Critical Evaluation of the EJB Transaction Model

Raul Silaghi and Alfred Strohmeier

Software Engineering Laboratory
Swiss Federal Institute of Technology in Lausanne
CH-1015 Lausanne EPFL, Switzerland

E-mail: {Raul.Silaghi, Alfred.Strohmeier}@epfl.ch

Abstract. Enterprise JavaBeans is a widely-used technology that aims at supporting distributed component-based applications written in Java. One of the key features of the Enterprise JavaBeans architecture is the support of declarative distributed transactions, without requiring explicit coding. In this paper, after a brief introduction of the concepts and mechanisms related to the EJB Transaction Model, we provide guidelines for their consistent use. We then evaluate the EJB Transaction Model on an Auction System case study. The encountered limitations are presented, and possible work-arounds are proposed for the auction system. We conclude with suggestions for enhancing the current EJB Transaction Model.

Keywords. EJB, Enterprise JavaBeans, Transactions, Auction System.

1 Introduction

For three decades, transaction processing has been a cornerstone of modern information technology: it is an indispensable asset in banking, stock trading, airline reservation systems, travel agencies, and so on. With the new millennium's proliferation of e-Commerce applications, business-to-business workflows, and broad forms of Web-based e-Services, transactional information systems are becoming even more important.

Transactions are a classic software structure for managing concurrent accesses to global data and for maintaining data consistency in the presence of failures. The notion of transaction was first introduced in database systems in order to correctly handle concurrent updates of data and to provide fault tolerance with respect to hardware failures [1]. A transaction groups an arbitrary number of operations on data objects (also referred to as *transactional objects*), making the operations as a whole appear indivisible to the application and with respect to other concurrent transactions.

The classic transaction scheme relies on three standard operations: *begin*, *commit*, and *abort*, which mark the boundaries of a transaction. After beginning a new transaction, all update operations on transactional objects are done on behalf of that transaction. At any time during the execution of the transaction it can *abort*, which means that the state of the system is restored (i.e., *rolled back*) to the state at the beginning of the transaction. Once a transaction has completed successfully (referred to as *committed*), the effects become permanent and visible to the outside world. The properties of trans-

actions are referred to as the ACID properties: *Atomicity*, *Consistency*, *Isolation*, and *Durability* [1].

Support for transactions is an essential component of the Enterprise JavaBeans architecture. The Enterprise JavaBeans architecture supports only *flat transactions*, despite the fact that the classic transaction model has been extended a long time ago to support nested transactions [2], and thus provides a more flexible support for concurrency and recovery.

By simply setting certain attributes in a deployment descriptor, a developer can make his or her enterprise bean be executed within a client's transaction context, within a new transaction context, or within no transaction context. This power of freeing the developer from writing transactional code, not to say to write his or her own transaction service, comes nevertheless at a price. The simplicity in using the EJB Transactional Model comes along with a certain rigidity that restricts the ways in which transactions may be used and constrains the developer to stick to a certain manner of building distributed transaction-enabled applications.

For most applications, the EJB Transaction Model is adequate and can be used in a straightforward way. For other applications, however, certain work-arounds are necessary and a very precise configuration of the deployment descriptors is needed. Based on a concrete case study, i.e., the auction system, we will present in this paper the limitations that were encountered in the EJB Transaction Model, pointing out certain features that would enhance the current model and would make it more open and flexible.

The rest of the paper is organized as follows: Section 2 provides an overview of the Enterprise JavaBeans Transaction Model, introducing concepts that define how transactions are handled and discussing some issues in using them; Section 3 briefly describes the auction system case study; Section 4 presents the implementation solution for the auction system on top of EJBs, highlighting some problems that may arise and proposing work-arounds when possible; Section 5 discusses some features that are missing in the current EJB Transaction Model, and Section 6 draws some conclusions.

2 Enterprise JavaBeans Transactions

This section presents an overview of the Enterprise JavaBeans Transaction Model, setting the scene for the analysis that will be performed on the auction case study.

The Enterprise JavaBeans architecture [3] is a component-based architecture for building distributed business applications. It aims at simplifying the development of complex systems in Java by defining six distinct roles in the application development and deployment life cycle. These roles may be performed by different parties.

One of the roles is the *Enterprise Bean Provider*. Typically performed by an application domain expert, the Bean Provider builds reusable components, called *enterprise beans*, that implement the business methods without concern for the distribution, concurrency, persistence, transaction, security, and other non-business-specific aspects of the application. Enterprise beans are further deployed in Containers on Application Servers. The *EJB Container Provider* together with the *EJB Server Provider* are the ones supposed to be experts in distributed systems, concurrency, persistence, transac-

tions and security. They must deliver tools for the deployment of enterprise beans, and a run-time system that provides the deployed beans with transactions and security management, distribution, management of resources, and other services. The other roles defined by the EJB specification are the *System Administrator*, the *Application Assembler*, and finally the *Deployer*.

