Checked Transactions in an Asynchronous Message Passing Environment

S.J. Caughey, M.C. Little & S.K. Shrivastava
Department of Computing Science,
University of Newcastle,
Newcastle upon Tyne, NE1 7RU, UK

Abstract

Traditionally transactions have been single-threaded. In such an environment the thread terminating the transaction is, by definition, the thread which performed the work. Therefore, transaction termination is implicitly synchronised with the completion of the transactional work. With the increased availability of both software and hardware multi-threading, transaction services are being required to allow multiple threads to be active within a transaction. In these systems it is important to guarantee that all threads have completed when a transaction is terminated, otherwise some work may not be performed transactionally.

In this paper we present a protocol for the enforcement of checked transactional behaviour within an asynchronous environment. We illustrate the use of the protocol within a proposed implementation for a CORBA-compliant Object Transaction Service intended for a soft real-time application which makes extensive use of concurrency and asynchronous message passing.

Keywords : transactions, CORBA, distributed systems, real-time

1. Introduction

Transactions (atomic actions) are used to guarantee the consistency of work performed within their scope. Transactions are units of work that have the following ACID characteristics:

- *Atomicity*: The transaction completes successfully (commits) or if it fails (aborts) all of its effects are undone (rolled back).
- *Consistency*: Transactions produce consistent results and preserve application specific invariants.
- *Isolation*: Intermediate states produced while a transaction is executing are not visible to others. Furthermore transactions appear to execute serially, even if they are actually executed concurrently.
- *Durability*: The effects of a committed transaction are never lost (except by a catastrophic failure).

Typically a transaction service operates by having objects register their involvement in a transaction with a *transaction manager*, which executes a multi-phase commit protocol when the transaction is committed. Traditionally transactions have been *single-threaded* (where a *thread* is defined to be an entity which performs work, e.g., a lightweight process, or a UNIX process.) In such an environment the thread terminating the transaction is, by definition, the thread which performed the work. Therefore, transaction termination is implicitly synchronised with the completion of the transactional work.

With the increased availability of both software and hardware multi-threading, transaction services are being required to allow multiple threads to be active within a transaction. In such systems it is important to guarantee that all of these threads have completed when a transaction is terminated, otherwise some work may not be performed transactionally. Although protocols exist for enforcing thread and transaction synchronisation in local and distributed environments (commonly referred to as *checked transactions* [1]), they assume that communication between threads is synchronous (e.g., via remote procedure call). A thread making a synchronous call will block until the call returns, signifying that any threads created have terminated. For example, the Object

Transaction Service (OTS), specified as one of the Common Object Request Broker Architecture (CORBA) Services [2] by the Object Management Group, specifies a checked transactions protocol based upon the X/Open model.

However, a range of distributed applications exist which require extensive use of concurrency in order to meet real-time performance requirements, and utilise asynchronous message passing for communication. In such environments it is difficult to guarantee synchronisation between threads, since the application may not communicate the completion of work to a sender, as is done implicitly with synchronous invocations.

In this paper we will describe a proposed implementation of checked transaction behaviour within a soft real-time application which uses message passing for communication. Our implementation places no restrictions on the type of invocation mechanism used within the application and has minimum impact upon programming effort. Although our application is required to use a CORBA compliant Object Request Broker (ORB) [3] for the middleware with an OTS implementation, the ideas presented are generally applicable to other message-oriented middleware and transaction service implementations.

2. A processing model

Before describing our application we will first define a processing model with which to describe our ideas. In this model all processing carried out within a system is performed by threads which execute system function and access system state. Both system state and function are encapsulated within objects, and threads progress by invoking objects i.e., invoking some function within an object. We assume complete distribution transparency within our model, so that invocation upon some object can be perceived as the invoking thread 'moving' to where the object resides, and executing the required operation there.

Invocation may be either synchronous or asynchronous. With synchronous invocation the invoking thread obeys call-return semantics, 'moving' to the object to execute the required function and then returning to the point in its program immediately succeeding the invocation. An asynchronous invocation can be described as the creation of a new thread whose task it is to execute the required function. The creating thread continues independent of the created thread. Upon completion of the invoked function the created thread terminates.

