View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

Porting OMTTs to CORBA

Raul Silaghi[1], Alfred Strohmeier[1], Jörg Kienzle[2]

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

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

[2]School of Computer Science
McGill University
Montreal, QC H3A 2A7, Canada

E-mail: Joerg.Kienzle@mcgill.ca

Abstract. The Common Object Request Broker Architecture standardizes a platform- and programming-language-independent distributed object computing environment. It also provides a standard for several distributed services. The Object Transaction Service provides an object-oriented framework for distributed transaction processing, especially for Online Transaction Processing in business applications. The current CORBA OTS allows multithreading inside a transaction, leaving, however, thread coordination to the application programmer, which can be dangerous. Based on the Open Multithreaded Transaction model, we present in this paper the design of a Thread Synchronization Coordinator, *ThreadSyncCoordinator*, which provides the desired thread control inside a multithreaded transaction. A blocking commit protocol ensures that once in a transaction, a thread cannot leave before the outcome of the transaction has been determined, guaranteeing the ACID properties for multithreaded transactions. We also show how the *ThreadSyncCoordinator* can be used to design and implement complex applications, e.g., an Online Auction System, in an elegant way.

Keywords. CORBA, Object Transaction Service, OMTTs, Transactions, Concurrency.

1 Introduction

Online Transaction Processing is the foundation of the world's business computing. It is the system that ensures that the last two seats on flight LX 1754 to Catania (Sicily) are assigned, *together*, to a honeymooning couple; that the balance printed on your ATM ticket in Taipei *exactly* matches the balance in the bank's central datastore in Zürich; or that the last discovered ancient sarcophagus is promised to only *one* museum. Moreover, this reliability is accomplished even in the face of (noncatastrophic) failure of hardware and software around the system.

Transactions are an important programming paradigm that simplify the construction of reliable business applications. Initially deployed in commercial applications to protect data in centralized databases [1], the transaction concept has been extended to the broader context of distributed data and distributed computation. Nowadays, it is widely accepted that transactions are key to constructing reliable distributed enterprise

applications, ensuring the correct handling of interrelated and concurrent updates of data and providing fault tolerance in the presence of failures.

A transaction is a unit of work comprised of several operations made on one or several shared system resources (also referred to as *transactional objects*), governed by the ACID properties: *Atomicity*, *Consistency*, *Isolation*, and *Durability* [1]. Once a new transaction is started, 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.

Along with the Common Object Request Broker Architecture (CORBA), the Object Management Group defined a set of standard object services. One of these services is the Object Transaction Service (OTS), which provides an object-oriented framework for distributed transaction processing. Besides the fact that the OTS supports flat and nested transactions, it also allows multithreading inside a transaction, leaving, however, thread coordination inside the transaction to the application programmer. Unfortunately, this can be dangerous. For example, a thread can decide to leave the transaction and perform some other operations before the outcome of the transaction has been determined, or a thread can roll back the transaction without notifying the other threads. In the OTS model, threads do not actually "join" a transaction, because the transaction support is not aware of concurrency. Instead, they get associated a *transaction context*, which makes them act on behalf of that transaction. Since transaction contexts can be passed around, a thread might get associated a transaction context even if it is already working within the scope of another transaction. In this case, the previous transaction context associated with the thread is simply discarded. Using such a model, it is very hard to guarantee the ACID properties for multithreaded transactions.

In order to overcome this drawback, we considered the Open Multithreaded Transaction (OMTT) model for controlling and structuring not only accesses to objects, as usual in transaction systems, but also threads taking part in transactions. Based on this model, we designed a Thread Synchronization Coordinator, *ThreadSyncCoordinator*, that sits between the individual client threads and the CORBA OTS, providing safe thread control inside a multithreaded transaction. With the help of this object, client threads are able to explicitly join an ongoing transaction, and get access to the shared transactional resources. A blocking commit protocol ensures that once in a transaction, a thread cannot leave before the outcome of the transaction has been determined. Events are used to signal all participant threads in a multithreaded transaction to vote on the outcome of the transaction.

The outline of the rest of this paper is as follows: Section 2 provides an overview of the CORBA Object Transaction Service, introducing the major components and interfaces, the different application programming models, and the problems related to multithreaded transactions. Section 3 describes briefly the Open Multithreaded Transaction model. Section 4 presents the design of the Thread Synchronization Coordinator that implements the OMTT model on top of CORBA OTS; some issues related to the proposed design are also discussed in this section. Section 5 shows how the Thread Syn-

chronization Coordinator can be used to design and implement an Online Auction System in an elegant way, and Section 6 draws some conclusions.

2 The CORBA Object Transaction Service

In this section, we provide a brief introduction to the Common Object Request Broker Architecture with a special emphasis on the Object Transaction Service and the way it provides transaction support to application developers. Towards the end, we highlight the major problems in the current CORBA OTS with respect to multithreaded transactions.

The Object Management Group (OMG) [2] is a standardization consortium involving more than six hundred international software companies. The Object Management Architecture (OMA) provides a framework which defines the functions supported by the component technology specifications within the OMG. The OMA Reference Model consists of the following components: the Object Request Broker, the Object Services, the Common Facilities, the Application Domains, and the Application Objects.

The *Object Request Broker* (ORB) is a software component that mediates the transfer of messages between distributed objects, hiding the underlying complexity of network communications from developers. The architecture and the specifications of the ORB are described in the Common Object Request Broker Architecture (CORBA) [3]. The CORBA standard enables transparent interoperability between applications in heterogeneous distributed environments. Due to its Interface Definition Language (IDL), CORBA allows the construction of complex applications in the form of a set of interacting software components that may communicate across the boundaries of networks, using different programming languages and operating systems. Mapping specifications exist from IDL to several programming languages, including Java, C++, C, Smalltalk, Python, COBOL, Ada, and Lisp, all available for download from the OMG web site at [2]. According to the standard, ORB implementations for different languages and platforms can work together using the Internet Inter-ORB Protocol (IIOP).

Object Services are a collection of basic services for using and implementing objects. These services are required to construct distributed applications, and are independent of application domains. They should be designed to do one thing *well*, and they should only be as complicated as they need to be. Not all services have to be provided by a CORBA vendor; however, the most important ones, such as Naming, Event, Notification, *Transaction*, Concurrency, Persistence, and Security, usually are.

2.1 The Object Transaction Service

Transaction processing systems have become ubiquitous and are the basis for all facets of commercial applications that rely on concurrent access to shared data. The *transaction paradigm* has been an integral part in designing reliable distributed applications. The *object computing paradigm* has been proven to increase productivity and improve quality in an application development that purports the reuse of components and distributed computing. Amalgamation of these paradigms successfully addresses the business requirements of commercial transaction processing systems.

OMG's CORBA Object Transaction Service (OTS) [4] provides transactional semantics to the distributed objects world. It enables multiple objects that are distributed over a network to participate in a single global transaction. A distributed application can use the IDL interfaces provided by the OTS to perform transactional work involving these distributed objects. While the ORB handles the complexity of network communication between distributed objects, the OTS provides a good framework to implement critical applications in distributed environments by providing transactional integrity.

