Asynchronous Active Replication in Three-tier Distributed Systems *

Roberto Baldoni, Carlo Marchetti and Sara Tucci Piergiovanni
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italia.
{baldoni,marchet,tucci}@dis.uniroma1.it

Abstract

The deployment of server replicas of a given service across an asynchronous distributed system (e.g. Internet) is a real practical challenge. This target cannot be indeed achieved by *classical software replication techniques* (e.g. passive and active replication) as these techniques usually rely on *group communication toolkits* that require server replicas to run over a *partially synchronous distributed system*. This paper proposes a *three-tier architecture* for software replication that encapsulates the need of partial synchrony in a specific software component of a *mid-tier* to free replicas (end-tier) and clients (client-tier) from the need of underlying partial synchrony assumptions. Then we propose how to specialize the mid-tier in order to manage active replication of server replicas.

1 Introduction

Active [27], passive [5], semi-passive [12] and quorum replication [19, 25] are well-known approaches used to keep consistent the internal states of a set of server replicas connected by a communication network. A replicated "*stateful*" service should indeed appear to clients as implemented by a single logical entity. Informally, consistency means that all replicas have to execute the same sequence of updates before failing. This can be obtained by combining (i) request ordering and (ii) atomic updates.

In order to maintain consistency despite replica failures, these approaches commonly employ group communication primitives and services such as *total order multicast*, *view synchronous multicast*, *group membership*, etc.. For example, in active replication clients and deterministic replicas employ a total order multicast primitive to ensure consistency. However such primitives to be implementable require replicas (and in some case even clients, e.g., [26]) to be deployed in a *partially synchronous* distributed system, as they rely on distributed agreement protocols[1]. A partially synchronous system is an asynchronous distributed system with some additional timing assumptions on message transfer delays, process speeds etc. e.g., the timed asynchronous model [17] and the asynchronous system with unreliable failure detectors [8]. Partial synchrony can be guaranteed (most of the time) only on small size

*Technical Report #05-02, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", February 2002

[1]It is well known that these primitives are not implementable in an asynchronous distributed system due to FLP impossibility result [18].

distributed systems deployed over either a LAN or a CAN (Controlled Area Network). This makes then impossible the deployment of server replicas over asynchronous distributed systems. This limits the dissemination of such replicated services across the Internet.

In this paper we first introduce two architectural properties, namely *Client/Server-Asynchrony* and *Client-Autonomy*, that when satisfied by a replication technique respectively allow (i) to deploy clients and server replicas (implementing a stateful service) within an asynchronous distributed system, and (ii) to avoid clients be involved in maintaining replica consistency. Then we propose, in the context of active replication, a three-tier architecture in which clients (the client-tier) interact with a middle tier (the mid-tier) that forwards client requests to server replicas (the end-tier) and it is responsible of maintaining consistency[2]. To achieve this, the mid-tier embeds two basic components, namely the *sequencer* and the *active replication handler*. The first component is only concerned with defining a total order on client requests, while the second masters the client/server interaction enforcing atomicity on the end-tier. This clear separation of concerns allows to *isolate the need of partial synchrony assumptions within the sequencer component*. As a consequence, clients and servers do not need any underlying synchrony assumption, i.e. this architecture satisfies *Client/Server-Asynchrony*. Moreover, *Client-Autonomy* is also satisfied as the only entity responsible of maintaining end-tier consistency is the mid-tier.

Finally we propose an implementation of the active replication handler, working on an asynchronous distributed system. The active replication handler receives a request from a client and gets a sequence number from the sequencer. Then it forwards the request with its sequence number to all end-tier server replicas. The latter execute requests according to their sequence numbers, computing the result. The active replication handler waits for the first result from end-tier and sends it back to client. As mid-tier components are involved in every client-server interaction, they are replicated in order that if an entity of a mid-tier component that was carrying out a client/server interaction crashes, another entity can conclude the job.

The remainder of this paper is organized as follows: Section 2 introduces the *Client/Server-Asynchrony* and *Client-Autonomy* architectural properties for software replication, illustrates active replication and defines its correctness conditions; Section 3 introduces a three-tier architecture that allows to satisfy these architectural properties; Section 4 details a mid-tier protocol whose correctness is shown in Section 5. Finally, Section 6 concludes the paper.

2 Software Replication

Software replication is a well known technique allowing to increase the availability of a service. The basic idea underlying software replication is to replicate a server on different sites of a communication network so that the service's clients can access different *replicas* in order to increase the probability of obtaining a reply to their requests. When the server is *stateless* (i.e., a result to a request depends only on the current request message content) improving service availability reduces to address the issue of letting the client (more or less) transparently invoke all of the replicas before returning a "no-response" exception. On the other hand, i.e. when dealing with servers maintaining an internal state (*stateful* servers), it arises the problem of maintaining consistency among the state of the replicated server to let clients interact with the stateful replicated server as if it was a singleton, highly available, object.

In this section we introduce two architectural properties for software replication and then we deal with the active replication technique, providing a set of correctness conditions that ensure live and consistent client/server

[2]Active, Passive, Semi-Passive and Quorum replication are actually examples of two-tiers (2T) software replication as clients (client-tier) *directly* interact with a set of replicas (end-tier).

interactions.

2.1 Architectural Properties for Software Replication

Apart from guaranteeing consistency, clients and server replicas implementing a replication technique should also enjoy the following architectural properties to match the nature of current distributed systems (e.g. client/server architectures, n-tiers architectures):

(P1) Client/Server-Asynchrony. A client (resp. a server replica) runs over an underlying distributed system without any timing assumption (i.e., an asynchronous distributed system);

(P2) Client-Autonomy. A client has not to participate in any coordination (distributed) protocol involving other clients and/or server replicas with the aim of maintaining server replica consistency.

P1 implies that clients and server replicas of a stateful service can be deployed across a system like the Internet. P2 allows to implement very thin clients that matches current technologies. It also widens the spectrum of devices that could host such clients (e.g. mobile phones, PDAs, etc).

Two-tier replication techniques usually do not satisfy P1 and in some case also P2, as they usually assume replicas to run over a partially synchronous distributed system and sometimes involve clients in maintaining server replica consistency. We address such issues in the context of active replication in the following subsections.

2.2 Active Replication

Active replication [27, 20] can be seen as a two-tier technique: it takes into account two types of players, namely clients and a set of *deterministic* server replicas that manage the replication protocol. As shown in Figure 1, a client sends a request to a group of deterministic replicas. Each replica independently executes the request and sends back the reply. We abstract the computation of a request *req* by a generic replica through the *compute(req)* method that returs a result (*res*). Assuming a crash failure model for processes, the client waits only for the first reply and filters out others.

