

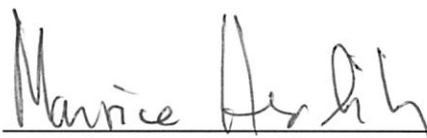
# Distributed Obstruction-Free Transactional Memory

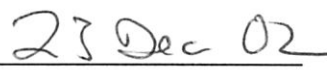
Cheng Zheng

Department of Computer Science

Brown University

Submitted in partial fulfillment of the requirements for the Degree of Master of Science  
in the Department of Computer Science at Brown University

 PROFESSOR  
\_\_\_\_\_  
Signature (Pro. Maurice Herlihy, Project Advisor)

  
\_\_\_\_\_  
Date

# Distributed Obstruction-free Transactional Memory

Cheng Zheng

## Abstract

A distributed transactional memory is a system that supports shared-objects in a distributed computing environment, and keeps atomic updates in the shared data objects by supporting transaction. This paper describes our experience on the implementation of distributed obstruction-free transactional memory as part of *Aleph toolkits*, a distributed shared object system implemented in Java [1]. We use an Optimistic algorithm to handle object access contention between transactions, and achieve "Obstruction-Free" object access in a distributed computing environment.

## 1. Introduction

In distributed computing, there are two computing model, one is *data-shipping* and another is *control-shipping*. *Distributed shared memory* (DSM) systems, whether page-based or object-based, implement a *data-shipping* model, in which the bits representing the object are moved among client caches on demand. By contrast, *remote method invocation* (RMI) systems implement a *control-shipping* model for objects. Calls to an object's methods are transformed to messages forwarded to a remote site that holds the object. A distributed transaction manager is needed in distributed computing to keep data consistence. In *control-shipping* systems, a transaction from start to commit or abort could span multiple distributed systems. The transaction manager needs to coordinate the execution of transaction among these distributed systems; this can be very complex and error prone. In *data-shipping* systems, a transaction from start to commit or abort is confined in a single system while the data accessed could reside at different distributed systems, and *shipped* to the system that is running the transaction. The transaction manager becomes much simpler than the one in "control-shipping" systems; it only needs to control the execution of transaction in a single system.

In this paper, we are going to discuss the implementation of distributed obstruction-free transactional memory in *Aleph toolkits* [1]. The Aleph toolkit is a collection of Java packages intended to support the construction of customized distributed share objects. Aleph supports both *data-shipping* and *control-shipping* systems. We are building a distributed transaction manager as part of *Aleph toolkits*, this transaction manager can manage transactions running on distributed shared memory systems that is based on data-shipping model. This transaction manager uses an optimistic algorithm to handle object access contention. By using this distributed obstruction-free transactional memory, we can use Aleph toolkit to build other distributed transaction applications.

## 2. The Aleph Toolkit

*Aleph toolkit* is a collection of java packages that implements a platform-independent distributed shared object system. A distributed program runs on a number of logical processor, called *Processing Elements* (PEs). Each PE is a Java Virtual Machine, with its own address space. Aleph provides the ability to start threads on remote processors, and to communicate either by shared objects (with transparent synchronization and caching), or by message passing. In this paper, we use shared object to build our distributed transaction manager. Structuring a distributed system as a toolkit allows programmers to "mix and match" different implementation of run-time system components without the need to restructure the application each time. In Aleph, there are several managers to handle different tasks of distributed system: A directory manager is used to locate an object and keep cached copies of data object synchronized; A transaction manger is used to coordinate the execution of transactions that guarantee an atomic change to data object; A event manager is used to monitor events occurrence and disseminate events to interested

*PEs*; A communication manager is used to pass messages between *PEs*. In Aleph, any one of these managers can have different implementation, e.g. there are three directory managers (home, arrow and hybrid) in Aleph toolkit now. We are building an optimistic transaction manager. Programmers can select the managers that are best fitted to their requirements.

In Aleph, *PEs* can share data structures called *global objects*, defined by *GlobalObject* interface. A global object is a container for a regular Java object; we can encapsulate a regular Java object inside a global object to make it accessible to other *PEs*, as showed in the following example:

In this case, *Account* is defined as a class, all objects shared by *PEs* must implement *java.io.Serializable*:

```
public class Account implements Serializable, TMLCloneable{
    Public int value;
```

The *main* method for *Account* class simply encapsulates the *Account* in a *GlobalObject* as following:

```
GlobalObject saving = new GlobalObject( new Account(iniBalance) );
GlobalObject checking = new GlobalObject( new Account(iniBalance) );
```

This object is passed as an argument to the constructor for the user's threads:

```
public UserThread( GlobalObject saving, GlobalObject checking ) {
```

The user thread accesses the *Account* within a transaction by opening the global object in "write" mode:

```
Account save = ( Account ) saving.open( transaction, "w" );
Account check = ( Account ) checking.open( transaction, "w" );
```

As above example, a *PE* uses the *open* method of *GlobalObject* to open an object at required mode, the *open* method in turn submits the global object open request to the directory manager, the directory manager then locates the global object, and ship the object back to the *PE*. We use *home* directory manager for this implementation.

