Consistent and Scalable Cache Replication for Multi-Tier J2EE Applications[⋆] [⋆⋆]

Francisco Perez-Sorrosal[1], Marta Patiño-Martinez[1], Ricardo 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 infrastructure of companies demanding higher and higher levels of quality of service (QoS) in terms of availability and scalability. At the core of data centers are multi-tier architectures providing service to applications. Replication is heavily used in this infrastructure for either availability or scalability but typically not for both combined. Additionally, most approaches replicate a single tier, making the non-replicated tiers potential bottlenecks and single points of failure. In this paper, we present a novel approach that provides both availability and scalability for multi-tier applications. The approach uses 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 infrastructure of companies demanding higher and higher levels of quality of service (QoS) in terms of availability and scalability. At the core of data centers lies a multi-tier middleware architecture providing services to applications. A multi-tier architecture provides separation of concerns in regard to 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. Most computation is done at this level and data is stored in the back-tier.

⋆ Patent pending.

⋆⋆ This work has been partially funded by the Spanish Research Council (MEC) under TIN2004-07474-C02-01, TIN2007-67353-C02-01, by the Madrid Research Foundation, S-0505/TIC/285 (cofunded by FEDER & FSE), by EUREKA/ITEA S4ALL (04025) funded by MITyC (FIT-3400005-2007-20) and by Automan (ARC funded by INRIA).

Replication can provide both scalability (load can be distributed across the replicas), and fault-tolerance (load submitted to a failed replica can be redirected to available replicas). Recent work on multi-tier replication, however, only replicates the application server tier while using a single database [2–7]. We call these shared database approaches *horizontal replication* (they replicate a single tier). The main shortcoming is that the shared database becomes a bottleneck and a single point of failure. An alternative is to replicate both tiers independently. However, attaining a consistent integration in a scalable way is still an open problem [8]. Furthermore, some of the approaches, e.g., FT-CORBA [9, 10] or [6] for J2EE focus on availability and use either primary-backup or active replication. Thus, they do not address scalability since neither technique allows the sharing of load among replicas.

J2EE application servers cache an object oriented view of the database items used by the application. In order to keep this view consistent with the database, application servers implement a concurrency control policy for the cache. They typically provide serializability as correctness criteria, implemented via locking or optimistic schemes, since databases have relied on serializability for a long time. However, today many databases provide snapshot isolation as the highest isolation level (e.g., Oracle, PostgreSQL, FireBird, etc.) and others implement it (MS SQL Server). Therefore, current application server implementations are incorrect when used with databases providing snapshot isolation. Snapshot isolation provides a similar level of isolation as serializability (it passes the tests for serializability of standard benchmarks such as the ones from TPC). Snapshot isolation is usually implemented via a multi-version mechanism in which transactions see a snapshot of the database as of transaction start [11]. Therefore, readers and writers do not interfere. In contrast, when implementing serializability, read-write conflicts lead to blocking or aborts, reducing the potential concurrency and the performance of the system. That is avoided by snapshot isolation.

In this paper we propose a replicated multi-version cache that improves performance by avoiding frequent access 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. The pair of application and database server is the unit of replication (*vertical replication*). This avoids that the database becomes a single point of failure and a bottleneck. Our replication solution is fully implemented 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. That is, using a single server, the cache provides caching transparency, i.e., its semantics is the same as a system that does not use caching. In a replicated system, it provides one-copy correctness and fault-tolerance, that is, the replicated system behaves as a non-replicated system (consistency) that never fails.

To the best of our knowledge this paper is the first to provide a scalable and integrated solution for the replication of both the application server and database tier. It is also the first paper to implement snapshot isolation for the cache of the application server tier so that it works properly with a database based on snapshot isolation. We have implemented the replicated multi-version cache and integrated it into a commercial open source J2EE application server, JOnAS [3]. The performance of the im-

plementation has been evaluated with the new industrial benchmark, SPECjAppServer [12]. The prototype outperforms the non-replicated application server and shows good scalability in terms of throughput and response time.

In the remainder of the paper, Section 2 introduces background on 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 performance evaluation is shown in Section 6. Section 7 presents related work, and Section 8 conclusions.

2 Background and Motivation

2.1 J2EE

J2EE [13] is a 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 the EJB 2.0 specification. There are three kinds of EJBs: *session beans* (SBs), *entity beans* (EBs) and *message driven beans*. We will not consider message driven beans 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. In contrast, SFSBs are associated with a client and keep session related information across invocations.

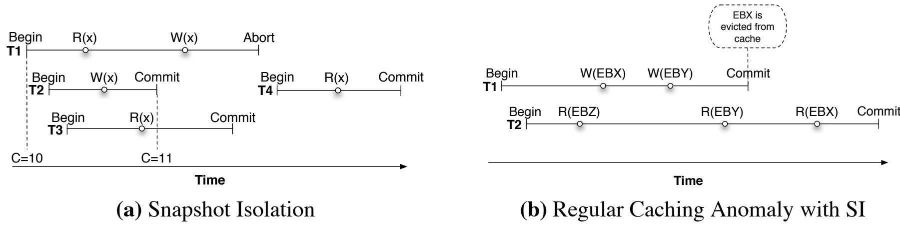
EBs model business data and are stored in some persistent storage, usually a database. EBs are shared by all the clients. An EB typically represents a tuple of the database. Thus, the set of EBs can be viewed as a cache of the database. Therefore, EBs are accessed within the context of transactions. EBs are typically managed by the application server (*container managed persistence*). The J2EE 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). Furthermore, it implements some concurrency control mechanism on the cache 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). In J2EE transactions can be handled either explicitly (*bean managed transactions*) or implicitly (*container managed transactions*, CMT). With CMTs, the container intercepts bean invocations and demarcates transactions automatically. We will focus on CMTs.

