Replicated Invocation

Stefan Pleisch* Arnas Kupsys André Schiper
Distributed Systems Laboratory
School of Computer and Communication Sciences
Swiss Federal Institute of Technology (EPFL)
CH-1015 Lausanne
{Stefan.Pleisch,Arnas.Kupsys,Andre.Schiper}@epfl.ch

Technical Report IC/2003/09

Abstract

In today's systems, application are composed from various components that may be located on different machines. The components 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. Moreover, to ensure fault-tolerance, components are generally replicated. This poses the problem of a replicated server invoking another replicated server. We call it 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 the paper we consider a different approach. Instead of preventing non-deterministic execution of servers, we discuss how to handle it. The paper first discusses the problem of non-deterministic replicated invocation. Then the paper proposes a different solution to solve these problems.

Keywords: FT Algorithms, FT Replication, Non-determinism, Passive Replication, Nested Invocation.

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 [21]). The 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 to the client, the component at the same time assumes the role of a client, by invoking a service on another server.

Consider a system where client C invokes server R (Figure 1 (a)). To process $C's$ request, server R invokes another server S, i.e., R itself acts as a client to server S. We denote by r (resp. s) the

*An important part of this work was performed while the author was working at the IBM Zurich Research Laboratory, CH-8803 Rüschlikon.

processing on R (resp. S). We say that s is a subinvocation or nested invocation of r. If no failures occur the servers update their states and R sends the reply to the client. However, a component may be subject to a failure. If R fails before sending the reply to C (Figure 1 (b)), C will eventually notice the failure, but not S (since S already finished the processing). The state of S will reflect invocation r, which is 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 more available.

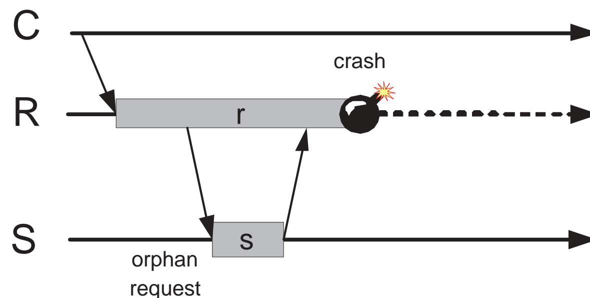


Figure 1: Nested request invocations. The processing of the request is shown by gray bars.

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 has been thoroughly studied [23, 3, 12], the problem of a replicated server invoking another server, has not been addressed in a satisfying manner. We call such invocation a *replicated invocation*. Replicated invocation in the context of deterministic servers causes a problem of *duplicate requests*. This problem is addressed in [15], where proxies are presented to filter the requests using their ID numbers. However, the proxy solution assumes deterministic replicas. Thus, it is not applicable for non-deterministic servers, because the requests sent by the replicas of non-deterministic servers may be not identical [22]. In the context of non-deterministic servers replicated invocation cause a different problem: the problem of orphan requests, presented above.

The work in [20] provides mechanisms to enforce deterministic execution. We, in contrary, suggest an approach that prevents orphan requests, even if the client replicas execute non-deterministically. Our approach 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 contrary to [8], we do not limit our approach to three-tier architectures [14] and stateless clients. Rather, we assume that the client replicas 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]. A failure of the client in such a scenario may prevent the termination of the transaction on the server, and thus any other client cannot 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. The problem of orphan subtransactions is discussed in 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. Finally, we relate our solution to the existing work in Section 6 and conclude the paper with Section 7.

2 Replicated Invocation

Replication is a widely used technique to address failures of a server. If a failure of one server replica occurs, another replica takes over and services clients' requests. If a replicated server acts as a client to another server, i.e., invokes a service on it, we call this invocation a *replicated invocation*. In Figure 2, the invocation from the replicated server R to the server S is a replicated invocation. The replicated invocation problem can be addressed in the context of deterministic or non-deterministic server R. We do not make any assumptions about the replication strategy that can be used by S (passive, active [25], semi-passive [6], or semi-active [23]), as this replication strategy is not relevant for the contribution of the paper.

2.1 Deterministic Servers

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

With deterministic servers *active replication* can be used. In active replication clients multicast the request to all server replicas, which process the requests in parallel (see Figure 2, 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[1]. Duplicate invocation filtering is addressed in [15, 17, 20]. The result of 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.

