Preventing Orphan Requests in the Context of Replicated Invocation*

Stefan Pleisch Arnas Kupšys André Schiper
Distributed Systems Laboratory (LSR)
Swiss Federal Institute of Technology (EPFL), CH-1015 Lausanne
{Stefan.Pleisch,Arnas.Kupsys,Andre.Schiper}@epfl.ch

Abstract

In today's systems, applications are composed from various components that may be located on different machines. The components (acting as servers) may have to collaborate in order to service a client request. More specifically, a client request to one component may trigger a request to another component. To ensure fault-tolerance components are generally replicated. Replicated components lead to the problem of a replicated server invoking another replicated server. We call this the problem of replicated invocation.

Replicated invocation has been considered in the context of deterministic servers. However, the problem is more difficult to address when servers are non-deterministic. In this context, work has been done to enforce deterministic execution. In this paper, we consider a different approach: instead of preventing non-deterministic execution of servers we discuss how to handle it. We present the problem of non-deterministic replicated invocation. Our solution then consists of sharing sufficient undo information among the replicas of the server invoking another server.

1. Introduction

In today's systems, applications are composed from various components that can be collocated but may also be located on different machines (e.g., in CORBA [22]). These components collaborate in order to service a client request. More specifically, a client request executed in one component may trigger a request to another component. While acting as a server component[1] to the client, the component at the same time assumes the role of a client, by invoking a service on another server.

To ensure that applications work even in the face of failures, replication is generally used within the components. While the problem of replicating a server and invoking a replicated server has been thoroughly studied [1, 4, 12, 24, 26], the problem of a replicated server invoking another (replicated) server has not been addressed in a satisfactory manner. We call the latter invocation a *replicated invocation*. Replicated invocation in the context of deterministic servers causes a problem of *duplicate requests*. This problem is addressed in [18], where proxies are presented to filter the requests using their ID numbers. However, the proxy solution assumes deterministic replicas. Hence, it is not applicable for non-deterministic servers, because the requests sent by the replicas of non-deterministic servers may not be identical [23]. In the context of non-deterministic servers, replicated invocation causes a different problem: the problem of *orphan requests*.

Informally, an orphan request occurs if a server processes a request from another server, but this request is not valid any more. Consider, for instance, a system where client C invokes replica R_i of replicated server R. To process $C's$ request, R_i invokes another server S, i.e., R_i itself acts as a client to server S. We denote by r_i (respectively, s) the processing on R_i (respectively, S). We say that s is a subinvocation or *nested invocation* of r_i. If no failures occur the servers update their states and R_i sends the reply to the client. However, a component may be subject to a failure. If R_i fails before sending the reply to C, C will eventually notice the failure, but not S (since S already finished the processing). The state of S will reflect invocation r_i, which has not finished properly. Hence, the state of S is inconsistent. In this case we call s an orphan request. If at this point some other client accesses S, there is a danger that the inconsistent state of S will propagate in the system. Note that the failure of S causes a different problem. Indeed, it does not result in an orphan request; rather, the state of S is no longer available.

The work in [14, 21] provides mechanisms to enforce deterministic execution. In contrast, our approach supports non-determinism while at the same time preventing orphan requests. It is based on the idea of exchanging sufficient undo information prior to the server invocation to allow other client replicas to undo the requests of failed client replicas. In contrast to [7], we do not limit our approach to

*This work has been partially supported by the IBM Zurich Research Laboratory and by OFES under contract number 01.0537-1 as part of the IST REMUNE project (number 2001-65002).

[1] In the following, a server component is called a *server*.

three-tier architectures (consisting of presentation, application logic, and data tier) [17] and stateless clients. Rather, we assume that client replicas R_j do maintain their own state. Moreover, we show that our approach allows us to prevent blocking when the server uses locking to ensure concurrency control [11]. Indeed, a failure of the client in such a scenario may prevent the termination of the transaction on the server, and thus no other client can access the locked data items.

The rest of this paper is structured as follows: We first introduce replicated invocation in Section 2. In Section 3 we specify the problem of replicated invocations in terms of transactions. Using the transaction model, an orphan request becomes an orphan subtransaction (Section 4). The core contribution of the paper is presented in Section 5. In this section, we present an orphan-subtransaction-free replicated invocation protocol in the context of non-deterministic execution. Wie argue about correctness issues in Section 6 and provide a brief qualitative evaluation in Section 7. Finally, we relate our solution to the existing work in Section 8 and conclude the paper in Section 9.

2. Replicated Invocation

We assume an asynchronous system which has no bounds on communication delays nor relative processing speeds. Processes (i.e., servers and clients) can crash and do not recover.[2] In such a system, accurate failure detection is impossible and consensus cannot be solved [6]. However this issue is orthogonal to the problem addressed in the paper. Indeed, the requirements for the failure detection mechanism are given by the existing building blocks used in our approach. Processes communicate via quasi-reliable channels: If processes p and q are correct (i.e., do not crash) and p send message m to q, then q eventually receives m. Note that quasi-reliable channels provide a more accurate model for TCP connections than reliable communication channels, which do not require p to be correct. We assume that the communication channels are FIFO.

Replication is a widely used technique to address failures of a server. Instead of relying only on a single server, the service is provided by multiple server replicas. If a failure of one server replica occurs, another replica takes over and services the clients' requests. As a consequence, the service is available to the clients despite of a failure. We say that a client invokes a replicated server or rather a service on it. If the client itself is replicated, we speak of a *replicated invocation*.