2.2 Snapshot Isolation

Snapshot Isolation (SI) [11] is a multi-version concurrency control mechanism used in databases (e.g., Oracle, PostgreSQL, MS SQL server). One of the most important properties of SI is that readers and writers do not interfere. This is a big gain compared to serializability, the traditional correctness criteria for databases, where a locking implementation prevents concurrent reads and writes on the same object while optimistic concurrency control aborts the reader.

SI can be implemented as follows. The system maintains a counter C of committed transactions. At commit time, C is incremented and the new value is assigned to the



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

Fig. 1. Snapshot Isolation

committing transaction T as *commit timestamp CT(T)*. At start, a transaction T receives as *start timestamp ST(T)* the current value of C. When a transaction writes a data item x, it creates a new (private) version of x. When reading a data item x a transaction T either reads its own version (if it has already performed a write on x) or it reads the last committed version as of the start of T. That is, it reads the version created by a transaction T' so that $CT(T')$ is the maximum CT of all transactions that wrote x and $CT(T') \leq ST(T)$. By reading from a snapshot, reads and writes do not interfere. However, if two concurrent transactions want to write the same data item, SI requires one to abort. Such conflicts can be detected at commit time. When a transaction T wants to commit, a validation phase checks whether there was any concurrent transaction $T\,'$ (i.e. $CT(T') > ST(T)$) that already committed and wrote a common data item. If such a transaction exists T aborts, otherwise it commits. If T commits, its changes (writeset) are made visible to other transactions that start after T_i commits.

Fig.1(a) shows an example with four transactions. We assume $C = 10$ and the transaction T with $CT(T) = 10$ updated x (not shown in the figure). Now $T1, T2$ and $T3$ start concurrently and all receive as start timestamp the value 10. $T2$ writes x. Its validation succeeds and $CT(T2)$ is set to 11. $T3$ reads the version of x created by T. Since it is read-only, no validation is necessary and it does not receive a commit timestamp. $T1$ reads the version of x created by T, and then writes x creating its own version. When $T1$ wants to commit, however, validation fails since there is a committed transaction $T2$, $CT(T2) > ST(T1)$, and $T2$ also wrote x. Therefore, $T1$ has to abort. Finally, $T4$ starts after $T2$ commits and receives $ST(T4) = 11$. It reads the version of x created by $T2$.

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. We denote the entity bean representing tuple x as EBX. When 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 then cached in memory so that it can be directly accessed by further transactions. Access to EBs is typically controlled via locking in order to provide serializability. However, this can lead to executions that neither provide serializability nor snapshot isolation when the database system uses snapshot isolation.

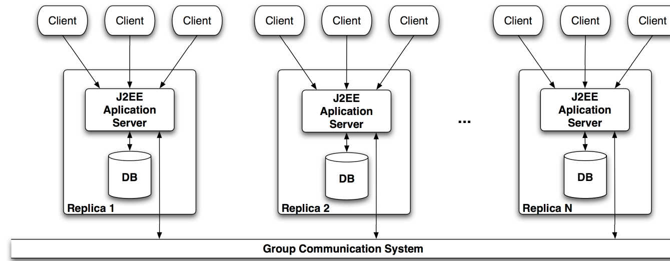


Fig. 2. Replication model.

Let's look at an example. Assume two transactions $T1$ and $T2$ start concurrently at the application server (Fig.1(b)). $T1$ wants to write x and y. For that, the application server reads the values of x and y into EBX and EBY, respectively, $T1$ gets locks and updates both beans. Concurrent transaction $T2$ reads z into EBZ, and then wants to read y but is blocked because $T1$ has a lock on EBY. Now $T1$ commits. The new values for x and y are written to the database and the locks are released. Both EBX and EBY remain cached. $T2$ receives the lock on EBY and reads the current value of EBY, namely the value written by $T1$. Now assume the cache replacement policy evicts EBX from the cache. Later, $T2$ wants to read x. x is reread into a new incarnation of EBX. However, since the database uses snapshot isolation and $T2$ is concurrent to $T1$ in the database, the old value of x is read. Therefore, $T2$ reads for y the value written by $T1$ but for x a previous value. This does neither conform to snapshot isolation nor to serializability.

In order to avoid the anomalies of J2EE caching when used with a snapshot isolation database, we propose a multi-version cache for EBs that enforces snapshot isolation. For each EB, instead of keeping a single copy in the cache, a list of potentially more than one version is cached. Each bean version EBX_i of data item x is tagged with the commit timestamp i of the transaction that created (and committed) this version. Additionally, the multi-version cache is replicated at several application servers in order to provide scalability, and availability. The semantics of the replicated multi-version cache is as if there was a single multi-version cache providing snapshot isolation (consistency).

3 Replication Model

We consider a vertical replication model in which a J2EE application server (AS) and a database (DB) are collocated in the same site [8]. 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).

In here, we are interested in container managed transactions, where each client request is automatically bracketed as a transaction by the application server. This means, there is a one-to-one relationship between requests and transactions. Clients call methods of a session bean, and the session bean may access other session or entity beans. A

client can be connected to any of the replicas. That replica will execute the transactions of the client. The changes of update transactions have to be executed at all replicas. Our protocol uses a group communication system [14] for communication among the replicas. The next section discusses the replication protocol when no failures occur. Section 5 discusses failure handling.

4 Replication Protocol

In this section, we describe the multi-version cache replication protocol (Fig.3). We first present the main ideas and then discuss the protocol in detail.

