

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 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 to solve the underlying agreement problem. 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 and clients 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 [23] and passive [7] are well-known approaches used to keep strongly consistent the internal states of a set of server replicas connected by a communication network. Informally, strong 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 let replicas 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., [22]) to be deployed in a *partially synchronous* distributed system, as they rely on distributed agreement protocols¹. A partially

synchronous system is an asynchronous distributed system with some additional timing assumptions on message transfer delays, process speeds etc. [14, 15]. As examples, the timed asynchronous model [17] or the asynchronous system model with unreliable failure detectors [10] are partially synchronous distributed systems models. Partial synchrony can be guaranteed (most of the time) only on small size distributed systems deployed over either a LAN or a CAN (Controlled Area Network). This makes then difficult the deployment of server replicas over asynchronous large scale distributed systems as service availability could be reduced by asynchrony even in the absence of replica failures.

In this paper we first introduce an architectural property, namely *Client/Server-Asynchrony*, that if satisfied by a replication technique allows to deploy clients and server replicas within an asynchronous distributed system. 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). The mid-tier is responsible for maintaining consistency. To achieve this, the mid-tier embeds two basic components, namely the *sequencer* and the *active replication handler*. The first component is 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 this architecture satisfies *Client/Server-Asynchrony*.

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.

in a deterministic way in an asynchronous distributed system due to FLP impossibility result [18].

¹It is well known that these primitives are not implementable

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 basics of software replication and the *Client/Server-Asynchrony* property, illustrates active replication and defines correctness conditions; Section 3 introduces a three-tier architecture that allows to satisfy these architectural properties; Section 4 details a provable correct mid-tier protocol whose correctness, due to lack of space, is proved in [3]. Finally, Section 5 concludes the paper.

2 Software Replication

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 replicas’ state to let clients interact with the stateful replicated server as if it was a singleton, highly available, object.

In this section we first introduce an architectural property 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 interactions.

2.1 A Desirable Property for Software Replication

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

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

This property allows clients and server replicas of a stateful service to be deployed across a system like the Internet.

Replication techniques are usually implemented adopting two-tier architectures. This results in replicas having to run agreement protocols to enforce consistency. As a consequence, two-tier architectures for software replication do not allow to satisfy *Client/Server Asynchrony* (see also Section 2.4). We address such issues in the context of active replication in the following subsections.

2.2 Active Replication

Active replication [21, 23] can be specified taking into account two types of players, namely clients and a set of *deterministic* server replicas. Each client request is received by each available replica that independently executes the request and sends back the reply to the client. We abstract the computation of a request *req* by a generic replica through the *compute(req)* method that returns a result (*res*). Assuming a crash failure model for processes, the client waits only for the first reply and filters out others.

2.3 Correctness of Active Replication

To enforce strong replica consistency, a *correct* implementation of an actively replicated deterministic service should satisfy the following properties [13]:

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).*

These 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 order². 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 each client request for the first time. Finally, the *Response Integrity* property guarantees that the replies returned to clients are generated by the service.

²As 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* [19].

2.4 Implementing Active Replication

To enforce the consistency of a service replicated using active replication, a *total order multicast* primitive 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 [13]. According to this classification, message ordering can be built by:

- 1.senders**(e.g. [1, 12, 11, 21, 16]): in this case senders agree on the order to submit requests to the replicas;
- 2.destinations** (e.g.[5, 10]): in this case replicas agree on the order to process client requests;
- 3.external processes** (e.g. [6, 8, 16, 20]): ordering may involve processes that are neither senders nor destinations.

In algorithms of the first class clients must synchronize among them to get a total order. This violates *Client-Asynchrony*.