Definition 1 (Replicated Invocation) *A replicated invocation occurs if a replicated client invokes a server (not necessarily replicated).*

[2] Actually, crashed processes can recover with a different process ID. From the application's perspective this corresponds to a new process.

In Fig. 1, the invocation from replicated server R to server S is a replicated invocation. If S is replicated, we do not make any assumptions about the replication strategy that can be used by S (e.g., passive [12], active [26], semi-passive [4], or semi-active [24]), as the replication strategy of S is not relevant for the contribution of the paper. Indeed, although S may be replicated, its replication strategy makes it behave like a non-replicated server from the point of view of R. For simplicity we thus represent server S as a single, non-replicated server, which is sufficient to illustrate the problem addressed in the paper and our solution. However, in real systems S would be replicated in order to prevent the existance of a single point of failure.

The replicated invocation problem can be addressed in the context of a deterministic or a non-deterministic server, R. Non-determinism can occur with respect to communication and computation. The former is caused by a different order of message arrivals at the replicas. The latter, i.e., non-determinism related to computation, occurs if the replica, for instance, is multithreaded, uses asynchronous system calls (e.g., interrupts), or invokes non-deterministic functions. A *deterministic server* (or *non-deterministic server*) is deterministic (non-deterministic) with respect to its computation, but not necessarily with respect to communication. In other words, the order in which client requests arrive at a deterministic server is arbitrary.

2.1. Deterministic Servers

Server replicas are said to be deterministic if, given the same initial state and the same request, all transit to the same state and return the same reply. We show that orphan requests do not occur with replicated deterministic servers.

In the case of deterministic servers *active replication* [26] can be used. In active replication clients multicast (using total order multicast) the request to all server replicas, which process the requests in parallel (see Fig. 1, ①). If this processing requires the invocation of another server, each replica issues exactly the same invocation ②. Because these invocations are identical, duplicate invocations can easily be detected and filtered, in order not to process them multiple times ③. This is done by having the replicas R_i assign IDs to their invocation.[3] Duplicate invocation filtering is addressed for instance in [18, 21]. The result of the processing on S is valid for every replica R_i and is multicast to them ④. Also, each replica R_i sends the reply back to the client C ⑤. Generally, the client accepts the first one and discards the others.

As long as there is at least one *correct* (not failed) replica R_i, the orphan request problem does not occur with repli-

[3] One could argue that the client can assign a unique ID to its invocation, which can be reused for the nested invocations as well. Unfortunately, this does not always work. Indeed, assume that processing on R leads to a number of invocations to S that is not known a priori. Hence, the request ID must be assigned by R.

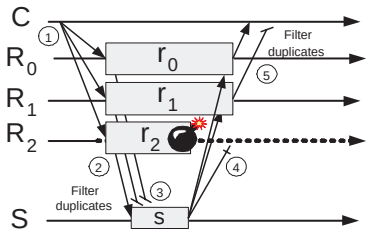

Figure 1. Deterministic server R is replicated using active replication. Horizontal arrows denote elapsed time.

cated deterministic server R.[4] The reason is that all replicas share the same request s (see Fig. 1) and thus the failure of one or multiple R_i does not leave s as an orphan.

2.2. Non-Deterministic Servers

Non-deterministic execution of R_i prevents the use of active replication; rather, it requires another replication strategy, such as passive (also called *primary-backup* [1]), semi-passive, or semi-active replication. Without loss of generality, we discuss in this paper the use of passive replication for the server R. However, our approach is also valid in case the server R is semi-passively or semi-actively replicated. In passive replication only one replica, the *primary*, executes C's request. The update is then sent to the backup replicas. The backup replicas do not directly communicate with C; rather, they only communicate with the primary. As only the primary executes the request, passive replication supports non-deterministic execution. However, a passively replicated server needs to handle failures of the primary. If the primary fails or is erroneously suspected, one of the backups takes over the role of the primary. The client C eventually times out, has to learn the identity of the new primary, and reissues the request.

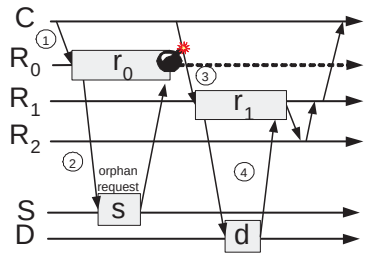

Figure 2. Non-deterministic server R is replicated using passive replication. With a failure of the primary.

Consider nested invocation in the context of the passively replicated non-deterministic server R (see Fig. 2). To service client C's request ①, the primary replica R_0 invokes server S ②, but fails before updating the backups. A new primary, say R_1, is elected and the client C resends its request ③. As the replicas of R are non-deterministic, R_1 might issue a different invocation to server S to serve the same request from C. It might even choose a different server D ④, or it might not issue the invocation at all. The result computed for r_0 thus cannot be reused for r_1, and d must be processed separately. This leaves s as an orphan request, which has a pending effect on the state of S. A new invocation of S at this point, would likely lead to an inconsistent reply. So the problem of the orphan request s needs to be addressed. In the rest of the paper we focus on replicated invocation in the context of non-deterministic replicated servers. In the next section, we introduce the specification and notation we use to model this problem.

3. Specification of Replicated Invocation with Non-Deterministic Servers

In this section, we give a specification of replicated invocation in terms of transactions. The problem of orphan requests is caused by the partial execution of C's request. As the transaction model addresses the issue of atomicity of a set of operations it is useful to model replicated invocation. Informally, a transaction always terminates by either committing its modifications, or aborting them.