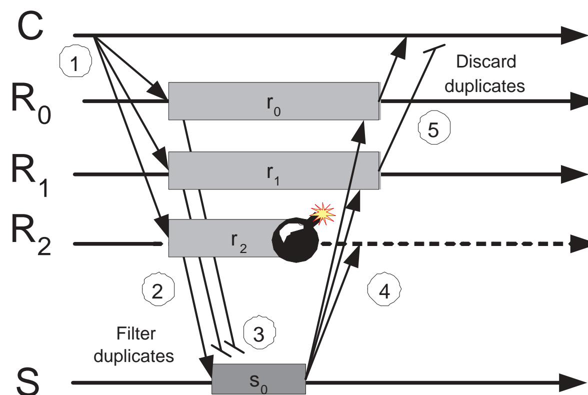


Figure 2: Deterministic server R, replicated using active replication.

The orphan request problem does not appear with replicated deterministic server R, as all replicas share the same request s_0 (see Figure 2). Consequently, the failure of one or multiple R_i does not leave s_0 as an orphan, as long as there is at least one *correct* (not failed) replica R_i.

[1]One could argue that the client can assign a unique ID to its invocation, which can be reused for the nested invocations as well. However, this does not work: the request ID must be assigned by R. Indeed, assume, that processing on R leads to multiple invocations to S: the client assigned ID is not enough.

2.2 Non-Deterministic Servers

Non-deterministic execution requires that the server R uses passive replication (Figure 3 (a)) [12]. In passive replication (also called *primary-backup* [3]) 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, passively replicated server needs to handle failures of the primary. If the primary fails, one of the backups takes over the role of the primary (Figure 3 (b)). The client C eventually time-outs, has to learn the identity of the new primary (e.g., R_1), and reissues the request.

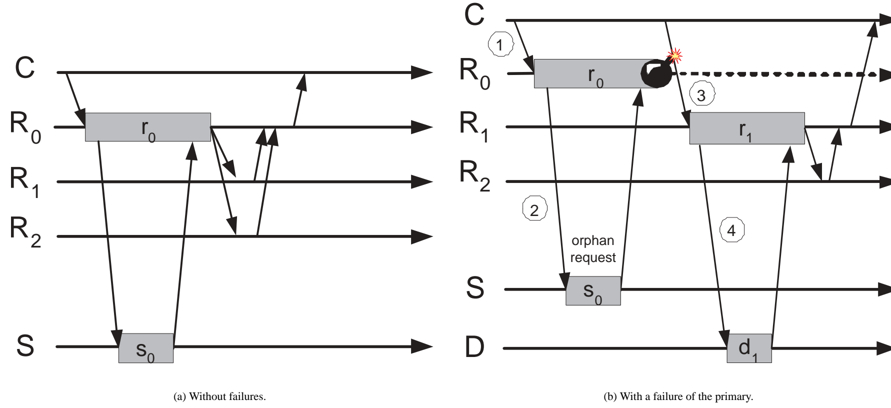


Figure 3: Non-deterministic server R, replicated using passive replication.

Consider nested invocation in the context of the passively replicated non-deterministic server R (see Figure 3 (b)). To serve client's C 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, to serve the same C's request, R_1 might issue a different invocation to server S. It also might choose a different server D ④, or it might not issue the invocation at all. The result computed for r_0 cannot be reused for r_1, and d_1 must be processed separately. This leaves s_0 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_0 needs to be addressed. In the rest of the paper we focus on replicated invocation in the context of non-deterministic replicated servers. Later in the paper we present a *replicated invocation protocol*, which handles the problem of orphan requests. But before, 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 request was caused by partial execution of C's request, thus the transaction model is useful, because it addresses the issue of atomicity of a set of operations. Informally, a transaction always terminates by either committing its modifications, or aborting them.

Transactions (recursively) decomposed into subtransactions are called *nested transactions* [18]. Every subtransaction forms a logically related subtask. A successful subtransaction becomes perma-

nent, 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 $ReadyToCommit_t$, cannot spontaneously abort any more (i.e., itself decide abort), but only aborts if its parent transaction aborts. Finally, $\rightarrow$ denotes the precedence operator as specified in [2]. 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 in terms of transactions (Section 3.1), and then extend this specification to also encompass the invocation between server R and server S (Section 3.2).

