Fully Distributed Three-Tier Active Software Replication

Carlo Marchetti, Roberto Baldoni, Sara Tucci-Piergiovanni, Antonino Virgillito,

Dipartimento di Informatica e Sistemistica

Università degli Studi di Roma “La Sapienza”

Via Salaria 113, 00198 Roma, Italia

Contact author: Roberto Baldoni, email: baldoni@dis.uniroma1.it

Abstract

Keeping strongly consistent the state of the replicas of a software service deployed across a distributed system prone to crashes and with highly unstable message transfer delays (e.g. the Internet), is a real practical challenge. The solution to this problem is subject to the FLP impossibility result, and thus there is a need for "long enough" periods of synchrony with time bounds on process speeds and message transfer delays to ensure deterministic termination of any run of agreement protocols executed by replicas. This behavior can be abstracted by a partially synchronous computational model. In this setting, before reaching a period of synchrony, the underlying network can arbitrarily delay messages and these delays can be perceived as false failures by some timeout-based failure detection mechanism leading to unexpected service unavailability. This paper proposes a fully distributed solution for active software replication based on a three-tier software architecture well-suited to such a difficult setting. The formal correctness of the solution is proved by assuming the middle-tier runs in a partially synchronous distributed system. This architecture separates the ordering of the requests coming from clients, executed by the middle-tier, from their actual execution, done by replicas, i.e., the end-tier. In this way clients can show up in any part of the distributed system and replica placement is simplified, since only the middle-tier has to be deployed on a well-behaving part of the distributed system that frequently respects synchrony bounds. This deployment permits a rapid timeout tuning reducing thus unexpected service unavailability.

Index Terms

Dependable distributed systems, software replication in wide-area networks, replication protocols, architectures for dependable services;

I. Introduction

Replication is a classic technique used to improve the availability of a software service. Architectures for implementing software replication with strong consistency guarantees (e.g., [8], [15], [20], [21], [27]–[29], [31], [35], [36]) typically use a two-tier approach. Clients send their requests to the replica tier that ensures all replicas are in a consistent state before returning a reply to the client. This requires replicas (and sometimes even clients, e.g., [34]) to run complex agreement protocols [12], [23]. From a theoretical viewpoint, a run of these protocols terminates if the underlying distributed system infrastructure ensures a time t after which (unknown) timing bounds on process speeds and message transfer delays will be established, i.e., a partially synchronous computational model [9], [18][1]. Let us remark that in practice partial synchrony only imposes that after t there will be a period of synchrony "long enough" to terminate a run [9].

Before the system reaches a period of synchrony, running distributed agreement protocols among replicas belonging to a two-tier architecture for software replication can be an overkill [1]. Under these conditions, the replicated service can show unavailability periods with respect to clients only due to replication management (even though the service remains correct). Intuitively, this can be explained by noting that replicas use timeouts to detect failures. Hence, if messages can be arbitrarily delayed by the network, then timeouts may expire even if no failure has occurred, causing the protocol to waste time without serving client requests. The use of large timeouts can alleviate this phenomenon at the price of reducing the capability of the system to react upon the occurrence of some real failure. One simple way to mitigate this problem is to observe that in a large and complex distributed system (e.g. the Internet), there can be regions that reach a period of synchrony before others, e.g. a LAN, a CAN etc. Therefore, placing replicas over one of such "early-synchronous" regions can reduce such service unavailability shortening the timeout tuning period. However, in many cases, the deployment of replicas is not in the control of the protocol deployer but it is imposed by organizational constraints of the provider of the service (e.g., a server may not be moved from its physical location).

In this paper we propose the use of a three-tier architecture for software replication to alleviate the unavailability problem that has been previously introduced. This architecture is based on the idea of "*physically interposing*" a middle-tier between clients (client-tier) and replicas (end-tier) and to "*layer*" a sequencer service on top of a total order protocol only within the middle-tier. This approach is motivated by the following main observation: three-tier replication facilitates a sharp separation between the replication logic (i.e., protocols and mechanism necessary for managing software replication) and the business logic embedded in the end-tier. Therefore, the middle-tier could be deployed over a region of a distributed system showing an early-synchronous behavior where timeouts can be quickly tuned, limiting thus service unavailability periods.

[1]This need of synchrony is a consequence of the fact that the problem of "keeping strongly consistent the state of a set of replicas" boils down to the Consensus problem. Therefore, it is subject to the FLP impossibility result [19], stating that it is impossible to design a distributed consensus protocols ensuring both safety and *deterministic* termination over an asynchronous distributed system.

We exploit the three-tier architecture to implement active replication over a set of deterministic replicas[2]. To this aim, the middle-tier is in charge of accepting client requests, evaluating a total order of them and forwarding them to the end-tier formed by deterministic replicas. Replicas process requests according to the total order defined in the middle-tier and return results to the latter. The middle-tier waits for the first reply and forwards it to clients.

We present a fully distributed solution for the middle-tier that does not rely on any centralized service. More specifically, the paper presents in Section II the formal specification of active software replication. Section III details the three-tier system model. Section IV introduces the formal specification of the main component of the middle-tier, namely the sequencer service, which is responsible for associating in a fault-tolerant manner a request of a client to a sequence number. In the same section, a fully distributed implementation of the sequencer service is proposed based on a total order protocol. Section V details the complete three-tier software replication protocol while its correctness proof is given in Section VI. Even though the paper focuses on problem solvability, it also discusses in Section VII both practicality of the assumptions done in the system model and efficiency issues of the proposed protocol. In particular it points out, firstly, how deploying the middle-tier in an early synchronous region can help in reducing the service unavailability problem and, secondly, the relation of partial synchrony with respect to implementations of total order built on top of different software artifacts e.g., unreliable failure detectors [9], group toolkits [8], the Timely Computing Base (TCB) [39].

Let us finally remark that to have a fast client-replicas interaction, the three-tier architecture needs the fast response of just one replica while the two-tier requires a majority of replicas to reply quickly. The price to pay by a three-tier architecture is an additional hop (i.e., a request/reply interaction) for a client-replica interaction. In the rest of the paper, Section VIII describes the related work and Section IX draws some conclusion.

II. A SPECIFICATION OF ACTIVE REPLICATION

Active replication [23], [30], [37] can be specified by taking into account a finite set of clients and a finite set of *deterministic* replicas. Clients invoke *operations* onto a replicated server by issuing *requests*. A request message req is a pair $\langle id, op \rangle$ in which $req.id$ is a unique request

[2]In [2] it has been shown that the three-tier approach to replication can also be used to handle non-deterministic replicas.

identifier (unique for each distinct request issued by every distinct client), and $req.op$ is the actual operation that the service has to execute. A request reaches all the available replicas that *process* the request invoking the *compute(op)* method, which takes an operation as an input parameter and returns a result (res). Replica determinism implies that the result returned by *compute(op)* only depends from the initial state of the replicas and from the sequence of processed requests. Results produced by replicas are delivered to clients by means of *replies*. A reply message rep is a pair $\langle id, res\rangle$ in which $rep.id$ is the unique request identifier of the original client request $req : rep.id = req.id$ and $rep.res$ is the result of the processing of req. Two requests req_1 and req_2 are equal, i.e., $req_1 = req_2$ *iff* $req_1.id = req_2.id$, and $req_1 = req_2 \Rightarrow req_1.op = req_2.op$.

A *correct* implementation of an actively replicated deterministic service satisfies the following properties[3]:

Termination. If a client issues a request $req \equiv \langle id, op\rangle$ then it eventually receives a reply $rep \equiv \langle id, res\rangle$, unless it crashes.

Uniform Agreed Order. If a replica processes a request req, i.e., it executes $compute(req.op)$, as i-th request, then the replicas that process the i-th request must process req as i-th request[4].

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

Response Integrity. If a client issues a request req and delivers a reply rep, then $rep.res$ has been computed by some replica performing $compute(req.op)$.

III. System Model

Processes are classified into three disjoint types: a set $\mathcal{C} = \{c_1, \ldots, c_l\}$ of client processes (client-tier), a set $\mathcal{H} = \{h_1, \ldots, h_n\}$ of active replication handler (ARH) replicas, a set $\mathcal{R} = \{r_1, \ldots, r_m\}$ of deterministic end-tier replicas. A process behaves according to its specification until it possibly crashes. After a crash event a process stops executing any action. A process is *correct* if it never crashes, otherwise it is *faulty*.