As well as performing object invocations, a thread may explicitly create another thread and specify the function that the created thread is to execute. *Note that in our model the creation of a new thread and an asynchronous invocation are equivalent.* Both result in a new thread which is independent of its creator. Throughout the remainder of this paper we shall refer only to asynchronous invocation in order to imply both.

A thread's 'involvement' i.e. association, with a particular transaction can be modelled as a record on a logical stack known as a transaction context. A thread becomes 'involved' whenever it invokes a transaction 'begin', in which case the new transaction is 'pushed' onto its transaction context. Multiple transactions are present within the transaction context when the thread is within a nested transaction. The transaction at the top of the stack is 'active'. Any object invoked by a thread which is 'active' may choose to register itself with the transaction, thereby ensuring its participation in the commit protocol for that transaction. Any thread which is 'active' within a transaction may execute the transaction 'end' at which time the transaction is 'popped' off the transaction context.

By default any newly created thread takes the same transaction context as its creator. A transaction may therefore have one or more 'involved' threads each of which may be 'active'.

3. Checked behaviour

Applications which do not create new threads and only use synchronous invocations within transactions implicitly exhibit checked behaviour. That is, it is guaranteed that whenever the transaction ends there can be no thread active within the transaction which has not completed its processing. This is illustrated in figure 1, in which (along with figures 2 and 3) vertical lines indicate the execution of function, horizontal lines message exchange, and the boxes represent objects. Figure 1 illustrates a client who starts a transaction by invoking a synchronous 'begin' upon a transaction manager. The client later performs a synchronous invocation upon object *a* which in turn invokes object *b*. Each of these objects registers itself as being involved in the transaction with the manager. Whenever the client invokes the transaction 'end' upon the manager, the manager is then able to enter into the commit protocol (of which only the final phase is shown here) with the registered objects before returning control to the client.