3.1 Invocation $C \longleftrightarrow R$

Consider Figure 3. We model the execution of C's request on server R as follows. Upon reception of C's request, the primary replica R_0 starts transaction t_0 (see Figure 4). This transaction contains subtransactions pr_0 (pr stands for *processing*) and up_0 (up stands for *update*). Subtransaction pr_0 executes the client request on the primary, subtransaction's up task is to update the backup replicas of R, i.e., R_1 and R_2.

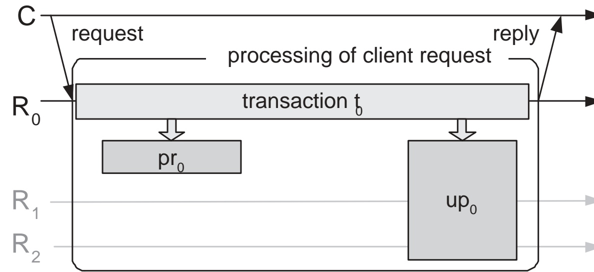


Figure 4: Representation of client's invocation in terms of (sub)transactions.

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 [5]. The invocation $C \longleftrightarrow R$ can be specified as follows (the subscript i refers to the primary replica):

1. $Abort_{t_i} \Rightarrow (Abort_{pr_i} \wedge Abort_{up_i})$
   If transaction t_i aborts, all of its subtransactions (i.e., pr_i and up_i) must abort. This is a standard nested transaction property.

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

3. *(Sequence)* $pr_i \rightarrow up_i$
   Transaction pr_i always executes before up_i. This is because up_i updates the backups with the results of pr_i execution.

4. *(Termination)* If R_i executes t_i, then all correct replicas of R eventually know the outcome (i.e. commit or abort) of t_i.

5. *(Non-triviality)* $ReadyToCommit_{pr_i} \wedge ReadyToCommit_{up_i} \Rightarrow$ outcome of t_i is commit.

The success of subtransactions pr_i and up_i is crucial for the success of t_i: in other words, t_i can only commit if the processing transaction pr_i and the update transaction up_i of the backup replicas have succeeded (Property 2). Note that this specifies a particular case of nested transactions, namely a distributed flat transaction [11]. We use the nested transaction model as it will be mandatory when we extend our specification to the invocation between R and S in Section 3.2.

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.

The termination property (Property 4) ensures that once transaction t_i is started, it eventually terminates by either commit or abort. Property 4 is also a liveness property, which ensures that the outcome of transaction t_i is eventually decided and that all subtransactions executing on correct processes eventually terminate. Clearly, if R_0 fails before committing t_0, then t_0 and its subtransactions will not be committed. Property 4 also ensures, that started transactions eventually terminate, even if the primary replica fails. This property is essential in preventing orphan subtransactions. The protocol we present later in this paper is designed to provide this property.

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

In our system model, we assume that crashed processes do not recover.[2] Consequently, the failure of a replica R_i 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 terms of transactions. 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 Figure 5). 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. Remember that server S is represented as a single, non-replicated server. For our discussion, it is not relevant whether S is replicated or not.

Figure 6 illustrates the hierarchy of transactions. The top-level transaction is t, which is started after reception of the client request (see Figure 5). It contains the subtransactions pr and up. Subtransaction pr, in turn, contains est. Subtransactions that are crucial to the commit of the top-level transaction are surrounded by dotted circles. In other words, if one of the crucial transactions aborts, transaction t aborts (see Property 2).

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

7. $Abort_{pr_i} \Rightarrow Abort_{est_i}$
   If subtransaction pr_i is aborted, est_i is aborted as well.

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

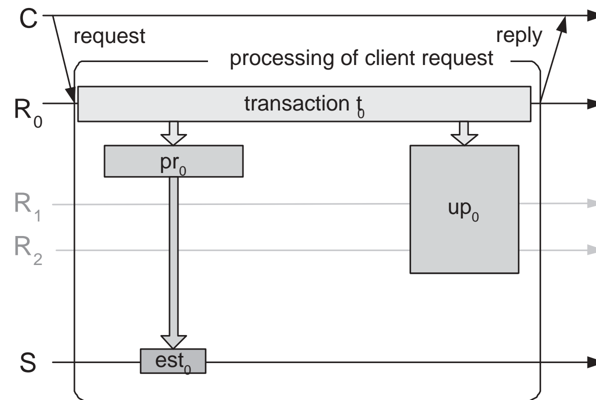


Figure 5: Representation of subtransactions (no failures).

