Eager Replication for Stateful J2EE Servers

Huaigu Wu[1], Bettina Kemme[1], and Vance Maverick[2]

[1] McGill University, Montreal, Canada
[2] Universitá di Bologna, Bologna, Italy

Abstract. Replication has been widely used in J2EE servers for reliability and scalability. There are two properties which are important for a stateful J2EE application server. Firstly, the state of the server and the state of the backend databases should always be consistent. Secondly, each request from a client should be executed exactly once. In this paper, we propose a replication algorithm that provides both properties. We use passive replication where a primary server executes a request, and all state changed within the application server by this request is sent to the backup replicas at the end of the execution. An agreement protocol guarantees the consistency between the state of all replicas and the database. A client side communication stub automatically resubmits requests in case of failures, and unnecessary resubmissions are detected by the server replicas. We have implemented the algorithm and integrated it into the JBoss application server. A performance study using the ECPerf benchmark shows the feasibility of our approach.

Keywords: stateful J2EE servers, transactions, eager replication, state consistency, exactly-once-execution

1 Introduction and Motivation

In recent years, the J2EE architecture has become popular for web-based applications and web services, providing services to manage transactions, persistence, security, and object life-cycles. In such architecture, clients are thin (e.g., web browsers). The front-end is a web server providing the presentation logic, the application server implements the business logic, and the database server manages persistent data. Web and application servers can also maintain data. This data, although non-persistent, can exist across several client requests. A typical example is session information. We say that such servers are stateful.

Important requirements for web-based applications are fault tolerance and high availability. Replication is an essential technique to achieve these goals. Typically, replication means that several instances of a server are started. If one instance crashes, the others can continue to work. The different tiers of the multi-tier architecture need different replication strategies. In this paper we examine the particular design problems of replicated J2EE application servers in detail.

The main challenge is to combine transactions, as supported by J2EE, with replication. In J2EE, most of the execution is performed within the boundaries of a transaction which can change the state of both the stateful application

R. Meersman, Z. Tari (Eds.): CoopIS/DOA/ODBASE 2004, LNCS 3291, pp. 1376–1394, 2004.

server and the persistent database. An important requirement in this context is *state consistency*. If a transaction commits, both the state of the application server and the database should be changed accordingly. If it aborts, none of the changes should remain. Database systems provide efficient abort mechanisms. Within J2EE, application programmers can specify a *compensation* method for each business method that will be called by the server in case of an abort, undoing the state changes performed within the application server. Another property of J2EE is *at-most-once* execution guarantee. If a client sends a request and receives a response, the request is executed exactly once. If the client does not receive a response, it does not know whether the operation has completed or not.

If we now replicate application servers for fault-tolerance, we have to keep transactions in mind. In particular, we want our replication algorithm to have two important properties. (i) State consistency must be maintained. That is, if a transaction commits at the database, each available replica of the application server has the state changes performed by this transaction. If the transaction aborts at the database, all available replicas know about the abort and are able to return to the previous state. (ii) The system should provide *exactly-once* transactions, i.e., as long as the client does not crash, a transaction is executed exactly once (if the client crashes the semantics should be *at-most-once*). That is, even if a replica crashes during the execution of a transaction, the others will continue and terminate the transaction as if no crash had happened.

In this paper, we propose a simple but effective and efficient replication algorithm providing state consistency and exactly-once transactions for application servers following the J2EE specification. The approach uses passive replication, where within a client session one server replica is the primary for this client while the others are backups. Only the primary executes the requests. Before a transaction commits, the state changes within the application server are multicast to the backups. State consistency is guaranteed by a special agreement protocol using a marker mechanism similar to [11]. It differs from traditional agreement protocols like 2-phase-commit since it has a highly reduced logging cost, does not require all participants to have executed the request before terminating, and relies on efficient and reliable group communication primitives. If the primary fails before the client disconnects, a backup takes over the client session. Exactly-once semantic is provided by a special client stub that automatically resends an outstanding request to the new primary if it suspects the old primary to have failed. Resubmissions of successfully executed requests are detected and lead to a return of the result without reexecution. We follow similar mechanisms as exploited in [5]. While the individual mechanisms used have already been proposed before, we are not aware of any solution that combines all, and applies them to the J2EE environment. We have implemented and integrated our approach into JBoss [13], an open-source J2EE compliant application server. Extensive performance evaluations show that the induced overhead is acceptable and compares favorably with that of other fault-tolerant solutions (e.g., JBoss's own replication solution [14], or Eternal, a fault-tolerance framework for CORBA [19,20]).

The rest of the paper is organized as follows. Section 2 introduces some background and assumptions. Section 3 presents the replication algorithm. Section 4

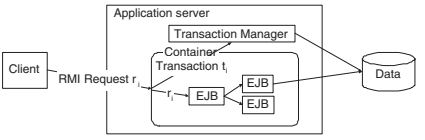


Fig. 1. Typical Execution Flow of a J2EE server

describes the implementation. Section 5 provides a performance evaluation. Section 6 discusses related works. Finally, Section 7 concludes the paper.

2 Background, Model, and Assumptions

We assume that the nodes running application servers can fail by crashing (no byzantine failures). We assume reliable, asynchronous communication and no network partitions (which are briefly discussed in Section 7). Furthermore, we assume both clients and database do not crash and the connection to the database is reliable. Client failures are easy to handle. Kistijantoro et al. [17] present how databases can be replicated to handle database crashes, and how this can be integrated into a J2EE application server. Other database replication mechanisms have been proposed in recent years [16,15,1,8]. We are currently extending our system to work with the replicated database system presented in [15].

2.1 J2EE Application Server

Basic architecture. In a J2EE application server [23], the business logic is implemented using special Java objects called *Enterprise JavaBeans (EJB)*. EJBs are also, in a more general way, referred to as *components*. We distinguish two categories: *session beans* (SB) and *entity bean* (EB)[1]. A session bean is a non-persistent object that represents the actions associated with a caller session. There are two subtypes. *Stateless session beans* (SLSB) do not maintain any internal state across method calls. *Stateful session beans* (SFSB) maintain internal state for the lifetime of a caller session. In contrast, an entity bean is an object that represents persistent data in persistent storage (mostly database system).