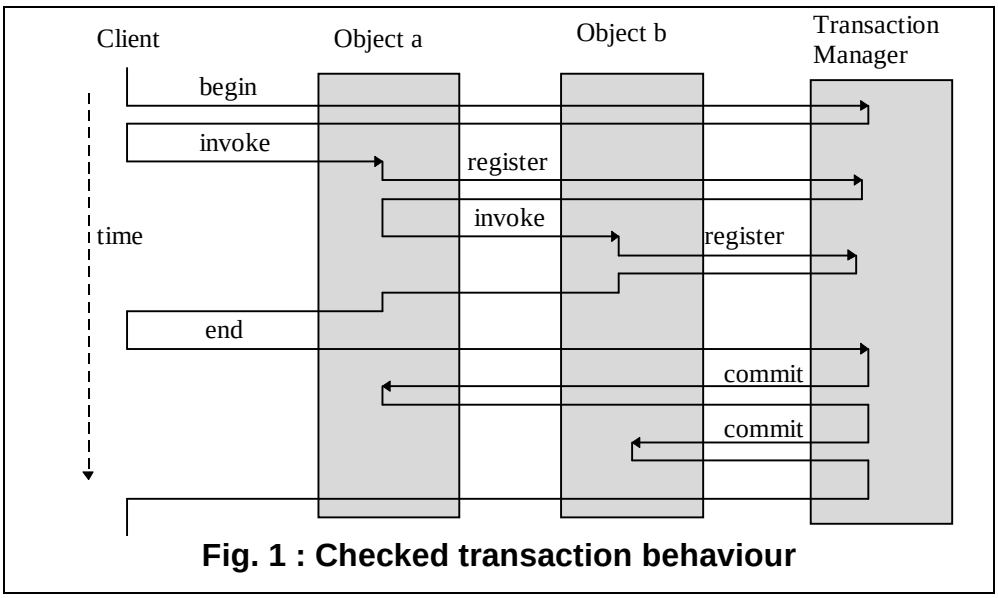

Fig. 1 : Checked transaction behaviour

However, when asynchronous invocation is allowed, explicit synchronisation is required between threads and transactions in order to guarantee checked behaviour. Figure 2 illustrates the possible consequences of using asynchronous invocation without such synchronisation. In this example a client starts a transaction and then invokes an asynchronous operation upon object *a* which registers itself within the transaction as before. *a* then invokes an asynchronous operation upon object *b*. Now, depending upon the order in which the threads are scheduled, it is possible that the client might call for the transaction to terminate. At this point the transaction manager knows only of *a*'s involvement within the transaction so enters into the commit protocol, with *a* committing as a consequence. Then *b* attempts to register itself within the transaction, and is unable to do so. If the application intended the work performed by the invocations upon *a* and *b* to be performed within the same transaction, this may result in application-level inconsistencies. If such failures are to be avoided, checked behaviour must be enforced regardless of the invocation mechanism.

We can avoid unchecked behaviour by requiring that, within a transaction, i) every thread created must synchronise with its creator (thread) immediately before termination; ii) a thread cannot synchronise with its creator until the threads it created have performed this synchronisation; iii) a transaction may not execute its

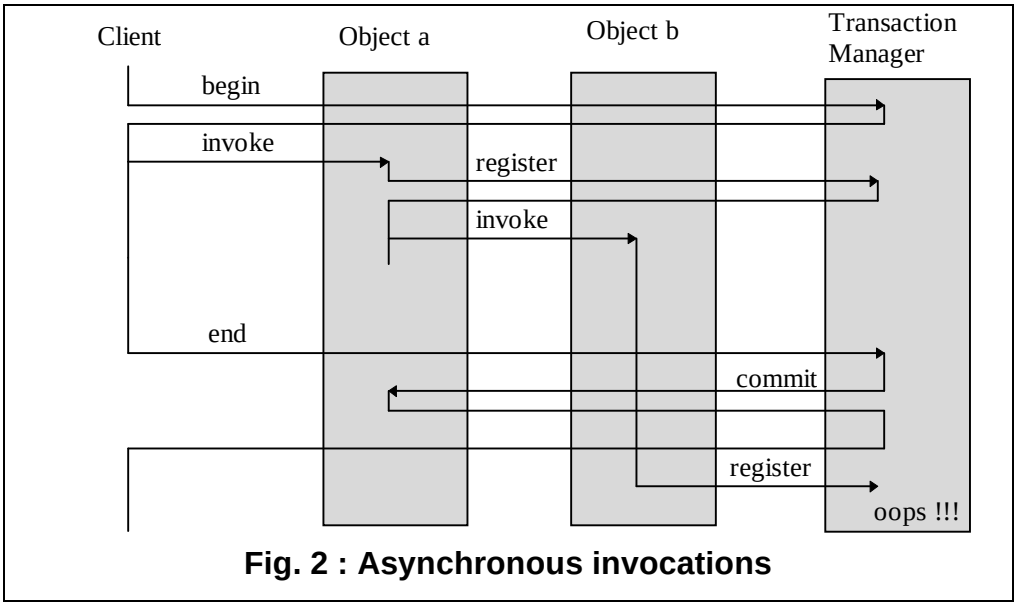

Fig. 2 : Asynchronous invocations

commit protocol until the threads created by the initiating thread i.e. the thread which initiated the transaction, have performed this synchronisation.

The protocol is illustrated in figure 3. Here the threads created by the invocation upon object *a* and *b* are required to inform their creator that they have terminated by sending a 'synch' message. The double parallel lines indicate where threads are awaiting 'synch's for any threads they created. The thread created by the invocation upon object *a* cannot terminate until the thread it created for the invocation upon object *b* returns a 'synch'; and the transaction 'end' may not proceed until the thread created for the invocation upon object *a* returns a 'synch'.

In this example the initiating thread is also the thread which terminates the transaction, and it is therefore implicit that all synchronisation is complete. However, as we described earlier, within our model *any* thread which is 'active' within the transaction may terminate the transaction. In such a case the transaction manager must delay executing the commit protocol until it recognises that the initiating thread has completed its synchronisation.

The effect of introducing this additional synchronisation is to turn asynchronous invocations, occurring within a transaction, into a form of deferred synchronous invocation. This ensures that the tree which represents all the threads involved within the transaction, rooted in the initiating thread, collapses eventually to the root prior to commit. Our solution requires (unavoidable) additional synchronisation messages to be exchanged but does not limit the amount of concurrency obtainable within a transaction and can be implemented so as to be largely transparent to the application programmer.