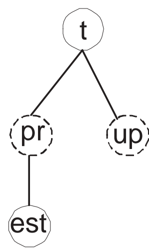


Figure 6: Model of the replicated invocation.

8. $ReadyToCommit_{est_i} \wedge Commit_{pr_i} \Rightarrow Commit_{est_i}$
   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.

Properties 7 and 8 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).

4 The Problem of Orphan Subtransactions with Replicated Invocation

In the previous section, we have specified replicated invocation in terms of a hierarchy of transactions. A standard solution to implement distributed transactions is to rely on a TP (transaction processing) monitor [11]. However TP monitors centralize the control and are prone to blocking. In particular, TP monitors usually rely on a two-phase commit protocol (2PC) to ensure atomicity. However, the 2PC protocol is blocking [1]. In contrast, we want a completely decentralized non-blocking solution (progress despite of failures).

According to the specification, 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. The failure of a replica R_i may interrupt the mechanism, that notifies the subtransactions of the commit or abort decision. Assume, for instance, that the primary fails before it has forwarded the commit or abort decision to S (Figure 7, left part). As a consequence, est_0 ignores the outcome of t_0 and thus cannot terminate. In this case, est_0 is called an *orphan subtransaction*. Clearly, orphan subtransactions are undesirable, because they maintain the 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.

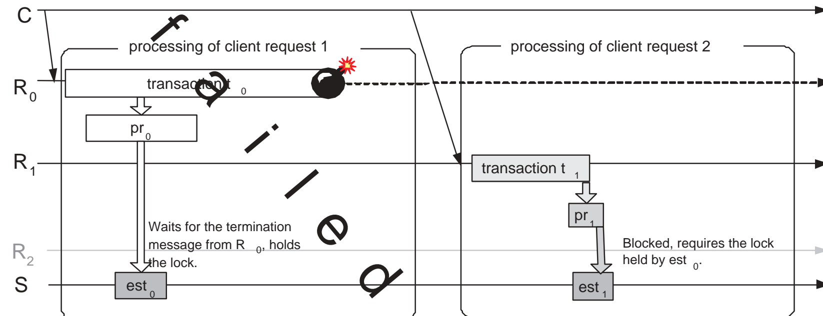


Figure 7: An orphan subtransaction est_0 on pessimistic server S.

Orphan subtransactions correspond to orphan requests presented in Section 2.2. To comply with the model, from now on orphan requests we will call orphan subtransactions. 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 solution is called *pessimistic processing*, and the server is called a *pessimistic server*. The first solution is called *optimistic processing*, and the server is called an *optimistic server*.

4.1.1 Blocking with Pessimistic Server S

Consider the case of an orphan subtransaction (Figure 7) 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.

4.1.2 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 semantically undoes the modifications caused by the original transaction. Assume, for instance, that transaction t reserves a ticket on a flight, then $compensate_t$ simply cancels this reservation.

Using an optimistic approach, blocking is prevented. Indeed, the locks held by subtransaction est_0 (Figure 8) 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, since the state of server S reflects est_0, but after the crash of R_0, est_0 is not valid any more.

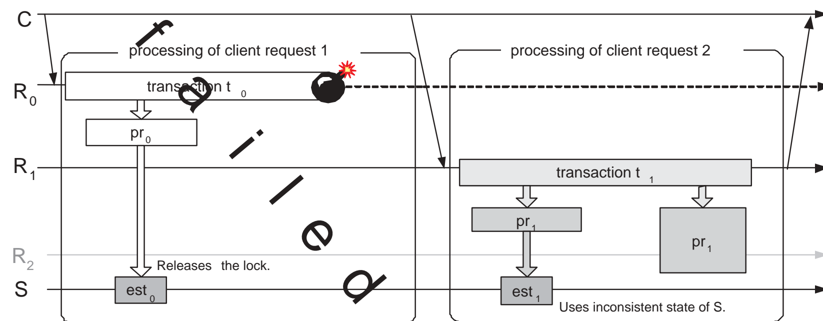


Figure 8: An orphan subtransaction est_0 on optimistic server S.

In the next section we present solutions for pessimistic and optimistic servers.