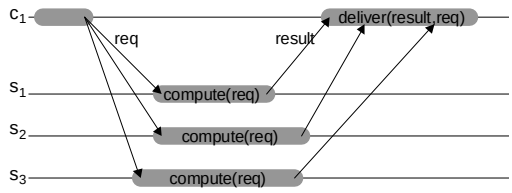

Figure 1: Active Replication technique

2.3 Correctness of Active Replication

Ideally, a client would like to interact with a replicated server as if it was a singleton, non-replicated, highly available server. To this aim, a *correct* implementation of an actively replicated deterministic service should satisfy the following properties [11]:

Termination. If a client issues a request, unless it crashes, it eventually receives a reply.

Uniform Agreed Order. If a replica processes a request req as the i-th request, then the replicas that process the i-th request will execute req.

Update Integrity. For any request req, every replica executes $compute(req)$ at most once, and only if a client has issued req request.

Response Integrity. If a client issues a request req and delivers a result res, then res has been computed by some replica which executed $compute(req)$.

Such properties enforce the consistency of an actively replicated service. In particular *Uniform Agreed Order* implies that if replicas execute requests, such requests are executed in a total order[3]. Note that such total order is defined *only on client requests*. The *Termination* property ensures live client/server interactions (i.e., the liveness of the replicated service). The *Update Integrity* property ensures that replicas compute replies only upon receiving for the first time each client request. Finally, the *Response Integrity* property guarantees that the replies provided to clients are generated by the service.

2.4 Implementing Active Replication

To enforce the consistency of a service replicated using active replication, a *total order multicast* primitive (e.g. [4]) is commonly employed. Therefore, in the case of active replication, satisfying the architectural properties of Section 2.1, turns out in checking if the implementation of the total order multicast primitive satisfies those properties.

A large number of total order multicast and broadcast algorithms has been proposed in the literature. An interesting classification can be found in [11][4]. According to this classification, message ordering can be built by:

Senders (e.g. [24, 10, 9, 1, 15]): in this case senders agree on the order to submit requests to the replicas;

Destinations (e.g.[8, 3]): in this case replicas agree on the order to process client requests;

External Processes (e.g. [6, 4, 23, 15]): ordering may involve processes that are neither senders nor destinations.

Algorithms of the first class do not fit the client/server interaction model as clients must synchronize among them to get a total order. This violates *Client-Autonomy*. Furthermore a client crash can prevent the liveness of server replicas.

Algorithms of the second and third classes can be adapted to satisfy *Client-Autonomy*. For instance, they can be adapted to the client/server interaction model by equipping clients with a retrasmission/redirection mechanism to cope with failures of server replicas or external processes. However, these algorithms do not satisfy the *Client/Server-Asynchrony* property as they are designed assuming that processes employ a distributed agreement protocol. Such protocol is implementable only if *all* of these processes run over an underlying distributed system with partial synchrony assumptions [13, 14]. As example, in [4] processes elect a sequencer process that is in charge of defining total order. Election is based on a group membership service that is impossible to implement in asynchronous distributed systems [7].

[3] As server replicas are assumed deterministic, if they execute *all* requests in the same order then they will produce the same result for each request. This satisfies *linearizability* [22].

[4] Let us note that many of such algorithms have been designed in the context of closed groups of homogeneous processes ([21]).

3 Three-tier (3T) active replication

3T active replication introduces a middle tier (mid-tier) interposed *between* asynchronous clients (client-tier) and asynchronous server replicas (end-tier). The mid-tier is in charge of accepting client requests, enforcing a total order on them and of forwarding requests to the end-tier that returns a result to the mid-tier, which finally provides clients with results. Moreover, inside the mid-tier, we separate the problem of agreeing on a total order of client requests (i.e., to satisfy the *Uniform Agreed Order* property) from the problem of letting all server replicas execute these ordered requests. The first problem is addressed by the *sequencer mid-tier component* and the second by the *active replication handler mid-tier component*. To ensure the liveness (i.e., the *Termination* property) of client/server interactions in presence of failures, the mid-tier must be fault tolerant i.e., both the sequencer and the active replication handler components must be replicated.

In the remainder of this section we introduce the system model and presents the 3T architecture for active replication.

3.1 System model

We consider a distributed system formed by a set of processes that communicate by message passing.

Processes can fail by crashing. After a crash event a process stops executing any action. A process is *correct* if it never fails.

Processes communicate through *reliable* asynchronous channels that are modelled by the following properties:

(C1) Channel Validity. If a process receives a message m, then m has been sent by some process.

(C2) Channel No Duplication. Messages are delivered to processes at most once.

(C3) Channel Termination. If a *correct* process sends a message m to a *correct* process, the latter eventually delivers m.

Previous properties actually states that processes run over an *asynchronous distributed system*.

Processes are actually of four disjoint types: a set $\{c_1, \ldots, c_l\}$ of client processes (client-tier), a set $\{h_1, \ldots, h_n\}$ of active replication handler (ARH) replicas (implementing the mid-tier ARH *component*[5]), a set $\{r_1, \ldots, r_m\}$ of deterministic server replicas (end-tier) and a single *correct* process, belonging to the mid-tier, namely the *Seq* component. The latter component actually abstracts the fault-tolerant implementation of the sequencer service whose system model assumptions will be discussed in Section 4.3.

We finally assume:

(A1) ARH Correctness. A majority of ARH replicas (i.e., $\lceil \frac{n+1}{2} \rceil$) is *correct*.

(A2) Replica Correctness. There is at least one *correct* end-tier replica.

3.2 The 3T Architecture

Figure 2 shows the components of the three-tier architecture for active replication. In the following we introduce the interfaces and a short functional description of each component.

[5]In this paper we use the term "component" to refer to a set of processes implementing a fault-tolerant service.

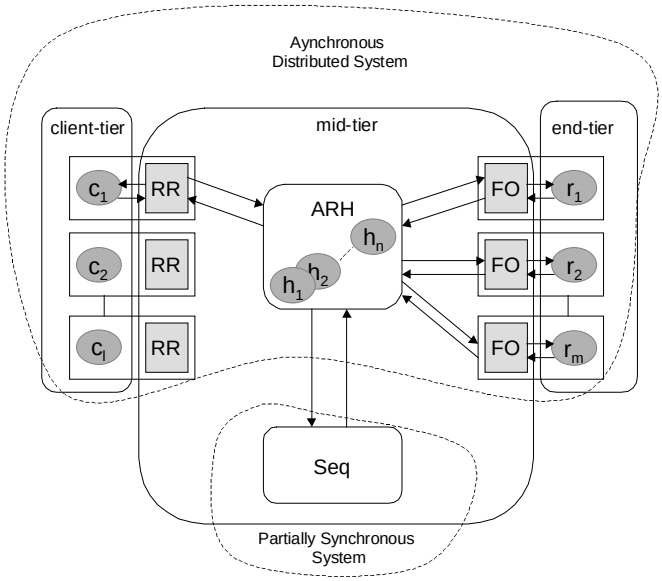

Figure 2: A Mid-tier Architecture for Active Replication

- **Retransmission/Redirection (RR).** To cope with ARH replica failures and with the asynchrony of communication channels (i.e. to enforce the *Termination* property), each client process embeds a RR message handler. This message handler exposes to client applications the IssueReq(*request*) method that accepts a request as input parameter. This method is invoked by client applications to submit requests, and blocks the client process until the result for the request is received. Operationally, RR labels each outgoing client request with a unique request identifier (*req_id*). It then sets a local timeout to transmit the request to a different ARH replica if a reply is not delivered before the timeout expiration, and it continues to issue the request towards different ARH replicas until a reply is eventually received.

