Fault-tolerance for Stateful Application Servers in the Presence of Advanced Transactions Patterns

Huaigu Wu Bettina Kemme
School of Computer Science, McGill University, Montreal, Quebec, Canada, H3A 2A7
hwu19, kemme@cs.mcgill.ca

Abstract

Replication is widely used in application server products to tolerate faults. An important challenge is to correctly coordinate replication and transaction execution for stateful application servers. Many current solutions assume that a single client request generates exactly one transaction at the server. However, it is quite common that several client requests are encapsulated within one server transaction or that a single client request can initiate several server transactions. In this paper, we propose a replication tool that is able to handle these variations in request/transaction association. We have integrated our approach into the J2EE application server JBoss. Our evaluation using the ECPerf benchmark shows a low overhead of the approach.

1 Introduction

Application servers (AS) have become a prevalent building block in current information systems. Clients send requests to an AS which accesses database systems to manage persistent data. The AS runs the application programs and maintains volatile data, such as session information, i.e., the server is *stateful*. Requests are executed in the context of transactions which provide durability for the persistent data, isolation from concurrent transactions, and atomicity. In the simplest execution model, each client request executes within its own individual transaction. In practice, however, execution can be more complex. For instance, the client can start a transaction, and then submit several requests in the context of this transaction before committing it. This is, e.g., often used when a web server (WS) is positioned between the real (internet) client and the AS. At the other extreme, one client request might create several independent transactions in the AS. Application programmers often chop the execution of a request into a set of small transactions to avoid lock contention at the database.

AS servers are often replicated to achieve 7/24 availability. If one replica crashes, the work assigned to this replica can failover to another replica. The challenge is to correctly handle requests and transactions that are active at the time of the crash. The AS replication solutions we are aware of only consider the simple case where one request is associated with exactly one transaction [15, 16, 14, 13, 28, 4, 3, 27]. In contrast, we propose a tool that is able to handle different execution patterns as described above. The system should provide *exactly-once* execution and *state consistency* even in the case of crashes [15, 27]. Assuming the 1-request/1-transaction pattern, exactly-once means that for each submitted client request, the server executes the corresponding transaction exactly once. State consistency guarantees that the state at AS replicas and database is always consistent. We refine these correctness properties to be able to capture advanced execution patterns.

Our tool is based on an existing protocol [27] which assumes the simple 1-request/1-transaction pattern. It uses a classical primary/backup approach [18, 21, 15, 13, 2]. One server replica is the primary executing client requests. It propagates state changes to the backup replicas whenever a transaction commits. If the primary fails, a backup replica fails over, reconstructs the state of the old primary, and continues the client connections. Requests that were active at the time the primary crashed (and only those) are automatically restarted at the new primary. This paper extends the basic tool to support advanced execution patterns.

Our goal is to provide a practical solution with little overhead. Hence, we have developed our replication tool within the context of a concrete AS architecture, namely J2EE [26] and integrated it into the open-source AS JBoss [17]. We believe, however, that the principle ideas can be applied to other kinds of application servers (e.g., CORBA, .NET), and hence, keep the algorithmic description as general as possible. Our performance analysis shows that the approach compares favorably with other fault-tolerant solutions.

2 Background

AS architecture We assume the application logic to be programmed within components (*Enterprise JavaBeans*

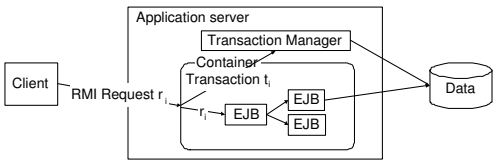


Figure 1. J2EE architecture

(EJB) in J2EE). Components can have state. We assume that a component is only associated with a single client (e.g., stateful session beans (SFSB) in J2EE), or that there is a concurrency control mechanism in place that allows at most one transaction to be active on the component (e.g., entity beans (EB) in J2EE). We assume all components to run within the same runtime environment (called container in J2EE). Additionally, the server provides a set of services like transactions and security. Figure 1 shows an J2EE architecture with a transaction service, and three EJBs. The client makes a request to a method of an EJB which in turn can call other EJBs and/or the database before returning a response to the client.

Most AS architectures provide two ways to access services. Either the components explicitly make service calls or such service calls are made automatically whenever a method of a component is activated. The transaction service is implemented by the transaction manager (TM). A transaction consists of a set of operations. If a transaction commits, all its operations succeed. The state changes in the database are persistent while changes on AS components usually remain volatile. If the transaction aborts, any changes performed so far on the database are undone by the database system. Whether the changed state on AS components is undone depends on the AS architecture. In J2EE, changes on SFSBs are not automatically undone. However, programmers can provide rollback methods for SFSBs which are automatically called by the J2EE server in the abort case. We say the AS server provides *full state consistency* if mechanisms exist to abort changes on components, otherwise it provides *relaxed state-consistency*. If a transaction accesses more than one database, a 2-phase commit protocol (2PC) is necessary at commit time for atomicity. The TM first sends a *prepare* request to all participating databases which return either with a *prepared* message or their decision to abort. If all databases have successfully prepared, the TM sends a *commit* decision to all databases, otherwise (at least one aborted) an *abort* confirmation. The databases terminate the transaction accordingly.

Communication between AS replicas is via a *group communication system* (GCS) [7]. A group member can multicast a message to all members (including itself). Sending a message with *reliable delivery* guarantees that when a member receives a message and does not fail for sufficiently long time then all members receive the message unless they fail. *Uniform-reliable delivery* is stronger: if any member receives a message (even if it fails immediately afterwards) all members receive the message unless they fail. Our algorithms require all messages of the same sender to be received in sending order (FIFO). The GCS automatically removes crashed members from the view of currently connected members and also provides explicit join and leave primitives. Upon a view change all members receive a view change message providing the *virtual synchrony* property: if members p and q receive both first view V and then V', they receive the same set of messages while members of V. In an asynchronous environment, the GCS might wrongly exclude a non-crashed member. In this case, we require the affected replica to shut down. An alternative would have been to apply semi-passive mechanisms [10].