2.1.1 Transaction Service Architecture

Figure 1 illustrates the major components and interfaces defined by the Transaction Service. The *transaction originator* is an arbitrary program that begins a transaction. The *recoverable server* implements an object with recoverable state that is invoked within the scope of the transaction, either directly by the transaction originator, or indirectly through one or more transactional objects.

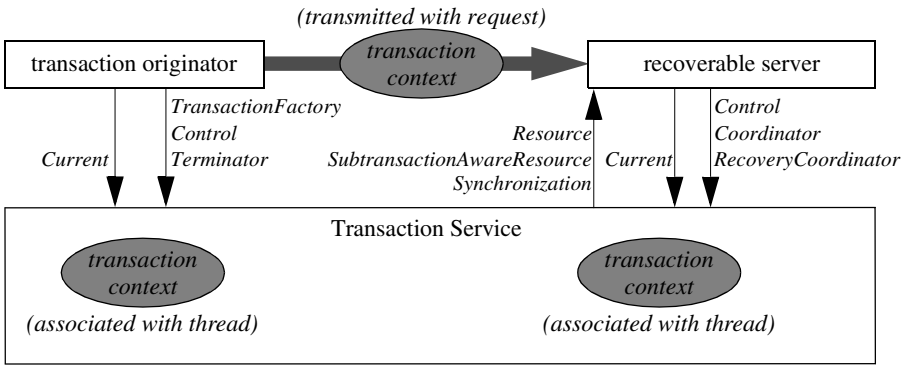

Fig. 1. Major Components and Interfaces of the Transaction Service [4]

The transaction originator issues a request to a *TransactionFactory* to create a new top-level transaction. The factory returns a *Control* object specific to the new transaction. From the developer's point of view, a *Control* object is the transaction representation at the application level. However, the *Control* does not directly support management of the transaction. Instead, it provides access to a *Terminator* and a *Coordinator*. Typically, it is the transaction originator that uses the *Terminator* to `commit` or `rollback` the transaction; however, this is not a constraint. Any thread that gains possession of a *Control* object, e.g., as a result of parameter passing, can get its *Terminator* object and invoke one of the two operations to end the corresponding transaction. The *Coordinator* can also be used to create subtransactions and to test relationships between transactions; however, its main purpose is to serve recoverable servers, which register *Resources* with the *Coordinator*. Each *Resource* implements the two-phase commit (2PC) protocol which is driven by the Transaction Service. A recoverable server may also register a *Synchronization* with the *Coordinator*. The *Synchronization* implements a dependent object protocol driven by the Transaction Service. A *SubtransactionAwareResource*, which tracks the completion of subtransactions, can also be registered with a recoverable server. A *Resource* uses a *RecoveryCoordinator* in certain failure cases to deter-

mine the outcome of the transaction and to coordinate the recovery process with the Transaction Service.

To simplify coding, most applications use the *Current* pseudo object, which provides access to an implicit per-thread transaction context. More details about the transaction context, and the ways to deal with transaction management and transaction context propagation are presented in the next section.

2.1.2 Issues Related to the Transaction Context

A transaction can involve multiple objects performing multiple requests. The scope of a transaction is defined by a *transaction context* that is shared by the participating objects. As part of the environment of each ORB-aware thread, the ORB maintains such a transaction context. The transaction context associated with a thread is either null (indicating that the thread has no associated transaction), or it refers to a specific transaction. A client thread can then issue requests and these requests will implicitly be associated with the client thread's transaction, i.e., they share the client thread's transaction context. It is permitted for multiple threads to be associated with the same transaction at the same time, in the same execution environment or in multiple execution environments, as it is presented in section 2.1.3.

When nested transactions are used, the transaction context remembers the *stack of nested transactions* started within a particular execution environment (e.g., process), so that when a subtransaction ends, the transaction context of the thread is restored to the context in effect when the subtransaction was begun. However, when the transaction context is passed between execution environments, the received context refers only to one particular transaction, not a stack of transactions.

The Transaction Service allows a client program to manage a transaction indirectly or directly. *Indirect* context management implies that the application program uses the `Current` object to associate the transaction context with the application thread of control. *Direct* context management implies that the application program manipulates itself the *Control* object and the other objects associated with the transaction. With these two models in place for managing transactions, propagating the transaction context can happen in two different ways, implicitly or explicitly. With *implicit* propagation, the transaction context associated with the client thread is passed on to the transactional objects without the client's intervention. With *explicit* propagation, the application passes the transaction context to transactional objects as explicit parameters in the method invocation. A client may use either form of context management and may control the propagation of the transaction context by using either method of transaction context propagation. This provides us with four *application programming models* that a client can use to communicate with transactional objects: indirect context management with implicit propagation, direct context management with explicit propagation, indirect context management with explicit propagation, and direct context management with implicit propagation.

The two most used application programming models, indirect/implicit and direct/explicit, are illustrated in Figure 2 a, and Figure 2 b, respectively, by means of short code snippets. Please notice the number of Transaction Service interfaces used in each approach, and the number of parameters in each request involving transactional objects.

```
CORBA.ORB orb = CORBA.ORB.init(...);
CosTransactions.Current tx_crt =
  CosTransactions.CurrentHelper.
    narrow(
      orb.resolve_initial_references(
        "TransactionCurrent"));
...
tx_crt.begin();
...
...
zürichAccount.deposit(amount);
...
...
tx_crt.commit(false);
...
```

a. Indirect and Implicit

```
CORBA.ORB orb = CORBA.ORB.init(...);
CosTransactions.TransactionFactory f =
  CosTransactions.
    TransactionFactoryHelper.narrow(
      orb.resolve_initial_references(
        "TransactionFactory"));
...
CosTransactions.Control control =
  f.create(0);
...
zürichAccount.deposit(amount,
                      control);
...
CosTransactions.Terminator t =
  control.get_terminator();
t.commit(false);
...
```

b. Direct and Explicit

Fig. 2. Application Programming Models

As a final remark, all interfaces defined by the Transaction Service specification [4] are located in the `CosTransactions` module.

2.1.3 OTS Support for Multithreaded Transactions

Using the OTS, a transactional application is not restricted to a single thread within a transaction. To allow multiple threads to participate in a transaction, a reference to the transaction `Control` must be passed to any thread that wills to join the transaction. If the direct/explicit application programming model is used, then this is enough. If the indirect/implicit model is used, then the threads still have to set their implicit transaction context by calling `CosTransactions.Current.resume` and passing the `Control` object as input parameter.

Thus, OTS actually allows multiple threads to access transactional objects on behalf of the same transaction, but without paying special attention to this additional form of cooperative concurrency. Figure 3 depicts such an OTS multithreaded transaction. One thread, here Thread `C`, starts a transaction `T1`. Other threads will eventually learn about the transaction's `Control` object and will be able to access transactional objects on behalf of `T1`. The OTS does not restrict the behavior of these threads in any way. They can spawn new threads, or terminate within the transaction. Any thread can commit or roll back the transaction `T1` at any time, here Thread `B`, and the transaction will be committed or rolled back regardless of what the other participating threads might have voted. Thread exit from a transaction is not coordinated.