Even without knowing anything about transactions, the Bean Provider must somehow tell the Container which beans, or which methods of a bean, or which segments of code, must be executed under the control of a transaction. The Bean Provider can choose between using programmatic transaction demarcation inside the enterprise bean code (this style is called *bean-managed transaction demarcation*) or declarative transaction demarcation performed automatically by the EJB Container (this style is called *container-managed transaction demarcation*).

With bean-managed transaction demarcation, the enterprise bean code demarcates transactions using the `javax.transaction.UserTransaction` interface. All resource accesses between the `UserTransaction.begin` and `UserTransaction.commit` calls are part of a transaction.

With container-managed transaction demarcation, the Container demarcates transactions following the instructions received from the Bean Provider in the *deployment descriptor*. These instructions can be set for the enterprise bean as a whole (in which case they apply to all enterprise bean methods) or selectively for individual methods in a bean.

The Enterprise JavaBeans architecture defines three types of enterprise bean objects: *session*, *entity*, and *message-driven* objects. Due to their one-to-one mapping to tables in a database, entity beans are the most interesting for concurrency and transactions. As a consequence, we will concentrate only on entity beans for the remainder of this paper.

We will introduce now some attributes that are part of the EJB Transaction Model and that guide the Container in providing transaction support. Some issues in using these attributes along with the concurrency support offered by the Container will also be presented.

2.1 Setting Transactional Attributes in the Deployment Descriptor

The EJB specification [3] does not require enterprise bean and EJB client developers to write any special code to use transactions. Instead, the Container manages transactions based on two deployment descriptor attributes associated with each enterprise bean or with each enterprise bean method in particular: the *transaction* attribute, and the *transaction isolation level* attribute.

While transaction attributes are well standardized by the EJB specification, the transaction isolation levels are not yet standardized. What the specification proposes, however, are a set of guidelines that should be followed by the EJB Container Providers. In what comes next we will present the transaction attributes and the transaction isolation levels as supported by the IBM WebSphere Application Server [5].

2.1.1 Setting the Transaction Attribute

The transaction attribute defines the transactional manner in which the Container invokes enterprise bean methods. The valid values for this attribute in decreasing order of transaction strictness are introduced in Table 1 together with their effect on the transaction context.

Table 1. Effect of the Bean's Transaction Attribute on the Transaction Context

Transaction attribute	Client transaction context	Bean transaction context
`TX_MANDATORY`	No transaction	Not allowed
	Client transaction	Client transaction
`TX_REQUIRED`	No transaction	New transaction
	Client transaction	Client transaction
`TX_REQUIRES_NEW`	No transaction	New transaction
	Client transaction	New transaction
`TX_SUPPORTS`	No transaction	No transaction
	Client transaction	Client transaction
`TX_NOT_SUPPORTED`	No transaction	No transaction
	Client transaction	No transaction

While the second column in Table 1 indicates whether or not the bean method is invoked from within a client transaction context, the third column indicates the exact transaction context in which the bean method will be executed, e.g., the client transaction context, a new transaction context, or no transaction context.

Another transaction attribute that is not presented in Table 1 is `TX_BEAN_MANAGED`; it notifies the Container that the bean class directly handles transaction demarcation by using the `javax.transaction.UserTransaction` interface. This attribute can only be set for session and message-driven beans, and not for entity beans, because entity beans must always be designed with container-managed transaction demarcation.

2.1.2 Setting the Transaction Isolation Level Attribute

The transaction isolation level determines how strongly one transaction is isolated from another. Within a transactional context, the isolation level associated with the first method invocation becomes the required isolation level for all other methods invoked within that transaction. If a method is invoked with a different isolation level from that of the first method, an exception is thrown. This constraint is mainly imposed by the underlying databases because most resource managers interpret a change in the isolation level in the middle of a transaction as an implicit *sync point*, committing the changes done so far (even if the transaction has not committed yet).

The possible values that can be set for the isolation level attribute (from strongest to weakest) are: `TRANSACTION_SERIALIZABLE`,

TRANSACTION_REPEATABLE_READ, TRANSACTION_READ_COMMITTED, and TRANSACTION_READ_UNCOMMITTED. None of these values permits two transactions to update the same data concurrently; one transaction must end before another one can update the same data. The values determine only how locks are managed for *reading* data. However, risks to consistency can arise from read operations when a transaction does further work based on the values read. For example, if one transaction is updating a piece of data and a second transaction is permitted to read that data after it has been changed but before the updating transaction ends, the reading transaction can make a decision based on a change that is eventually rolled back. Thus, the second transaction risks making a decision on transient data.