3 Model

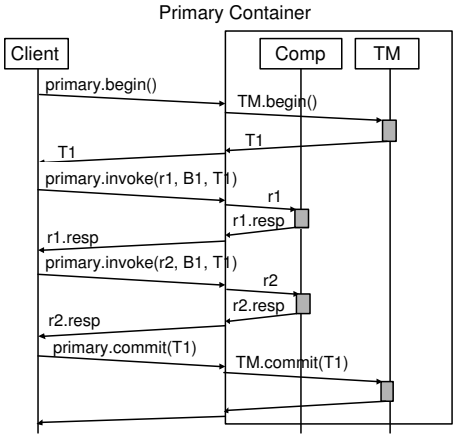
We make the following assumptions. All communication is asynchronous and reliable (no network partitions). Individual components within a server do not fail but an AS only fails entirely by crashing (no byzantine behavior). For now, databases and clients do not crash. We discuss their crashes in Section 5. Clients and components are single-threaded that block when waiting for the response of a request but execution does not need to be deterministic. For space reasons, we only consider full state consistency. Our solutions for relaxed state consistency handle transaction abort differently but use similar reasoning.

In a non-replicated system without crash, we assume each request to execute successfully, and the server to provide a *correct response*. This could be an abort exception if the transaction fails, e.g., due to application semantics.

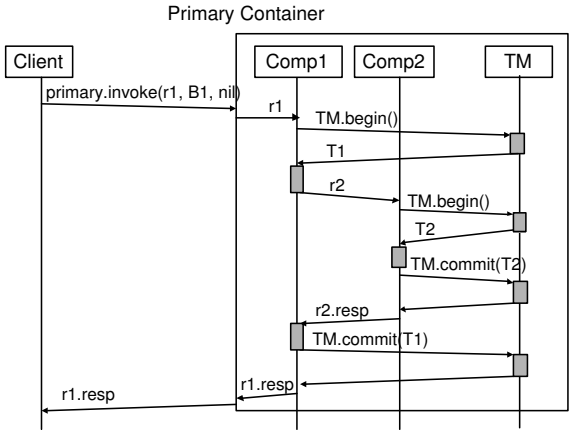
If a non-replicated server crashes before returning a correct response to a request, the client receives a *failure exception*, and the request is executed at-most-once. All state at the AS and the database connections are lost. We assume the standard database behavior in this case: the database aborts all active transactions except those in the *prepared* state (in case of 2PC) which will remain active until the database receives the *commit/abort* decision.

Execution Patterns We classify execution patterns by the number of client requests involved in a transaction and the number of transactions generated by a request. In the *1-1 pattern* (1-request/1-transaction), each client request initiates a single transaction T, and the entire request execution on the AS and the database is performed within T.

A first extension allows several client requests to run within a single transaction (see Fig 2(a)) leading to a *N-1 pattern* (N-requests/1-transaction). A client first sends a *begin* transaction request, then several requests for components, and finally a *commit/abort* request. The client receives responses before the corresponding transaction terminates. This pattern requires the AS to export the be-



(a) N-1: N-requests/1-transaction



(b) 1-N: 1-request/N-transactions

Figure 2. Execution Patterns

gin/commit/abort methods of the TM. It is often used when a web-server (WS) runs between the real client and the AS. In this case, the real client makes a request to a component in the WS (e.g., a servlet) which is in turn the client for the AS and makes calls to the AS. Controlling transactions from outside the AS has also become important for web-services.

The second extension allows a client request to generate more than one transactions leading to a *1-N pattern* (1-request/N-transactions). We explain the semantics using the example of Fig 2(b). A client request to a component initiates a transaction T_1. When the component creates a sub-request to another component during the execution, T_1 is suspended and a new transaction T_2 is started[1]. The sub-request is executed within the context of T_2. When T_2 commits, T_1 resumes, and further execution again happens within the context of T_1. T_2 might commit while T_1 aborts, or vice versa. However, T_2 always terminates before T_1. We call T_1 the *outer* and T_2 the *inner* transaction. An outer transaction might have several inner transactions, and an inner transaction might be the outer transaction of another inner transaction. The 1-N pattern is widely used in practice when a long execution needs to be chopped into small transactions in order to increase concurrency within the database [24, 20]. The programmer typically provides compensating transactions for each transaction type in order to guarantee that if not all transactions commit the effects of already committed transactions are undone.

Finally, the two extensions can be combined to a *N-N pattern*. Several client requests might execute within transaction T_1 while T_1 initiates several inner transactions. Furthermore, the patterns above can be refined by considering that each transaction might access only one database or several databases. Only in the latter case 2PC is needed.

Correctness For space reasons we are not able to provide a formal correctness definition or formally prove the correctness of the algorithms. Instead, we resort to a more informal reasoning. We state correctness as a set of properties that have to be provided despite crashes of individual AS replicas. From the perspective of the client, correctness means *exactly-once request execution*, that is, the client receives exactly one correct response for each submitted request, and no failure exception. From the perspective of the server we require *exactly-once transaction execution* and *full state consistency*. The first requires that each transaction commits at most once. If it does not commit, it aborts due to application semantics or database exceptions (e.g., deadlock). The second requires that if a transaction T commits, the database has committed T and all non-crashed AS replicas have the state changes performed by T. If T aborts, the database has aborted T, and none of the AS replicas has the state changes performed by T. Additionally, in order to synchronize client and server perceived correctness we require what we call *request / transaction matching*. This means that each committed transaction must be result of the successful execution of a request or sequence of requests, namely the one that produces the correct response seen by the client. This requirement is motivated by the fact that the execution of a request might be replayed if its first execution was interrupted by a crash. However, if the failed execution led to the commit of a transaction and the second, successful execution follows a different execution path that would not include this very same transaction, then we would have a *ghost* transaction that does not refer to any execution that the client perceives as successful.

4 Replication Algorithms

Our replication tool uses a primary/backup approach. It consists of a client algorithm (which is downloaded to the client when it connects to the AS) and a server part. We assume the replication tool obtains control before a request

[1] In J2EE, the application programmer can specify at deployment time that a call to a method should always generate a new transaction.

is sent to the TM or a component, and after the call returns.

