# Auction System Design Using Open Multithreaded Transactions

**Jörg Kienzle, Alfred Strohmeier**
Software Engineering Laboratory
Swiss Federal Institute of Technology Lausanne
CH - 1015 Lausanne EPFL
Switzerland
email: Joerg.Kienzle, Alfred.Strohmeier@epfl.ch

**Alexander Romanovsky**
Department of Computing Science
University of Newcastle
NE1 7RU, Newcastle upon Tyne
United Kingdom
email: Alexander.Romanovsky@newcastle.ac.uk

## Abstract

*Open Multithreaded Transactions form an advanced transaction model that provides features for controlling and structuring not only accesses to objects, as usual in transaction systems, but also threads taking part in transactions. The model allows several threads to enter the same transaction in order to perform a joint activity. It provides a flexible way of manipulating threads executing inside a transaction by allowing them to be forked and terminated, but it restricts their behavior in order to guarantee correctness of transaction nesting and isolation among transactions. In addition, transactions are exception handling contexts, and the model therefore provides forward and backward error recovery. In this paper we show that the model is indeed powerful, and that a complex application, i.e. an online auction system, can be designed and implemented in a very elegant way.*

## 1 Introduction

Modern applications must respond to an increasing number of requirements. To satisfy user expectations, applications offer more and more functionality, and hence grow more complex. Fancy user interfaces, multi-media features or interaction with real-time devices, e.g. sensors, require software to promptly respond to external stimuli and to be able to perform several operations simultaneously. There is also an increasing need for integrating different systems and applications, which results in heterogeneous and possibly distributed systems. Moreover, the ever increasing popularity of the Internet and the growing field of e-commerce have led to an explosion of the number of distributed systems in operation. Such systems typically are required to provide highly available services, and must satisfy hundreds of clients simultaneously.

One approach for satisfying the ever increasing demand in software is to improve software development methods [1, 2, 3], whereas another one is to make available more powerful implementation infrastructures. In our work we focussed on the latter approach.

Concurrent systems can be classified into *cooperative* systems, where individual components collaborate, share results and work for a common goal, and *competitive* systems, where the individual components are not aware of each other and compete for shared resources [4].

Many researchers view all object-oriented systems as inherently concurrent, since objects themselves are "naturally concurrent" entities. In reality, concurrency adds a new dimension to system structure and design. Concurrent systems are extremely difficult to understand, design, analyze and modify.

Sophisticated object-oriented systems often need more advanced and elaborate concurrency features than those offered by current programming languages. Multiple objects must usually be accessed or updated jointly to correctly reflect the real world, hence great care must be taken to keep related objects globally consistent. Any interruption of updates to objects, or the interleaving of updates and accesses, can break the overall consistency of an object system. Transactions [5] address this kind of problems.

This paper is a follow-up paper of [6], in which we introduced in detail the open multithreaded transaction model. Here we describe the design and implementation of an online auction system, an example of a dynamic system with cooperative and competitive concurrency, and how the aforementioned problems have been solved by structuring the application using open multithreaded transactions. Section 2 briefly reviews the open multithreaded transaction model, emphasizing the way transactional objects are handled; section 3 then introduces the auction system case study, and section 4 presents its design based on open multithreaded transactions; section 5 gives insight on the prototype implementation of the application, section 6 discusses the advantages of using open multithreaded transactions and lessons learnt during the case study, and the last section draws some conclusions.

## 2 Open Multithreaded Transactions

The open multithreaded transaction model [6, 7] is a transaction model that provides features for controlling and structuring not only accesses to objects, as usual in transaction systems, but also threads taking part in transactions. The model allows several threads to enter the same transaction in order to perform a joint activity. It provides a flexible way of manipulating threads executing inside a transaction by allowing them to be forked and terminated, but it restricts their behavior in order to guarantee correctness of transaction nesting and isolation among transactions. Lightweight and heavyweight concurrency are treated in the same way in the open multithreaded transactions model, meaning that what is called thread here might as well be a process executed on a single machine or in a distributed setting.

**Life-cycle of an Open Multithreaded Transaction**

Any thread can start a transaction. A thread wishing to work on behalf of an already existing transaction can do so by joining it. In order to join, it has to learn (at runtime) or to know (statically) the identity of the transaction it wishes to join. Threads working on behalf of an open multithreaded transaction are referred to as *participants*. External threads that create or join a transaction are called *joined participant*s; a thread created inside a transaction by some other participant is called a *spawned participant.*

Within an open multithreaded transaction, threads can access a set of transactional objects. Although individual threads evolve independently inside an open multithreaded transaction, they are allowed to collaborate with other threads of the transaction by accessing the same transactional objects.

All participants *finish* their work inside a transaction by voting on the transaction outcome. Possible votes are *commit* and *abort*. Voting abort is done by raising an external exception (see below). The transaction commits if and only if all participants vote *commit*.