In *home* directory [2], each global object is associated with a fixed *PE*, termed that object's "home". The home keeps track of the number, status and location of all cached copy of that object. There can be only one read/write cached copy, or multiple read-only cached copies. If a client has a cached copy of the object, it keeps track of whether the object is busy (in use by a local thread), or if so, whether the home has requested the copy to be returned or invalidated. If a *PE* wants to acquire an exclusive access to a global object held by another, it does the following:

1. It sends a *RetrieveRequest* message to the object's home.
2. The home checks whether the object has been held by other *PEs*: if yes, it sends a *ReleaseRequest* message to the *PE* holding the cached copy, and *RetrieveRequest.run()* blocks; if not, it sends the object to the *PE* that requested in *RetrieveResponse* message.
3. At the *PE* holding the copy, the *ReleaseRequest.run()* method blocks while the cached copy is in use. When object becomes free, the method invalidates the copy, and returns a *ReleaseResponse* message to the home as confirmation.
4. At the home, the blocked *RetrieveRequest.run()* is notified, and it sends a *RetrieveResponse* message containing the current object copy to the requesting client.

There are two blocks (process waiting) in the *home* directory; we need to make some changes to the default behave of home directory to remove the blocks in our Obstruction-Free Transaction protocol. In next few sections, we will explain how we can remove these blocks.

### 3. Obstruction-Free transaction protocol

The Distributed Transactional Memory system is based on *data-shipping* model. A transaction, from transaction start to transaction commit or abort, is confined in a single process element (*PE*). The transaction manager uses an Optimistic Algorithm to handle object access contention between transactions, and achieve Obstruction-Free object access among transactions that are running in the same or different *PEs*. In this algorithm, if a transaction *A* wants to open an object *O* for reading or writing, it will ask Directory Manager to retrieve the object *O*, if the object *O* is free, the current owner of object *O* will ship the object *O* to the *PE* that is running the transaction *A*. If the object *O* is being opened by another transaction *B*, the transaction *A* will back off the attempt for a randomized period of time before another attempt to open the object *O*. The transaction *A* will keep make this attempt for pre-defined times, if the object *O* is still opened by transaction *B* after so many attempts, the transaction *A* can force transaction *B* to abort, and take over the object *O*.

### 4. Implementation

First, we need a transactional memory thread that is running within one *PE*, and able to invoke a transactional memory thread within another *PE*, and optionally wait for that thread to finish. We can extend Aleph remote thread class to create a transactional memory thread, *TMThread*. In order to support transaction, the *TMThread* class should have methods to start transaction, commit or abort transaction. The objects that are accessed in *TMThread* could reside at other *PEs*, we should use Aleph *GlobalObject* within *TMThread*. *TMThread* uses *GlobalObject* open method to open objects it needs.

```
public class TMThread extends RemoteThread {
    Transaction transaction;
    ...
    public void beginTransaction() {
        this.transaction = new Transaction();
    }
    public boolean commitTransaction() { ... }
    public void abortTransaction() { ... }
    public void run() { ... }
}
```

*UserThread* extends *TMThread*, and uses *GlobalObject* (saving, checking account) as shared data object. It also implements the *run()* method, which is invoked at remote *PE* when *UserThread* starts another *UserThread* at remote *PE*.

```
static class UserThread extends TMThread {
    ...
    GlobalObject saving, checking;
    Transaction transaction;
    UserThread(int count, GlobalObject saving, GlobalObject checking) {
        ...
        this.saving = saving;
        this.checking = checking;
    }

    public void run() {
        ...
        beginTransaction();
        ...
        Account save = (Account) saving.open(transaction, "w");
        save.credit(amount);
    }
}
```

```

...
Account check = (Account)checking.open(transaction, "w");
check.debit(amount);
if ( !commitTransaction() )
    abortTransaction();
}
}