We first provide an overview of the algorithm for the 1-1 pattern, and then discuss the N-1, 1-N and N-N algorithms when only one database is accessed. For space and readability reasons we only present the solution for N-1 in algorithmic form. We also shortly discuss how the algorithms can be adjusted to work with more than one database.

4.1 1-1 Algorithm Overview

Our 1-1 algorithm is from [27] which in turn had combined ideas from [14, 13]. The client replication algorithm intercepts each request submitted from the client to the server, attaches a unique id, and forwards the request to the current primary. Upon a failure exception, it resends the request with the same id to the new primary. This repeats until it receives a correct response.

The primary executes a client request within a transaction T. At commit time, it propagates all changes performed by T on components together with the request/response pair in a single *committing* message (uniform-reliable delivery), and enters the identifier $txid$ of T into the database as part of the transaction. When it receives its own committing message, it commits T, returns the response to the user, and multicasts a *committed* message (reliable delivery). If T aborts at some time during the execution, an *aborted* message is multicast (reliable delivery) containing the request/abort response pair.

When the primary crashes, a backup becomes the new primary and performs failover. For a given request with associated transaction T, the new primary might have received before the crash (1) not yet any message, (2) the committing message but no decision message, or the (3) commit/aborted messages. In the first case, our failure assumptions guarantees that the database aborted T. In the second case, the database might have committed or aborted T. The new primary checks whether T's txid was inserted in the database. If yes, the database had committed T, and the new primary applies the component changes included in the committing message, and stores the request/response pair. Otherwise it ignores the committing message. In case (3), if the decision was commit, the new primary applies the component changes of the committing message, if the decision was abort, it ignores them. In any case, it stores the request/response pair. After failover, when the new primary receives a request from a client it checks first whether it has recorded a corresponding request/response pair, and if yes, returns immediately the response. Otherwise it executes the request according to the primary algorithm.

Resubmitting requests in failure cases but avoiding reexecution if the request was already successfully executed provides exactly-once execution and proper request/transaction matching. Inserting the txid into the database allows the new primary to check wether the database transaction committed, and apply or disregard the AS state accordingly, hence providing state consistency.

4.2 N-1 Algorithm

In the N-1 model the client can include several requests within a single transaction. If the primary now crashes before a transaction T commits, the database aborts T, but the client might have already received the responses for some of the requests belonging to T. These responses now refer to a transaction that aborted at the database due to a crash. We have implemented two approaches to address this problem. The *N-1-best-effort* algorithm is fast but only provides at-most-once execution in some cases. The *N-1-ordered* alternative achieves better transparency at the price of higher overhead during normal processing.

N-1-best-effort In the first approach, the main adjustments are at the client side. The client replication algorithm keeps all requests and corresponding responses for each transaction. If the primary crashes while a transaction was active, the client algorithm replays the execution at the new primary. If it leads to the same results as the original execution, it was successful and failover is completely transparent. If it leads to different results, the replay was unsuccessful and the reexecuted transaction is aborted. The real client, having seen the old non-repeatable responses, is informed with a failure exception, and hence transparency is lost.

Figures 3, 4, 5 show the N-1-best effort algorithm. *Request* has an identifier rid while a *Response* captures the response to a request. A single *CEU* object ceu at the client replication algorithm keeps track of the execution within the current transaction. It contains the transaction identifier $txid$ of type TID, and all requests executed so far together with their responses (RR). The server maintains an *EU* object for each currently active transaction (one per client). *EU* keeps track of transaction identifier $txid$ and the set of components $COMP$ that have been accessed so far. In contrast to the 1-1 algorithm, the servers do not need to keep track of request/response pairs. The $content$ of a *Message* object depends on the type of message.

The client replication algorithm (Figure 3) intercepts begin, invoke and commit requests. We ignore abort requests for space reasons. For simplicity, the algorithmic description assumes that the client submits requests in the correct order (begin/invoke/invoke.../commit). An abort induced by application semantics or by the database (deadlock, etc.) is considered a correct response. In this case, we expect the client to submit a new begin transaction as next request.

Upon intercepting the begin request (Fig. 3(a)). the ceu object is initialized and the request is forwarded to the current primary until it is successfully executed. Upon an invoke request (Fig. 3(b)), the response from the primary is

```
void begin ()
 1. while (true)
 2.   ceu.initialize();
 3.   ceu.txid = primary.begin();
 4.   if (∄ failure Exception) return;
 5.   else find a new primary;
```
(a) transaction begin

```
Response invoke (Request req, Component comp)
 1. Generate req.rid;
 2. while (true)
 3.   Response resp = primary.invoke(req, comp, ceu.txid);
 4.   if (∃ abort Exception) throw abort Exception;
 5.   if (∄ failure Exception)
 6.     ceu.RR ∪ = {(req, comp, resp)};
 7.     return resp;
 8.   else
 9.     while (∃ failure Exception)
10.       find a new primary;
11.       replay(ceu);
12.       if (∃ replay failure)
13.         ceu.initialize();
14.         throw replay failure;
```
(b) regular request

```
void commit ()
 1. while (true)
 2.   primary.commit(ceu.txid);
 3.   if (∄ failure Exception)
 4.     ceu.initialize();
 5.     if (∃ abort Exception) throw abort Exception;
 6.     else return;
 7.   else
 8.     while (∃ failure Exception)
 9.       find a new primary;
10.       if (primary.is_committed(ceu.txid))
11.         ceu.initialize();
12.         return;
13.       else
14.         replay(ceu);
15.         if (∃ replay failure)
16.           ceu.initialize();
17.           throw replay failure;
```
(c) transaction commit

```
void replay (CEU ceu)
 1. ceu.txid = primary.begin();
 2. if (∃ failure Exception) throw failure Exception
 3. else
 4.   for each (oreq, ocomp, oresp) ∈ ceu.RR
 5.     Response nresp = primary.invoke (oreq, ocomp, ceu.txid);
 6.     if (∃ failure exception) throw failure Exception
 7.     else if (∃ abort exception) throw replay failure
 8.     else if (nresp != oresp)
 9.       primary.abort(ceu.txid);
10.       throw replay failure;
```
(d) replay

