Exactly Once Interaction in a Multi-tier Architecture*

Bettina Kemme†, Ricardo Jiménez-Peris*, Marta Patiño-Martínez*, Jorge Salas*

†McGill University, Montreal, Canada, kemme@cs.mcgill.ca
*Universidad Politcnica de Madrid (UPM), Madrid, Spain, {rjimenez,mpatino }@fi.upm.es, jsalas@alumnos.upm.es

Abstract

Multi-tier architectures are now the standard for advanced information systems. Replication is used to provide high-availability in such architectures. Most existing approaches have focused on replication within a single tier. For example there exist various approaches to replicate CORBA or J2EE based middle-tiers, or the database tier. However, in order to provide a high-available solution for the entire system, all tiers must be replicated. In this paper we analyze what is needed to couple two replicated tiers. Our focus is to analyze how to use independent replication solutions, one for each tier, and adjust them as little as possible to provide a global solution.

1 Introduction and Background

Current information systems are often built using a multi-tier architecture. Clients connect to an application server (also called middle-tier) which implements the application semantics. The application server in turn accesses a database system (also called backend tier) to retrieve and update persistent data. Application server (AS) and database (DB) together build the server-side system, while the client is external and usually an independent unit. In this paper, we do not consider architectures where an AS calls another AS or several DBs. The standard mechanisms to provide high-availability for the server system are logging with fast restart of failed components, or replication. [4, 3] look at fault-tolerance across tiers via logging. The idea of replication is that for both AS and DBS, there are several server replicas, and if one server replica crashes others can take over. Existing replication solutions, both in academia and an industry have focused on the replication of a single tier. For instance, [8, 12, 19, 22, 15, 14, 13, 31, 30, 26, 5, 18, 16] only look at AS replication. Many of these approaches do not even consider that the AS accesses a database via transactions which have to be handled in case of an AS replica crash. Only recent solutions take such database access into account. In regard to database replication, some recent approaches are [2, 17, 6, 23, 1, 24, 7, 9, 25, 30, 21]. Again, these approaches do not consider that the client (namely the AS server) might be replicated. Note that an alternative way to achieve high availability is logging with fast restart of failed components.

However, in order to attain high availability, all tiers should be replicated. Providing a correct replication solution when considering a single tier has already shown to be non-trivial. Providing the same degree of correctness when multiple tiers are replicated is even more challenging [20, 10].

We first have to understand the relationship between AS and DB. Typically, both maintain some state. The DB contains all data that can be accessed by different users (shared data) and that should survive client sessions. The AS maintains volatile data. For instance, the J2EE specification for AS distinguishes between data that is only accessible by a single client during the client session (kept within stateful session beans), and data cached from the DB that is accessible to all clients (kept within entity beans). There is typically no data that is shared between clients but is not persisted in the DB. Typically, for each client request, a transaction is started, and all actions are performed within the context of this transaction. If all actions are successful the transaction commits before a response is returned to the user. Otherwise, the transaction aborts, all state changes performed so far are undone (typically both in the AS and DB), and the client is notified accordingly. A transaction might abort because of some application semantics (e.g., not enough credit available). If now either the AS or the DB crashes, request execution is interrupted and the client usually receives a failure exception. The task of replication is to hide such failures from the client. Instead, replication should provide exactly-once execution despite any possible failures. That is, a client receives for each request submitted exactly one response. This response might be an abort notification that was caused by the application semantics but no failure exception. Furthermore, the server has either executed and committed exactly one transaction on behalf of the client request or, in case of an abort notification, no state changes on behalf of the request are reflected at the server.

*This work has been partially supported by the MDER-Programme PSIIRI: Projet PSIIRI-016/ADAPT (Québec), by NSERC (Canada) Rgpin 23910601, by the European Commission under the Adapt project grant IST-2001-37126, and by the Spanish Research Council (MEC) under grant TIN2004-07474-C02-0