Transactions (recursively) decomposed into subtransactions are called *nested transactions* [19]. Every subtransaction forms a logically related subtask. A successful subtransaction becomes permanent, i.e., commits, if all its parent transactions (the transaction that encompasses this subtransaction) commit as well. In contrast, a parent transaction can commit (provided its parent transaction commits) although some of its subtransactions may have aborted. A subtransaction is ready to commit, if it has successfully executed and is waiting for the commit or abort decision of its parent transaction. A ready-to-commit transaction t, denoted $ReadyToCommit_t$, can no longer spontaneously abort (i.e., itself decide abort), but only aborts if its parent transaction aborts. Finally, $\rightarrow$ denotes the precedence operator as specified in [15]. More specifically, if $t_1 \rightarrow t_2$, t_1 is executed before t_2. In other words, any operation of t_1 that conflicts with an operation of t_2 is executed before that operation of t_2.

We first specify the invocation between the client C and the server R (denoted $C \longleftrightarrow R$), i.e., the traditional passive replication approach, in terms of transactions (Section 3.1), and then extend this specification to also encompass the invocation between server R and server S (denoted $R \longleftrightarrow S$) in Section 3.2.

3.1. Invocation $C \longleftrightarrow R$

We model the execution of C's request on server R (see Fig. 2) as follows. Upon reception of C's request, the pri-

[4] Usually, replication techniques in the asynchronous system model assume that a majority of replicas do not fail [12].

mary replica R_0 starts transaction t_0 (see Fig. 3). This transaction contains subtransactions pr_0 (pr stands for *processing*) and up_0 (up stands for *update*). For the moment, we ignore the invocation to server S in Fig. 3. Subtransaction pr_0 executes the client request on the primary, subtransaction up_0's task is to update the backup replicas of R, i.e., R_1 and R_2.

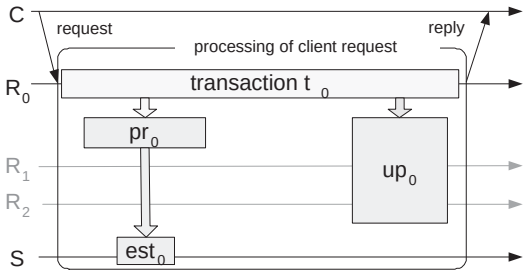

Figure 3. Representation in terms of (sub)transactions. Invocations between C, R, and S.

The specification is stated in terms of properties of transactions. We mention only those that are related to the replicated invocation, and omit basic transaction properties. The full set of properties for nested transactions can be found in [3]. The invocation $C \longleftrightarrow R$ can be specified as follows (the subscript $prim$ refers to the primary replica, e.g., R_0 in Fig. 3):

1. (*Abort Atomicity*) If transaction t_{prim} aborts, all of its subtransactions (i.e., pr_{prim} and up_{prim}) must abort.
   $Abort_{t_{prim}} \Rightarrow (Abort_{pr_{prim}} \wedge Abort_{up_{prim}})$

2. (*Cruciality*) If one of subtransactions pr_{prim} or up_{prim} aborts, t_{prim} also aborts: pr_{prim} and up_{prim} are crucial for t_{prim}. Transaction t_{prim} doesn't make sense without either one of these transactions.
   $Abort_{pr_{prim}} \vee Abort_{up_{prim}} \Rightarrow Abort_{t_{prim}}$

3. *(Sequence)* Transaction pr_{prim} always executes before up_{prim}. This is because up_{prim} updates the backups with the results of the execution of pr_{prim}.
   $pr_{prim} \rightarrow up_{prim}$

4. *(Termination)* If R_{prim} executes t_{prim}, then all correct replicas of R eventually know the outcome (i.e. commit or abort) of t_{prim}. This is modeled by either an abort or a commit of transaction up_{prim}.
   $ReadyToCommit_{t_{prim}} \Rightarrow (Abort_{up} \vee Commit_{up})$

5. *(Non-triviality)* If both subtransactions pr_{prim} and up_{prim} have successfully executed (i.e., are ready to be committed), eventually transaction t_{prim} is committed.
   $ReadyToCommit_{pr_{prim}} \wedge ReadyToCommit_{up_{prim}}$
   $\Rightarrow Commit_{t_{prim}}$

Property 1 is a standard nested transaction property. The success of subtransactions pr_{prim} and up_{prim} is crucial for the success of t_{prim}: in other words, t_{prim} can only commit if the processing transaction pr_{prim} and the update transaction up_{prim} of the backup replicas have succeeded (Property 2).

The sequence property (Property 3) is inherited from passive replication: first the client request is processed on the primary, then the backups are updated with the result obtained from the processing. Note that this property, together with Property 2, specifies a particular case of nested transactions, namely a distributed flat transaction [11]. We use the nested transaction model because it is needed when we extend our specification to the invocation between R and S in Section 3.2.

The termination property (Property 4) ensures that once transaction t_{prim} is ready to commit, its subtransaction up_{prim} eventually terminates by either commit or abort. Although the primary may fail, transaction up_{prim} eventually must be terminated. As a consequence, the other replicas need to somehow learn of the failure of up_{prim}. Property 4 is thus also a liveness property, which ensures that the outcome of transaction t_{prim} is eventually decided and that all subtransactions executing on correct processes eventually terminate. This property is essential in preventing orphan requests or subtransactions.

Finally, the non-triviality property (Property 5) specifies, that if both subtransactions pr_{prim} and up_{prim} are ready to be committed, then the outcome of t_{prim} is commit. Note that we do not require that t_{prim} be committed by R_{prim} (where t_{prim} executes), as R_{prim} may have failed. Moreover, the specification still allows R_{prim} to always immediately abort pr_{prim} despite this property.

