CHAPTER 16

# Transactions

**TRANSACTIONS ARE A NECESSARY PART** of many distributed operations. Frequently two or more objects may need to synchronize changes of state so that they all occur, or none occur. This happens in situations such as control of ownership, where one party has to give up ownership at the same time as another asserts ownership. What has to be avoided is only one party performing the action, which could result in the property having either no owners or two owners.

The theory of transactions often includes mention of the "ACID" properties:

- **Atomicity**: All the operations of a transaction must take place, or none of them do.
- **Consistency**: The completion of a transaction must leave the participants in a "consistent" state, whatever that means. For example, the number of owners of a resource must remain at one.
- **Isolation**: The activities of one transaction must not affect any other transactions.
- **Durability**: The results of a transaction must be persistent.

The practice of transactions is that they use the two-phase commit protocol. This requires that participants in a transaction are asked to "vote" on a transaction. If all participants agree to go ahead, then the transaction "commits," which is binding on all the participants. If any "abort" during this voting stage, this forces abortion of the transaction for all participants.

Jini has adopted the syntax of the two-phase commit method. It is up to the clients and services within a transaction to observe the ACID properties if they choose to do so. Jini essentially supplies the mechanism of two-phase commit and leaves the policy to the participants in a transaction.

## Transaction Identifiers

Restricting Jini transactions to a two-phase commit model without associating a particular semantics to it means that a transaction can be represented in a simple way, as a `long` identifier. This identifier is obtained from a transaction manager and

will uniquely label the transaction to that manager. (It is not guaranteed to be unique between managers, though—unlike service IDs.) All participants in the transaction communicate with the transaction manager using this identifier to label which transaction they belong to.

The participants in a transaction may disappear, or the transaction manager may disappear. As a result, transactions are managed by a lease, which will expire unless it is renewed. When a transaction manager is asked for a new transaction, it returns a `TransactionManager.Created` object, which contains the transaction identifier and lease:

```
public interface TransactionManager {
    public static class Created {
        public final long id;
        public final Lease lease;
    }
    ...
}
```

A `Created` object may be passed around between participants in the lease, and one of them will need to look after lease renewals. All the participants will use the transaction identifier in communication with the transaction manager.

## TransactionManager

A transaction manager looks after the two-phase commit protocol for all the participants in a transaction. It is responsible for creating a new transaction with its `create()` method. Any of the participants can force the transaction to abort by calling `abort()`, or they can force it to the two-phase commit stage by calling `commit()`.

```
public interface TransactionManager {

    Created create(long leaseFor) throws ...;
    void join(long id, TransactionParticipant part,
              long crashCount) throws ...;
    void commit(long id) throws ...;
    void abort(long id) throws ...;
    ...
}
```

When a participant joins a transaction, it registers a listener of type `TransactionParticipant`. If any participant calls `commit()`, the transaction manager starts the voting process using all of these listeners. If all of these are prepared to commit, then

the manager moves all of these listeners to the commit stage. Alternatively, any of the participants can call `abort()`, which forces all of the listeners to abort.

## `TransactionParticipant`

When an object becomes a participant listener in a transaction, it allows the transaction manager to call various methods:

```
public interface TransactionParticipant ... {

    int prepare(TransactionManager mgr, long id) throws ...;
    void commit(TransactionManager mgr, long id) throws ...;
    void abort(TransactionManager mgr, long id) throws ...;
    int prepareAndCommit(TransactionManager mgr, long id) throws ...;
}
```

These methods are triggered by calls made upon the transaction manager. For example, if one client calls the transaction manager to abort, then the transaction manager calls all the listeners to abort.

The "normal" mode of operation (that is, when nothing goes wrong with the transaction) is for a call to be made on the transaction manager to commit. It then enters the two-phase commit stage where it asks each participant listener to first `prepare()` and then to either `commit()` or `abort()`.

## `Mahalo`

`Mahalo` is a transaction manager supplied by Sun as part of the Jini distribution. It can be used without any changes. It runs as a Jini service, like `reggie`, and like all Jini services it has two parts: the part that runs as a server, needing its own set of class files in `mahalo.jar`, and the set of class files that need to be available to clients in `mahalo-dl.jar`. It also needs a security policy, an HTTP server, and log files.

