Highly Available Long Running Transactions and Activities for J2EE Applications*

Francisco Pérez-Sorrosal[1], Jaksa Vuckovic[2], Marta Patiño-Martínez[1], Ricardo Jiménez-Peris[1]
[1]*Universidad Politécnica de Madrid, Spain* [2]*Universtà di Bologna, Italy*
{fpsorrosal,mpatino,rjimenez}@fi.upm.es *vuckovic@cs.unibo.it*

Abstract

Today's business applications are typically built on top of middleware platforms such as J2EE and use transactions that have evolved into long running activities able to adapt to different circumstances. Specifications, such as the J2EE Activity Service, have arised for applications requiring that support. These applications also demand high availability to prevent financial losses and/or service level agreements (SLAs) violations due to service unavailability or crashes. Replication is a means to attain high availability but current middleware does not provide highly available transactions. In the advent of crashes, running transactions abort and the application is forced to re-execute them, what results in a loss of availability and transparency. Most approaches using J2EE consider the replication of either the application server or the database. This results in poor availability when the non-replicated tier crashes. This paper presents a novel J2EE replication support for both, application server and database layers providing highly available transactions and long running activities. Failure masking is transparent to client applications. A prototype has been implemented and evaluated.

1 Introduction

The increasing degree of sophistication and complexity of modern business applications is demanding support for flexible long running activities. Due to the long running nature of business activities, traditional ACID transactions are not adequate for them. Several specifications have been proposed in the last years such as the activity service for CORBA [29] and J2EE [33], WS-Coordination/WS-Transaction [22] and WS-CAF [25]. They provide the infrastructure to support complex long-running business applications based on advanced transactional models [10].

Furthermore, there has also been a growing demand for attaining higher levels of availability in business applications. Companies depend more and more on their IT infrastructures what results in an increasing need for more reliable and available middleware platforms. This need is becoming more acute with the rapid expansion of SLAs among e-business partners. Server outages can violate the SLA and result in substantial financial losses. Many other applications, such as workflows, online analytical processing (OLAP) and scientific applications, run long transactions and computations that also keep the state across transactions. In these applications, high availability support for transactions and long running activities is very important since the amount of computation that can be lost due to a crash is very significant. Moreover, since many of these applications keep state across transactions and activities, they require transparent failure masking. This means that simply aborting the current transaction in case of a crash to maintain a consistent state is not enough. In many cases, the application will be unable to restart the processing due to the state kept internally across transactions [5], like, for instance, variables in the client application code or interactions with third parties. To attain high availability, replication is used. However, current middleware support usually fails to satisfy industry expectations for high availability mainly due to two reasons: 1) It does not provide highly available transactions. In the advent of a replica failure, current solutions abort all ongoing transactions. In many applications, like workflows or orchestrated web services, there is no way to replay the aborted transactions. That is, current solutions do not solve the availability problem. This fact is exacerbated in the case of long running activities, mostly ignored by current middleware platforms; 2) Focuses on the replication of a single tier with the subsequent loss of availability when the non-replicated tier crashes.

In this paper, we present a set of replication algorithms following a primary backup approach that tackle these two issues in the context of J2EE application servers. A first

*This work has been partially funded by the Spanish Research Council (MEC, TIN-2004-07474-C02-01), the Madrid Regional Research Council (CAM, P-TIC-285-0505) and the European EUREKA/ITEA S4All project under MITyC grant FIT-340000-2005-144.

contribution consists in combining the replication of the application server (AS) and database tiers. This makes our platform resilient to any single point of failure. What it is more, it also guarantees the consistency between the two tiers in any failure scenario. The combination of the replication of both tiers is especially novel in that replication is not made along tiers (i.e. replicating the AS tier independently of the database), but across them (i.e. replicating pairs of AS and database). The former approach is far from being trivial and requires some sophisticated logic to enable the consistent integration of both replicated tiers [20]. The latter approach, the one taken in this paper, enables an integral replication solution fully achieved at the AS tier and without modifying the database. This fact is important for pragmatic reasons since it enables the use of the replication platform with any existing database and without requiring access to the database code. A second contribution lies in that our solution supports highly available transactions and long running activities. Replica failures are transparently masked in front of clients. That is, running transactions and activities are not aborted. Upon failover, processing continues at the point the primary was when it failed. All previous solutions (that did not require specific hardware such as Tandem systems), to the best of our knowledge, abort ongoing transactions failing to satisfy the high-availability requirement. The provision of high availability in this context raises many challenges. Unlike previous work, it has to deal with services used by the application (e.g. the transaction manager, the activity service engine, etc.), in addition to business components.

We have implemented and evaluated a prototype integrated into the JBoss AS [18]. We have also implemented a prototype of the J2EE Activity Service for long running activities. The performance has been evaluated both with the ECPerf benchmark and a custom benchmark for long running activities.

The paper is structured as follows. Section 2 introduces J2EE. Section 3 defines the system model. Section 4 presents our replication algorithms. They are evaluated in Section 5. Section 6 presents related work. Finally, Section 7 concludes the paper.

2 J2EE

J2EE [32] is an extensible framework that provides a distributed component model along with other useful services such as persistence and transactions. J2EE components are called *Enterprise Java Beans* (EJBs). There are three types of EJBs: *session beans* (SBs), *entity beans* (EBs) and *message driven beans* (MDB). We will not consider MDBs in this paper. SBs represent the business logic and their lifetime is bounded to the lifetime of clients (i.e., they are volatile). SBs are further classified as stateless (SSBs) and stateful (SFSBs). SSBs do not keep any state across method invocations. SFSBs may keep state across invocations of a client (conversational state). EBs model business data and are stored in some persistent storage, usually a database.