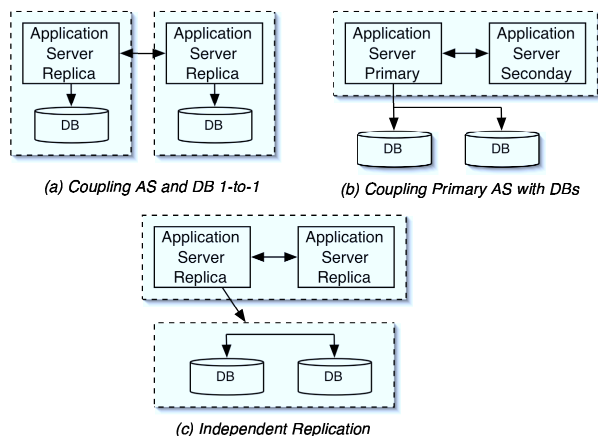

Figure 1: Replication Architectures

In order to handle an AS or DB crash, both AS and DB should be replicated. The idea is that any state changes performed by a transaction are known at all replicas before the response is returned to the client. In this case, if a replica fails, the state changes of successfully executed transactions are not lost. We see two ways to perform replication across tiers. A *tightly coupled* approach has one global replication algorithm that coordinates replication within and across tiers. The algorithm is developed with the awareness that both tiers are replicated. In contrast, a *loosely coupled* approach takes two existing replication algorithms, one for each tier, plugs them together and adjusts them such that they work correctly together.

For simplicity, we only look at primary/backup approaches. Each client has a primary AS replica which executes all the requests of this client and sends the state changes to the backup replicas. When the primary fails, a backup takes over as new primary for this client. A *single primary* approach requires all clients to connect to the same primary, in a *multiple primary* approach each replica is primary for some clients and backup for the other primaries.

Fig. 1 (a) and (b) show tightly coupled approaches. Only the AS is responsible to coordinate replication. We use the term DB copies instead of replicas to express that the DB does not actively control replication.

Fig. 1 (a) presents a tightly coupled *vertical replication* approach. Each AS replica is connected to one DB copy, and each AS replica must make sure that its DB copy contains the same updates as the other DB copies. That means, the AS primary of a client has to send not only all state changes within the AS to the AS backups but also enough information so that the AS backups can update their DB copies correspondingly. Within J2EE, if all DB access is done via entity beans (no SQL statements within session beans), then this can be achieved by sending both changes on session and entity beans to AS backups since the entity beans reflect all DB changes. Otherwise, SQL statements might have to be re-executed at the AS backups. If either an AS replica or a DB copy fail, the corresponding DB copy (resp. AS replica) has to be forced to fail, too. This approach has several challenges. All AS replicas have to guarantee to commit the same transactions with the same AS and DB changes despite the possibility of interleaved execution of different client requests at different replicas[1]. But even if DB access is not interleaved (e.g., using a single primary), guaranteeing the same DB changes at all sites might be difficult if non-determinism is involved (e.g., SQL statements contain time values). Furthermore, when an AS / DB replica pair recovers, the AS replica must assure that the DB copy receives the current DB state. The DB copy cannot help with this task because it is not even aware of replication. Hence, recovery is a challenging task.

Fig. 1 (b) is an example of tightly coupled *horizontal replication* with a single primary AS. The AS primary is connected to all DB copies and performs the necessary updates on all these copies. At the time the AS primary crashes, a given transaction might have committed at some DB copies, be active at others, and/or has not even started at some. When an AS backup takes over as new AS primary, it has to make sure that such transaction eventually either commits or aborts at all DB copies. One solution is to perform all DB updates within a single distributed transaction that terminates with a 2-phase commit protocol (2PC). If during the 2PC the AS primary informs the AS backups in which phase a transaction is (e.g., before prepare, after prepare, etc.), the new AS primary can commit or abort any outstanding transactions appropriately [15]. However, 2PC is very time consuming. Since the 2PC was only introduced for replication purposes this solution very expensive. Also, DB recovery is again a challenge.

A loosely coupled integration approach is shown in Fig. 1(c). Since so many solutions exist for replication of the individual tiers the idea is to simply couple any replication solution for one tier with a replication solution for the other tier. Assume the replication solution for the AS tier guarantees exactly-once execution under the assumption that AS replicas might crash but the DB to be accessed is reliable. Further assume the replication solution for the DB tier expects a non-replicated client and guarantees that each transaction either commits or aborts at all replicas. Finally assume that the DB provides an interface such that its clients are actually not aware that they are connected to a replicated DB but view it as a single, reliable DB. The question is whether plugging these two replicated tiers together without any further actions on either of the tiers really provides exactly-once execution across both tiers in the presence of AS and DB crashes.