Overview

When a request is submitted to a replica, a transaction is started at the application server (AS) and at the database (DB). The transaction might read data that is already cached at the AS or that has to be read from the DB. The cache protocol makes sure that the correct version is read according to snapshot isolation. If the transaction is read-only, it simply commits locally and the result is returned to the client. If the transaction updates a data item x, a new private version of the corresponding entity bean EBX is created. At commit time, the replication protocol multicasts all EB versions created by the transaction (i.e. its *writeset*) using a total order multicast provided by the group communication system. That is, although different replicas might multicast at the same time, all replicas receive all writesets in the same order. At each replica, the replication protocol now validates incoming writesets in the same order. If validation determines that a concurrent transaction that already validated had an overlapping writeset, validation fails and the transaction is aborted. If validation succeeds, the replica assigns a commit timestamp, tags the EB versions of the transaction with the commit timestamp and adds them to the cache. Then the transaction commits at AS and DB. When the transaction commits at the local replica, the result is returned to the client. Each replica validates the same set of update transactions in the same order, and decides on the same outcome for each individual transaction. Thus, each committed transaction has the same commit timestamp at each replica.

Protocol Details

We now discuss in detail how transactions are executed. When a transaction is submitted to a replica R, the replica starts a local transaction T at the local AS and a transaction t at the local DB. The correlation between AS transactions and DB transactions is stored in a table (Fig.3 line 6). Each transaction T at the AS will be associated with a *start timestamp* $ST(T)$ when it starts (line 3), and a *commit timestamp* $CT(T)$ at commit time (line 54) which is assigned from a counter that is increased every time a transaction commits. The start timestamp $ST(T)$ of a transaction T is the highest commit timestamp $CT(T')$, and indicates that T should read the committed state that existed just after the commit of T'. We assume that the initial start timestamp is 0 at all replicas. Each bean version $EBX_{CT(T)}$ of a data item x is tagged with the commit timestamp $CT(T)$ of the transaction T that updated (and committed) this version. When a transaction T reads a data item x it has written before, it reads its own version (lines 10-11). Otherwise, it first looks for EBX in the cache. It reads the version EBX_i such that

$i \leq ST(T) \wedge \nexists EBX_j : i < j \leq ST(T)$, i.e. it reads the last committed version as of the time it starts (lines 12-13). If no appropriate bean is cached in memory, the transaction reads x from the DB and the corresponding EBX version is created (lines 15-18). Since a transaction is started at the DB when a transaction starts at the AS, and the DB provides snapshot isolation, the DB will return the correct version for x. 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 DB does not show the versions associated to tuples, when a version of data item x is read from the DB the corresponding tag of the EBX version is unknown. Thus, the bean is tagged with -1.

In order to guarantee that transactions always read the appropriate bean version, the cache guarantees that for each data item x the following holds: (i) If both T and T' updated x, and $CT(T') > CT(T)$ and the version $EBX_{CT(T)}$ is cached, then $EBX_{CT(T')}$ is also cached. That is, if the cache contains a certain version of a bean, then it also contains all later versions of this bean; (ii) If there exists a version $EBX_{CT(T)}$ and there exists an active local transaction T' that is concurrent to T, i.e., $ST(T') > CT(T)$, then $EBX_{CT(T)}$ is cached. That is, a version is cached at least as long as there exists a concurrent local transaction that has not yet terminated. Having all these versions cached is important for reads.

Note that our approach requires the DB to provide snapshot isolation. This is needed because the cache cannot keep the entire database, i.e., all versions of all tuples. To show that snapshot isolation is needed at the DB level, assume the DB uses the isolation level read committed (or serializability via locking). Assume further that a transaction T_i reads and modifies x while a concurrent transaction T_j updates y and commits. Assume now further that T_i wants to read y and, due to lack of memory, the version of EBY that T_i needs to read, was evicted from the cache. Hence, it has to read it from the DB. Assuming transactions run at the DB with read committed (or serializability) isolation level, T_i reads the value of y committed by T_j. That is, it will not read the value of EBY at the time T_i started. Thus, the AS cannot provide by itself the snapshot isolation level.

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. When a transaction T_i wants to update a data item x, it has to first acquire a write lock on EBX (line 25). Write locks guarantee that at most one transaction updates data item x at any time. If another transaction holds a lock on EBX, T has to wait until the lock is released which is done at transaction abort (line 45) or commit (line 73). Lock requests are inserted into a FIFO wait-queue, i.e., when a transaction releases a lock the first in the wait queue receives it. Once a transaction T has a lock, a version check on the version EBX_j with the highest version number in the cache is performed. If $j > ST(T)$, then this version was created by a concurrent, already committed transaction, and T must abort (lines 26-27). If no such version exists, T can perform the update, i.e., create its own version and add it to its writeset (line 29-31). For now, this version is only seen by T itself. This guarantees that a transaction observes its own updates (lines 10-11) and prevents other transactions from observing uncommitted changes.

