# Consistent and Scalable Cache Replication for Multi-Tier J2EE Applications*

F. Perez-Sorrosal[1], M. Patiño-Martinez[1], R. Jimenez-Peris[1], and Bettina Kemme[2]

[1] Facultad de Informática, Universidad Politécnica de Madrid (UPM), Spain
{fpsorrosal,mpatino,rjimenez}@fi.upm.es
[2] McGill University, Quebec, Canada
kemme@cs.mcgill.ca

**Abstract.** Data centers are the most critical infrastructures of companies and they are demanding higher and higher levels of quality of service (QoS) e.g., availability, scalability... At the core of data centers we find multi-tier architectures providing service to applications. Current infrastructure for multi-tier systems has focused exclusively in providing high availability using replication. Most approaches replicate a single tier, becoming the non-replicated tier a bottleneck and single point of failure. In this paper, we present a novel approach that provides availability and scalability for multi-tier applications. The approach lies in a replicated cache that takes into account both the application server tier (middle-tier) and the database (back-end). The underlying replicated cache protocol fully embeds the replication logic in the application server. The protocol exhibits good scalability as shown by our evaluation based on the new industrial benchmark for J2EE multi-tier systems, SPECjAppServer.

**Keywords: scalability of middleware, replication, caching, reliability, fault-tolerance.**

## 1 Introduction

The new vision of Enterprise Grids [1] is demanding for the creation of highly scalable and autonomic computing systems for the management of companies' data centers. Data centers are the most critical infrastructures of companies and they are demanding higher and higher levels of quality of service (QoS), i.e., availability, scalability... At the core of data centers we find multi-tier middleware architectures providing services to applications. Multi-tier architectures provide separation of concerns between presentation (front end), business logic (middle-tier), and data storage (back-end). Clients interact with the front end, which acts as a client of the middle-tier or application server. All the computation is done at this level and data is stored in the back-tier (in general, a database).

Current approaches for providing availability in multi-tier architectures replicate the application server tier (middle-tier) and share a common database. Recent work in J2EE [2] take this approach and allow sharing the load among application servers. We call these shared database approaches horizontal replication (they replicate a single tier).

* Patent pending.

The main shortcoming of horizontal replication is that the shared database becomes a bottleneck (limiting scalability) and a single point of failure (lack of availability for the database tier). An alternative is to combine two replicated tiers (i.e., application server and database). However, this approach may lead to inconsistencies and attaining a consistent integration in a scalable way is still an open problem [3]. In FT-CORBA, replication was either primary-backup or active replication so, they did not address scalability [4, 5].

J2EE application servers cache an object oriented view of the database items used by the application. In order to keep consistent this view with the database application servers implement some caching policy to match the isolation provided by the database. Current application servers provide serializability, since databases have relied on serializability as the correctness criterion for transactions for a long time. However, today many databases provide snapshot isolation as the highest isolation level (e.g., Oracle, PostgreSQL, MS SQL Server, etc.). Snapshot isolation provides a very similar level of consistency (it passes the tests for serializability of standard benchmarks such as the ones from TPC). Snapshot isolation [6] is a multi-version concurrency control in which transactions see a snapshot of the database as it was when the transaction started. Moreover, readers and writers do not conflict. Read-write reduce the potential concurrency and the performance of the system. Therefore, current application server implementations are incorrect when used with databases providing snapshot isolation.

In this paper we propose a replicated multi-version cache that improves performance by avoiding many accesses to the database. It also provides availability, consistency, and scalability. In our architecture each application server is connected to a local copy of the database, this pair is the unit of replication (vertical replication). This contrasts with previous approaches where the database is shared by all the replicas of the middle-tier. Thus, avoiding a single point of failure and a bottleneck. Our replication solution is fully achieved within the application server tier on top of an off-the-shelf 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. The replicated cache is based on snapshot isolation and provides full consistency. On a single server, the cache provides caching transparency. That is, the semantics is the same as the one the system would have without caching. On a replicated system, it provides one-copy correctness, that is, the replicated system behaves as the non-replicated one.

To the best of our knowledge this paper is the first to provide a complete, consistent and scalable solution for the replication of both the application server and database tier. It is also the first paper to address the use of snapshot isolation in application servers in a consistent manner with the database.

We have implemented the replicated multi-version cache and integrated it into a commercial open source J2EE application server, JOnAS [7]. The performance of the implementation has been evaluated with the new industrial benchmark, SPECjAppServer [8]. The prototype outperforms the non-replicated application server and shows a good scalability for an increasing number of replicas both in terms of throughput and response time.

The paper is structured as follows. Section 2 introduces the background of the paper: J2EE and snapshot isolation. Sections 3 and 4 present the replication model and the

cache protocol, respectively. Failure handling is described in Section 5. The results of the performance evaluation are shown in Section 6. Section 7 presents related work. Finally, Section 8 concludes the paper.

## 2 Background and Motivation

### 2.1 J2EE

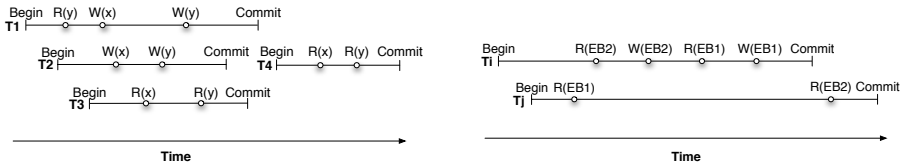
J2EE [9] is an extensible framework that provides a distributed component model along with other useful services such as persistence and transactions. J2EE components are called *Enterprise Java Beans* (EJBs). In this paper, we consider EJB 2.0 specification. There are three kinds of EJBs: *session beans* (SBs), *entity beans* (EBs) and *message driven beans*. We will not consider the former ones in this paper. SBs represent the business logic and are volatile. SBs are further classified as stateless (SLSBs) and stateful (SFSBs). SLSBs do not keep any state across method invocations. On the other hand, SFSBs may keep state across invocations of a client (conversational state). If a method execution changes the state of a SFSB, that state will be available to the same client upon the following invocation. EBs model business data and are stored in some persistent storage, usually a database. EBs are shared by all the clients.

EBs are typically managed by the application server (*container managed persistency*) and accessed within a transaction. The application server (AS) takes care of reading from and writing to the database by generating the adequate SQL statements (object oriented to relational model translation). Since EBs are cached tuples of the database, the application server should implement concurrency control mechanisms to satisfy the isolation level provided by the database, typically serializability.

In J2EE, transactions are coordinated by the *Java Transaction Service* (JTS). Transactions access this service using the *Java Transaction API* (JTA). J2EE allows to handle transactions either explicitly (*bean managed transactions*) or implicitly (*container managed transactions*, CMT). With CMTs, the container intercepts bean invocations and demarcates transactions automatically.