- **Sequencer (Seq) component.** The sequencer component is a fault-tolerant service that returns a unique and consecutive sequence number for each *distinct* request. This is the basic building block to get the *Uniform Agreed Order* property. In particular, Seq exposes two methods of the Seq interface:

  - GetSeq(*req_id*) that returns the sequence number associated with *req_id*;
  - GetReqID(*seq*) that returns either the request identifier associated to *seq* (if any) or *null* (otherwise).

- **Active Replication Handler (ARH) component.** The ARH component is the core of the replication logic: by exploiting Seq, it orders all incoming client requests and ensures at least one copy of each client request be eventually delivered at every non-crashed end-tier replica. Once replicas return results, ARH returns them to clients. To achieve *Termination* despite failures, ARH component is implemented by a set of replicas and each replica maintains an internal state composed by a set of $\langle req\_id, request \rangle$ pairs. Each ARH replica has access to two methods:

  - UpdateMaj(*req_id*, *request*) that updates a majority of ARH replica states;
  - FetchMaj() that reads a majority of replica states;

- **Filtering and Ordering (FO).** FO is a message handler placed in front of each end-tier replica (i) to avoid repeated execution of the same client request (possibly sent by ARH) and (ii) to ensure ordered execution of

client requests according to the number assigned by Seq to each request. Previous features allow to enforce *Uniform Agreed Order* and *Update Integrity* on the end-tier replicas.

RR and FO message handlers are therefore co-located with the client and with the end-tier server replica respectively. Hence a crash of a client (resp. server replica) implies the crash of its RR (resp. FO) and vice-versa.

4 The mid-tier protocol

In this section we present the protocol run by the mid-tier to enforce the correctness properties of Section 2.3. To this aim, we first explain an introductory example showing how a client/server interaction is carried out in absence of failures. Then we propose a mid-tier protocol implementation of the components depicted in Figure 2.

4.1 Introductory Example

Figure 3 shows a failure-free run of the mid-tier protocol. In this scenario, a client c_1 invokes RR method IssueReq($request_1$) to issue request $request_1$. This method first assigns a unique request identifier req_id_1 to $request_1$ and then it sends the $\langle req\_id_1, request_1 \rangle$ pair to an ARH replica (e.g., h_1). Upon receiving the $\langle req\_id_1, request_1 \rangle$ pair, h_1 first updates a majority of ARH replicas ($\{h_1, h_3\}$) invoking UpdateMaj($req\_id_1, request_1$). Then h_1 invokes the sequencer GetSeq(req_id_1) method to assign a unique sequence number (1 in the example) to the request (identified by req_id_1). Hence h_1 sends the pair $\langle 1, request_1 \rangle$ to all end-tier replicas and starts waiting for the first result from an end-tier replica. The FO component of each non-crashing end-tier replica checks if the incoming request is the expected one, i.e. if the request sequence number is 1. In the example, the FO component of r_1 and r_2 verifies this condition, and invoke $compute(request_1)$ of its replica that produces a result. Then FO sends to h_1 the pair $\langle 1, result_1 \rangle$. Upon delivering *the first* pair, h_1 sends $\langle req\_id_1, result_1 \rangle$ back to c_1. h_1 discards following results produced for $request_1$ (not shown for simplicity in Figure 3). Then h_1 serves $request_2$ sent by c_2: it updates a majority of ARH replicas ($\{h_1, h_2\}$), gets the sequence number 2 from the sequencer, sends $\langle 2, request_2 \rangle$ to all end-tier replicas and waits for the first reply from end-tier. Note that in this scenario r_3 receives request $\langle 2, request_2 \rangle$ before receiving $\langle 1, request_1 \rangle$. However, upon receiving $\langle 1, request_1 \rangle$, the r_3 FO component executes both the requests in the correct order returning to h_1 both the results. This ensures that the state of r_3 evolves consistently with respect the state of r_1 and r_2. When h_1 receives the first $\langle 2, result_2 \rangle$ pair, it sends the result back to client.

4.2 RR message handler

The RR pseudo-code is shown in Figure 4. RR maintains a state variable $\#cl\_seq$ increased at each new client request to form a unique request identifier along with the client identifier cl_id. To issue a request, a client invokes the IssueReq() method (line 4). Such method blocks the client process and forwards the (uniquely identified) client request to ARH replicas, until a result is received. In detail, the IssueReq() method first assigns to the ongoing request the unique request identifier $req\_id = \langle cl\_id, \#cl\_seq \rangle$ (line 8), and then enters a loop (line 9). Within the loop, the client (i) sends the request to an ARH replica (line 10) and (ii) sets a local timeout (line 11). Then, a result is returned by IssueReq() if the client process receives a result within the timeout period for the request (line 14). Otherwise another replica is selected (line 15) and the request is sent again towards such a replica (line 10).

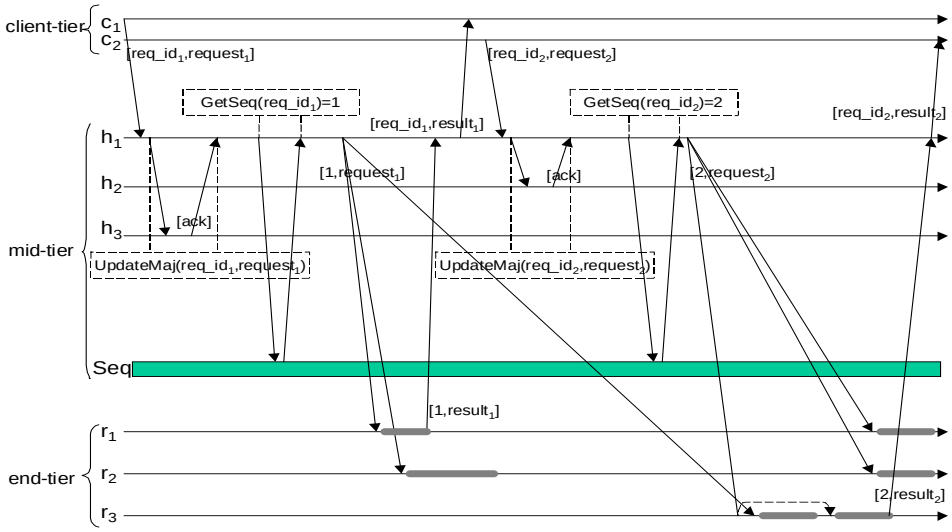

Figure 3: a Failure-free Run

4.3 Seq component

The Seq component is a fault-tolerant implementation of a *sequencer service*. A sequencer service dynamically assigns unique and consecutive sequence numbers to unique request identifiers (defining a total order) and allows to query its internal state to inspect the total order. In particular, it receives *assignment requests* from other processes and returns an integer positive sequence number denoted $\#seq$ for each *distinct* request. A process issues an assignment request invoking the GetSeq(req_id) method where req_id is an unique request identifier. To cope with assignment request retrasmissions, 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 request identifier and $a.\#seq$ is the sequence number returned by the sequencer service.