The transaction manager is the service that handles transactions in J2EE. J2EE provides the *Java Transaction API* (JTA) to demarcate transactions. The transactional attributes of an EJB indicate whether its methods must run in a transaction, if a transaction must be already running or if a new transaction must be initiated.

Traditional transactions may be too restrictive for applications that need to relax some of the ACID properties. For instance, workflows may last for long periods of time (days, months). In this context, keeping locks across invocations drastically reduces the system concurrency. For this reason, several advanced transaction models have been proposed [10]. The *J2EE Activity Service* specification (J2EEAS) allows the implementation of advanced transaction models in J2EE.

The J2EEAS consists of two components: the *activity service* itself and one or more *high level services* (HLSs) (Fig.1). The activity service provides an abstract unit of work called *activity*, that may or may not be transactional. An activity may encapsulate a transaction or be encapsulated by a transaction. Activities may be nested.

HLSs are defined on top of the activity service and represent advanced transaction models. Applications use a HLS to demarcate activities, which produce an *outcome*. In order to implement a transaction model using the J2EEAS developers must provide the demarcation points (signals), the outcomes and the state transitions (signalset).

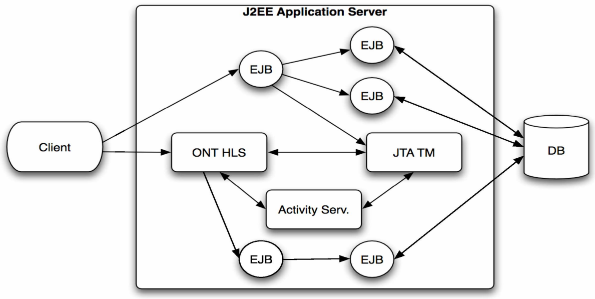

Figure 1. J2EE services and components

We have implemented the open nested transactions model (ONT) [34] as defined in [33]. ONTs are similar to traditional nested transactions (as specified in J2EE). However ONTs relax the isolation property of nested transactions in order to be used in the context of long running transactions. In this way, a workflow can be seen as an atomic unit of work that is divided into smaller activities which are ACID transactions. When a subtransaction commits, all the resources are released. Due to the relaxation of isolation, if the global activity fails, atomicity is achieved by logically undoing (compensating) committed activities. In order to compensate a committed transaction, when that transaction commits, it registers a compensator with its parent. This process is done recursively. When the top-level ONT commits, the registered compensators are discarded. If the top-level ONT aborts, compensators are applied in reverse order of completion. Figure 1 shows the possible interactions among a client, EJBs, the transaction manager (TM), the ONT HLS and the activity service.

3 Model

In our model, a *replica* is the pair AS and database. That is, ASs do not share the database. The set of all replicas is called a *cluster* (Fig.2). We will consider that a replica fails if either the database or the AS fails.

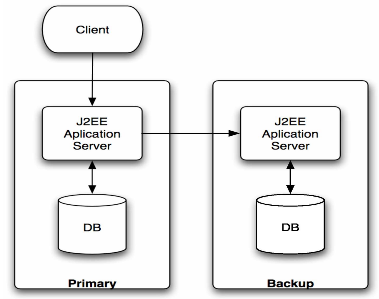

Figure 2. Replication model

ASs communicate using a group communication system (GCS) supporting strong virtual synchrony [14]. GCSs provide *reliable multicast* and *group membership* [8]. Group membership services provide the notion of view (currently connected and active group members). Changes in the composition of a view (*member crash* or *new members*) are eventually delivered to the application. We assume a primary component membership [9]. In a primary component membership, views installed by all members are totally ordered (there are no concurrent views), and for every pair of consecutive views there is at least one member that survives from the one view to the next one. Strong virtual synchrony ensures that messages are delivered in the same view they were sent (sending view delivery) and that two members transiting to a new view have delivered the same set of messages in the previous view (virtual synchrony). Group communication primitives can be classified attending to the order guarantees and fault-tolerance provided. *FIFO ordering* delivers all messages sent by a group member in FIFO order. With regard to reliability, *reliable multicast* ensures that all available members deliver the same messages. *Uniform reliable multicast* ensures that a message that is delivered by a member (even if it fails), will be delivered at all available members. We assume that multicast messages are delivered to the sender of the message (self delivery). In the algorithms presented in this paper, ASs communicate using uniform FIFO multicast.

4 J2EE Replication

In this section we present a suite of replication algorithms for high available transactions (HA replication). The proposed algorithms follow a primary-backup replication scheme and take care of both the state of the AS and the database. That is, there is one replica (primary) that processes client requests and sends the state changes (checkpoint) to the rest of the replicas (backups). Backups just apply the changes the primary sends. Clients invoke EJBs residing on the primary and EJBs may invoke other EJBs. If the primary fails (either the AS or the database), a backup will take over and become the new primary. Clients will now interact with the new primary.