Point-to-point communication primitives. Clients, replicas and active replication handlers communicate using *reliable* asynchronous point-to-point channels modelled through the **send**(m, p_j)

[3]These properties are a specialization to the active replication case of the properties proposed in [16]

[4]As replicas are deterministic, if they process *all* requests in the same order before failing, then they will produce the same result for each request. This satisfies *linearizability* [26].

and **deliver**(m, p_j) primitives. The **send** primitive is invoked by a process to send a message m to process p_j. **deliver**(m, p_j) is an upcall executed upon the receipt of a message m sent by process p_j. Channels satisfy the following properties:

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

(C2) Channel Non-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.

Total Order broadcast Communication primitives. ARH replicas communicate *among themselves* using a *uniform total order broadcast* (or *uniform atomic broadcast*) primitive, i.e., ARH replicas have access to two primitives, namely **TOCast(**m**)** and **TODeliver(**m, h_i**)**, used to broadcast a totally ordered message m to processes in $\mathcal{H}$ and to receive a totally ordered message m sent by some process $h_i \in \mathcal{H}$, respectively. The semantics of these primitives are the following [25]:

(TO1) Validity. If a *correct* process h_i invokes **TOCast(**m**)**, then all *correct* processes in $\mathcal{H}$ eventually execute **TODeliver(**m, h_i**)**.

(TO2) Uniform Agreement. If a process in $\mathcal{H}$ executes **TODeliver(**m, h_ℓ**)**, then all correct processes in $\mathcal{H}$ will eventually execute **TODeliver(**m, h_ℓ**)**.

(TO3) Uniform Integrity. For any message m, every process in $\mathcal{H}$ executes **TODeliver(**m, h_ℓ**)** at most once and only if m was previously sent by $h_\ell \in \mathcal{H}$ (invoking **TOCast(**m**)**).

(TO4) Uniform Total Order. If a processes h_i in $\mathcal{H}$ first executes **TODeliver(**m_1, h_k**)** and then **TODeliver(**m_2, h_ℓ**)**, then no process can execute **TODeliver(**m_2, h_ℓ**)** if it has not previously executed **TODeliver(**m_1, h_k**)**.

We assume that any TO invocation terminates. This means that it is necessary to assume that in the distributed system formed by ARHs and their communication channels, there is a time t after which there are bounds on process speeds and message transfer delays, but those bounds are unknown, i.e., *a partial synchrony assumption* [18] [9].

Failure Assumptions. The assumption on the termination of the TO primitives implies that if the specific uniform TO implementation can tolerate up to f failures, then

(A1) ARH Correctness. There are at least $n - f$ *correct* ARH replicas.

Moreover we assume:

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

Practicality of these assumptions will be further discussed in Section VII.

IV. The Sequencer Service

The sequencer service is available to each ARH replica. This service returns a unique and consecutive sequence number for each *distinct* client request and it is the basic building block to satisfy the *Uniform Agreed Order* property of active replication. Furthermore, the service is able to retrieve a request (if any) associated to a given sequence number. This contributes to the enforcement of the *Termination* property despite ARH replica crashes. We first propose a specification and then a fully distributed and fault-tolerant implementation of the sequencer service (DSS).

A. Sequencer Specification

The sequencer service exposes two methods, namely GetSeq() and GetReq(). The first method takes a client request req as input parameter and returns a positive integer sequence number $\#seq$. The second method takes a positive integer $\#seq$ as input parameter and returns a client request req previously assigned to $\#seq$ (if available), or $null$ otherwise. Formally, the sequencer service is specified as follows.

Properties. We denote with $\text{GetSeq}_i() = v$ (resp. $\text{GetReq}_i() = v$) the generic invocation of the GetSeq() (resp. GetReq()) method performed by the generic ARH replica $h_i \in \mathcal{H}$ that terminates with a return value v.

A *correct* implementation of the sequencer service must satisfy properties S1...S6 described below. In particular, to ensure live interactions of correct ARH replicas with the sequencer service, the following liveness property must hold:

(S1) Termination. If h_i is correct, $\text{GetSeq}_i()$ and $\text{GetReq}_i()$ eventually return a value v.

Furthermore, the following safety properties on the $\text{GetSeq}_i()$ invocations must hold:

(S2) Agreement.

$\forall\ (\text{GetSeq}_i(req) = v,\ \text{GetSeq}_j(req') = v'), req = req' \Rightarrow v = v'$

(S3) Uniqueness. $\forall(\text{GetSeq}_i(req) = v, \text{GetSeq}_j(req') = v'), v = v' \Rightarrow req = req'$

(S4) Consecutiveness. $\forall \text{GetSeq}_i(req) = v, (v \geq 1) \wedge (v > 1 \Rightarrow \exists req'$ s.t. $\text{GetSeq}_j(req') = v - 1)$

The *Agreement* property (S2) guarantees that two ARH replicas cannot obtain different sequence numbers for the same client request; the *Uniqueness* property (S3) avoids two ARH replicas to obtain the same sequence number for two distinct client requests; finally, the *Consecutiveness* property (S4) guarantees that ARH replicas invoking GETSEQ() obtain positive integers that are also consecutive, i.e., the sequence of client request ordered according to the sequence numbers obtained by ARH replicas does not present "holes".

Finally, upon invoking GETREQ(), ARH replicas must be guaranteed of the following safety properties.

(S5) Reading Integrity. $\forall\ \text{GETREQ}_i(\#seq) = v\ \Rightarrow ((v = null) \vee (v = req\ s.t.\ \text{GETSEQ}_j(v) = \#seq))$

(S6) Reading Validity. $\forall\ \text{GETSEQ}_i(req) = v \Rightarrow\ \text{GETREQ}_i(v-k) = v'$, $0 \leq k < v$, $v' \neq\ null$

The *Reading Integrity* property (S5) defines the possible return values of the GETREQ() method that returns either $null$ or a client request assigned to the sequence number passed as input parameter. Note that a GETREQ() method implementation that always returning $null$ satisfies this property. To avoid such an undesirable behavior, the *Reading Validity* property (S6) states that if an ARH replica h_i invokes $\text{GETSEQ}_i(req)$ that returns a value $v = \#seq$, it will be then able to retrieve all the client requests $req_1, \ldots, req_{\#seq}$ assigned to a sequence number $\#seq'$ such that $1 \leq \#seq' \leq \#seq$.

B. A Fully Distributed Sequencer Implementation

The implementation is based on a uniform total order broadcast primitive exploitable by ARH replicas (see Section III) used to let the ARHs agree on a sequence of requests. In particular, each DSS class locally builds a sequence of requests which is updated upon receiving each request *for the first time*. Following receipts of requests already inserted in the sequence are simply filtered out. As requests are received in a total order, the local sequence of each DSS class evolves consistently with others.

The DSS class pseudo-code run by each ARH replica h_i is presented in Figure 1. It maintains an internal state composed by the $Sequenced$ array (line 1) that stores in the i-th location the client request assigned to sequence number i, and by a $\#LocalSeq$ counter (line 2) pointing to the first free array location (initialized to 1).

```
CLASS DSS
1  ARRAY $Sequenced := [null, null \ldots]$;
2  INTEGER $\#LocalSeq := 1$;
3  REQUEST GETSEQ($req$)
4    **begin**
5      **if** $(\nexists \#seq : Sequenced[\#seq].id = req.id)$
6        **then** **TOCast**$(req)$;
7      **wait until** $(\exists \#seq : Sequenced[\#seq].req\_id = req\_id)$;
8      **return** $(\#seq)$;
9    **end**
10 REQUEST GETREQ($j$)
11   **begin**
12     **return** $(Sequenced[j])$;
13   **end**
14 **when** $($**TODeliver**$(req, h_\ell))$ **do**
15   **if** $(\nexists \#seq : Sequenced[\#seq] = req)$
16     **then** $Sequenced[\#LocalSeq] := req$;
17          $\#LocalSeq := \#LocalSeq + 1$;
```

Fig. 1. Pseudo-code of the Sequencer class run by ARH replica h_i

The class handles three events, i.e., (i) the invocation of the GETSEQ() method (line 3), (ii) the invocation of the GETREQ() method (line 10), and (iii) the arrival of a totally ordered message (line 14).