Figure 3. N-1-best-effort at the client side

captured (lines 3-7), or, if the primary crashes a replay is initiated at the new primary (lines 9-14). Upon a commit request (Fig. 3(c)), if no crash happens, the commit returns to the user. If the transaction aborts because of database semantics, the response is an abort exception (lines 3-6). If a crash occurred before the server returns from the commit,

```
TID begin ()
 1. new EU eu;
 2. eu.txid = TM.begin_Transaction();
 3. return eu.txid;
```
(a) transaction begin

```
Response invoke (Request req, Component comp, TID txid)
 1. find eu corresponding to txid;
 2. eu.COMP ∪ = {comp};
 3. return comp.invoke(req);
```
(b) regular request at primary

```
void commit (TID txid)
 1. find eu corresponding to txid;
 2. for each comp ∈ eu.COMP
 3.   set comp.state to current state of corresp. component;
 4. new committing Message m1;
 5. m1.content = {eu};
 6. multicast m1 by uniform reliable delivery;
 7. insert eu.txid into database;
 8. wait until receive m1;
 9. TM.commit_Tranasction(txid);
10. if (∄ abort Exception)
11.     new committed Message m2;
12.     m2.content = {eu.txid};
13.     multicast m2 by reliable delivery;
14. else
15.     new aborted Message m3;
16.     m3.content = {eu.txid};
17.     multicast m3 by reliable delivery;
18.     throw abort Exception;
```
(c) transaction commit

```
Bool is_committed (TID txid)
 1. if txid can be found in database return true
 2. else return false;
```
(d) check outcome of transaction

Figure 4. N-1-best-effort at primary

the transaction might have committed before the crash or it aborted upon the crash. The client algorithm checks at the new primary (lines 9-10). We will see later how the new primary answers such request. If the transaction committed, the commit request returns successfully (lines 11-12). Otherwise, the transaction is replayed at the new primary (lines 13-17). The replay (Fig. 3(d)) starts a new transaction and resubmits each request of the old execution (lines 1-5). If one of these requests receives a different response than the original execution, the reexecuted transaction is aborted throwing a replay failure exception to the client (lines 7-10). It is now up to the client to act upon this. Otherwise, reexecution has been successful and the algorithm continues with the request that was active at the time of the crash. Note that after the reexecution the state of the new primary (or the database) might not be exactly the same as the state of the old primary after the first execution, but this does not really matter because only responses but not server state is visible to the client. Throughout the algorithm additional AS crashes reset the algorithm to the appropriate place.

The server (Figure 4) creates an eu object upon transaction begin (Fig. 4(a)), and keeps track of each compo-

```
void failover ()
 1. new Eu eu, new set COMP;
 2. in order of reception process each committing message m
 3.   eu = m.content
 4.   if (∃ aborted message m' with m'.content == eu.txid)
 5.    or (∄ committed message m' with m'.content == eu.txid
 6.        and eu.txid does not exist in database)
 7.          ignore committing message
 8.   else // transaction committed
 9.     for each comp ∈ eu.COMP
10.       if (∃ c ∈ COMP && c == comp)
11.          c.state = comp.state
12.       else COMP ∪ = comp;
13. for each comp ∈ COMP
14.    create corresponding component;
15.    set component's state to comp.state;
```

Figure 5. N-1-best-effort failover at backup

nent accessed by a request (Fig. 4(b)). At commit time (Fig. 4(c)), we send the committing message including the final state for each accessed component, and insert the txid into the database (lines 1-8). Then we commit the transaction. If commit was successful, we send a commit message (lines 9-13) and the commit completes. If commit is not successful due to database semantics, the primary informs the backups about the abort so that they can discard the committing message, and returns the exception to the client (lines 15-18). When asked by the client replication algorithm, the *is_committed* routine (Fig. 4(d)), checks in the database for the txid and returns the answer.

The backup, during normal processing, stores all received messages in a FIFO queue. Figure 5 shows the failover. Committing messages are processed in FIFO order to track the latest state of each component (lines 2-3). If the corresponding transaction committed, we determine which components were affected (lines 8-12). Otherwise the committing message is ignored (lines 4-7). Finally, all necessary components are recreated (13-15). Alternatively, component recreation can also happen *lazily* after failover during normal processing of the new primary: whenever a client makes a request to a component we first look whether it is in COMP and if yes, remove it from COMP, initialize it, and adjust the component state before performing the corresponding operations.

Reasoning about correctness We have to look at the timepoints at which the primary can crash. (1) If it crashes when the client attempts to start a transaction, the database transaction (if already started) aborts, and the new primary has not yet done anything. The client algorithm simply restarts the transaction. (2) If it crashes sometime during the execution before the client submits the commit request, the database transaction aborts and the new primary does not know anything about the transaction. The client algorithm replays the execution at the new primary. If it is successful, execution continues with the last request submitted. In this case the client perceives exactly-once execution for its requests, the server executes the corresponding transaction exactly once, the state between database and new primary is consistent, and client and server execution matches. If the replay is not successful, the transaction aborts. The client is informed and exactly-once is not provided. However, state consistency is provided, and since neither the client nor the server perceives a successful outcome, execution matches. (3) If the primary crashes after the client had submitted the commit request but before receiving a response, the cases are the same as those in the 1-1 algorithm: the new primary might not yet have received any message, might have received the committing but not the committed message, or might have received the commit/abort message. The failover mechanism guarantees the state consistency between database and AS server by applying the state changes of the committing message if and only if the database transaction has committed. The client checks at the new primary whether the transaction had committed. If it committed, no reexecution takes place. This provides exactly-once execution at both client and server and execution matching. If it did not commit, replay is initiated as in situation (2) above. In total, this protocol provides state consistency, request/transaction matching, and at-most-once execution. AS failures are transparent in some but not all cases. Note that if the transaction aborts due to application semantics but the primary crashes before returning the response, the client replication algorithm will actually replay the transaction. This does not violate any of the properties.