Once a spawned participant has given its vote, it *terminates* immediately. Joined participants are not allowed to leave a transaction, i.e. they are blocked, until the outcome of the transaction has been determined. If a participating thread “disappears” from a transaction without voting on its outcome, the transaction is aborted, as this case is considered an error.

**Exceptions and Open Multithreaded Transactions**

The open multithreaded transaction model incorporates disciplined exception handling adapted to nested transactions. It allows individual threads to perform *forward error recovery* by handling an abnormal situation locally. If local handling fails, the transaction support applies *backward error recovery* and reverses the system to its “initial” state.

The model distinguishes *internal* and *external* exceptions; the latter ones are also called *interface* exceptions. The set of internal exceptions for each participant consists of all exceptions that might occur during its execution. There are three sources of exceptions inside an open multithreaded transaction:

- An internal exception can be raised explicitly by a participant.
- An external exception raised inside a nested transaction is raised as an internal exception in the parent transaction.
- Transactional objects accessed by a participant of a transaction can raise an exception to signal a situation that violates the consistency of the state of the transactional object.

All these situations give rise to a possibly inconsistent application state. If a participant does not handle such a situation, the application’s correct behavior can not be guaranteed.

A participant must therefore provide handlers for all internal exceptions. If such a handler is not able to deal with the situation, it can signal an external exception[1].

If any participant of a transaction signals an external exception, the transaction is aborted, the exception is propagated to the containing context, and the exception `Transaction_Abort` is signalled to all other joined participants. If several joined participants signal an external exception, each of them propagates its own exception to its own context.

**Transactional Objects**

In the open multithreaded transactions model participants perform their work by invoking operations on transactional objects. To guarantee the ACID properties, operation invocations made by participants must be controlled at two levels. Guaranteeing the isolation property of all updates made within a transaction with respect to other transactions running concurrently is the first concern. This can be achieved using existing optimistic or pessimistic concurrency control techniques. The second concern is data consistency. To ensure correct updates, data access operations performed concurrently by participants of the same transaction must be executed within mutual exclusion. This can be achieved by using monitors or similar techniques found in modern concurrent languages.

Early error detection and consistency of application state is important for modern applications. Existing transactional systems mainly rely on the programmer to write a transaction in such a way that it preserves consistency. In the open multithreaded transaction model, developers of transactional objects can help the application programmer to write consistency-preserving code by developing so-called *self-checking transactional objects*

1. If a participant “forgets” to handle an internal exception, the external exception `Transaction_ Abort` is signalled, and the application consistency is restored by aborting the transaction.

that encapsulate invariants. For such objects, methods are decorated with pre- and post-conditions. When an invariant, a pre- or a post-condition is violated by the execution of a method, an exception is propagated to the participant that has invoked the operation. [1]The participant must then handle this internal exception in order to address a potential inconsistency. If handling fails, the transaction is aborted, all the changes made to transactional objects on behalf of the transaction are undone and an external exception is propagated to the calling context.

**Object Creation and Deletion**

Most data stored in transactional objects is persistent, i.e. it continues to exist when the application terminates. An application might want to create new objects, but also use objects created during previous runs. This is why such objects must provide three housekeeping operations: `Create`, `Restore` and `Delete`.

`Create` physically creates the object on some storage unit, whereas `Restore` attempts to read a previously stored state from the storage unit. `Delete` deletes the data stored on the unit.

In order to provide atomicity and durability in the presence of crash failures, the transaction support must be able to redo the operations of a committed transaction that have been accidentally lost, or to undo the effects of a partially executed transaction.

The effect of a transaction consists in the set of state changes of all accessed transactional objects, but may also include the creation and deletion of transactional objects. If a transaction that created new objects aborts, these objects must be deleted, and likewise, if objects have been deleted, they must be recreated. The transaction support must therefore monitor all creation and deletion operations made on transactional objects, and record them, together with state information of the objects for deletion operations. Only then the transaction support is able to reverse the creation or deletion operations in case the transaction aborts.

## 3 Case Study Description

The informal description of the auction system presented in this section is inspired by the auction service example presented in [9], which in turn is based on auction systems found on various internet sites, e.g. `www.ebay.com`, `www.ubid.com` or `www.ibazar.com`.

1. There is a considerable body of research related to designing objects / classes together with developing their pre- and post-conditions and invariants, as well as to developing executable conditions to be checked at run-time. The best known example is B. Meyer's “design by contract” methodology supported by features of Eiffel [8].

**General Requirements**

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 are supported, namely *English auctions*, *Dutch auctions*, *1st Price auctions*, *2nd Price auctions*, etc.

The *English auction* is the most well-known form of 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 reset with every new bid can be associated with the auction.

In a *Dutch auction*, the starting price is set to a high price. Then, following a predefined interval, e.g. once per day, this price is lowered by a certain amount. The first bidder wins the auction.