As seen in Figure 3, the OTS model is quite general and flexible, and may be suitable for many business applications. However, it leaves thread coordination inside a transaction to the application programmer, and this can be error-prone. For example, a thread can decide to leave the transaction and perform some other operations before the outcome of the transaction has been determined, like Thread `A` in Figure 3. If this thread makes further use of any information that has been computed inside `T1`, e.g., modifies other transactional objects accordingly, then this might lead to information smuggling if `T1` gets rolled back later on. Another unpredictable outcome might arise when a thread rolls back `T1` without notifying the other threads. It might even happen that a

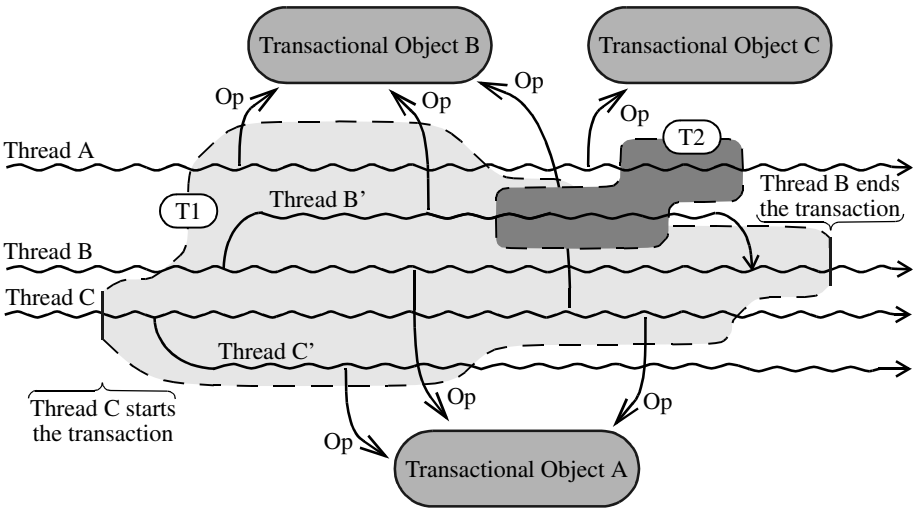

Fig. 3. Multithreaded Transaction in CORBA OTS

thread gets associated with a new transaction context (`Control` object) although it is already working on behalf of another transaction. In Figure 3, for example, Thread `B'` switches directly from `T1` to `T2`. A thread might also forget to vote on the outcome of a transaction, for instance because an exception has caused the program to skip over the `commit` statement. As a result, the transaction will hold resources for a potentially unlimited amount of time. Finally, transactional objects might not be aware of intra-transaction concurrency either. If they do not provide mutual exclusion for update operations, concurrent execution of operations might corrupt their state.

It seems obvious that the CORBA OTS does not really integrate concurrency and transactions; one might better say that concurrency and transactions coexist. The main drawback of this model is that there is no real transaction border, making it hard to guarantee the ACID properties for multithreaded transactions.

3 The Open Multithreaded Transaction Model

In this section we provide a brief overview of the Open Multithreaded Transaction model, stressing out mainly the rules imposed by OMTTs for controlling and structuring not only accesses to objects, as usual in transaction systems, but also threads taking part in transactions.

Open Multithreaded Transactions (OMTTs), first introduced in [5] and then fully described in [6], form an advanced transaction model that 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 when necessary in order to guarantee correctness of transaction nesting and enforcement of the ACID properties.

The life cycle of an open multithreaded transaction is depicted as a state diagram in Figure 4. Any thread can create an open multithreaded transaction becoming its first

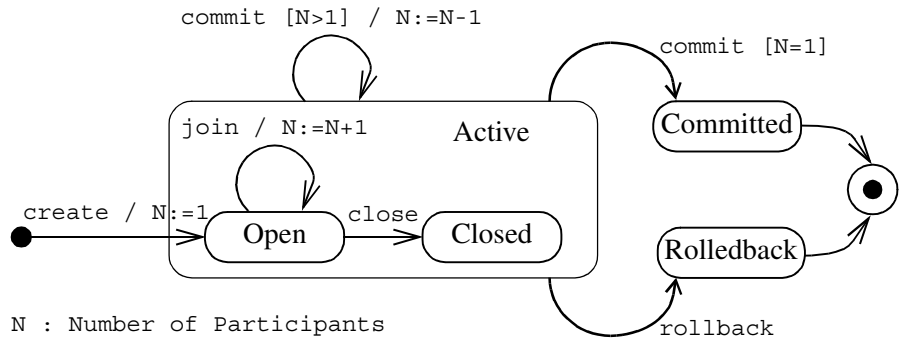

Fig. 4. Life Cycle of an Open Multithreaded Transaction

joined participant. The newly created transaction is said to be *Open*, and as long as it remains as such, other threads are allowed to *join* it, thus becoming joined participants of the transaction as well. A thread can join an open multithreaded transaction if and only if it does not participate in any other transaction. Otherwise, information local to a thread could be passed between transactions that should normally be isolated. Open multithreaded transactions can be *nested*. A participant of a transaction that starts a new transaction creates a nested transaction. Joined participant threads may spawn new threads which automatically become *spawned participants* of the innermost transaction in which the spawning thread participates. Any participant can decide to *close* the transaction at any time. Once the transaction is *Closed*, no new client threads can join the transaction anymore; however, a joined participant can still spawn new threads. All participants *finish* their work inside an open multithreaded transaction by voting on the transaction outcome. The only possible votes are *commit* or *rollback*. In order for a transaction to commit, all its participants must have voted *commit*. If any of the participants votes *rollback*, the transaction is rolled back. Participants are not allowed to leave the transaction (they are *blocked*) until its outcome has been determined. This means that all participant threads of a *committing* open multithreaded transaction exit synchronously. This rule prevents information smuggling by not allowing threads to make use of, or to reveal uncommitted information to the outside world. If a transaction is *rolled back*, the participants may exit asynchronously.

Figure 5 shows two open multithreaded transactions: `T1` and `T1.1`. Thread `C` creates the transaction `T1`, and threads `A`, `B`, and `D` join it. Threads `A`, `B`, `C`, and `D` are therefore *joined participants* of the multithreaded transaction `T1`. Inside `T1` thread `C` forks a new thread `C'` (a *spawned participant*), which performs some work inside the transaction and then terminates. Thread `B` also forks a new thread, thread `B'`. `B` and `B'` perform a *nested transaction* `T1.1` inside of `T1`. `B'` is a spawned participant of `T1`, but a joined participant of `T1.1`. In this example, all participants of `T1` vote `commit`. The joined participants `A`, `C`, and `D` are therefore *blocked* until the last participant, here thread `B`, has finished its work and given its vote.

Even though the OMTT model incorporates several other features, such as disciplined exception handling adapted to nested transactions, we consider they go beyond the purpose of this paper and they will not be addressed here.