In the following, we analyze this issue in detail. We take existing replication solutions for the two tiers, and analyze which failure cases are handled correctly and for which cases changes or enhancements have to be made to one or both of the replication algorithms in order to provide correctness across the entire server system. We first look at single primary approaches, and then discuss the challenges associated with multiple primary solutions.

[1] J2EE AS replication with a non-replicated DB is simpler, since the concurrency control of the central DB handles all access to shared data.

2 Application Server Replication

Our example of a single primary AS approach is taken from [29]. Other approaches use similar techniques [15, 13]. The approach is for J2EE architectures, assumes a reliable centralized database and reliable communication. An AS replica might crash. If it was connected to the DB and had an active transaction at the DB (no commit submitted yet), the DB aborts this transaction upon connection loss. For space reasons we bring a simplified version that does not consider application induced aborts.

The replication algorithm has a client, primary, backup and failover part. At the client, a client replication algorithm (CRA) intercepts all client requests, tags them with an identifier and submits them to the AS primary (after performing replica discovery). If the CRA detects the failure of the primary, it reconnects to the new primary. Furthermore, it resubmits the last request with the same identifier if the response was still outstanding. In J2EE, upon receiving a request, the AS server first initiates a transaction and then calls the session bean associated with this request. The session bean might call other session or entity beans. Each of these beans might also access the DB. The primary replication algorithm (PRA) intercepts transaction initiation to associate the request with the transaction. It intercepts the calls to beans in order to capture the state changes. When it intercepts the commit request, it sends a checkpoint containing the state of all changed session beans, the request identifier and the client response to the AS backups. Additionally, a marker containing the request identifier is inserted into the DB as part of the transaction. Backups confirm the reception of the checkpoint. Then, the PRA forwards the commit to the DB, and the response is returned to the client. For each session bean, backups only keep the state of the bean as transmitted in the last two checkpoints that contain the bean. If the primary fails, one backup is elected as new primary NP. For each client c, NP performs the following failover steps. Let r with associated transaction t_r be the last request of c for which NP received a checkpoint cp_r. NP checks whether t_r committed at the DB by looking for the marker in the DB. If it exists, t_r committed. Otherwise, it aborted due to the crash of the old primary. NP does not perform checks for earlier requests of c because each new checkpoint is an implicit acknowledgement that previous transactions of c committed. Also, if t_r committed, NP keeps the response rp_r found in cp_r. NP sets the state of each session bean b to the state found in the last checkpoint $cp_{r'}$ containing b and transaction $t_{r'}$ committed. Then, NP starts the PRA algorithm. If the CRA did not receive a response for the last request r, it resubmits it to NP. Either NP has stored rp_r and immediately returns it or it reexecutes r like a new request.

To see why this leads to exactly-once execution, we can distinguish the following timepoints at which the AS primary can crash. (1) If it fails before sending the checkpoint cp_r, then the corresponding transaction t_r aborts, and the new primary NP has no information about r. The CRA resubmits r and it is executed as a new request. (2) If it fails after sending cp_r but before committing t_r at the DB, t_r aborts. NP checks in the DB but does not find the marker, hence ignores the state changes and response found in cp_r. The CRA resubmits r and it is executed as a new request. (3) If it fails after committing t_r but before returning the response, NP finds the marker, applies the state changes on the session beans, and keeps the response rp_r. When CRA resubmits r, NP immediately returns rp_r. r is not again executed. (4) If it fails after returning rp_r to the client, t_r committed, NP has the state changes on beans, and the CRA does not resubmit r providing exactly-once.

3 Database Server Replication

Commercial databases have provided high-availability solutions for a long time [11]. However, since the documentation available to us is not very precise, the following describes our suggestion of a highly-available solution with a single DB primary and one DB backup (adjusted from [21]).

All communication with the DB is via the JDBC driver provided by the DB. The JDBC driver runs in the context of the application. Upon a connection request from the application, the JDBC driver connects to the DB primary (address can be read from a configuration file). The application submits transaction commands and SQL statements through the JDBC driver to the DB primary where they are executed. Upon the commit request from the application, the DB primary propagates all changes performed by the transaction in form of a writeset to the backup. It waits until the backup confirms the reception of the writeset. Then it commits the transaction and returns the ok to the application. Writesets are sent in FIFO order, and the backup applies the writesets in the order it receives it.