`Mahalo` can be started using a command line like this:

```
java  -Djava.security.policy=policy.all \
      -Dcom.sun.jini.mahalo.managerName=TransactionManager \
      -jar /home/jan/tmpdir/jini1_0/lib/mahalo.jar \
      http://'hostname':8080/mahalo-dl.jar \
      /home/jan/projects/jini/doc/policy.all \
      /tmp/mahalo_log public &
```

## A Transaction Example

The classic use of transactions is to handle money transfers between accounts. In this scenario there are two accounts, one of which is debited and the other credited.

This is not a very exciting example, so we shall try a more complex situation. Suppose a service decides to charge for its use. If a client decides this cost is reasonable, it will first credit the service and then request that the service be performed.

The actual accounts will be managed by an Accounts service, which will need to be informed of the credits and debits that occur. A simple Accounts model is one in which the service gets some sort of customer ID from the client, and passes its own ID and the customer ID to the Accounts service, which manages both accounts. This is simple, it is prone to all sorts of e-commerce issues that we will not go into, and it is similar to the way credit cards work!

Figure 16-1 shows the messages in a normal sequence diagram. The client makes a `getCost()` call to the service and receives the `cost` in return. It then makes a `credit()` call on the service, which makes a `creditDebit()` call on the Accounts service before returning. The client then makes a final `requestService()` call on the service and gets back a `result`.

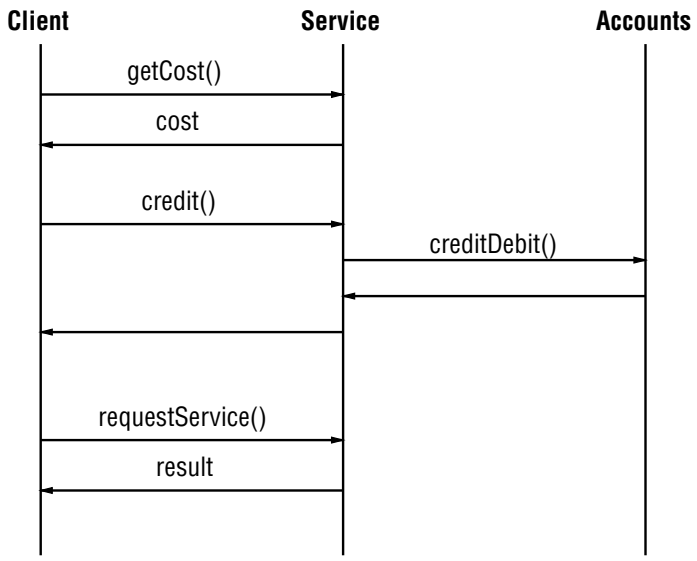


*Figure 16-1. Sequence diagram for credit/debit example*

There are a number of problems with the sequence of steps that can benefit by using a transaction model. The steps of `credit()` and `creditDebit()` should certainly be performed either both together or not at all. But in addition there is the

issue of the quality of the service—suppose the client is not happy with the results from the service and would like to reclaim its money, or better yet, not spend it in the first case! If we include the delivery of the service in the transaction, then there is the opportunity for the client to abort the transaction before it is committed.

Figure 16-2 shows the larger set of messages in the sequence diagram for normal execution. As before, the client requests the cost from the service, and after getting this, it asks the transaction manager to create a transaction and receives back the transaction ID. It then joins the transaction itself. When it asks the service to credit an amount, the service also joins the transaction. The service then asks the account to `creditDebit()` the amount, and as part of this, the account also joins the transaction. The client then requests the service and gets the result. If all is fine, it then asks the transaction manager to `commit()`, which triggers the prepare-and-commit phase. The transaction manager asks each participant to `prepare()`, and if it gets satisfactory replies from each, it then asks each one to `commit()`.