Although we target long running transactions, first we present a replication algorithm for transactions whose lifetime is a single client invocation to simplify the presentation. Then we will extend this algorithm with client-demarcated transactions that may invoke several times the AS within a single transaction, and finally, we will see how to replicate long running activities based on ONTs. The HA replication algorithms provide the following consistency properties in the absence of catastrophic failures (all replicas fail): *1) Exactly once execution*. Every request is processed exactly once despite replica failures. *2) Replica state consistency*. After a client receives a reply, it is guaranteed that all running replicas have the same state. That is, the backups contain the same set of checkpoints and if the failover is performed, the backups will reach to the same state that the primary had (the same SFSBs with the same state, the same committed EB updates, the same database committed state, the same uncompleted transactions and uncompleted activities with the same associated updates and reads, and the same activity trees). *3) Highly available processing*. Every client request eventually receives its outcome despite replica failures. That is, from the client perspective, transactions and activities never abort due to replica failures.

It should be highlighted that the first and second properties are provided by previous work, but only in the context of a single replicated tier (the application server tier). Our proposed algorithms fulfill these two properties replicating both tiers and therefore, without exhibiting any single point of failure. More importantly, our approach is the only one that fulfils the third condition to the best of our knowledge.

4.1 One Request Transactions

In this algorithm we only consider transactions that start and complete during a client invocation to an EJB. This EJB may invoke other EJBs. The transaction will commit or abort before the client invocation finishes.

The client as part of the algorithm sends in each invocation a *request identifier* (*rid*). Rid consists of the client identifier and a request number. There is a request number per client that is increased each time that client invokes an EJB. This is done transparently by the client stub, so the client application code is not changed.

When the primary receives an EJB invocation from a client, it starts a transaction and executes that request. This request may call other EJBs, resulting in a nested invocation. The primary collects all the changes done to the invoked EJBs (checkpoint) and the associated rid in a table (*changes table*) whose key is the client id. Request changes also include the creation and deletion of EJBs. Before returning the results to the client, the primary commits the transaction and FIFO multicasts to the backups the changes and the results of the request. Therefore, at most one message is sent to the backups per client invocation. If no EJB is modified, the primary does not send any message. The primary returns the result to the client after it delivers the multicast message with the EJB changes, if any. Uniform multicast guarantees that if the primary has delivered that message, the backups will also deliver it. The primary commits transactions sequentially and FIFO multicasts the changes in the commit order to the backups. This guarantees that backups will commit transactions in the same order as the primary. If the transaction commits, the backups will receive the committed EJBs changes and apply them. If the transaction aborts, no message corresponding to that transaction is sent to the backups. Backups process primary messages (applying the changes) in the delivery order (FIFO). Backups store the request identifier and the results in a table (*results table*). A backup keeps the results of a request message till it receives a new request from the same client.

Failures. If an AS fails, the GCS informs all the ASs about the failure delivering a new view. If a backup has failed (it does not appear in the new view), nothing is done. If the primary fails, one of the backups will be chosen deterministically to become the new primary (e.g. the replica with the smallest identifier in the view). Since strong virtual synchrony is assumed and messages are uniform multicast, all the backups that belong to the new view have delivered the same set of messages the old primary delivered when the new view is delivered. So, all of them will have the same EJB changes and anyone can take over as the new primary. If a database fails, the AS connected to it will detect the failure and shut down itself. Therefore, it will also fail. The new primary must apply the changes received from the old primary before starting to process client requests in order to ensure it has the same state the old primary had when it failed. If those changes are not applied by the time the view change is delivered, the processing of new client requests will be delayed until all outstanding changes are applied.

In order to be able to fail over, the client stub needs to know the available ASs. The client stub receives the view of the cluster from the primary. Each view has a unique identifier (*view id*). The client stub attaches the view id to each request. If the primary detects that the client view id is not the id of the current view, it will piggyback the new view to the reply. So, if the primary fails, a client invocation will not succeed and the client stub will try to contact another AS from the last view received. That replica may or may not be the new primary. If the chosen AS is the new primary, it will process the request; otherwise it will reply with a message indicating the new primary.

If the primary fails while processing a request, the client stub will receive an exception and resubmit that request to another server. Due to the use of uniform multicast, if the primary sent the changes before failing and delivered that message, the rest of the replicas also delivered the message and have the request changes and results. If the primary did not deliver that message, uniformity guarantees that the rest of the replicas will either all deliver the message or none. In the former case, the new primary has delivered the changes produced by that request. When the client resubmits the request, the new primary will access the results table using the rid, detect that the request is a duplicate (the rid is in the results table) and send the cached reply to the client. In the latter case, none of the backups is aware of that request because the primary failed before the corresponding changes were delivered. So, the new primary will access the results table and not find that rid, and therefore will execute the request. Otherwise, all of them are aware of it and then the behaviour is as in the former case.

Note that the primary does not send any message to the backups if a request does not update any SFSB or EB ("read-only" request since SSB changes are not stored). Therefore, when a client resubmits a "read-only" request, the new primary is not aware of that request (the rid is not stored in the results table). The new primary will execute the request and return the results to the client.

4.2 Client-Demarcated Transactions

Clients may demarcate transactions in J2EE using the Java Transaction API (JTA). A client-demarcated transaction may bracket several invocations to EJBs. Transactions are started, committed or aborted by the server transaction manager on behalf of the client. When a client begins a transaction, the primary will create a transaction and generate a transaction identifier (*tid*). The primary stores this tid and the corresponding client identifier in a *transaction table* to associate that client with that transaction. Then, it multicasts this information (*begin message*) to the backups and suspends the transaction before returning to the client. The client attaches the tid and a request number to each EJB invocation from that transaction. When the primary receives a request within that transaction, it accesses the transaction table to resume the associated transaction and then processes the request. The primary processes requests as in the basic algorithm (the primary stores the EJB changes, multicasts them together with the request results) but, before returning to the client it suspends the associated transaction. This transaction suspension just deassociates the thread from the transactional context (this is in fact done by applications servers even if they are not replicated).