Formally, a correct sequencer service implementation must satisfy the following properties:

(S1) Assignment Validity. If $a \in A$ then (i) there exists a process p that issued an assignment request GetSeq(req_id) and (ii) $req\_id = a.req\_id$.

(S2) Assignment Response Validity. If a process p delivers a reply $\#seq$, then (i) p issued an assignment request GetSeq(req_id), and (ii) $\exists a = \langle req\_id, \#seq \rangle \in A$;

(S3) Bijection. $\forall a_i, a_j \in A : a_i.\#seq \neq a_j.\#seq \Leftrightarrow a_i.req\_id \neq a_j.req\_id$

(S4) Consecutiveness. $\forall a_i \in A : (a_i.\#seq \geq 1) \wedge (a_i.seq > 1 \Rightarrow \exists a_j : a_j.\#seq = a_i.\#seq - 1)$

(S5) Sequencer Termination. If a process p issues a request then, unless p crashes, it eventually delivers a reply.

A particular sequencer implementation that satisfies the previous specification can be found in [2]. In this implementation sequencer replicas run in the timed asynchronous system model [17] and adopts an optimistic passive replication protocol to maintain replica consistency.

Furthermore, a sequencer service receives from processes *query requests* through the GetReqID($\#seq$) method, which either returns req_id if exists $a_i = \langle req\_id, \#seq \rangle$ or *null* otherwise. We thus introduce the following property to capture the sequencer behavior upon the receipt of a query request:

```
RR message handler
1  hlist := ⟨h_1, ..., h_n⟩;
2  Integer #cl_seq := 0;
3  Identifier req_id;
4  Result IssueReq(request)
5    begin
6      Integer i := 1;
7      #cl_seq := #cl_seq + 1;
8      req_id := ⟨cl_id, #cl_seq⟩;
9      loop
10       send ["Request", ⟨req_id, request⟩] to hlist(i);
11       t.setTimeout := period;
12       wait until ((deliver ["Result", ⟨req_id, result⟩] from h ∈ hlist) or (t.expired()))
13       if (not t.expired())
14         then return (result);
15         else  i := (i + 1) mod |hlist|;
16     end loop
17   end
```

Figure 4: Pseudo-code of RR Message Handler

(S6) Query Response Integrity. If a process p invokes GetReqID($\#seq$) then:

- if $\exists\langle req\_id, \#seq\rangle \in A$ when the sequencer receives a query request then if GetReqID($\#seq$) returns, it returns req_id;
- else if GetReqID($\#seq$) returns, it returns $null$;

The sequencer service can be implemented by a set of server replicas adopting well known replication techniques such as active or passive replication. These techniques employ *group communication primitives* (e.g., total order multicast, view-synchronous multicast etc.) based on distributed agreement protocols. Therefore sequencer replicas have to run within a partially synchronous distributed system, in order to overcome the FLP impossibility result [18].

We provide, in the Appendix, a modified version of the sequencer implementation proposed in [2] that satisfies S1–S6 .

4.4 The Active Replication Handler (ARH) component

The basic idea underlying the ARH protocol is to let each ARH replica enforce end-tier consistency *independently* from other ARH replicas, i.e. each ARH replica forwards to the end-tier every client request at most once (actually, exactly once if the ARH replica is correct).

Figure 5 illustrates the pseudo-code of an ARH replica, based on the following basic data structures and communication primitives.

Data Structures

- the *LocalState* variable is a set of $\langle req\_id, request\rangle$ pairs. *LocalState* is used by ARH replica to hold information on client requests despite client and ARH replica failures. It is read and written by the UpdateMaj and FetchMaj communication primitives (see below).
- the *LastServedReq* variable stores the sequence number associated by Seq to the last client request that the ARH replica has forwarded to the end-tier (returning the result to the client);

- the $\#seq$ variable stores the sequence number associated by Seq to a client request that ARH has received by a client.

Communication Primitives.

- UpdateMaj() accepts as input parameter a $\langle req\_id, request \rangle$ pair p and multicasts p to *all* ARH replicas $\{h_1, \ldots, h_n\}$. An ARH replica upon receiving such an update message, inserts p in its *LocalState* variable and then acknowledges the message receipt. The UpdateMaj() method returns the control to the invoker as soon as it has received $\lceil \frac{n+1}{2} \rceil$ acknowledgement messages;
- FetchMaj() does not take input parameters and returns the union of the $\langle req\_id, request \rangle$ pairs contained in the *LocalState* variables of a strict majority of ARH replicas. Upon invocation, FetchMaj() multicasts a state request message to *all* ARH replicas $\{h_1, \ldots, h_n\}$. An ARH replica, upon receiving a state request message responds with a state message containing the current value of *LocalState*. The FetchMaj() method waits until it has received $\lceil \frac{n+1}{2} \rceil$ state messages, computes the union and returns the latter to the invoker;

An implementation of the above communication primitives is given in the Appendix.

ARH Protocol. Each ARH replica runs the protocol depicted in Figure 5. In particular, upon receiving a message $\langle req\_id, request \rangle$ from a client c (line 4), an ARH replica h_i (i) updates a majority of ARH replicas exploiting the UpdateMaj() primitive[6] (line 5) and (ii) invokes the sequencer to assign a sequence number (stored in the $\#seq$ variable) to the request (line 6). Hence if $\#seq > LastServedReq + 1$ then some other ARH replica served other client requests with sequence number comprised in the interval $[LastServedReq + 1, \#seq - 1]$. In this case, to preserve the protocol termination, h_i sends such requests again to the end-tier. To this aim:

- invokes the FetchMaj() primitive (line 8) to retrieve the union of a majority of local states[7];
- it sends to server replicas all the requests with sequence number comprised $[LastServedReq + 1, \#seq - 1]$, along with their sequence numbers that are retrieved using the sequencer (lines 9-12);

Finally h_i sends to server replicas the current client request along with its sequence number (line 13), updates the *LastServedReq* variable (line 14) and then waits for the first result from a server replica (line 15). Upon the receipt of the result, h_i forwards the result to the client.

4.5 The FO message handler

Figure 6 illustrates the FO message-handler pseudo-code. FO has an internal state composed by (i) the *ExpectedSeq* variable, storing the sequence number associated with next request to execute, and (ii) the *Executed* array, storing in the j-th position the result of the request with sequence number j.