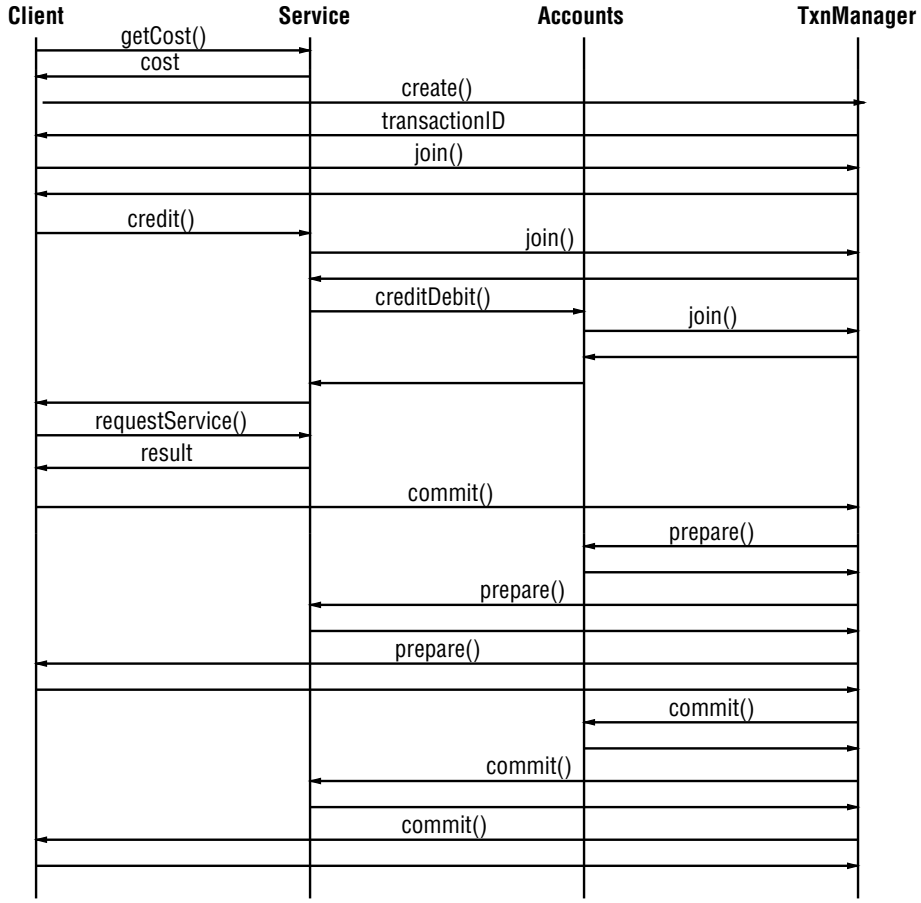


*Figure 16-2. Sequence diagram for credit/debit example with transactions*

There are several points of failure in this transaction:

- The cost may be too high for the client. However, at this stage the client has not created or joined a transaction, so this doesn't matter.
- The client may offer too little by way of payment to the service. The service can signal this by joining the transaction and then aborting it. This will ensure that the client has to rollback the transaction. (Of course, it could instead throw a `NotEnoughPayment` exception—joining and aborting is used for illustrating transaction possibilities.)
- There may be a time delay between finding the price and asking for the service. The price may have gone up in the meantime! The service would then abort the transaction, forcing the client and the accounts to roll back.
- After the service is performed, the client may decide that the result was not good enough, and refuse to pay. Aborting the transaction at this stage would cause the service and accounts to roll back.
- The Accounts service may abort the transaction if sufficient client funds are unavailable.

## *PayableFileClassifierImpl*

The service we will use here is a version of the familiar file classifier that requires a payment before it will divulge the MIME type for a filename. A bit unrealistic, perhaps, but that doesn't matter for our purposes here.

There will be a `PayableFileClassifier` interface, which extends the `FileClassifier` interface. We will also make it extend the `Payable` interface, just in case we want to charge for other services. In line with other interfaces, we shall extend this to a `RemotePayableFileClassifier` and then implement this with a `PayableFileClassifierImpl`.

The `PayableFileClassifierImpl` can use the implementation of the `rmi.FileClassifierImpl`, so we shall make it extend this class. We also want it to be a participant in a transaction, so it must implement the `TransactionParticipant` interface. This leads to the inheritance diagram shown in Figure 16-3, which isn't really as complex as it looks.