A J2EE server provides a set of services, including transaction, persistence, and JDBC services. Business logic can be executed within the context of transactions. Transactions can be bean managed (developers explicitly set *begin/commit/ abort* statements) or container managed. For container managed transactions, the simplest way is that the container automatically creates one transaction per client request. That is, it demarcates a client request with the corresponding transaction statements as shown in Figure 1. When a client request enters the server, the container intercepts it, and sends a *begin* request to the transaction manager before the original request is forwarded to the destination EJB. If the EJB creates sub-requests to other EJBs their execution will run within the same transaction. When execution finishes just before the

[1] Message beans are a third kind of EJBs. They are outside the scope of this paper.

response is returned to the client, the container sends a *commit* request to the transaction manager. Any changed state within an EB will be written back to the database. The state of SFSB remains volatile. J2EE provides concurrency control for EBs to avoid concurrent transactions to access the same bean. If a transaction accesses more than one database, the transaction manager will trigger a 2-phase-commit protocol.

Assumptions. In regard to the application server, this paper makes a couple of assumptions. (i) We assume both clients and EJBs (except of SLSB) to be single-threaded with blocking requests (the caller of an EJB is blocked until it receives a reply). This is a typical programming model, and hence, we believe it is not a severe restriction. Consequently, SFSBs do not need any concurrency control since they are associated with a single client with at most one outstanding request. (ii) We assume that the execution of a client request spans exactly one transaction. This is the simplest and most popular model used in practice. More complicated models often already violate correctness in centralized systems without failure, making reasoning about correctness in a replicated environment difficult. We will discuss other models briefly in Section 7. (iii) We assume that a regular J2EE server (without replication) correctly handles state consistency and at-most-once execution in the failure-free case[2]. (iv) For space reasons, we do not discuss the case where a transaction accesses more than one database.

Furthermore, we rely on the following, standard behavior of J2EE application servers and database systems. (v) If the J2EE server crashes, the J2EE server's state and its connections to the database will be lost, and the database aborts all active transactions. That is, the database contains the changes of all committed transactions but no changes of aborted transactions or transactions active at the time of the crash. (vi) We say that if a request is correctly executed, its response is a *positive response*. This includes an abort exception if a transaction does not succeed (e.g., due to application semantics) but no machine crashes. If a request fails due to a server crash, the response will be an appropriate exception, referred to as *negative response*. J2EE guarantees that the client will eventually receive either a positive or negative response (unless the client fails).

2.2 Group Communication

We have chosen to use a *group communication system* (GCS) [10] as the communication channel. In a GCS, a group consists of a set of members. In our case the application server replicas are the members. A member can multicast a message to all group members (including itself) or send point-to-point messages. Messages are guaranteed to be delivered if no failures occur, otherwise at-most-once. Different multicast protocols provide different ordering and reliability properties. *Reliable delivery* guarantees that when a member receives a message and does not fail for sufficiently long time then all members receive the message unless they fail. *Uniform-reliable delivery* is stronger in the sense that if any member receives a message (even if it fails immediately afterwards) all

[2] The J2EE specification defines the transaction synchronization interface enabling programmers to provide rollback methods, and we assume that they are in place.

members receive the message unless they fail. In our context, we are interested in FIFO ordered messages (messages of the same sender are received in sending order), and in totally ordered messages (if two members receive messages m and m', they both receive them in the same order). GCS also provides group membership service, maintaining a view of the currently connected members. Whenever the view configuration changes, the GCS informs all member applications by delivering a view change message with the new view. The typical property for group membership is *virtual synchrony*: If members p and q receive both first view V and then V', they receive the same set of messages while members of V. The GCS provides explicit join and leave primitives, and automatically removes crashed members from the view using a failure detection mechanism. Due to the asynchrony of the network, the GCS might exclude a correct member. In this case, we assume that the affected application server replica shuts down itself and attempts to rejoin the group.

3 Replication Algorithm

Replication mechanisms can be categorized by three parameters. (i) Replication can be either *active* or *passive*. In an active scheme, a request is sent to and executed at all replicas. The client receives a response as long as one replica is available (duplicate suppression must be in place). In passive replication, only the primary replica executes the request, and propagates updated state to the backup replicas. If the primary fails, failover takes place, and one of the backups becomes the new primary. Active replication requires deterministic behavior, and induces heavy load, since all replicas have to execute all requests. Passive replication allows for non-determinism. Although primarily designed for fault-tolerance, it has some potential for scalability, since applying changes sent from the primary is usually less time consuming than executing the requests themselves. The spared resources can be used to perform other tasks. For instance, each replica could be primary for a subset of requests, and backup for the others. On the negative site, passive replication requires complex state propagation and failover. (ii) The primary can propagate state to the backups in different ways. Using *cold* propagation, the primary stores the state information on persistent storage which can be accessed at failover by the new primary. In this case, the new primary can actually be initiated only when needed after a crash. Using *warm* propagation, the primary sends the state to the backups directly, e.g., via messages. This alleviates the load on the database but introduces message overhead. Backup instances must exist but failover is usually faster than in case of cold propagation. (iii) The *propagation time* defines when state propagation takes place. Using *eager* propagation the state is propagated some time before the response is returned to the user, in *lazy* propagation, only some time after. Eager replication increases user response time but can guarantee consistency. Lazy replication provides fast response but consistency might be lost if the primary crashes after a response is returned but before state propagation.

Our replication algorithm uses passive replication to allow non-deterministic execution, and to avoid redundant computation. For simplicity, we only present a

system with one primary site[3]. For EBs, we choose cold replication since changes are always written to the database at commit time by default. SFSBs use warm replication to achieve faster failover, and alleviate the load on the database. Since our main goal is state consistency, we use eager replication. In order to achieve exactly-once, requests are automatically resent upon a negative response. At the servers, duplicate execution is avoided by keeping track of requests and their responses. In order to guarantee state consistency and keep message overhead low, the primary sends all changes performed by a transaction on SFSBs in a single message to the backups before it commits the database transaction. Additionally, the primary enters a marker into the database as part of the transaction. This is necessary since the primary might crash after sending the message but before database commit. In case of failover after the primary crashes, the new primary can determine for each state change message it received before the crash whether the corresponding transaction has committed at the database by checking whether its marker exists in the database or not, and then act accordingly.