For each receipt of a "*TORequest*" message from an ARH replica h_i (line 3), FO waits until the request sequence number *seq* is lower than or equals *ExpectedSeq* (line 4). It suspends the execution of the following statements until the condition at line 4 holds. When such condition holds FO *atomically* executes statements (5–8). In particular, if the request is the expected request ($seq = ExpectedSeq$, line 5), FO computes the request result, stores the result in *Executed* and increases the *ExpectedSeq* variable. Finally, FO sends a "*TOResult*" message, containing the request sequence number and the request's result, to h_i.

[6] In this way other ARH replicas are able to retrieve *all* client requests and to forward them to the end-tier. This is done to let *all* request messages reach the end-tier despite client and ARH replica failures.

[7] As each client request is stored at a majority of ARH local states (line 5), the execution of FetchMaj() at line 8 store into *LocalState all* the client requests served by ARH replicas.

```
ActiveReplicationHandler
 1  Set LocalState := ∅;
 2  Integer LastServedReq := 0;
 3  Integer #seq;
 4  when (deliver ["Request", ⟨req_id, request⟩] from c) do
 5    UpdateMaj(⟨req_id, request⟩);
 6    #seq := Seq.GetSeq(req_id);
 7    if (#seq > LastServedReq + 1)
 8      then LocalState := FetchMaj();
 9           for each j : LastServedReq < j < #seq
10             do req_id_j := Seq.GetReqID(j);
11                request_j := (request : ⟨req_id_j, request⟩ ∈ LocalState);
12             for each  r_ℓ ∈ {r_1, ..., r_m} do send ["TORequest", ⟨j, request_j⟩] to r_ℓ;
13    for each  r_ℓ ∈ {r_1, ..., r_m} do send ["TORequest", ⟨#seq, request⟩] to r_ℓ;
14    LastServedReq := max(LastServedReq, #seq);
15    wait until (deliver ["TOResult", ⟨#seq, result⟩] from r_k ∈ rlist)
16    send ["Result", ⟨req_id, result⟩] to c;
```

Figure 5: Pseudo-code of an ARH replica h_i

```
FO message handler
1  Array Executed;
2  Integer ExpectedSeq := 1;
3  when (deliver ["TORequest", ⟨seq, request⟩] from h_i) do
4    wait until (seq ≤ ExpectedSeq)
5    if (seq = ExpectedSeq)
6      then result := compute(request);
7           Executed[seq] := result;
8           ExpectedSeq := ExpectedSeq + 1;
9    send ["TOResult", ⟨seq, Executed[seq]⟩] to h_i;
```

Figure 6: Pseudo-code of FO message handler

4.6 Handling Failures

As end-tier server replicas execute requests according to the sequence numbers assigned by the sequencer, if an end tier replica never receives a request with sequence number *seq*, it will never execute requests with sequence number grater than *seq*. Therefore *each* client request associated with a sequence number *must* eventually reach the end-tier to enforce *Termination*.

Figure 7 shows a simple failure free-run of the mid-tier protocol. The run is split into two phases separated by the assignment of a unique sequence number $\#seq$ to a request $req = \langle req\_id, request \rangle$.

The first phase starts with the sending by a client c of a request req to h_1 (denoted A in Figure 7) and ends just before the assignment of a sequence number to the request by the sequencer (denoted B in Figure 7). The second phase starts from B and ends when the client delivers the result (denoted C in Figure 7).

Phase 1. During phase 1 req has not yet been ordered by the sequencer. As a consequence, a failure of h_1 during this phase has no effect on the active replication correctness properties. In such a case, if the client does not crash, it retransmits the request till it reaches a correct ARH replica that will enter phase 2.

Phase 2. The mid-tier protocol ensures that even if both h_1 and c crash in this phase, then another ARH replica will be able to retrieve *req*. In this way, ARH replicas are able to send *all* the requests ordered by the sequencer. In particular *req* will eventually reaches a correct end-tier replica (despite a minority of ARH crashes).

Note that to ensure mid-tier protocol termination, the mid-tier protocol only need one correct end-tier replica.

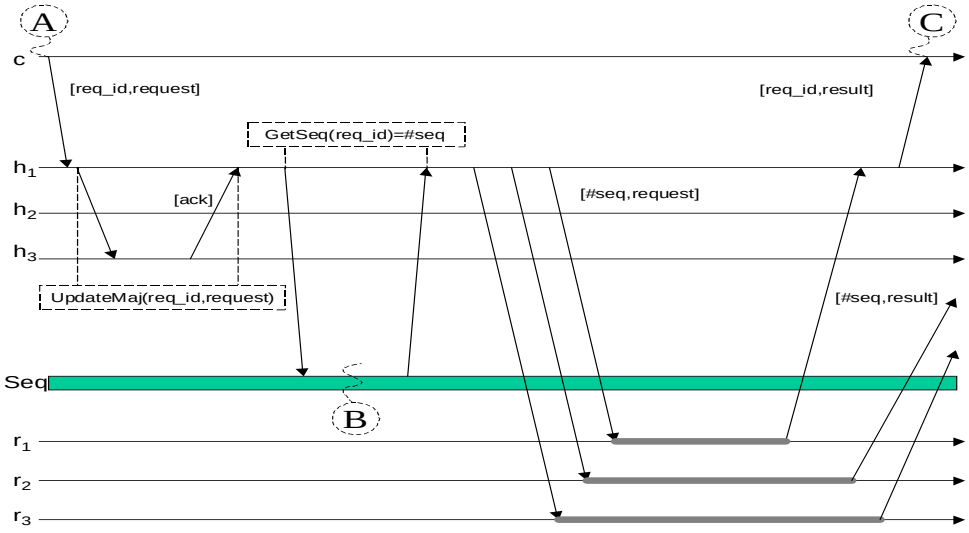

Figure 7: Main Protocol Phases

4.7 Discussion

In this section we emphasize some of the main characteristics of the presented protocol for three-tier replication.

Architectural Properties. The proposed 3T architecture satisfies the*Client/Server-Asynchrony* of Section 2.1 as from the system model clients and servers can be deployed within an asynchronous distributed systems. *Client-Autonomy* is also met as clients do not play any role in enforcing consistency on the end-tier replicas. Clients are only required to be equipped with a retrasmission/redirection mechanism.

On the Minimal Assumption of Partial Synchrony. The aim of the proposed 3T architecture was actually to confine as much as possible the need of partial synchrony. As shown, synchrony is actually needed only for implementing the sequencer service (ARH run on an asynchronous distributed system). Another option would be to merge both ARH and sequencer functionality in a single mid-tier component. This approach eliminates the hop required by ARH replicas to invoke Seq. However, the need of partial synchrony assumption embraces the whole mid-tier.

Possible Optimizations For the sake of simplicity we presented the asynchronous three-tier active replication protocol without dealing with the possible optimizations outlined below.

- *Packing messages sent to the end-tier.* ARH replicas send to the end-tier a distinct "TORequest" message for each client request they have not sent to the end-tier (see Figure 5, lines 9–12) before sending the current client request (line 13). This could actually be done more efficiently by sending to FOs a *single* message containing both the current client request and the requests to send to the end-tier to enforce *Termination*;

- *Packing messages sent to the sequencer.* Each ARH performs a distinct invocation to Seq each time it needs to retrieve the request identifier assigned to a sequence number (line 10). This step could be optimized by generalizing the GetReq() method of the Seq interface.