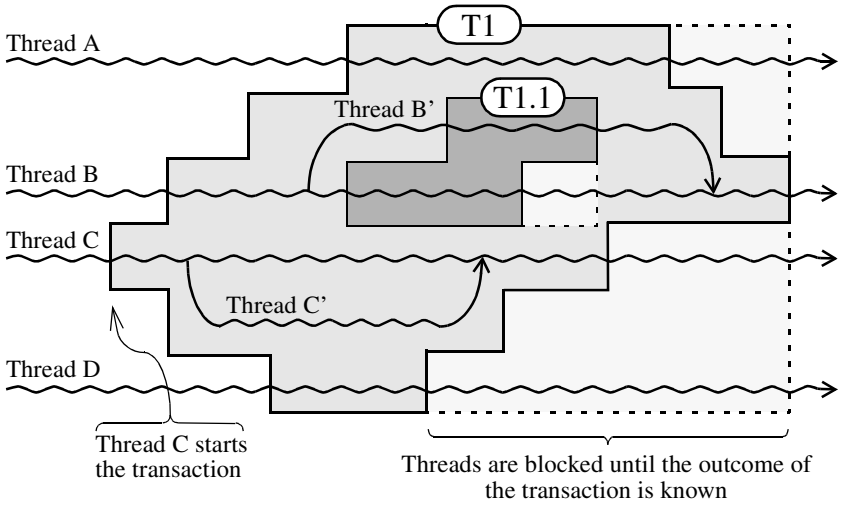

Fig. 5. An Open Multithreaded Transaction

OPTIMA (OPen Transaction Integration for Multithreaded Applications) [7] is the name of an object-oriented framework that provides the necessary run-time support for OMTTs. A prototype of the OPTIMA framework, available for download at [8], has been implemented for the concurrent object-oriented programming language Ada 95. 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, procedural, object-based, and object-oriented interfaces for the transaction framework have been implemented.

4 Porting OMTTs to CORBA

In order to overcome the problems that might appear in CORBA multithreaded transactions, as presented in section 2.1.3, we will present in this section the design of a Thread Synchronization Coordinator, *ThreadSyncCoordinator*, that implements the behavior of OMTTs on top of CORBA OTS. The *ThreadSyncCoordinator* sits between the individual client threads and the CORBA OTS, allowing several threads to explicitly *join* the same transaction in order to perform a joint activity on some shared transactional resources. Moreover, thread control is improved inside a transaction, by enforcing a *blocking commit protocol*, which ensures that once in a transaction, a thread cannot leave before the outcome of the transaction has been determined. With the help of the *ThreadSyncCoordinator* we make sure that the rules imposed by OMTTs, as described in section 3, are respected by the participating threads, so that we can ensure that the ACID properties for CORBA multithreaded transactions are met.

4.1 The Design of the Thread Synchronization Coordinator

The design of the Thread Synchronization Coordinator is shown in Figure 6 by means of a class diagram compliant with the Unified Modeling Language (UML) [9] notation. Although not complete, the diagram shows all classes, attributes, and operations re-

ferred to in the sequel of this section. Design patterns [10], [11] were used in order to maximize modularity and flexibility.

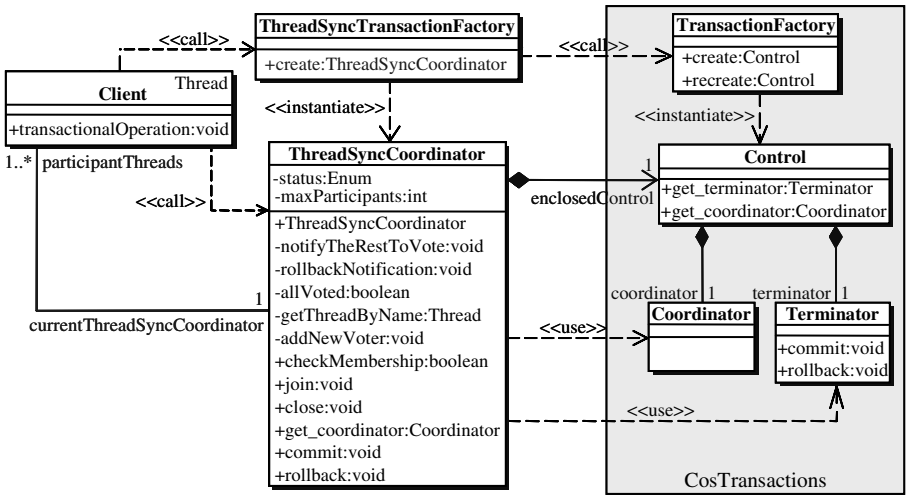

Fig. 6. Design of the Thread Synchronization Coordinator

A `ThreadSyncCoordinator` has attributes for:

- transaction management: the attribute `enclosedControl` links it to a CORBA OTS `Control` object;
- thread management: the attribute `status`, indicating the status of the multithreaded transaction, the attribute `participantThreads`, yielding the list of all threads participating in the multithreaded transaction together with some associated information concerning their vote, and the attribute `maxParticipants`, representing the maximum number of participants in the multithreaded transaction.

A `ThreadSyncCoordinator` makes visible to the clients four operations: `join`, `close`, `commit`, and `rollback`. The creation of a `ThreadSyncCoordinator` is handled following the Factory design pattern by the `ThreadSyncTransactionFactory`, which provides one operation to the client, i.e., `create`.

The sequence diagram presented in Figure 7 illustrates that we use the current facilities offered by the CORBA OTS, and only enhance them with thread synchronization when dealing with multithreaded transactions.

The `ThreadSyncTransactionFactory` acts like a proxy for the `TransactionFactory` provided by the CORBA OTS. However, unlike in the Proxy design pattern, the client is still allowed to use the CORBA OTS support directly (by connecting to the `TransactionFactory` and obtaining `Control` objects), if s/he does not plan to have multithreaded transactions, or if s/he does not care about the blocking commit protocol for multithreaded transactions.

By invoking the `TransactionFactory` of the CORBA OTS, we get a CORBA OTS `Control` object, which is further passed to the constructor of the `ThreadSyncCoordinator`. This `Control` object, which is encapsulated inside the

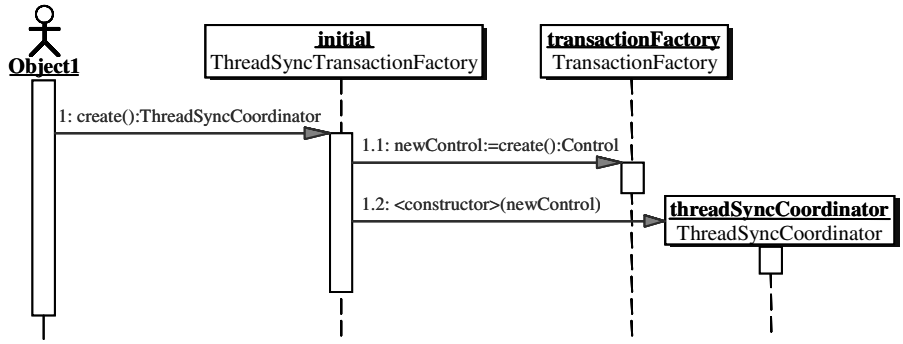

Fig. 7. Creating `ThreadSyncCoordinator` Object Instances

`ThreadSyncCoordinator`, will be used to interact with the transaction support offered by the CORBA OTS. Even if we will require to pass around the `ThreadSyncCoordinator` object, so that other threads can join it and participate in the same multithreaded transaction, we will use the encapsulated `Control` object for registering multiple `Resources` with the same transaction, and thus, make use of the CORBA OTS two-phase commit protocol for transaction completion. A multithreaded transaction with thread synchronization inside is actually mapped to a normal CORBA transaction with an explicit external thread control mechanism. Our `ThreadSyncCoordinator` is just a *wrapper* for the CORBA OTS `Control` object, providing it with additional functionality, i.e., `join`, `close`, `commit`, and `rollback`, for better management of threads that participate in the same transaction (corresponding to the encapsulated `Control` object), for forcing them to vote on the outcome of the transaction, for blocking them to leave before the outcome of the transaction has been determined, and for ensuring in this way that the ACID properties are met for multithreaded transactions as well.