Increasing the chances for exactly-once Reexecution might not succeed if non-determinism occurs which can happen because of database access. For example, assume before the primary crash, T_1 reads and updates x, and returns a response to the client. Then the primary crashes before T_1 commits. At the new primary assume a transaction T_2 reads and updates x before T_1 resubmit its request. Hence, T_1's replay reads a different value of x than during the original execution. This might lead to a different response if the value of x affects the response. To avoid such behavior, we propose an alternative algorithm *N-1-ordered* that works for database systems that guarantee serializability through strict 2-phase locking. With N-1-ordered, the reexecution of all database access is performed in the same order as during the original execution. During normal processing, each database access is assigned a unique increasing identifier. Before the response for the request is returned, a message with the identifiers of all access triggered by the request is multicast to the backups. At the time of resubmission after the primary crashes, each replayed database access must be executed according to its original order and new requests may not start until all resubmissions have completed. In the example above, when T_2's request is submitted before T_1 resubmits its request, it

```
Result interceptDatabaseAccess (Request client_req, SQL sql)
 1. while (RDBAT.size > 0 and client_req.rid != RDBAT.First().rid)
 2.   client_req waits until timeout or notification;
 3.   if (timeout)
 4.     TOA = RDBAT;
 5.     RDBAT.empty();
 6.     notify each waiting client_req;
 7. Result r = execute sql in the database;
 8. if (req.rid == RDBAT.First().rid)
 9.   RDBAT.removeFirst();
10.   notify each waiting client_req;
11. else
12.   if (∄ toa ∈ TOA with toa.rid = req.rid) // original execution
13.     new DBA dba;
14.     dba.rid = req.rid;
15.     dba.order = counter++;
16.     DBAT ∪ = {dba};
17. return r;
```
(a) database access

```
Response invoke (Request req, Component comp, TID txid)
 1. find eu corresponding to txid;
 2. eu.COMP ∪ = {comp};
 3. Response resp = comp.invoke(req)
 4. if (req is a client request)
 5.   new set cds;
 6.   for each dba ∈ DBAT with dba.rid == req.rid;
 7.     cds ∪ = {dba}; DBAT \ = {dba}
 8.   if (cds != ∅)
 9.     new ordering Message m;
10.     m.content = {(txid, cds)};
11.     multicast m by reliable delivery;
12. return eu.resp;
```
(b) regular request leading to multicast ordering message

```
void failover ()
 1. ...
 2. new list RDBAT;
 3. in order of reception process each ordering message m
 4.   if ∃ committed message m' with m'.content = m.content.txid
 5.    or m.content.txid exists in database
 6.     discard m;
 7.   else
 8.     for each dba ∈ m.content.cds
 9.       RDBAT.add(dba);
10. sort RDBAT ascendingly according to dba.order and set counter
    accordingly;
```
(c) handling ordering message at failover

Figure 6. N-1-ordered extensions

has to wait until T_1's request is reexecuted to guarantee that T_1 again reads the same data as in the original execution.

Figure 6 shows how the N-1-ordered algorithm extends the N-1-best-effort algorithm. During normal processing, the N-1-ordered algorithm intercepts all accesses to the database (Fig. 6(a)). We assume the algorithm knows from which client request each database access is triggered. For the original execution (no resubmission) a *DBA* object records the pair of the rid of the client request and a unique increasing number which represents the order of the access. A $DBAT$ set captures all DBA objects. Before the response for a client request is returned, an $ordering$ message is multicast to all backups containing all corresponding DBA objects (Fig. 6(b)). Requests that do not access the database do not require a multicast message. At failover, the new primary will handle all the ordering messages received before the crash of the old primary (Fig. 6(c)). We discard each ordering message for which the corresponding transaction has committed, since client requests involved in this transaction will not be resubmitted (lines 3-6). Otherwise, the DBA objects contained in the ordering message will be recorded in a list $RDBAT$ and sorted in ascending database access order (lines 8-10). When a client request is replayed at the new primary, each resulting database access will be intercepted (Fig. 6(a)). If the order of the replayed database access is not the smallest as recorded in the $RDBAT$, it has to wait until all database accesses with smaller order have been executed (lines 1-2). A new request (not resubmitted) has to wait until $RDBAT$ is empty. Only when these conditions are fulfilled, the database access can be executed (line 7). In order to handle clients that do not replay (e.g., they crashed by themselves), there is a timeout (lines 3-6) of how long a request is blocked. When the first timeout is triggered, all waiting requests will be notified, and all further request will not be blocked by the ordering mechanism to guarantee termination. For instance, in above example, if T_1 does not resubmit its request within a certain time, T_2's request will execute. In case of timeout, the $RDBAT$ will be emptied, but the remaining DBA objects in the $RDBAT$ will be temporarily stored in a list TOA to avoid reassigning access numbers to replayed database accesses that had a timeout (lines 4 and 12).

4.3 1-N Algorithm

Our 1-N algorithm extends the 1-1 algorithm in order to handle outer and inner transactions and the relationship between them. For that, we use a special request identification system. A client request has the request id given by the client replication algorithm. Now assume a transaction T was initiated by a request with request id rid. Then, the i'th request made within T that starts an inner transaction receives as request id $rid.i$, i.e., a concatenation of the request id of the outer transaction T and a counter that keeps track of how many inner transactions T started.

The algorithm handles inner transactions in the same way as outer transactions. At commit time of any inner or outer transaction, the request/response pair and the changed components are multicast as in the 1-1 algorithm. Since we use FIFO multicast and inner transactions always terminate before their corresponding outer transaction, the messages of an inner transaction always arrive at the backups before the messages of the corresponding outer transaction. Assume transaction T_1 triggered by client request with request id 1 submits a request with request id 1.1 that starts inner

transaction T_2. When the primary crashes, there are three main cases. (1) Both T_1 and T_2 were still active, (2) both had terminated (commit or application induced abort), or (3) T_2 had terminated while T_1 was active. The new primary is able to distinguish these three cases by examining the messages it has received for T_1 and T_2 during normal processing and checking for transaction identifiers in the database if necessary. In the first case, it will not apply any component changes even if it had received the committing message. The client will resubmit request 1 and the new primary reexecutes starting with T_1. All correctness properties are provided. In the second case, the new primary has installed the component changes of both transactions. If the client replication algorithm resubmits the request 1, T_1's response is immediately returned without reexecution.