When a transaction wants to commit, if the transaction was read-only it is simply committed at the DB (lines 36-37). Otherwise, the writeset is multicast to all replicas

```
Data:
timestamp = 0;
cache = ∅ ;
committedTx = ∅;
transactionTable = ∅;
mutex;;
oldestActiveTx = array[1..NumberReplicas] of Int = 0;
1  begin(T)
2      set mutex ;
3      ST(T) = timestamp;
4      WS(T) = ∅;
5      t = begin transaction in the DB;
6      store(transactionTable, T, t);
7      release mutex ;
8  end
9  read(T, EBX)
10     if EBX_Tprivate ∈ WS(T) then
11         return EBX_Tprivate ;
12     else if
       ∃EBX_i ∈ cache : i = max(j) | EBX_j ∈ cache ∧ j < ST(T)
       then
13         return EBX_i ;
14     else
15         t = getTx(transactionTable, T);
16         EBX_-1 = read(t,EBX) from the DB;
17         cache = cache ∪{EBX_-1};
18         return EBX_-1;
19     end
20 end
21 write(T, EBX, value)
22     if ∃EBX_Tprivate ∈ WS(T) then
23         write(EBX_Tprivate, value);
24     else
25         acquire lock on EBX for T;
26         if ∃EBX_i ∈ cache | i > ST(T) then
27             abort(T);
28         else
29             create(EBX_Tprivate);
30             WS(T) = WS(T) ∪{EBX_Tprivate};
31             write(EBX_Tprivate, value);
32         end
33     end
34 end
35 commit(T)
36     if WS(T) == ∅ then
37         Commit (getTx(transactionTable, T)) in DB;
38     else
39         multicast(WS(T), T, minLocalTx(transactionTable));
40     end
41 end
42 abort(T)
43     ∀ EBX_Tprivate ∈ WS(T) do
44         delete(EBX_Tprivate);
45         release lock on EBX;
46     end
47     abort getTx(T) in DB;
48     delete(transactionTable, T);
49 end
50 upon delivery of (WS(T), T, oldestLocalActiveTx)
51     set mutex;
52     oldestActiveTx[Sender(T)] = oldestLocalActiveTx;
53     if ∄T_K ∈ committedTx : ST(T) > CT(T_k)∧
       WS(T) ∩ WS(T_K) ≠ ∅ then
54         CT(T) = ++timestamp;
55         if local(T) then
56             ∀EBX_Tprivate ∈ WS(T) do
57                 replace tag Tprivate with tag CT(T);
58                 cache = cache ∪{EBX_CT(T)};
59             end
60         else
61             t = begin transaction in the DB;
62             store(transactionTable, T, t);
63             ∀EBX_Tprivate ∈ WS(T) do
64                 if ∃ local transaction LT that has lock on EBX then
65                     abort(LT)
66                 end
67                 acquire lock on EBX for T (put lock request at begin of
                   wait queue);
68                 replace tag Tprivate with tag CT(T);
69                 cache = cache ∪{EBX_CT(T)};
70             end
71         end
72         commit (getTx(transactionTable, T)) in the DB;
73         ∀EBX ∈ WS(T) release lock on EBX;
74         committedTx = committedTx ∪{T};
75         delete(transactionTable, T));
76     release mutex;
77 end
78 garbageCollection()
79     oldestTx = min(oldestActiveTx);
80     ∀T ∈ committedTx do
81         if CT(T) < ST(oldestTx) then
82             committedTx = committedTx - {T} ;
83         end
84     end
85     oldestLocalTx = oldestActiveTx[R];
86     ∀EBX ∈ cache do
87         if ∃EBX_i ∈ cache ∧ i ≠ −1 ∧ i < ST(oldestLocalTx) then
88             cache = cache - EBX_i;
89             if EBX_-1 ∈ cache then cache = cache-EBX_-1;
90         end
91     end
92 end
```

Fig. 3. Replicated Cache Protocol for Replica R

using a total order multicast (line 39). All replicas in the multicast group (including senders) receive all messages in the same order. When a replica processes such a message (line 50), the corresponding transaction performs a final validation (line 53). This will help to find conflicts among transactions that executed at different replicas. Since all transactions perform deterministic validation in the same order, all replicas decide on the same outcome. A transaction passes validation, if there is no transaction in the system that is concurrent, already committed and has overlapping changes.

When a transaction T passes validation, it receives a commit timestamp (line 54). At the local replica, the private versions are tagged with the commit timestamp and added to the cache (lines 55-59). T and the corresponding DB transaction t commit and the

locks are released (lines 72-73). Note that committing the DB transaction automatically propagates the changes to the DB. The protocol keeps track of all committed transactions (line 74) for validation purposes. At a remote replica, a DB transaction is started for T (lines 61-62). T first gets the locks on the data items. If a local transaction T' has a lock on one of the data items it has to abort because it is concurrent to T, has updated the same data item and is not yet validated and committed (lines 64-65). T has to be the first to get the lock (line 67). Then, the versions sent in the message are tagged with the commit timestamp and added to the cache (lines 68-69). From there, the transaction commits as in the local replica (lines 72-75).

If a transaction T does not pass validation, nothing has to be done. At remote replicas, the message can simply be discarded since nothing has yet been done on behalf of T. At the local replica, T can only fail validation if a conflicting remote transaction T' was received between sending and receiving T. In this case, however, as described above T' found the lock held by T and T was already forced to abort.

Note that messages are processed serially, that is, one after the other, in order to guarantee that validation and commit order are the same at all replicas. Furthermore, starting transactions have to be coordinated with committing transactions in order to guarantee that transactions see, in fact, the correct snapshot. Therefore, an appropriate mutex is set (lines 2,7, 51 and 76).

Examples

We illustrate the execution along two examples. Fig.4 shows an example of the evolution of the cache on a single replica (ignoring the replication part). We assume the cache is empty and the commit counter is at value 10. Transactions $T1$ and $T2$ obtain the same start timestamp (10) and each creates a corresponding DB transaction. Then, $T2$ reads x. Since no bean version exists in the cache, the data item is read from the DB and a version EBX_{-1} is created. The value of x is a. Now $T1$ reads x and y. Since EBX_{-1} is cached and $-1 \leq 10$, $T2$ reads EBX_{-1}. Furthermore y is read from the DB and stored in EBY_{-1}. Its current value is b. Now $T1$ updates EBX to the value c and EBY to the value d. For that, it creates private versions of EBX and EBY. Finally, it requests the commit. It receives commit timestamp $CT(T1) = 11$, the versions are tagged with this timestamp and added to the cache. The corresponding DB transaction commits meaning that the changes are transferred to the DB. Since the DB implements SI, new versions for both x and y are created also in the DB. When $T2$ now reads y, it does not read EBY_{11}, since $11 > ST(T2)$. Instead, it reads EBY_{-1} that is, the old value b of y. Since $T2$ is read-only, it simply commits in the DB and no commit timestamp is assigned.