In our system model, we assume that crashed processes do not recover[5] (see Section 2). Consequently, the failure of a replica R_{prim} erases all traces of the transaction on R, unless the other replicas have been updated.

3.2. Invocation $R \longleftrightarrow S$

In the previous section, we have specified the invocation between C and R. In this section, we extend this specification to the cases where the primary R_0 invokes transaction est_0 (*external server transaction*) on another server S (see Fig. 3). The invocation between R and S corresponds to the replicated invocation presented in Section 2.2. Transaction est_0 is a subtransaction of transaction pr_0. Recall that server S is represented as a single, non-replicated server. For our discussion, it is not relevant whether S is replicated or not.

Compared to the model of the invocation $C \longleftrightarrow R$, the invocation of S adds another level of nesting. Indeed, the

[5]This is the standard assumption that forces a protocol to be non-blocking. In other words the protocol presented later is non-blocking.

subtransaction pr now contains subtransaction est. While subtransactions pr and up are crucial for the successful outcome of t, subtransaction est may not be. In other words, in some applications pr can commit although est aborts.

Replicated invocation between R and S can thus be specified by the properties mentioned in Section 3.1 and the following two additional properties:

6. (*Remote Abort Atomicity*) If subtransaction pr_i is aborted, est_i is aborted as well.
   $Abort_{pr_i} \Rightarrow Abort_{est_i}$

7. (*Remote Commit Atomicity*) If subtransaction est_i has successfully executed and is ready to be committed, and its parent transaction pr_i commits, then est_i is also committed.
   $ReadyToCommit_{est_i} \wedge Commit_{pr_i} \Rightarrow Commit_{est_i}$

Properties 6 and 7 ensure that subtransaction est_i eventually is terminated, i.e., either commits or aborts. Clearly, we assume here, that server S is available, which is the case if S is fault-tolerant (i.e., if S is itself replicated). Note that properties 6, 1, and 2 specify that if up aborts then est must also be aborted.

4. The Problem of Orphan Subtransactions with Replicated Invocation

According to Properties 1 to 7 the outcome of the entire execution (i.e., commit or abort) is decided by the top-level transaction, and then this decision is propagated to the subtransactions, which in turn propagate it to their subtransactions. However, the failure of a replica R_i may interrupt the mechanism that notifies the subtransactions of the commit or abort decision. In this case, est_0 is unaware of the outcome of t_0 and thus cannot terminate (see Fig. 3): est_0 is called an *orphan subtransaction*. Clearly, orphan subtransactions are undesirable, because they maintain locks on data items and prevent other transactions from accessing these items. Note that subtransaction est_0 cannot spontaneously abort, because its parent transaction decides the final outcome.

Depending on the processing of server S (optimistic or pessimistic) orphan subtransactions cause different problems.

4.1. Pessimistic $vs.$ Optimistic Server S

To ensure transaction atomicity, data items are locked. When a transaction starts, the needed locks are acquired; when the transaction finishes, the locks are released, and the result of the processing becomes visible in the system. If the locks are not available for some transaction *t*, the processing blocks until the locks are released by the transaction holding the locks. A *subtransaction* holding the locks has two options upon finishing its processing: (1) it can release the locks immediately (i.e., temporary commit), or (2) it can wait for the commit/abort decision from a higher entity (i.e., parent transaction), keeping the locks on the data. The latter option, i.e., option (2), is called *pessimistic processing*, and the server is called a *pessimistic server*. Option (1) is called *optimistic processing*, and the server is called an *optimistic server*.

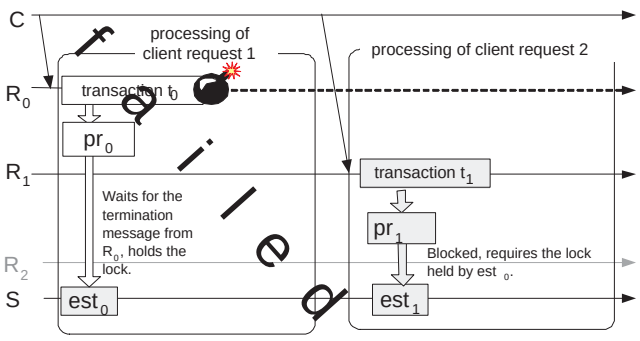

Figure 4. An orphan subtransaction est_0 on pessimistic server S.

Blocking with Pessimistic Server S. Consider the case of an orphan subtransaction (Fig. 4) with a pessimistic server S. Assume that after the crash of the primary R_0, the new primary R_1 calls the same server S, and executes subtransaction est_1, which accesses some of the same data items accessed by est_0. In this case est_1 has to wait until est_0 releases the locks. Hence, the entire client R is blocked. Blocking of R is undesirable, as it acts itself as a server for other applications. Moreover, other clients may also block when accessing server S.

Inconsistency with Optimistic Server S. The problem with optimistic servers is different: the temporary commit might have to be undone. This can be handled by *compensating actions*: to abort a committed transaction a *compensating transaction* [9, 10] is executed on the server. A compensating transaction t^{comp} semantically undoes the modifications caused by the original transaction. Assume, for instance, that transaction *t* reserves a ticket on a flight, then t^{comp} simply cancels this reservation.

Using an optimistic approach, blocking is prevented. Indeed, the locks held by subtransaction est_0 (Fig. 5) are immediately released and the data items are again accessible by est_1 (unless another transaction has acquired them in the meantime). However, in this case, subtransaction est_0 needs to be compensated, as the state of server S reflects est_0, but after the crash of R_0, est_0 is not valid any more. In the next section we present solutions for pessimistic and optimistic servers.