5 Replicated Invocation Protocol in the Case of Non-Deterministic Execution

This section first presents 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

5.1.1 The Problem of Finding out S

To prevent orphan subtransactions in replicated invocations, it is crucial that a new primary replica R_i is able to find out the identities of the servers that have been accessed by the previous primary R_j. This allows R_i to send abort message or compensating transactions to S. How can S be known to R_i? The identity of S is trivially known if (1) it can be deterministically computed by the replicas of R, or (2) if the set of servers is sufficiently small. In the latter case, a message is sent to all servers to find out the one that was invoked by R_j. Here we address the more complex cases in which the identity of S cannot be found a posteriori. This is especially the case if:

- the set of servers is large, and
- the identity of S is dynamically computed during the processing of R_i. 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_i to find out the identity of S computed by other replica R_j.

5.1.2 Sending Undo Information

We call *undo information*, the information that allows to undo a particular request; it includes the name of the server S to which the requests is sent, and the description of an action to perform. The solution to orphan requests/subtransactions consists to make 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 request* and *compensation requests*:

1. unterminated orphan subtransactions on pessimistic servers need to be terminated, and
2. terminated orphan subtransactions on optimistic servers need to be compensated.

In case (1), *termination requests* are `COMMIT` and `ABORT` messages[3]. In case (2), to restore the consistent state of the system, compensating actions are included in the undo information. Note however, that compensating the request of a replica is not easy. For example the sequence of requests $(rq_x; rq_y; compensate_rq_x)$ must be a valid sequence and must be semantically equivalent to the sequence that consists only of rq_y.

5.2 The Protocol

The *Replicated Invocation Protocol* for non-deterministic execution is presented in Figures 9, 10 and 11. Figure 9 first introduces two message types: *StandardRequest* and *UndoInformation*. Messages of the first type contain two fields: the request itself and the request id, which uniquely identify

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

the request in the system. Messages of the type *UndoInformation* contain four fields: compensating request, request identification, parent request id and target. The compensating request is used with optimistic server and contains the request, which semantically undoes the results of the original request. The request identification holds the id of the original request. It is used in the pessimistic server case. The other two fields are used for both pessimistic and optimistic types of servers. The parent id field holds the id of client's request to R, which triggered the request to S. The target field identifies the server S, to which the UndoInformation message has to be sent, if needed.

The protocol itself consists of five procedures executed on the primary of R. When the primary gets a request from client C, Procedure 1 (Fig. 10) is executed. If the request was not processed before, the primary starts a transaction, which in our model corresponds to transaction t in Figure 5. The transaction contains two nested subtransactions, which correspond to procedures *Process Request* (Procedure 2) and *Update Backups* (Procedure 3), presented later. After the client's request is processed and the backups are updated, the processing on distant pessimistic servers must be committed. Undo information, sent to backups during the processing, must be garbage collected. Procedure 1 terminates after sending the reply to the client.

Procedure 2 (Fig. 10) corresponds to the transaction pr in Figure 5. Assume that during processing, the primary needs to send a nested request to some other server S. Before doing so, a message of type UndoInformation is prepared for that request and multicast to the backups (this multicast is denoted by *Uniform-VScast* in Procedure 3) [4]. The content of the undo message depends on the type of server the original request is sent to.

Procedure 3 (Fig. 11) corresponds to the transaction up in Figure 5. The updates are multicast to the backups also using uniform VScast mentionned above.

Procedure 4 (Fig. 11) is called when a replica becomes a primary, which happens if the previous primary fails or is wrongly suspected to have failed. Before starting serving client's requests, the new primary must take care of orphan subtransactions. Managing orphan subtransactions in Procedure 5 (Fig. 11), depends on the type of server: pessimistic or optimistic. We describe each case separately.