In our second example (Fig. 5) we assume two replicas $R1$ and $R2$. We assume the commit counter at each replica is 10 when transaction $T1$ starts at $R1$ and $T2$ at $R2$. Both receive start timestamp 10. Now assume both read data item x, reading it from the local DB and loading it into EBX_{-1}. The current value is a. Now both transactions update EBX. Since they run in different replicas, both acquire the lock, and create their own private EBX versions. $T1$ sets the new value b, $T2$ sets the new value c. When $T1$ and $T2$ 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. Let's first have a look at $R1$. When $T1$ is delivered at $R1$, its validation succeeds. $T1$ is local at

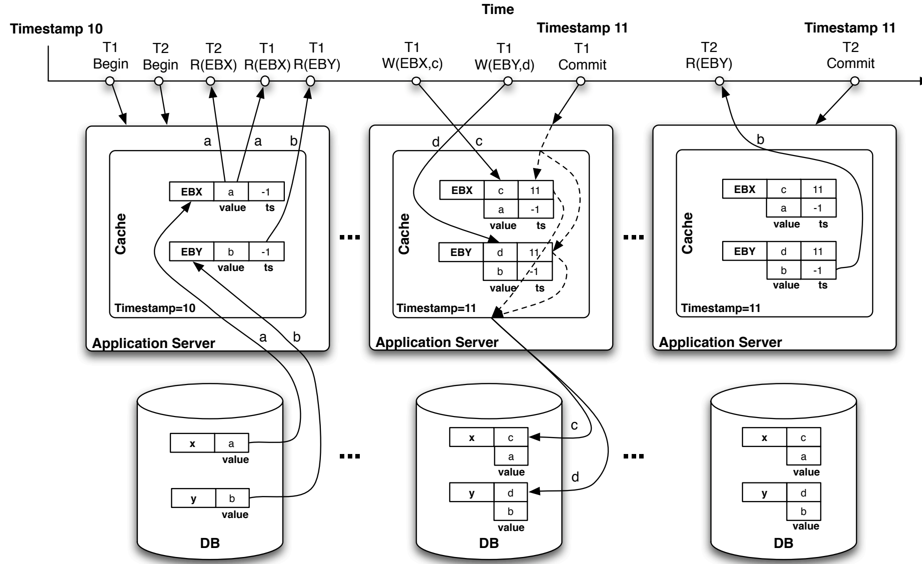


Fig. 4. Evolution of the cache in a single replica.

$R1$. It receives the $CT(T1) = 11$ and its version is tagged ($EBX_{11} = b$) and added to the cache. $T1$ commits at the DB. When now $T2$ is delivered at $R1$ validation fails since $T1$ is concurrent ($CT(T1) > ST(T2)$), conflicts, and has already committed. Therefore, nothing is done with $T2$ at $R1$. At replica $R2$ transactions are validated in the same order. $T1$'s validation succeeds. $T1$ is a remote transaction at $R2$ and has to acquire the locks. However, $T2$ has a lock on EBX. $T2$ is local and has not yet validated. Therefore, it is aborted, its private version discarded and its lock released. $EBX_{11} = b$ is created and added to the cache. The value is propagated to the DB and the transaction committed. When later $T2$ is delivered at $R2$, validation fails. The transaction has already aborted, and nothing has to be done. Therefore, the two replicas commit the same transactions and keep the same values (with the same version tag) in both the cache and the DB.

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 (no corresponding data item exists in the DB), 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 DB.

Deletions create a tombstone version of the EB. The tombstone is also a private version of the transaction until 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

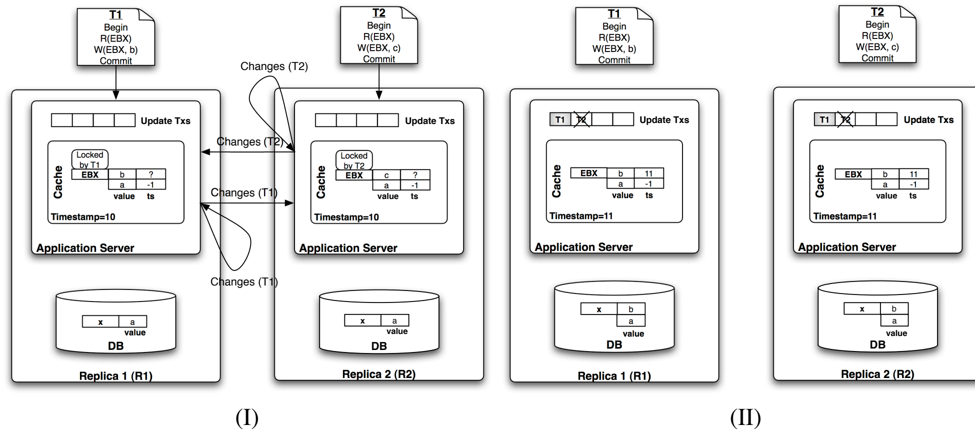


Fig. 5. Two concurrent conflicting transactions

deleted. When the transaction commits, the tombstone version will become public. Even after transaction commit previous versions of the EB cannot be removed, since there might be active transactions associated to older snapshots (all transactions that started before the one that deleted the EB committed), that may read the older 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 garbageCollection). Each replica removes versions that are older than the oldest start timestamp among local active transactions (lines 85-88). If a version of an EB, EB_i, is not needed (there is no local active transaction with start timestamp smaller than i), then version EB_{-1} is also not needed, since EB_{-1} is older (line 89). Moreover, if EB_{-1} is not evicted from the cache and a new transaction T ($ST(T) > i$) reads EB, it would read EB_{-1}, which is incorrect, since it should read a later committed version (EB_i or even a later version, since $ST(T) > i$).

Uncommitted updated EBs are pinned in the cache. If the cache gets full with pinned EBs, the J2EE application server writes locked 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 AS brings the hibernated EB to memory including all EB versions and their tags. Note that updated EBs whose changes have been committed will be evicted from the cache according to the cache policy.