### 2.2 Snapshot Isolation

Snapshot Isolation (SI) [6] is a multi-version concurrency control mechanism used in databases (e.g., Oracle, PostgreSQL, SQL server, ...). One of the most important properties of SI is that readers and writers do not conflict. This is a big gain compared to serializability, the traditional correctness criteria for databases, where a write operation prevents any reader. SI avoids the ANSI SQL phenomena described in [6]. The database keeps a timestamp. When a transaction $T_i$ begins, it gets a *start timestamp*. This timestamp denotes the snapshot of the transaction which consists of the latest committed values of the database (version). Every read operation of $T_i$ is performed on the snapshot associated to $T_i$. Writes of $T_i$ are done in a private copy. Subsequent read operations of $T_i$ will read its own writes. When $T_i$ finishes it gets a *commit timestamp*. The two timestamps of a transaction are used to validate the transaction, that is, to determine if the transaction will commit or abort. $T_i$ will commit if there is no other concurrent committed transaction $T_j$ that has written some common object. That is, $T_j$ commit timestamp




**(a)** Snapshot Isolation **(b)** Regular Caching Anomaly with SI

**Fig. 1.** Snapshot Isolation

is in the interval of the start and commit timestamp of $T_i$. If $T_i$ commits, its changes (writeset) are made visible to other transactions that start after $T_i$ commits. So, several committed versions of the same object may exist at a given point in time. Each version belongs to a different snapshot. For instance Fig.1.a shows four transactions. Both $T1$ and $T2$ write $y$, so each transaction will create a version of $y$. When $T2$ commits, this version of $y$ will be visible to transactions that start after $T2$ commits (e.g., $T4$). When $T3$ reads $x$ and $y$ it will read the values they had before $T1$ started (a transaction reads the committed values at the time it started). $T1$ will not commit, since it is concurrent to $T2$, $T2$ committed and they conflict (both write $x$ and $y$).

### 2.3 Application Server Caching and Replication

J2EE implementations provide caching for entity beans as follows. An entity bean (EB) represents a cached tuple of the database at the application server. So, before accessing an EB, if it is not in memory the corresponding tuple is read from the database. If the EB is updated, the associated tuple will be updated at the database when the corresponding transaction commits. The EB is cached in memory, so other transactions accessing that EB will not read it from the database. However, this caching is not multi-version and fails to provide snapshot isolation. Let us show an example. Given two concurrent transactions $T_i$ and $T_j$ (Fig. 1.b). $T_j$ reads $EB1$, that is, the corresponding tuple is read from the database and $EB1$ is cached by the application server. Then, $T_i$ reads from the database $EB2$ and updates it. Later, $T_i$ reads $EB1$ from the cache, updates it and commits. Both updates are done in memory and then written back to the database when the transaction commits. Then, $T_j$ reads $EB2$. $T_j$ should read the value $EB2$ had when $T_j$ started. However, since the application server cache is not multi-version and $EB2$ is in memory, $T_j$ will read the cached value written by $T_i$. Therefore, snapshot isolation is violated.

One solution to this problem is to suppress caching at the application server and submit every read operation to the database. However, this solution is poor in terms of performance.

In order to avoid the anomalies of J2EE caching when used with a snapshot isolation database, we propose a multi-version cache for entity beans. For each entity bean (EB), instead of keeping a single copy at the cache of an application server, a list of potentially more than one version is cached. Each version $EB_i$ of an EB is tagged with the commit timestamp $i$ of the transaction that updated (and committed) this version. The multi-

version cache is replicated in several application servers in order to provide scalability, availability, and data consistency. The semantics of the replicated multi-version cache is as if there was a single multi-version cache.

## 3 Replication Model

We consider a vertical replication model in which a J2EE application server (AS) and database (DB) are collocated in the same site [3]. This is the unit of replication, also called replica. Each AS communicates with its DB and with the remaining ASs. That is, DBs are not shared among ASs. The set of all replicas is called a *cluster* (Fig.2). Sites fail by crashing. We will consider that a replica fails if either the database or the AS crashes. This is natural since they are collocated.

The replication protocol relies on group communication. Group communication provides multicast and the notion of view (currently connected cluster members). We will use reliable and totally ordered multicast [10]. The multicast also provides strong virtual synchrony that guarantees that the relative order of delivering membership changes (views) and multicast messages is the same at all replicas.

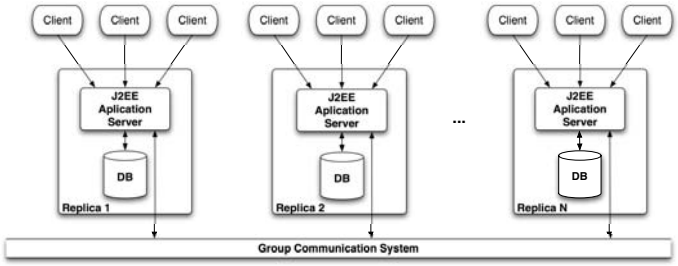



**Fig. 2.** Replication model.

Since we are considering a scenario with container managed transactions, each client request will be automatically bracketed as a transaction. So, there is a one to one relationship between requests and transactions.

In here, we are interested in container managed transactions, where each client request is automatically bracketed as a transaction by the application server using the Java Transaction API. So, there is a one to one relationship between requests and transactions. Clients access to a session bean. A session bean may access another session or entity beans. A client can be connected to any of the replicas. That replica will execute client transactions. If the transaction is read only (it does not modify any data), it will be just executed at that replica. Otherwise, the changes of update transactions will be multicast to the rest of the replicas. Then, the transaction is validated. Upon successful validation the transaction is committed and the result is sent to the client.

## 4 Replication Protocol

In the following sections we describe the protocol in detail (Fig.3). First, we describe how transactions are handled locally (we refer to them as local transactions) by each replica and how the multi-version cache works. Then, we describe how transaction changes are propagated and handled by the rest of the replicas (we refer to them as remote transactions).

**Local Transaction Processing.**

Entity beans are used within transactions and transactions run under snapshot isolation. When a transaction $T$ starts at the application server (AS), a transaction $t$ is also started at the database. The correlation between AS transactions and database transactions is stored in a table (Fig.3.begin.II, III). Each transaction at the AS will be associated with a *start timestamp*, when it starts (begin.I) and a *commit timestamp* at commit time. The commit timestamp $CT(T)$ of transaction $T$ is an increasing number that reflects the snapshots through which the data progresses (i.e., number of committed transactions). The start timestamp $ST(T)$ of a transaction $T$ is the highest commit timestamp at the start time of $T$. That is, it represents the last committed transaction $T'$ and indicates that $T$ should be able to read the versions written by $T'$ and transactions that committed before $T'$. We assume that the initial start timestamp is 0 at all replicas. Our protocol will guarantee that all update transactions in the system will be committed at all replicas in the same order, and thus, will receive the same commit timestamp at all replicas. Each version $EBX_i$ of an EBX is tagged with the commit timestamp $i$ of the transaction that updated (and committed) this version. When a transaction reads an EB, it will first look for the EB in the cache. The EB version a transaction will access is the last committed version as of the time the transaction started. That is, the EB version that has a version identifier $i$, such that $i <= ST(T) \wedge \nexists EBX_j : i < j <= ST(T)$ ((Fig.3).read.I,II). If an EB is not cached in memory, then it is read from the database (read.III). Since a transaction is started at the database when a transaction starts at the application server, and the database provides snapshot isolation, the database will return the correct version of the EB. This process guarantees that each transaction observes a snapshot as of the start of the transaction and therefore it does not violate snapshot isolation. Since the database does not show the versions associated to tuples, when an EB is read from the database its version number is unknown. Thus, an EB read from the database is tagged with -1.