Case (3) is more difficult. The client resubmits request 1, triggering the start of a new transaction T_1' at the new primary. If the execution is deterministic, T_1' will submit the very same request 1.1 that initiated T_2 on the old primary. Since the new primary keeps the request/response pair for T_2, the new primary immediately returns the response without reexecution. With this, T_2 is executed exactly once and its execution is part of the correct response returned to the client. However, if execution is not deterministic, T_1' might not make the same request 1.1 as in the original execution. Our first solution to handle this problem assumes that compensating transactions exist. Recall that if the 1-N pattern is used to chop a long execution into small pieces, compensating transactions are often provided by programmers. In this case, in the example above, when the discrepancy between the old and new request with id 1.1 is detected, the new primary first calls T_2's compensating transaction and then continues execution. That is, T_2 effects are undone and it appears as if it had never executed. Without compensating transactions the approach becomes best-effort since ghost transactions like T_2 cannot be undone. Since T_2 does not belong to an execution that was perceived correct by the client, T_2 violates the request/transaction matching requirement. In our solution, a client is informed about existing ghost transactions whenever they are detected. Note however that without compensating transactions the N-1 pattern allows even during normal processing that an inner transaction commits while the outer aborts.

4.4 N-N Algorithm

For an N-N execution, the 1-N and N-1 algorithms have to be merged. Complexity arises because of the following situation. Assume a transaction T_1 executing on behalf of one or more client requests. Now assume that T_1 initiates an inner transaction T_2. In the 1-N model, there was one specific request made by T_1 that was executed in the context of T_2. However, with an N-N pattern, T_1 can first start transaction T_2, then make a couple of requests that are executed within T_2, and then request the commit of T_2. If the primary fails after committing T_2 but before committing T_1, the client replays T_1. In order for the reexecution to be successful, it must resubmit all requests associated with T_2, otherwise T_2 becomes a ghost transaction. Furthermore, none of these resubmitted requests may actually be reexecuted because T_2 already committed. Hence, the server must keep track of all request/response pairs associated with committed inner transactions.

4.5 A transaction accesses several databases

In order to handle 2PC, we adjust an idea proposed in [16] for replication of stateless AS to work with stateful AS. For that, we have to slightly change the commit handling of our algorithms (see Figure 4(c)). The primary intercepts the first *prepare* request sent by the TM to a database and multicasts a *preparing* message to the backups before forwarding the request to the database. Then it intercepts the first decision (commit/abort) that the TM sends to one of the databases. In case of commit, it sends a $committing$ message as in our previous algorithms before forwarding the commit to the database. After the transaction has terminated at all databases, the response is returned to the client and a corresponding $commit/abort$ message is multicast to the backups. No txid needs to be inserted into the database.

At the time the old primary crashes, the new primary might have received for a given transaction (1) not yet any message, (2) the $prepared$ message, (3) the $committing$ message, (4) the $abort/commit$ message. In the first case, our failure assumptions guarantee an abort of the corresponding transaction at all databases. In case (2), some might have aborted the transaction, others might be blocked in the prepared state. The new primary can now force all databases to abort the transaction if they have not yet done so. In case (3), some databases might have committed the transaction, others might be blocked, and the backup has received the component state changes. The new primary can now ask all databases to commit the transaction if they have not yet done so. In the last case, nothing needs to be done because all databases and the new primary have the correct state after transaction execution.

5 Client and Database Failures

If the database crashes the AS server has to wait until it recovers (unless the database is replicated itself which is outside the scope of this paper). Upon recovery, the database aborts transactions that were active at the time of the crash. The AS can easily determine whether a transaction has committed by looking for the txid in the database or by being aware of the steps of the 2PC protocol. In case

a transaction was active at the time of crash, and hence, aborted, the AS primary can easily replay the transaction in the 1-1 and 1-N patterns. In the N-1 case it has to forward the abort exception to the client replication algorithm with a request to initiate the replay of the transaction.

If the client crashes, a 1-1 or 1-N execution can simply finish the execution. A N-1 or N-N execution should abort the transaction if the client had not yet submitted the commit request because the AS server only has partial information about the transaction. However, in the N-N pattern this could result in an inner transaction committed while the outer aborted due to the client crash. If a compensating transaction for the inner transaction exists we can apply it.

6 Evaluation

We have integrated the approach into the J2EE server JBoss [17]. For that we used the ADAPT J2EE replication framework [5] which provides interceptor points and functionality like getting and setting component state. As GCS we used Spread [1]. In J2EE both SFSBs and EBs can contain state. However, the state of EBs is always written back to the database at commit time. Hence, committing messages only contain SFSB state changes. We improved the 1-1 algorithm as presented in [27] by parallelizing some tasks. The 2PC solution does not change the TM but uses wrapper objects to intercept the requests from the TM to the databases. Our replication tool detects the execution pattern depending on the requests it intercepts, and automatically applies the corresponding algorithm.