Fig. 1 presents an example of a deployment descriptor highlighting most of the elements that were previously introduced. It describes an entity bean called Account, having three fields, all container-managed. Towards the end, the transaction and isolation level attributes are specified for the whole bean, meaning that every single business method provided by this bean will be executed with the same transaction attribute and the same isolation level.

```
<?xml version='1.0' encoding="ISO-8859-1" ?>
<ejb-JAR>

<!-- ...just part of a deployment descriptor... -->

<entity-bean dname="Account.ser">
<primary-key>account.AccountKey</primary-key>
<re-entrant value="false"/>
<container-managed>accountId</container-managed>
<container-managed>type</container-managed>
<container-managed>balance</container-managed>

<home-interface>account.AccountHome</home-interface>
<remote-interface>account.Account</remote-interface>
<enterprise-bean>account.AccountBean</enterprise-bean>

<jndi-name>Account</jndi-name>

<transaction-attr value="TX_REQUIRED"/>
<isolation-level value="SERIALIZABLE"/>
<run-as-mode value="SYSTEM_IDENTITY"/>
</entity-bean>

</ejb-JAR>
```

Fig. 1. Deployment Descriptor Example

2.2 Issues in Using the EJB Transaction Model

Sequential Access within the same Transaction Context. An entity bean object may be accessed by multiple clients in the same transaction. A program A may start a transaction, and then call program B and program C in the same transaction context. If the

programs B and C access the same entity bean object, the topology of the transaction creates a *diamond*. In this scenario, the programs B and C will access the entity object *sequentially*. Concurrent access to an entity object in the same transaction context would be considered an application programming error, and it would be handled in a Container-specific way.

The EJB specification requires that the Container provides support for *local diamonds*. In a local diamond, all components (here A, B, C, and the entity bean) are deployed in the same EJB Container. Distributed diamonds are not required to be supported by an EJB Container. However, if the EJB Container Provider chooses to support distributed diamonds, then the specification requires that it provides a consistent view of the entity bean's state within a transaction. Two ways of how this can be achieved are proposed in the specification.

Concurrent Access from Multiple Transactions. For concurrent access from multiple transactions, the EJB specification mentions two different strategies that the Container typically uses to achieve proper synchronization. In the first one, the Container acquires exclusive access to the entity object's state in the database. It activates a single instance of the entity bean and serializes the access from multiple transactions to this instance, as shown in Fig. 2.

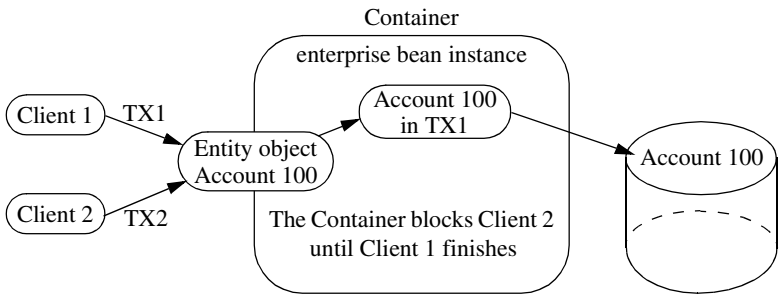

Fig. 2. Multiple clients can access the same entity object using a single instance

In the second one, the Container activates multiple instances of the entity bean, one for each transaction in which the entity object is being accessed, and relies on the underlying database to perform the transaction synchronization during the accessor method calls performed by the business methods, and by the `ejbLoad`, `ejbCreate`, `ejbStore`, and `ejbRemove` methods.

While the second strategy just passes the problem to the next in line, the first one might lead to deadlock, as presented in section 4.1.

Transaction Isolation Level Attribute Issues. The choice of the *transaction isolation level* attribute depends on several factors, which include: the acceptable level of risk to data consistency, the acceptable levels of concurrency and performance, the isolation levels supported by the underlying database. The first two factors are related. Decreasing the risk to consistency requires to decrease concurrency because reducing the risk to consistency requires holding locks for longer periods. The longer a lock is held on a piece of data, the longer concurrently running transactions must wait to access that data.

The TRANSACTION_SERIALIZABLE value protects data by eliminating concurrent access to it. Conversely, the TRANSACTION_READ_UNCOMMITTED value allows the highest degree of concurrency but entails the greatest risk to consistency. These two factors need to be balanced appropriately depending on the application.

The third factor means that although the EJB specification allows one to request one of the four levels of transaction isolation, it is possible that the database being used in the application does not support all of the levels. Also, vendors of database products implement isolation levels differently, so the precise behavior of an application can vary from database to database.

Transaction Attribute Issues. Attention must be paid to the possible values of a transaction attribute. In particular, the TX_REQUIRES_NEW value, as shown in Table 1, directs the container to always invoke a bean method within a *new transaction context*, regardless of whether the client invokes the method within or outside of a transaction context. Please notice that by *"new transaction"* it is meant that a new, top-level transaction is started and no nesting, imbrication, or overlapping is implied.

The scenario presented in Fig. 3 illustrates how misusing the TX_REQUIRES_NEW value for the transaction attribute might lead to the violation of the all-or-nothing property of transactions. We considered a bean method m1 with the transaction attribute set to TX_REQUIRED, and another bean method m2 with the transaction attribute set to TX_REQUIRES_NEW. For rendering the example more realistic, we supposed that the two methods belong to two different entity beans, deployed in different Containers on different Application Servers.