```

The *open()* method in *GlobalObject* calls the *open()* method in *TransactionManager*:

```

public class GlobalObject implements Externalizable {
    private static TransactionManager tManager =
        TransactionManager.getManager();
    ...
    public Object open (Transaction transaction, String mode)
        throws AlephException {
        return tManager.open(this, transaction, mode);
    }
}

```

The *TransactionManager* keeps track of *Transactions* running in its *PE*, *GlobalObjects* accessed by these *Transactions*, and the changes made to the *GlobalObjects*. The *TransactionManager* uses the *DirectoryManager* to locate *GlobalObject*, uses *ContentionManager* to handle *GlobalObject* access contention, and uses *Locator* to keep track of versions of objects during a transaction. The *ExponentialBackoff* Algorithm is used in *ContentionManager* when *GlobalObject* is accessed by other *Transaction*.

```

public class OptiTransactionManager extends TransactionManager {
    /* setup jtm contention manager to use ExponentialBackoff algorithm. */
    private static ExponentialBackoff cManager = new ExponentialBackoff();

    /* map transaction -> vector of global objects */
    private static Hashtable t2object = new Hashtable();

    /* map global object -> Locator */
    private static Hashtable g2locator = new Hashtable();

    static final DirectoryManager directory = DirectoryManager.getManager();
}

```

The *open()* method in *OptiTransactionManager* calls the *open()* method in *DirectoryManager* with a Boolean value *force* that is determined by *ExponentialBackoff* contention manager. If *force* is true, it will ask *DirectoryManager* to take over the *object* by abort the other transaction that is holding the *object*; if *force* is false, the *DirectoryManager* will try to open the *object* and return it if it is not opened by other transaction, otherwise, it will return *null* immediately without waiting for the *object* free. We made the change to the default behave of *DirectoryManager* by adding a Boolean *force* to the *open()* method of *DirectoryManager*, and makes *open()* method "non-blocking".

```

public Object open(GlobalObject object, Transaction t, String mode)
    throws AlephException {

    Object obj = getAlreadyOpen(t, object);
    if (obj != null)
        return obj;
    boolean force = cManager.prepare(object, mode);    //inform Contention
    Manager of first try.
    for (int attempt = 0;; attempt++){
        if (t.isAborted()){ //quit if we're return already aborted.

```

```

        throw new AlephException();
    }
    obj = directory.open(object, mode, force);
    if (obj != null) {
        Locator start = new Locator((TMCloneable)obj, t);
        g2locator.put(object, start);
        return start.getNewObject();
    }

    // uh, oh Contention!
    force = cManager.react(object, attempt, mode);
}
}

```

The *ExponentialBackoff* algorithm will determine the frequency and number of attempts made to call *DirectoryManager* *open()* method. Initially it set *force* to false, if *open()* method returns null value, it will sleep for a period of randomized time before making another attempt to call *open()*. It will make this attempt for *MAX\_RETRIES* times, if *object* is still opened by other *Transaction*, it then set *force* to true, *DirectoryManager* will then abort the other *Transaction* and take over the *object*.

```

public class ExponentialBackoff implements ContentionManager, Constants {
    static final int MAX_RETRIES = 128;
    static final int MIN_BACKOFF_LOG = 4;
    static final int MAX_BACKOFF_LOG = 12;
    int logBackoff = 0;
    ...
    public boolean prepare(GlobalObject object, String mode) {
        this.logBackoff = MIN_BACKOFF_LOG;
        return false;
    }
    ...
    public boolean react(GlobalObject object, int attempt, String mode)
        throws AlephException {

        int modeName = parseMode(mode);
        // If this is an early attempt, back off for a bit
        if (attempt < MAX_RETRIES) {
            // Backoff for a random duration
            int mask = (1 << this.logBackoff) - 1;
            int backoff = random[random_counter++ & RANDOM_MASK] & mask;
            try {
                Thread.currentThread().sleep(backoff);
            } catch (InterruptedException e) {}
            this.logBackoff *= 2;
            if (logBackoff > MAX_BACKOFF_LOG)
                logBackoff = MAX_BACKOFF_LOG;
            // Don't force yet
            return false;
        }

        // Just go straight in if this is for write access
        if (modeName == WRITE_MODE && attempt < (2 * MAX_RETRIES)){
            return true;
        }

        // Livelock!
    }
}