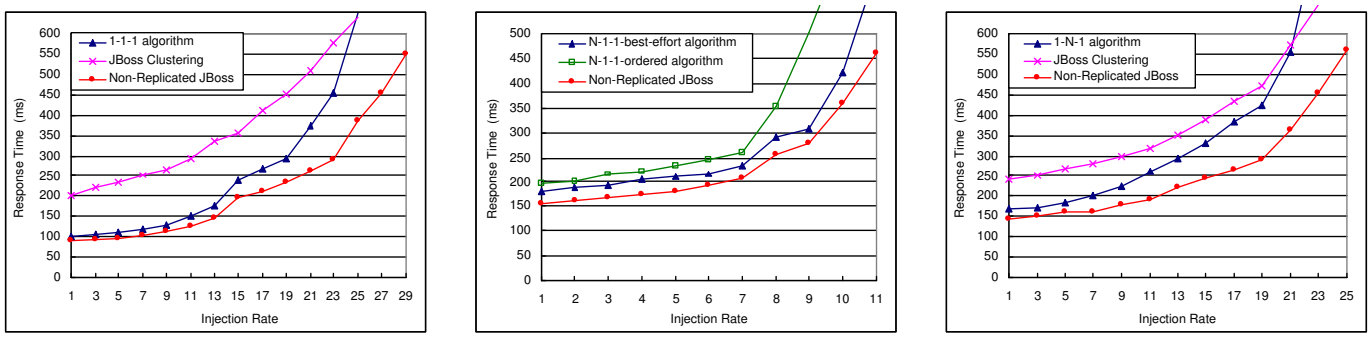
Our performance evaluation analyzes the replication overhead during normal processing using the ECperf benchmark [25]. The ECperf application is split into customer, manufacturing, supplier and corporate domains. The *transaction injection rate* (IR) is an indicator of the load submitted to the system (transactions per second). Results contain the average response time of *order entry* transactions of the customer domain in miliseconds, and the maximum achievable throughput measured in business operations per minute. Results are only measured over the steady state phase (10 minutes) of each test run. We also analyzed failover times but do not present them since the numbers for all tested cases were similar to [27]. Our configuration consists of one machine emulating clients, one web server machine, two machines running JBoss application server 3.2.3., and one machine running the DB2 database system. All machines were PCs 3.0 GHz Pentium 4 with 1 GB of RAM) running RedHat Linux.

Our evaluation compares (1) a regular, non-replicated JBoss server as baseline for comparison; (2) two JBoss server replicas using our replication tool; (3) two JBoss server replicas using JBoss's own replication solution called JBoss clustering. JBoss clustering propagates state to backups on a component basis just before the component returns from a method call. Hence, if several components are called within one client request, several messages are sent. For more detail see related work. For both (2) and (3) one server was primary for all clients. We first looked at the 1-1, N-1, and 1-N patterns using one database, and then look at the 1-1 pattern accessing several databases. We looked at the patterns individually to understand the impact of the particular mechanisms responsible.

1-1 algorithm Figure 7(a) shows the average response times of order entry transactions at increasing IR for the 1-1 execution pattern. Response times for the 1-1 algorithm are 40% better than the original implementation in [27]. At low load, the new 1-1 algorithm adds 15 ms (15% overhead). As a comparison, [21] also indicates around 15% overhead for FT-CORBA (primary-backup) compared to non-replicated CORBA. JBoss clustering adds around 120 ms (120% overhead). The high overhead is due because it sends state after each method invocation while our approach sends one message per transaction. Response times for all setups increase steadily with increasing load until saturation points which is around 27 IR for the non-replicated JBoss, 23 for JBoss clustering and the 1-1 algorithm.

N-1 algorithm Figure 7(b) shows the response times for the N-1 execution pattern. We modified the ECperf implementation so that each order entry transaction contains on average 5 order requests. The figure does not show results for JBoss clustering since response times are five times as high as in the 1-1 model. Response times are generally higher than for the 1-1 model shown in Fig. 7(a) since several client requests are included in one transaction. Compared to no replication, the N-1-best-effort algorithm adds again about 15% overhead while N-1-ordered adds 30%. The latter has higher overhead since it propagates the order in which database access takes place at the end of each client request. Considering that these are five additional messages, the overhead is quite small. This is true because the messages are small and only sent with reliable delivery. In regard to throughput, all configurations saturate much earlier due to CPU overhead. N-1-ordered saturates at 8 IR, N-1-best-effort at 9 IR, and the non-replicated JBoss at 10 IR.

1-N algorithm Figure 7(c) shows the response times for the 1-N execution pattern. We changed the ECperf implementation such that each order entry request triggers an outer transaction which on average contains three inner transactions. Again, response times are generally higher than for the 1-1 execution pattern since now each order entry request includes several transactions. In absolute times, the 1-N algorithm takes more additional time than the 1-1 algorithm in Figure7(a) since we now have to send an additional uniform-reliable message for each inside transaction. In contrast, JBoss clustering adds the same time (120



(a) 1-1 (b) N-1 (c) 1-N

Figure 7. ECperf Response Time Comparison

ms) as in the 1-1 pattern since the replication mechanism is not related to transactions. In terms of throughput, the 1-N algorithm saturates at 21 IR, JBoss clustering saturates at 23 IR, and the non-replicated JBoss saturates at 25 IR. The 1-N algorithms saturates earlier than JBoss because of the increased bookkeeping to guarantee exactly-once execution and state consistency, and to detect ghost transactions.

1-1 with 2PC For this experiment we have not used the ECPerf but a simpler evaluation. A client submits one request to a SFSB which performs two database updates that either access the same database (no 2PC) or different databases (requiring a 2PC). Table 1 shows the average response time at a load of 10 transactions per second, and the maximum achievable throughput. Accessing one database, the 1-1 algorithm adds 5.4 ms to the response time of the non-replicated JBoss reflecting a 15% increase, while with a 2PC, the 1-1 algorithm has an overhead of 8.3 ms (it has to send an additional *preparing* message) but this reflects an increase of only 8%. The maximum throughput for the 1-1 algorithm compared to the non-replicated case is around 90% with a 2PC and 86% when one database is accessed. The 1-1 algorithm performs, in relative terms, better with a 2PC than without because the total response times with a 2PC is so much higher than if no 2PC is necessary.

In summary, these experiments show that our solutions in general incur little overhead for all typical execution patterns. Our replication tool clearly outperforms JBoss's clustering mechanism in all cases in terms of response time, and is similar in terms of saturation point.

7 Related Work

Looking at J2EE servers, JBoss [17] makes each replica primary for some clients. Since state is propagated each time a component returns from a method call, several messages might be sent for each client request. As a result, if the primary crashes in the middle of executing a request the state changes for some components might be propagated but others not. Pramati [23] logs state changes in the database and receives them upon recovery after a crash. WebLogic [6] uses a single primary server replica, and only propagates changes after commit. None of the systems supports advanced execution models and only Pramati provides state consistency for the 1-1 pattern.