Writesets ($committedTx$) are also garbage collected. Since they are used for validation, a writeset can only be garbage collected when there are no more active concurrent transactions in the system (lines 79-82).

Session Replication. Stateful session beans (SFSBs) keep conversational state from a client and their replication is not required to provide consistency and availability of

EBs. 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 replicates the state of SFSBs after each method invocation. Papers [15, 7] focus on this topic.

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, the necessary replication logic can be incorporated in a way fully transparent to clients. We have extended the stubs to be able to perform replica discovery and load-balancing, relying on IP-multicast. When a client wants to connect, the stub IP-multicasts a message to an IP-multicast address associated to the application server cluster. Clients are identified by a unique client identifier. When the replicas in the cluster receive a connection request, one of them (depending on the client identifier) returns to the stub a list of available replicas (their IPs) as well as an indication of their current load. Replicas multicast information about their load periodically (e.g., piggybacked on the writeset message). The stub then selects a replica randomly with a selection probability inversely proportional to the load of the replicas to attain load balancing. The stub connects to the selected replica and sends all client requests to this replica (sticky client). Each request receives a unique number (a counter kept at the stub that is incremented after each successful request).

The AS replicas build a group using the group communication service. The group communication system provides the notion of view (currently connected cluster members). Whenever a member fails, the available members are informed via a view change message. The group communication system provides strong virtual synchrony that guarantees that the relative order of delivering view changes and multicast messages is the same at all replicas. The total order multicast used for the writeset messages in the replication protocol also provides reliable delivery guaranteeing that all available replicas receive the same set of messages [14]. Furthermore, the writeset also contains the client identifier, request identifier plus the response that is going to be returned to the client. The remote replicas store for each client the latest request identifier, the outcome of the transaction (commit/abort), and in case of commit, the response.

Let us now consider the failover logic at the application server side. Each replica consists of a pair of AS and DB. If any of them fails or the site in which they are collocated fails, the replica is considered as failed. If only the AS or the DB fails, the other component automatically shuts down. We assume only crash failures.

At the client side the failure will be detected when the stub times out waiting for the response to an outstanding request. The stub will reconnect to a new replica and resubmit that request. Notice that we are considering container managed transactions, where each client request will be automatically bracketed as a transaction. Thus, there is a one-to-one relationship between requests and transactions. A failure can now occur

at two logical timepoints. (1) The replica failed before multicasting the writeset related to the request to the other replicas. (2) The replica failed after multicasting the writeset.

If there have been previous interactions with that client, the new replica to which the stub connects to will have the last state of the stateful session bean (SFSB) associated to the client and processes the resubmitted client request in the following way. In case (1), the new replica does not yet have any information about this request and thus, will process it as a new request. In case (2) it has already stored the request identifier and the outcome of the corresponding transaction. It recognizes the resubmitted request as a duplicate for which it has already the outcome stored. If the outcome was commit, it returns the response to the client. Otherwise, it returns an exception to the client notifying that the transaction was aborted since snapshot isolation could not be guaranteed (the failed replica would have done the same if it had not failed).

6 Evaluation

6.1 Evaluation Setup

The evaluation has been performed in a cluster of 10 machines 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 a PostgreSQL v.8.2 database. JGroups [16] is used as group communication system.

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 [12]. In this application there is a workload generator (driver) that emulates automobile dealers interacting with the system through HTTP. The driver injects three different 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 writes, 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 $\times$ 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).

We compare the results of our replicated multi-version cache with the traditional caching of JOnAS (no replication) and a replicated application server (JOnAS) with 2 replicas sharing a single database (horizontal replication) where only stateful session beans are replicated.

6.2 SPECjAppServer Benchmark Results

Fig. 6(a) shows the overall throughput with increasing loads. The figure shows graphs for traditional caching without replication, horizontal replication with 2 replicas (HR Shared DB) and our approach for 1-10 replicas. The first noticeable fact is that traditional caching and horizontal replication can only handle a load up to 3 Ir. In contrast,

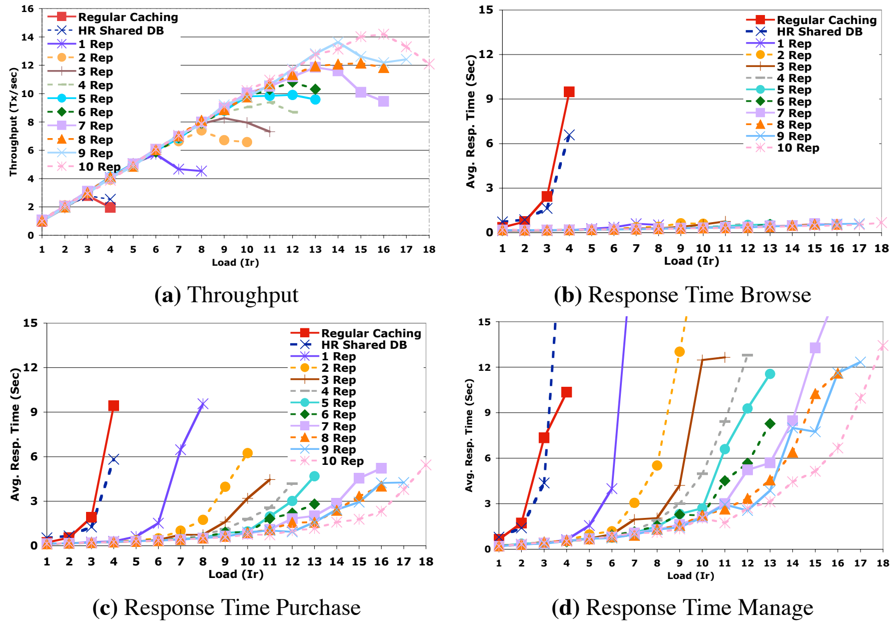


Fig. 6. SPECjAppServer Results

our replicated multi-version cache outperforms these two implementations by a factor of 2, even if there is only one replica. The reason is that the multi-version cache is able to avoid many database reads compared to regular caching. Horizontal replication did not help because the shared database was already saturated with two application server replicas. With 3 replicas (not shown), the system deteriorated and did not even achieve 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 load of 6 Ir and throughput of 6 Tx/sec with a single replica (and 3 Ir resp. 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. By increasing the 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 horizontal replication.