In particular, upon the invocation of the GETSEQ() method, it is firstly checked whether the client request (passed as input argument by the invoker) has been already inserted into a $Sequenced$ array location or not (line 5). If it is not the case, the client request is multicast to all other sequencers (line 6). When the request has been sequenced, i.e., it appears in a location of the $Sequenced$ array (line 7), its position in the array is returned to the invoker as the request sequence number (line 8).

Upon the invocation of the GETREQ() method (line 10), the class simply returns the value contained in the array location indexed by the integer passed as input parameter (line 12). Therefore, if the array location contains a client request, the latter is returned to the invoker,

$null$ is returned otherwise.

Finally, when a totally ordered message is delivered to the DSS class by the total order multicast primitive (line 14), it is firstly checked if the client request contained in the message already appears in a location of the $Sequenced$ array (line 15). If it is not the case, the client request is inserted into the $Sequenced$ array in a position indexed by the $\#LocalSeq$ that is then incremented (lines 16–17).

V. A FULLY DISTRIBUTED MIDDLE-TIER PROTOCOL

The proposed protocol strives to maximize service availability by allowing *every* non-crashed ARH replica to *concurrently* (i) accept client requests, (ii) order these requests, (iii) forward ordered requests to the end-tier, (iv) receive results, and (v) return results to clients. As a consequence, the replication scheme can shift from a passive one (if the clients send their requests to a single ARH replica) to a form of active replication (if clients send their request to all ARH replicas)[5].

In order to enforce the active replication specification in the presence of ARH replica failures and asynchrony of communication channels, we embed within client and end-tier replica processes two message handlers, i.e., RR (*retransmission and redirection* handler) within clients, and FO (*filtering and ordering* handler) within end-tier replicas. These handlers intercept and handle messages sent by and received from the process they are co-located with.

In particular, RR intercepts all the operation invocations of the client and generates request messages that are (i) uniquely identified and (ii) periodically sent to all ARH replicas according to some retransmission policy, until a corresponding reply message is received from some ARH replica. Examples of distinct implementations of the retransmission policy could be: (i) sending the client request to all ARH replicas each time a timeout expires or (ii) sending the request to a different ARH replica each time the timeout expires.

FO intercepts all incoming/outgoing messages from/to ARHs in order to ensure *ordered request execution* (operations are computed by replicas according to the request sequence number piggybacked by ARHs) and *duplicate filtering* (the same operation contained in repeated requests is computed only once). Request messages arriving out of order at FO are enqueued until they

[5]This replication scheme has been named *asynchronous replication* in [22] (see Section VIII)

can be executed. FO also stores the result computed for each operation by its replica, along with the sequence number of the corresponding request message. This allows FO to generate a reply message upon receiving a retransmitted request, thus avoiding duplicate computations and at the same time contributing to the implementation of *Termination*.

An implementation of FO and RR is presented in [33].

A. Introductory examples

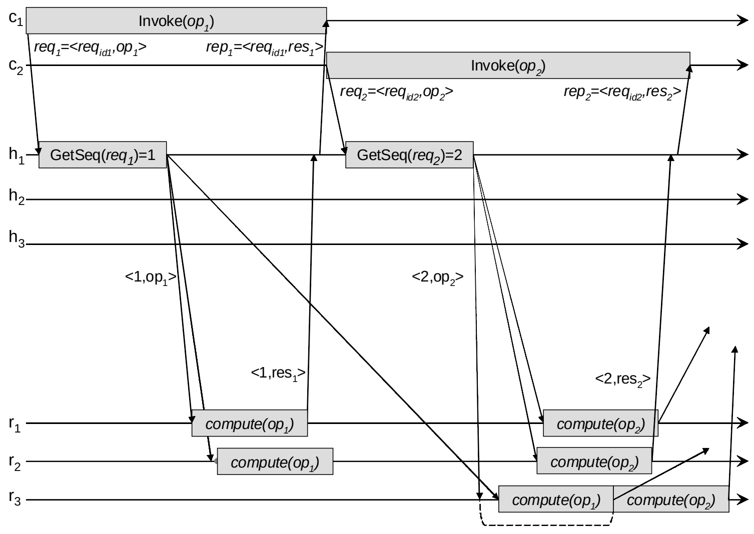

Fig. 2. A failure-free run of the fully distributed three-tier active replication protocol

Let us introduce the middle-tier protocol using two simple introductory examples.

Failure-free run (Figure 2). In this scenario, client c_1 invokes the *Retransmission/Redirection* (RR) $\textsc{Invoke}(op_1)$ method to perform operation op_1. This method creates a uniquely identified request message $req_1 = \langle req_{id1}, op_1 \rangle$ and then it sends req_1 to an ARH replica (e.g., h_1). Upon receiving req_1, h_1 invokes $\textsc{GetSeq}(req_1)$ on the DSS class to assign a unique sequence number (1 in the example) to req_1. Then h_1 sends a message containing the pair $\langle 1, op_1 \rangle$ to all end-tier replicas and starts waiting for the first result. The *Filtering and Ordering* (FO) message handler of each end-tier replica checks if the sequence number of the request received is the expected one with respect to the computation of the replica it wraps, i.e., if the request sequence number is 1 in this scenario. In the example, the FO handlers of r_1 and r_2 immediately verify this condition, and thus invoke $compute(op_1)$ on their replicas that produce the result res_1. Then FO sends a message to h_1 containing the pair $\langle 1, res_1\rangle$. Upon delivering *the first* among these messages, h_1 sends a reply message $\langle req_{id1}, res_1\rangle$ back to c_1. h_1 discards following results produced for operation op_1 by end-tier replicas (corresponding messages are not shown in Figure 2 for simplicity). Then h_1 serves req_2 sent by c_2. To do so, h_1 gets the req_2's sequence number (2) from the DSS class, sends a message containing a pair $\langle 2, op_2\rangle$ to all end-tier replicas and waits for the first reply from the end-tier. Note that in this scenario r_3 receives $\langle 2, op_2\rangle$ *before* receiving $\langle 1, op_1\rangle$. However FO executes operations in the order imposed by sequence numbers. Therefore, upon receiving $\langle 1, req_1\rangle$, the FO handler of r_3 executes both the operations in the correct order and returns 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 and contributes to enforcement of the *Uniform Agreed Order* property (see page 5). As soon as h_1 receives the first $\langle 2, res_2\rangle$ pair, it sends the result back to the client.

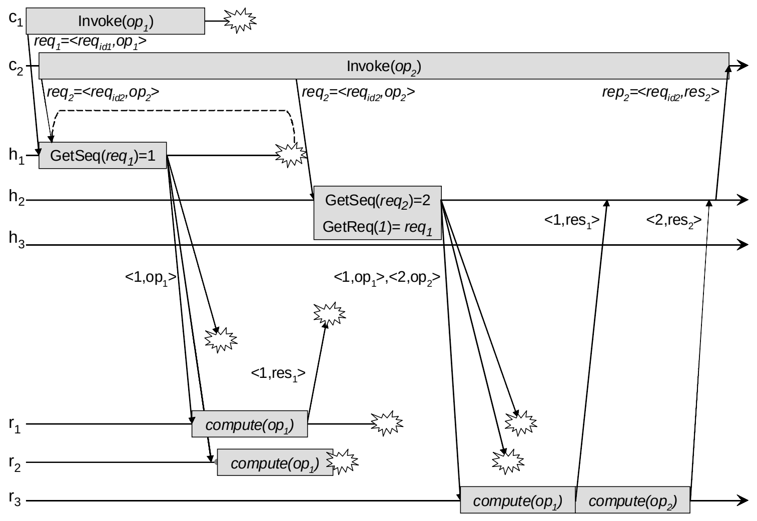

Fig. 3. A run of the fully distributed three-tier active replication protocol in presence of failures

Run in the presence of failures (Figure 3). As in the previous example, c_1 invokes op_1 that through the RR component reaches h_1 in a message containing the $\langle req_{id1}, op_1\rangle$ pair. Then h_1 gets a unique sequence number (1) for the request by invoking the sequencer. However in this scenario h_1 crashes after having multicast the request to the end-tier. As channels are assumed reliable only among *correct* processes, the request may not be received by some end-tier replicas.