The first new element in this hierarchy is the interface `Payable`:

```
package common;

import java.io.Serializable;
```

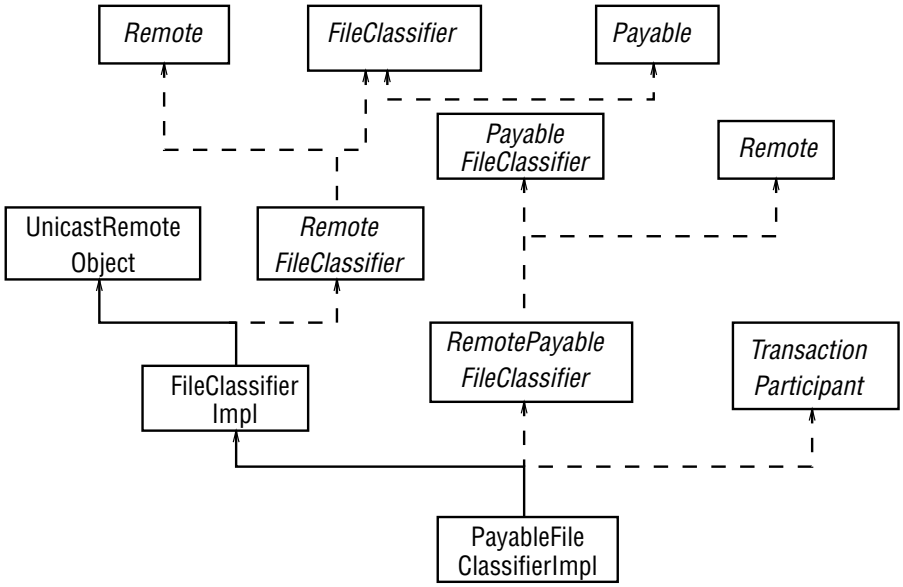


*Figure 16-3. Class diagram for transaction participant*

```
import net.jini.core.transaction.server.TransactionManager;

/**
 * Payable.java
 */

public interface Payable extends Serializable {

    void credit(long amount, long accountID,
                TransactionManager mgr,
                long transactionID)
        throws java.rmi.RemoteException;

    long getCost() throws java.rmi.RemoteException;
} // Payable
```

Extending `Payable` is the `PayableFileClassifier` interface:

```
package common;

/**
 * PayableFileClassifier.java
 */
```

```
public interface PayableFileClassifier extends FileClassifier, Payable {

} // PayableFileClassifier
```

`PayableFileClassifier` will be used by the client to search for the service. The service will use a `RemotePayableFileClassifier`, which is a simple extension to this:

```
package txn;

import common.PayableFileClassifier;
import java.rmi.Remote;

/**
 * RemotePayableFileClassifier.java
 */

public interface RemotePayableFileClassifier extends PayableFileClassifier, Remote
{

} // RemotePayableFileClasssifier
```