```

```

        throw new AlephException("livelock detected!");
    }

```

A transaction can commit or abort the changes it made to objects. If a transaction commit successfully, the newer version of objects is made permanently, otherwise, the older version of objects is restored. We use *Locator* to keep track of versions of *GlobalObject* as following:

```

public class Locator {
    private Transaction transaction;
    private TMCloneable oldObject;
    private TMCloneable newObject;

    public Locator(TMCloneable object, Transaction transaction) {
        this.transaction = transaction;
        this.newObject = (TMCloneable) object.clone();
        this.oldObject = object;
    }

    public void commit() {
        oldObject.copy(newObject); // commit the change,
                                   // copy newObject into oldObject, and oldObject returned.
    }

    *Abort the changes, doing nothing, oldObject returned.
    public void abort() {}
}

```

We made some changes to “home” *DirectoryManager* to make it “non-blocking”, and able to abort other *Transaction* to accommodate *OptiTransactionManger*, as showed in step 3 of following process. In this new *home* directory, if a *PE* wants to acquire an exclusive access to a global object held by another, it does the following:

1. It sends a *RetrieveRequest* message to the *object*’s home.
2. The home checks whether the *object* has been held by other *PE*s: if yes, it sends a *ReleaseRequest* message to the *PE* holding the cached copy, and *RetrieveRequest.run()* blocks; if not, it sends the *object* to the *PE* that requested in *RetrieveResponse* message.
3. At the *PE* holding the copy, the *DirectoryManager* will pass the *releaseRequest* to *ClientSide*. If Boolean *force* is true, *DirectoryManager* will first abort the *Transaction* that held the *GlobalObject* before calling *ClientSide releaseRequest* method. At *ClientSide*, if *force* is true, *releaseRequest* will wait for the *Transaction abort()* complete that should be quick, or wait for *object* free as original *home DirectoryManager*; if *force* is false and *GlobalObject* is busy, *releaseRequest* will send back *ReleaseResponse* with “null” *object* immediately. If *object* is free, *releaseRequest* will send back *ReleaseResponse* with its copy of *object*.
4. At the home, the blocked *RetrieveRequest.run()* is notified, and it sends a *RetrieveResponse* message containing the current *object* copy or null *object* to the requesting client.

*DirectoryManager*:

```

public void releaseRequest (PE from, GlobalObject key, boolean force) {
    if (force){ //abort other transaction, and take over GlobalObject key
        Transaction transaction = tManager.getTransaction(key);
        if (DEBUG)
            Aleph.debug("abort transaction:" + transaction);
        if (transaction != null)
            transaction.abort();
    }
    getClientSide(key).releaseRequest(from, force);
}

```



```

    }

    ClientSide:
    public synchronized void releaseRequest(PE from, boolean force) {
        ...
        while (force && (writer || readers > 0)) {
            try{ wait(); } catch (InterruptedException e) {};
        }
        requested = false;
        if ( writeValid && !writer && readers == 0) {
            ReleaseResponse release = new ReleaseResponse(key, object);
            release.send(from);
        }
        ...
    } else {
        ReleaseResponse release = new ReleaseResponse(key, null);
        release.send(from);
    }
}

```

The above changes to *home DirectoryManager* make it compatible with both the old non-transactional applications and our new Obstruction-free distributed transactional memory applications.

## 5. Experiments

We have created a simple bank account application as an example of testing Obstruction-free distributed transactional memory. In this bank account application, there are one checking account and one saving account. We debited some random amount of money from saving account, waited for a random period of time between 0 and WAIT (WAIT = 1250, 2500, and 5000 mini-second), then credited it into checking account in a single transaction. We ran this transaction 10 times at every PE, among 2 up to 32 PEs. To prove these transactions are atomic, the total amount of checking and saving account should remain the same after these transactions. We also changed MAX\_RETRIES values in *ExponentialBackoff* algorithm to 64 and 128, which is the maximum number this algorithm should try to open a global object before abort other transaction to take over the global object. We logged the percentage of transitions aborted among all transactions, and elapsed time it took to finish all transactions in every run of experiments.

From the experiment results (Figure 1 & 2), we can see the following trends:

1. The percentage of aborted transactions increases with the number of PEs.
2. The longer the transaction takes, the more transactions will be aborted.
3. The more times a transaction try to open an object, the fewer transactions will be aborted, but it takes longer to complete all transactions.

We can also conclude from these experiments that as long as given enough time for the transaction to complete we can achieve the zero aborted transaction. We believe that an adaptive Max\_Retries to the transaction execution time will dramatically reduce the number of aborted transactions. Intuitively, the longer a transaction take, the longer we should wait for the object to free. For the applications with expected transaction time, we can easily implement the Max\_Retries value corresponding to the transaction time. But for the applications with variant transaction time, we have no choice but to use a fixed Max\_Retries value.