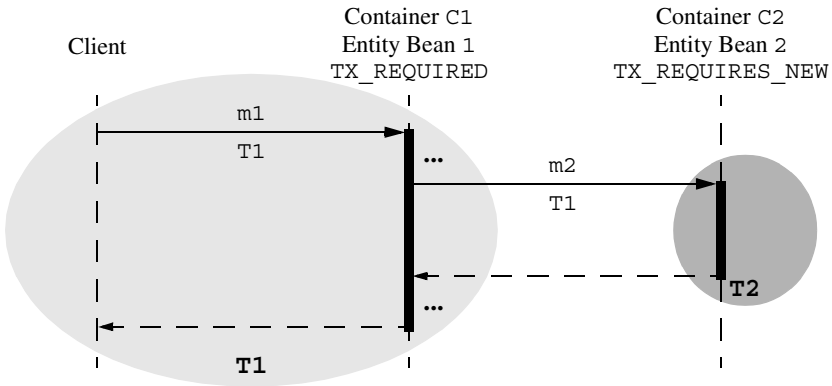

Fig. 3. Changing Transaction Contexts according to the Transaction Attribute

Imagine now that a client calls the bean method m1 from within a transaction context T1. Receiving the call to m1 together with the transaction context T1 that comes with the call, the container C1 follows the exact instructions found in the deployment descriptor for the invoked method, i.e., it will execute method m1 within the client transaction context T1 (see Table 1). When method m2 is called from within m1, the container C1 passes the transaction context with the invocation. Receiving the call to m2 together with the transaction context T1, it is the turn of the container C2 to follow the

instructions found in the deployment descriptor for the invoked method, i.e., it will execute method `m2` within a new transaction context. First of all, however, container `C2` suspends the association of the transaction context `T1` with the current thread, and only then, it will start a new top-level transaction `T2` and it will invoke the business method `m2`. The container will resume the suspended transaction association after the business method `m2` and the new transaction `T2` have been completed.

In our example, performing certain operations in the new top-level transaction `T2` will not guarantee the all-or-nothing property for `T1`. Why? Because once transaction `T2` commits, there is no way to roll it back if later on transaction `T1` aborts. The changes made on behalf of `T2` will persist even if `T1` rolls back. As a conclusion, all the operations that must be performed in a transaction context should not cross the boundaries of other transactions, not even if those transactions were created from within the main transaction. However, other operations that do not interfere with the main transaction could be invoked in separate transactions, e.g., garbage collection is independent of whether we commit or roll back our main transaction.

3 Case Study Description

The auction system is an example of an inherently dynamic, distributed, and concurrent application, with multiple auctions going on and with clients participating in several auctions at the same time. As a consequence, the auction system becomes an excellent case study for testing the performance of new transaction models, in our case the EJB Transaction Model.

The informal description of the auction system presented in this section is inspired by the auction service example presented in [6], which in turn is based on auction systems found on various internet sites, e.g. `www.ebay.com` or `www.ubid.com`.

The auction system runs on a set of computers connected via a network. Clients access the auction system from one of these computers. The system allows the clients to buy and sell items by means of auctions. Different types of auctions may be imagined, like *English*, *Dutch*, *1st Price*, *2nd Price*. In the *English auction*, the item for sale is put up for auction starting at a relatively low minimum price. Bidders are then allowed to place their bids until the auction closes. Sometimes, the duration of the auction is fixed in advance, e.g., 30 days, or, alternatively, a time-out value, which resets with every new bid, can be associated with the auction.

Any client interested in using the auction system services must first register with the system by filling out a *registration form*. All registered users must deposit a certain amount of money or some other security with the auction system at registration time. The money is transferred to an account under control of the auction system.

Once the registration process is completed, the client becomes a *member* of the auction system. Whenever a member wants to make use of the services provided s/he must first *login* to the system using his or her username and password, provided at registration time. Once logged, the member may choose from one of the following possibilities: start a new auction, browse the current auctions, participate in one or several ongoing auctions by placing bids, or deposit or withdraw money from his or her account. To bid

on an item the participant simply has to enter the amount of the bid. A valid bid must fulfill all the following requirements:

- The bidder has sufficient funds on his or her account.
- The member placing the bid is not the member having started the auction.
- The auction has not expired.
- The new bid is higher than the current highest bid. If nobody has placed a bid yet, then the bid must be at least as high as the minimum price requested by the seller.

If the auction closes and at least one valid bid has been made, then the auction ends successfully and the participant having placed the highest bid wins the auction. The money is withdrawn from the account of the winning participant and deposited on the account of the seller, minus a commission, which is deposited on the account of the auction system for the provided services.

If an auction closes, and no participant has placed a valid bid, then the auction was unsuccessful and no charge is required for the provided services.

The auction system must be able to tolerate failures. Crashes of any of the host computers must not corrupt the state of the auction system, e.g., money transfer from one account to the other should not be executed partially.

4 The EJB Solution for the Auction System

In this section, we will present how the auction system was implemented on top of EJBs. Certain design decisions will be motivated by pointing out limitations that were encountered in the EJB Transaction Model. A deadlock situation that can arise in the EJB Solution will be presented and some possible work-arounds will be proposed.