It has to be noted that now the primary sends committed and uncommitted changes to the backups. The state sent consists of the SFSBs and EBs modified in the current invocation. The state of SFSBs is not transactional. Even if a SFSB method runs within a transaction, if the transaction aborts that state is not undone to the previous state. Therefore, we will consider those changes as committed changes. When the client calls the commit (abort) operation, the primary will resume the associated transaction, invoke the commit (abort) operation, and multicast a commit (abort) message with the tid to the backups. If SFSBs implement the *afterCompletion* method, the container will execute that method after committing (aborting) the transaction. That method may change the state of the EJB. In this case, the primary will also send those changes in the commit (abort) message.

An EJB method may start a new independent transaction when it is invoked. In this case, the enclosing (client) transaction is suspended, and a new inner transaction is started. That is, in J2EE it is possible to run several transactions on behalf of a single client invocation, by suspending/resuming transactions. This implies that a message to a backup can carry changes from several transactions.

When each client request corresponds exactly to a single transaction, backups only need to apply committed changes. They do not need to know which reads were performed in the database. However, when transactions span multiple client requests, this is not the case anymore. Reads performed by uncommitted transactions at the primary are important to guarantee consistency if the primary fails, since they affect the serialization of transactions (i.e. by setting read locks). For instance, if there are two transactions T1 and T2, T1 has read object a and T2 has begun. Then, the primary fails. T1 and T2 are recreated on the new primary. Now, T2 modifies object a and commits. T1 performs other operations and reads a again. The value of a is different to the value read previously, though, both reads are executed within the same transaction (non-repeatable reads). If the primary would not have failed, T2 would block when trying to modify a till T1 finishes. In order to implement the same semantics for failure and failure free scenarios, the primary in addition to the changes also sends information about database reads. Since results of reads can be very bulky, the primary sends the SQL read statement submitted by the container to the database.

When a backup receives a begin transaction message, it just stores the information received in the transaction table. Upon a backup receives changes from the primary, it applies all committed changes and stores the request results in the results table. A backup will delay the application of uncommitted changes till it processes the commit message. When a backup processes the commit message, it applies the uncommitted changes for that transaction, discards the reads associated to the transaction, and stores the result of the transaction (commit) in the result table. If a backup receives an abort message, it discards uncommitted changes and reads.

Failures. If the primary fails before a client transaction completes, the new primary will recreate that transaction. The new primary will not process any client request till it has applied all the messages sent by the old primary. When this happens, the primary checks if there are uncompleted transactions in the *transaction table*. If this is the case, the new primary will recreate a transaction for each of these transactions, apply uncommitted changes and execute the associated reads in the order they were sent by the old primary. This guarantees that each recreated transaction will hold the same state it held at the old primary, therefore guaranteeing consistency of recreated transactions.

If the primary fails between two client invocations, the client just needs to contact the new primary. If the primary fails while running an invocation, the client stub will resubmit the invocation behaving as in the previous algorithm. If the primary fails when the client called the begin operation, there are two cases. If the transaction is in the transaction table, the new primary will send back the tid. Otherwise, the new primary did not receive the begin message in the previous view and will process it now. Finally, if the commit (abort) operation failed, the new primary might have the result and return it to the client, or not have it, in which case it will execute the operation. Replicating the uncommitted state on each client invocation avoids the abortion of the client transaction in case the primary fails. The new primary resumes the transaction transparently providing highly available transaction support.

4.3 Open Nested Transactions

Like client-demarcated transactions, ONT activities are started, completed and compensated on the server and may involve multiple server invocations. ONT activities are demarcated either by the client or by SBs. This can yield to arbitrarily complex ONT activity trees as it happens with nested transactions in JTA. The main difference from the replication point of view is that whenever a child ONT activity succeeds, a compensator is registered with its parent. This procedure is performed recursively until the top-level ONT activity is reached. When the top activity commits, registered compensators are discarded. Whenever an ONT activity aborts, the registered compensators are executed. Therefore, in order to replicate ONT activities and provide transparent failover the primary must send the corresponding ONT activity tree state with all this information to the backups.

When a client starts an ONT activity, the primary generates an *activity identifier* (*aid*) and stores it with a client identifier in an *activity table*. The primary multicasts an *activity begin* message to the backups with that information, suspends the ONT activity and returns the *aid* to the client. When a client submits a request, it attaches the *aid* and a request identifier. The primary resumes the associated ONT activity and executes the request. That request may start a nested ONT activity. The primary will register that ONT activity in the ONT activity table as a child ONT activity of the client ONT activity (*ONT activity tree*). If the nested ONT activity succeeds, a compensator is registered with the parent ONT activity to be invoked in case the parent ONT activity fails. The primary will multicast the request identifier, the request result, the changes of the completed nested ONT activity as committed changes, the uncompleted parent ONT activity changes as uncommitted ones, read statements, the ONT activity tree with the status of each ONT activity in the tree and the registered compensators before returning to the client. In the case under consideration, the child activity has committed and therefore, the ONT activity tree has a single ONT activity (the parent). When the client submits a *commit* operation, the primary resumes and commits the associated ONT activity, multicasts the ONT activity outcome, and returns to the client. Now the *aid* is removed from the ONT activity table. If the parent ONT activity fails, the primary executes the compensators and afterwards, it collects all the EJB changes. Then, it multicasts the changes and the ONT activity outcome. After delivering this message, it returns the outcome to the client.