Even when the replicated cache configurations saturate (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 with 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.

Fig. 6(b-d) show the response time for browse, purchase and management transactions with increasing load. Interestingly, browse transactions (i.e., read-only transactions) are not affected by the saturation of update transactions. As can be seen in Fig. 6(b) the response time graphs are almost flat independently of the number of replicas even at high loads 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 that both regular caching and horizontal replication saturate with Ir = 3, since the response times increase exponentially for browse transactions.

Purchase transactions (Fig. 6(c)) are quite different since they are update transactions. The response time for all configurations reaches saturation (it grows exponentially) 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 provides low response times until saturation is reached. Finally, the more replicas the system has, the more graceful is the degradation of the response time at the saturation point. This is important since acceptable response times can be provided in case of short-lived peaks.

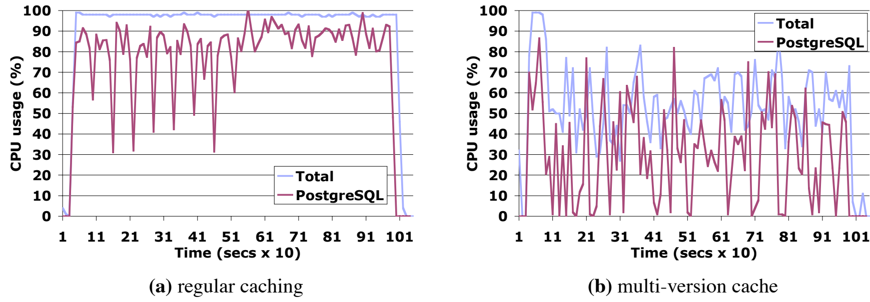
The different behavior of the purchase transactions compared to browse transactions has to do with the fact that update transactions propagate their changes to all the replicas in the system, and also have to write changes to the database. Thus, a higher overhead is created leading to worse response time. This behavior is even more noticeable in the case of manage transactions, which have the highest percentage of updates (Fig. 6(d)). 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. 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.

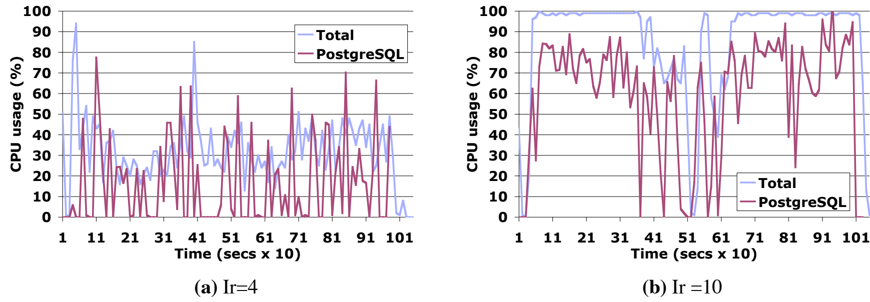
The results for regular caching and our multi-version cache with a single replica for Ir = 4 are shown in Fig. 7. At this load, the system is saturated with a 100% usage of the CPU with 1 replica and regular caching (Fig. 7(a)). The database consumes most of the CPU. There are depressions in the utilization 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 underusing the CPU. The single replica multi-version cache configuration shows a significantly smaller CPU usage (Fig. 7(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 for Ir = 4.

Examining the 2-replica configuration of our replicated cache for Ir = 4 (Fig. 8(a)), the results are quite different. Although there are some high peaks in the CPU usage, the area covered is much smaller than for the 1-replica configuration. The overall CPU



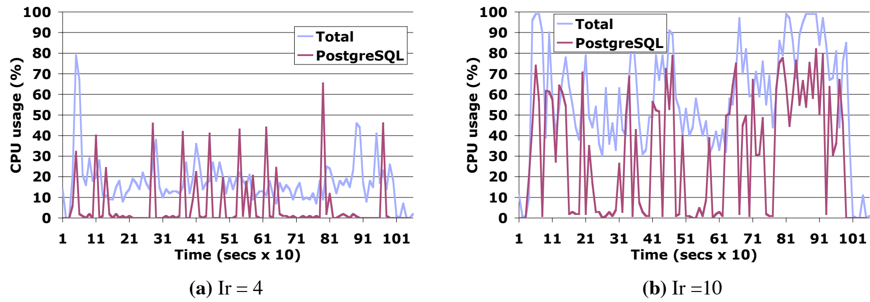
(a) regular caching **(b)** multi-version cache

Fig. 7. CPU usage: One replica, Ir = 4



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

Fig. 8. Multi-version cache. CPU usage: Two replicas



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

Fig. 9. Multi-version cache. 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. 9(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 the load is 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. 8(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% which means that the database is the bottleneck. The 6-replica configuration is not saturated in this setting (Fig. 9(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, which 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 kind of server without replica configurations. The operating system takes care of distributing CPU to the servers according to 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 that read only transactions (50% of the load) are not affected by this overhead. The multi-version cache compensates the replication overhead by improving the caching efficiency and reducing the database access (rows at the bottom) in a 27.6% and 51.5%, respectively.

6.5 Scalability Analysis

In this section we measure the scalability of the replicated cache, i.e., how much we can increase the load when increasing the number of replicas. To measure the scalability we take the response time (RT) threshold of the SPECjAppServer benchmark, 2 seconds, and observe for each configuration (i.e. number of replicas) the maximum load (Ir) for which the response time remained below the 2-second threshold. Additionally, in order to observe the behavior under peak loads, we have also measured the maximum load for a 5-second threshold.