Note that our approach requires the database to provide snapshot isolation. This is needed because it is unlikely that the cache can keep the entire database, i.e., all versions of all tuples. For instance, assume the database uses the isolation level read committed. Assume further that a transaction $T_i$ reads and modifies $EB1$ while a concurrent transaction $T_j$ updates $EB2$ and commits. Assume now further that, due to lack of memory, the version of $EB2$ that $T_i$ needs to read, is removed from the cache. Hence, it has to read it from the database. Since transactions run at the database with read committed isolation, $T_i$ reads the value committed by $T_j$. That is, it will not read the value of $EB2$ at the time $T_i$ started. In this case, the transaction at the application server will violate snapshot isolation.

timestamp = 0
cache = $\emptyset$
committedTX = $\emptyset$
transactionTable = $\emptyset$
oldestActiveTx = array[1..NumberReplicas] of Int =0

**begin(T)**

I. ST(T)= timestamp
II. t = begin transaction in the DB
III. store(transactionTable, T, t)

**read(T, EBX)**

I. if $EBX_{ST(T)} \in$ cache then
  1. return $EBX_{ST(T)}$
II. else if $\exists EBX_y \in cache \wedge y = max(x) \mid EBX_x \in cache \wedge y < ST(T)$ then
  1. return $EBX_y$
III. else
  1. t= getTX(transactionTable(T))
  2. $EBX_{-1}$ =read(t,EBX)from the DB
  3. cache = cache $\cup\{EBX_{-1}\}$
  4. return $EBX_{-1}$

**write(T, EBX, value)**

I. if $\exists EBX_{private}$ created by T $\vee$ (locked(EBX) $\wedge$ lockedBy(EBX)= T) then
  1. write(EBX, value)
II. else if lockedBy(EBX)$\neq$ T then
  1. block(T)
III. else
  1. lock(EBX) by T
  2. if $\exists EBX_i \in cache \mid i > ST(T)$ then
    a. abort(T)
  3. else
    a. create($EBX_{private}$) by T
    b. write(EBX, value)

**commit(T)**

I. if ReadOnly(T) then
  1. commit to the DB(getTX(T))
II. else
  1. multicast(changes(T), T, oldestLocalActiveTx)

**abort(T)**

I. $\forall$ EBX | lockedby(EBX)= T do
  1. delete($EBX_{private}$)
  2. unlock(EBX) – unblock first blocked TX
II. abort in the DB (getTransaction(T))
III. delete(transactionTable, T)

**upon delivery of (changes(T), T, oldestLocalActiveTx)**

I. CT(T) = timestamp + 1
II. oldestActiveTx[Sender(T)]=oldestLocalActiveTx
III. if validate (changes(T), T) then
  1. timestamp = timestamp + 1
  2. if local(T) then
    a. $\forall EBX \in$ changes(T) do
    b. cache = cache $\cup\{EBX_{CT(T)}\}$
    c. unlock(EBX) – abort blocked TXs
    d. enddo
  3. else $\forall EBX \in$ changes(T) do
    a. if locked(EBX)= LT $\wedge$ Local(LT) then abort(LT)
    b. lock(EBX) by T
    c. recreate(EBX)
    d. cache = cache $\cup\{EBX_{CT(T)}\}$
    e. t = begin transaction in the DB
    f. store(transactionTable, T, t)
    g. apply (getTX(T), changes(T)) to the DB
    h. unlock (EBX) – unblock blocked TX
    i. enddo
  4. commit(getTX(T)) to the DB
  5. committedTx = committedTx $\cup\{$T $\}$
  6. delete(transactionTable(T))
IV. else if Local(T) then abort(T)

**validate(changes(T), T)**

I. if $\exists T_K \in$ commitedTX $\wedge$ concurrent(T, $T_K$) $\wedge$ changes(T) $\cap$ changes($T_K$) $\neq \emptyset$ then return false
II. else return true

**garbageCollection()**

I. oldestTx = min(oldestActiveTx)
II. $\forall EBX \in$ cache do
  1. if $\exists EBX_i \in$ cache $\wedge i \neq -1 \wedge i < oldestTx$ then
    a. cache = cache - $EBX_i$
    b. if $EBX_{-1} \in cache$ then cache = cache-$EBX_{-1}$

**Fig. 3.** Replicated Cache Protocol

In snapshot isolation, when two concurrent transactions update the same data item, only one may commit, the other has to abort. In order to detect such conflicts early, we use locking and version checking. A transaction $T_i$ has to set a write lock on an EB, $EBX$ before updating it (write.I,III). This lock will prevent any other transaction from updating that EB. If another transaction requests a lock on $EBX$, it will be blocked (write.II). Once a transaction $T_i$ has a lock it knows no other transaction will be allowed to write $EBX$ at the same time. It now performs a version check on the version

$EBX_j$ with the highest version number in the cache. If the version is larger than the start timestamp $ST(T_i)$ of $T_i$, then this version was created by a concurrent transaction and $T_i$ must abort (write.III.2). Otherwise $T_i$ can perform the update, i.e., create its own version (write.III.3.a). This version will only be seen by the transaction. This guarantees that a transaction observes its own updates and prevents other transactions from observing uncommitted changes (write.I) If the transaction aborts (Fig.3.abort), the private versions of updated EBs are just discarded, and the locks released. The first waiting transaction is unblocked and receives the lock. The database transaction is also aborted. After transaction termination (commit or abort) all held locks are released.

If at the time of version check there is no cached version of the $EBX$, we allow $T_i$ to continue. For this to be correct, we have to guarantee that a version $EBX_{ST(T_i)}$ is not removed from the cache as long as there is a transaction $T_j$ running on any replica that is concurrent to $T_i$. If we removed this version, if $T_j$ would want to update $EBX$ it would not be able to detect the conflict. We will discuss later how the system can determine when an EB version is no longer needed for conflict detection and can be garbage collected.