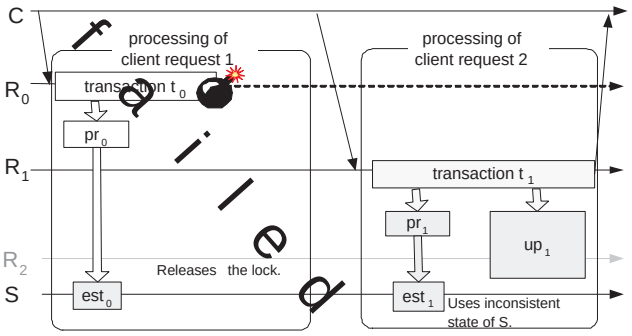

Figure 5. An orphan subtransaction est_0 on optimistic server S.

5. Replicated Invocation Protocol

In the previous section we have used transactions to model the problem of replicated invocation in the context of a passively replicated client R. In particular, we have used orphan subtransactions as a model of orphan requests. In this section, we show an approach that prevents orphan requests. For this purpose, we first present the basic idea to solve the problem of orphan requests/subtransactions in the context of replicated invocation, and then the protocol that implements this idea.

5.1. Basic Idea: Sharing Undo Information Among Replicas of R

The Problem of Finding Out About S. To prevent orphan requests in replicated invocations, it is crucial that a new primary replica R_j is able to find out the identities of the servers that have been accessed by the previous primary R_i. This allows R_j to send abort message(s) or compensating transaction(s) to S. How can S be known to R_j? The identity of S is trivially known if (1) it can be deterministically computed by the replicas of R, or (2) the set of servers is sufficiently small. Note that in case (1), the replicas still may generate different requests to S, or not send any request to S at all. In case (2), a message is sent to all servers to find out which one has been invoked by R_j. In the following, we address the more complex cases in which the identity of S cannot be deduced a posteriori. This is especially the case if

- the identity of S is dynamically computed during the processing of R_j. In other words, the identity of S is not known to the replica prior to the processing of C's request, and it is impossible for R_j to find out the identity of S computed by another replica R_i, and
- the set of potential servers is large.

Sending Undo Information. We call *undo information*, the information that allows a particular request to be undone; it includes the name of the server S to which the request is sent, and the description of an action to perform. The solution to orphan requests consists of making the undo information available to other replicas of R before the primary invokes S. In the context of the undo information, we distinguish between *termination requests* and *compensation requests*:

(1) unterminated orphan requests (or subtransactions) on pessimistic servers need to be terminated, and

(2) terminated orphan requests (or subtransactions) on optimistic servers need to be compensated.

In case (1), *termination requests* are `COMMIT` and `ABORT` messages.[6] In case (2) compensating actions are included in the undo information in order to restore the consistent state of the system. However, note that compensating the request of a replica is not easy. For example the sequence of requests $(rq_x; rq_y; rq_x^{comp})$ must be a valid sequence and must be *semantically* equivalent to the sequence that consists only of rq_y. Note that this is not a consequence of our solution; rather, this assumption is required also in the case S is accessed by multiple different clients.

If a new primary R_j is elected as a result of an erroneous suspicion of the old primary R_i, R_i can itself send the termination or compensation request to S, if needed. However, from the perspective of the replicas R_k $(k \neq i)$ it is impossible to distinguish between an erroneous and a correct suspicion of R_i (see Section 2).

5.2. The Protocol

The *Replicated Invocation Protocol* for non-deterministic execution is presented in Figures 6 and 7. The protocol consists of six procedures and one task executed on the primary of R. When the primary gets a request from client C, Procedure 1 is executed. If the request was not processed previously, the primary starts processing it. This corresponds to transaction t_0 in our model (see Fig. 3). After executing procedures *Process_Request* (Procedure 2) and *Update_Backups* (Procedure 3) the processing on remote pessimistic servers must be committed. Procedure 1 terminates after sending the reply to the client.

Procedure 2 corresponds to transaction pr_0 in our model (see Fig. 3). Assume that during processing, the primary needs to send a nested request to some other server S. Before doing so, a message of type *UndoInfo* is prepared for that request and sent to the backups. The content of the undo message depends on the type of server the original request is sent to. Upon reception, the undo information messages are stored locally on backups R_j in the set U_j. Then, server S is invoked.

[6] We assume that the execution of termination requests is idempotent.

```
New Message TYPE StandardRequest = {req, ID};
   req - (the request to be sent);
   ID - (ID, which uniquely specifies the request);

New Message TYPE UndoInfo = {comp, reqID, parentID, target};
   comp - (compensating request, used only with optimistic servers);
   reqID - (ID of the request to S this undo information corresponds to);
   parentID - (ID of the request from client C, whose processing
              triggered the undo message);
   target - (the server, to send this undo message to, if needed);

Pessimistic(S) - (predicate that evaluates to true if S is pessimistic);
Optimistic(S) - (predicate that evaluates to true if S is optimistic);
```

Figure 6. Message type declaration and predicate definition.

Procedure 3 multicasts the result of the request processing to the backups (this multicast is denoted by *Uniform-VScast*).[7] It corresponds to the transaction up_0 in our model (see Fig. 3) and to uniform VScast traditionally used in passive replication.[8]

Procedure 4 is called when a replica becomes a primary, which occurs if the previous primary fails or is wrongly suspected to have failed. Before starting to serve the clients' requests, the new primary takes care of orphan requests.

Managing orphan requests in Procedure 5 depends on the type of server: pessimistic or optimistic. In the next two subsections, we describe each case separately. We assume the same system as in Fig. 2, i.e. R_0 is the initial primary. If R_0 crashes, R_1 takes the role of the primary.