Algorithms of the second and third classes can be adapted to satisfy *Client-Asynchrony*. For instance, they can be adapted to the client/server interaction model by equipping clients with a retransmission/redirection mechanism to cope with failures of destinations or external processes. Concerning *Server-Asynchrony*, it is easy to see that it is not satisfied by algorithms falling in the second class. Algorithms of the third class are commonly designed assuming that the “external process” is elected among clients or servers that have to run in a partially synchronous distributed system. As example, in [6] 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 [9].

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 (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 present 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 *eventual reliable asynchronous channels* that are modelled by the following properties:

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

Channel No Duplication. *Messages are delivered to processes at most once.*

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 $\{c_1, \dots, c_l\}$ of client processes (client-tier), a set $\{h_1, \dots, h_n\}$ of active replication handler (ARH) replicas (implementing the mid-tier ARH *component*), a set $\{r_1, \dots, r_m\}$ of deterministic server replicas (end-tier) and a single *correct* process, belonging to the mid-tier, namely the *Seq* component. This component actually abstracts the fault-tolerant implementation of the sequencer service specified in Section 4.3.

We finally assume:

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

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

3.2 The 3T Architecture

Figure 1 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.

- **Retransmission/Redirection message handler (RR).** RR is a message handler embedded by each client to cope with ARH replica

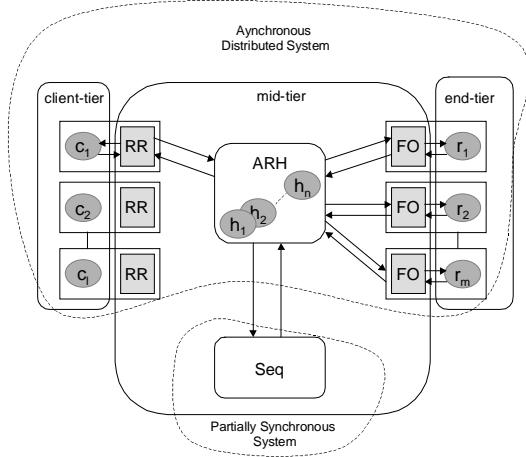


Figure 1. A Mid-tier Architecture for Active Replication

failures and with the asynchrony of communication channels (i.e., to enforce the *Termination* property). RR 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 received within the timeout, and it continues to issue the request towards different ARH replicas until a reply is eventually received.

- **Sequencer (Seq).** The sequencer 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).** ARH 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 is eventually delivered to every non-crashed end-tier replica. Once replicas return results, ARH returns them to clients. To achieve *Termination* despite failures, ARH 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 message handler (FO).** FO is a message handler placed in front of each end-tier replica (i) to ensure ordered execution of client requests according to the number assigned by Seq to each request and (ii) to avoid repeated executions of the same client request (possibly sent multiple times by ARH). Previous features allow to enforce *Uniform Agreed Order* and *Update Integrity* on the end-tier replicas.

RR and FO message handlers are co-located with the client and with the end-tier server replica respectively. Hence we assume that the 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 detail the protocols of the components depicted in Figure 1.

4.1 Introductory Example

Figure 2 shows a failure-free run of the mid-tier protocol. In this scenario, a client c_1 invokes RR method `ISSUEREQ(request1)` 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. 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(request1)` on its replica that produces a result $result_1$. 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 2). 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 Retransmission/Redirection

The RR pseudo-code is shown in Figure 3. RR maintains a state variable $\#cl_seq$ increased for 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). 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 ARH replica is selected (line 15) and the request is sent again towards such a replica (line 10).

4.3 Sequencer

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 a unique request identifier. To cope with assignment request retransmissions, 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:

Assignment Validity. *If $a \in A$ then (i) there exists a process p that issued an assignment request*

GETSEQ(req_id) and (ii) $req_id = a.req_id$.

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

Bijection. $\forall a_i, a_j \in A : a_i.\#seq \neq a_j.\#seq \Leftrightarrow a_i.req_id \neq a_j.req_id$

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

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

A fault-tolerant sequencer implementation that satisfies the previous specification is described in [2, 4].