If the DB primary crashes the JDBC driver looses its connections. The driver automatically reconnects to the backup which becomes the new primary. At the time of crash a connection might have been in one of the following states. (1) No transaction was active on the connection. In this case, failover is completely transparent. (2) A transaction T was active and the application has not yet submitted the commit request. In this case, the backup does not know about the existence of T. Hence, T is lost. The JDBC driver returns an appropriate exception to the application. But the connection is not declared lost, and the application can restart T. (3) A transaction T was active and the application already submitted the commit request, but it did not receive the commit confirmation from the old DB primary before its crash. In this case, the backup (a) might have received and applied T's writeset and committed T, or (b) it did not receive T's writeset before the crash. Hence, it does not know about the existence of T, and T must be considered aborted as under case (2).

Let's have a closer look at case 3. Generally, if a non-replicated DB crashes after a commit request but before returning the ok, the application does not know the outcome of the transaction. With replication, however, we can do better. When a new transaction starts at the DB primary,

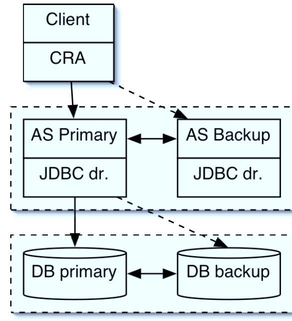

Figure 2: Loose Coupling of single primary AS and DB

the DB primary assigns a unique transaction identifier and returns it to the JDBC driver. Furthermore, the identifier is forwarded to the backup together with the writeset. If the DB primary crashes before returning the ok for a commit request, the JDBC driver connects to the backup and inquires about the commit of the in-doubt transaction (using the transaction identifier). If the backup did not receive the writeset before the crash (case 3b), it does not recognize the identifier and informs the JDBC driver accordingly. The JDBC driver returns the same exception to the application as in case 2. If the backup received the writeset (case 3a), it recognizes the identifier, and returns the commit confirmation to the JDBC which informs the application. In this case, failover is transparent. Garbage collection is quite simple because for each connection the JDBC driver might ask only for the outcome of the last transaction.

One has to be aware that, due to the asynchrony in the system, the backup might receive the inquiry about a transaction from a driver and after that it receives the writeset for the transaction (the primary had sent the writeset before the crash but the backup had not yet retrieved it from the communication channel). In order to handle this correctly, the backup does not immediately return to the JDBC driver if its does not find the transaction identifier. Instead, before allowing any JDBC requests, it switches to failover and first applies and commits all outstanding writesets that were successfuly transferred to the backup before the primary's crash. Only then, it responds to JDBC requests.

The approach above is actually quite similar to the combination of CRA/PRA algorithm for AS replication where the JDBC driver takes over the task of CRA. The main difference is that in AS replication, each request was executed in an individual transaction that started at the AS. With this, it is easy to provide exactly-once, and failover is completely transparent. In contrast, in the DB environment, the application starts and ends a transaction, and several requests can be embedded in this transaction. Hence, if the primary crashes in the middle of executing the transaction, the application receives a failure exception. Hence, execution is actually at-most once.

4 AS / DB Integration

Fig. 2 shows how the algorithms of Sections 2 and 3 are coupled. We can distinguish different failure cases.

4.1 DB primary fails, AS primary does not fail

We look at the state of each connection between AS primary and DB primary at the time the DB primary crashes.

If no transaction was active, the AS primary does not even notice that the driver reconnects to the DB backup. If a transaction t_r triggered by client request r was active but the AS primary had not yet submitted the commit, the JDBC driver returns a failure exception and the AS primary knows that t_r aborted. t_r might already have changed some state (beans) at the AS primary leading to inconsistency. The task of the AS primary is to resolve this inconsistency and hide the DB crash from the client, i.e., provide exactly-once execution for r despite the DB primary crash. This task is actually quite simple. The AS primary has to undo the state changes on the beans executed on behalf of t_r. Then, it simply has to restart the execution of r initiating a new transaction. The JDBC driver has already connected to the DB backup which is now the new DB primary. The AS primary is not even aware of this reconnection. Reexecuting the client request is fine since all effects of the first execution have been undone at the AS and the DB, and no response has yet been returned to the client.

