

Chapter 1

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 a sketch of the correctness proof.

Keywords: Synchronization, Open Distributed Systems, Timed Asynchronous model.

Introduction

Since the middle of 80s, the abstraction of process group has been of primary importance in designing fault-tolerant distributed applications

(Birman and Joseph, 1987). A group is a set of cooperating processes able to order events (such as message receipts, failures, recoveries etc.) identically by using specific group operations and mechanisms such as broadcast communication primitives, views operations, state transfer, failure detection mechanisms etc. (Birman et al., 1991). 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.

Recently, distributed systems are moving towards *open* distributed architectures in which *loosely coupled* and *independent* entities cooperate in order to meet a common goal. In particular, these environments generally assume very thin clients (embedding at most a redirection/retransmission mechanism) that interact with more thick entities. *Three-tier architectures* (e.g. Baldoni and Marchetti, 2001; Guerraoui and Frolund, 2001) and open *middleware platforms* (e.g. OMG, 2001) are examples of such open distributed systems.

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 and replication (for nice surveys about these topics refer to Raynal, 1986 and Guerraoui and Shiper, 1997 respectively). 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 replica¹.

In the context of a closed group, the solutions proposed for this class of problems are mainly either based on a virtual synchronous environment fully distributed (Birman and Joseph, 1987) or token-based (Moser et al., 1996). Both solutions do not fit well open architectures as clients should implement functionalities much more complex than a simple retransmission/redirection mechanism becoming, thus, thick entities. These reasons make appealing the service approach for synchronization in such architectures.

which independent processes can obtain a sequence number to globally order successive events they generate

In this chapter 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 globally order successive relevant events they generate. Such a specification allows different processes to get a different sequence number for each distinct relevant events despite, for example, multiple receipts of the same request by the sequencer. Moreover, the sequence number associated by the sequencer to each request has to be consecutive.

Then, we provide 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 (Fetzer and Cristian, 1999a). 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, due to lack of space, we provide a sketch of the correctness of our implementation with respect to the sequencer specification. The interested readers can refer to (Baldoni et al., 2001) for the formal correctness proof.

1. 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 is 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 requests do not have “holes” in a sequence starting from one. Property (P5) expresses the fact that the service is live.

2. 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* (Fetzer and Cristian, 1999c; Fetzer and Cristian, 1999a).

2.1 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 are *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 :=  $\langle r_1, \dots, r_n \rangle$ ;
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 :=  $\langle cl\_id, \#cl\_seq \rangle$ ;
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  $r \in rlist$ ) 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

2.2 Replica Processes

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 $[s, t]$ is 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* (Fetzer and Cristian, 1999a) 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 (Fetzer and Cristian, 1999b). 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?*(\cdot) boolean function returns *true*;
- the leader election protocol underlying the *Leader?*(\cdot) boolean function takes at least 2δ for a leader change;
- when a majority of replicas forms a stable partition in a time interval $[t, t + \Delta t]$ ($\Delta t \gg 2\delta$), 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 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 $\mathcal{S} \subseteq \{r_1, \dots, r_n\} : |\mathcal{S}| \geq \lceil \frac{n+1}{2} \rceil$ 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 crash² and (ii) there will eventually exist a leader replica in \mathcal{S} .

3. 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 (Budhiraja et al., 1993).

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 1.*

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

1. The availability of the leader election service running among replicas (see Section 2). 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 1.

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.

3.1 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 $\{ta_1, \dots, ta_k\}$ 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.

- 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*.

3.2 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.

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 duration³ $T = 2\delta(1 + \rho)$ 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 leader at the end of the invocation (line 17).

Let us finally present the following observations that will be used in Section 4 to show the correctness of our protocol:

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

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

```

1  BOOLEAN WRITEMAJ(MSG msgtosend)
2  begin
3    BOOLEAN succeeded := ⊥;
4    INTEGER i := 0;
5    if (Leader?())
6      then TA-broadcast ([“Write”, msgtosend]);
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 ≥ ⌈ $\frac{n+1}{2}$ ⌉)
13           then succeeded := ⊤;
14             if (recmsg is Assignment)
15               then state.TA := state.TA ∪ recmsg;
16               else state.epoch := recmsg;
17         return (succeeded and Leader?())
18       end loop
19  end

```

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

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

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:

Definition 1. A tentative assignment ta is a definitive assignment iff exists a primary p such that p executed WRITEMAJ(ta) = ⊤.

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 1). However, by filtering out non-definitive assignments during the *computing_sequencer_state* procedure, we let the state of a primary contain only definitive assignments. Thus we enforce the bijection property only on definitive assignments (Theorem 3 in Section 4).