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

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 `GETREQID(#seq)` returns req_id ; else it returns null.*

Interested readers can find in [3] a modified version of the sequencer implementation proposed in [2, 4] that satisfies the afore sequencer service specification.

Implementing a fault-tolerant replicated sequencer mandates the solution of agreement problems. As a consequence, sequencer replicas have to run within a partially synchronous distributed system, in order to overcome the FLP impossibility result [18] and to enforce the service availability. Let us note that the sequencer service can be implemented by a set of server replicas exploiting *group communication primitives* (e.g., total order multicast) based on distributed agreement protocols.

4.4 Active Replication Handler

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 4 illustrates the pseudo-code of an ARH replica, based on the following 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

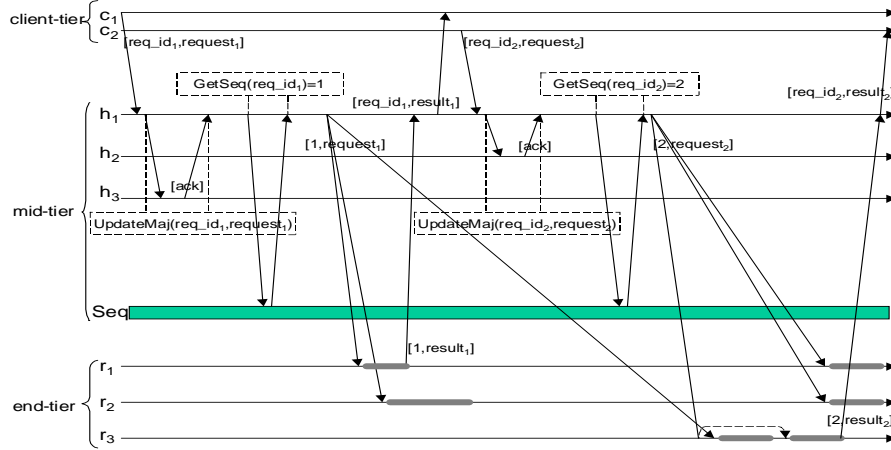


Figure 2. a Failure-free Run

```

RR MESSAGE HANDLER
1  hlist := ⟨h1, ..., hn⟩;
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 3. Pseudo-code of RR Message Handler

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 served;
- 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 $\{h_1, \dots, h_n\}$. 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 set of $\langle req_id, request \rangle$

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 $\{h_1, \dots, h_n\}$. 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;

Due to the lack of space, an implementation of the above primitives is given [3].

ARH Protocol. Each ARH replica runs the protocol of Figure 4. 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() primitive³ (line 5) and (ii)

³In 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.

```

ACTIVE REPLICATION HANDLER
1 SET LocalState :=  $\emptyset$ ;
2 INTEGER LastServedReq := 0;
3 INTEGER #seq;
4 when (deliver ["Request",  $\langle req\_id, request \rangle$ ]) from c do
5   UPDATEMAJ( $\langle req\_id, request \rangle$ );
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_idj := Seq.GETREQID(j);
11      requestj := (request :  $\langle req\_idj, request \rangle \in LocalState$ );
12      for each rl ∈ {r1, ..., rm} do send ["TORequest",  $\langle j, requestj \rangle$ ] to rl;
13 for each rl ∈ {r1, ..., rm} do send ["TORequest",  $\langle \#seq, request \rangle$ ] to rl;
14 LastServedReq := max(LastServedReq, #seq);
15 wait until (deliver ["TOResult",  $\langle \#seq, result \rangle$ ]) from rk ∈ rlist
16 send ["Result",  $\langle req\_id, result \rangle$ ] to c;

```

Figure 4. Pseudo-code of an ARH replica h_i

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 states⁴;
- 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 first result, h_i forwards the result to the client.

4.5 Filtering and Ordering

Figure 5 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 to *ExpectedSeq* (line 4). When such condition holds FO *atomically* executes statements (5–8). In particular, when a request has $seq = ExpectedSeq$ (line 5), FO computes the result, stores the result in *Executed*

⁴As each client request is stored in 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.

and increases the *ExpectedSeq* variable. Finally, FO sends a "TOResult" message, containing the request sequence number and the request's result, to h_i .