For update transactions, after validation, the application server propagates the updated EBs to the other replicas (commit.II). The changes are propagated using reliable and total order multicast. This kind of multicast guarantees that all replicas including the sender will receive the same messages in the same order. They will all perform validation (upon.II) in the order they receive the messages. Thus validation will be deterministic and have the same outcome at all replicas. Note that processing messages is done serially. An update transaction fails validation (Fig.3.validate), if there is a transaction in the system that is concurrent, already committed and has overlapping changes. In fact, for local transactions, when they are delivered and were not yet aborted, validation will always pass, thus, we defer the details of validation when we discuss remote transactions in the next section.

At the local replica, when an update transaction succeeds validation, each modified EB (private version) is tagged with the transaction commit timestamp that is incremented with each committed transaction (upon.III.1). Then, updated private versions are tagged with the commit timestamp, made public (upon.III.2.b) and added to the EB version lists. Releasing the locks (upon.III.2.c) will trigger the abort of waiting transactions. The final steps (upon.III.4-6) commit the database transaction, and keep appropriate track of the committed transaction. The database commit will automatically propagate the EB changes to the database. However, it should be noted that the EBs are still cached in memory (until they are evicted by the caching policy). Fig.4 shows an example of the evolution of the cache on a single replica (without remote transactions).

Since locking might yield deadlocks, a deadlock detection and resolution mechanisms is in place. Notice that deadlocks will rarely occur, since they only involve write-write conflicts.

**Remote Transaction Processing.** As already mentioned, once an update transaction has been processed locally, its updates are multicast to all replicas including the local replica (the one at which the transaction was executed). This local transaction is a remote transaction at the rest of the replicas. Since all replicas may execute update transactions, each replica will validate (determine the outcome of) all update transac-

tions to enforce snapshot isolation at the global level. As a consequence, all update transactions will commit in the same order at all replicas and therefore, the state of the replicated cache will be consistent across replicas and all databases have the same state.

The first step at a remote replica after receiving a propagation message is to assign a commit timestamp (Fig.3.upon.I). The validation process (upon.II, Fig.3.validate) looks for concurrent committed transactions that have updated some common EBs (i.e. they had write conflicts) at different replicas. The validation checks whether the transaction under validation conflicts with some previous successfully validated concurrent transaction (i.e. the transactions that were not known before its propagation). If a lock held by a non-validated (local and uncommitted) transaction is found while creating the new version of an EB, the local transaction is aborted (it is concurrent, conflicts and has not been validated yet) (upon.III.3.a). If the validation phase of a remote transaction succeeds, a transaction is started at the database and the changes of the transaction must be applied. In order to apply the changes a new version of each updated EB will be created, tagged with the transaction commit timestamp and added at the beginning of the corresponding version list in the cache (the local cache is updated) (upon.III.3.b-d). Also a database transaction is started (upon.III.3.e). The remaining steps are as with local transactions (upon.III.4-6) where the database transaction commits, the commit counter is increased and the transaction is stored in the list of committed transactions. If a remote transaction fails validation nothing has to be done. The message is simply ignored.

For instance, two replicas $R1$ and $R2$ may execute two transactions $T1$ and $T2$, respectively, that read and update $EB1$ concurrently (e.g., they have the same start timestamp: $ST(T1) = 0$, $ST(T2) = 0$). When they finish at their local replicas, their changes are multicast. Let us assume the total order is $T1, T2$ and there is no other concurrent conflicting transaction (Fig. 5). When $T1$ is delivered it gets its commit timestamp ($CT(T1) = 1$).The validation of $T1$ will succeed (there is no other concurrent committed transaction) at $R1$ and commit. Then, $T2$ is processed. It receives $CT(T2) = 2$. During the validation of $T2$ it will be found that $T1$ is concurrent ($ST(T1) = ST(T2)$ and $CT(T1) < CT(T2)$), conflicts and $T1$ committed. Therefore, $T2$ will abort. At replica $R2$ transactions are validated in the same order. However, during the validation of $T1$ it will be found that $EB1$ is locked (by $T2$), $T2$ will be aborted (upon.III.3.a) and $T1$ commit. Thus, when $T2$ is later delivered, validation will fail. Therefore, the two replicas will commit the same transactions (Fig. 6).

**Dealing with Creation and Deletions of EBs.** Creation and deletion of EBs is also handled by the protocol (not shown in Fig.3). When a new EB is created, a private version is created for the transaction and there is no other version available. A lock is also set on the EB to prevent concurrent creations of the same EB (with the same primary key). When the transaction commits, the version becomes available for transactions that started after the creating transaction committed and the corresponding tuple is inserted in the database.

Deletions create a tombstone version of the EB. The tombstone is also a private version of the transaction till commitment. If the transaction tries to access the EB, it will not find it, since the protocol will find the tombstone and recognize the EB as deleted. When the transaction commits, the tombstone version will become public. Even

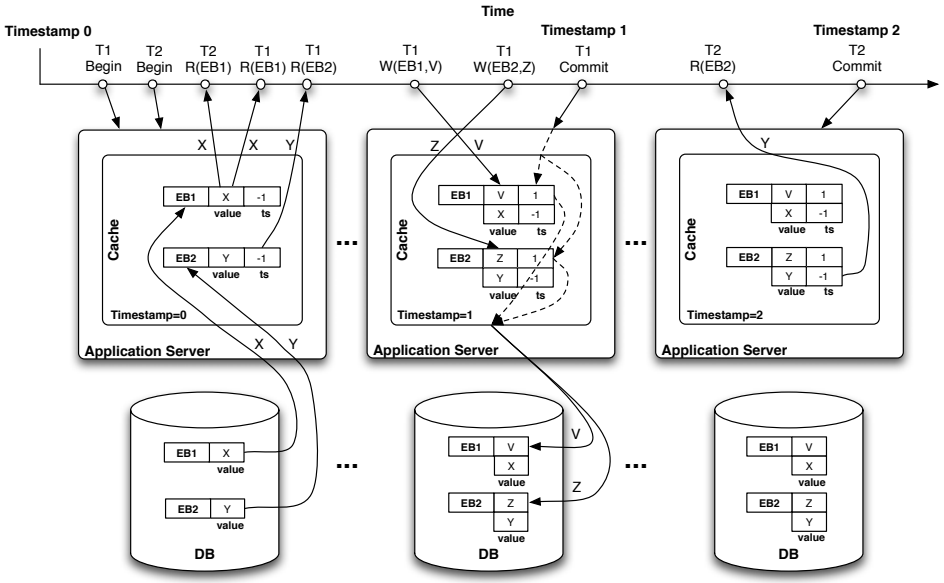



**Fig. 4.** Evolution of the cache in a single replica.

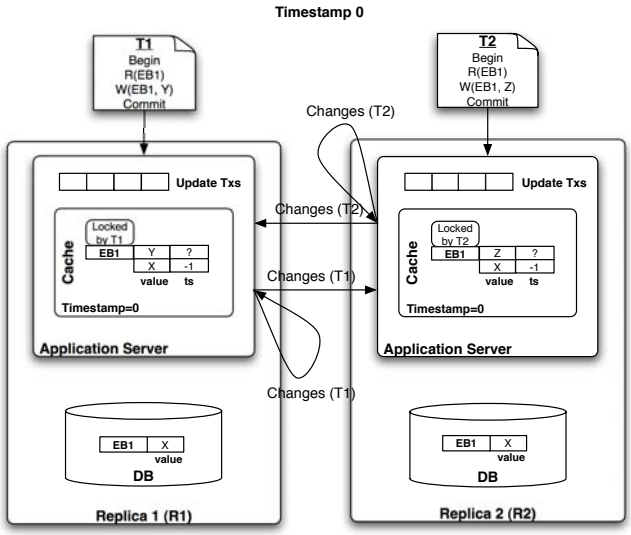



**Fig. 5.** Two concurrent conflicting transactions (I).

after transaction commit previous versions of the EB cannot be removed, since there