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 3.1. 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-broadcast()** (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 4 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 $maj_state.epoch \geq \max\{e' : \text{some replica executed } \text{WRITEMAJ}(e') = \top\}$.*

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

3.3 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).

```

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); %  $T = H() + 2\delta(1 + \rho)$  %
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 ( $i \geq \lceil \frac{n+1}{2} \rceil$ )
15            then succeeded := ⊤;
16          return (succeeded and Leader?(sender), 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  $r_i \neq sender$ )
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

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:

1. 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.
2. 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 1) by guaranteeing that when *there are multiple assignments with the same sequence number, the one with the greatest epoch number is a definitive assignment*. The filter is shown in Figure 7 from line 23 to line 25.

3. to impose a primary epoch number e by using a `WRITEMAJ()` function. Epoch number 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.

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_id1`, 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_id2` $\neq req_id_1$, 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).

Replica 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_id2`, 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_id1`. Then, upon the arrival of a new request `req_id2`, it invokes `WRITEMAJ()`, exhibits a performance failure

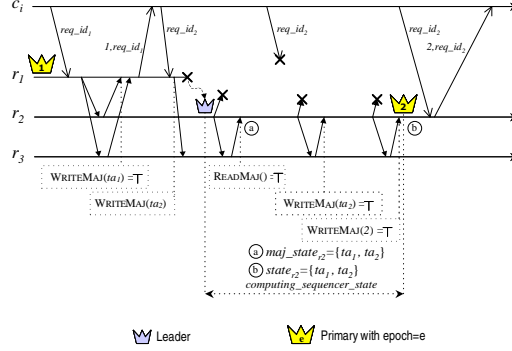


Figure 5. Example of a Run of the Sequencer Protocol

and updates only replica r_3 (ta_2 is a non-definitive assignment). As a consequence r_1 loses 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.

3.4 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);

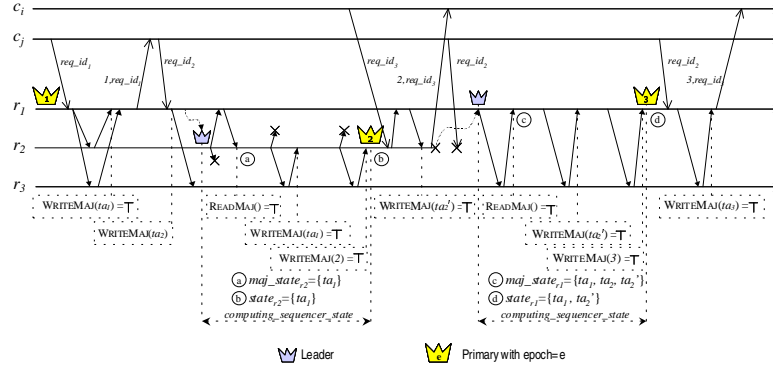


Figure 6. Example of a Run of the Sequencer Protocol

```

CLASS SEQUENCER
1  TENTATIVE ASSIGNMENT  $ta$ ;
2  STATE  $state := (\emptyset, 0)$ ;
3  BOOLEAN  $primary := \perp$ ;  $connected := \perp$ ;
4  INTEGER  $seq := 0$ ;
5  loop
6    when ((A-deliver ["GetSeq",  $req\_id$ ] from  $c$ ) and  $primary$ ) do
7      if ( $\exists ta' \in state.TA : ta'.req\_id = req\_id$ )
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  $req\_id.cl\_id$ ;
13         else  $primary := \perp$ ;
14     when (not Leader?()) do
15        $primary := \perp$ ;
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 ( $ta \neq null$ )
21           then  $connected := WriteMaj (ta)$ ;
22           if ( $connected$ )
23             then for each  $ta_j, ta_\ell \in maj\_state.TA$  :
24                   ( $ta_j.\#seq = ta_\ell.\#seq$  and ( $ta_j.\#epoch > ta_\ell.\#epoch$ ))
25                   do  $maj\_state.TA := maj\_state.TA - \{ta_\ell\}$ ;
26                    $state.TA := maj\_state.TA$ ;  $seq := last(state.TA).\#seq$ ;
27             if (WriteMaj ( $maj\_state.epoch + 1$ ) and  $connected$ )
28               then  $primary := \top$ ;
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 $primary = \perp$ (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 *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 variable⁴.
- 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.

4. Sketch of the Correctness Proof

In this section, due to lack of space we show a sketch of the correctness proof. The formal correctness proof can be found in (Baldoni et al., 2001). This sketch shows that our sequencer implementation satisfies the properties defined in Section 1. 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 .

Definition 2. A primary sequence $\mathcal{P} = \langle p_1, \dots, p_k \rangle$ is a sequence of replica identifiers r_i where p_i represents the i -th replica executing statement 28.

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. The existence of a tentative assignment $ta \in state_{p_i}.TA$ implies that a WRITEMAJ(ta) has been executed at line 11 by a primary p_j , ($j \leq i$). 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. \square

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. \square

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. As ta_i and ta_j are two assignments belonging to $state_{p_i}.TA$, they are definitive assignments as informally shown in the two examples of

section 3.3. As $ta_i, ta_j \in state_{p_i}.TA$ then two primaries exist such that executed respectively $WRITEMAJ(ta_i) = \top$ and $WRITEMAJ(ta_j) = \top$. Without loss of generality, we assume $WRITEMAJ(ta_i) = \top$ is executed before $WRITEMAJ(ta_j) = \top$ by a primary p . As p increases the seq variable (statement 9) each time it executes a new assignment it follows that two assignments with the same sequence number have never been executed. Moreover p contains in its state ta_i (Observation 1). This implies that when p receives a client request such that $req_id = ta_i.req_id$, it will execute always lines 7-8. It follows that it never executes a new assignment with the same req_id .