In the remainder of this paper we shall describe a proposed implementation within a CORBA compliant ORB, which guarantees checked transaction behaviour irrespective of the invocation mechanism used by an application.

4. The application domain

Our target application domain is that of telecommunication. Applications within this domain often have to handle potentially large numbers of concurrent events in a timely fashion. Failure to do so is seen as a failure of the application by the event source and will result in repeated retries and ultimately in user dissatisfaction. This is typical of many telecommunication applications (e.g., call processing). The application endeavours to meet these soft real-time requirements through the extensive use of concurrency. Concurrency is obtained by allowing the application to create threads either explicitly or, more commonly, through the extensive use of asynchronous invocation.

The application uses a CORBA-compliant ORB to support communicating objects distributed within a number of address spaces throughout a loosely coupled system. Asynchronous invocation is implemented as a CORBA 'one-way' call. The CORBA specification for a 'one-way' call requires 'best-effort' semantics where 'best-effort' is defined by the ORB implementation. In

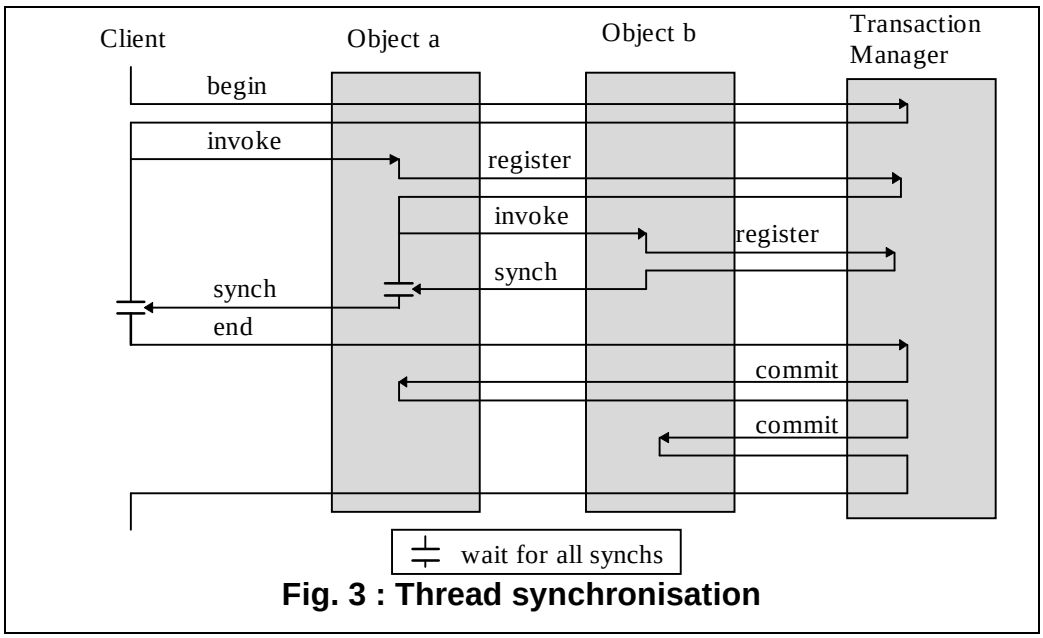

Fig. 3 : Thread synchronisation

the case of our target application, the ORB simply endeavours to deliver a message but does not return an acknowledgement of delivery.

Due to the application's stringent performance requirements, the threads which process asynchronous invocations are not created dynamically but are maintained in a per-address space pool, with a thread being assigned to carry out the processing associated with each invocation. When the processing is complete the thread returns to the pool.

Figure 4 illustrates that the application consists of a number of distributed spaces each of which contains a pool of threads and a number of objects. An underlying ORB allows both synchronous and asynchronous invocations upon objects to be performed in a distribution transparent manner. Some of the spaces have external interfaces over which they may receive (and generate) large numbers of concurrent events.