We are not going to enter into concrete implementation details for any of the presented classes or methods, but we will use small code snippets to point out some important aspects that the implementor should follow in order to get the intended functionality. Moreover, we consider that the names of the methods presented in the class diagram are sufficiently eloquent to give the implementor a good hint about their purpose.

4.1.1 Joining and Closing a Multithreaded Transaction

Like in the current CORBA OTS model, where transactions are modeled by using `Control` objects at the application level, multithreaded transactions with thread synchronization are modeled by using `ThreadSyncCoordinator` objects.

A thread can join a `ThreadSyncCoordinator` object, and implicitly the corresponding multithreaded transaction, by simply calling the `join` method and providing its name, e.g., `Thread.currentThread().getName()` in Java (the `Thread` class does not implement `Serializable`, so we can only send its name). In order to achieve this, the client thread has to learn (at run-time) or to know (statically) the identity of the `ThreadSyncCoordinator` it wishes to join. Moreover, the multithreaded transaction needs to be in the *Open* status. Only threads that have already joined the

ThreadSyncCoordinator are allowed to invoke operations on transactional objects on behalf of the enclosing multithreaded transaction.

Only participant threads of a multithreaded transaction are allowed to close the transaction by calling the close method. By closing a multithreaded transaction we are blocking any further joins from other threads. This feature has been introduced for two reasons:

- There might be static systems in which one of the participants (most probably the creating thread) knows how many participants are needed to successfully complete the transaction. In that case, it can specify the number of participants during creation of the ThreadSyncCoordinator object. As soon as this number of participants is reached, the ThreadSyncCoordinator object automatically closes.
- In dynamic systems, i.e., systems where at transaction creation time the number of participants is not known, there is a potential *livelock*, even though all participants behave correctly. In order to successfully commit a multithreaded transaction, all participants must vote commit. However, new participants can arrive at any time. This might lead to the situation where all current participants have decided to commit, but before they can do so, a new participant arrives. It will take some time for this participant to realize that all the work inside this transaction has already been completed. Once it has, it also commits. But during this time, a new participant might have arrived, and so on. In order to prevent this from happening, the transaction must be closed at some point. For some applications, it makes sense to close the transaction as soon as one of the participants has voted commit. Other applications might want to leave the decision to a participant that plays a special role (like the *seller* in the auction system example presented in section 5.2).

A discussion could be opened here on the operations that should be allowed inside a closed multithreaded transaction: whether the participants should be allowed to continue to invoke operations on transactional objects, or they should be constrained to vote on the outcome of the transaction, i.e., the only operations allowed would be commit or rollback. We considered the first approach in our implementation.

4.1.2 Committing or Rolling Back a Multithreaded Transaction

In order to implement synchronous exit, the ThreadSyncCoordinator must have a means to suspend the execution of its participants. In Java this is done using the wait() method. As shown in Figure 8, the ThreadSyncCoordinator object suspends a client thread when it votes commit and when there are still other participants working on behalf of the transaction. Successively, all participant threads will go to sleep in ①, waiting for the last voting thread. The last participant, which will take the then branch, triggers the final commit of the multithreaded transaction by getting the Terminator object of the enclosed Control and calling the CORBA OTS commit on this Terminator, and then wakes up the sleeping threads by calling notifyAll() on the object that was considered for synchronization, in our case the ThreadSyncCoordinator. The CORBA OTS two-phase commit protocol will ensure a

```
public void commit(
    String clientThread) {
 if (! checkMembership(
   clientThread)) return;
 synchronized (this) {
  addNewVoter(clientThread);
  if (allVoted()) {
   Terminator terminator =
    enclosedControl.get_terminator();
   terminator.commit();
   this.notifyAll();
  } else {
   notifyTheRestToVote();
   wait();
  }
 } // end synchronized
 return:  ①
}
```

```
public void rollback(
    String clientThread) {
 if (! checkMembership(
   clientThread)) return;
 synchronized (this) {
  Terminator terminator =
    enclosedControl.get_terminator();
  terminator.rollback();
  rollbackNotification();
  this.notifyAll();
 } // end synchronized
 return;
}
```

Fig. 8. `ThreadSyncCoordinator`'s `commit` and `rollback` Operations

synchronous update of the changes made to different transactional objects on behalf of the committing multithreaded transaction.

A similar approach is used for implementing the `rollback` operation, except the fact that the blocking protocol is not needed anymore. Once a client thread votes `rollback`, all the participant threads may exit asynchronously, and changes made to transactional objects on behalf of the multithreaded transaction will be undone by the CORBA OTS two-phase commit protocol.

Both the `commit` and `rollback` operations begin with a first check of the rights the client thread has within the current `ThreadSyncCoordinator`. Only participant threads are allowed to commit or to roll back the enclosing multithreaded transaction. Also, events ([12], or its evolved successor [13]) are used to signal all participant threads in a multithreaded transaction to vote on the outcome of the transaction once a client thread has voted `commit`, or to just let them know that the multithreaded transaction has rolled back. Of course, if the client does not want to complicate the structure of his or her application by using events, we can imagine that a fixed "reasonable" timeout, decided on a per-application basis, could be set for allowing all the other participants to vote.

An obvious problem that has not been discussed yet are *deserters*, i.e., threads participating in a multithreaded transaction that suddenly disappear without voting on the outcome of the transaction. This can happen if a thread is explicitly killed, or when the process of a participant thread dies incidentally. This special cases are treated as errors, and will cause the multithreaded transaction to roll back, ensuring the all-or-nothing semantics of transactions.

4.2 Discussion

In this paper, we focused only on the blocking commit protocol, which ensures that once in a transaction, a thread cannot leave before the outcome of the transaction has been determined. However, other important issues are still to be addressed. One of the

most important probably is concerning concurrency within the same transaction. Since the OMG's Concurrency Control Service [14] does not address this issue (providing only two ways to acquire locks: on behalf of a transaction, or on behalf of the current thread, but the thread must be executing *outside* the scope of a transaction), it becomes the job of the transactional objects to provide additional concurrency control mechanisms (e.g., *synchronized* methods in Java) for preventing corruption of their state when accessed within a multithreaded transaction.

For the time being, the `ThreadSyncTransactionFactory` and the `ThreadSyncCoordinator` are implemented as stand alone CORBA objects. A potential client has to use the Naming Service [15] to locate them, and only then s/he can make use of their services. Ideally, the provided functionality could be integrated in the future versions of the CORBA OTS specification, so that ORB vendors will have to implement it (if they want to be compliant with the specification) and provide it directly to the application developers.