In particular, in Figure 3 the request is received only by r_1 and r_2. Furthermore, c_1 crashes. This implies that req_1 will no longer be retransmitted. Then h_2 serves request req_2 sent by client c_2. Therefore it gets a sequence number (2) invoking GETSEQ(req_2). By checking this sequence number against a local variable storing the maximum sequence number assigned by h_2 to the requests forwarded to the end-tier, h_2 determines that it has not previously sent to end-tier replicas the request assigned to sequence number 1, i.e. req_1. As this request could have been sent by a *faulty* ARH replica, in order to enforce the liveness of the end-tier replicas, h_2 sends to the end-tier a message containing the $\langle 1, req_1 \rangle$ pair, in addition to sending the message containing the $\langle 2, req_2 \rangle$ pair necessary to obtain a response to the pending client request req_2. Therefore, h_2 first invokes the sequencer GETREQ(1) method to obtain req_1 and then sends to end-tier replicas both the $\langle 1, req_1 \rangle$ and $\langle 2, req_2 \rangle$ pairs. In this way the unique correct replica of this scenario, i.e., r_3, is maintained live and consistent by h_2. As usual, h_2 returns to c_2 the result of op_2 as soon as it receives res_2 from r_3.

The following section details the protocols run by ARHs.

B. ARH Protocol

We distinguish the following message types:

Messages exchanged between the client-tier and the middle-tier. We denote with "$Request$" the messages sent by clients to ARH replicas, and with "$Reply$" the messages following the inverse path;

Messages exchanged between the middle-tier and the end-tier. These messages contain sequence numbers produced by the sequencer and used by replicas to execute requests in a unique total order. Therefore we denote with "$TORequest$" (*totally ordered request*) the messages sent by ARH replicas to end-tier replicas, and with "$TOReply$" (*totally ordered reply*) the messages following the inverse path.

As depicted in Figure 4, each ARH replica embeds a local DSS class ($Sequencer$) that implements the sequencer service as described in Section IV, which is initialized at line 3. The internal state of each ARH replica is represented by the $LastServedReq$ integer (line 1), which dynamically stores the maximum sequence number among the numbers assigned by h_i to the requests forwarded to the end-tier. $\#seq$ (line 2) is a variable used to store the sequence number assigned by $Sequencer$ to the client request currently being served by h_i. ARH replicas handle

```
ARH
 1  INTEGER $LastServedReq := 0$;
 2  INTEGER $\#seq$;
 3  $Sequencer :=$ new DSS();
 4  when ($deliver$ ["Request", $req$] from $c$) do
 5    $\#seq := Sequencer.\textsc{GetSeq}(req)$;
 6    if $(\#seq > LastServedReq + 1)$
 7      then for each $j : LastServedReq < j < \#seq$
 8            do $req_j := Sequencer.\textsc{GetReq}(j)$;
 9            for each $r_\ell \in \{r_1, ..., r_m\}$ do send ["TORequest", $\langle j, req_j.op\rangle$] to $r_\ell$;
10    for each $r_\ell \in \{r_1, ..., r_m\}$ do send ["TORequest", $\langle \#seq, req.op\rangle$] to $r_\ell$;
11    $LastServedReq := max(LastServedReq, \#seq)$;
12    wait until (deliver ["TOReply", $\langle \#seq, res\rangle$] from $r_k \in \{r_1, ..., r_m\}$)
13    send ["Reply", $\langle req.id, res\rangle$] to $c$
```

Fig. 4. Pseudo-code of an ARH replica h_i

only one event, i.e., the arrival of a client request in a "$Request$" type message (line 4). In particular, upon the receipt of a client request, h_i first invokes $Sequencer$ to assign a sequence number to the request (stored in the $\#seq$ variable, line 5). Then ARH controls whether $\#seq$ is greater than $LastServedReq + 1$. Note that if $\#seq > LastServedReq + 1$, then some other ARH replica served some other client requests with sequence number comprised in the interval $[LastServedReq + 1, \#seq - 1]$. In this case, as shown in the second example of the previous section, h_i sends these requests again to the end-tier (lines 7–9) in order to preserve the protocol *Termination* property (S1) despite possible ARH failures. Requests are retrieved from *Sequencer* (line 8) thanks to the *Reading Validity* property (S6). Then h_i sends to server replicas the "$TORequest$" message containing (i) the operation contained in the client request currently being served and (ii) the sequence number $\#seq$ assigned to the request (line 10). Finally h_i updates the $LastServedReq$ variable (line 11) and waits for the first "$TOReply$" message containing $\#seq$ as sequence number of the result (line 12). Upon the receipt of the result, h_i forwards the result to the client through a "$Reply$" message (line 13).

VI. Correctness Proofs

In this section we first show the correctness proof of the sequencer described in Section IV-A and then the complete middle-tier protocol described in Section V.

A. The sequencer

Theorem 1 ((S1) Termination): If h_i is correct, GETSEQ$_i$() and GETREQ$_i$() eventually return a value v.

Proof: By contradiction. Suppose h_i is correct and invokes a sequencer class method that never returns. We distinguish two cases, i.e., either the method is GETREQ() (lines 10–13) or it is GETSEQ() (lines 3–9):

GETREQ() invocation. In this case the invocation can never block: as soon as the content of the j-th position of the array is read, the method returns. Contradiction.

GETSEQ() invocation. We further distinguish two cases: either $\exists \#seq : Sequenced[\#seq].id = req.id$ or $\nexists \#seq : Sequenced[\#seq].id = req.id$ when the **if** statement at line 5 is evaluated.

- In the first case, line 7 is executed immediately after line 5 and the clause of the **wait** statement is satisfied. As a consequence, $\#seq$ is returned to the invoker at line 7. Contradiction.
- In the second case, statement 6 is executed, i.e., the client request is multicast to other ARH replicas. As $\nexists \#seq : Sequenced[\#seq].id = req.id$, the execution blocks at statement 7. As the multicast is executed by a correct replica (by hypothesis), from the *Validity* property (TO1) of the total order primitive, it follows that statement 14 will eventually be executed. Therefore at the end of statement 17 there holds $\exists \#seq : Sequenced[\#seq].id = req.id$ and this in turns provokes the execution to satisfy the clause of the **wait** statement at line 7. As a consequence, $\#seq$ is returned to the invoker. Contradiction.

■

Theorem 2 ((S2) Agreement): $\forall\ (\text{GETSEQ}_i(req) = v,\ \text{GETSEQ}_j(req') = v'), req = req' \Rightarrow v = v'$.

Proof: By contradiction. Suppose h_i invokes GETSEQ$_i(req)$ that returns $\#seq_i$, h_j invokes GETSEQ$_j(req)$ that returns $\#seq_j$ and $\#seq_i \neq \#seq_j$.

From the pseudo-code of Figure 1 (lines 7–8), it follows that in h_i, $Sequenced[\#seq_i] = req$ and in h_j, $Sequenced[\#seq_j] = req$. To insert a request into the $Sequenced$ array, a generic

ARH replica must execute statement 16 that is executed iff the two conditions at lines 14–15 hold. These conditions imply that each ARH replica inserts a client request into $Sequenced$ at most once. Without loss of generality, we suppose that every message delivered to each ARH replica contains a distinct request. As a consequence, statement 16 is executed by both h_i and h_j each time statement 14 is executed, the $Sequenced$ array in each ARH replica reflects the order of its message deliveries, and $Sequenced[k]$ contains the k-th message delivered at statement 14. Then h_i has delivered $m = req$ as $\#seq_i$-th message, while h_j has delivered $m = req$ as $\#seq_j$-th message. Without loss of generality, suppose that $\#seq_i = \#seq_j - 1$. This implies that h_j delivered at least one message $m' \neq m$ *before* m. This violates property TO4 of the total order primitive. Contradiction. ∎

Theorem 3 ((S3) Uniqueness): $\forall(\textsc{GetSeq}_i(req) = v, \textsc{GetSeq}_j(req') = v'), v = v' \Rightarrow req = req'$.