We are working with the implementors of the ORB to identify areas where the ORB could be tailored to better suit this type of application [4]. We are also in the process of adding the Arjuna OTS-compliant transaction service [5] for use within the application. Transactions are primarily required for certain consistent updates to the distributed databases upon which the application depends. However, as the application is very large (containing many thousands of lines of code) and constantly evolving, it is impossible to predict how and where transactions might be used within the application in the future. It is therefore essential that transactions be capable of being utilised anywhere within the application. Additionally it is a requirement that the addition of transactions should have minimal impact upon the current API so as to avoid extensive changes to existing code and to protect the programmer from unnecessary complication. The service should conform to recognised standards so as to be capable of future inter-operation with standards compliant services e.g., persistence services or other transaction services. All of these requirements can be satisfied by a suitable OTS implementation.

5. Using the CORBA Object Transaction Service

The CORBA OTS specification describes the functionality and interfaces of a service intended to support the use of transactions in applications composed of distributed (CORBA-compliant) objects. The specification describes how objects which wish to participate within a transaction must register themselves with a transaction manager. When a transaction commits the transaction manager engages all participants in a 2-phase commit protocol.

The transaction services may use either *explicit* or *implicit transaction propagation* to transmit knowledge of a transaction's existence to an invoked object. Explicit propagation is the responsibility of the application programmer who must deliver the transaction context as an invocation parameter, whilst implicit propagation is automatically managed by the OTS without programmer intervention. The use of explicit propagation makes it impossible for the OTS to *guarantee* checked behaviour since it relies upon the programmer in order to operate correctly. Fortunately our application utilises implicit propagation only. Explicit propagation will not be considered further in this paper. The implementation is further simplified by the application-specified limitation that only the initiating thread may terminate the transaction and that no nested transactions are allowed. (Our transaction service already supports both of these

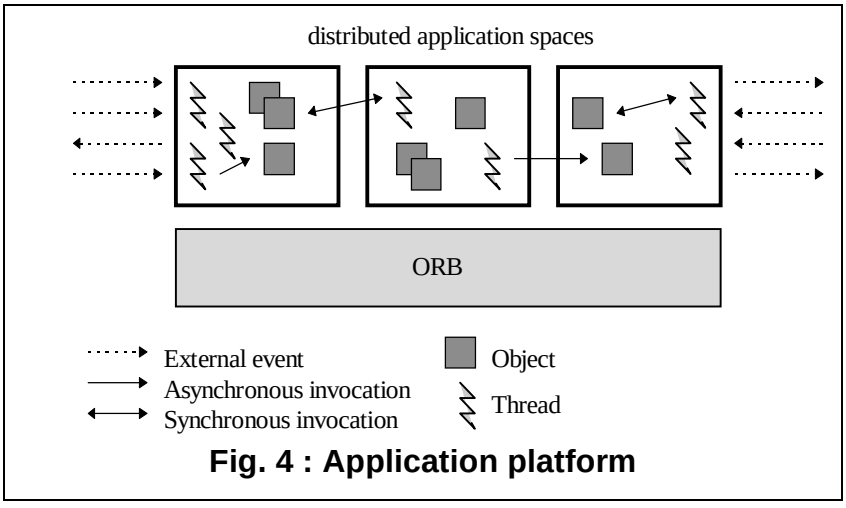

Fig. 4 : Application platform

and the implementation described in the following section could be adapted to operate without these limitations with relatively minor changes).

The OTS specification [1], section 10.4, informs us that "There are many possible implementations of checking in a Transaction Service". However, it only gives details of an implementation which provides equivalent function to that provided by the request/response inter-process models defined by X/Open. The implementation relies upon an application using only synchronous or deferred synchronous invocations within transactions, and upon additional checks imposed by the OTS and the ORB to ensure all that deferred invocations have completed before executing the commit protocol.

Our initial thoughts were to implement this version of checked behaviour. To do so would require replacing all asynchronous invocations within transactions with explicit deferred synchronous invocations. Unfortunately, although most of the activity within our application is non-transactional, it is difficult, if not impossible, to identify which invocations will be transactional. Indeed the same invocation might occur both within and outside of transactions. The difficulty is increased by the fact that, as the application is constantly evolving, invocations which today are only invoked outside of transactions might at some time in the future be invoked within.