Maybe the most important requirement for auctions is that they must be fault-tolerant. All-or-nothing semantics must be strictly adhered to. Either there is a winner, and the money has been transferred from the account of the winning bidder to the seller's account and the commission has been deposited on the auction system account, or the auction was unsuccessful, in which case the balances of the involved accounts remain untouched. Allowing the possibility of a total rollback while an auction is active and participants are placing their bids, would mean to place everything in a long-living transaction that would commit when there is a winner, or abort if something goes wrong during the lifespan of the auction. This idea is not at all in the spirit of the EJB Transaction Model, where transactions are supposed to last small time units; in this way, they do not block access to transactional objects from other ongoing concurrent transactions for a long time period. Since the auction system is inherently concurrent and very dynamic, with multiple auctions going on and with clients placing bids in several auctions, a long-living transaction acting on behalf of an auction would block the access to the accounts of several participants, thus blocking the other ongoing concurrent auctions from advancing.

Following the EJB specification and having in mind all the considerations presented in section 2.2, the solution that we came up with for implementing the auction system is to break the whole lifespan of an auction into small operations and execute them within separate transactions when their time comes.

Fig. 4 presents the entity beans used to model the auction system. Each entity bean represents an object view of data in different tables in the same or different databases: `MemberBean` handles the personal information of the members, `AccountBean` keeps the evidence of the accounts in the system, and `AuctionBean` manages the auctions in the system.

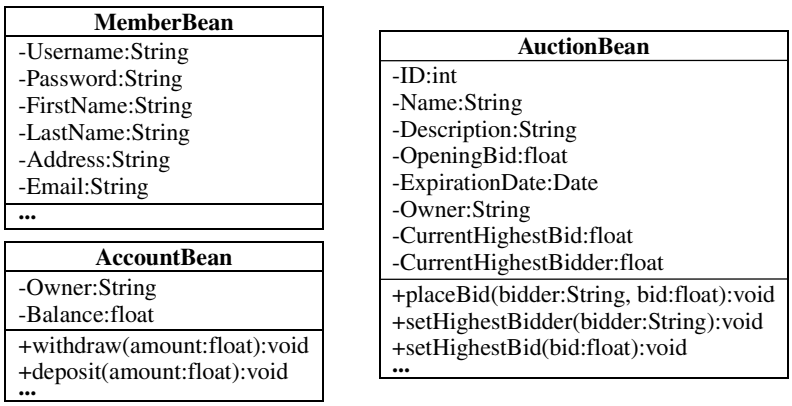

Fig. 4. The Entity Beans in the EJB Solution

In most of the cases, `deposit` and `withdraw` operations can very well be executed in two different transactions. A member would like to `deposit` some money in his or her account and the system will perform this operation within a transaction. Later on, s/he would like to `withdraw` some money from his or her account and the system will perform this new operation within a new transaction. In this case, a new transaction is needed because the two operations are not at all related and there is no reason of undoing the `deposit` operation if the `withdraw` operation fails.

However, when several operations are somehow inter-connected for achieving a certain goal, they can no longer be executed in separate transactions. They have to be executed in the same transaction for the sake of preserving the ACID properties. This is the case of the `placeBid` method which encapsulates small operations that have all to be executed in the same transaction context for preserving a consistent state of the auctions. When a participant places a bid in an auction we have to withdraw from his or her account the corresponding amount of money, protecting in this way the system from participants that would overdraw their accounts by placing bids in several auctions without actually having all that money. However, when s/he gets overbidden in the same auction, we are required to give the money back and make a new withdraw from the new bidder's account corresponding to the amount of the new bid. One possible solution to achieve this behavior is sketched in Fig. 5.

In the `placeBid` method, we withdraw first from the new bidder's account the amount of money s/he wanted to bid, we give the money back to the previous highest bidder, and then we update the information in the current `AuctionBean` object, i.e., the new highest bidder and the new highest bid. If something goes wrong somewhere within this method, the new bid should not be considered valid and the state of all implicated transactional objects (here the `AuctionBean` object and the two `Account-`

```
void AuctionBean::placeBid(bidder:String, bid:float) {
     ...
     //Withdraw the bidded amount of money
     getAccount(bidder).withdraw(bid);

     //Give back the money to the previous highest bidder
     getAccount(getHighestBidder()).deposit(getHighestBid());

     //Update the information in the AuctionBean entity bean
     setHighestBidder(bidder);
     setHighestBid(bid);
     ...
}
```

Fig. 5. The PlaceBid Method of the AuctionBean

Bean objects) should be restored. In order to have the Container execute the whole placeBid method within the same transaction context, the *transaction attribute* of all the involved enterprise beans must be configured accordingly. For this particular example we should have:

- TX_REQUIRED for the placeBid method of the AuctionBean,
- TX_REQUIRED for the withdraw method of the AccountBean,
- TX_REQUIRED for the deposit method of the AccountBean,
- TX_REQUIRED for the setHighestBidder method of the AuctionBean,
- TX_REQUIRED for the setHighestBid method of the AuctionBean.