During a *1st Price auction*, all bidders place one secret bid. When the auction closes after a specified amount of time, the bidder that made the highest bid wins the auction. The *2nd Price auction* is based on the same principle. However, the winner, i.e. the bidder that placed the highest bid, must pay only the amount of the next best bid.

**Registration**

Any client interested in using the auction system services must first register with the system by filling out a *registration form* on which he or she must provide his or her real name, postal address and email address, and a desired username and password.

Moreover, 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 a bank account under control of the auction system. When bidding for goods, the sum of the bids placed by a client may never exceed the money available on his or her account.

Once the registration process is completed, the client becomes a *member* of the auction system.

**Login**

A member of the auction service that wants to make use of the system must first login to the system using his or her username and password. Once logged, the member may choose from one of the following possibilities:

- Start a new auction,
- Browse the current auctions,
- Participate in an ongoing auction,
- Consult the history of other members, or
- Deposit or withdraw money from his or her account.

**Starting an Auction**

A member wanting to start a new auction must fill out an *item form* describing the item to be put up for auction. Required information includes a title, a detailed description of the item the member wants to sell, and an opening bid. In addition, the type of auction to be used must be specified.

Once the item form has been submitted successfully, the system starts the auction and inserts it into the list of current auctions.

**Browsing the List of Current Auctions**

Any member logged into the auction system is allowed to browse the list of current auctions. The information available in the current auction list is the title of the auction, the auction type, the description of the item for sale, the expiration date, and other data related to the specific kind of auction.

**Participating and Bidding in an Auction**

While browsing the list of current auctions, a member can decide to participate in one or several of them. To bid on an item, a participant simply has to enter the amount of the bid. A valid bid must fulfill the following requirements:

- The amount left on the bank account of the member that wants to place a bid is at least as high as the sum of all his or her pending bids plus the new bid. This requirement ensures that a member is always in the position to pay for all items he or she placed bids on.
- The member placing the bid is not the member having started the auction. This rule prohibits a seller to bid in his or her own auction.
- The auction has not expired.
- In *English auctions*, the new bid must be higher than the current bid. If nobody has placed a bid yet, then the bid must be at least as high as the opening bid.
- In *Dutch auctions*, the new bid is usually equal to the current bid. In general, bids that are higher than the current bid are also accepted.
- In *1st Price auctions* and *2nd Price auctions*, the new bid must be at least as high as the opening bid.
- If any of the previously stated requirements is not met, the auction system rejects the bid.

**Closing an Auction**

The time of closure of an auction depends on the type of auction. If an auction closes, and no participant has placed a valid bid, then the auction was unsuccessful. In that case, the auction system does not charge any money for the provided services.

If the auction closes and at least one valid bid has been made, then the auction ends successfully. In that case, 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 two percent, which is deposited on the account of the auction system for the provided services.

**Member History**

The auction system keeps track of all auctions started or won by a member. Any member can consult the history of other members.

**Delivery of the Goods**

The auction site `Ibazar`, for instance, trusts its members to effectively send the goods that have been sold in an auction to the winning member. In these systems, the winning member can, once he or she has received the item, vote on the quality of the delivery. This vote will be registered in the history of the seller.

Other systems provide a special escrow service that blocks the money of the winning bidder until the seller sends the goods. Only when the goods have been received and the bidder is satisfied, the money gets transferred to the seller account.

**Fault-Tolerance Requirements**

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 Application Design

The auction system is an example of a dynamic system with cooperative and competitive concurrency. The concurrency originates from multiple connected members, who each may participate in or initiate multiple auctions simultaneously. Inside an auction, the members cooperate by bidding for the item on sale. On the outside, the auctions compete for external resources, such as the user accounts. The system must be dynamic, since a member must be able to join an ongoing auction at any time.

To deal with this complexity, the design of the auction system is based on open multithreaded transactions.

**Transactional Objects in the Auction System**

Any data used from within a transaction and also any data that should survive crashes must be encapsulated inside a transactional object.

The transactional objects needed for implementing the auction system are fairly easy to identify. Every concrete transactional object must implement the operations `Create`, `Restore` and `Delete`. The relationships among the classes are presented in Figure 1 (for brevity, `T_` stands for `Transactional_`).

Registration and history information of members are stored in the `Transactional_Member_Information` class. In addition to providing a constructor method `Create`, the class also offers a method that allows a member to change password, and a method `Add_To_History` that appends a successful auction to the history of a user.