The chosen name, i.e., Thread Synchronization Coordinator, was very much influenced by the interface and the functionality that is provided to the client. Even though we use a `Factory`, explicit propagation, and encapsulate a CORBA OTS `Control`, the functionality of the `ThreadSyncCoordinator` object is not at all similar to the one provided by a CORBA OTS `Control`. It is not the responsibility of the application programmer to get the `Terminator` and to `commit` the transaction on the CORBA OTS. Instead, the interface is much closer to the CORBA OTS `Current` interface, providing operations like `commit`, `rollback`, and even `begin` in a slightly different way. However, the transaction context propagation is not performed implicitly, which is the case with the CORBA OTS `Current`. Finally, the CORBA OTS `Coordinator` came the closest to the functionality provided. Just as the OTS `Coordinator` is used for registering `Resources` so they participate in the two-phase commit protocol when a transaction ends, the `ThreadSyncCoordinator` allows threads to join (it can be seen as a registration as well) a multithreaded transaction, providing them with a blocking commit protocol until the outcome of the transaction has been determined.

As already shown in Figure 5, nested transactions and spawned threads are also supported within OMTTs, and their use is clearly illustrated in the implementation of the Online Auction System presented in section 5.2. For this, additional information is kept with the `ThreadSyncCoordinator`. For example, the transaction hierarchy, i.e., a list of subtransactions and a reference to the parent transaction, is managed by the `ThreadSyncCoordinator` as well. The blocking commit protocol is enforced at each level of nesting by different `ThreadSyncCoordinator` objects, one for each multithreaded transaction. We also make a difference between *joined* and *spawned* participants, and their role in committing a multithreaded transaction. Two additional rules restrict thread behavior inside and outside of a multithreaded transaction: a thread created inside an open multithreaded transaction must also terminate inside the transaction; and, a thread created outside of an open multithreaded transaction is not allowed to terminate inside the transaction.

The identity of the calling thread, its *name*, needs to be sent as a parameter every time for at least one of the following reasons. First of all, we might need it for updating our local information about the threads participating in the enclosing multithreaded

transaction, like in the case of `join`. Second of all, we might need it for validation purposes, like in the case of `close`, `commit`, and `rollback`. Everytime a client thread invokes an operation on a `ThreadSyncCoordinator` we must ensure that it had joined it previously. Another reason, which is more technical this time, is related to the fact that the thread identity changes when making distributed calls. Client requests are executed in some sort of *TCP-Connection* threads, which have nothing in common with the calling client threads. Moreover, attention must be paid when using Graphical User Interface elements on the client, since all Java GUI interactions are handled inside a special thread, i.e., the *AWT-EventQueue-0*.

The `create` operation provided by the `ThreadSyncTransactionFactory` does not take any parameter, which means that a thread can create as many `ThreadSyncCoordinator` objects as it wants. In order to participate in a multithreaded transaction, a client thread has to explicitly join one of these `ThreadSyncCoordinator` objects. Moreover, in order to be compliant with the OMTT model, it must be ensured that a client thread can join *only one* `ThreadSyncCoordinator`, and thus participate in *only one* multithreaded transaction. Since this check cannot be done at the `ThreadSyncCoordinator` level (it is not natural for a `ThreadSyncCoordinator` to know about all the others `ThreadSyncCoordinators` currently existing in the system), it has to be done at a higher level, where the developer decides to keep track of all ongoing multithreaded transactions (like the `AuctionManager` in the auction system example presented in section 5.3).

A similar functionality could be implemented on top of the CORBA OTS `Current`, so that application developers familiar with the indirect/implicit application programming model can use the benefits of multithreaded transactions without worrying about thread control inside a transaction.

5 Online Auction System Implementation Using Enhanced CORBA Multithreaded Transactions

An Online 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 CORBA multithreaded transactions with thread control provided by `ThreadSyncCoordinator` objects.

5.1 Online Auction System Case Study Description

The informal description of the auction system presented in this section is inspired by the auction service example presented in [16], 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, which will be considered in this case study, 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.

After a first *registration* phase, the user becomes a *member* of the auction system. Then, s/he has to *log on* to the system for each session in order to use the services provided. All members must deposit a certain amount of money to an account under control of the auction system. 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. Each bid is validated in order to ensure that the bidder has sufficient funds, that a bidder does not place bids in his or her own auction, and that the new bid is higher than the current highest bid.

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.

5.2 Enhanced CORBA Multithreaded Transactions: An Elegant Match

The auction system is an example of a dynamic system with cooperative and competitive concurrency. Concurrency originates from the multiple connected members, who may each participate in or initiate multiple auctions simultaneously. Inside an auction, the members cooperate by bidding for the item on sale. On the outside, concurrent auctions compete for external resources, such as the user accounts.

The number of participants in an auction is not fixed in advance. Therefore, auctions must also be dynamic, allowing members to join ongoing auctions at any time.

And, at last, the most important requirement for auctions is to 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.

All these requirements can be met if an individual auction is encapsulated inside a multithreaded transaction, and if the enclosing `ThreadSyncCoordinator` object is provided to the other participants, so that they can join the multithreaded transaction. A graphical illustration of an *English auction* is shown in Figure 9.

Since the OMTT model requires a thread to be participant in *only one* multithreaded transaction, every member must spawn a new thread that will act on his or her behalf inside one particular auction. In this way, the original member thread can continue its

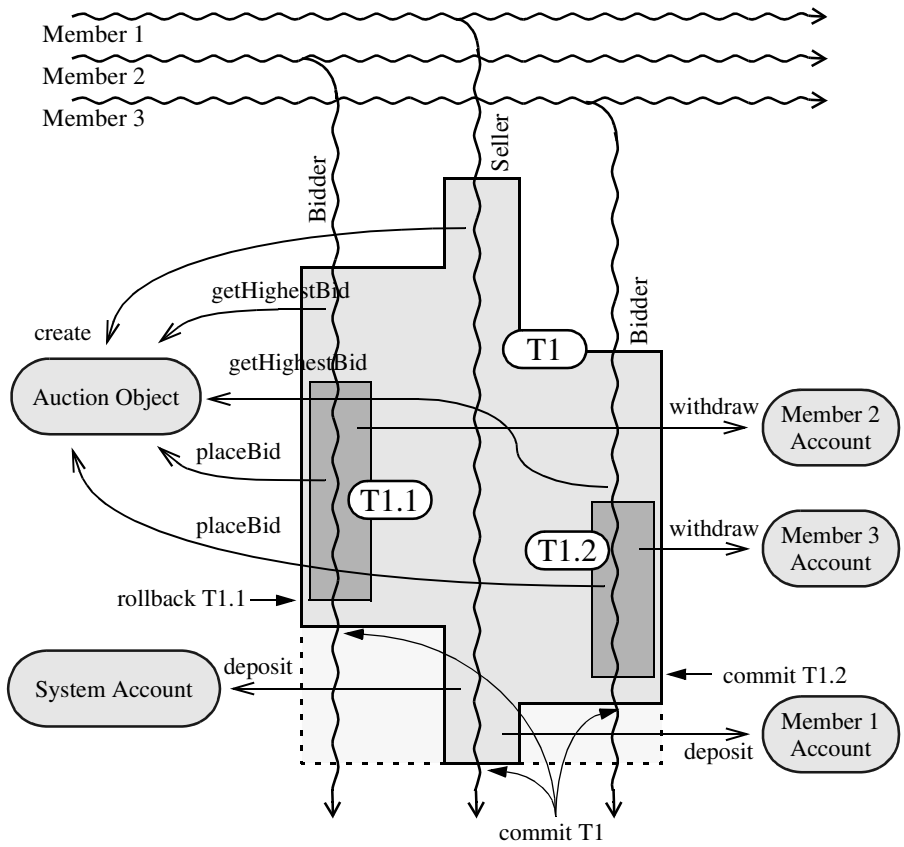

Fig. 9. The English Auction Multithreaded Transaction

work and maybe spawn other threads to join other multithreaded transactions. As a result, a member can participate in several auctions simultaneously.