Backups store the *aid* and the client id when they receive a begin message. When a backup receives a message with changes, it updates the ONT activity tree according to the one in the message and proceeds as in the previous algorithm. Uncommitted changes will be applied when a commit message is received, and discarded if the message is abort. In that case, the compensators are executed.

Failures. When the new primary finishes processing messages from the previous primary, it reconstructs all unfinished ONT activities. For each ONT activity in the table, it traverses top-down the corresponding ONT activity tree. The new primary creates an (possibly nested) ONT activity and associates the registered compensators, if any, for each node in the ONT tree. Then, it applies uncommitted changes, and executes the reads performed on behalf of the ONT. Now, the primary can resume request processing.

5 Evaluation

We have implemented the replication algorithms in the JBoss AS. Our implementation is based on the ADAPT framework [2]. It supports the prototyping of replication protocols in JBoss using interceptors. The ADAPT framework provides two types of interceptors: *client component monitor* (CCM) and *component monitor* (CM). The client component monitor intercepts all the outgoing invocations at the client side. It implements the client side of the replication algorithms. The CCM is dynamically loaded by the client when getting the stub of a remote EJB. There is one CCM instance per client. The CM is the server side counterpart responsible for intercepting both, remote invocations from the client to the EJBs, and local invocations between EJBs. It implements the server side of the replication algorithm.

We have evaluated the overhead of the replication algorithms both in a traditional transactional application and in an application based on the ONTs model. We used a non-replicated JBoss and JBoss clustering as baselines for the experiments. Although, none of them provides the availability and fault-tolerance guarantees our algorithms do. In JBoss clustering there is a single shared database. The replication algorithm of JBoss is not transaction aware and transfers the state of an SFSB at the end of each invocation that modifies the SFSB. JBoss clustering is configured to execute as a primary-backup. We have also measured the time needed for failover when the primary fails.

The experiments were run in a cluster of 2 GHz AMD dual-processor PCs with 512 MB of RAM running Red Hat Linux 9.0. We used JBoss 3.2.3 AS [18], PostgreSQL 7.3.2 database, JGroups [1] as a GCS and JASS [26], our implementation the activity service. In all experiments the clients, each instance of JBoss and the database were run in separate hosts. All the results of the experiments have been obtained over the steady phase of the test.

5.1 Transaction Replication

The evaluation of the replication algorithm has been done using ECperf [31]. It simulates a supply chain, defining four application domains: corporative, order entry, supply chain management and manufacturing. ECperf measures the throughput in *BBops* per minute (*benchmark business operations*). BBops are the sum of the number of transactions of the order entry application and the number of work orders the manufacturing application generates.

The main parameter in the tests is the *injection rate (Ir)*, which models the load injected to the system. The number of clients is five times the Ir in the order entry application, and three times the Ir in the manufacturing application. The ECPerf target determines for a given load, the conditions that the performance metrics of the system should fulfil. It specifies a maximum response time for all the requests (2 seconds for the order entry domain and 5 seconds for the manufacturer domain) and also that the response time corresponding to the percentile 90% is no more than a 10% of the average.

Figure 3 shows the overall average response time for the order entry transactions (the ones with stricter response time) under ECPerf. As expected, non-replicated JBoss offers the lowest response time, since it does not incur any overhead due to replication. Till 10 Ir the response time of our algorithm (HA Replication) is similar to the one of JBoss clustering (JBoss Primary-Backup) and the non-replicated JBoss. From 10 to 20 Ir the overhead of replication has a noticeable impact on the response time of our algorithm becoming higher than the one of JBoss primary-backup. However, the response time is still within the limits admitted by ECPerf for order entry transactions (2 seconds). This means that for moderate loads the overhead of our replication algorithm is negligible and only for high loads it results in an increased, although still reasonable, response time.

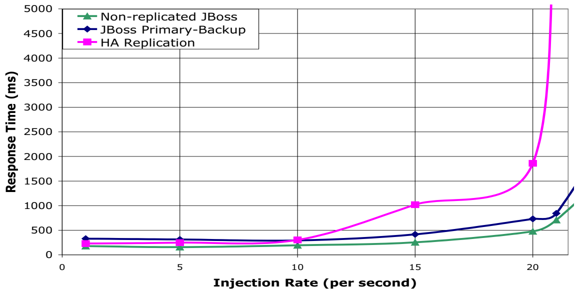

Figure 3. ECperf results: Response time

Figure 4 shows the maximum throughput with respect to the ECPerf target. We only show the dots corresponding to experiments that fulfilled the ECPerf target. All configurations fulfilled the ECPerf target till 20 Ir. HA Replication saturates for higher Irs. JBoss Primary-Backup and the non-replicated JBoss accomplished the target for a load of 21 Ir. This means that with respect the ECPerf target, our replication algorithm has a loss of throughput of 5%. It is also worth to notice that the non-replicated JBoss degrades more gracefully under saturation. In particular for an Ir of 21, our replication algorithm was far from reaching the ECPerf target, whilst JBoss Primary-Backup was not that far for a load of 22 Ir. This means that the results in terms of the ECPerf target are very promising for the replication algorithm proposed. The additional cost of HA Replication is also natural since it is transaction aware and takes into account both the state of the database (entity beans) and the application server (stateful session beans).