Proof: By contradiction. Suppose that h_i invokes $\textsc{GetSeq}_i(req)$ that returns $\#seq$ and h_j invokes $\textsc{GetSeq}_j(req')$ that returns $\#seq$, and $req \neq req'$. From the sequencer *Agreement* property (S2), let us suppose $i = j$ without loss of generality. However, if h_i invokes $\textsc{GetSeq}(req)$ and $\textsc{GetSeq}(req')$ both returning $\#seq$, from statements 7–8 (Fig 1) it follows $Sequenced[\#seq] = req$ (when $\textsc{GetSeq}(req)$ is invoked) and $Sequenced[\#seq] = req'$ (when $\textsc{GetSeq}(req')$ is invoked), i.e., the value of the $Sequenced[\#seq]$ location has been modified between two method invocations. By noting that the value of the generic $Sequenced$ array location is written at most once (statements 16-17), i.e., once the location indexed by $LocalSeq$ has been written there's no way to write it again, from $req \neq req'$ it follows $Sequenced[\#seq] \neq Sequenced[\#seq]$. Contradiction. ∎

Theorem 4 ((S4) Consecutiveness): $\forall \textsc{GetSeq}_i(req) = v, (v \geq 1) \wedge (v > 1 \Rightarrow \exists req'$ s.t. $\textsc{GetSeq}_j(req') = v - 1)$

Proof: By contradiction. First suppose that h_i invokes $\textsc{GetSeq}(req)$ that returns a sequence number $\#seq < 1$. From pseudo-code of Figure 1 statements 7–8, it follows that $\exists \#seq : Sequenced[\#seq].id = req.id \wedge \#seq < 1$. Therefore, h_i previously executed statement 16 having $LocalSeq = \#seq < 1$. However, $LocalSeq$ is initialized to 1 and is never decremented. Contradiction. Therefore $\#seq \geq 1$.

Suppose that h_i invokes $\textsc{GetSeq}_i(req)$ that returns $\#seq > 1$ and that there do not exist a client request req' and an ARH replica h_j such that if h_j invokes $\textsc{GetSeq}_j(req')$, it obtains

$\#seq - 1$ as result. As h_i obtained $\#seq$ as the result of the $\textsc{GetSeq}_i(req)$, the sequencer *Agreement* property (S2) ensures each ARH replica h_j that successfully invokes $\textsc{GetSeq}_j(req)$, returns $\#seq$. The sequencer *Termination* property (S1) ensures that the method eventually returns in correct replicas. Then, let h_j be a correct replica that invokes $\textsc{GetSeq}_j(req)$; from $\#seq > 1$, it follows that when h_j executes statement 16, $LocalSeq > 1$. Therefore $LocalSeq$ has been previously incremented at statement 17, i.e., h_j previously inserted in $Sequenced[\#seq - 1]$ the content of a message $m = req'$ and this implies $Sequenced[\#seq - 1] \neq null$. This implies that eventually, if h_j invokes $\textsc{GetSeq}_j(req')$, it will obtain $\#seq - 1$ as invocation result. Contradiction. ■

Theorem 5 ((S5) Reading Integrity): $\forall\ \textsc{GetReq}_i(\#seq) = v \ \Rightarrow ((v = null) \vee (v = req$ s.t. $\textsc{GetSeq}_j(v) = \#seq))$.

Proof: By contradiction. Suppose h_i invokes $\textsc{GetReq}_i(\#seq)$ that returns a value v, $v \neq null$ and $\forall \textsc{GetSeq}_j(v) \neq \#seq$. Without loss of generality suppose $i = j$ and that h_i first invokes $\textsc{GetReq}_i(\#seq) = v$ and then $\textsc{GetSeq}_i(v)$. From pseudo-code in Fig. 1 it follows $v = Sequenced[\#seq]$ (statement 12) and that $Sequenced[\#seq] \neq null$ (by hypothesis). From statement 16, $Sequenced[\#seq] \neq null$ implies $Sequenced[\#seq] = v = req$. From statement 15, it follows that req is inserted *only* in $Sequenced[\#seq]$. Therefore, from statements 5-8, $\textsc{GetSeq}_i(req) = \#seq$. Contradiction. ■

Theorem 6 ((S6) Reading Validity): $\forall\ \textsc{GetSeq}_i(req) = v \Rightarrow\ \textsc{GetReq}_i(v - k) = v'$, $0 \leq k < v$, $v' \neq\ null$

Proof: By contradiction. Suppose $\textsc{GetSeq}_i(req) = v$ and $\textsc{GetReq}_i(\#seq - k)$, $0 \leq k < \#seq$, returns $v = null$. Without loss of generality, suppose $v = \#seq = 2$ and $k = 1$. By hypothesis, $Sequenced[2] \neq null$ and this implies that $LocalSeq$ has been previously incremented (passing from 1 to 2) at statement 17. This in turn implies that $Sequenced[1]$ has been previously written upon the delivery (at statement 14) of a client request req, i.e., $Sequenced[1] = req$. As writings in the locations of the $Sequenced$ array are performed at most once (from statements 16-17 and by noting that $LocalSeq$ is never decremented), when h_i invokes $\textsc{GetReq}_i(1)$ that returns $v = null$, it follows (statement 12) that $req = null$. Contradiction. ■

B. The three-tier protocol

The following assumption let handling process crashes in a uniform way, i.e., without considering partial or independent failures of co-located components.

No independent failures of co-located components. RR, DSS and FO are co-located with the client, with the ARH replica and with the end-tier replica processes, respectively. We assume that co-located components do not fail independently. This implies that a crash of a client, ARH replica, end-tier replica process implies the crash of its RR, DSS, FO, respectively, and vice-versa.

1) Preliminary Lemmas:

Lemma 1: Let req_1 and req_2 be two requests sent to the end-tier by some ARH replica at statement 9 or at statement 10 into two "TORequest" messages [$``TORequest'', \langle \#seq_1, req_1.op \rangle$] and [$``TORequest'', \langle \#seq_2, req_2.op \rangle$], then $\#seq_1 = \#seq_2 \Leftrightarrow req_1 = req_2$.

Proof:

By contradiction. We distinguish the following three cases.

- *Both requests are sent by some ARH replica at statement 10.* Noting that $\#seq_1$ and $\#seq_2$ are the return values of the GETSEQ() method invocation performed at statement 5, and suppose by contradiction $\#seq_1 = \#seq_2$ and $req_1 \neq req_2$. From the sequencer *Uniqueness* property (S3), it follows that $\#seq_1 = \#seq_2$ implies $req_1 = req_2$. Contradiction. On the other hand, suppose by contradiction $req_1 = req_2$ and $\#seq_1 \neq \#seq_2$. From the sequencer *Agreement* property (S2), it follows that $req_1 = req_2$ implies $\#seq_1 = \#seq_2$. Contradiction.
- *A request (say, req_1) is sent by some ARH replica at statement 9 and the other (req_2) is sent at statement 10.* Note that req_1 is returned at statement 8 from a GETREQ() invocation with input argument $\#seq_1$. As at statement 5 $\#seq > \#seq_1$, from the sequencer *Reading Validity* property (S6) it follows $req_1 \neq null$. Therefore, from the sequencer *Reading Integrity* property (S5), it follows that $\text{GETSEQ}_j(req_1) = \#seq_1$. Furthermore, as in the previous case, $\#seq_2$ is the return value of a GETSEQ() method invocation performed at statement 5, i.e., $\text{GETSEQ}_i(req_2) = \#seq_2$. Suppose by contradiction $\#seq_1 = \#seq_2$ and $req_1 \neq req_2$. Again, from the sequencer *Uniqueness* property (S3) it follows that $\#seq_1 = \#seq_2$ implies $req_1 = req_2$. Contradiction. On the other hand, suppose by contradiction $req_1 = req_2$, and $\#seq_1 \neq \#seq_2$. From the sequencer *Agreement* property

(S2) it follows that $req_1 = req_2$ implies $\#seq_1 = \#seq_2$. Contradiction.