In Figure 9, member `1` starts a new auction, creating a new *seller* thread. Once the item form has been completed, the `create` method is invoked, which creates a new `ThreadSyncCoordinator` object, and together with it, a new CORBA multithreaded transaction, here named `T1`, is started. Only then, the client auction with the provided parameters is created and automatically added to the list of current auctions. In our example, member `2` decides to participate. A new *bidder* thread is created, which joins the multithreaded transaction `T1` (using, of course, the `ThreadSyncCoordinator` object). It queries the amount of the current bid by invoking the `getHighestBid` method on the auction object. Before placing the bid, a new `ThreadSyncCoordinator` object is created, and together with it, a new CORBA multithreaded 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 `placeBid`. Please notice that at this point,

member 2 has not yet voted on the outcome of the subtransaction T1.1, which means that it can still be either committed or rolled back later on.

In the meantime, member 3 joins the auction, spawning also a *bidder* thread, which joins the 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 placeBid. Once the bidder thread of member 2 gets to know this, it consequently roll(s)back the subtransaction T1.1 (by talking to its corresponding ThreadSyncCoordinator object), 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 9, no other bidders 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. Since the blocking commit protocol is enforced by the associated ThreadSyncCoordinator, the bidder thread of member 2 will be blocked until the outcome of the multithreaded transaction is determined, i.e., until the other two participating threads give their vote.

Once the auction closes, the bidder 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, and finally also commits T1.

Only now that all participants have voted commit, the ThreadSyncCoordinator will invoke the CORBA OTS commit and will let the two-phase commit protocol 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, and the deposit on the account of member 1.

5.3 Online Auction System Design and Implementation

Figure 10 presents a UML class diagram describing our design of the auction system that was previously presented in section 5.1. Inside the auction system, it is the task of the AuctionManager to create auctions, to associate auctions with multithreaded transactions, and to keep track of the current ongoing multithreaded transactions. It is its responsibility to check whether a client thread has already joined a transaction, e.g., joinedSomewhere(), and to block it from joining other transactions. However, it is the responsibility of the ThreadSyncCoordinator to check if a client thread has previously joined it or not, and thus to accept or refuse operations called by a client inside the multithreaded transaction (in our case, operations on Auction objects).

Two UML collaboration diagrams show briefly how an Auction is actually created (Fig. 11) and how client bids are handled by the auction system (Fig. 12). The String in the method signatures represent the client thread's identity, i.e., its name,