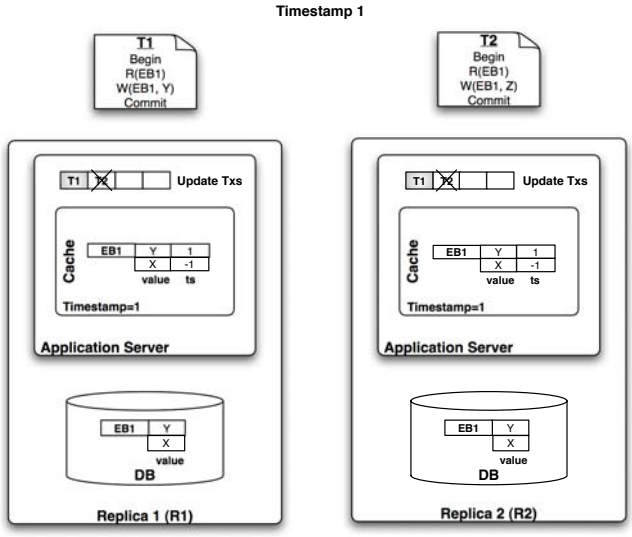



**Fig. 6.** Two concurrent conflicting transactions (II).

might be active transactions associated to older snapshots (all transactions that started before the one that deleted the EB committed), that may read the EB.

**Version Garbage Collection.** Since EB versions are kept in memory (in the cache), they should be removed to free space in the cache when they are not needed. For this purpose, there is a garbage collection mechanism that discards unneeded EB versions (Fig.3 garbage collection). This is achieved by piggybacking in the message that propagates the transaction changes the oldest start timestamp associated to an active local transaction at that replica. Each replica will remove versions older than the oldest start timestamp among all replicas (garbage.II.1). A vector with these identifiers is kept at each replica to assist the garbage collection. More difficult is to deal with EB versions tagged -1. If a newer version of an EB $EB_i$ is not needed (there is no active transaction at any replica with start timestamp smaller than $i$), then version $EB_{-1}$ is also not needed. Note that if the cache if full, what J2EE application servers do is to evict EBs from the cache to a local disk repository (not the database) by means of a standard hibernation mechanism. Thanks to this our versioning is not affected by the eviction policy of the cache. When a hibernated EB is going to be accessed the application server brings the hibernated EB to memory (all the EB versions).

**Session Replication.** Stateful session beans (SFSBs) keep conversational state from a client and their replication is not required to provide data consistency and availability. However, if they are not replicated, a failure of a replica will cause the loss of the conversational state kept in the SFSB corresponding to all previously run transactions by the client at that replica. The conversation could not be resumed after the failover, what results in loss of session availability. For this reason, the replication protocol also repli-

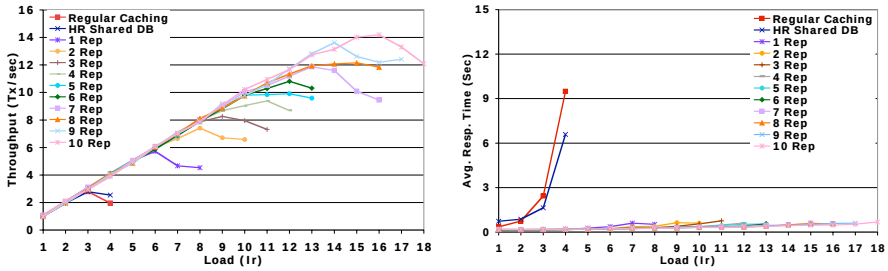
cates the state of SFSBs after each method invocation. Since SFSBs are only replicated to increase availability when a failure occurs one could consider not to replicate them to all replicas. The submission 29512 to Middleware 2007 considers this issue.

## 5 Failure Handling

Clients connect to the application server through stubs that are obtained from the application server through JNDI (Java Naming and Directory Interface). Since stubs are generated by the application server, we can incorporate the necessary replication logic in a fully transparent way to clients. We have introduced in the stub the capability of performing replica discovery, resorting to IP-multicast. The stub IP-multicasts a message to an IP-multicast address associated to the application server cluster. Replicas have an identifier (from 0 to $n-1$). Client requests are univocally identified with a unique client identifier and a unique request number (a counter kept at the stub that is incremented after each successful request). One replica (using this identifier) in the cluster returns the stub with the list of available replicas (their IPs) as well as an indication of their current load. Replicas multicast information about their load periodically. The stub then selects a replica randomly giving a selection probability inversely proportional to the load of the replicas to attain load balancing. The stub connects to the selected replica. When the stub succeeds in connecting to one replica, it will send all subsequent client requests to it. If a replica fails, the stub will time out and connect to a new replica.

Let us now consider the failover logic at the application server side. Each replica contains a pair of application server (AS) and database. If any of them fails or the site in which they are collocated fails, the replica is considered as failed. There are mechanisms to detect partial failures (only the AS or the database fails) and shutting down the replica to enforce a crash failure. Since we are considering a scenario with container managed transactions, each client request will be automatically bracketed as a transaction. So, there is a one to one relationship between requests and transactions. At the client side the failure will be detected when the stub times out while there is an outstanding request. The stub will reconnect to a new replica and resubmit that request. Then, there are three scenarios: (1) the request was delivered but it was not processed by the failed replica, (2) the replica failed after processing the request but before the updates were multicast to the other replicas, (4) the replica multicast the resulting updates to the other replicas and then it failed.

If there have been previous interactions of that client, the new replica the stub connects to will have the last checkpointed state of the stateful session bean (SFSB) associated to the client. It will deserialize the SFSB and then process the client request. In case (1), since requests are univocally identified with a unique client identifier and a unique request number, the new replica will identify this request as a non-previously seen request and process it as a regular request. Case (2) is similar. The remaining replicas do not know about the transaction. So failover is handled as in (1). Case (3) can be deal with, but it may be expensive because the responses ASs send to clients must also be sent in the multicast message. So, the new replica will recognize the resubmitted request as a duplicate, and will discard it, since the first request was successfully multicast by the failed replica. If the transaction succeeded at validation, the replica will return the




**(a)** Throughput **(b)** Response Time (Browse)

**Fig. 7.** SPEC Results

checkpointed request response to the client. Otherwise, it will discard the checkpointed response and return an exception to the client notifying that the transaction was aborted due to snapshot isolation could not be guaranteed.