In the following we describe the algorithm in more detail by splitting it into five parts. One at the client, one at the primary and one at the backup during normal processing, one at the new primary at time of failover when the old primary crashes, and one for recovery when a new replica joins the group.

3.1 Client Algorithm

The client algorithm runs at the Client Replication Manager (CRM) located at the client side. The CRM maintains a list of server replicas, and a pointer to the current primary[4]. When a client submits a request, the CRM intercepts it, generates a unique request id *rid*, and forwards the call to the primary (including the *rid*). A positive response is returned to the client. If the response is negative, the CRM has to find the new primary. For that purpose, we assume that each server exports a method `is_primary` which returns true if the server is currently primary. Hence, the CRM goes through its server list calling `is_primary` one by one. As soon as one server returns true, the CRM resubmits the request using the same *rid* to the new primary. If no new primary is found or after a maximum number of retries, an exception is returned to the client.

3.2 Primary Algorithm

The primary keeps two main data structures. For each active client request (i.e., no positive response has yet been sent to the client), it keeps an *ActiveExecution* object. This object contains the request and its *rid*, the response if already generated, a link to the associated transaction, and links to all EJBs that are accessed during the execution of the request. *ActiveExecution* objects are kept in a hash table *AE* for fast retrieval. Furthermore, for each fully executed client

[3] We are currently extending our system to allow for several primaries, each serving a subset of clients and being backups for others. Conceptually, this is quite easy

[4] At startup, the CRM reads the server list from a file, and uses the first as primary. After that, the primary sends new lists to the CRM whenever view changes occur.

request where a response has be generated, a pair $(rid, response)$ is kept in a hash table RR. Since an EJB can call another EJB, there are not only client requests and responses, but also *internal requests* and corresponding responses. They are not kept in RR nor in the *ActiveExecution* object. An internal request does not have an *rid*, but is otherwise conceptually the same as a client request.

Let's first recall the execution logic of a client request in a regular J2EE model as depicted in Figure 1. When the container intercepts a client request, it first calls the transaction manager (TM) to begin a new transaction, then forwards the request to the EJB and waits for the response, then calls the TM to commit the transaction, and finally returns the response to the client. Each call to an EJB carries the identifier *txid* of the transaction that is associated with this call. The primary algorithm extends now this basic scheme. The example of a successful execution of a simple request is depicted in Figure 2. A *replication manager RM* intercepts (i) the *begin* call to the TM to do some initialization, (ii) requests to EJBs to keep track of stateful EJBs, and (iii) the *commit/abort* call to the TM to complete the replication algorithm. Note that (ii) also contains sub-requests from one EJB to another. We will now describe the replication algorithm according to the letters indicated in Figure 2.

(A) When intercepting the *begin*, an *ActiveExecution* object *aeo* is created.
(B) Once the transaction has been created it is linked to the *aeo* object.
(C) When a request (client or internal) to a component is intercepted, the RM first checks whether the corresponding response can already be found in RR (client request only). If yes, the response is returned without reexecution of the request. Otherwise, a link to the EJB is added to *aeo* for SFSB and EB.
(D) When the call returns and it is the response to a client request, both the *rid* of the request and the response are stored in *aeo*.
(E) When intercepting the *commit* request to the TM, the state transfer is initiated. A message *committing* is created containing the content of the corresponding *aeo*. For each link to a SFSB within *aeo* the state of the SFSB is included in the message. For EBs, only the identifiers are included to guarantee that EBs can be initialized correctly at the backups. A flag u is included in the message indicating whether the transaction updated the database or not[5]. The *committing* message uses uniform-reliable delivery (to guarantee all-or-nothing delivery) and FIFO order (to guarantee that the backups receive state changes in correct order). If u is true, the primary additionally inserts a marker record containing the transaction id *txid* into a special table *CommittedTransaction* within the database.
(F) When the primary receives its own *committing* message (guaranteeing that all backups will receive it), the *commit* request is finally sent to the TM.
(G) After transaction commit and if $u = true$, the primary informs the backups by sending a message *committed*. This message is not needed for correctness but speeds up failover. Hence, it is only sent using reliable, FIFO delivery, and the primary does not wait to receive the message itself. At last, *aeo* is

[5] We determine if a transaction updates the database by intercepting and parsing SQL statements sent to the database through JDBC.