In the third case the DB primary fails after the AS primary submitted the commit request for t_r but before the ok was returned. In this case, the JDBC driver detects whether the DB backup committed t_r or not. Accordingly, it returns a commit confirmation or exception to the AS primary (case 3 of Section 3). In case of commit, the AS primary is not even aware of the DB failover and returns the response to its client as usual. In case of an exception it should behave as above. It should undo the state changes on beans performed by t_r and reexecute r. There is one more issue. Since the AS primary first transfers the checkpoint cp_r for r to the AS backups and then submits the commit to the DB, the AS backups have cp_r containing the changes of aborted transaction t_r. There are now two cases. Firstly, the AS primary successfully reexecutes r and sends a new checkpoint for r to the AS backups. In this case, the AS backups should discard the old, invalid checkpoint. Secondly, the AS primary might crash during reexecution before sending a new checkpoint. In this case, the AS backup that takes over as new AS primary checks for the marker (corresponding to the old checkpoint) but will not find it in the DB, and discard the checkpoint. That is, in any case, the old invalid checkpoint is ignored.

In summary, little has to be done in case of the crash of the DB primary in order to correctly couple the two replication algorithms. The only action that has to be performed is the following: whenever the AS primary receives a failure exception from the JDBC driver for a transaction t, it has to abort t at the AS level, and restart the execution of the corresponding client request.

4.2 DB primary does not fail, AS primary fails

When the AS primary fails its connections to the DB primary are lost. The DB primary aborts each active transaction for which it did not receive the commit request before the crash. This is the same behavior as that of a centralized DB system. At AS failover, the new AS primary connects to the DB primary and checks for the markers for the last checkpoints it received from the old AS primary. Since it is connected to the same DB replica as the old AS primary was, it will read the same information as in a centralized DB system. As a result, nothing has to be done in case of the crash of the AS primary in order to correctly couple the two replication algorithms. The failover actions of the AS replication algorithm of Section 2 are correct, whether the AS is connected to a reliable centralized DB system or to a replicated DB based on the algorithm of Section 3.

4.3 Both DB and AS primaries fail

Crash at the same time This is possible if DB and AS primaries run on the same machine, and the machine crashes. In this case the JDBC driver of the new AS primary connects to the new DB primary. Nevertheless, failover can be performed in exactly the same way. There is only one issue. The new DB primary may not execute any requests from the new AS primary before it has applied and committed all writesets it has received from the old DB primary, i.e., before failover is completed. Otherwise, the new AS primary could check for a marker for a request r, not find it, and only after that the new DB primary processes the writeset of the corresponding transaction t_r and commits t_r. In this case, the new AS primary would discard r's checkpoint and reexecute r leading to a new transaction t'_r although t_r already committed at the DB.

Crash at different times The interesting case is if the AS primary first fails, the new AS primary performs failover, and while checking for markers in the DB primary, the DB primary crashes. Checking for a marker is a simple transaction. If the DB primary fails in the middle of execution, the JDBC driver returns a failure exception to the new AS primary. The new AS primary can simply resubmit the query, and the JDBC driver redirects it to the new DB primary where it will be answered once the new DB primary has processed all writesets from the old DB primary.

4.4 Summary

The discussion above shows that with the two particular AS and DB replication algorithms, the coupling is extremely simple. There is only one slight modification to the AS replication algorithm. Since the failure of the DB primary is not completely transparent (the application receives failure exceptions for any active transaction), the AS might have to reexecute a request if the DB primary fails. No other changes have to be performed.

5 Multiple Primary Approaches

Recall that with multiple primaries, each replica can be primary of some clients and backup for the other primaries.

5.1 Multiple AS Primaries

Extending above single primary AS algorithm to allow for multiple primaries is straightforward as long as client sessions are sticky (a client always interacts with the same AS replica during the lifetime of a session unless the AS replica crashes), and as long as access to shared data is synchronized via the DB tier [2]. Some load balancing mechanism is needed to assign new clients to one of the AS replicas but the basics of the replication algorithm can remain the same.

Coupling with a single DB primary We can use the failover mechanism of the single AS primary solution presented in Section 4 without any changes. If any of the AS replicas fails, only the clients for which this AS replica was primary must be failed over to another AS replica.