The implementation of this service joins the transaction, finds an `Accounts` service from a known location (using unicast lookup), registers the money transfer, and then performs the service. This implementation doesn't keep any state information that can be altered by the transaction. When asked to `prepare()` by the transaction manager it can just return `NOTCHANGED`. If there was state, the `prepare()` and `commit()` methods would have more content. The `prepareAndCommit()` method can be called by a transaction manager as an optimization, and the version given in this example follows the specification given in the "Jini Transaction" chapter of *The Jini Specification* by Ken Arnold et al. The following program gives this service implementation:

```
package txn;

import common.MIMEType;
import common.Accounts;
import rmi.FileClassifierImpl;
//import common.PayableFileClassifier;
//import common.Payable;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.transaction.server.TransactionParticipant;
import net.jini.core.transaction.server.TransactionConstants;
import net.jini.core.transaction.UnknownTransactionException;
```

```
import net.jini.core.transaction.CannotJoinException;
import net.jini.core.transaction.CannotAbortException;
import net.jini.core.transaction.server.CrashCountException;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;

/**
 * PayableFileClassifierImpl.java
 */

public class PayableFileClassifierImpl extends FileClassifierImpl
    implements RemotePayableFileClassifier, TransactionParticipant {

    protected TransactionManager mgr = null;
    protected Accounts accts = null;
    protected long crashCount = 0; // ???
    protected long cost = 10;
    protected final long myID = 54321;

    public PayableFileClassifierImpl() throws java.rmi.RemoteException {
        super();

        System.setSecurityManager(new RMISecurityManager());
    }

    public void credit(long amount, long accountID,
                       TransactionManager mgr,
                       long transactionID) {
        System.out.println("crediting");

        this.mgr = mgr;

        // before findAccounts
        System.out.println("Joining txn");
        try {
            mgr.join(transactionID, this, crashCount);
        } catch(UnknownTransactionException e) {
            e.printStackTrace();
        } catch(CannotJoinException e) {
            e.printStackTrace();
        } catch(CrashCountException e) {
```

```
            e.printStackTrace();
        } catch(RemoteException e) {
            e.printStackTrace();
        }
        System.out.println("Joined txn");


        findAccounts();

        if (accts == null) {
            try {
                mgr.abort(transactionID);
            } catch(UnknownTransactionException e) {
                e.printStackTrace();
            } catch(CannotAbortException e) {
                e.printStackTrace();
            } catch(RemoteException e) {
                e.printStackTrace();
            }
        }

        try {
            accts.creditDebit(amount, accountID, myID,
                              transactionID, mgr);
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
        }


    }

    public long getCost() {
        return cost;
    }

    protected void findAccounts() {
        // find a known account service
        LookupLocator lookup = null;
        ServiceRegistrar registrar = null;

        try {
            lookup = new LookupLocator("jini://localhost");
        } catch(java.net.MalformedURLException e) {
            System.err.println("Lookup failed: " + e.toString());
```

```
        System.exit(1);
    }

    try {
        registrar = lookup.getRegistrar();
    } catch (java.io.IOException e) {
        System.err.println("Registrar search failed: " + e.toString());
        System.exit(1);
    } catch (java.lang.ClassNotFoundException e) {
        System.err.println("Registrar search failed: " + e.toString());
        System.exit(1);
    }
    System.out.println("Registrar found");

    Class[] classes = new Class[] {Accounts.class};
    ServiceTemplate template = new ServiceTemplate(null, classes,
                                                   null);
    try {
        accts = (Accounts) registrar.lookup(template);
    } catch(java.rmi.RemoteException e) {
        System.exit(2);
    }
}

public MIMEType getMIMEType(String fileName) throws RemoteException {

    if (mgr == null) {
        // don't process the request
        return null;
    }

    return super.getMIMEType(fileName);
}


public int prepare(TransactionManager mgr, long id) {
    System.out.println("Preparing...");
    return TransactionConstants.PREPARED;
}

public void commit(TransactionManager mgr, long id) {
    System.out.println("committing");

}
```

```
    public void abort(TransactionManager mgr, long id) {
        System.out.println("aborting");
    }

    public int prepareAndCommit(TransactionManager mgr, long id) {
        int result = prepare(mgr, id);
        if (result == TransactionConstants.PREPARED) {
            commit(mgr, id);
            result = TransactionConstants.COMMITTED;
        }
        return result;
    }


} // PayableFileClassifierImpl
```

## *AccountsImpl*

We shall assume that all accounts in this example are managed by a single `Accounts` service that knows about all accounts by using a `long` identifier. These should be stored in permanent form, and there should be proper crash-recovery mechanisms, etc. For simplicity, we shall just use a hash table of accounts, with uncommitted transactions kept in a "pending" list. When commitment occurs, the pending transaction takes place.

Figure 16-4 shows the `Accounts` class diagram.

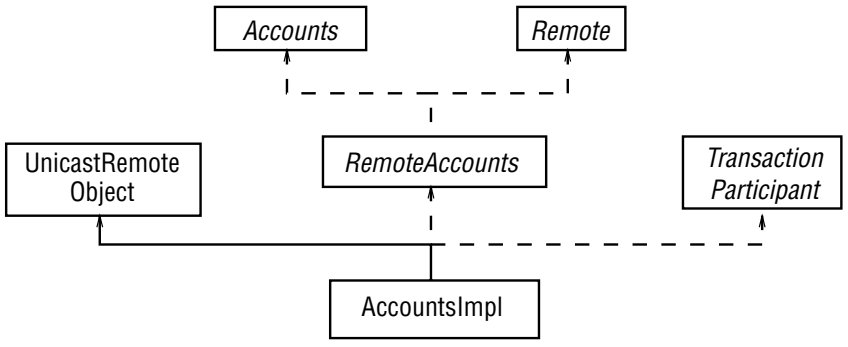


*Figure 16-4. Class diagram for Accounts*

The `Accounts` interface looks like this:

```
/**
 * Accounts.java
 */

package common;

import net.jini.core.transaction.server.TransactionManager;

public interface Accounts  {

    void creditDebit(long amount, long creditorID,
                     long debitorID, long transactionID,
                     TransactionManager tm)
        throws java.rmi.RemoteException;

} // Accounts
```