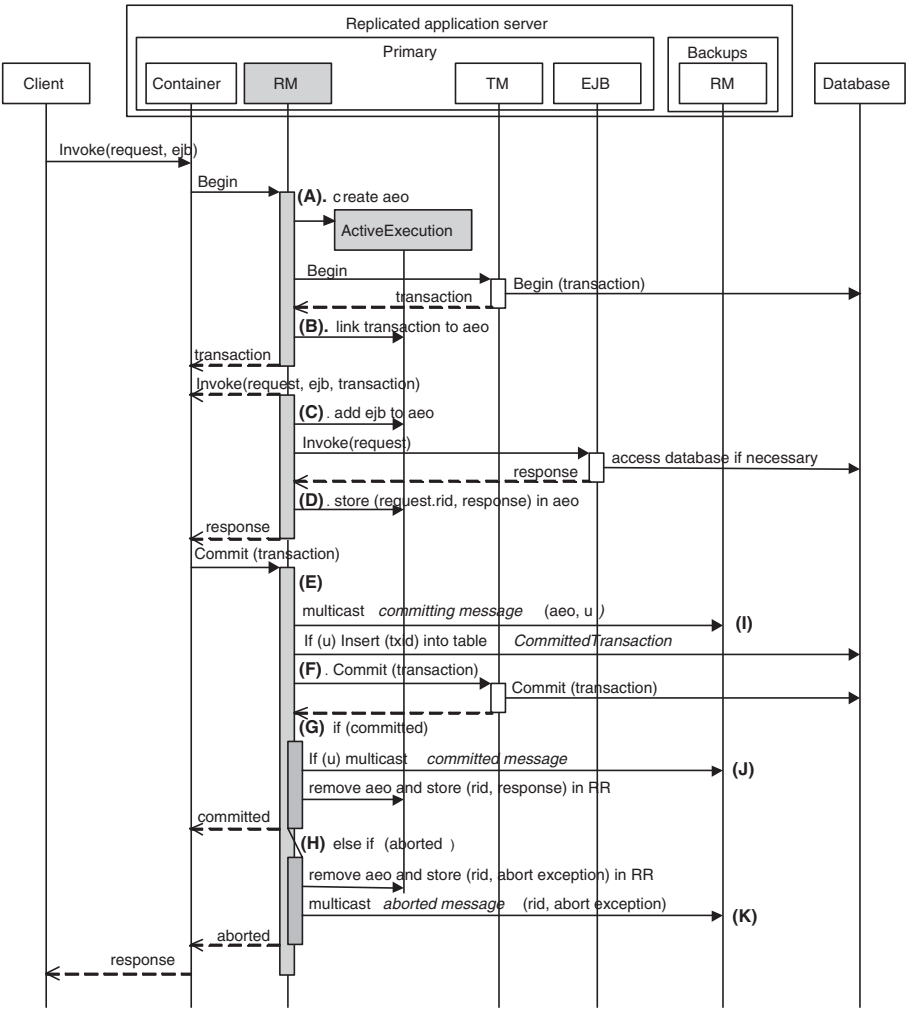


Fig. 2. Execution logic of the primary algorithm

removed from AE, the rid/response pair is stored in RR, and the response returned to the client.

(H) A transaction can abort at several places, e.g., when the TM executes the commit request or in the middle of the execution of an EJB method. In the first case, the client request was already completely executed, and hence, the RM had recorded the response to this request. In the second case, the response was not yet generated. However, in both of the above cases, the TM generates an abort exception which must be forwarded to the client. The RM intercepts the exception and inserts it together with the rid into RR. An *aborted* message will be multicast to the backups using uniform-reliable delivery to guarantee that the backups know about the change of

outcome before returning to the client. There exist alternative mechanisms for abort handling but we do not discuss them for space reasons.

3.3 Backup Algorithm

The backup algorithm used during normal processing is designed to put as little load as possible on the backups in order to allow backups to perform other tasks. In particular, it does not immediately apply all state changes. An EJB might be changed many times but only the last state is relevant at the time of failover.

The backup uses the following data structures. Incoming messages are put in a FIFO queue Q and processed one at a time. The backup maintains a hash table CM containing the content of *committing* messages, and a hash table RR containing rid/response pairs. Furthermore, the set ES contains the latest state of each SFSB and the identifiers of EBs as received in *committing* messages. We again show the algorithm according to events depicted in Figure 2.

(I) Upon receiving a *committing* message, if $u = true$, its content will be temporarily stored in CM, else it will be accordingly stored in ES and RR.
(J) Upon receiving a *committed* message, the corresponding information from CM is removed and accordingly stored in ES and RR. For each SFSB that did not already have an entry in ES, the SFSB is initialized to guarantee fast failover (the state, however is not set).
(K) Upon receiving an *aborted* message, the corresponding information from CM is removed. Furthermore, the rid with the abort response piggybacked in the *aborted* message is stored in RR. In this case, ES is not updated.

3.4 Failover Algorithm

When a backup receives a view change message from the GCS, it checks whether the current primary is still member of the view. If not, it determines whether it is then new primary (using any deterministic mechanism). If it is the new primary, the failover algorithm starts. During failover, calls to the `is_primary` method block. Once failover is completed, the method will return that this replica is now primary. We will first present a simple but slow failover algorithm:

1. For each message content in CM (we have received the *committing* but no *committed* message), we check whether it has committed at the database or not. For that, we check in the *CommittedTransaction* table in the database for the corresponding *txid* marker. If we find the marker, we store the content accordingly in ES and RR, and initialize any new SFSBs. If there is no marker, we discard the content of the message.
2. For all SFSBs in ES, we set the state to the state in ES. For all EBs in ES we load the state from the database.
3. The replica switches to the primary algorithm.

We now briefly outline why the combination of primary, backup and failover algorithms guarantees state consistency and exactly-once execution in case of primary crash. Recall that all database transactions active at the primary when

it crashes will be automatically aborted at the database. We analyze now the cases where the primary crashes during request execution leading to a negative response and resubmission of the request. (i) If the primary crashes before sending the *committing* message, the backups do not have the SFSB state changes, and the database transaction will be aborted. This provides state consistency. When the CRM resubmits the request to the new primary, it will be handled as a new request. (ii) If the primary crashes after sending the *committing* but before transaction commit, at failover the new primary will look for the *txid* marker in the database but not find it. Hence, it knows that the transaction aborted, and will disregard the *committing* message. Again, the system behaves as if the request had never been processed. (iii) If the primary fails after transaction commit but before sending the *committed* message, the new primary will again look for the marker, and will find it this time. Hence, it will consider the state of the EJBs in the *committing* message. Again, application server and database have a consistent state. Upon resubmission of the request, the new primary will immediately return the result. (iv) When the primary fails after sending the *committed* message, the new primary knows that the transaction committed without need to check for the marker. (v) If the primary sends the *committing* message, the transaction aborts and the primary crashes before sending the *aborting* message, the new primary will check for the marker and not find it. Upon resubmission of the request, it will be reexecuted at the new primary (possibly again leading to abort). (vi) If the primary had not sent the *committing* message, the transaction aborts but the primary crashes before sending the *aborting* message, the new primary will accept the resubmission as a completely new request. (vii) Finally, if the primary crashes after sending the *aborting* message, then the new primary will return the abort response upon a resubmission of the request. In all cases, this is the correct behavior.