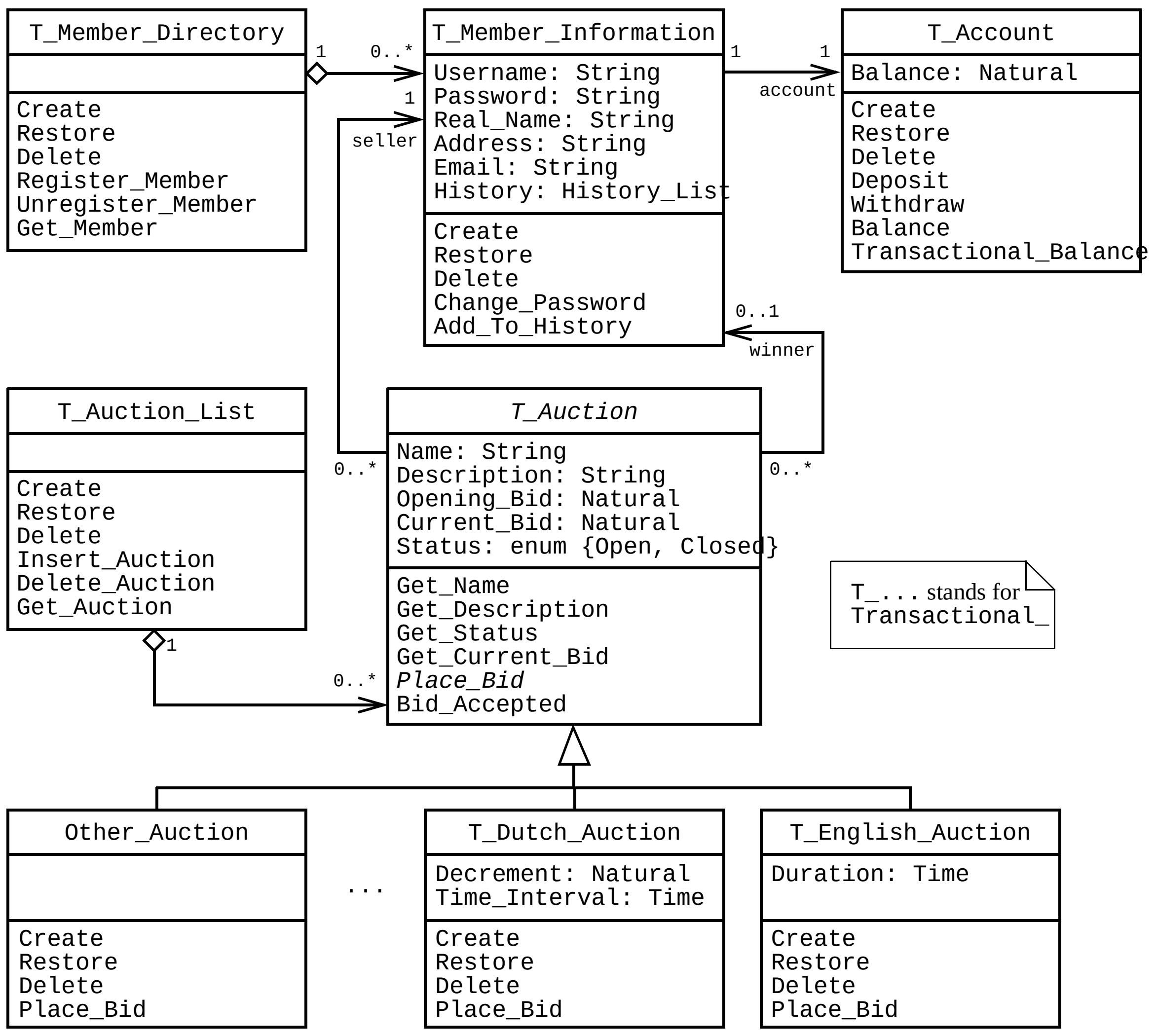

**Figure 1:** Transactional Objects found in the Auction System

The Transactional_Member_Directory class defines the set of all registered members. It provides methods to register and unregister members, and to retrieve the Transactional_Member_Information object of a member.

The abstract Auction class represents auctions. It defines methods to query information about the auction, and a method that can be called by a bidder to know if a previously placed bid has been accepted. The Auction class has several concrete subclasses, e.g. English_Auction and Dutch_Auction, that define the different auction types. Every subclass must implement the required operations Create, Restore and Delete, plus the Place_Bid operation.

The Auction_List class contains the list of all current auctions. It provides the usual operations available on lists, i.e. Insert_Auction, Remove_Auction and Get_Auction.

Finally, every member has a bank account, represented by the Transactional_Account class.

**The Transactional_Account Class**

The Deposit, Withdraw and Balance operations have the usual semantics. A Withdraw is only possible if there is enough money on the account, since members are not allowed to overdraw their account. If this is not the case, the exception Not_Enough_Funds is raised.

In addition, the Transactional_Account class provides the required housekeeping operations Create, Restore and Delete required for transactional objects, plus a Transactional_Balance operation. This operation is similar to the Balance operation, but it allows one to query the current balance of the account even if other active transactions have modified the balance, thereby relaxing the isolation among concurrent transactions. Such an operation is necessary, for it allows the owner of the account to query how much money is available for new bids in case his or her other bids placed in other auctions are accepted.