and this is the implementation:

```
/**
 * AccountsImpl.java
 */

package txn;

// import common.Accounts;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.transaction.server.TransactionParticipant;
import net.jini.core.transaction.server.TransactionConstants;
import java.rmi.server.UnicastRemoteObject;
import java.util.Hashtable;
// import java.rmi.RMISecurityManager;
// debug
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;
// end debug

public class AccountsImpl extends UnicastRemoteObject
    implements RemoteAccounts, TransactionParticipant, java.io.Serializable {

    protected long crashCount = 0; // value??
```

```
    protected Hashtable accountBalances = new Hashtable();
    protected Hashtable pendingCreditDebit = new Hashtable();

    public AccountsImpl() throws java.rmi.RemoteException {
        // System.setSecurityManager(new RMISecurityManager());
    }

    public void creditDebit(long amount, long creditorID,
                            long debitorID, long transactionID,
                            TransactionManager mgr) {

        // Ensure stub class is loaded by getting its class object.
        // It has to be loaded from the same place as this object
        java.rmi.Remote stub = null;
        try {
            stub = toStub(this);
        } catch(Exception e) {
            System.out.println("To stub failed");
            e.printStackTrace();
        }
        System.out.println("To stub found");
        String annote =
java.rmi.server.RMIClassLoader.getClassAnnotation(stub.getClass());
        System.out.println("from " + annote);
        try {
            Class cl = java.rmi.server.RMIClassLoader.loadClass(annote,
"txn.AccountsImpl_Stub");
        } catch(Exception e) {
            System.out.println("To stub class failed");
            e.printStackTrace();
        }
        System.out.println("To stub class ok");

        // mgr = findManager();
        try {
            System.out.println("Trying to join");
            mgr.join(transactionID, this, crashCount);
        } catch(net.jini.core.transaction.UnknownTransactionException e) {
            e.printStackTrace();
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
        } catch(net.jini.core.transaction.server.CrashCountException e) {
            e.printStackTrace();
        } catch(net.jini.core.transaction.CannotJoinException e) {
```

```
            e.printStackTrace();
        }
        System.out.println("joined");
        pendingCreditDebit.put(new TransactionPair(mgr,
                                                   transactionID),
                               new CreditDebit(amount, creditorID,
                                               debitorID));
    }

    // findmanager debug hack
    protected TransactionManager findManager() {
        // find a known account service
        LookupLocator lookup = null;
        ServiceRegistrar registrar = null;
        TransactionManager mgr = null;

        try {
            lookup = new LookupLocator("jini://localhost");
        } catch(java.net.MalformedURLException e) {
            System.err.println("Lookup failed: " + e.toString());
            System.exit(1);
        }

        try {
            registrar = lookup.getRegistrar();
        } catch (java.io.IOException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        } catch (java.lang.ClassNotFoundException e) {
            System.err.println("Registrar search failed: " + e.toString());
            System.exit(1);
        }
        System.out.println("Registrar found");

        Class[] classes = new Class[] {TransactionManager.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
        try {
            mgr = (TransactionManager) registrar.lookup(template);
        } catch(java.rmi.RemoteException e) {
            System.exit(2);
        }
        return mgr;
    }
```

```
    public int prepare(TransactionManager mgr, long id) {
        System.out.println("Preparing...");
        return TransactionConstants.PREPARED;
    }

    public void commit(TransactionManager mgr, long id) {
        System.out.println("committing");

    }


    public void abort(TransactionManager mgr, long id) {
        System.out.println("aborting");
    }

    public int prepareAndCommit(TransactionManager mgr, long id) {
        int result = prepare(mgr, id);
        if (result == TransactionConstants.PREPARED) {
            commit(mgr, id);
            result = TransactionConstants.COMMITTED;
        }
        return result;
    }

    class CreditDebit {
        long amount;
        long creditorID;
        long debitorID;

        CreditDebit(long a, long c, long d) {
            amount = a;
            creditorID = c;
            debitorID = d;
        }
    }

    class TransactionPair {

        TransactionPair(TransactionManager mgr, long id) {

        }
    }
} // AccountsImpl
```