- *Both requests are sent by some ARH replica at statement 9.* In this case both req_1 and req_2 are returned at statement 8 from a GETREQ() invocation with input argument $\#seq_1$ and $\#seq_2$, respectively. In both cases, at statement 5, $\#seq > \#seq_1$ and $\#seq > \#seq_2$. From the sequencer *Reading Validity* property (S6), there follow $req_1 \neq null$, and $req_2 \neq null$. Therefore, from the sequencer *Reading Integrity* property (S5), it follows that $\text{GETSEQ}_j(req_1) = \#seq_1$, and that $\text{GETSEQ}_i(req_2) = \#seq_2$. Suppose by contradiction $\#seq_1 = \#seq_2$ and $req_1 \neq req_2$. Also in this case, from the sequencer *Uniqueness* property (S3) it follows that $\#seq_1 = \#seq_2$ implies $req_1 = req_2$. Contradiction. On the other hand, suppose by contradiction $req_1 = req_2$ and $\#seq_1 \neq \#seq_2$. From the sequencer *Agreement* property (S2) it follows that $req_1 = req_2$ implies $\#seq_1 = \#seq_2$. Contradiction.

■

Lemma 2: If an ARH replica h_i has $LastServedReq = k$, then it has already sent to end-tier replicas k $``TORequest''$ messages, i.e., $[``TORequest'', \langle \#seq_n, req_n.op \rangle]$ for each $n : 1 \leq n \leq k$.

Proof: By contradiction. Assume that h_i has $LastServedReq = k > 0$ ($k = 0$ is a trivial case) and it has not sent to end-tier a “TORequest” message $[``TORequest'', \langle \#seq_j, req_j.op \rangle]$ such that $j : 1 \leq j \leq k$.

Without loss of generality, consider the first time that h_i sets $LastServedReq$ to k at line 11. As $LastServedReq$ is initialized to 0 at line 2, and for each $\#seq$ returned by GETSEQ() at line 5 there holds $\#seq > 0$ (from the sequencer *Consecutiveness* property (S4)), when $LastServedReq$ is set to $\#seq = k$ at line 11, this implies $\#seq = k$ at line 5. We distinguish two cases:

- $k = \#seq = 1$. This is a trivial case: the condition at line 6 does not hold, then h_i has sent a $[``TORequest'', \langle 1, req_1.op \rangle]$ to all end-tier replicas (line 10). Contradiction.
- $k = \#seq > 1$. In this case the condition at line 6 holds, then h_i executed lines 7-9 before updating $LastServedReq$ to k at line 11. This implies that h_i has sent to all end-tier replicas a “TORequest” message $[``TORequest'', \langle \#seq_n, req_n.op \rangle]$ for each $n : 1 \leq n \leq k$. Contradiction.

■

2) Theorems: For the sake of brevity, we will refer to the properties introduced so far using their identifiers. As an example, we will refer to *Channel Validity, No Duplication, and Termination* as C1, C2, and C3.

Theorem 7 (Termination): If a client issues a request $req \equiv \langle id, op \rangle$ unless it crashes, it eventually receives a reply $rep \equiv \langle id, res \rangle$.

Proof: By contradiction. Assume that a client issues a request $req \equiv \langle id, op \rangle$, it does not crash and it does not deliver a result. The correctness of the client, along with the retransmission mechanism implemented by the RR handler, guarantee that req is eventually sent to all ARH replicas. Therefore, from A1 and C3, it follows that a *correct* ARH replica h_c eventually delivers the client request message.

From the algorithm of Figure 4, upon receiving the req, h_c invokes GETSEQ(REQ) (line 5) that terminates due to S1. This method returns the sequence number $\#seq$ associated with the current request.

Lemma 2 ensures that at line 11 all requests such that their sequence number is lower than or equal to $LastServedReq$ (including the current request) have been sent to the end-tier replicas by h_c. A2 and C3 guarantee that at least a correct end-tier replica r_c receives all the requests. This ensures that the FO handler, which executes requests according to their sequence numbers, eventually invokes $compute(req.op)$ within r_c, and then it sends back the result in a TOreply message.

From the correctness of h_c and r_c, and from C3, it follows that the result is delivered to h_c (Figure 4 line 12) that thus sends the reply $rep \equiv \langle req.id, res \rangle$ to the client (line 13). For similar reasons, the RR handler eventually delivers the result to the client that thus receives the result. Contradiction. ■

Theorem 8 (Uniform Agreed Order): If an end-tier replica processes a request *req*, i.e., executes $compute(req.op)$), as i^{th} request, then every other end-tier replica that processes the i^{th} request will execute *req* as i^{th} request.

Proof:

By contradiction. Assume that an end-tier replica r_k executes req as i^{th} request and another end-tier replica r_h executes as i^{th} a request req' and $req \neq req'$.

The FO handlers of r_h and r_k ensure that requests are executed at most once and according to the sequence numbers attached to them by ARH replicas at line 10 of the pseudo-code depicted in Fig. 4.

Therefore the i^{th} request processed by r_k, i.e., req, is associated with a sequence number $\#seq = i$, i.e., r_k delivered a "*TORequest*" message containing $\langle i, req.op \rangle$. For the same reasons, r_h delivered a "*TORequest*" message containing $\langle i, req'.op \rangle$. From Lemma 1, it follows $req = req'$. Contradiction. ■

Theorem 9 (Update Integrity): For any request *req*, every end-tier replica executes *compute(req.op)* at most once, and only if a client has issued *req*.

Proof: The case in which the same request is executed twice is trivially addressed by noting that FO handlers filter out duplicates of TOrequest messages.

Assume thus by contradiction that an operation op executed by a replica, has not been invoked by a client.

The FO handler executes only operations contained in TOrequest messages delivered to r_k. From C1, it follows that if r_k delivers a "*TORequest*" message containing $\langle seq, req.op \rangle$, then the TOrequest message has been sent by an ARH replica h_i either at line 9 or at line 10 (see Figure 4).

If the message has been sent at line 10, h_i has received req at line 4. This request has been then sent from a client in a request message (from C1). Contradiction.

Otherwise, if the TOrequest has been sent at line 9, from S5 and S6 there exists an ARH replica h_j that has previously executed GETSEQ$_j$(*req*). As GETSEQ$_j$(req) (line 5) is always executed *after* the delivery of a client request message (line 4) it follows that req has been sent by a client in a request message (from C1). Contradiction. ■

Theorem 10 (Response Integrity): If a client issues a request *req* and delivers a reply *rep*, then *rep.res* has been computed by some end-tier replica, which executed $compute($**req.op**$)$.

Proof: By contradiction. Assume that a client issues a request req and delivers a reply rep and $rep.res$ has not been computed by an end-tier replica.

From C1, if a client delivers rep then an ARH replica h_i has previously sent a reply message containing rep to the client. From the algorithm of Figure 4, if h_i sends a reply message containing $rep \equiv \langle req.id, res \rangle$ (line 13) to the client then (i) h_i received a client request message req from the client (line 4), (ii) h_i invoked GETSEQ$_i$(req) returning $\#seq$ (line 5) and (iii) it has successively delivered $\langle \#seq, res \rangle$ from a replica (line 12). From C1, $\langle \#seq, res \rangle$ has been sent by the FO handler of an end-tier replica r_k. This implies that the request has been previously executed invoking $compute(req.op)$. Contradiction.

■

VII. Practical Issues

Practicality of the assumptions. Most of the assumptions introduced in Section III are necessary to ensure the termination property of our protocol. These include the assumption on one replica being always up, the assumption on reliable point-to-point communication channels and the assumptions on partial synchrony of the region of the distributed system where ARHs are deployed, which is at the base of the termination of the TO broadcast primitive. Let us note that if one of these assumptions is violated, only liveness is affected (i.e., the three-tier protocol blocks), while safety is always preserved. The assumption on the number of correct replicas is the weakest one under which the system is still able to provide the service. The assumption on point-to-point communication channels allows link failures, as long as they are repaired in a finite time. In practice it is implemented by message retransmission and duplicate suppression.