In this way, when the placeBid method is first invoked, a new transaction context will be created (supposing that it is not already called from within a transaction context), and it will be passed around to all the other method invocations that are made within placeBid. If one of the invoked methods calls at its turn other methods of other enterprise beans, then those methods should also be configured with the transaction attribute set to TX_REQUIRED in the deployment descriptors of those beans.

Based on these considerations, the *English Auction*, as implemented on top of EJBs, is graphically presented in Fig. 6. We identified three main operations that must be executed in a transactional way: the creation of a new auction, placing a bid in an auction, and ending an auction.

By simply filling an *item form*, Member 1 will create a new AuctionBean object within a transaction T1, and, automatically, a new row will be added in the table of all auctions. In a few seconds the displays of all the logged members will be refreshed, and thus, they will see the new proposed auction. Member 2 decides to participate, and places his bid. Once the method placeBid has been invoked on the AuctionBean object, four operations will be executed within the same transaction context (T2) on different beans. First we will withdraw the new bid from the account of member 2. Then some money will be returned to the previous highest bidder (this is not the case here since member 2 is the first bidder). Finally, the information concerning the current highest bidder and current highest bid will be updated in the AuctionBean object. Later on, Member 3 decides to overbid member 2 in the same auction, thus it will invoke the

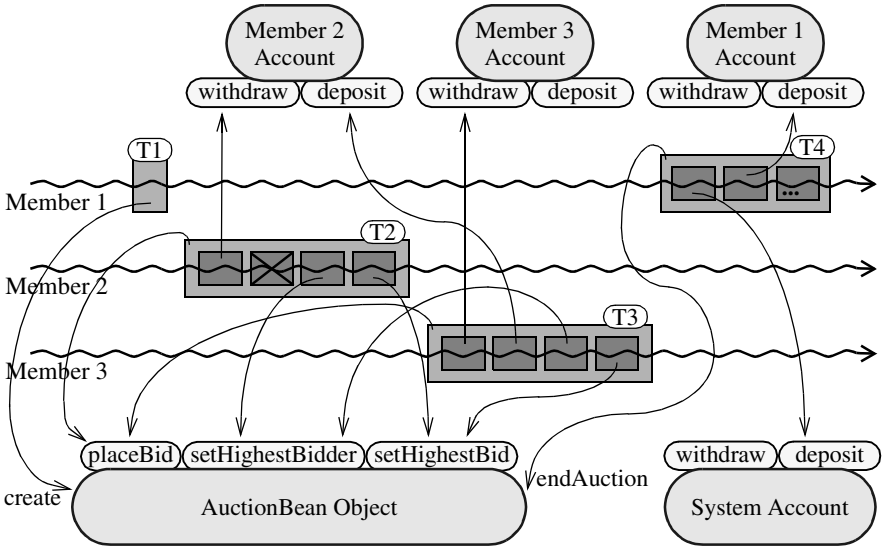

Fig. 6. The *English Auction* with EJBs

`placeBid` method on the same `AuctionBean` object. Within the same transaction context (`T3`) we will: withdraw the new bid from the member 3's account, give back the money that member 2 has previously paid, and update the information in the `AuctionBean` object. In our example we considered that no other member overbids member 3. Once the auction closes, the `endAuction` method is invoked on the `AuctionBean` object. Here we considered that the member that created the auction closes it by invoking `endAuction`. If the auction terminates due to time limit, then it will be a separate auction system thread that will call the `endAuction` method. At least two operations must be executed within the same transaction context (here `T4`) when closing an auction: `deposit` a certain percentage of the amount of the final bid on the system account as a commission, and `deposit` the rest of the amount of the final bid on the seller's account (here member 1's account). Another operation that might also be performed when an auction finishes is to mark it as `closed`, so no other bids can be made.

4.1 A Possible Deadlock in the EJB Solution

Due to all these `withdraw-deposit` operations that have to be done on several accounts, a deadlock situation might appear in the EJB Solution.

Consider for instance two auctions and two participants in both auctions. Suppose now that participant A is the current highest bidder in Auction 2, and that participant B is the current highest bidder in Auction 1, and that both overbid each other, i.e., participant A overbids participant B in Auction 1, and participant B overbids participant A in Auction 2. As already presented in the previous section, the `placeBid` method, together with all the four operations that are chained inside it, will be executed within the same transaction context. Fig. 7 presents the scenario where the `placeBid` method invoked by participant A is executed in the transaction context `T1`, and the `placeBid` method invoked by participant B is executed in the transaction context `T2`. We repre-

sented the last two operations inside the `placeBid` method, i.e., `setHighestBidder` and `setHighestBid`, under the name of `update`.