The only complete solution therefore would have been to replace *all* asynchronous invocations throughout our application with deferred synchronous. This was deemed unacceptable due to the programming effort involved and the consequent loss in performance caused by generating and awaiting replies to invocations. Instead our proposed implementation dynamically turns asynchronous invocations into deferred synchronous, if and only if, the invocation is occurring within a transaction. This modification to the invocation mechanism occurs at runtime, and is completely transparent to the programmer.

6. Illustrative implementation

6.1 Overview

The aim of our proposed implementation is to allow programmers the continued use and benefits of concurrency within transactions whilst ensuring checked behaviour. As the application obtains concurrency through the creation of threads we must ensure that threads created within transactions synchronise before the end of the transaction in order to indicate that they have completed their processing. As shown in Section 4, this can be achieved by transforming asynchronous invocations ('one-way' calls in our application) into deferred synchronous calls and by ensuring all replies to these calls have been received before proceeding.

In order to convert 'one-way' calls into deferred synchronous, we require the ability to manipulate the invocation protocol stack of the ORB. This can be accomplished by using message interceptors, supported by several commercial ORBs, such as Orbix, or, as in our case, providing alternative implementations of the asynchronous message send and receive primitives. (Because we are working with the ORB implementors to produce an ORB for this type of application, we can make these modifications). This implementation requires no further modification or enhancement to either the transaction system or the ORB. Note, only those 'one-way' calls which are invoked within transactions will be affected.

In the remainder of this section we describe an illustrative implementation to demonstrate the use of our protocol. We shall present our implementation in the form of pseudo-code and give an example of its use in the section following that. Note that our pseudo-code does not include error checking, the time-outs necessary to abort in the face of thread failures, or concurrency control details.

6.2 The Obituary class

Each thread within a transaction owns an obituary object which keeps a count of the number of threads from whom a 'synch' is required. A call of 'Synchronise' will block until all necessary 'synch's have been received.

```
class Obituary
{
  Mutex wait_for_synchs = // initialise as
                          // unlocked
  Int children_count = 0;

  Void Add_Child ()
  { if (children_count++ == 1)
          wait_for_synchs.Lock ();
  }

  Void Remove_Child ()
  { if (children_count-- == 0)
          wait_for_synchs.Unlock ();
  }
```

```
  Void Synchronise ()
  { wait_for_synchs.Lock ();
    // blocks until there are no more
    // synchs to be received
  }
} // end of class Obituary
```

6.3 The Thread class

The Thread class is shown below. Thread objects which are within a transaction hold a 'children' Obituary object, and (unless they are the thread which initiated the transaction) a reference to a 'parent' Obituary object.

```
Class Thread
{
  // unchanged variables and functions
  ........

  static Thread Current ();
    // returns the currently
    // executing thread

  // modified constructor
  Void Thread (// parameters)
  { Register_Parent
          (Current ().Register_Child ());

    // execute the thread as normal
    .........
  }

  Obituary children = null;
  Obituary parent = null;

  Void Transaction_Started ()
  { children = new Obituary ();}

  Void Transaction_Ended ()
  { // we must be in a transaction
    children.Synchronise ();
    children = null;
  }

  Void Register_Parent(Obituary parent_in)
  { if (parent_in != null)
    { // the parent was in a transaction -
      // so we are now in it
      children = new Obituary ();
       parent = parent_in;
    }
  }

  Void Deregister_Parent ()
  { if (children != null)
    { // we are in a transaction
      children.Synchronise ();
      children = null;
      parent.Remove_Child ();
      parent = null;
    }
  }

  Obituary Register_Child ()
  { if (children != null)
      // we are in a transaction
      children.Add_Child ();
    return children;
  }
} // end of class Thread
```

6.4 The Send and Receive primitives

```
Void Send (Object o, Message m)
{ // prepend the obituary object,
  // if there is one
  m.Prepend
    (Thread.Current ().Register_Child ());

  // carry on with normal send
  ……..
}

Void Receive (Message m)
{ // this thread has been assigned from
  // the free pool in order to perform the
  // invocation

  // register the parent's obituary
  // object, if there is one
  Thread.Current ().Register_Parent
                          (m.Remove ());

  // carry on with the invocation as
  // normal
  ……..

  // get children synchs and then synch
  // with parent, if there is one
  Thread.Current ().Deregister_Parent ();

  // the thread now returns to the free
  // pool
}
```

Note that the collective effect of a 'Send' followed by a 'Receive' (in terms of its effects upon threads) is the equivalent to that achieved by one thread explicitly creating a new Thread.