The assumption on partial synchrony does not mean that the timing bounds have to hold *always*. Practically, these bounds have to hold only for a period of time which is *long enough* to let complete any run triggered by an invocation of the TO broadcast primitive[6].

Efficiency of the three-tier architecture. Up to this section we have focused the attention on problem solvability of active software replication using a three-tier protocol. In the following we discuss under which setting the proposed architecture reduces the problem of unexpected service unavailability pointed out in the introduction. Firstly, three-tier architecture assumes that a *service deployer* may place end-tier replicas according to the strategy of the organization that wants to provide the service. Under this given condition a *protocol deployer* has to select a region of a distributed system in which to deploy the middle-tier, i.e., the ARHs. For the three-tier replication protocol to be efficient, the protocol deployer selects a region that better than others enjoys the following two properties:

- the region shows an "early-synchronous" behavior. Early synchronous means that the distributed system will reach synchrony bounds of a partially-synchronous system very early

[6]This follows from the absence of an explicit notion of time in asynchronous system models, in which the term "long enough" cannot be further characterized and it is commonly replaced by "always".

in any run triggered by an invocation of a TO broadcast primitive. Synchronous distributed systems or systems that exhibit a synchronous behavior ”most-of-the-time” are specific instantiations of an “early-synchronous” distributed systems.

- as many as possible of the point-to-point reliable channels established among ARHs, end-tier replicas, and (possibly) clients show a short latency and a low loss rate;

As explained below in this section, the first point enables both fast reaction to real failures within the middle-tier and infrequent false failure suspicions. This reduces unexpected service unavailability.

Once the previous point has been guaranteed, the second point maximizes the probability of a short service time for a request. In our protocol, the receipt of the first reply of an end-tier replica at the middle-tier triggers indeed the sending of the reply back to the client. This also points out an interesting tradeoff between the number of end-tier replicas (and thus also of channels between the middle and the end tiers) and the maximization of the probability of providing a short end-to-end service time.

Total Order Implementation Selection. As pointed out above, the protocol deployer is in charge of deploying the middle-tier in a distributed system region that quickly reaches and maintains synchrony bounds. This follows from the protocol run by ARHs, which, to be efficient, requires rapid termination of the TO primitive most of the time. To do so, it is important to note that TO implementations are typically built on top of *software artifacts*, i.e., software modules characterized by the properties they provide. Some examples follow:

- TO implementations built on top of an *unreliable failure detector*, e.g. $\Diamond S$, which is characterized by specific completeness (safety) and accuracy (liveness) properties [10];
- TO implementations provided for the virtual synchrony programming model, adopted by several group toolkits (e.g., [5], [8]), which rely on the specification of a *membership service* to enforce liveness [13].
- TO implementations developed on top of the “Timely Computing Base” (TCB, [39], [40]), which includes a well-specified *timed agreement service* [14].

The liveness properties of these software artifacts and the associated TO implementations are typically implemented using timeouts. TO implementations are very sensitive to the values of these timeouts whose definition is up to the protocol deployer. Differently from two-tier approaches, the proposed protocol allows the protocol deployer to account for these issues by selecting a well-defined and possibly highly controlled system region that - independently from end-tier replica deployment - lets a software artifact and thus the associated TO implementation work at their best as frequently as possible.

Let us finally remark that clients and replicas have no constraint from the point of view of synchrony requirements. As a consequence, a service deployer does not have to take this issue into account when deploying the end-tier replicas, and clients can show up in any part of the distributed system.

Garbage Collection. Two points in the proposed three-tier protocol are critical with respect to resource consumption: (i) the memory used by the sequencer service implementation (i.e., ARH) grows linearly with the number of client requests, and (ii) FO handlers store all the results computed by the co-located end-tier replicas.

To address both these issues, it is worth noting that the RR co-located with each client can be configured to ensure that (i) clients may transmit a request for a new operation only if the result of the former operation has been already received, and (ii) requests are uniquely identified through a pair composed by a unique client identifier and a local sequence number, i.e. $req.id = \langle c_i, seq_{c_i} \rangle$ (this is the approach followed in the RR implementation appearing in [33]). Using these simple serialization and request identification mechanisms, upon receiving a client request message (e.g., $req = \langle \langle c_i, seq_{c_i} \rangle, op \rangle$, where seq_{c_i} is incremented by RR each time a request is sent by c_i), ARH and FO can delete from their memories all operations and associated results pertaining requests issued by the same client and having a local sequence number seq'_{c_i} that satisfies $seq'_{c_i} < seq_{c_i}$. Furthermore, upon receiving req ARH and FO are also enabled to discard request messages from the same client that satisfy the former inequality. This follows from noting that if RR is blocking then a client c_i issuing a request req has certainly received the reply to all former request (issued with a local sequence number lower than seq_{c_i}). It is possible to show that the described mechanisms permit to bound memory consumption of both ARH and FO components by a linear function of the number of clients without impacting on the protocol correctness. Let us finally remark that implementing the mechanisms outlined above passes through a simple modification of the specification and of the design of the distributed sequencer implementation. Moreover, line 9 of the protocol of Fig. 4 has to be modified in order to forward the overall client request to FO (not just the requested operation). A complete proposal for garbage collection has been described in [32].

VIII. Related Work

In the recent years, the use of the three-tier architectural pattern for building distributed systems is gaining growing popularity in both industry and research communities.

In particular, the most relevant contribution is due to Alvisi et al. in [42]. In this work, authors exploit a three-tier architecture for implementing active software replication to tolerate *byzantine* faults of clients, middle-tier and end-tier replicas by running agreement protocols exclusively within the middle-tier, and to further exploit the tiers physical separation to enforce some degree of confidentiality (a faulty client never receives information that it is not authorized to get). The focus of this work is on *separating* byzantine agreement from end-tier replica computation *to favor infrastructure scalability and to enforce confidentiality*. In our work, we adopt a similar scheme to show how this can be used to isolate the synchrony requirements necessary for building an efficient solution, and decouple these requirements from replica deployment. Further, tolerating only crash failures enables replication of non-deterministic replicas to be handled through light changes to the codes of ARH and of FO. In particular, upon getting a result computed by an end-tier replica, each FO returns to ARH - along with the reply - a state update obtained from the same replica. ARHs store ordered state updates - e.g., using a second sequencer instance - and upon receiving the following request from a client they forward to FOs the necessary state update(s) along with the sequenced request(s), so that FOs can update the corresponding replicas and make them consistent before letting them compute the results of new requests. Let us note that this replication scheme differs from passive replication since there is no notion of a primary replica and thus no delays in the presence of the fault of a specific end-tier replica [2].

A similar architectural pattern is used by Verissimo, Casimiro and Fetzer in [40], presenting an architectural construct, namely the Timely Computing Base (TCB), for real-time applications running in environments with uncertain timeliness. The TCB assumes the existence of a small part of a system satisfying strict synchrony requirements used to implement a set of services, i.e., *timely execution*, *duration measurement* and *timing failure detection*. These services are in turn exploited by the remaining large-scale, complex, asynchronous part of the system to run only the control part of their algorithms with the support of the TCB services. Therefore, the TCB can be regarded as a *coverage amplifier of synchrony assumptions* for the execution of the time-critical functions of a system, e.g., of the TCB services. As remarked in the previous section, the agreement service provided by TCB is one of the software artifacts on which efficient TO primitives for the three-tier replication protocol are built.

In [24], Guerraoui and Schiper define a generic consensus service based on a client-server architecture for solving agreement related problems, e.g., atomic commitment, group membership, total order multicast etc. In this architecture, a set of *consensus servers* run a consensus protocol on behalf of *clients*. Authors motivate this architectural choice to favor modularity and verifiability. As in the three-tier protocol, the architecture confines to a well-defined system region the solution of agreement problems.

Three-tier systems have gained notable popularity in the transactional system area, in which these architectures are used to sharply separate the client (or presentation) logic (implemented by the client-tier), the business logic (implemented by the middle-tier), and the data (maintained in the end-tier), thus favoring isolation, modularity and maintainability. Current solutions to reliability in commercial three-tier systems are typically transactional [6], [7], [11], [41] and incur in significant overheads upon the occurrence of failures. As a consequence, recent works e.g., [17], [38], [43], compose software replication and high-availability with transaction processing in three-tier architectures. Notably, in [22], Frølund and Guerraoui adopt a three-tier architecture to coordinate distributed transactions while enforcing *exactly once* semantics despite client reinvocations. According to this scheme, the middle-tier acts as a replicated, highly-available and centralized transaction manager that coordinates distributed transactions involving a set of database managers accessed through standard interfaces.