In order to maximize concurrent execution, the operations of the transactional account class must be analyzed to determine conflicting operations. Strict concurrency control designates Withdraw and Deposit as *modifi-*

*ers*, `Balance` and `Transactional_Balance` as *observers*. Unfortunately, for accounts, strict concurrency control unnecessarily restricts concurrency. Analyzing the semantics of the operations of the `Transactional_Account` class reveals that some of them commute.

The compatibility table for the operations of the `Transactional_Account` class is given in Figure 2. It is based on backward commutativity.

| | Deposit(y) | Withdraw(y) | Balance | Trans_Bal |
|---|---|---|---|---|
| Deposit(x) | yes | yes | no | yes |
| Withdraw(x) | no | yes | no | yes |
| Balance | no | no | yes | yes |
| Trans_Bal | yes | yes | yes | yes |

**Figure 2:** Compatibility Table for the `Transactional_Account` Class

Note that the table is not symmetric. A `Deposit` operation commutes with a `Withdraw` operation, but `Withdraw` does not commute with `Deposit`. This is due to the fact that the `Withdraw` operation can not be completed successfully if there is not enough money on the account. An uncommitted `Deposit` operation could give the illusion that a withdraw is possible, but if the deposit is rolled back later on, the withdraw would not be valid anymore.

The `Transactional_Account` is an example of a *self-checking transactional object*. If a withdraw can not be completed, the exception `Not_Enough_Funds` is raised. The participant of the open multithreaded transaction that invoked the `Withdraw` operation is forced to address this abnormal situation by providing a local exception handler. Otherwise, the exception crosses the transaction boundary and the transaction is aborted.

## Open Multithreaded Transactions in the Auction System

Open multithreaded transactions are used throughout the design of the auction system according to the following rules:

- Any operation that might potentially interfere with other operations executed concurrently must be encapsulated inside a transaction.
- Any set of operations that should never be executed partially must be encapsulated inside a transaction. This includes also the creation of several transactional objects that logically belong together.
- Any set of operations that might have to be undone must be encapsulated inside a transaction or subtransaction.
- Threads that want to cooperate by accessing the same transactional objects must be participants of the same open multithreaded transaction.

The following two sections present the design of two open multithreaded transactions found in the auction system. The *Registration* transaction uses a single-threaded, non-nested transaction, whereas the *English auction* transaction is based on multithreaded, nested transactions.

### Registration Transaction

A client wanting to become a member of the auction system must first register with the system by filling out the *registration form*. As a consequence, a new `Account` with the initial deposit is created for the member. Then, a new `Member_Information` object is created, initialized with all relevant data from the registration form and a reference to the new account, and finally inserted into the `Member_Directory`.

These three operations, namely creating the `Account` and `Member_Information` objects and updating the `Member_Directory`, must be performed atomically, since a partial execution, e.g. creating the `Member_Information` object without registering it in the `Member_Directory`, would lead to permanent storage leaks. The two `Create` operations and the `Register` operation are therefore encapsulated in a transaction as shown in Figure 3. In this case, the structure is identical to the one of a flat transaction, namely single-threaded, without subtransactions.

### English Auction

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 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.

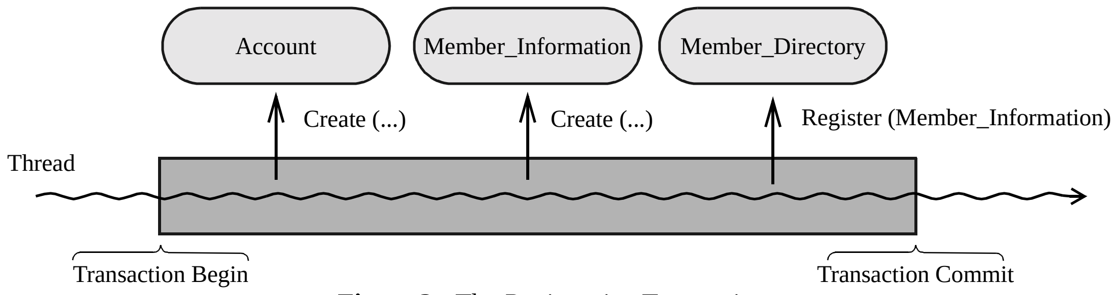

**Figure 3:** The *Registration* Transaction

Auctions are complicated interactions among multiple participants. They incorporate cooperative and competitive concurrency. The participants of an auction cooperate by placing bids on the same item. Members are allowed to participate in several auctions at the same time. Concurrently executing auctions compete for the money on the member accounts.

The number of participants of an auction is not fixed in advance. Therefore, auctions must also be dynamic: new participants must be able to join the auction at any time.

All these requirements can be met if an individual auction is encapsulated inside an open multithreaded transaction. A graphical illustration of an *English auction* is shown in Figure 4.

Every member creates a new thread that acts on behalf of the member inside the auction transaction. As a result, members can participate in multiple auctions at the same time.

In Figure 4, member 1 starts a new auction, creating a new seller thread. Once the item form has been completed, a new open multithreaded transaction, here named `T1`, is started. Then, the seller creates a new auction object by invoking the `Create` function, and inserts it into the list of current auctions.