## *Client*

The final component in this application is the client that starts the transaction. The simplest code for this would just use the blocking `lookup()` method of `ClientLookupManager` to find first the service and then the transaction manager. We will use the longer way to show various ways of doing things.

This implementation uses a nested class that extends `Thread`. Because of this, it cannot extend `UnicastRemoteObject` and so is not automatically exported. In order to export itself, it has to call the `UnicastRemoteObject.exportObject()` method. This must be done before the call to join the transaction, which expects a remote object.

```
package client;

import common.PayableFileClassifier;
import common.MIMEType;

import java.rmi.RMISecurityManager;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.transaction.server.TransactionConstants;
import net.jini.core.transaction.server.TransactionParticipant;
// import com.sun.jini.lease.LeaseRenewalManager;
import net.jini.lease.LeaseRenewalManager;
import net.jini.core.lease.Lease;
import net.jini.lookup.entry.Name;
import net.jini.core.entry.Entry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

/**
 * TestTxn.java
 */

public class TestTxn implements DiscoveryListener {

    PayableFileClassifier classifier = null;
    TransactionManager mgr = null;

    long myClientID; // my account id
```

```
public static void main(String argv[]) {
    new TestTxn();

    // stay around long enough to receive replies
    try {
        Thread.currentThread().sleep(100000L);
    } catch(java.lang.InterruptedException e) {
        // do nothing
    }
}

public TestTxn() {
    System.setSecurityManager(new RMISecurityManager());

    LookupDiscovery discover = null;
    try {
        discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
    } catch(Exception e) {
        System.err.println(e.toString());
        System.exit(1);
    }

    discover.addDiscoveryListener(this);

}

public void discovered(DiscoveryEvent evt) {

    ServiceRegistrar[] registrars = evt.getRegistrars();

    for (int n = 0; n < registrars.length; n++) {
        System.out.println("Service found");
        ServiceRegistrar registrar = registrars[n];

        new LookupThread(registrar).start();
    }

        // System.exit(0);
}

public void discarded(DiscoveryEvent evt) {
    // empty
}
```

```
    public class LookupThread extends Thread implements TransactionParticipant,
java.io.Serializable {

        ServiceRegistrar registrar;
        long crashCount = 0; // ???

        LookupThread(ServiceRegistrar registrar) {
            this.registrar = registrar;
        }


        public void run() {
            long cost = 0;

            // try to find a classifier if we haven't already got one
            if (classifier == null) {
                System.out.println("Searching for classifier");
                Class[] classes = new Class[] {PayableFileClassifier.class};
                ServiceTemplate template = new ServiceTemplate(null, classes,
                                                               null);

                try {
                    Object obj = registrar.lookup(template);
                    System.out.println(obj.getClass().toString());
                    Class cls = obj.getClass();
                    Class[] clss = cls.getInterfaces();
                    for (int n = 0; n < clss.length; n++) {
                        System.out.println(clss[n].toString());
                    }
                   classifier = (PayableFileClassifier) registrar.lookup(template);
                } catch(java.rmi.RemoteException e) {
                    e.printStackTrace();
                    System.exit(2);
                }
                if (classifier == null) {
                    System.out.println("Classifier null");
                } else {
                    System.out.println("Getting cost");
                    try {
                        cost = classifier.getCost();
                    } catch(java.rmi.RemoteException e) {
                        e.printStackTrace();
                    }
                    if (cost > 20) {
```

```
                System.out.println("Costs too much: " + cost);
                classifier = null;
            }
        }


    }

    // try to find a transaction manager if we haven't already got one
    if (mgr == null) {
        System.out.println("Searching for txnmgr");

        Class[] classes = new Class[] {TransactionManager.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);


        /*
        Entry[] entries = {new Name("TransactionManager")};
        ServiceTemplate template = new ServiceTemplate(null, null,
                                                       entries);
        */

        try {
            mgr = (TransactionManager) registrar.lookup(template);
        } catch(java.rmi.RemoteException e) {
            e.printStackTrace();
            System.exit(2);
        }
        if (mgr == null) {
            System.out.println("Manager null");