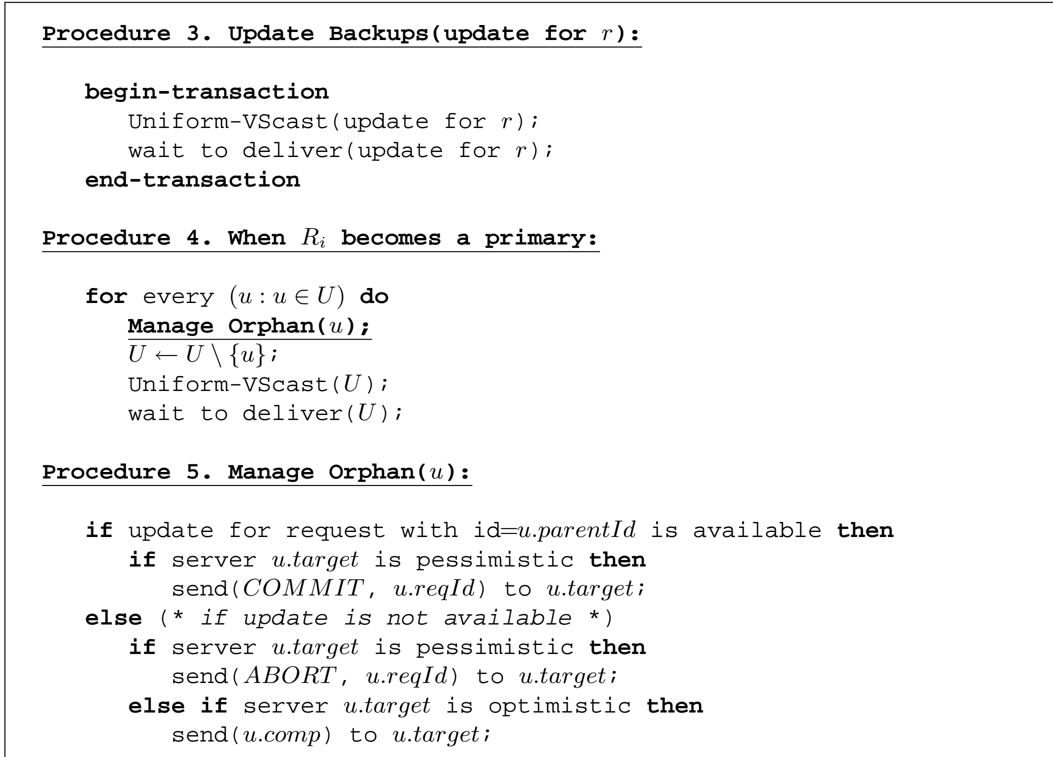
```
New Message TYPE StandardRequest = {req, id};
   req - a request to be sent;
   id - identification number, which uniquely specifies the request;

New Message TYPE UndoInformation = {comp, reqId, parentId, target};
   comp - compensating request;
   reqId - identification number of the request,
          to which this undo message relates;
   parentId - client request id, which processing
              triggered undo message;
   target - the server, to send this undo message to, if needed;
```

Figure 9: Message type declaration.

[4] In the context of group communication, this multicast corresponds to what is called *uniform view synchronous broadcast* [4, 24]. Roughly speaking, uniform view synchronous broadcast ensures that if some process delivers the message, then all correct processes eventually deliver the message. 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 really needed to understand the contribution of the paper.

```
r : StandardRequest;
u : UndoInformation;
U : set of UndoInformation messages;

Procedure 1. Upon reception of request r from C:

   if update for request with id=r.id is available then
      send(reply for r) to C;
   else
      begin-transaction
         Process Request(r);
         Update Backups(update for r);
         for every (u : u ∈ U and u.reqId = r.id) do
            if server u.target is pessimistic then
               send(COMMIT, u.reqId) to u.target;
            U ← U \ {u};
            Uniform-VScast(U);
            wait to deliver(U);
         send(reply for r) to C;
      end-transaction

Procedure 2. Process Request(r):

   begin-transaction
      ...
      if primary needs to send nested request to S then
         new s : StandardRequest;
         s.req ← request to S;
         s.id ← assign unique id;
         new u : UndoInformation;
         u.parentId ← r.id;
         u.target ← S;
         if server S is pessimistic then
            u.comp ← NULL;
            u.reqId ← s.id;
         else if server S is optimistic then
                  u.comp ← compensating request for s;
                  u.reqId ← NULL;
         U ← U ∪ {u};
         Uniform-VScast(U);
         wait to deliver(u);
         send(s) to S;
         wait for reply;
      ...
   end-transaction
```

Figure 10: Replicated invocation protocol (Part 1).



```
Procedure 3. Update Backups(update for r):

   begin-transaction
      Uniform-VScast(update for r);
      wait to deliver(update for r);
   end-transaction

Procedure 4. When R_i becomes a primary:

   for every (u : u ∈ U) do
      Manage Orphan(u);
      U ← U \ {u};
      Uniform-VScast(U);
      wait to deliver(U);

Procedure 5. Manage Orphan(u):

   if update for request with id=u.parentId is available then
      if server u.target is pessimistic then
         send(COMMIT, u.reqId) to u.target;
   else (* if update is not available *)
      if server u.target is pessimistic then
         send(ABORT, u.reqId) to u.target;
      else if server u.target is optimistic then
         send(u.comp) to u.target;
```

Figure 11: Replicated invocation protocol (Part 2).