Let us finally remark that the protocol presented in this paper is one of the results derived from the Interoperable Replication Logic (IRL) project, carried out in our department. This project is investigating the three-tier approach to software replication in different settings. Main results of this project can be found in [3], [4], [33]. Specifically, [3] exploits three-tier approach to develop a middleware platform supporting the development of fault-tolerant CORBA applications according to the FT-CORBA specification. A simple three-tier replication protocol is outlined in this paper (middle-tier replicas adopts a primary backup scheme and use perfect failure detectors). Another three-tier replication protocol, appearing in [4], [33], implements active replication through a *centralized* sequencer service. This protocol incurs a larger message complexity than the one proposed in this paper and exhibits a single point of failure that shall be removed using classical replication techniques.

IX. Conclusion

Software services replicated using a two tier approach within a partially synchronous distributed system can suffer from unexpected unavailability during periods in which timing bounds do not hold. The problem is even worse if the deployment of server replicas cannot be controlled. In this case, the only way to reduce this undesirable effect is to design replication protocols that can take advantage of regions of a large and complex distributed system that show an early-synchronous behavior. In this paper we have presented a three-tier protocol for software replication well-suited to such a setting and proved its formal correctness. The protocol aims to reduce the risk of such unexpected service unavailability by deploying the middle-tier over an "early-synchronous" region of the distributed system (e.g. a LAN in a networked distributed system) while leaving, at the same time, clients and end-tier replicas to be deployed everywhere. The main feature of this protocol is that it allows a fully distributed implementation of the middle-tier and it ensures the termination of a request/reply interaction despite the crash of all end-tier replicas but one.

Let us finally remark that the availability of the middle-tier in our protocol becomes a crucial point. However this environment is managed, then all necessary low-level measures can be taken to maximize the probability that a client is able reach the middle-tier, for example backup lines to the Internet, multiple connections to external routers etc.

Acknowledgements

The authors would like to thank Adnan Noor Mian and the anonymous reviewers for their valuable comments and suggestions that greatly improved the content of this work. The authors owe one of the anonymous reviewers a beer.

References

[1] O. Bakr and I. Keidar. Evaluating the running time of a communication round over the internet. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 243–252. ACM Press, 2002.

[2] R. Baldoni and C. Marchetti. Software replication in three-tier architectures: is it a real challenge? In *Proc. of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 133–139, Bologna, Italy, November 2001.

[3] R. Baldoni and C. Marchetti. Three-tier replication for ft-corba infrastructures. *Softw. Pract. Exper.*, 33(8):767–797, 2003.

[4] R. Baldoni, C. Marchetti, and S. Tucci-Piergiovanni. Asynchronous Active Replication in Three-tier Distribuuted Systems. In *Proc. of the 9th IEEE Pacific Rim Symposium on Dependable Computing*, pages 19–26, Tsukuba, Japan, 2002.

[5] B. Ban. Design and implementation of a reliable group communication toolkit for java. Cornell University, September 1998.

[6] P. Bernstein, V. Hadzilacos, and H. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA, 1987.

[7] P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan-Kaufmann, 1997.

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

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

[10] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving Consensus. *Journal of ACM*, 43(4):685–722, July 1996.

[11] D. Chappel. How Microsoft Transaction Server Changes the COM Programming Model. *Microsoft System Journal*, 1998.

[12] Marc Chérèque, David Powell, Philippe Reynier, Jean-Luc Richier, and Jacques Voiron. Active replication in delta-4. In *FTCS*, pages 28–37, 1992.

[13] G.V. Chockler, I. Keidar, and R. Vitenberg. Group Communications Specifications: a Comprehensive Study. *ACM Computing Surveys*, 33(4):427–469, December 2001.

[14] M. Correia, L.C. Lung, N. F. Neves, and P. Verissimo. Efficient Byzantine-Resilient Reliable Multicast on a Hybrid Failure Model. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 2–11, Osaka, Japan, October 2002.

[15] 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.

[16] Xavier 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.

[17] Z. Dianlong and W. Zorn. End-to-end transactions in three-tier systems. In *Proc. of the 3rd International Symposium on Distributed Objects and Applications (DOA01)*, pages 330 –339, 2001.

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

[19] 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.

[20] R. Friedman and E. Hadad. FTS: a High-Performance CORBA Fault-Tolerance Service. In *Proc. of the 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 61–68, 2002.

[21] R. Friedman and A Vaysburd. Fast Replicated State Machines over Partitionable Networks. In *Proc. of the 16th IEEE International Symposium on Reliable Distributued Systems (SRDS)*, October 1997.

[22] R. Guerraoui and S. Frølund. Implementing E-Transactions with Asynchronous Replication. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):133–146, 2001.

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

[24] R. Guerraoui and A. Schiper. The Generic Consensus Service. *IEEE Transactions on Software Engineering*, 27(1):29–41, January 2001.

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

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

[27] I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, Hebrew University, Jerusalem, Israel, 1994. also Tech. Rep. CS95-5.

[28] I. Keidar and D. Dolev. Efficient Message Ordering in Dynamic Networks. In *Proc. of the 15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–86, May 1996.

[29] B. Kemme and G. Alonso. A Suite of Database Replication Protocols based on Group Communications. In *Proc. of the 18th International Conference on Distributed Computing Systems (ICDCS)*, May 1998.

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

[31] S. Landis and S. Maffeis. Building Reliable Distributed Systems with CORBA. *Theory and Practice of Object Systems*, 3(1), 1997.

[32] C. Marchetti. A three-tier architecture for active software replication. Technical report, Ph. D. Thesis, Dipartimento di Informatica e Sistemistica, Università degli Studi di Roma La Sapienza, 2003.

[33] C. Marchetti, S. Tucci-Piergiovanni, and R. Baldoni. A three-tier replication protocol for large scale distributed systems. *IEICE Transactions on Information Systems: Special Issue on Dependable Computing (selection of PRDC-02 papers)*, 86-D(12):2544–2552, 2003.

[34] L. Moser, P. M. 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.

[35] L.E. Moser, P.M. Melliar-Smith, and P. Narasimhan. Consistent Object Replication in the Eternal System. *Theory and Practice of Object Systems*, 4(3):81–92, 1998.

[36] D. Powell, G. Bonn, D. Seaton, P. Verissimo, and F. Waeselynck. The delta-4 approach to dependability in open distributed computing systems. In *Proc. of the 18th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-18)*, New York, USA, June 1988.

[37] Fred B. Schneider. Replication Management Using the State Machine Approach. In S. Mullender, editor, *Distributed Systems*. ACM Press - Addison Wesley, 1993.

[38] A. Vaysburd. Fault tolerance in three-tier applications: focusing on the database tier. In *Proc. of the International Workshop on Reliable Middleware Systems (WREMI99)*, pages 322–327, 1999.

[39] P. Verissimo and A. Casimiro. The Timely Computing Base model and architecture. *IEEE Transactions on Computers - Special Issue on Asynchronous Real-Time Systems*, 51(8):916–930, August 2002.

[40] P. Verissimo, A. Casimiro, and C. Fetzer. The Timely Computing Base: Timely Actions in the Presence of Uncertain Timeliness. In *Proc. of the 1st International Conference on Dependable Systems and Networks*, New York, USA, 2000.

[41] G.R. Voth, C. Kindel, and J. Fujioka. Distributed application development for three-tier architectures: Microsoft on Windows DNA. *IEEE Internet Computing*, 2(2):41–45, 1998.

[42] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from

execution for byzantine fault tolerant services. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 253–267. ACM Press, 2003.

[43] W. Zhao, L.E. Moser, and P.M. Melliar-Smith. Unification of replication and transaction processing in three-tier architectures. In *Proc. of the 22th Interbational Conference on Distributed Computing Systems (ICDCS02)*, pages 263 –270, 2002.