CHAPTER 23

Transactions

Transactions are a necessary part of many distributed operations. Frequently, two or more objects will 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 either no owners or two owners.

In this chapter, we'll examine the Jini transaction manager and show how this can be used to give transaction processing in Jini.

Two-Phase Commit Protocol

The theory of transactions often refers to 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, however, is that they use the *two-phase commit protocol*. This requires that participants in a transaction be asked to *vote* on a transaction. If all agree to go ahead, then the transaction *commits*, which is binding on all the participants. If any participants *abort* the transaction during this voting stage, then it forces abortion of the transaction on 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 desire. Jini essentially supplies the mechanism of two-phase commit and leaves the policy of meaning to the participants in a transaction.

Transactions Overview

Restricting Jini transactions to a two-phase commit model without associating particular semantics to this means that a transaction can be represented in a simple way: just as a `long` identifier. This identifier is obtained from a transaction manager, and it 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, and the transaction manager may disappear. So 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, containing 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. One of them will need to look after lease renewals. All the participants will use the transaction identifier in communication with the transaction manager.

Transaction Manager

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 through its `create()` method. Any of the participants may force the transaction to abort with `abort()`, or they can force the transaction 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 may call `abort()`, which forces all of the listeners to abort.

Transaction Participant

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 it calls all the listeners to abort.

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

mahalo Transaction Manager

`mahalo` is a transaction manager supplied by Sun as part of the Jini distribution. It can be used as is, and it runs as a Jini service, like `reggie`. If `LaunchAll` has been used to start `reggie`, then it will also have started `mahalo`.

`mahalo` implements the service `TransactionManager`.

Transactions Example

The classic use of transactions is to handle money transfers between accounts. In this case, there are two accounts, one of which is debited and the other credited. This is not too exciting as an example, so we will try a more complex situation. A service may decide 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 that the service gets, say, a customer ID from the client, and passes its own ID and the customer ID to the accounts service, which manages both accounts. Simple, prone to all sorts of e-commerce issues that I have no intention of going into, and similar to the way credit cards work!

Figure 23-1 shows the messages in a normal sequence diagram. The client makes a call, `getCost()`, to the service, and receives the `cost` in return. It then makes another call, `credit()`, on the service, which makes a call `creditDebit()` on the accounts before returning. The client then makes a final call, `requestService()`, on the service and receives a `result`.