Another potential problem with the pure *ExponentialBackoff* algorithm is that it could increase the number of aborted transactions in the following scenario:

1. Transaction A requests to open object O that is used by transaction B, Transaction A will back off the request for a random period of time. Transaction A makes the retry and back off many times, but before reaching the Max\_Retries.



- Transaction *B* releases the object *O* while transaction *A* is back off, but transaction *C* requests to open object *O*, and get hold of object *O* since *O* is free at this moment.
- Transaction *A* makes another retry and reaches the Max\_Retries count, it will abort transaction *C*. If we hold on the object *O* for transaction *A*, we can avoid aborting transaction *C*.

In order to prevent this problem, we need to use priority when decide which transaction can acquire an object. The more tries a transaction makes to request an object, the higher priority it has to acquire the object. But if a transaction does not make the request for an expected time, we should lower its priority, so other transaction have chance to acquire the object in case of a transaction dies, or network failure.

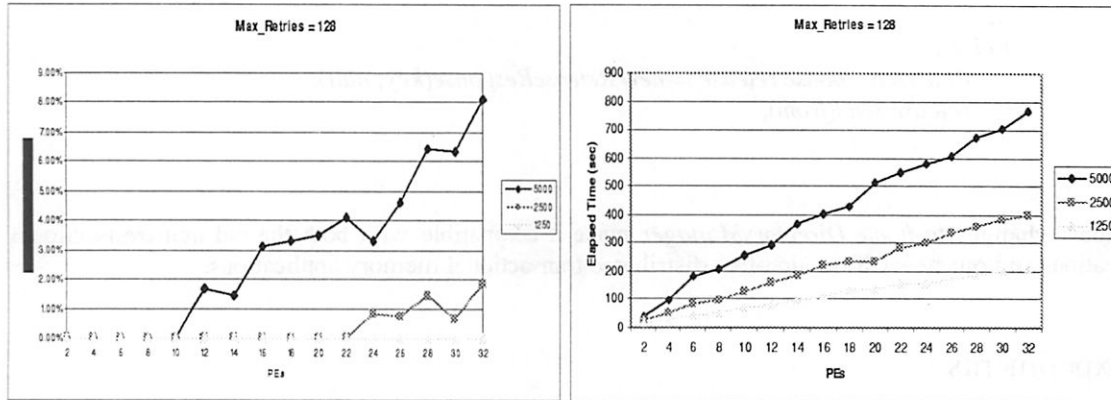


Figure 1. Experiment results with Max\_Retries = 128

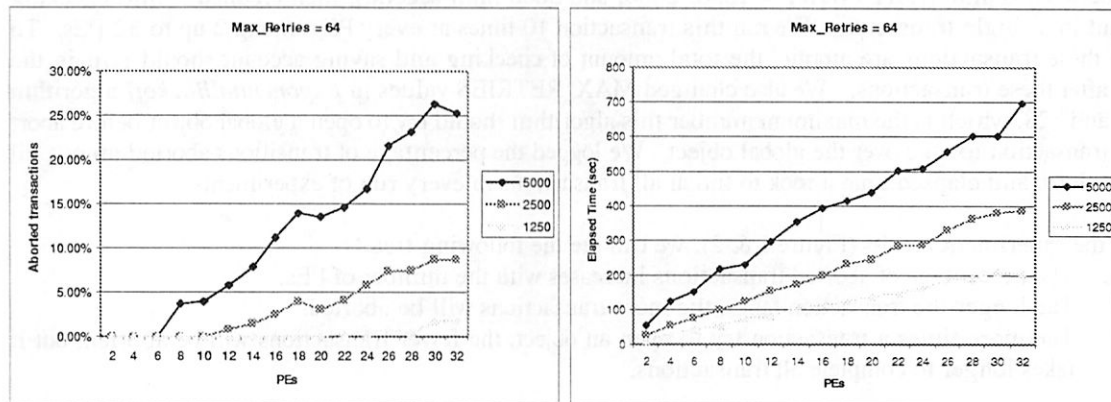


Figure 2. Experiment results with Max\_Retries = 64

Reference:

- [1] M Herlihy. The Aleph Toolkit: Support for Scalable Distributed Shared Objects. *Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing*, January 1999, Orlando, FL.
- [2] M.P. Warres and M.P. Herlihy. A Tale of Two directories: Implementing Distributed Shared Objects in Java. *ACM Java Grande Conference*. June 1999, Palo Alto CA.