```

FO MESSAGE HANDLER
1 ARRAY Executed;
2 INTEGER ExpectedSeq := 1;
3 when (deliver ["TORequest",  $\langle seq, request \rangle$ ]) from  $h_i$  do
4   wait until ( $seq \leq ExpectedSeq$ )
5   if ( $seq = ExpectedSeq$ )
6     then result := compute(request);
7     Executed[seq] := result;
8     ExpectedSeq := ExpectedSeq + 1;
9   send ["TOResult",  $\langle seq, Executed[seq] \rangle$ ] to  $h_i$ ;

```

Figure 5. Pseudo-code of FO message handler

4.6 Discussion

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.

Note that clients do not play any role in enforcing consistency, they are only equipped with a simple retransmission/redirection mechanism. This allows to implement very thin clients that match current technologies. It also widens the spectrum of devices that could host such clients (e.g. mobile phones, PDAs, etc).

The proposed 3T architecture confines as much as possible the need of partial synchrony. As shown, synchrony is actually needed only for implementing the sequencer service (other entities 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 would embrace the whole mid-tier.

Let us note that the three-tier active protocol has been presented omitting several possible optimization, detailed in [3].

5 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 timing assumption on the system connecting replicas in order to maintain consistency. This limits the dissemination of replicated stateful services over the asynchronous, large scale systems as the Internet.

This paper introduces a *Client/Server-Asynchrony* property that captures the possibility to deploy replicas of a service across asynchronous systems. Then we presented a 3T architecture for handling active software replication. 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.

Then we presented an implementation of a provable correct active replication protocol whose correctness, due to lack of space, is shown in [3].

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, Università di Roma “La Sapienza”, <http://www.dis.uniroma1.it/~irl>, november 2001.
- [3] R. Baldoni, C. Marchetti, and S. Tucci-Piergiovanni. Active Replication in Asynchronous Three-Tier Distributed System. Technical Report 05-02, Dipartimento di Informatica e Sistemistica, Università di Roma “La Sapienza”, <http://www.dis.uniroma1.it/~irl>, february 2002.
- [4] R. Baldoni, C. Marchetti, and S. Tucci-Piergiovanni. A fault-tolerant sequencer for timed asynchronous systems. In *Proceedings of the 8th Euro-Par Conference*, pages 578–588, Paderborn, Germany, August 2002.
- [5] K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.
- [6] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.
- [7] N. Budhiraja, F. Schneider, S. Toueg, and K. Marzullo. The Primary-Backup Approach. In S. Mullender, editor, *Distributed Systems*, chapter 8, pages 199–216. Addison Wesley, 1993.
- [8] R. Carr. The Tandem Global Update Protocol. *Tandem Systems Review*, June 1985.
- [9] 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.
- [10] T. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, pages 225–267, Mar. 1996.
- [11] 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.
- [12] 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.
- [13] X. Défago. *Agreement-Related Problems: From Semi Passive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 2000. PhD thesis no. 2229.
- [14] 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.
- [15] C. Dwork and N. L. L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
- [16] P. Ezhilchevan, R. Macedo, and S. 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.
- [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] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.
- [20] F. Kaashoek and A. S. Tanenbaum. Fault tolerance using group communication. *ACM Operating Systems Review*, 25(2):71–74, April 1991.
- [21] L. Lamport. Time, Clocks and the Ordering of Event in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.
- [22] L. Moser, P. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, April 1996.
- [23] F. Schneider. Replication Management using state-machine approach. In S. Mullender, editor, *Distributed Systems*, chapter 7, pages 169–198. Addison Wesley, 1993.