- *Implementing a selective* FetchMaj(). FetchMaj() currently retrieve the *entire* state of a majority of ARH replicas. However, the information needed by ARH to execute the statements 9–12 consists of only *some* specific request. Therefore FetchMaj() can be specialized to retrieve only a specific sequence of requests.

Exploiting failure detection. Even though the number of messages sent by the mid-tier protocol is large, deploying ARH replicas within an asynchronous distributed systems augmented with *unreliable failure detectors* associated with each ARH [8] does not ensure a message saving. For preserving correctness could indeed suffice to resend to the end-tier replicas only messages sent by crashed ARH replicas. This can be only guaranteed if failure detectors are *perfect* (i.e., they do not make mistakes when considering a process as crashed). Otherwise, if failure detectors have a low accuracy (i.e., they make many mistakes suspecting correct processes as crashed), the number of messages sent to the end-tier replicas could also be bigger than the one generated by the proposed protocol.

5 Correctness Proof

In this section we use the properties C1, C2 and C3 of Section 3.1, S1...S6 of Section 4.3 and the assumption A1 and A2 of Section 3.1 to prove that the mid-tier protocol described in Section 4 satisfies the correctness properties for active replication described in Section 2.3.

Lemma 1. *If an ARH replica h_i has local variable $LastServedReq = k$ then it has sent to all end-tier replicas, k requests $\{request_1, ..., request_k\}$ where each $request_j$ is associated with a sequence number seq_j such that $seq_j = j$.*

Proof. (Sketch.) Assume by contradiction that h_i has local variable $LastServedReq = k$ and it has not sent to end-tier a $request_j$ associated with the $seq_j = j$ such that $1 \leq j \leq k$. By the algorithm of Figure 5 the variable $LastServedReq$ is initialized to 0 at line 2. Suppose without loss of generality that h_i serves its first client request by delivering $request_1$ at line 4. In this case at line 7 $LastServedReq$ equals 0. Hence we assume that $LastServedReq$ is updated to $k > 0$ at line 14. It implies that at line 6 $\#seq$ is set to k. Note that k is returned by GetSeq() that always returns positive values (from S4). We distinguish two cases:

- $k = 1$. This is a trivial case: at line 7 the condition does not hold, then h_i sends $request_1$ to all end-tier replicas along with the sequence number $seq_1 = \#seq = 1$. Contradiction.

- $k > 1$. In this case at line 7 the condition holds, then h_i executes lines 8-12.

  At line 8 h_i assigns to $LocalState$ variable the return value of FetchMaj() (A1 ensures FetchMaj() termination). Hence $LocalState$ contains all the $\langle req\_id, request\rangle$ pairs contained in the local states of a majority of ARH replicas. As every ARH replica invokes UpdateMaj($\langle req_j, request_j\rangle$) after the receipt of every client request $\langle req_j, request_j\rangle$ (line 5) and before invoking the sequencer (line 6), $LocalState$ includes all pairs $\langle req\_id_j, request_j\rangle$ such that GetSeq(req_id_j) has been executed by any ARH replica.

  Then h_i executes the **for** statement (lines 9–12): for each sequence number j such that $1 \leq j \leq k$, the associated req_id_j is obtained invoking GetReqID(j) (line 10). S5 ensures the termination of this invocation. From S4, req_id_j is contained in the sequencer state and then (from S6) the returned value is not $null$. As a consequence, for each $req\_id_j \neq null$ the corresponding $request_j$ can be retrieved from $LocalState$ (line 11)

and the pair $\langle j, request_j \rangle$ is thus sent the to all end-tier replicas (line 12). Therefore, when $LastServedReq$ is set to k (line 14) all requests $\{request_1, ..., request_k\}$ with sequence number $1 \leq seq_j = j \leq k$ are sent to all end-tier replicas. Contradiction.

□

Theorem 1 (Termination.). *If a client issues a request, unless it crashes, it eventually receives a reply.*

Proof. (Sketch.) Assume by contradiction that the client issues a request, it does not crash and it does not deliver a result. It means, by the algorithm of Figure 4, that the client, after sending a request at line 10, never executes deliver at line 12. However, if the deliver is not executed within a time interval a new ARH replica is selected (line 15) and the same request is sent to that replica (line 10). From A1 and C3, a correct ARH replica h_i eventually delivers the client request.

By the algorithm of Figure 5 when h_i receives the request, invokes UpdateMaj() (line 5) that terminates thanks to A1. Then h_i invokes GetSeq() (line 6) that also terminates (S5). This method returns the sequence number $\#seq$ associated with the current request.

Lemma 1 ensures that at line 14 all requests such that their sequence number is lesser than or equal to $LastServedReq$ (including the current request) have been sent to the end-tier replicas.

A2 and C3 guarantee that at least one end-tier replica r_k receives all the requests. From the algorithm shown in Figure 6 these requests are executed according to their sequence numbers (line 4 - 8). Each time a request is executed, $expectedSeq$ is increased by one and this wakes up a suspended instance at line 4 (if any). As line 5 - 8 are executed atomically, no multiple execution of suspended instances with the same $\#seq$ is possible. Therefore, eventually $\#seq = expectedSeq$ and the request identified by $\#seq$ is executed by the end-tier replica (line 6). Finally the result is sent to h_i (line 9). From C3 the result eventually arrives at h_i (Figure 5 line 15), then h_i sends the result to client (line 16) that eventually delivers it (Figure 4 line 12) contradicting the initial assumption. □

Theorem 2 (Uniform Agreed Order). *If an end-tier replica processes a request* request*, identified by* req_id*, as i^{th} request, then every other end-tier replica that processes the i^{th} request executes* request*, identified by* req_id*.*

Proof. (Sketch.) Assume by contradiction that an end-tier replica r_k executes a request $request$, identified by req_id, as i^{th} request and another end-tier replica r_h executes as i^{th} a request $request'$ identified by $req\_id' \neq req\_id$.

From the algorithm shown in Figure 6 the received requests are executed according to their sequence numbers (line 4 - 8). In particular, each time a request is executed, $expectedSeq$ (initialized to 1 at line 2) is increased by one and this wakes up a suspended instance at line 4 (if any). As line 5 - 8 are executed atomically, no multiple execution of suspended instances with the same $\#seq$ is possible. It implies that the i^{th} request processed by r_k, i.e. $request$ identified req_id, is associated with a sequence number $\#seq = i$, i.e. r_k delivered a "$TORequest$" message containing $\langle i, request \rangle$. For the same reasons, r_h delivered a "$TORequest$" message containing $\langle i, request' \rangle$. From the algorithm shown in Figure 5, it follows that two requests with different identifiers have been assigned to the same sequence number by the sequencer. This violates S3. Contradiction. □

Theorem 3 (Update Integrity). *For any request* request*, identified by* req_id*, every end-tier replica executes compute*(request) *at most once, and only if a client has issued request.*

Proof. (Sketch.) First assume by contradiction that an end-tier replica r_k executes twice a single request $request$ (identified by req_id).