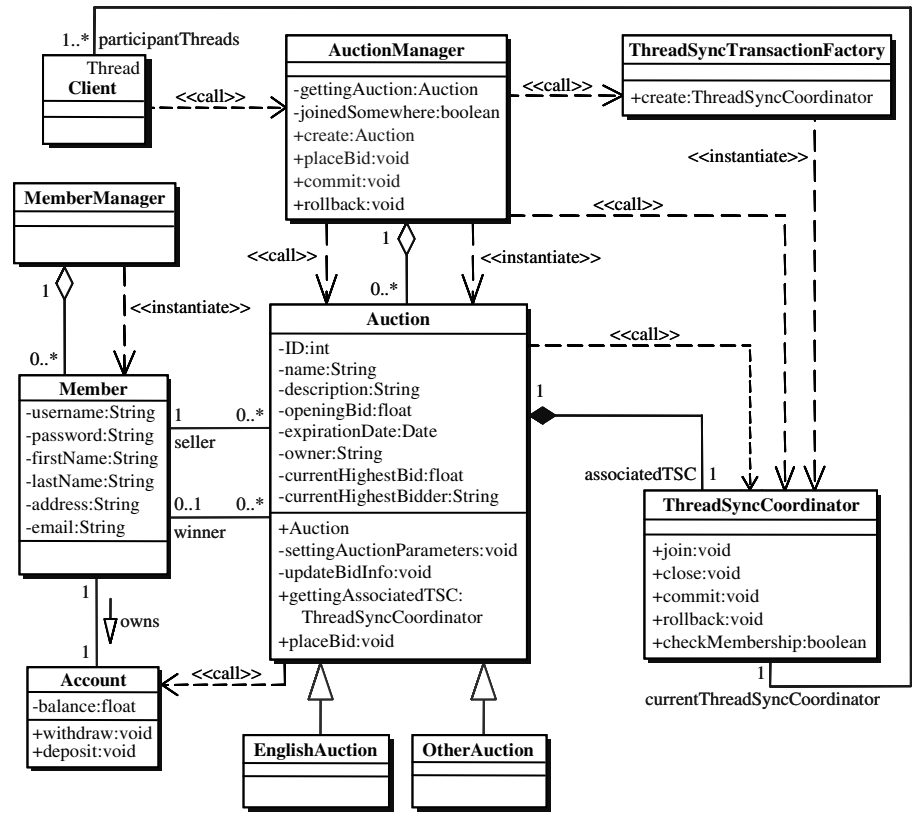

Fig. 10. The Design of the Online Auction System

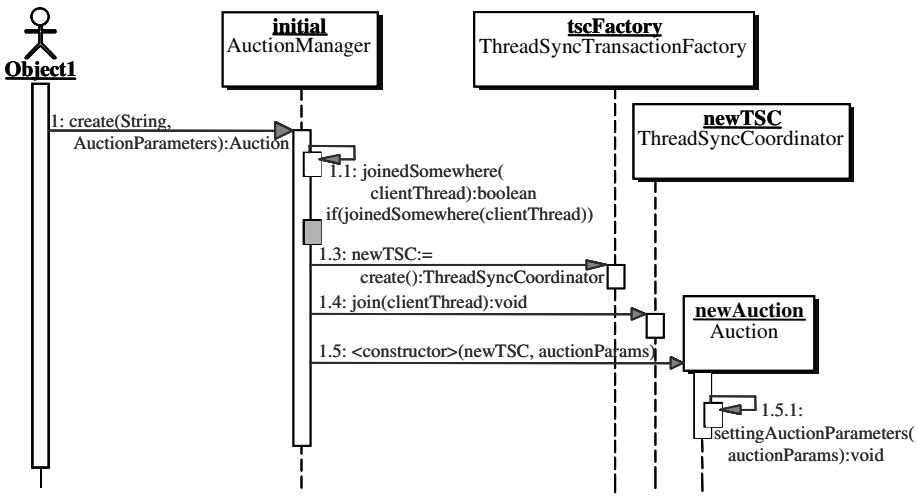

Fig. 11. Creating an Auction inside the Auction System

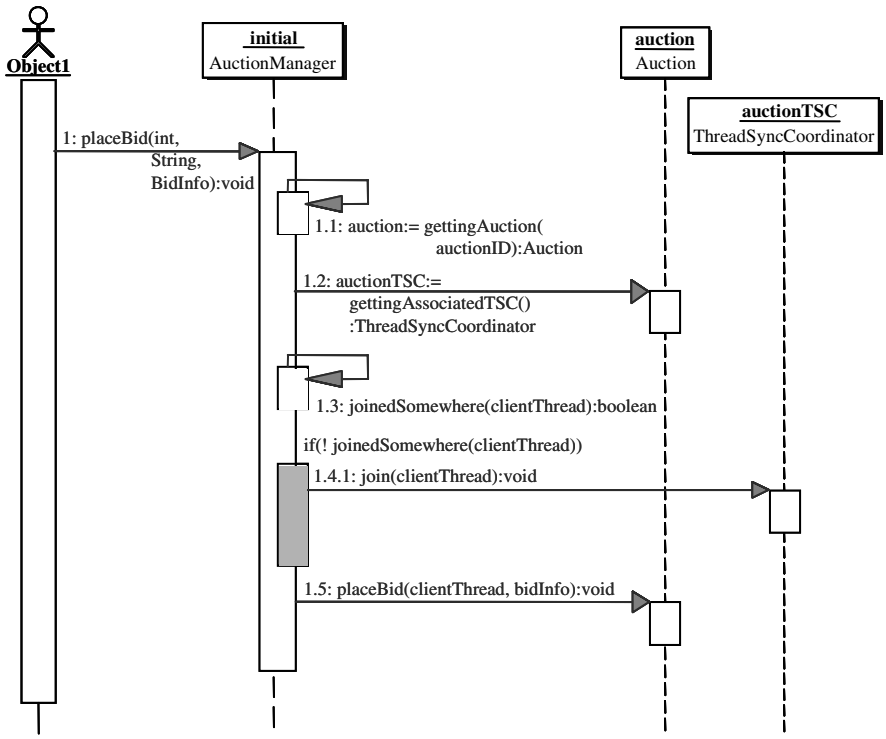

Fig. 12. Placing a bid in an Auction

further referred as `clientThread` in the diagrams. Besides all the checkings that are done at different levels, please notice that the constructor of an `Auction` takes as parameter one `ThreadSyncCoordinator` object, which will handle thread coordination for that particular `Auction`.

The shaded rectangles in Figure 11 and Figure 12 indicate `then` branches of `if` statements. The `then` branch in Figure 11 does not contain anything, which indicates a `return` to the calling object. So, the diagram should be read as: if the client thread is already a member somewhere else, then `return`, because it is not allowed to create another multithreaded transaction.

One should also notice that the client thread is forced by the `AuctionManager` to join the `ThreadSyncCoordinator` associated with the `Auction` it wants to bid in (Fig. 12, operation 1.4.1).

6 Conclusions

Allowing application developers to use multithreading inside a transaction can be dangerous when there is no support for thread coordination inside the transaction. Threads can decide to leave the transaction and perform some other operations before the outcome of the transaction has been determined, or a thread can get associated a new trans-

action context while already acting on behalf of another transaction. This freedom makes it very hard for application developers to guarantee the ACID properties for multithreaded transactions.

In order to overcome this drawback, we considered the Open Multithreaded Transaction model, which, based on a few rules, constrains the participating threads to behave in a disciplined way, so that we can guarantee that the ACID properties are met for multithreaded transactions as well. Further on, the OMTT model was ported to CORBA by implementing a Thread Synchronization Coordinator, *ThreadSyncCoordinator*, that sits between the individual client threads and the CORBA OTS, providing the desired thread control inside a CORBA multithreaded transaction. Thanks to the *ThreadSyncCoordinator*, client threads are now able to explicitly join a transaction, and get access to the shared transactional resources, simulating somehow that the transaction support is aware of concurrency. A blocking commit protocol ensures that once in a transaction, a thread cannot leave before the outcome of the transaction has been determined. Events are used to signal all participant threads in a multithreaded transaction to vote on the outcome of the transaction.

Implementing the Online Auction System has shown how the complexity of a dynamic, distributed, and concurrent application can be reduced by structuring it using enhanced CORBA multithreaded transactions with thread control provided by the *ThreadSyncCoordinator*.

References

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

[2] Object Management Group, Inc.: http://www.omg.org/

[3] Object Management Group, Inc.: *The Common Object Request Broker: Architecture and Specification*, v3.0, July 2002.

[4] Object Management Group, Inc.: *Transaction Service Specification*, v1.3, September 2002.

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

[6] 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.

[7] Kienzle, J.; Jiménez-Peris, R.; Romanovsky, A.; Patiño-Martinez, M.: *Transaction Support for Ada*. Proceedings of the 6th International Conference on Reliable Software Technologies, Ada-Europe, Leuven, Belgium, May 14-18, 2001. LNCS Vol. **2043**, Springer Verlag, 2001, pp. 290 – 304.

[8] Kienzle, J.: *OPTIMA: OPen Transaction Integration for Multithreaded Applications*. http://www.cs.mcgill.ca/~joerg/sel/research/optima.html

[9] Object Management Group, Inc.: *Unified Modeling Language Specification*, v1.5, March 2003.

[10] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] Metsker, S. J.: *Design Patterns: Java™ Workbook*. Addison-Wesley, 2002.

[12] Object Management Group, Inc.: *Event Service Specification*, v1.1, March 2001.

[13] Object Management Group, Inc.: *Notification Service Specification*, v1.0.1, August 2002.

[14] Object Management Group, Inc.: *Concurrency Service Specification*, v1.0, April 2000.

[15] Object Management Group, Inc.: *Naming Service Specification*, v1.2, September 2002.

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

[17] Silaghi, R.; Strohmeier, A.: *Critical Evaluation of the EJB Transaction Model*. Proceedings of the 2nd International Workshop on scientiFic engIneering of Distributed Java applIcations, FIDJI, Luxembourg-Kirchberg, Luxembourg, November 28-29, 2002. LNCS Vol. **2604**, Springer-Verlag, 2003, pp. 15 – 29. An extended version is also available as Technical Report, EPFL-IC-LGL N° IC/2002/069, September 2002.

[18] Siegel, J.: *CORBA 3 Fundamentals and Programming, Second Edition. Includes New CORBA Component Model and Persistence Service*. John Wiley & Sons, 2000.

[19] Tari, Z.; Bukhres, O.: *Fundamentals of Distributed Object Systems: The CORBA Perspective*. John Wiley & Sons, 2001.

[20] Vogel, A.; Rangarao, M.: *Programming with Enterprise JavaBeans™, JTS, and OTS: Building Distributed Transactions with Java™ and C++*. John Wiley & Sons, 1999.

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

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