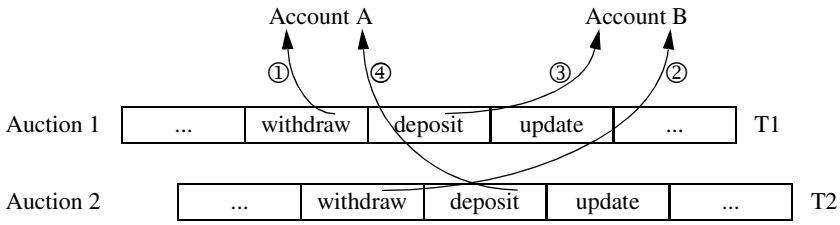

Fig. 7. Deadlock Situation in the EJB Solution

Due to the isolation between transactions (see section 2.2), when the `withdraw` operation will be performed on account A (Fig. 7 ①) from within transaction `T1`, access to this account will be locked until the transaction `T1` finishes. The same happens with the `withdraw` operation on account B (Fig. 7 ②), which locks access to account B until the transaction `T2` finishes. This situation ends in a deadlock since the two `deposit` operations (Fig. 7 ③ , ④) will wait for their target accounts to be unlocked, which will never happen since neither one of the two transactions can finish.

4.2 Proposed Work-Arounds for Avoiding the Deadlock Situation

One solution to avoid the deadlock situation would be to have a certain *random timeout* after which we abort a transaction. In our example, once transaction `T1` aborts, transaction `T2` can continue and commit. In this case, the participant A will have to re-issue a call to the `placeBid` method and hope that this time it would work.

Another solution is to have ordered access to the involved accounts. The order is dictated by the account numbers that are involved in the same `placeBid` method. We will introduce a new operation, called `dummy`, that will be the first operation executed inside a `placeBid` method. The `dummy` operation will target the account with the smallest number with the only purpose of getting its lock. If, for example, we have to `withdraw` from Account `2` and `deposit` in Account `1`, then a `dummy` operation will be performed first on the account that has the smallest number (see Fig. 8).

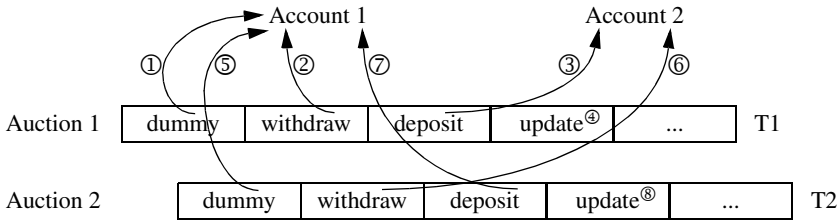

Fig. 8. Proposed Work-around for the Deadlock Situation

In this way, if two `placeBid` methods are dealing with the same two accounts, like in the deadlock situation, they will both try to perform first the `dummy` operation on the same account, i.e., the one with the smallest number, trying to get its lock until the end of the enclosing transaction. One of the transactions will get the lock first. The other one

will have to wait, thus not performing other operations on other accounts and not blocking other transactional objects. The battle is done once, at the beginning, and after that everything should go on smoothly without other blockings.

The order in which the operations inside the two `placeBid` methods will be executed changes due to the anticipated blocking behavior introduced by the `dummy` operation. In Fig. 8 we used encircled numbers to show the exact order in which those operations are going to be executed.

5 Discussion

In this section, we will discuss the drawbacks of the EJB Solution for the auction system, highlighting those limitations in the EJB Transaction Model that led to these drawbacks. We will also point out certain features that would enhance the current EJB Transaction Model.

First of all, the proposed EJB Solution for the auction system does not provide the desired *all-or-nothing* semantics of transactions. In an ideal case, we can imagine having one long-living transaction for each auction, which can be rolled back at any time while the auction is still open, returning the system in the previous consistent state. Such a solution is proposed by [7], where the same auction system is modeled on top of open multithreaded transactions (OMTTs) [8] in a very natural way. In the EJB Solution, we can roll back only small increments. For example, once a new `AuctionBean` object is created, there is no possibility to undo it, only by explicitly removing it from the table in the database. The same thing happens with the `placeBid` method. We can roll it back if something happens while inside, but once it finishes, there is no way to come back to the previous state. With this approach, the system is always in a consistent intermediate state and it will even persist to system crashes, which is not the case for the OMTT Solution, where everything is lost and has to be restarted from the beginning. In the EJB Solution, after a system crash, all the information about the created auctions, about the current highest bidders and bids in all auctions, about the balances of all accounts, will simply be restored from the corresponding tables in the database(s). In this way, all bids made by a member are remembered and s/he can continue exactly from the same state where the system crashed. So, we could say that fault tolerance is provided by the persistency of the underlying database, while in the OMTT Solution, fault tolerance is provided automatically by the underlying transaction support.

Due to the isolation between transactions, bean objects are locked until the transaction that has locked them commits or aborts. From this perspective, we can understand why the EJB Transaction Model does not encourage the usage of long-living transactions. A lot of bean objects can be involved in a long-living transaction, which can reduce system efficiency and throughput, as there is no support for partial rollbacks, early-release locks, savepoints, or compensating actions, like in Sagas [9]. It should be possible to release bean objects during a long-lasting transaction execution. Or, a method-commutativity table should be created for each bean, marking some methods as non-conflicting. This would increase the Container's knowledge about the bean, and, consequently, increase the potential for sharing a particular bean object with other transactions.