Speeding up failover. Setting and loading the state of all EJBs at the time of failover can be very time consuming, and hence, might not be transparent to the user (the CRM at the client will be blocked during failover). Therefore, we suggest a lazy approach. We only perform the first step of above failover algorithm, i.e., handle messages in CM for which we have not yet received the *committed* message. Then, we switch to the primary algorithm. As a result, all EJBs are initialized but their state is not up-to-date. Hence, the primary algorithm during normal processing has to be extended slightly to work correctly after failover. When a request is intercepted at the RM, the RM checks first whether the EJB exists. If not, this is a new EJB and it is initialized. If the EJB already exists, the RM checks whether the EJB has an entry in ES. If yes, its state is set to the state found in ES (for SFSB) or loaded from the database (for EB). Then the EJB's entry in ES is removed. Once ES is empty, no checks have to be performed anymore. This procedure slows down request execution shortly after failover but makes the failover itself much faster.

3.5 Recovery Algorithm

When a failed replica recovers or a new replica joins it has to first receive the current state, and then will become a backup. Thus, one of the existing replicas,

referred to as the *peer*, must send its current state to the joining replica. Either the current primary or any existing backup can serve as peer. Choosing a backup as peer avoids extra load on the primary, and simplifies the procedure. Hence, we currently have only implemented the case where a backup is the peer. If there is more than one backup available, we need to choose one of them as peer. In our algorithm, each backup who is willing to become the peer multicasts a *willing* message to all replicas using uniform-reliable delivery (to guarantee all or nothing) and total order delivery. The backup whose *willing* message is the first to be delivered will become the peer.

When a backup receives the first *willing* message and it is the sender, it delays the processing of any new messages coming from the primary. It generates a *recovery* message containing the content of CM, RR, and ES and sends it to the joining site using point-to-point communication. While waiting for the *recovery* message, the joining replica might have already received messages from the primary and enqueued in Q (it starts receiving messages when the GCS delivers the view change). Once the joining site receives the *recovery* message, it initializes its data structures accordingly. The *recovery* message might already contain the state of some of the messages in Q. Hence, these messages must be removed from Q before the backup algorithm can start processing messages from Q. In order to determine which messages to remove, we timestamp all messages.

4 Implementation

We have implemented this algorithm using the ADAPT J2EE replication framework [4]. As GCS we used JBora/Spread [3,9]. The ADAPT framework is an extension of a J2EE server, allowing replication algorithms to be plugged in. It is currently based on the JBoss J2EE server [13]. The framework defines an API for the interaction between the replication algorithm and the containing server system. The replication developer codes an algorithm by implementing the API, and deploys it by moving the classes into the server's deployment directory.

When a component is invoked at runtime, the framework transfers control to the replication algorithm. Through the API, the algorithm sees an abstracted view of the component, the invocation, and the other elements of the server. The algorithm may perform any actions, such as setting component state or communicating with other replicas, before continuing the invocation. The main advantage of using the ADAPT framework, rather than modifying the server directly, is that it simplifies replication programming. The custom API insulates the replication algorithm from the details of the server implementation. Further, the algorithm is centralized in just a few classes, rather than scattered throughout the server system. Figure 3 shows how the framework separates the J2EE server from the replication algorithm implemented within the replication managers. Although the framework adds overhead, our performance analysis shows that the replication tasks themselves consume significantly more time than the use of the framework. Hence, we believe that the overhead is well worth the modularity that this approach provides.

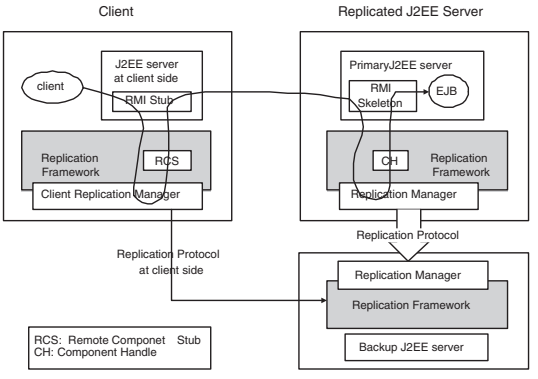


Fig. 3. ADAPT Framework separates replication algorithm from J2EE server

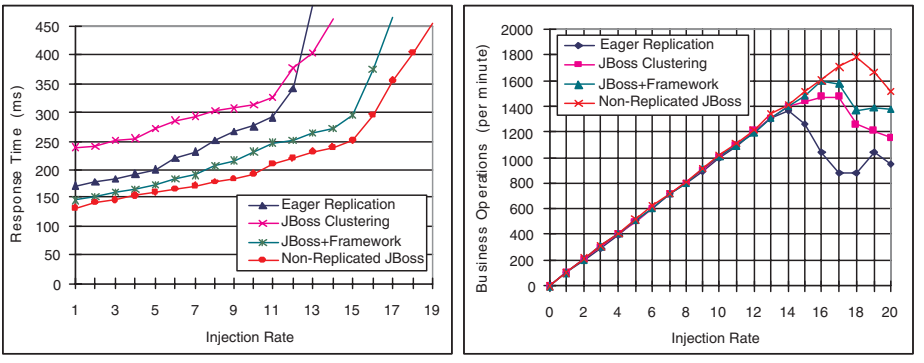
5 Performance Evaluation

We used three suites of experiments to evaluate our system. First, we use the ECperf benchmark [24] to evaluate the performance on a "real" application and compare it with JBoss's existing replication technique. A second test suite presents a series of micro benchmarks that show the performance for different components and database access patterns. The third experiment evaluates failover. All tests were run on a cluster of PCs (3.0 GHz Pentium 4 with 1 GB of RAM) running RedHat Linux. The configuration consists of one machine emulating clients, one machine running the web server (if needed), two machines running JBoss application server 3.2.3. instances, and one machine running DB2 as our database system.

5.1 Evaluation Based on ECperf Benchmark

ECperf [24] emulates businesses involved in manufacturing, supply chain and order/inventory management. The application is split into four domains: customer, manufacturing, supplier domain, and corporate. The main configuration variable is the *transaction injection rate* (IR) which refers to the rate at which a specified subset of business transaction requests are injected into the system.