There exists many proposals for replication of CORBA components, e.g., [8, 9, 22, 19, 12, 2]. Most of them do not consider database access. [28] extend the CORBA based fault-tolerant Eternal system [22] to work correctly with a database backend tier. The transaction context within the components is replicated, and duplicate requests to the database are suppressed. However, component execution has to be deterministic. [13] combine replication and transactions for CORBA using an approach similar to the 1-1 algorithm. However, none of the approaches provides advanced execution patterns.

Phoenix [4, 3] for .NET handles the 1-1 pattern with one database using checkpoints and request/reply logging. Failover starts from the last checkpoint and applies logged requests assuming piecewise deterministic behavior [11].

Outside of any concrete AS architecture, [15] provides exactly-once execution for stateful AS for the 1-1 pattern accessing one database. [14, 16] provide exactly-once semantics for *stateless* AS for the 1-1 pattern. Our algorithms use similar mechanisms to check the status of database transactions. We advance this existing work by looking at stateful AS and advanced execution patterns, and by integrating our solution into a real AS architecture.

There has been a lot of work on database replication. However, the underlying model is quite different since it does not consider state changes outside the database.

8 Conclusion

This paper presents a replication tool for AS servers that is able to combine replication and transactions for advanced execution patterns providing strong correctness properties.

Model	Algorithm	Response Time (ms)	Tx numbers (per second)
one database	Non-replicated JBoss	34.9	30
	1-1 algorithm	40.3	26
more than one database	Non-replicated JBoss	103.5	10
	1-1 algorithm	111.8	9

Table 1. 1-1 execution accessing one or more than one database

Our solution does not require any special properties from the component implementations, the clients or the database. We have integrated our solution into the J2EE server JBoss, but we believe that the main ideas can be applied to other architectures. Our approach has comparable or better performance than existing solutions while providing stronger semantics for many different execution patterns.

Our current research looks at fault-tolerance if a web-server is applied between client and AS. Any web-server replication must be coordinated with our AS replication. We are also currently extending our algorithms to allow more than one primary in order to distributed client request across several replicas.

References

[1] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The Spread toolkit: Architecture and performance. Technical Report CNDS-2004-1, Cen. for Netw. and Distr. Systems, Johns Hopkins Univ., 2004.

[2] R. Baldoni and C. Marchetti. Three-tier replication for FT-CORBA infrastructures. *Software – Practice and Experience*, 33(8), 2003.

[3] R. Barga, S. Chen, and D. Lomet. Improving logging and recovery performance in Phoenix/App. In *Int. Conf. on Data Engineering (ICDE)*, 2004.

[4] R. Barga, D. Lomet, and G. Weikum. Recovery guarantees for general multi-tier applications. In *Int. Conf. on Data Engineering (ICDE)*, 2002.

[5] A. Bartoli, V. Maverick, S. Patarin, and H. Wu. A Framework for Prototyping J2EE Replication Algorithms. In *Int. Symp. on Distrib. Objects and Applications (DOA)*, 2004.

[6] BEA Systems Inc. *BEA WebLogic Server, release 7.0: Programming WebLogic Enterprise JavaBeans*, 2002.

[7] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4), 2001.

[8] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz. AQuA: an adaptive architecture that provides dependable distributed objects. In *Symp. on Reliable Distributed Systems (SRDS)*, 1998.

[9] X. Défago, P. Felber, and A. Schiper. Replicating CORBA objects: a marriage between active and passive replication. In *Work. Conf. on Distrib. Appl. and Interop. Systems*, 1999.

[10] X. Défago and A. Schiper. Semi-passive replication and lazy consensus. *J. Parallel Distrib. Comput.*, 64(12), 2004.

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

[12] P. Felber, R. Guerraoui, and A. Schiper. Replication of CORBA objects. In S. Shrivastava and S. Krakowiak, editors, *Advances in Distributed Systems*. LNCS 1752, Springer, 2000.

[13] P. Felber and P. Narasimhan. Reconciling replication and transactions for the end-to-end reliability of CORBA applications. In *Int. Symp. on Distributed Objects and Applications (DOA)*, 2002.

[14] S. Frølund and R. Guerraoui. A pragmatic implementation of E-transactions. In *Symp. on Reliable Distributed Systems (SRDS)*, 2000.

[15] S. Frølund and R. Guerraoui. X-ability: a theory of replication. In *Symp. on Princ. of Distrib. Comp. (PODC)*, 2000.

[16] S. Frølund and R. Guerraoui. E-transactions: End-to-end reliability for three-tier architectures. *IEEE Transactions on Software Engineering (TSE)*, 28(4), 2002.

[17] T. J. Group. JBoss application server. http://www.jboss.org.

[18] R. Guerraoui and A. Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4), 1997.

[19] M.-O. Killijian and J. C. Fabre. Implementing a reflective fault-tolerant CORBA system. In *Symp. on Reliable Distributed Systems (SRDS)*, 2000.

[20] S. Kounev and A. P. Buchmann. Improving data access of J2EE applications by exploiting asynchronous messaging and caching services. In *Int. Conf. on Very Large Data Bases (VLDB)*, 2002.

[21] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. A fault tolerance framework for CORBA. In *Int. Symp. on Fault-Tolerant Computing*, 1999.

[22] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Strongly consistent replication and recovery of fault-tolerant CORBA applications. *Journal of Computer System Science and Engineering*, 32(8), 2002.

[23] Pramati Technologies Private Limited. *Pramati Server 3.0 Administration Guide*, 2002.

[24] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems*, 20(3), 1995.

[25] SUN Microsystems Inc. $ECperf^{TM}$ *specification, v1,1*.

[26] SUN Microsystems Inc. *JAVA 2 Platform Enterprise Edition Specification, V1.3*.

[27] H. Wu, B. Kemme, and V. Maverick. Eager replication for stateful J2EE servers. In *Int. Symp. on Distributed Objects and Applications (DOA)*, 2004.

[28] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Unification of replication and transaction processing in three-tier architectures. In *Int. Conf. on Distributed Computing Systems (ICDCS)*, 2002.