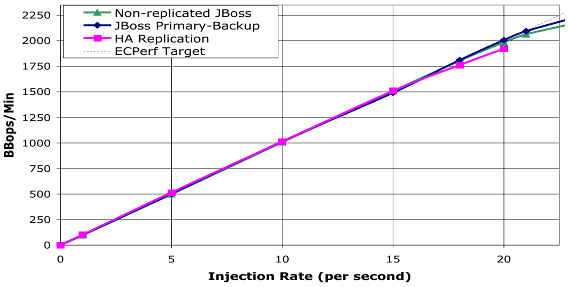

Figure 4. ECperf results: Throughput

Table 1 shows the number of messages sent from the primary to the backup and their average size for the order entry domain and 10 Ir. JBoss Primary-Backup and HA Replication send a similar number of messages for the order entry, though, JBoss clustering does not replicate EBs. The number of messages sent by our replication algorithm increases when we take into account the manufacturing domain (Table 1). It sends about three times more messages than JBoss primary-backup. In both application domains the messages are also three times larger than the ones of JBoss. This extra overhead is reasonable taking into account that JBoss only replicates the state of SFSBs that are scarcely used in ECPerf (only the shopping cart is an SFSB) and does not replicate the database (EBs). This means that when a stateless session bean is invoked and only accesses EBs, JBoss replication does not send any message, whilst our algorithm sends a message with the changed EBs.

Order entry domain (Ir=10)		
System	**Number of Msgs.**	**Avg. size(bytes)**
JBoss Primary-Backup	5169	797
HA Replication	5261	2073
Order entry and manufacturing domains (Ir=10)		
System	**Number of Msgs.**	**Avg. size(bytes)**
JBoss Primary-Backup	5406	786
HA Replication	17183	2095

Table 1. Number and size of messages

5.2 Advanced Transactions

The goal of this experiment is to evaluate the overhead of our replication algorithm for applications with long running activities. We have implemented the J2EE Activity Service specification and the ONTs and integrated them into JBoss. We are not aware of any benchmark to evaluate the J2EEAS itself or any advanced transaction model. So, we have built a custom benchmark that consists of a shopping cart application derived from the one in the ECperf order entry application domain. We have reused all the ECPerf infrastructure to inject the load and evaluate whether the target was fulfilled. That is, this benchmark has same level of strictness as ECPerf. The client for the shopping cart application is also an adaptation of the ECperf entry order application in order to generate client requests as ECperf does. The injection rate (*Ir*) is the parameter used for determining the experiment load.

The benchmark works as follows. Every client adds between five and fifteen items to the cart. The first time a client adds an item, an ONT representing the cart is created (*Top-level TX* in Fig.5). Each time that a client adds an item to the cart, a child transaction of the top-level ONT is started (*Middle TX* in Fig.5). Within this transaction, the application updates the client credit and stock of the item accordingly. Each of these two actions is a nested ONT transaction (*Leaf TX* in Fig.5). When any of these ONTs commits, a compensating action is registered in the parent transaction (*Middle TX*). The compensating action for the item increases the quantity of the item in the stock. The one for the customer increases the customer credit. If the customer has enough credit and the selected item is available in the stock, the item is added to the cart, and the middle ONT registers the compensators with the top-level ONT. On the other hand, if one of the checks fails, the middle transaction aborts and the registered compensators are executed. Finally, the client decides whether to buy the contents of the cart (the top-level ONT commits or aborts). If the top-level ONT commits the compensators are forgotten. Otherwise, the compensators are executed by the top-level ONT undoing the changes previously made.

We have run the benchmark with the replication algorithm (HA Replication) and with JBoss extended with the J2EE activity service. The latter uses three sites, one for JBoss with J2EEAS, one for the database and another one for clients. HA Replication uses five sites: one for the clients, two sites for each of the two JBoss with the J2EEAS, and two sites for each of the two databases. The JBoss clustering configuration was not used this time, since JBoss clustering does not support the replication of J2EEAS.

The throughput of both the non-replicated JBoss and our replication algorithm are very close (Fig.6). Both are able to reach the target till an Ir of 7. The throughput beyond 7 Ir keeps similar till Ir=10. At that point, our replication algorithm degrades very fast, whilst the non-replicated JBoss exhibits a more or less flat curve before falling. The reason for the faster degradation of our replication algorithm is the incurred overhead for replicating EBs, that although is moderate, once the server saturates it becomes noticeable. Table 2 shows the CPU usage. For high loads, HA Replication is almost saturated with an usage of 99%. In summary, our replication algorithm shows a good behaviour for long client interactions. The overhead of replicating ONTs is not perceptible in terms of throughput. Since the main motivation of our algorithm precisely targets long transactions, the attained results are very promising.

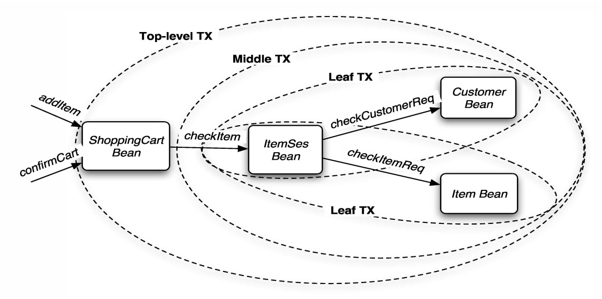

Figure 5. J2EEAS benchmark

Configuration	CPU Utilization Ir=3	CPU Utilization Ir=8
JBoss Non-Replicated	27%	60%
HA Replication	42%	99%