# 6 Evaluation

## 6.1 Evaluation Setup

The evaluation has been performed in a cluster of 10 bi-processors (20 CPUs) connected through a 100 Mbps switch. Sites have 2 AMD Athlon 2GHz CPUs, 1 GB of RAM, two 320 GB hard disks and run Fedora Linux. Each replica consists of one JOnAS v4.7.1 application server (AS) and PostgreSQL v.8.2 database. JGroups [11] is used as group communication system.

We compare the results of our replicated multi-version cache with the traditional caching of JOnAS and a replicated application server with 2 replicas sharing a single database (horizontal replication) where only stateful session beans are replicated.

We use the dealer application of SPECjAppServer in our evaluation. SPECjAppServer is a benchmark developed by SPEC (System Performance Evaluation Cooperative) to measure the performance of J2EE application server implementations [8]. In this application there is a workload generator (driver) that emulates automobile dealers interacting with the system through HTTP. The driver injects three different business transaction types: purchase vehicles (25%), manage customer inventory (25%), browse vehicle catalog (50%). Browse transactions are read-only, purchase transactions have a significant amount of writing, and management transactions exhibit the highest fraction of updates.

The main parameter in the tests is the injection rate (Ir), which models the injected load. The number of clients is Ir x 10. The SPECjAppServer specifies a maximum response time for all requests (2 seconds). Furthermore, the response time corresponding to the 90% percentile may be at most 10% higher than the average response time. The throughput is measured as the business transactions completed per second (Tx/sec).

### 6.2 SPECjAppServer Benchmark Results

Fig. 7(a) shows the overall throughput and Fig. 7(b) together with Fig 8(a-b) show the response time for browse, purchase and management transactions with increasing loads. The figures show graphs for regular caching, horizontal replication with 2 replicas (HR Shared DB) and our approach with 1-10 replicas.

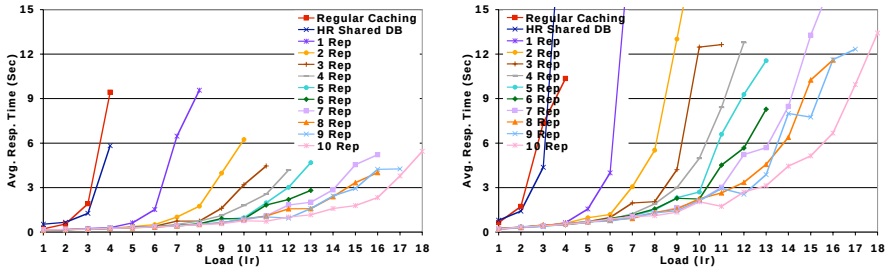
When looking at the throughput (Fig. 7(a)), the first noticeable fact is that traditional caching and horizontal replication can only handle a load Ir up to 3. In contrast, our replicated multi-version cache outperforms these two implementations even if there is only one replica by a factor of 2. The reason is that the multi-version cache is able to avoid many database reads compared to the traditional caching implementation. Horizontal replication did not help because the shared database was already saturated with two application server replicas. When testing with three replicas (not shown in the figures), the system deteriorated and not even achieved an Ir of 1. The replicated multi-version cache is able to handle a load up to 14 Ir achieving a throughput of 14 Tx/sec with 10 replicas compared to a throughput of 6 Ir and 6 Tx/sec with a single replica (and 3 Ir and 3 Tx/sec with traditional caching). That is, by adding new replicas a higher number of clients can be served.

At the beginning, adding a new replica will increase the throughput by 2 Tx/sec, after a certain number of replicas the increase is 1 Tx/sec. From nine to ten replicas the gain is around 0.5 Tx/sec. The reason is that changes performed by update transactions have to be applied at all replicas. With increasing load each replica spends more time applying changes and has less capacity to execute new transactions. Nevertheless, the scale-up achieved with our approach by far outperforms the existing solutions.

Even when the replicated cache configurations saturates (the throughput is lower than the injected load), configurations with a higher number of replicas exhibit a more graceful degradation. For instance, for Ir =13, both the 5-replica an 8-replica configuration are saturated. However, the achieved throughput at 8 replicas is higher than with 5 replicas, providing clients a better service. This is very important, since it will help the system to cope with short-lived high peak loads without collapsing.

Another important conclusion from the response time experiments, it is that browse transactions are not affected by the saturation of update transactions. As could be seen in Fig. 7(b) the response time curves are almost flat independently of the number of replicas and even for the largest Ir when the system reaches saturation. The reason is that for read-only queries our application server caching is very effective avoiding expensive database access in many cases. Also, read-only transactions do not require communication. We can observe the saturation of regular caching and horizontal replication at an Ir of 3, since the response times increase exponentially for browse transactions (Fig. 7(b)).

Purchase transactions (Fig. 8(a)) are quite different since they are update transactions. The response time for all configurations reaches saturation at some time point. The response times for traditional caching and horizontal replication are worse than for the multi-version approach even for low loads showing that our caching strategy saves expensive access to the database. Furthermore, the replicated architecture can provide low response times until saturation is reached. Finally, the more replicas the system has,




**(a)** Response Time Purchase **(b)** Response Time Manage

**Fig. 8.** SPEC Results

the more graceful is the degradation of the response time at the saturation point. This is important to provide acceptable response times in case of short-lived peak times.

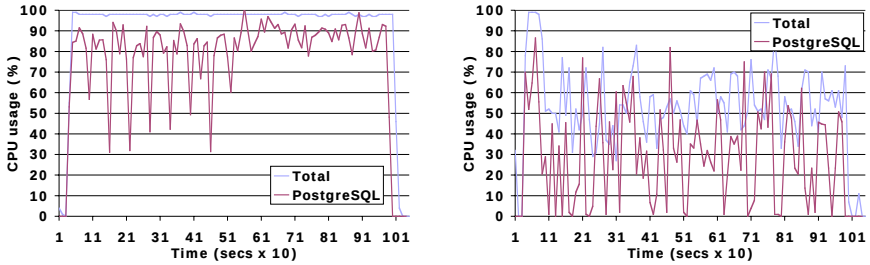
The different behavior of the purchase transactions compared to browse transactions has to do with the fact that update transactions propagate their changes to all the replicas in the system, and also have to write changes to the database. Thus, bottleneck components are accessed leading to the degradation of the response time. This behavior is even more noticeable in the case of manage transactions, which have the highest percentage of updates (Fig. 8(b)). Again, however, degradation of response times is more graceful with larger number of replicas.

