tolerance*, IEEE Computer - Special Issue on Fault Tolerance **30** (1997), 68–74.

[14] IRL Project Web Site, *http://www.dis.uniroma1.it/∼irl*.

[15] C. Marchetti, A. Virgillito, and R. Baldoni, *Design of an Interoperable FT-CORBA Compliant Infrastructure*, Proceedings of the 4th European Research Seminar on Advances in Distributed Systems (ERSADS01) (Bertinoro (Bologna), Italy), May 2001, pp. 155–160.

[16] L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, R.K. Budhia, and C.A. Lingley-Papadopoulos, *Totem: A Fault-Tolerant Multicast Group Communication System*, Communications of the ACM **39** (1996), no. 4, 54–63.

[17] Object Management Group (OMG), Framingham, MA, USA, *CORBAServices: Common Object Services Specification*, OMG Document formal ed., July 1997, OMG Final Adopted Specification.

[18] Object Management Group (OMG), Framingham, MA, USA, *Fault Tolerant CORBA Specification, V1.0*, OMG Document ptc/2000-12-06 ed., April 2000, OMG Final Adopted Specification.

[19] Object Management Group (OMG), Framingham, MA, USA, *The Common Object Request Broker Architecture and Specifications. Revision 2.4.2*, OMG Document formal ed., February 2001, OMG Final Adopted Specification.

[20] M. Raynal, *Algorithms for Mutual Exclusion*, MIT Press, 1986.