5.2.1 Pessimistic Server S

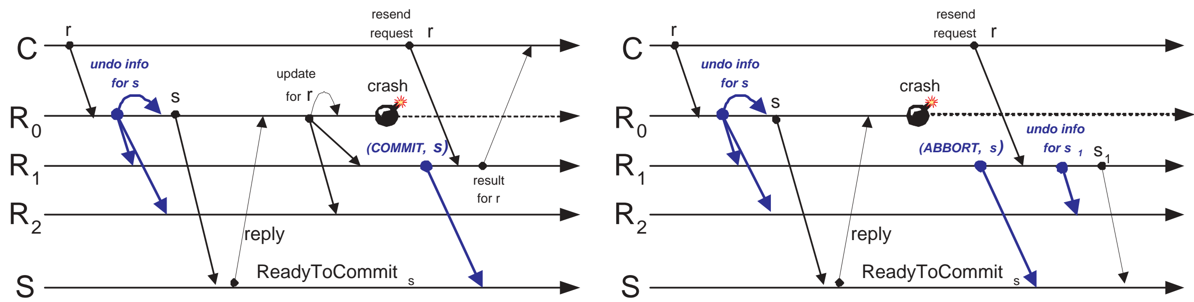
If server S executes pessimistically, a termination message is always required. Indeed, assume that the primary R_0 fails after updating the backups, but before sending the result to the client (Fig. 12 (a)). In this case, as the new primary R_1 has received the update, a `COMMIT` message is sent to S together with the id of the request to be committed. In contrary, an `ABORT` message is sent to S by R_1, if R_0 fails before it updates the backups (Fig. 12 (b)). When C resends its request, this request is executed by R_1.

Consider the particular case of R_0's failure after sending the undo information, but before sending the request to S. In the case of pessimistic server S, no special mechanisms are needed. Indeed, termination messages not related to an actual request are simply ignored by S.

5.2.2 Optimistic Server S

To undo the request s sent to optimistic server S, an compensating request is used. Consider first the case where no compensating request is required (Fig. 13 (a)). 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 is updated, when C resends its request, the new primary R_1 simply returns the result previously computed by R_0.

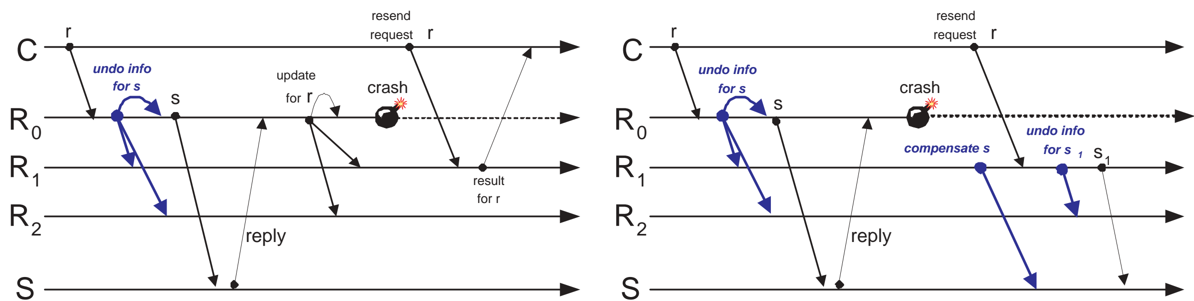
However, if R_0 fails before updating the backups (Fig. 13 (b)), the effects caused by s on S need to be undone. Thus a compensating request is sent to server S. Eventually, C resends its request to the new primary R_1, which recomputes the result. Note that the order of compensating an original



(a) The primary fails, after backups are updated.

(b) The primary fails, before backups are updated.

Figure 12: Primary's R_0 failure, after invoking pessimistic server S.



(a) The primary fails, after backups are updated.

(b) The primary fails, before backups are updated.

Figure 13: Primary's R_0 failure, after invoking optimistic server S.

requests is not significant: the compensating request can be sent at any time after the new primary is elected. This is a consequence of the properties of the compensating request (see Section 5.1.2).

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), the new primary will use this undo information to send a compensating request to S (see Procedure 5). Assume that u arrives at S before s. S must handle this case: if s has not been received, then u is not executed but stored to be reused in case s eventually arrives (if it does at all). Note that such early undo messages are possible even if R_0 fails after sending s.