Procedure 6 enables garbage collection of the undo information on the backups. Undo information u can only be garbage collected when it is ensured that S eventually applies the undo request related to u. Hence, the primary needs to wait until it receives an acknowledgement (ACK) from S, which indicates that the undo request with ID $u.reqID$ has been delivered by S. Note that server S needs to send an ACK back also for duplicate undo requests, which may occur if the former primary crashed before it was able to execute Procedure 6 and the new primary resends the undo request.

Periodically, the primary multicasts (using uniform VScast, mentioned above) the set of undo request IDs whose undo information has become obsolete (Task 1). In a prac-

[7] In the context of group communication, this multicast corresponds to what is called *uniform view synchronous broadcast* [2, 25]. Roughly speaking, uniform view synchronous broadcast ensures that if some process VSdelivers the message, then all correct processes eventually VSdeliver the message. For simplicity, we assume Sending View Delivery [2] as provided by the algorithm in [25]. However, our approach can be easily extended to encompass also Same View Delivery. More information about using group communication for passive replication can be found in [12]. We do not discuss these issues here, since they are not needed to understand the contribution of the paper.

[8] If processes can communicate over *reliable* communication channels, then uniform VScast is not needed to send the undo information to the backups. Rather, simple point-to-point communication is sufficient.

```
1  r : StandardRequest;
2  u : UndoInfo;
3  U_prim : set of UndoInfo messages;
4  UReqIDs : set of IDs for Garbage Collection;
5
6  Procedure 1. Upon reception of request r from C on
7               the primary R_prim:
8    if update for request with ID=r.ID is available then
9      send(reply for r) to C;
10   else
11     Process_Request(r);
12     Update_Backups(update for r);
13     for every (u : u ∈ U_prim and u.parentID = r.ID) do
14       if Pessimistic(u.target) then
15         send(COMMIT, u.reqID) to u.target;
16     send(reply for r) to C;
17
18 Procedure 2. Process_Request(r) on primary R_prim:
19   ...
20   if primary needs to send nested request to S then
21     new s: StandardRequest;
22     s.req←request to S; s.ID←assign unique ID;
23     new u: UndoInfo;
24     u.parentID←r.ID; u.target←S;
25     if Pessimistic(S) then
26       u.comp←NULL; u.reqID←s.ID;
27     else if Optimistic(S) then
28       u.comp←compensating request for s;
29       u.reqID←s.ID;
30     U_prim ← U_prim ∪ {u};
31     Uniform-VScast(u);
32     wait to deliver(u);
33     send(s) to S;
34     wait for reply;
35   ...
36
37 Procedure 3. Update_Backups(update for r):
38   Uniform-VScast(update for r);
39   wait to deliver(update for r);
40
41 Procedure 4. When R_i becomes a primary:
42   for every (u : u ∈ U_prim)do
43     Manage_Orphan(u);
44
45 Procedure 5. Manage_Orphan(u):
46   if update for request with ID=u.parentID is available then
47     if Pessimistic(u.target) then
48       send(COMMIT, u.reqID) to u.target;
49   else (* if update is not available *)
50     if Pessimistic(u.target) then
51       send(ABORT, u.reqID) to u.target;
52     else if Optimistic(u.target) then
53       send(u.comp, u.reqID) to u.target;
54
55 Procedure 6. Upon reception of ACK for u.reqID from S:
56   UReqIDs ← UReqIDs ∪ {u.reqID};
57   U_prim ← U_prim \ {u};
58
59 Task 1. Garbage_Collection:
60   loop(* do periodically *)
61     Uniform-VScast(UReqIDs);
62     wait to deliver(UReqIDs);
63     UReqIDs ← ∅;
```

Figure 7. Replicated invocation protocol.

tical setting, the messages related to garbage collection can be piggybacked onto the messages of the next uniform VScast in Procedure 3. Upon reception of the messages related to garbage collection (not shown in Fig. 7) the backups discard all the corresponding undo information. If they have not yet received the undo information that corresponds to a particular undo request ID, they store the undo request ID for later use.

Pessimistic Server S. If server S executes pessimistically, a termination message is always required. Indeed, assume that primary R_0 fails after updating the backups, but before sending the result to the client. In this case, as the new primary R_1 has received the update, a `COMMIT` message is sent to S together with the ID of the request to be committed. In contrast, an `ABORT` message is sent to S by R_1, if R_0 fails before it updates the backups (see Fig. 8). When C resends its request, this request is executed by R_1.

A particular case arises if R_0 fails after having sent the undo information to the backup replicas, but before sending the request to S. As the backup replicas have received the undo information u (see Procedure 2, lines 31-32), the new primary will use this undo information to send a termination message to S (see Procedure 5, line 51). Similarly, the termination message may arrive at S before the original request. Server S must handle this case: if the original request has not been received, then the termination message is not executed, but stored to be reused in case it eventually arrives (if it does at all). Such early termination messages are possible even if R_0 fails after sending the original request.

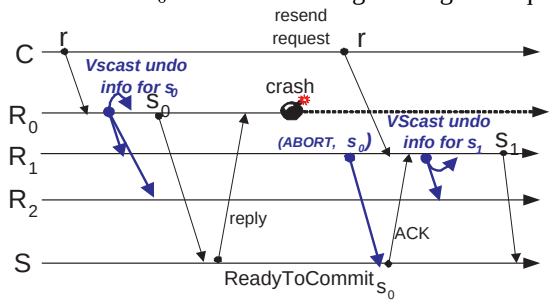

Figure 8. Primary R_0's failure after invoking pessimistic server S. The primary fails before backups are updated.