In this experiment, we evaluate the following architectures. (1) A regular, non-replicated JBoss server as baseline for comparison. (2) The JBoss server including the framework without the replication algorithm to evaluate the overhead of an abstraction layer useful for reusability and platform independence. (3) Two application server replicas using our eager replication algorithm. (4) Two application server replicas using JBoss's own replication solution [14], which we refer to as JBoss clustering. For both (3) and (4) we did not take advantage of load balancing, and submitted all requests to one server. JBoss clustering uses passive, warm, and eager replication. If a client request triggers execution on several stateful components, state transfer takes place individually once execution on the component has terminated. Although eager, problems occur if state



(a) Response Time (b) Throughput

Fig. 4. ECperf Comparison

propagation for some of the components was successful but the primary fails before committing the transaction at the database. In this case the backups have a partially replicated state while the database transaction aborted. Hence, JBoss clustering does not provide state consistency or exactly-once execution.

Figure 4 shows the results of the experiment measured over the steady state phase of the run (the ramp-up and ramp-down phases are ignored). Figure 4(a) shows the average response time for order entry transactions of the customer domain. At low throughput, the framework adds around 10 ms to the non-replicated JBoss, our algorithm (indicated as Replicated JBoss) adds 25 ms while JBoss clustering adds around 100 ms. This gives an overhead of around 25% for our algorithm plus the framework, and 70% for JBoss clustering. The latter performs so badly because it sends state after each method invocation while our solution only communicates at the end of the transaction. As a comparison, Moser et. al. [18] indicate around 15% overhead for FT-CORBA (primary-backup) compared to non-replicated CORBA. With increasing IR, the response time in all systems increases steadily until saturation point. The gap between non-replicated JBoss and the eager algorithm increases slightly but steadily, while it remains nearly the same for JBoss clustering until around 11 IR beyond which it becomes significantly worse. More information about the saturation point can be found in Figure 4(b). This figure uses the average *business operations per minute* to represent the maximum achievable throughput when the IR increases. The maximum in each curve shows the system shortly before saturation. Our algorithm leads to saturated at an IR of 15 due to CPU overhead. JBoss clustering saturates at 17 IR (also due to CPU) while the non-replicated JBoss saturates at 19 IR. In this case, our DB2 server is the bottleneck although the CPU was also already quite heavily loaded. By optimizing the DB2 configuration (we used the default configuration of DB2 with only small adjustments), the CPU might become the limiting resource also in this case. The reason for earlier saturation of our approach compared JBoss clustering is higher CPU overhead (keeping old responses to guarantee exactly-once, keeping information during transaction execution to send all state in a single message).

In summary, we believe that our approach provides acceptable performance considering the strong consistency guarantees that it provides. It compares favorable with JBoss's clustering mechanism. Nevertheless, the overhead is not negligible. We believe, however, that more "engineering" work in optimizing our in-memory data structures could lead to further improvement. In order to better understand where to start such optimizations, the next section presents results on simpler applications in order to detect potential bottlenecks.

5.2 Component Analysis

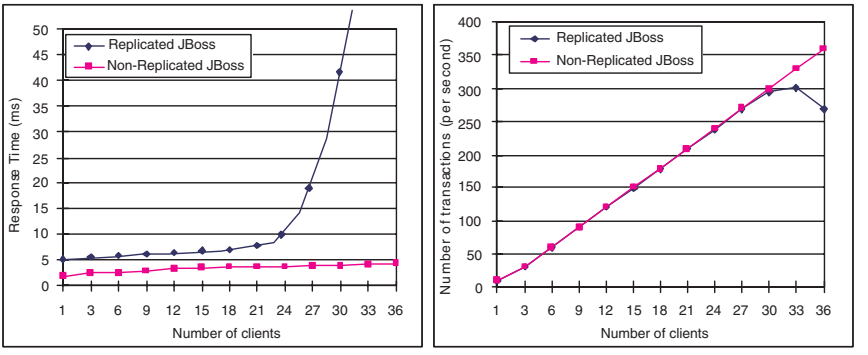
In our second experiment suite, we evaluate the overhead of replication for different components and component combinations. We consider the following cases. *Test 1:* No database access takes place. *Test 2:* Database access (update) takes place but no conflicts occur at the database. That is, different clients access different tuples. *Test 3:* Database access takes place and all transactions conflict. That is all requests access the same tuple. In Test 1, a request triggers the execution of a single method of an SFSB. Test 2 and 3 have two different versions. In the first, a request executes only on one SFSB which makes the database call. In the second, a request calls a SFSB, which calls an EB to access the database.

The main configuration variable is the number of clients. Each client is configured to submit 10 requests per second. However, since a client does not submit a new request before it receives the response for the previous request, if the execution time is longer than 100 ms, the real injection rate is smaller than 10/sec.

Test1: No database access. Figure 5 shows (a) the average response time and (b) the throughput achievable with increasing number of clients. Response times increase slowly for both the replicated and non-replicated system. Below the saturation point, the replication algorithm (including the framework) has an overhead of around 4 ms. This is very low in total numbers, but means an overhead of around 100% for medium number of clients since response times are generally very small. This is the worst case scenario for our algorithm since it contains only SFSBs which all must be replicated. At 24 clients, response times increase sharply due to CPU saturation, and the final saturation is after 33 clients. The non-replicated system does only saturate at around 66 clients again due to CPU overhead. Since the system is CPU bound, and the non-replicated system takes half the time to execute one request compared to the replicated system, it can execute double as many requests before saturation.

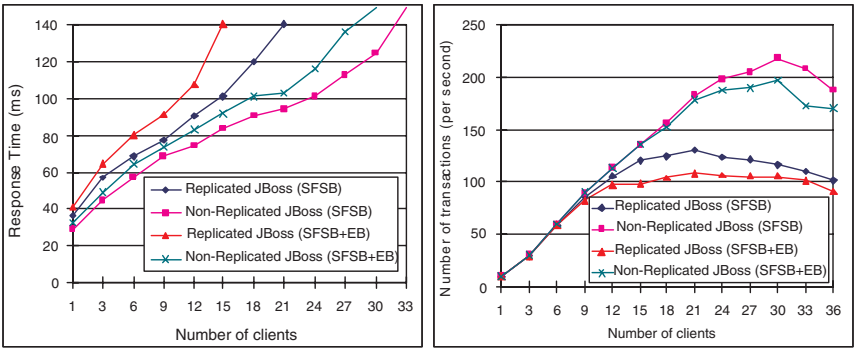
There are two solutions to improve the results of the replicated system. The first is to improve the implementation of the algorithm (e.g., data structures, access paths). This, however, can only succeed to a certain point. After that, alternative replication strategies have to be found, e.g., lazy replication.