6 Related Work

Most of the work performed in the context of replicated invocation assumes deterministic execution. For example, Mazouni's work [16, 17, 15] 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.

In contrast, Narasimhan, instead of assuming determinism, enforces determinism (in the context of multithreaded applications). The work was performed in the context of Eternal [20], a replication infrastructure for CORBA objects. The work introduces the notion of *MT-domain* (MT stands for multithreaded), to refer to any CORBA client or server that supports multiple (application level or ORB level) threads, which may access shared data. MT-domain contains one or more CORBA objects. The Eternal system enforces deterministic behavior within the MT-domain by allowing only *a single logical thread of control*, at any point in time, within each replica of the MT-domain. Each replica maintains a consistent queue of messages containing operations and responses destined for the MT-domain. Based on this incoming sequence of messages, the scheduler at each replica decides on the immediate or delayed delivery of the messages to that replica. These decisions are consistent and deterministic for every replica.

Zhao, Moser and Melliar-Smith [26] unify fault-tolerant CORBA (FT-CORBA) and CORBA Object Transaction Service (OTS) in the context of a three-tier architecture. Their work also assumes deterministic execution. The proposed infrastructure replicates transactional application servers to protect business logic tier from failures. The middle-tier is augmented with an automatic transaction retry mechanism, which in the case of failure prevents client from reissuing the request (this prevents duplicate invocations from the client-tier). Replicated out-bound gateways are introduced between the middle-tier and the data-tier: they are responsible for filtering duplicate invocations and manage transaction retry. If failure occurs and an ongoing transaction is *not* $ReadyToCommit$, the infrastructure, transparently to the client, aborts and retries the transaction. Abort is achieved by resetting the state of all objects involved in the transaction to the last checkpoint done by the logging mechanism. All logged messages, up to, but not including the one that started the transaction, are replayed. The ones within the aborted transaction are discarded. Finally the message that initiated the transaction is replayed at the transaction's initiator (the transaction is retried). If fault occurs when the transaction is $ReadyToCommit$, the infrastructure reissue the logged request of the second phase of 2PC protocol.

Frølund and Guerraoui [7] present a correctness criterion for exactly-once in the context of replication, that addresses also non-determinism in the execution, and external side-effects. They also propose a replication protocol, called *asynchronous replication*. The protocol is targeted towards the

classical three-tier architecture, with slim client, stateless application servers, and databases. In contrary, our approach is more general in that it also addresses statefull components. Indeed, 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 statefull components clearly leads to stronger requirements, e.g., the update of all replicas.

7 Conclusion

In the paper we have presented the problem of orphan invocations. We have shown that the problem, which is easily addressed with deterministic replicated servers, remains in the context of non-deterministic replicated servers. The protocol for preventing orphan invocations is based on undo information, sent by a server R_i to its replicas before issuing the nested invocation to S. Our protocol handles both pessimistic and optimistic handling of the invocation on S.

The approach presented in this paper has two limitations. This limitations might however be 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 5.2.1). 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.

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 [22], relaxed schemes of our approach may yield better performance in particular application contexts.

Acknowledgments. We would like to thank Matthias Wiesmann for his comments on an earlier version of this paper.

References

[1] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, Massachusetts, USA, 1987.

[2] K.P. Birman and T.A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1995.

[3] N. Budhirja, K. Marzullo, F.B. Schneider, and S. Toueg. The primary-backup approach. In Mullender [19], pages 199–216.

[4] G.V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 4(33):1–43, December 2001.

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

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

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

[8] S. Frølund and R. Guerraoui. X-ability: a theory of replication. *Distributed Computing*, 14(4):231–249, December 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 *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] C. Loosley and F. Douglas. *High-Performance Client/Server*. Wiley Computer Publishing, New York, USA, 1998.

[15] K. R. Mazouni. *Étude de l'invocation entre objets dupliqués dans un système réparti tolérant aux fautes*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, January 1996. In French.

[16] K. R. Mazouni, B. Garbinato, and R. Guerraoui. Invocation support for replicated objects. Technical Report 95/120, École Polytechnique Fédérale de Lausanne, Switzerland, May 1995.

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

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

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

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

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

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

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

[24] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *IEEE 13th Intl. Conf. Distributed Computing Systems*, pages 561–568, May 1993.

[25] F.B. Schneider. Replication management using the state-machine approach. In Mullender [19], pages 169–198.

[26] W. Zhao, L.E. Moser, and P.M. 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.