Optimistic Server S. To undo request s that has been sent to the optimistic server S, a compensating request is used. Consider first the case where no compensating request is required. In this case, the primary (i.e., R_0) executes C's request (which requires the sending of a request to S), updates the backups, and crashes. As the state of the backups has been updated, the new primary R_1 simply returns the result previously computed by R_0 when C resends its request.

However, if R_0 fails before updating the backups, the processing on S needs to be undone. Hence, a compensating request is sent to server S. Eventually, C resends its request to the new primary R_1, which recomputes the result. Note that the order of compensating an original request is not significant. This is a consequence of the properties of the compensating request (see Section 5.1).

Similarly to the case of a pessimistic server, optimistic server S also needs to store undo messages that arrive before the corresponding original request. Moreover, duplicate undo messages are ignored.

6. Correctness Issues

In this section, we argue about the correctness of our approach. Basically, we have to show that our algorithm satisfies Properties 1 - 7. As Properties 1 to 5 are closely related to uniform VScast and passive replication, the reader is referred to [25, 12]. Hence, we only give an informal proof of properties 6 and 7 here. For this purpose, we first prove the following lemma:

Lemma 1 *If primary R_i VSdelivers undo information u and a new primary R_j $(j \neq i)$ takes over (because of a failure of R_i or an erroneous suspicion), R_j has also VSdelivered the undo message u.*

The result follows directly from the property of uniform VScast. □

Note that the uniformity is needed here. Indeed, assume that the primary VSdelivers the undo message u, invokes S and then fails. Although the primary has failed, the other group members must VSdeliver u to prevent orphan requests on S.

From this lemma, the proof of Properties 6 and 7 is immediate. When R_j becomes primary, it first processes all undo information by sending it to the corresponding servers $S, \ldots$. This and the fact that undo information u is only garbage collected when an ACK for $u.reqID$ has been received from S ensure that Properties 6 and 7 are satisfied.

7. Evaluation

7.1. Message Costs

Compared to the costs of passive replication (more specifically uniform VScast) our mechanism adds an additional overhead. We are interested in the costs of the execution in which no processes fail or are erroneously suspected by the failure detection mechanism (which generally is the case most of the time) and (1) compute the total number of messages and (2) the messages in the critical path of the execution. A message is in the critical path if the algorithm cannot proceed until this message is received.

We assume that R consists of n replicas. The total number of additional messages is $3(n-1)$ and corresponds to

the cost of uniformally VScasting undo information u [25]. Among these messages, $2(n-1)$ are in the critical path.

The costs of the undo messages are added to every single remote server invocation. Hence, response time with respect to the client request increases. On the other hand, these messages are usually very small and can be piggybacked onto update messages of other requests to reduce the total number of messages, however, at the cost of increased response time.

7.2. Limitations

The approach presented in Section 5.2 has two limitations. However, we believe that these limitations are inherent to the replicated invocation itself, and not at all related to our solution.

The first drawback is that server(s) S are not allowed to spontaneously abort unterminated invocations. In our solution, the client replicas R are responsible for terminating pending invocations, and the server(s) S relies entirely on the replicas R. In other words, the server(s) S must trust the clients R to do their job.

A pessimistic server S needs to support the abort/commit of a transaction (i.e., invocation) by another process than the one that has issued the invocation (see Section 5.2). To our knowledge, although a mechanism to pass on the responsibility for a transaction to another process is foreseen in the XA Specification for distributed transaction processing [13], this mechanism seems not to encompass the situation where processes fail. Rather, in this case, the unterminated transaction is simply aborted.

8. Related Work

Most of the work performed in the context of replicated invocation assumes deterministic execution [16, 18, 27].

Arjuna [16] uses active replication and thus assumes deterministic execution of the replicas.

Mazouni's work [18] addresses transparency of the replication technique in the context of replicated invocation. More specifically, the replication mechanism of the client needs to be hidden from the server, and vice-versa. Mazouni advocates the use of proxies to achieve transparency, for both the invocation and the reply to the invocation. Hence, a proxy is located with each client and server replica. To achieve transparency, these proxies also filter duplicate invocations and results, assuming that the clients and the actively replicated servers are deterministic.

Zhao, Moser, and Melliar-Smith [27] unify fault-tolerant CORBA and the CORBA Object Transaction Service in the context of a three-tier architecture. Their work also assumes deterministic execution. The proposed infrastructure replicates transactional application servers (application-logic tier) to protect them from failures. Moreover, they are augmented with an automatic transaction retry mechanism, which in the case of failure prevents the client from reissuing the request (this prevents duplicate invocations from the client tier). Replicated gateways are introduced between the application-logic tier and the data tier: they are responsible for filtering duplicate invocations and manage transaction retry. If a failure occurs and an ongoing transaction is not $ReadyToCommit$, the infrastructure, transparently to the client, aborts and retries the transaction. For this purpose, the state of all objects involved in the transaction is checkpointed [11].

In contrast, Narasimhan enforces determinism (in the context of multithreaded applications) instead of assuming determinism. The work was performed in the context of Eternal [21], a replication infrastructure for CORBA objects. Determinism is enforced by allowing only a single logical thread of control within each replica. Although multiple threads may exist within the replicas, all of them relate to the same logical thread of control. Consistent dispatching of threads within replicas is achieved using a deterministic operation scheduler.