7. Example

We will now demonstrate our solution with a simple example of a transaction which contains asynchronous invocations. Our description will focus on thread management without going into the transactional details. The client's view of the transaction (expressed in pseudo-code) is detailed below and is independent of our implementation of checked behaviour. The functions 'Function1' and 'Function3', which are not described, do not involve any asynchrony internally. (Note that our example is a simplification of the use of the OTS).

```
// application start

1) Transaction t = OTS.begin ();
                    // Transaction begin

ClassX x = // bind to x
x.Function1 (); // synchronous invocation

ClassY y = // bind to y
Message m = // create a message which will
            // invoke 'Function2'
2) Send (y, m); // asynchronous invocation

5) t.commit (); // Transaction end

// application end

class ClassY
{

3)  Function2 ()
  { // do some work
    ……

    ClassZ z = // bind to z
    Message m = // create a message which
                // will invoke 'Function3'
2)  Send (z, m);

    // do some more work
    ……
4)  }

  // rest of the class
  ……..
}
```

We will now describe in more detail the actions which our implementation performs at each to the relevant points numbered in the pseudo-code above.

1. A new transaction is begun and 'Transaction_Started' invoked by it on the current thread.
2. When the message is sent, 'Register_Child' is invoked upon the current thread to record the fact that another child thread is involved, and the identity of the parent is sent with the message.
3. When the message is received, the thread handling the message invokes 'Register_Parent' to record the parent thread.
4. When the thread finishes processing the message then, prior to the completion of the asynchronous 'receive', the thread invokes 'Deregister_Parent' in order to 'synch' this child.
5. The transaction is terminated and 'Transaction_Ended' invoked by it on the initiating thread. This blocks until all synchronisation is complete after which the commit protocol may be executed.

8. Conclusions

We have presented here a protocol for the enforcement of checked transactional behaviour within an asynchronous environment. The protocol has been described within a proposed implementation for a CORBA-compliant Object Transaction Service intended for a soft real-time application which makes extensive use of concurrency and asynchronous message passing. The protocol we have outlined above has the advantages that a) it may be implemented so as to be largely transparent to the programmer, b) it allows the continued unrestricted use of threading, irrespective of whether a function is being executed within a transaction or not, and c) it has very low costs for non-transactional processing. The only costs incurred are the need to send, and wait for, an extra synchronisation message, indicating thread termination, required for every asynchronous invocation performed within a transaction. These costs are only borne within transactions.

9. Acknowledgements

The work reported here has been supported in part by grants from GPT Ltd. and UK Engineering and Physical Sciences Research Council (grant no. GR/L 73708).

Thanks are given to S. Wheater (Newcastle University) and H. Blair (GPT Ltd.) for their comments on the paper.

10. References

1. "Distributed Transaction Processing: The XA Specification, X/Open Document C193", X/Open Company Ltd., Reading, U.K., ISBN 1-85912-057-1.

2. "CORBAservices : Common Object Services Specification", Object Management Group, March 31st, 1995.

3. "Common Object Request Broker Architecture and Specification", Revision 2.0, Object Management Group, July, 1995.

4. H. Blair, S.J. Caughey, H. Green and S.K. Shrivastava, "Structuring Call Control Software Using Distributed Objects", Proc. of TreDS'96, Intl. Workshop on Trends in Distributed Systems, Aachen 1996, LNCS 1161**,** pp. 94-107.

5. G.D. Parrington, S.K. Shrivastava, S.M. Wheater and M.C. Little, "The Design and Implementation of Arjuna," USENIX Computing Systems Journal, Vol 8, No 3, 1995.