### 6.3 CPU Analysis

In this section we look at the CPU usage of the database and the application server during 16 minutes of executing the benchmark in order to understand the results of the benchmark evaluation. Each of the following figures shows two graphs. One graph is the CPU usage of the database and the other is the overall CPU usage. The gap between the two graphs is mostly the application server (and replication protocol) CPU usage.

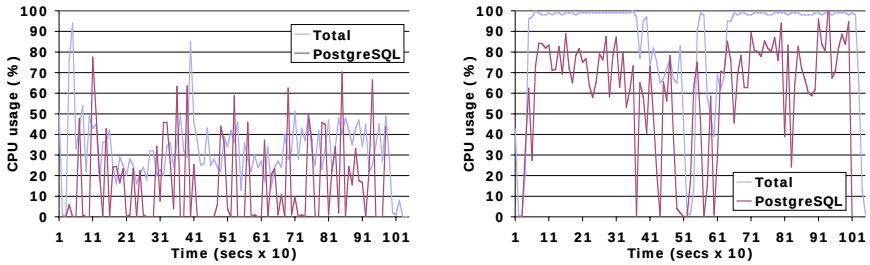
The results at a load of 4 Ir for regular caching and a single replica with our multi-version cache are shown in Fig.9. At this load, the system is saturated with a 100% usage of the CPU with 1 replica and regular caching (Fig.9.a). It can also be seen that the database consumes most of the CPU. There are depressions in the utilization graph of the database. They have to do with the way PostgreSQL handles updates. Periodically, when buffers are full, it stops transaction processing and forces data to disk. This results in under using the CPU. The single replica multi-version cache configuration shows a significantly smaller CPU usage (Fig. 9.(b)). The CPU usage of the database is much smaller due to the multi-version cache. This saves database access and reduces the CPU resources required by the database instance. Thus, the system is not saturated at an Ir of 4.

Examining the 2-replica configuration of our replicated cache for Ir=4 (Fig. 10(a)), the results are quite different. Although there are some high peaks in the CPU usage, the area covered is much smaller than for 1-replica configuration. The overall CPU



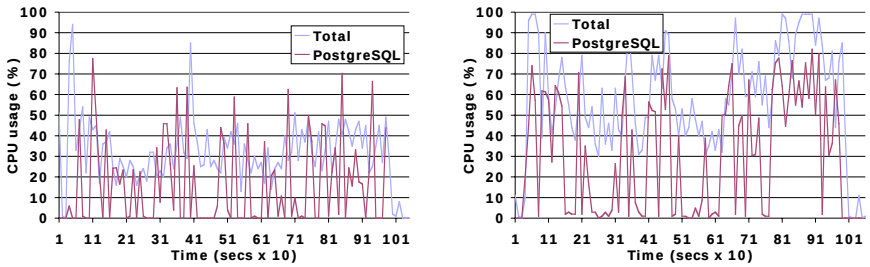

**(a)** regular caching **(b)** multi-version cache

**Fig. 9.** CPU usage: One replica, Ir=4




**(a)** Ir=4 **(b)** Ir =10

**Fig. 10.** Multi-version cache. CPU usage: Two replicas




**(a)** Ir=4 **(b)** Ir =10

**Fig. 11.** CPU Usage: Six replicas

usage has been significantly reduced. This means that for the same load the overhead at each replica is smaller, resulting in an effective sharing of the load. In the 6-replica configuration and Ir=4 (Fig. 11(a)), CPU usage is even further reduced with a very low amount of CPU devoted to the database. This explains the scalability of our approach. The more replicas in the system, the better is the load distributed.

| | No Replication | Replication | When |
|---|---|---|---|
| JGroups | | 130,00 | update tx |
| Replication Classes | | 143,00 | update tx |
| Entity Serialization | | 3,20 | update tx |
| SFSB Serialization | | 4,91 | SFSBs update |
| Entity bean caching | 152,00 | 110,00 | always |
| DB Access | 227,00 | 110,00 | always |

**Table 1.** JProfiler Results

Fig. 10.(b) shows the 2-replica configuration when it is saturated at Ir=10. At this load, the database usage of the CPU amounts to 80% what means that the database is the bottleneck. The 6-replica configuration is not saturated in this setting (Fig. 11.(b)) since the CPU usage of the database is lower. This confirms the effective distribution of the entire load (application server and database load) among replicas, what results in the scalability of the approach.

Another important conclusion is the efficacy of collocating application and database server on the same site which distinguishes our vertical replication approach from previous solutions. It enables adapting the CPU resources needed by each server without replica configurations. The operating system takes care of distributing CPU to the servers according their needs.

### 6.4 Profiling Tool Results

We also used a profiling tool, JProfiler, to analyze the differences in response time between the application server with and without the multi-version cache. It measures both the replication overhead and the savings obtained by the multi-version cache. Since the profiling tool introduces a very high overhead the profiling could only be done with a single replica and the lowest Ir of 1. The results show the overall number of seconds spent during the whole experiment on methods with different functionalities (Table 1). The group communication system (JGroups) and the replication classes introduce a non-negligible overhead as expected. However, it must be noticed read only transactions (50% of the load) are not affected by this overhead. The multi-version cache compensates this additional overhead by improving the caching efficiency and reducing the database access in a 27.6% and 51.5%, respectively.

### 6.5 Scalability Analysis

In this Section we measure the scalability of the replicated cache. To measure the scalability we take the response time (RT) threshold of the SPECjAppServer benchmark, 2 seconds, and see for each configuration (i.e. number of replicas) which was the maximum Ir with a response time below 2 seconds. Since it is also interesting the scalability under peak situations, we have also taken a second threshold, 5 seconds, to see how much the replica configurations scales under peak loads.

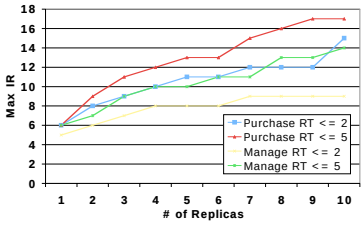



**Fig. 12.** Scalability Analisys

Fig. 12 shows the scalability results. For browse transactions we do not show any scalability curves since for all tested replica configurations and injected Ir the response time was quite below 2 seconds. That is, browse transactions exhibit linear scale up till the tested injected Ir. For purchase transactions, we can see that the scalability is better for a lower number of replicas than for a high number of replicas. For instance, 5 replicas manage 110 clients (Ir=11), that is, an average of 110/5=22 clients per replica. Recall that the number of clients is ten times the Ir. Whilst 10 replicas manage 150 clients (Ir=15), that is, an average of 15 clients per replica. The resulting scalability is pretty good taking into account the substantial fraction of updates involved in manage transactions. Purchase transactions scale better with a threshold of RT$<=$5. With this threshold it tolerates an Ir 3 units higher (30 clients).