From the algorithm of Figure 6 $request$ is delivered by r_k in a "$TORequest$" message along with an associated sequence number seq (line 3). Assume that r_k executes $request$ once. It means that at line 5 $seq = expectedSeq$

and at line 6 r_k executes *compute(request)*. Note that (i) lines 5-8 are executed atomically, (ii) at line 8 the *expectedSeq* variable is incremented by one and (iii) the request *request* identified by *req_id* can never be assigned to a sequence number $seq' \neq seq$ by S3. Therefore even if the *request* identified by *req_id* can delivered again the condition at line 5 can never hold. Hence the same request can never be executed twice. Contradiction.

Assume now by contradiction that *request*, identified by *req_id* has not been issued by a client.

From C1, if r_k delivers a $\langle seq, request \rangle$ in "*TORequest*" message (line 3), the request has been sent by an ARH replica h_i either at line 12 or at line 13 (see Figure 5). If the request has been sent at line 13, h_i has delivered $\langle req\_id, request \rangle$ (line 4). This request has been the sent from a client (from C1). Contradiction. Otherwise, if request has been sent at line 12, the request $\langle req\_id, request \rangle$ is contained in the *LocalState* variable. It means that there exists an ARH replica h_j that has previously executed UpdateMaj($\langle req\_id, request \rangle$). As UpdateMaj($\langle req\_id, request \rangle$) (line 5) is always executed after the delivery of a client request $\langle req\_id, request \rangle$ (line 4) it follows that $\langle req\_id, request \rangle$ has been sent by a client (from C1). Contradiction. □

Theorem 4 (Response Integrity). *If a client issues a request* ⟨req_id,request⟩ *and delivers a reply* ⟨req_id,result⟩*, then* result *has been computed by some end-tier replica, which executed compute*(request)*.*

Proof. (Sketch.) Assume by contradiction that a client issues a request ⟨*req_id,request*⟩ and delivers a reply ⟨*req_id,result*⟩ and *result* has not been computed by an end-tier replica.

From C1, if a client delivers $\langle req\_id, result \rangle$ then an ARH replica h_i has previously sent $\langle req\_id, result \rangle$. From the algorithm of Figure 5, if h_i sends $\langle req\_id, result \rangle$ (line 16) to the client then (i) it received a client request message ⟨*req_id,request*⟩ from the client (line 4), (ii) it assigned a $\#seq$ sequence number to the request (line 6) and (iii) it has successively delivered $\langle \#seq, result \rangle$ from a replica (line 15). From C1, $\langle \#seq, result \rangle$ has been sent by an end-tier replica r_k. From algorithm of Figure 6, r_k sends such $\langle \#seq, result \rangle$ to h_i (line 9) only if $Executed[\#seq] = result$. It implies that the request has been previously executed invoking *compute*(*request*). Contradiction. □

6 Conclusions

Server replicas of a stateless service can be deployed across an asynchronous distributed system. This does not hold if replicas implement a stateful service following classical replication techniques as they need some additional timing assumption on the underlying system connecting the replicas in order to use group toolkits . This for example limits the dissemination of replicated stateful services over the Internet.

In this paper two properties have been defined, namely *Client/Server-Asynchrony* and *Client-Autonomy* that capture the possibility for a replication technique to deploy its replicas across the Internet supporting very thin client. Then we presented a 3T architecture for handling active software replication on the end-tier replicas. The 3T architecture meets the *Client/Server-Asynchrony* property by confining the need of partial synchrony assumptions within a sequencer service detached from clients and server replicas. *Client-Autonomy* property is also satisfied as only the mid-tier is involved in maintaining replica consistency.

Then we presented an implementation of a mid-tier protocol, discussed some optimizations and showed the protocol correctness.

References

[1] Y. Amir, L. Moser, P. M. Melliar-Smith, D. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):93–132, November 1995.

[2] R. Baldoni, C. Marchetti, and S. Tucci-Piergiovanni. Fault Tolerant Sequencer: Specification and an Implementation. Technical Report 27.01, Dipartimento di Informatica e Sistemistica, Universitá di Roma "La Sapienza", http://www.dis.uniroma1.it/˜baldoni/publications, november 2001.

[3] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[4] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[5] N. Budhiraja, F.B. Schneider, S. Toueg, and K. Marzullo. The Primary-Backup Approach. In S. Mullender, editor, *Distributed Systems*, chapter 8, pages 199–216. Addison Wesley, 1993.

[6] R. Carr. The Tandem Global Update Protocol. *Tandem Systems Review*, June 1985.

[7] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the Impossibility of Group Membership. In *Proc. of the 15th ACM Symposium of Principles of Distributed Computing*, 1996.

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

[9] F. Cristian. Asynchronous Atomic Broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116, February 1991. Also: 1st IEEE Workshop on Management of Replicated Data, Houston, TX, November, 1990.

[10] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic Broadcast: From Simple Diffusion to Byzantine Agreement. In *Proc. of the 15th International Conference on Fault-Tolerant Computing*, Austin, Texas, 1985.

[11] X. Dèfago. *Agreement-Related Problems: From Semi Passive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 2000. PhD thesis no. 2229.

[12] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 43–50, West Lafayette, IN, USA, October 1998.

[13] D. Dolev, C. Dwork, and L. Stockmeyer. On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[14] C. Dwork and N.A. Lynch L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[15] P.D. Ezhilchevan, R.A. Macedo, and S.K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing System (ICDCS-15)*, pages 269–306, Vancouver, Canada, May 1995.

[16] C. Fetzer and F. Cristian. A Highly Available Local Leader Election Service. *IEEE Transactions on Software Engineering*, 25(5):603–618, 1999.

[17] C. Fetzer and F. Cristian. The Timed Asynchronous Distributed System Model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[18] M. Fischer, N. Lynch, and M. Patterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[19] D. Gifford. Weighted voting for replicated data. In *Proc. of 7th ACM Symposium on Operating System Principles*, pages 150–162, 1979.

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

[21] V. Hadzilacos and S. Toueg. Faul-Tolerant Broadcast and Related Problems. In S. Mullender, editor, *Distributed Systems*, chapter 16. Addison Wesley, 1993.

[22] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[23] F. Kaashoek and A. S. Tanenbaum. Fault tolerance using group communication. *ACM Operating Systems Review*, 25(2):71–74, April 1991.

[24] L. Lamport. Time, Clocks and the Ordering of Event in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[25] Dahlia Malkhi. Quorum Systems. In Joseph Urban and Partha Dasgupta, editors, *The Encyclopedia of Distributed Computing*. Kluwer Academic Publishers, 2000.

[26] 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(4):54–63, April 1996.

[27] F.B. Schneider. Replication Management using state-machine approach. In S. Mullender, editor, *Distributed Systems*, chapter 7, pages 169–198. Addison Wesley, 1993.