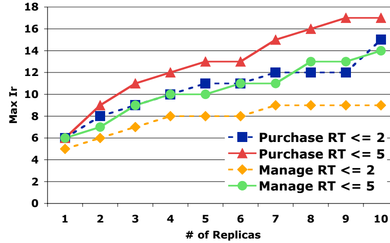


Fig. 10. Scalability Analisys

Fig. 10 shows the scalability results. For browse transactions we do not show any graphs since for all tested replica configurations and injected Ir the response time was well below 2 seconds. For purchase transactions, we can see that for small configurations the sustainable load increases sharply when increasing the number of replicas, while it only increases slightly when there are already many replicas in the system. This means, 5 replicas are able to manage a total of 110 clients (Ir = 11), that is, an average of 110/5=22 clients per replica. 10 replicas manage 150 clients (Ir = 15), that is, an average of 15 clients per replica. Still, scalability is considerably good considering the substantial fraction of updates involved in purchase transactions.

For manage transactions the system does not scale as well as for purchase transactions. This is expected since manage transactions have a higher percentage of updates resulting in a higher replication overhead. For this kind of transactions the system does not scale beyond 6 replicas for the 2-second threshold. However, if we look at the tolerance to peak loads (threshold RT $<=$ 5 secs) having additional replicas is beneficial. Manage transactions with a threshold RT$<=$5 scale almost as well as purchase transactions. That is having 10 replicas, the system can still provide reasonable response time (below 5 seconds) at high loads, while this is not the case for 6 replicas.

7 Related Work

Early work in application server replication looked mainly at CORBA and focused on fault-tolerance [9]. This work resulted in the FT-CORBA specification [17], 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 addressed in [10].

[15] 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. [7] is also a primary-backup approach for the replication of J2EE servers supporting multiple transactional patterns (e.g. several client requests may be encapsulated within one server transaction or a single client request can initiate several server transactions). [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. At commit time, every

read entity bean is re-read to check whether it was modified. This approach has the shortcoming of all horizontal approaches since the shared database becomes the bottleneck. The certification is heavier than the validation of our replication protocol since it has to re-read every read entity bean.

On the theoretical side, papers [5, 4] 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. [6] applies this technique in a J2EE environment.

In contrast to aforementioned approaches, our proposed replicated cache provides both scalability and availability and avoids that the shared database becomes a single point of failure and a bottleneck.

[18] 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 a high level of consistency via snapshot isolation and it is a real implementation evaluated with an industrial benchmark.

[19] 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 [20], Oracle9iAS [21], WebLogic clustering [22] and WebSphere 6.0 [23]. 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 database [24]. 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 resulting in a very heavy-weight 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 the use of off-the-shelf databases. The replicated multi-version cache scales even for update workloads, and takes advantage of modern snapshot-isolation databases such as Oracle and PostgreSQL. The implementation is based on a commercial J2EE application server, JOnAS. A 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: International Conference on Distributed Systems (ICDCS). (2003)
3. Bull: JOnAS Clustering, https://wiki.objectweb.org/jonas/Wiki.jsp?page=JOnASClustering
4. Frølund, S., Guerraoui, R.: Implementing e-transactions with asynchronous replication. IEEE Trans. Parallel Distributed Systems **12**(2) (2001) 133–146
5. Frølund, S., Guerraoui, R.: e-transactions: End-to-end reliability for three-tier architectures. IEEE Trans. Software Engineering **28**(4) (2002) 378–395
6. 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
7. Wu, H., Kemme, B.: Fault-tolerance for stateful application servers in the presence of advanced transactions patterns. In: Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS). (2005) 95–108
8. Kemme, B., Jimenez, R., Patiño, M., Salas, J.: Exactly once interaction in a multi-tier architecture. In: VLDB DIDDR Workshop. (2005)
9. Narasimhan, P., Moser, L.E., Melliar-Smith, P.M.: Eternal - a component-based framework for transparent fault-tolerant CORBA. Software: Practice and Experience **32**(8) (2002)
10. Zhao, W., Moser, L.E., Melliar-Smith, P.M.: Unification of Transactions and Replication in Three-Tier Architectures Based on CORBA . IEEE Transactions on Dependable and Secure Computing **2**(1) (2005) 20–33
11. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: ACM SIGMOD Conference. (1995) 1–10
12. SPEC: SPECjAppServer 2004 Benchmark, http://www.spec.org/jAppServer/. (2004)
13. Sun Microsystems: Java 2 Platform Enterprise Edition v1.4. (2003)
14. Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: A comprehensive study. ACM Computer Surveys **33**(4) (2001)
15. Perez-Sorrosal, F., Patiño-Martínez, M., Jiménez-Peris, R., Vuckovic, J.: Highly Available Long Running Transactions and Activities for J2EE Applications. In: Proc. of the IEEE Int. Conf. on Distributed Computing Systems (ICDCS). (2006)
16. JGroups: A Toolkit for Reliable Multicast Communication http://www.jgroups.org.
17. OMG: Fault Tolerant CORBA. Object Management Group (2000)
18. 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
19. Attar, M., Ozsu, M.T.: Alternative architectures and protocols for providing strong consistency in dynamic web applications. WWW Journal **9**(3) (2006) 215–251
20. The JBoss Group: JBoss Application Server. http://www.jboss.org.
21. Oracle: Oracle9iAS Containers for J2EE. EJBs Developers Guide, Rel. 2 (9.0.4) (2003)
22. BEA Systems: WebLogic Server 7.0. Programming WebLogic Enterprise JavaBeans . (2005)
23. IBM: WebSphere 6 Application Server Network Deployment. (2005)
24. The JBoss Group: JBoss Cache. http://labs.jboss.com/jbosscache/.