As ta_i is definitive, if p stops serving client requests the successive primaries contain ta_i in their states before they start serving requests. This derives from Observation 4 and from the fact that a definitive assignment is never filtered out at the statement 25 as we shown in example 2 section 3.3. As a consequence any primary successive to p never executes a new assignment ta with $ta.req_id = ta_i.req_id$ or $ta.\#seq = ta_i.\#seq$. \square

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. 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), it follows that for each assignment executed by a primary $ta_i.\#seq \geq 1$.

The existence of an assignment $ta_i \in state_{p_i}.TA$ implies that a $WRITEMAJ(ta_i)$ has been executed at line 11 by a primary p_j , ($j \leq i$). We have two cases:

(p_j executes statement 11 for the first time). Being primary, p_j completed *computing_sequencer_state* procedure (lines 17–29). This implies that it has previously executed a $WRITEMAJ(ta_j) = \top$ with a given $ta_j.\#seq$ (line 21). As p_j , before executing line 11, executes line 9, it increases seq by one, hence $ta_j.\#seq = ta.\#seq - 1$.

(p_j already executed statement 11 at least one time). In this case p_j has previously executed $WRITEMAJ(ta_j = \top)$ at statement 11 with a given $ta_j.\#seq$. When p_i executes $WRITEMAJ(ta_i)$ at statement 11, from line 9, it follows $ta.\#seq = ta'.\#seq + 1$.

As $WRITEMAJ(ta_j) = \top$ from Definition 1 it follows that is a definitive assignment. From Observation 4 and from the fact that a definitive assignment is never filtered out at the statement 25 as we shown in example 2 section 3.3, it follows that $ta_j \in state_{p_i}.TA$. \square

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 5) and never receives a reply. c eventually sends the request identified by req_id to every replica $r_i \in \{r_1, \dots, r_n\}$. 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. \square

5. Conclusions

In this paper we presented the specification of a sequencer service which allows thin, independent clients to get a unique and consecutive sequence number in order to globally order successive relevant events they generate. 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. A sketch of the correctness proof of the implementation, with respect to the specification, has been also given.

Notes

1. In an asynchronous distributed system where processes can crash this problem cannot be solved (unless a minimum degree of synchrony is added to the system) as it is equivalent to solve the consensus problem (Fischer et al., 1985).
2. Note that at any given time t' (with $t' < t$) any number of replicas can simultaneously suffer a performance failure.
3. 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.
4. Due to the time taken by the the leader election protocol (Fetzer and Cristian, 1999b) (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().

References

- (2001). *The Common Object Request Broker Architecture and Specifications. Revision 2.4.2*. Object Management Group (OMG), Framingham, MA, USA, OMG Document formal edition. OMG Final Adopted Specification.
- Baldoni, R. and Marchetti, C. (2001). Software replication in three-tiers architectures: is it a real challenge? In *Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FT-DCS'2001)*, pages 133–139, Bologna, Italy.

- Baldoni, R., Marchetti, C., and Tucci-Piergiovanni, S. (2001). Fault Tolerant Sequencer: Specification and an Implementation. Technical Report 27.01, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", <http://www.dis.uniroma1.it/~baldoni/publications>.
- Birman, K. and Joseph, T. (1987). Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76.
- Birman, K., Schiper, A., and Stephenson, P. (1991). Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314.
- Budhiraja, N., Schneider, F., Toueg, S., and Marzullo, K. (1993). *The Primary-Backup Approach*, chapter 8, pages 199–216. Addison Wesley.
- Fetzer, C. and Cristian, F. (1999a). A Fail-aware Datagram Service. *IEEE Proceedings - Software Engineering*, 146(2):58–74.
- Fetzer, C. and Cristian, F. (1999b). A Highly Available Local Leader Election Service. *IEEE Transactions on Software Engineering*, 25(5):603–618.
- Fetzer, C. and Cristian, F. (1999c). The Timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657.
- Fischer, M., Lynch, N., and Patterson, M. (1985). Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382.
- Guerraoui, R. and Frolund, S. (2001). Implementing E-Transactions with Asynchronous Replication. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):133–146.
- Guerraoui, R. and Shiper, A. (1997). Software-Based Replication for Fault Tolerance. *IEEE Computer - Special Issue on Fault Tolerance*, 30:68–74.
- Moser, L., Melliar-Smith, P., Agarwal, D., Budhia, R., and Lingley-Papadopoulos, C. (1996). Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63.
- Raynal, M. (1986). *Algorithms for Mutual Exclusion*. MIT Press.