5.2 Multiple DB Primaries

Many recent systems [17, 23, 27, 24, 30, 21] allow an application to connect to any DB replica which executes the transaction locally and at commit time multicasts the writeset to the other DB replicas. Since transactions on different DB replicas might access the same data, conflicts must now be detected across the system. A typical solution is to use the primitives of a group communication system (GCS) [28]. The replicas build a group and writesets are multicast such that all group members receive the writesets in the same total order. If two transactions are concurrent and conflict then the one whose writeset is delivered later must abort at all replicas. There exist many solutions to detect such conflicts using locking, optimistic validation, snapshots, etc. GCS also detect the crash of group members and inform surviving members with a view change message. Writesets are usually multicast with a uniform reliable delivery guaranteeing that if one DB replica receives a writeset each other replica also receives it or is removed from the group.

Failover after the crash of a DB replica is proposed in [21] and nearly as described in Section 3. The replicated DB has one fixed IP multicast address. To connect the extended JDBC driver multicasts a discovery message to this address. DB replicas that are able to handle additional workload respond and the driver connects to one of them. Let's denote it with db. If db crashes, the JDBC driver reconnects to another DB replica db'. Only crash case 3 of Section 3 where the commit for a transaction t was submitted but db crashes before returning an answer must be handled slightly different than in Section 3. Due to the asynchrony of message delivery, the JDBC driver might inquire about the commit of t at db', and only afterwards db' receives t's writeset. In order to handle this correctly t's identifier contains information that t was executed at db.

[2]Transactions on different AS replicas may access shared data via entity beans but access is synchronized with the DB before commit.

Then, db' waits until the GCS informs it about the crash of db. According to properties of the GCS, db' can be sure that it either receives t's writeset before the view change removing db (and then, tells the JDBC driver about the outcome), or not at all (and then, returns a failure exception).

Coupling with a single AS primary Assume the AS primary is connected to DB replica db.

If neither the AS primary nor db fail, then the only difference to Section 4 is that a transaction t_r might now abort at the DB tier at commit time because of a conflict. The AS primary can hide such abort could from the AS client by undoing the AS state changes of t_r and reexecuting r as done in Section 4.1 when the transaction aborts due the crash of the DB primary.

If the AS primary does not fail but db fails, the AS primary might receive an abort or failure exception for a transaction t_r. As in Section 4.1, the AS primary aborts t_r at the AS level and reexecutes r.

If the AS primary fails and the new AS primary connects again to db, the situation is as in Section 4.2.

The only really interesting case is if the AS primary fails, and the new AS primary NP connects to DB replica $db' \neq db$ (this might happen due because of load-balancing issues or because db also fails). NP checks for markers in db'. These are simple read only transactions. However, we have again the problem of asynchrony. Although db' might have received t_r's writeset it might still execute it while NP checks for r's marker. Hence, NP will not find the marker but t_r later commits. Conceptually, the problem is similar to the JDBC driver inquiring about the commit of a transaction but the DB replica might not yet have processed the writeset. The difference is that the JDBC driver is part of the DB replication system. Hence, coordination is simpler. When db' receives an inquiry from the JDBC driver for a transaction that was executed on db, it knows it has to wait until it either receives the writeset or a view change message from the GCS. However, when the NP looks for a marker, this is a completely new, local transaction, and db' cannot know that this transaction actually inquires about t_r.

In order to allow NP to connect to any DB replica db' we suggest to extend both the AS and DB replication solutions slightly. Firstly, we make a JDBC connection object a "state" object which keeps track of the last transaction t_r associated with the connection. t_r's identifier implicitly contains the identifier of the DB replica it is executed on (e.g., db). Secondly, we make the submission of the commit request over a given connection to the replicated DB an idempotent operation. We show shortly how this is achieved. Furthermore, the new AS primary NP has to perform the following actions at failover. Instead of checking for the marker of r for the last checkpoint cp_r of a client, NP submits the commit request for t_r using the connection object found in cp_r. At this timepoint, the connection object is not really connected to any DB replica. Hence, it connects to any DB replica db' and inquires about the commit of t_r. Assume first that $db' = db$. db, before the old AS primary crashed might have already received t_r's commit request or not. In the first case, it had either committed t_r or aborted due to conflict. In the second case, it has aborted t_r due to the crash of the AS primary. Hence, it returns the corresponding outcome to the driver which returns it to NP. In case of commit, NP applies the state in cp_r and keeps track of the response, otherwise it discards cp_r and restarts execution of r when the client resubmits. If $db' \neq db$, then db' can detect by looking at t_r that t_r was originally executed at db. db' knows that the driver would only send a commit inquiry of a transaction executed on db if db crashed. Hence, it waits until it has received from the GCS t_r's writeset or the view change message excluding db from the group. In the first case, it returns a commit/abort answer depending on conflicts. In the second case, it returns a failure exception. The driver forwards this decision to NP which handles cp_r accordingly.