Test 2: Conflict-free database access. Figure 6 shows the results of test 2, in which transactions access the database but concurrent transactions never conflict. The figure contains graphs both for the SFSB only and SFSB/EB combinations. Let's first have a look at the SFSB only case. Compared to Figure 5(a) for no database access, response times increase more steeply, and are generally higher. This is due to the database access. When the number of clients is



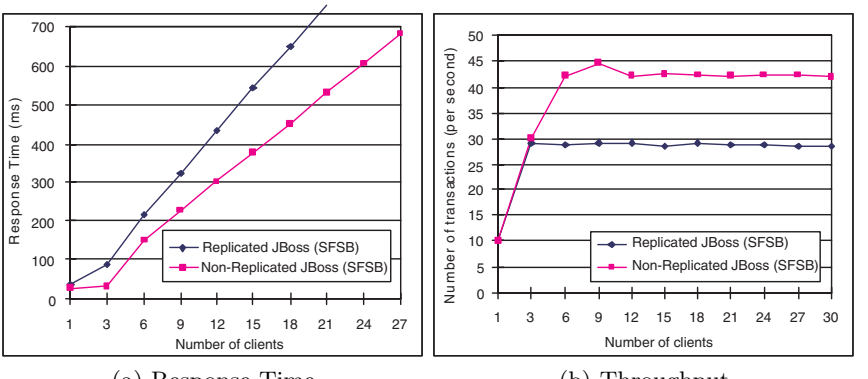
(a) Response Time (b) Throughput

Fig. 5. No database access



(a) Response Time (b) Throughput

Fig. 6. Conflict-free database access



(a) Response Time (b) Throughput

Fig. 7. Conflicting database access

smaller than 15, the overhead of the replication algorithm is stable at around 15 ms. The total time spent in the replication algorithm is higher than with no database access (4 ms) because the marker has to be inserted into the database (if a transaction does not update the database, no marker is inserted). In this scenario, 15 ms only mean an overhead of 20% for medium client numbers since transaction execution is generally long. With more than 15 clients, the time spent in the replication algorithm increases linearly with the number of clients and the throughput increases only very slowly. At 15 clients, the CPU overhead is around 85%. After that, it does not increase fast because the system always waits for operations at the database to complete. The saturation point is at 22 clients. The non-replicated server reaches saturation with 33 clients.

When database access is filtered through EBs, response times both for the non-replicated and the replicated system are generally higher due to the EB overhead (see, e.g., [7], for a comparison of SFSB and EB). However, the relative performance is similar to the SFSB only case.

The conclusion is the easy observation that if the original system has high execution times, than the overhead of the replication algorithm has not such a big relative effect than with small execution times.

Test 3: Conflicting database access. Figure 7 shows the results when all transactions conflict at the database. We only present the SFSB only case, since the effect of using EBs is similar to test 2. Generally, response times (Figure 7(a)) are much larger than in test 2 due to the long blocking times at the database. They increase sharply with the number of clients for both replicated and non-replicated case. The difference between replicated and non-replicated system is bigger than in test 2 and also increases faster than in test 2. The reason is that the replication algorithm generally increases the execution time for each transaction. Assume transaction T1 holds a lock and T2 and T3 wait for the lock. The time T1 needs longer to finish due to replication is also added to T2's and T3's execution time. Additionally, the longer execution time of T2 is added to T3's execution time. This means, waiting times are cumulative. We can also see that the maximum throughput (Figure 7(b)) is only around 1/4 of the one in test 2 for both the replicated and non-replicated system due to the blocking.

As a conclusion, although the CPU is not saturated, the CPU overhead of replication limits its performance. Although the response time increase is due to longer waiting times at the database, it is caused by the computation overhead.

5.3 Failover

In this experiment, we evaluate the *failover time*, i.e., the period from the time point the backup detects the primary's crash to the completion of failover. We evaluate the lazy approach as described in Section 3.4. The main factor to impact the failover time is the number of transactions in CM since we need to query the *committedTransaction* table for each of its entries.

Figure 8 shows the results of an experiment where we crashed the primary after different running times of ECPerf with an IR of 5. It indicates the number of entries found in CM at the time of failover and the time needed for failover. Both remain stable over the different runs. That is, failover is independent of the time

Running Time (minutes)	30	60	120	240	600
No. entries in CM	2	3	5	2	4
failover time (ms)	88	93	116	84	102

Fig. 8. Failover time for different running time of ECPerf at 5 IR

the primary was running before the crash. Instead, the failover is impacted by the throughput at the primary. The more transactions are running at the same time, the more transactions might exist in CM at a given time. We conducted a second experiment where we run ECPerf with increasing IR and crashed the primary after 30 minutes. The failover time increased from 40 ms at 1 IR to 160 ms at 11 IR. Once the primary saturates, message propagation becomes bursty. As a result, just before the crash, the backup might have received many messages which must be processed first. In this case, failover time becomes much longer.

6 Related Work

Nowadays, most J2EE products provide some form of replication. We already discussed JBoss's clustering solution in Section 5. WebLogic [6] uses passive, warm, and lazy replication. The primary propagates the state soon after returning the response to the client to keep replicas as consistent as possible. Pramati [22] uses passive, cold, and eager replication. State changes are immediately written into the database. If the primary crashes in the middle of execution, the database transaction aborts and with it the state changes of the application server. None of the above solutions provides exactly-once semantics, and only Pramati guarantees state consistency.