Table 2. CPU usage in J2EEAS benchmark

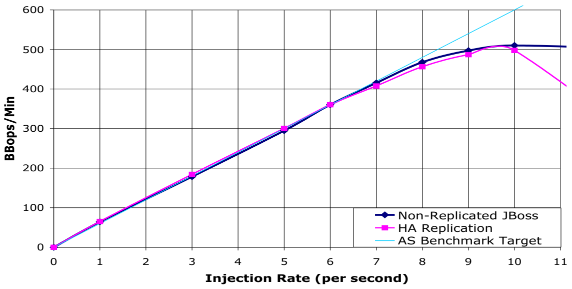

Figure 6. J2EEAS results: Throughput

5.3 Failover

Here, we measure the time needed to complete the failover starting from the instant at which the new primary detects the failure of the previous primary. Since the protocol is based on group communication, the failure detection of the primary is instrumented by means of view changes. When the primary fails, the underlying group communication infrastructure delivers a view with the new membership. The backup that appears first in the view takes the role of new primary and the failover is triggered.

The duration of the failover mainly consists of two components: 1) The time needed to recreate the transaction (and activity tree); 2) the number of updated entity beans (EBs) that have not been yet committed. Upon failover, the transaction (and activity tree) is recreated and updates corresponding to uncommitted transactions are replayed at the new primary. The number of uncommitted transactions depends on the number of concurrent clients. The experiment measures the failover time for an increasing number of concurrent clients and of uncommitted EBs per client (1, 5, and 10 EBs per client).

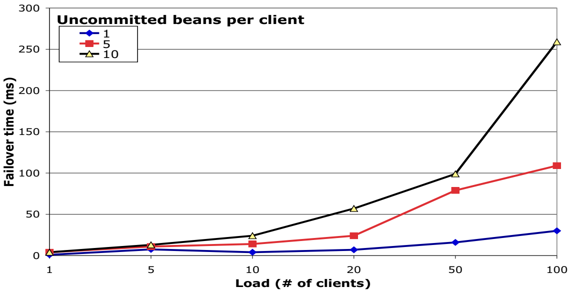

Figure 7. Failover time

Figure 7 shows that even for large loads (100 clients) and the highest number of uncommitted EBs per client (10 EBs), the time taken by the failover is quite affordable (250 ms). In this worst case scenario the new primary has to recreate one hundred transactions (one per client) and apply the changes of 1,000 uncommitted EBs during failover. The failover time is almost independent of the number of EBs for small loads (till ten clients). When the number of clients increases, the number of EBs becomes the dominant factor of the failover time. We can conclude that the failover time is quite reasonable even for high loads. As far as the failover time is smaller than the time failure detection takes, the failover overhead is acceptable. Failure detection time is typically around 1 second to prevent false failure suspicions when the system is overloaded, so failover shows good performance.

6 Related Work

Most of J2EE ASs provide some replication facilities (SSBs for load balancing and SFSBs). JBoss open source AS [18] provides replication for SSBs and SFSBs [19]. The state of SFSBs is multicast to the rest of the replicas after each method invocation. The database is always shared among all JBoss replicas (becoming a single point of failure). Another popular open source J2EE AS, JOnAS [27], only replicates SSBs. Oracle9iAS [30] and WebLogic [7] replicate SFSBs at the end of every method invocation. WebLogic clustering has cluster-aware stubs for EJBs. So, if the primary fails, the stub retries on another replica. It is transaction aware, although it does not provide highly available transactions. If an EJB is running within a transaction, replication occurs after the transaction commits. That is, if there is a failure, ongoing uncommitted transactions cannot be resumed at a backup. They abort and must start from scratch. A similar behavior is found in the Data Replication Service (DRS), which is in charge of handling replication in WebSphere 6.0 [17]. WebSphere also implements the Activity Service. Activities are treated as transactions. Beans accessed within an activity are not replicated until the activity completes.

Replication for CORBA has received a lot of attention [24, 12, 23, 11, 3, 28]. However, none of these approaches addressed the integration of replication and transactions, although it was recognized as a challenging problem [13]. [36] presents a transaction-aware implementation of FT-CORBA. Papers like [15] defines formally exactly-once correctness in multi-tier systems. They study the replication of stateful and stateless ASs with a shared database. Each client request is executed as a single transaction. For each transaction a "marker" is inserted in a shared database. The new primary will look for this marker during failover in order to ensure exactly once execution of each client request. In this case, the database is a single point of failure (there is a single shared instance). There is no transaction high availability, i.e., when a server crashes all the ongoing transactions are aborted. [35] applies this technique for transactions spanning a single client request in JBoss. Our replication protocol handles transactions that span across several client requests (very common in J2EE).

The integration of replication in J2EE application servers and a replicated database has been studied in [20]. This paper shows that careful engineering is needed to integrate the replication protocols of two different systems.

Phoenix [6, 4] provides primary/backup replication for .NET. Component state is replicated periodically and requests between checkpoints are logged. If the primary crashes, the backup recovers the last checkpoint and applies the logged requests since the checkpoint. In [5] the authors propose ODBC support for highly available transactions in the presence of short-lived database server failures. They target, as we do, an scenario with long running transactions (such as OLAP) where the amount of lost work may be very high. Their approach consists in making the database connection state persistent, so it can be recovered upon a crash of the database server. Therefore, database crashes are masked transparently to client applications.

The use of the CORBA Activity Service [29] to implement advanced transaction models was studied in [16]. The J2EE Activity Service [33] is based on this specification. Currently, several efforts for supporting long running activities are emerging in the web services arena, such as WS-Coordination/WS-Transaction [22], and WS-CAF [25, 21].