Jimenez et al. [14] enforce determinism of transactional multithreaded replicas in the context of active replication. More specifically, they identify two levels of non-determinism: external and internal. External non-determinism corresponds to non-determinism related to communication, while internal non-determinism relates to computation (see Section 2), in particular thread scheduling. External non-determinism is handled using totally ordered multicast. Internal non-determinism is addressed with deterministic thread scheduling and selective message reception from two-level queues. In [14], no replicated invocation is considered in this context.

Frølund and Guerraoui present a correctness criterion for exactly-once in the context of replication [8], that also addresses non-determinism in the execution and external side-effects. They also propose a replication protocol, called *asynchronous replication* [7]. The protocol is targeted towards the classical three-tier architecture, with slim client, stateless application servers, and databases. In contrast, our approach is more general in that it also addresses stateful components (our approach does not make the distinction between clients and servers). Rather, any client can at the same time act as a server for another client. Assuming stateful components clearly leads to stronger requirements, e.g., the update of all replicas.

A large body of work in the context of checkpointing and rollback recovery exists [5]. However, even if undo messages appear in our paper and in checkpointing/rollback recovery, the issues are only loosely related. Checkpointing techniques do not address availability: progress is only possible upon recovery. In contrast, the paper addresses issues in the context of replication, a technique masking failures, i.e., allowing progress even while processes are down.

9. Conclusion

In the paper we have presented the problem of orphan requests. In the context of replicated invocations, orphan requests occur when a server replica R_i invokes another server S, but fails before updating the other replicas R_j $(j \neq i)$. Hence, the results of the execution on R_i are lost. As the state of server S reflects the invocation by R_i, the state of S may become inconsistent with respect to the other replicas R_j. To our knowledge, this problem, which is easily addressed with deterministic replicated servers R [18], has not been solved in the context of non-deterministic replicated servers. In this paper, we propose a protocol for preventing orphan invocations based on undo information shared by R_i with replicas R_j. More specifically, R_i sends undo information to its replicas before issuing the nested invocation to S. Based on this undo information, another replica R_k $(k \neq i)$ can undo R_i's invocation on S in case R_i fails or is erroneously suspected. Our protocol handles both pessimistic and optimistic execution of the invocation on S.

In the future, we plan to quantitatively evaluate our approach and compare its overhead to deterministic execution. Also, by studying in more detail the sources of non-determinism [23], relaxed schemes of our approach may yield better performance in particular application contexts.

References

[1] N. Budhirja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. In Mullender [20], pages 199–216.

[2] G. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 4(33):1–43, December 2001.

[3] P. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems (TODS)*, 19(3):450–491, 1994.

[4] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proc. of the 17th Symposium on Reliable Distributed Systems (SRDS'98)*, pages 43–50, West Lafayette, Indiana, Oct. 1998.

[5] E. Elnozahy, L. Alvisi, Y.-M. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[6] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. In *Proc. of 2nd ACM Symposium on Principles of Database Systems*, pages 1–7, Atlanta, Georgia, Mar. 1983.

[7] S. Frølund and R. Guerraoui. Implementing e-transactions with asynchronous replication. *IEEE Transactions on Parallel and Distributed Systems*, 12(2):133–146, Feb. 2001.

[8] S. Frølund and R. Guerraoui. X-ability: a theory of replication. *Distributed Computing*, 14(4):231–249, Dec. 2001.

[9] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of ACM SIGMOD Int. Conference on Management of Data and Symposium on Principles of Database Systems*, pages 249–259, 1987.

[10] J. Gray. The transaction concept: virtues and limitations. In *Proc. of Int. Conference on Very Large Databases*, pages 144–154, Cannes, France, 1981.

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, USA, 1993.

[12] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, April 1997.

[13] ISO/IEC. *Information Technology - Distributed Transaction Processing - The XA Specification*, 1st edition, 1996. ISO/IEC 14834.

[14] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arevalo. Deterministic Scheduling for Transactional Multithreaded Replicas. In *Proc. of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, pages 164–173, Nuremberg, Germany, Oct. 2000.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[16] M. Little and S. Shrivastava. Replicated k-resilient objects in Arjuna. In *Proc. of 1st IEEE Workshop on Replicated Data*, pages 53–58, Houston, Texas, Nov. 1990.

[17] C. Loosley and F. Douglas. *High-Performance Client/Server*. Wiley Computer Publishing, New York, USA, 1998.

[18] K. Mazouni, B. Garbinato, and R. Guerraoui. Filtering duplicated invocations using symmetric proxies. In *Proc. of 4th Int. Workshop on Object Orientation in Operating Systems (IWOOOS'95)*, Lund, Sweden, Aug. 1995.

[19] J. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, MA, USA, 1985.

[20] S. Mullender, editor. *Distributed Systems*. Addison-Wesley, Reading, Massachusetts, USA, 2nd edition, 1993.

[21] P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, University of California, Santa Barbara, USA, September 1999.

[22] OMG. Common Object Request Broker Architecture (CORBA). http://wwww.corba.org.

[23] S. Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6(3):289–316, May 1994.

[24] D. Powell. Delta4: A generic architecture for dependable distributed computing. volume 1 of *ESPRIT Research Reports*. Springer Verlag, 1991.

[25] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *Proc. of 13th Int. Conference on Distributed Computing Systems (ICDCS'93)*, pages 561–568, Pittsburgh, Pennsylvania, USA, May 1993.

[26] F. Schneider. Replication management using the state-machine approach. In Mullender [20], pages 169–198.

[27] W. Zhao, L. Moser, and P. Melliar-Smith. Unification of replication and transaction processing in three-tier architectures. In *Proc. of Int. Conference on Distributed Computing Systems (ICDCS'02)*, pages 290–297, Vienna, Austria, July 2002.