With this mechanism, there is actually no need for the marker mechanism. Instead of looking for the marker, the new AS primary simply submits the commit request over the connection object copy. It either receives the outcome of the transaction (commit/abort) or a failure exception. Hence, this extended functionality of the DB replication algorithm – allowing a resubmission of a commit request (with idempotent characteristics) – provides additional functionality over a centralized system. As a result, the AS replication algorithm can be simplified avoiding to insert a marker for each transaction.

6 Conclusions

This paper analyzes various approaches for replication both at AS and DB tier. The main focus is to combine typical existing replication solutions, developed for the replication of one tier, to provide a replication solution for the entire multi-tier architecture. We show that only minor changes need to be performed to the existing solutions in order to provide exactly-once execution across the entire system. One main issue is that the replicated AS tier should hide DB crashes from its own clients. This is easy to achieve. The second main issue is for the AS tier to detect whether a given transaction committed at the DB tier in the presence of crashes of AS and/or DB replicas. A transparent solution is embedded in a replication aware JDBC driver.

References

[1] C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *ACM/IFIP/USENIX Int. Middleware Conf.*, 2003.

[2] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: Are these mutually exclusive? In *SIGMOD Int. Conf. on Management of Data*, 1998.

[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] BEA Systems. *WebLogic Server 7.0. Programming WebLogic Enterprise JavaBeans*, 2005.

[6] K. Böhm, T. Grabs, U. Röhm, and H.-J. Schek. Evaluating the coordination overhead of synchronous replica maintenance. In *Euro-Par*, 2000.

[7] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: flexible database clustering middleware. In *USENIX Annual Technical Conference, FREENIX Track*, 2004.

[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] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *Int. Conf. on Data Engineering (ICDE)*, 2004.

[10] E. Dekel and G. Goft. ITRA: Inter-tier relationship architecture for end-to-end QoS. *The Journal of Supercomputing*, 28, 2004.

[11] S. Drake, W. Hu, D. M. McInnis, M. Sköld, A. Srivastava, L. Thalmann, M. Tikkanen, Ø. Torbjørnsen, and A. Wolski. Architecture of highly available databases. In *Int. Service Availability Symposium (ISAS)*, 2004.

[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)*, Nürnberg, Germany, 2000.

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

[16] The JBoss Group. JBoss application server. http://www.jboss.org.

[17] J. Holliday, D. Agrawal, and A. El Abbadi. The performance of database replication with group communication. In *Int. Symp. on Fault-Tolerant Computing (FTCS)*, 1999.

[18] IBM. *WebSphere 6 Application Server Network Deployment*, 2005.

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

[20] A. I. Kistijantoro, G. Morgan, S. K. Shrivastava, and M. C. Little. Component replication in distributed systems: a case study using Enterprise Java Beans. In *Symp. on Reliable Distributed Systems (SRDS)*, 2003.

[21] Y. Lin, B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. Middleware based data replication providing snapshot isolation. In *SIGMOD Int. Conf. on Management of Data*, 2005.

[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] E. Pacitti, P. Minet, and E. Simon. Replica consistency in lazy master replicated databases. *Distributed and Parallel Databases*, 9(3), 2001.

[24] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1), 2003.

[25] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *ACM/IFIP/USENIX Int. Middleware Conf.*, 2004.

[26] Pramati Technologies Private Limited. *Pramati Server 3.0 Administration Guide*, 2002. http://www.pramati.com.

[27] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong Replication in the GlobData Middleware. In *Workshop on Dependable Middleware-Based Systems*, 2002.

[28] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group communication specification: A comprenhensive study. *ACM Computing Surveys*, 33(4), 2001.

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

[30] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *Int. Conf. on Data Engineering (ICDE)*, 2004.

[31] W. Zhao, L.E. Moser, and P.M. Melliar-Smith. Unification of transactions and replication in three-tier architectures based on CORBA. *IEEE Transactions on Dependable and Secure Computing*, 2(1):20– 33, 2005.