Other members consulting the current auction list will now see the new auction. In our example, member 2 decides to participate. A new bidder thread is created, which joins the open multithreaded transaction `T1`. It queries the amount of the current bid by invoking the `Get_Current_Bid` operation on the auction object. Before placing the bid, a new subtransaction, here named `T1.1`, is started. Within the subtransaction, the required amount of money is withdrawn from the account of member 2. Since there is enough money on the account, the withdrawal completes successfully and the bid is announced to the `Auction` object by calling `Place_Bid`.

In the meantime, some other member, member 3, joins the auction, spawning also a bidder thread, which joins the open multithreaded transaction `T1`. After consulting the current bid, member 3 decides to overbid member 2. Again, a subtransaction is started, here named `T1.2`, and the required amount of money is withdrawn from the account of member 3. The new bid is announced to the `Auction` object by calling `Place_Bid`. Once the bidder thread of member 2 gets to know this, it consequently aborts the subtransction `T1.1`, which in turn rolls back the withdrawal performed on the account of member 2. The money returned to the account of member 2 can now be used again for placing new bids.

In the example shown in Figure 4, no other bidder enter the auction, nor does member 2 try to overbid member 3. The bidder thread of member 2 has therefore completed its work inside the auction, and commits the global transaction `T1`. Once the auction closes, the bidder