Another feature that is offered by OMTTs are nested transactions, which give the developer the possibility to make partial undos by rolling back a subtransaction and all its children, without causing abortion of the whole open multithreaded transaction. In the auction system, such partial undos are related to returning the money to a bidder once s/he gets overbidden. In the EJB Solution, we handle this by having compensation operations in the upcoming transaction. In the OMTT Solution, it is achieved using nested transactions. When a user places a bid, the money is withdrawn from his or her account inside a nested transaction. Later on, if someone places a higher bid, the money is returned to the account by simply aborting the nested transaction.

One step forward towards providing nested transactions in the EJB Transaction Model, would be to support abort-dependent and commit-dependent transactions. In this way, transactions would be able to change their behavior based on the state of another transaction. Once a transaction aborts, the corresponding abort-dependent transactions will also abort. A transaction commits if all its corresponding commit-dependent transactions have already committed. If applications are mostly based on transactions, it is desirable to express bindings and dependencies between them [10], [11].

The EJB specification is not clear regarding multithreaded transactions, contrary to the full support of multithreading in OMTTs. In the Java Transaction API (JTA) [12], however, it is mentioned that each thread has associated a transaction context, which is either *null* or refers to a specific global transaction. The transaction-to-thread association is managed transparently by the Transaction Manager. Multiple threads may concurrently be associated with the same global transaction. This can be achieved by creating or spawning threads from within an already existing transaction context. However, it is not clear how a newly created thread can be associated with a previously started transaction, i.e., how threads can join already existing transactions.

Some other limitations of the EJB Transaction Model that we will not enter into details in this paper are: transactions cannot manage their locks according to the application's requirements, the set of values for the transaction attribute is very limiting, bean methods cannot be associated with several transaction attributes, bean methods cannot be dynamically associated with a particular transaction attribute, no support for asynchronous operations, and no constraint for distributed diamond support.

6 Conclusions

Even if the EJB specification does not require the Bean Provider to have any programming knowledge for concurrency, transactions, and other services, s/he must first accomplish a detailed analysis of all the enterprise beans' methods before starting the configuration of all deployment descriptors. Any misuse of the values of the transaction and isolation level attributes might lead to incorrect applications. Changing the values defined by the Bean Provider for these two attributes is highly error-prone. Only the implementor of the bean knows exactly the semantics of the methods, and is qualified to select the appropriate policies.

By implementing the auction system on top of EJBs, a certain rigidity of the EJB Transaction Model was sensed. We discovered several limitations of the EJB Transaction Model and we proposed work-arounds when possible. A deadlock situation was

identified in the EJB implementation, and some solutions to avoid it were proposed. We also presented certain features that are missing in the EJB Transaction Model and that, we believe, would enhance the current model and would make it more open and flexible.

References

[1] Gray, J.; Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[2] Moss, J. E. B.: *Nested Transactions, An Approach to Reliable Computing*. Ph.D. Thesis, MIT, Cambridge, April 1981.

[3] Sun Microsystems: *Enterprise JavaBeans™ Specification*, v2.0, August 2001.

[4] Sun Microsystems: *Java™ 2 Platform, Enterprise Edition Specification*, v1.4, Proposed Final Draft, August 2002.

[5] IBM: *WebSphere® Application Server*. http://www.ibm.com/websphere/

[6] Vachon, J.: *COALA: A Design Language for Reliable Distributed Systems*. Ph.D. Thesis #2302, Swiss Federal Institute of Technology, Lausanne, Switzerland, December 2000.

[7] Kienzle, J.; Romanovsky, A.; Strohmeier, A.: *Auction System Design Using Open Multithreaded Transactions*. Proceedings of the 7th International Workshop on Object-Oriented Real-Time Dependable Systems, San Diego, California, USA, January 2002. IEEE Computer Society Press, Los Alamitos, CA, 2002, pp. 95 – 104.

[8] Kienzle, J.: *Open Multithreaded Transactions: A Transaction Model for Concurrent Object-Oriented Programming*. Ph.D. Thesis #2393, Swiss Federal Institute of Technology, Lausanne, Switzerland, April 2001.

[9] Garcia-Molina, H.; Salem, K.: *Sagas*. Proceedings of the SIGMod Annual Conference, San Francisco, California, USA, May 1987. ACM Press, pp. 249 – 259.

[10] Elmagarmid, A. K.: *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, 1992.

[11] Jajodia, S.; Kerschberg, L.: *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, 1997.

[12] Sun Microsystems: *Java™ Transaction API (JTA) Specification*, v1.0.1, April 1999.

[13] Sun Microsystems: *Java™ Transaction Service (JTS) Specification*, v1.0, December 1999.

[14] Software Engineering Laboratory: *Open Multithreaded Transactions - The Auction System Case Study*. http://lglwww.epfl.ch/research/omtt/auction.html

[15] Weikum, G.; Vossen, G.: *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers, 2002.