A Communication Primitives

In this section we show the pseudo-code of the communication primitives introduced in Section 3, i.e. UpdateMaj() (Figure 8(a)) and FetchMaj() (Figure 8(b)). Both these primitives are based on the Listener() thread, running in each end-tier processes r_i (Figure 8(c)).

UpdateMaj(). This primitive takes an input parameter $\langle request, req\_id \rangle$ that is sent to each ARH replica (line 3). The Listener() thread of each replica sends an acknowledgement message upon the delivery of this message (see Figure 8(c), line 6). When a majority of acknowledgments has been received (line 4) the primitive exits.

FetchMaj(). This primitive does not take input parameters and returns as output parameter a set of $\langle req\_id, request \rangle$. It sends a state request message to all ARH replicas (line 5). This causes the Listener() thread to send its replica state (*LocalState*) as the reply. (Figure 8(c), line 7). Once a majority of replies has been received, the primitive returns in *ResultSet* variable value containing the union of the *LocalState* variables values received by a majority of ARH replicas (lines 7–11).

B Overview of a Sequencer Service Implementation

In this Section we present a sequencer service fault-tolerant implementation that satisfies S1–S6 (see Section 4.3). The presented implementation run in a set of sequencer replica processes $\{s_1 \dots s_q\}$ running within a timed asynchronous system model [17] and exploiting the leader election service described in [16] to implement an optimistic passive replication scheme. Further details can be found in [2][8].

B.1 GetSeq() and GetReqID()

The GetSeq() and GetReqID() methods are presented in Figure 9. The GetSeq() primitive (Figure 9(b)) accepts as input parameter a unique request identifier *req_id* and returns an integer sequence number assigned by the sequencer to the request. To this aim, it sends a "*GetSeq*" message to a sequencer replica and sets a timeout (lines 5–6). Then if it receives a reply during the timeout period it returns a result, otherwise it continues to invoke different replicas until a reply is eventually received (lines 7–10). The GetReqID() primitive works in the same way, except for accepting a sequence number *seq* as input parameter and for sending "*GetReqID*" messages to

[8] Actually, [2] satisfies S1–S5 while the following implementation differs from the one proposed in [2] for statements 14–17, i.e. for the implementation of the GetReqID() method, in order to satisfy also S6.

```
1  UpdateMaj(⟨req_id, request⟩)
2    begin
3      for each  h_ℓ ∈ {h_1, ..., h_n}  do send ["Update", ⟨req_id, request⟩] to h_ℓ;
4      wait until (deliver ["Ack"] from set ⊆ {h_1, ..., h_n} :| set |= ⌈(n+1)/2⌉)
5    end
```

(a) the UpdateMaj() Pseudo-code

```
1  Set FetchMaj()
2    begin
3      Integer i := 0;
4      State ResulSet := ∅;
5      for each  h_ℓ ∈ {h_1, ..., h_n}  do send ["Fetch", ] to h_ℓ;
6      repeat
7        when (deliver ["LocalState", LocalState_sender] from (h_j)) do
8          ResultSet := ResultSet ∪ LocalState_sender;
9          i := i + 1;
10     until i = ⌈(n+1)/2⌉
11     return (ResultSet);
12   end
```

(b) the FetchMaj() Pseudo-code

```
1  Thread Listener()
2    begin
3      when (deliver [typemsg, recmsg] from h_j) do
4        case  typemsg
5        {"Update"} : LocalState := LocalState ∪ {recmsg};
6                     send ["Ack" ] to h_j;
7        {"Fetch"} :  send ["LocalState", LocalState] to h_j;
8        end case
9    end
```

(c) the Listener() Pseudo-code

Figure 8: ARH Communication Primitives

sequencer replicas. Thus the latter returns a (possibly *null*) request identifier as soon as such message is received by a sequencer replica.

B.2 A Sequencer Implementation

The sequencer protocol pseudo-code executed by s_i is presented in Figure 10. It consists of an infinite loop (lines 5–33) during which four types of events are handled, i.e. (i) the receipt by a primary sequencer replica of a "*GetSeq*" message (lines 6–13), (ii) the receipt by a primary sequencer replica of a "*GetReqID*" message (lines 14–17), (iii) the notification from the underlying leader election service of the loss of the leadership (line 18), and (iv) the notification from the underlying leader election service that s_i is the new leader (line 20). To maintain replica consistency, the sequencer exploit the **WriteMaj()** and **ReadMaj()** communication primitives that respectively allow (i) to insert an assignment into the state of a majority of sequencer replicas and (ii) to compute the union of such a majority of sequencer states. For further details refer to [2].

```
1  Integer GetSeq(req_id)
2    begin
3      Integer i := 0;
4      loop
5        send ["GetSeq", req_id] to s_i;
6        t.setTimeout := period;
7        wait until ((deliver ["Seq", seq] from s ∈ {s_1 ... s_q}) or (t.expired()))
8        if (not t.expired())
9          then return (seq);
10         else  i := (i + 1) mod q;
11     end loop
12   end
```

(a) The GetSeq() Pseudo-code

```
1  ReqId GetReqID(seq)
2    begin
3      Integer i := 0;
4      loop
5        send ["GetReqID", seq] to s_i;
6        t.setTimeout := period;
7        wait until ((deliver ["Req", req_id] from s ∈ {s_1 ... s_q}) or (t.expired()))
8        if (not t.expired())
9          then return (req_id);
10         else  i := (i + 1) mod q;
11     end loop
12   end
```

(b) The GetReqID() Pseudo-code

Figure 9: Primitives for Invoking the Sequencer

```
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 sender) and primary) do
 7      if (∃ta' ∈ state.TA : ta'.req_id = req_id)
 8        then A-send ["Seq", ta'.#seq, req_id] to sender;
 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 sender;
13                else  primary := ⊥;
14    when ((A-deliver ["GetReqID", seq] from sender) and primary) do
15      if (∃req_id : ta_i = ⟨req_id, seq⟩ ∈ state.TA)
16        then A-send ["Req", req_id] to sender;
17        else  A-send ["Null", null] to sender
18    when (not Leader?()) do
19      primary := ⊥;
20    when ((Leader?()) and (not primary)) do
21      (connected, maj_state) := ReadMaj ();    % computing_sequencer_state %
22      if (connected)
23        then ta := last(maj_state.TA);
24              if (ta ≠ null)
25                then connected := WriteMaj (ta);
26                      if (connected)
27                        then for each ta_j, ta_ℓ ∈ maj_state.TA :
28                                    (ta_j.#seq = ta_ℓ.#seq) and (ta_j.#epoch > ta_ℓ.#epoch)
29                                    do maj_state.TA := maj_state.TA − {ta_ℓ};
30                             state.TA := maj_state.TA; seq := last(state.TA).#seq;
31              if (WriteMaj (maj_state.epoch + 1) and connected)
32                then primary := ⊤;
33  end loop
```

Figure 10: The Sequencer Protocol Pseudo-code Executed by s_i