Comparing manage with purchase transactions, purchase transactions scale better. This is also expected since manage transactions involve a higher percentage of updates resulting in a higher replication overhead. For this kind of transactions using more than 6 replicas does not increase the scalability anymore. However, if we look at the tolerance to peak loads (threshold RT $<=$ 5 secs) it can be seen that having additional replicas it is beneficial. Manage transactions with a threshold RT$<=$5 scale almost as purchase transactions. What means that with 10 replicas the peaks that will be born by the system will be significantly more intense than the ones born by the system with 6 replicas.

## 7 Related Work

Early work in application server replication took place around CORBA [4]. This work resulted in the FT-CORBA specification [12], where the application server is replicated and the database is shared (horizontal replication). This results in solutions providing availability for the application server tier. Replication of CORBA with transactional consistency has been solved in [5].

[13] presents a primary-backup approach for the replication of J2EE servers. It provides session availability, as we do in this paper. However, being primary-backup does not provide any scalability. [2] introduces a caching algorithm for J2EE application servers. Application servers are replicated and share the database (horizontal replication). Consistency is guaranteed through a certification protocol that before committing a transaction, reads every read entity bean and checks whether it was modified. This ap-

proach has the shortcoming of all horizontal approaches, the shared database becomes the bottleneck. The certification is much heavier than the one our replicated cache performs since it forces to re-read every read entity bean.

On the theoretical side, papers [14, 15] defined formally exactly-once correctness in multi-tier systems. They study the replication of stateful and stateless application servers with a shared database. In these proposals, 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. [16] applies this technique in a J2EE environment.

Our proposed replicated cache provides scalability and availability in contrast with the aforementioned previous work in horizontal replication that only provides availability and exhibits in the shared database a single point of failure and bottleneck.

[17] also explores middle-tier caching. The authors propose a freshness approach for data consistency in which inconsistency is bounded to miss a maximum number of update transactions (termed freshness). This consistency is very relaxed and contrasts sharply with the strong consistency provided by our approach. The simulation performed in the paper is evaluated with an ad-hoc benchmark. Our approach provides full consistency and it is a real implementation evaluated with an industrial benchmark.

[18] studies different approaches for providing consistent caching in dynamic web applications. This approach shares the same strong consistency goal as our multi-version cache. The main difference lies in that our approach also provides scalability.

Clustering (replication) is a facility provided by many commercial J2EE application servers. However, current approaches focus on the replication of SFSBs and rely on a shared database. This is the case of JBoss open source J2EE application server [19], Oracle9iAS [20], WebLogic clustering [21] and WebSphere 6.0 [22]. The state of SFSBs is multicast to the rest of the replicas after each method invocation. JBoss Cache is a replicated transactional cache for entity beans with a shared the database. It provides two ways to maintain data consistency: replication and invalidation. With replication, every entity bean in the cache is replicated to the rest of the replicas at the end of a transaction. That includes all data read by the transaction, which may be a huge amount of data. If the invalidation policy is used, only the primary keys of the entity beans are sent. Then, these entity beans are invalidated in the cache of the rest of the replicas, which must read the entity beans from the database. Moreover, there is a two-phase-commit protocol (2PC) in order to commit a transaction what results in a very heavyweight protocol. This approach only provides availability of the application server tier, and does not provide scalability, unlike our replicated cache.

## 8 Conclusions

We have presented a replicated multi-version cache that achieves integral replication of multi-tier systems. The replication protocol takes into account both the application server and the database encapsulating the replication logic within the application server. This enables to use the approach with off-the-shelf databases. The replicated multi-version cache is implemented at the application server level that allows a scalable solu-

tion even for update workloads. The replicated cache enables to take advantage of modern snapshot-isolation databases such as Oracle and PostgreSQL. The implementation has been accomplished in a commercial J2EE application server, JOnAS. The thorough evaluation has been performed using an industrial benchmark, SPECjAppServer, and the results have demonstrated the good scalability of the approach.

## References


1. Enterprise Grid Alliance: EGA Reference Model (2005)
2. Leff, A., Rayfield, J.T.: Improving application throughput with enterprise javabeans caching. In: ICDCS. (2003)
3. Kemme, B., Jimenez, R., Patiño, M., Salas, J.: Exactly once interaction in a multi-tier architecture. In: VLDB DIDDR Workshop. (2005)
4. Narasimhan, P., Moser, L.E., Melliar-Smith, P.M.: Eternal - a component-based framework for transparent fault-tolerant CORBA. SPE **32**(8) (2002) 771–788
5. Zhao, W., Moser, L.E., Melliar-Smith, P.M.: Unification of Transactions and Replication in Three-Tier Architectures Based on CORBA . IEEE TDSC (2005)
6. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: ACM SIGMOD. (1995)
7. Bull: JOnAS Clustering, https://wiki.objectweb.org/jonas/Wiki.jsp?page=JOnASClustering
8. SPEC: SPECjAppServer 2004 Benchmark, http://www.spec.org/jAppServer/. (2004)
9. Sun Microsystems: Java 2 Platform Enterprise Edition v1.4. (2003)
10. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: A comprehensive study. ACM Computer Surveys **33**(4) (2001)
11. : JGroups: A Toolkit for Reliable Multicast Communication. http://www.jgroups.org.
12. OMG: Fault Tolerant CORBA. Object Management Group (2000)
13. Perez-Sorrosal, F., Patiño-Martínez, M., Jiménez-Peris, R., Vuckovic, J.: Highly available long running transactions and activities for j2ee applications. In: ICDCS. (2006)
14. Frølund, S., Guerraoui, R.: e-transactions: End-to-end reliability for three-tier architectures. IEEE Trans. Software Engineering **28**(4) (2002) 378–395
15. Frølund, S., Guerraoui, R.: Implementing e-transactions with asynchronous replication. IEEE Trans. Parallel Distributed Systems **12**(2) (2001) 133–146
16. Wu, H., Kemme, B., Maverick, V.: Eager Replication for Stateful J2EE Servers. In: Proc. of Int. Symp. on Distributed Objects and Applications (DOA). (2004) 1376–1394
17. Bernstein, P.A., Fekete, A., Guo, H., Ramakrishnan, R., Tamma, P.: Relaxed-currency serializability for middle-tier caching and replication. In: SIGMOD Conference. (2006) 599–610
18. Attar, M., Ozsu, M.T.: Alternative architectures and protocols for providing strong consistency in dynamic web applications. WWW Journal (2006)
19. The JBoss Group: JBoss Application Server. http://www.jboss.org.
20. Oracle: Oracle9iAS Containers for J2EE. EJBs Developers Guide, Rel. 2 (9.0.4) (2003)
21. BEA Systems: WebLogic Server 7.0. Programming WebLogic Enterprise JavaBeans . (2005)
22. IBM: WebSphere 6 Application Server Network Deployment. (2005)