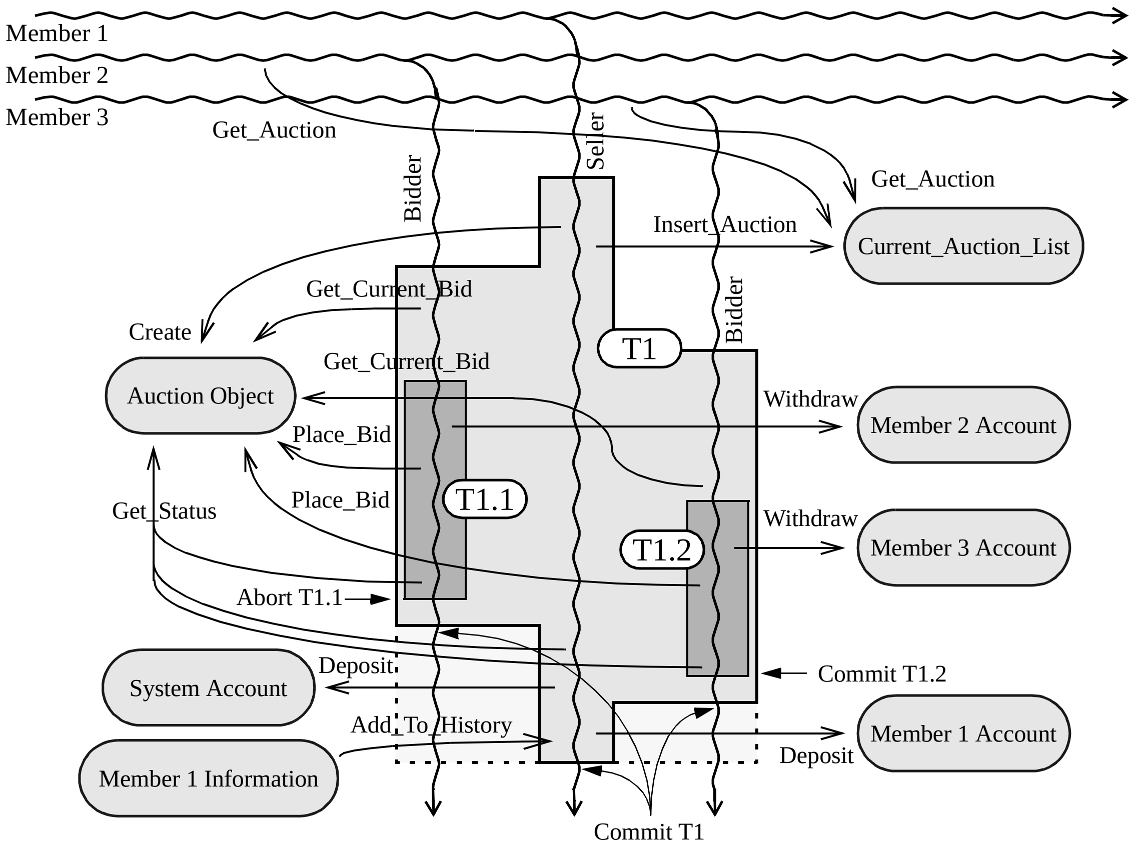

**Figure 4:** The *English Auction* Transaction

```
task body Bidder_Task is
   Auction : Transactional_Auction_Ref := …
   My_Account : Transactional_Account_Ref:= …
   Auction_Title : String := … ①
   Current_Bid, My_Bid : Natural;
   Auction_Transaction : Transaction
      (new String’ (Auction_Title); ②
begin
   while Get_Status (Auction.all) /= Closed
      loop ⑥
      Current_Bid :=
         Get_Current_Bid (Auction.all); ④
      GUI.Get_Bid_From_User (My_Bid,
                             Current_Bid); ⑤
      begin
         declare
            Subtransaction : Transaction; ⑦
         begin
            Withdraw (My_Account.all, My_Bid); ⑧
            Place_Bid (Auction.all, My_Bid); ⑩
            while Bid_Accepted (Auction.all)
               and Get_Status (Auction.all)
                  /= Closed loop
               delay A_While; ⑪
            end loop;
            if Bid_Accepted (Auction.all) then
               Commit_Transaction
                  (Subtransaction); ⑫
            else
               GUI.Notify_User (Overbid);
               raise Transaction_Abort; ⑬
            end if;
         exception
            when Not_Enough_Funds =>
               GUI.Notify_User
                  (Not_Enough_Funds);
               raise Transaction_Abort; ⑨
         end;
      exception
         when Transaction_Abort => null; ⑭
      end;
   end loop;
   Commit_Transaction (Auction_Transaction); ⑮
end Bidder_Task; ⑯
```

**Figure 5:** Implementation of the *Bidder* Task

thread of member 3 gets to know that it has won the auction. It then commits the subtransaction `T1.2`, which confirms the previous withdrawal. It also commits the global transaction `T1`. The seller thread in the meantime deposits two percent of the amount of the final bid on the account of the auction system as a commission, deposits 98% of the amount of the final bid on the account of member 1, inserts the `Auction` object into the history of the `Member_Information` object of member 1, and finally also commits `T1`.

Only now that all participants have voted commit, the transaction support will make the changes made on behalf of `T1` persistent, i.e. the creation of the auction object, the bidding, the withdrawal from the account of member 3 (inherited from subtransaction `T1.2`), the deposit on the auction system account, the deposit on the account of member 1, and the insertion of the auction object into the history of the `Member_Information` object of member 1.

## 5 Implementation

[7] describes the design of an object-oriented framework called OPTIMA, which provides the necessary runtime support for open multithreaded transactions. Since applications from many different domains can benefit from using transactions, it is important to allow an application programmer to customize the framework. This flexibility is achieved with the help of design patterns. Class hierarchies with classes implementing standard transactional behavior are provided, but a programmer is free to extend the hierarchies to tailor the framework to the application-specific needs. The framework supports among others optimistic and pessimistic concurrency control, different recovery strategies (i.e. Undo/Redo, NoUndo/Redo, Undo/NoRedo), different caching techniques, different logging techniques (i.e. physical logging and logical logging), different update strategies (in-place and deferred), and different storage devices. Among pessimistic concurrency control, the framework provides built-in support for lock-based concurrency control, with strict read / write or commutativity-based locking. The feasibility and the elegance of the interface for application programmers depend on the available features of the programming language.

The OPTIMA framework has been implemented for the concurrent object-oriented programming language Ada [10]. It has been realized in form of a library based on standard Ada only. This makes the approach useful for all settings and platforms which have standard Ada compilers. Based on the features offered by Ada 95, a procedural, an object-based and an object-oriented interface for the transaction framework have been implemented.

**Auction System Implementation**

The implementation of the auction system is based on the implementation of the OPTIMA framework for Ada. Figure 5 shows the Ada code of the thread `Bidder_Task` that represents a user that wants to bid for an item in an English auction. The `Bidder_Task` collaborates with the `Seller_Task` through the `Transactional_Auction` object.

The references to the user account object and auction object are handed to the task during initialization, as well as the name of the auction ①. The `Bidder_Task` then wants to join the open multithreaded transaction previously started by the `Seller_Task`.

The sample code uses the object-based interface to the transaction support. Joining a transaction is done in this case by declaring a transaction object, here named `Auction_Transaction`, passing to the constructor the name of the transaction to join (in this case, the name of the auction is used to identify the desired transaction) ③. As a result, the entire Ada block, i.e. from ③ to ⑯, is executed from within the transaction.

The `Bidder_Task` now obtains the current bid from the `Auction` object ④, and asks the member to place a bid ⑤. As long as the auction is not closed, the member is allowed to place a bid ⑥.

If the user places a bid, a new subtransaction is started by opening a new Ada block and declaring another transaction object ⑦. An attempt is made to withdraw the required money from the member's bank account ⑧. If there is not enough money on the account, the account object raises the `Not_Enough_Funds` exception. The internal exception is handled locally: a notification is sent to the user, and the subtransaction is aborted ⑨. If the withdrawal succeeds, the bid is sent to the `Auction` object ⑩.

Now, the `Bidder_Task` must wait until either the auction closes, or some other bidder places a higher bid ⑪. If finally the bid is accepted, the subtransaction is committed ⑫. Otherwise, the user is notified that someone else has placed a higher bid, the subtransaction is aborted, resulting in a rollback of the `Withdraw` operation, and the user is given a chance to place a new bid ⑬.