7 Conclusions

We have presented a novel approach for replication in J2EE ASs. This approach brings several innovations. The first one is to make the J2EE replication transaction and activity aware. Another novelty, and possibly the most important one, is that it provides highly available support for transactional applications and long running activities that might span multiple client interactions with the AS. This is important in many applications (such as workflows, OLAP, . . .) in which part of their state and part of the processing is delegated to a J2EE AS in presence of replica failures. For these applications, retrying is either very complex or not possible at all. With the presented solution, running transactions and long running activities continue their processing despite replica failures transparently to client applications. Finally, the protocols provide consistent replication of both the AS and database tier. Previous approaches only replicated the AS tier and forced to use a shared database that became a single point of failure.

The results of the evaluated prototype have shown that the overhead is in many cases negligible, especially in the targeted scenario of long running transactions and activities.

References

[1] B. Ban. *JGroups*. http://www.jgroups.org.

[2] O. Babaoglu, A. Bartoli, V. Maverick, S. Patarin, J. Vuckovic, and H. Wu. A Framework for Prototyping J2EE Replication Algorithms. In *DOA*, 2004.

[3] R. Baldoni and C. Marchetti. Three-tier Replication for FT-CORBA Infrastructures. *Software: Practice and Experience*, 33(8):767–797, 2003.

[4] R. S. Barga, S. Chen, and D. Lomet. Improving Logging and Recovery Performance in Phoenix/App. In *ICDE*, 2004.

[5] R. S. Barga, D. Lomet, T. Baby, and S. Agrawal. Persistent Client-Server Database Sessions. In *EDBT*, 2000.

[6] R. S. Barga, D. Lomet, and G. Weikum. Recovery Guarantees for General Multi-Tier Applications. In *ICDE*, 2002.

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

[8] K. Birman. *Building Secure and Reliable Network Applications*. Prentice Hall, NJ, 1996.

[9] G. V. Chockler, I. Keidar, and R. Vitenberg. Group Communication Specifications: A Comprehensive Study. *ACM Computer Surveys*, 33(4), 2001.

[10] A. K. Elmagarmid, editor. *Database Transaction Models*. Morgan Kaufmann, 1992.

[11] J. Fabre and T. Perennou. A Metaobject Architecture for Fault-Tolerant Distributed Systems: the FRIENDS Approach. *IEEE Transactions on Computers*, 47:78–95, 1998.

[12] P. Felber, R. Guerraoui, and A. Schiper. The Implementation of a CORBA Object Group Service. *Theory and Practice of Object Systems*, 4(2):93–105, 1998.

[13] P. Felber and P. Narasimhan. Reconciling Replication and Transactions for the End-to-End Reliability of CORBA Applications. In *DOA*, 2002.

[14] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical Report TR95-1537, CS Dep., Cornell Univ., 1995.

[15] S. Frølund and R. Guerraoui. e-Transactions: End-to-End Reliability for Three-Tier Architectures. *IEEE Trans. Software Engineering*, 28(4):378–395, 2002.

[16] I. Houston, M. C. Little, I. Robinson, S. K. Shrivastava, and S. M. Wheater. The CORBA Activity Service Framework for Supporting Extended Transactions. *SPE*, 33(4), 2003.

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

[18] JBoss Group. *JBoss App. Server*. http://www.jboss.org.

[19] JBoss Group. *JBoss Clustering*, 2002.

[20] B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, and J. Salas. Exactly Once Interaction in a Multi-tier Architecture. In *Proc. of VLDB Workshop on Design, Implementation and Deployment of Database Replication*, 2005.

[21] M. Little. Models for Web Services Transactions. In *SIGMOD Conf.*, page 872, 2004.

[22] Microsoft, IBM and BEA. *WS-Coordination/WS-Transaction Specification*, 2005.

[23] G. Morgan, S. Shrivastava, P. Ezhilchelvan, and M. Little. Design and Implementation of a CORBA Fault-tolerant Object Group Service. In *DAIS*, 1999.

[24] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Eternal - A Component-based Framework for Transparent Fault-tolerant CORBA. *SPE*, 32(8):771–788, 2002.

[25] OASIS. *Web Services Composite Application Framework (WS-CAF)*, 2005.

[26] Objectweb. *JASS*. http://jass.objectweb.org.

[27] Objectweb. *JOnAS App. Server*. http://jonas.objectweb.org.

[28] OMG. *Fault Tolerant CORBA*. OMG, 2000.

[29] OMG. *Additional Structuring Mechanisms for the OTS Specification 1.0*. September 2002.

[30] Oracle Corp. *Oracle9iAS Containers for J2EE. EJBs Developer's Guide, Rel. 2 (9.0.4)*, 2003.

[31] Sun Microsystems. *ECperf spec. v1.1 final release*, 2003.

[32] Sun Microsystems. *J2EE spec. v1.4*, 2003.

[33] Sun Microsystems. *JSR 95: J2EE Activity Service for Extended Transactions*, Mar. 2004.

[34] G. Weikum and H. J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In *Database Transaction Models*, chapter 13. MKP, 1992.

[35] H. Wu, B. Kemme, and V. Maverick. Eager Replication for Stateful J2EE Servers. In *DOA*, pages 1376–1394, 2004.

[36] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Unification of Replication and Transaction Processing in Three-Tier Architectures. In *ICDCS*, pages 290–300, 2002.