As an example of replication in CORBA, the Eternal system [19,20] is based on the FT CORBA architecture [21]. Eternal supports active replication and both warm and cold passive replication. Determinism is required even for passive replication since it uses lazy propagation. The primary replicates the state to backups periodically in form of checkpoints. Between two checkpoints, all messages from clients and the database are logged. At failover, the new primary first restores the state of the last checkpoint, and then replays logged requests. Phoenix/COM+ [2] is based on .NET using passive, cold, and lazy replication. It has similarities to the Eternal system. States are replicated periodically, and requests between two checkpoints are logged. However, it distinguishes non-deterministic events from deterministic events. For non-deterministic events, it uses eager replication to avoid the problems that exists in Eternal. However, it is unclear how to determine whether an event is non-deterministic or not. Neither Eternal nor Phoenix/COM+ explicitly discuss state consistency.

There are also some general solutions that are not developed within the context of a specific application server architecture. e-Transactions [11] offer exactly-once semantics for stateless application servers. When a replica of an application server executes a request, it inserts the response and a marker into the database. If the server crashes before sending the response and the client resubmits the request to the new primary, the new primary checks whether a marker for this request exists in the database. If yes, the response will be retrieved and returned

without re-executing the request. X-ability [12] provides a general replication solution for stateful servers without considering the specifics of a particular application server architecture. Eternal, Phoenix/COM+, e-Transactions, and X-ability are all implemented in different environments with different conditions. In our approach, we have taken advantage of some of J2EE's properties leading to a different solution.

7 Conclusion and Future Work

This paper describes a replication algorithm that provides both state consistency and exactly-once semantics for stateful J2EE application servers. The algorithm was integrated into the JBoss application server using the ADAPT framework. The performance evaluation shows that for a wide range of throughputs the overhead is acceptable and comparable (e.g., FT-CORBA) or better (JBoss's clustering method) than existing solutions..

Our current work attempts to enhance the current system in several ways. First, we are evaluating the impact of different failover strategies and the costs of recovery. Second, we are looking into optimizing our algorithm to reduce the CPU overhead. Third, we are extending the system to handle network partitions. In such situation, a client might resubmit a request to a backup or a backup becomes a new primary although the old primary has not failed. Finally, we are looking into more advanced transaction models. For instance, a transaction can span more than one request, or a request can span more than one transaction. In the first case, failover must be able to handle transactions for which a client has received some but not all responses. In the second case, failover must be able to handle requests for which some but not all transactions have committed.

References

1. C. Amza, A. L. Cox, and W. Zwaenepoel. Distributed versioning: Consistent replication for scaling back-end databases of dynamic content web sites. In *Proc. of Int. Middleware Conf.*, Rio de Janeiro, Brazil, 2003.
2. R. Barga, D. Lomet, and G. Weikum. Recovery guarantees for general multi-tier applications. In *Proc. of the Int. Conf. on Data Engineering (ICDE)*, San Jose, California, 2002.
3. A. Bartoli, C. Calabrese, M. Prica, E. Antoniutti Di Muro, and A. Montresor. Adaptive message packing for group communication systems. In *OTM Workshop on Reliable and Secure Middleware*, Catania, Sicily, Italy, 2003.
4. A. Bartoli, V. Maverick, S. Patarin, and H. Wu. A Framework for Prototyping J2EE Replication Algorithms. In *Proc. of the Int. Symposium on Distributed Objects and Applications (DOA)*, Agia Napa, Cyprus, 2004.
5. A. Bartoli, M. Prica, and E. Antoniutti Di Muro. A replication framework for program-to-program interaction across unreliable networks and its implementation in a servlet container. Technical report, DEEI, University of Trieste, Italy, 2004.
6. BEA Systems Inc. *BEA WebLogic Server, release 7.0: Programming WebLogic Enterprise JavaBeans*, September 2002.

7. E. Cecchet, J. Marguerite, and W. Zwaenepoel. Performance and scalability of EJB applications. In *Proc. of OOPSLA*, Seattle, Washington, 2002.
8. E. Cecchet, J. Marguerite, and W. Zwaenepoel. Partial replication: Achieving scalability in redundant arrays of inexpensive databases. In *Proc. of Int. Conf. on Principles of Distributed Systems*, La Martinique, French West Indies, 2003.
9. Center for Networking and Distributed Systems (CNDS), Johns Hopkins University. The Spread toolkit. http://www.spread.org/.
10. G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4), 2001.
11. S. Frølund and R. Guerraoui. A pragmatic implementation of e-transactions. In *Proc. of Symp. on Reliable Distributed Systems (SRDS)*, Nürnberg, Germany, 2000.
12. S. Frølund and R. Guerraoui. X-ability: a theory of replication. In *Proc. of Symp. on Principles of Distributed Computing (PODC)*, Portland, Oregon, USA, 2000.
13. The JBoss Group. JBoss application server. http://www.jboss.org.
14. The JBoss Group. *JBoss Clustering*, 2002.
15. R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Scalable database replication middleware. In *Proc. of Int. Conf. on Distributed Computing Systems (ICDCS)*, Vienna, Austria, 2002.
16. B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Trans. on Database Systems*, 25(3), 2000.
17. A. I. Kistijantoro, G. Morgan, S. K. Shrivastava, and M. C. Little. Component replication in distributed systems: a case study using Enterprise Java Beans. In *Proc. of the Symp. on Reliable Distributed Systems (SRDS)*, Florence, Italy, 2003.
18. L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. A fault tolerance framework for CORBA. In *Proc. of the Int. Symp. on Fault-Tolerant Computing*, Washington, DC, USA, 1999.
19. P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. State synchronization and recovery for strongly consistent replicated CORBA objects. In *Proc. of the Int. Conf. on Dependable Systems and Networks*, Göteborg, Sweden, 2001.
20. P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Strongly consistent replication and recovery of fault-tolerant CORBA applications. *Journal of Computer System Science and Engineering*, 32(8), 2002.
21. Object Management Group. *Fault Tolerant CORBA Specification*, December 1999.
22. Pramati Technologies Private Limited. *Pramati Server 3.0 Administration Guide*, 2002. http://www.pramati.com.
23. SUN Microsystems Inc. *JAVA 2 Platform Enterprise Edition Specification, V1.3*, October 2000.
24. SUN Microsystems Inc. *ECperf^{TM} specification, V1.1, final release*, 2003. http://java.sun.com/j2ee/ecperf.