Each time a member overbids some other member, a subtransaction is aborted. Attempts to overdraw a member account also result in aborting a subtransaction. Such rollbacks are part of the normal life cycle of an auction, and should not affect the outcome of the auction in general. This is why the external exception `Transaction_Abort` propagated by the subtransaction upon rollback can be safely ignored ⑭.

In any case, once the auction closes, the global transaction is committed by invoking the `Commit_Transaction` operation on the `Auction_Transaction` object ⑮.

## 6 Discussion

**Advantages of using Open Multithreaded Transactions in the Auction System Case Study**

Using open multithreaded transactions in the design of the auction system has greatly simplified the problems introduced by the inherent concurrency and dynamicity of the system. The application state remains consistent in spite of concurrent auctions. Executing an individual auction inside a transaction automatically provides the desired all-or-nothing semantics: either the auction completes as a whole, or no money is transferred between accounts. Fault tolerance is also provided automatically by the underlying transaction support.

Inside an auction, partial undo functionality has been achieved using nested transactions. When a user places a bid, the money is withdrawn from his account inside a nested transaction. Later on, if someone places a higher bid, the money is returned to the account by aborting the nested transaction. That way, a user that participates in several auctions concurrently can not cheat and overdraw his account.

**Graphical User Interface**

During implementation it turned out that it was not straightforward to connect the graphical user interface and the threads taking part in an open multithreaded transaction.

Standard graphical user interfaces are single-threaded, e.g. there is a single thread that executes the callbacks registered for buttons, etc. In the auction system, a separate thread represents a user in an auction as a participant of the transaction. Commands from the user interface must be sent to this thread, because it only is allowed to work on behalf of the transaction. It was therefore necessary to build a bridge between the graphical user interface and the business logic of the application. Commands from the user result in callbacks from the GUI, which are then dispatched to the corresponding threads.

## 7 Conclusion

The auction system case study has shown how the inherent complexity of a dynamic, distributed, concurrent application can be reduced by structuring the execution with open multithreaded transactions. Reasoning about fault tolerance issues and consistency of the overall system is also made a lot easier. Without open multithreaded transactions, the application programmer would have to deal with threads, object creation and deletion, and exception propagation in an ad hoc way, making the process error prone.

Due to the isolation property and disciplined exception handling, open multithreaded transactions act as firewalls for errors, since they can not propagate to the outside. This fact, together with the ability to nest open multithreaded transactions, makes them ideal units of fault tolerance.

The prototype implementation of the OPTIMA framework and the auction system can be downloaded from `http://lglwww.epfl.ch/research/OMTT/optima.html`.

## 8 Acknowledgements

Jörg Kienzle has been partially supported by the Swiss National Science Foundation project FN 2000-057187.99/1. Alexander Romanovsky has been partially supported by the EC IST RTD Project on Dependable Systems of Systems (DSoS).

## 9 References

[1] S. Sendall and A. Strohmeier: “From Use Cases to System Operation Specifications”. In S. Kent and A. Evans (Eds.), *UML’2000 - The Unified Modeling Language: Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000*, pp. 1–15, Lecture Notes in Computer Science **1939**, Springer Verlag, 2000.

[2] M. M. Kandé and A. Strohmeier: “Towards a UML Profile for Software Architecture”. In S. Kent and A. Evans (Eds.), *UML’2000 - The Unified Modeling Language: Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000*, pp. 513–527, Lecture Notes in Computer Science **1939**, Springer Verlag, 2000.

[3] S. Sendall and A. Strohmeier: “Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML”. In *UML 2001 - The Unified Modeling Language: Modeling Languages, Concepts and Tools, Fourth International Conference, Toronto, Canada, October 1-5, Martin Gogolla (Ed.),* pp. 391 – 405, Lecture Notes in Computer Science **2185**, Springer Verlag, 2001.

[4] P. A. Lee and T. Anderson: “Fault Tolerance - Principles and Practice”. In *Dependable Computing and Fault-Tolerant Systems*, Springer Verlag, 2 ed., 1990.

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

[6] J. Kienzle, A. Romanovsky, and A. Strohmeier: “Open Multithreaded Transactions: Keeping Threads and Exceptions under Control”. In *Proceedings of the 6th International Worshop on Object-Oriented Real-Time Dependable Systems, Universita di Roma La Sapienza, Roma, Italy, January 8th - 10th, 2001*, pp. 209 – 217, 2001.

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

[8] B. Meyer: *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ 07632, USA, 2 ed., 1997.

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

[10] J. Kienzle, R. Jiménez-Peris, A. Romanovsky, and M. Patiño-Martinez: “Transaction Support for Ada”. In *Reliable Software Technologies - Ada-Europe’2001, Leuven, Belgium, May 14-18, 2001*, pp. 290 – 304, Lecture Notes in Computer Science **2043**, Springer Verlag, 2001.