            return;
        }

    }

    if (classifier != null && mgr != null) {
        System.out.println("Found both");
        TransactionManager.Created tcs = null;


        System.out.println("Creating transaction");
        try {
            tcs = mgr.create(Lease.FOREVER);
```

```
                } catch(java.rmi.RemoteException e) {
                    mgr = null;
                    return;
                } catch(net.jini.core.lease.LeaseDeniedException e) {
                    mgr = null;
                    return;
                }

                long transactionID = tcs.id;

                // join in ourselves
                System.out.println("Joining transaction");

                // but first, export ourselves since we don't extend
UnicastRemoteObject
                try {
                    UnicastRemoteObject.exportObject(this);
                } catch(RemoteException e) {
                    e.printStackTrace();
                }

                try {
                    mgr.join(transactionID, this, crashCount);
                } catch(net.jini.core.transaction.UnknownTransactionException e) {
                    e.printStackTrace();
                } catch(java.rmi.RemoteException e) {
                    e.printStackTrace();
                } catch(net.jini.core.transaction.server.CrashCountException e) {
                    e.printStackTrace();
                } catch(net.jini.core.transaction.CannotJoinException e) {
                    e.printStackTrace();
                }

                new LeaseRenewalManager().renewUntil(tcs.lease,
                                                     Lease.FOREVER,
                                                     null);
                System.out.println("crediting...");
                try {
                    classifier.credit(cost, myClientID,
                                      mgr, transactionID);
                } catch(Exception e) {
                    System.err.println(e.toString());
                }
```

```
                    System.out.println("classifying...");
                    MIMEType type = null;
                    try {
                        type = classifier.getMIMEType("file1.txt");
                    } catch(java.rmi.RemoteException e) {
                        System.err.println(e.toString());
                    }

                    // if we get a good result, commit, else abort
                    if (type != null) {
                        System.out.println("Type is " + type.toString());
                        System.out.println("Calling commit");
                        // new CommitThread(mgr, transactionID).run();

                        try {
                            System.out.println("mgr state " +
mgr.getState(transactionID));
                            mgr.commit(transactionID);
                        } catch(Exception e) {
                            e.printStackTrace();
                        }

                    } else {
                        try {
                            mgr.abort(transactionID);
                        } catch(java.rmi.RemoteException e) {
                        } catch(net.jini.core.transaction.CannotAbortException e) {
                       } catch( net.jini.core.transaction.UnknownTransactionException
e) {
                        }
                    }
                }
            }

            public int prepare(TransactionManager mgr, long id) {
                System.out.println("Preparing...");
                return TransactionConstants.PREPARED;
            }

            public void commit(TransactionManager mgr, long id) {
                System.out.println("committing");
            }
```

```
        public void abort(TransactionManager mgr, long id) {
            System.out.println("aborting");

        }

        public int prepareAndCommit(TransactionManager mgr, long id) {
            int result = prepare(mgr, id);
            if (result == TransactionConstants.PREPARED) {
                commit(mgr, id);
                result = TransactionConstants.COMMITTED;
            }
            return result;
        }

    } // LookupThread

    class CommitThread extends Thread {
        TransactionManager mgr;
        long transactionID;

        public CommitThread(TransactionManager m, long id) {
            mgr = m;
            transactionID = id;
            try {
                Thread.sleep(1000);
            } catch(Exception e) {
            }
        }

        public void run() {
            try {
                mgr.abort(transactionID);
            } catch(Exception e) {
                e.printStackTrace();
            }
        }
    } // CommitThread

} // TestTxn
```

## Summary

Transactions are needed to coordinate changes of state across multiple clients and services. The Jini transaction model uses a simple model of transactions, with details of semantics being left to the clients and services. The Jini distribution supplies a transaction manager, called `Mahalo`, that can be used.