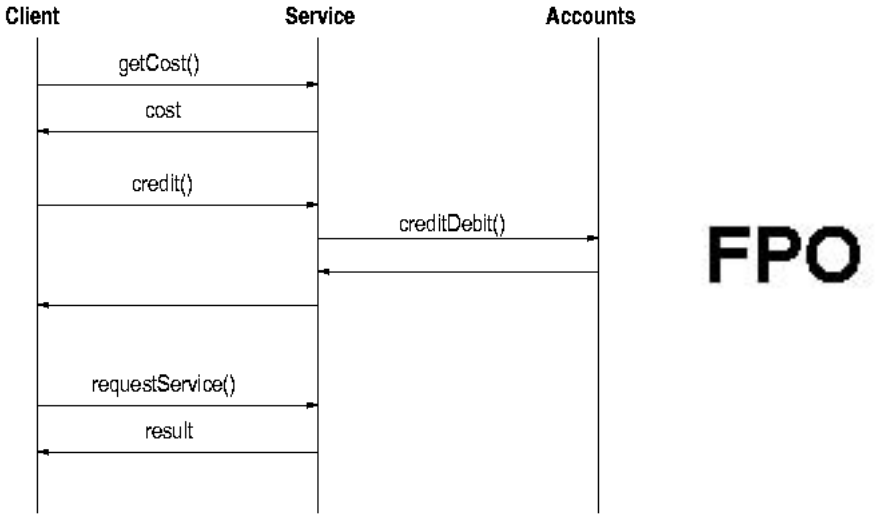
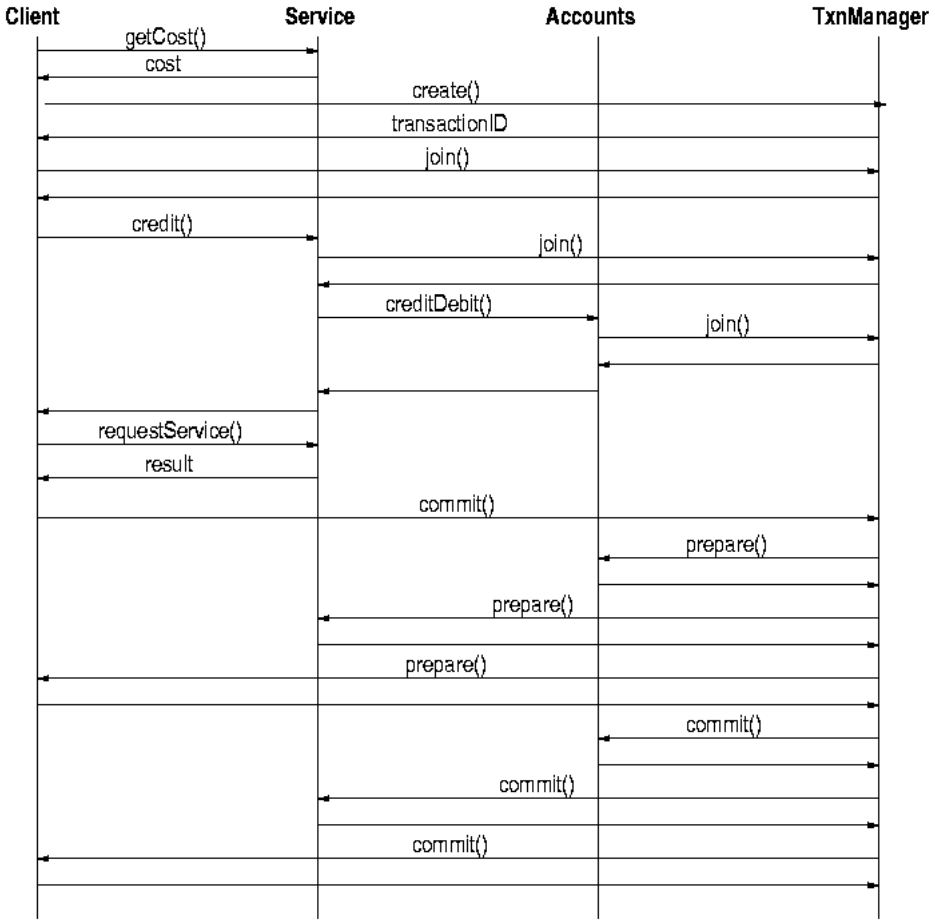


Figure 23-1. *Sequence diagram for credit/debit*

There are a number of problems with this sequence of actions that can benefit from the use of a transaction model. The steps of `credit()` and `creditDebit()` should certainly be performed either together or not at all. But in addition there is the issue of the quality of the service. For example, suppose the client is not happy with the results from the service, and it would like to reclaim its money or, better yet, not spend it in the first place. 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 23-2 shows the larger set of messages in the sequence diagram for "normal" execution. As before, the client requests the cost from the service. After getting this cost, the client asks the transaction manager to create a transaction and receives 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()`.



FPO

Figure 23-2. *Sequence diagram for credit/debit with transactions*

The points of failure in this transaction include the following:

- The cost may be too high for the client. However, at this stage, the client has not created or joined a transaction, so it's not an issue.
- The client may offer too little in the way of payment to the service. The service can signal this by joining the transaction and then aborting it, which ensures that the client has to roll back the transaction. (Of course, the service 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, and 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 is a version of the familiar file classifier that requires a payment before it will divulge the MIME type for a file name. A bit unrealistic, perhaps, but that doesn't matter for our purposes here. There will be an interface, `PayableFileClassifier`, 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'll extend the `PayableFileClassifier` to a `RemotePayableFileClassifier` and then implement it with a `PayableFileClassifierImpl`.

The `PayableFileClassifierImpl` can use the implementation of the `rmi.FileClassifierImpl`, so we will 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 of Figure 23-3, which isn't as complex as it looks.

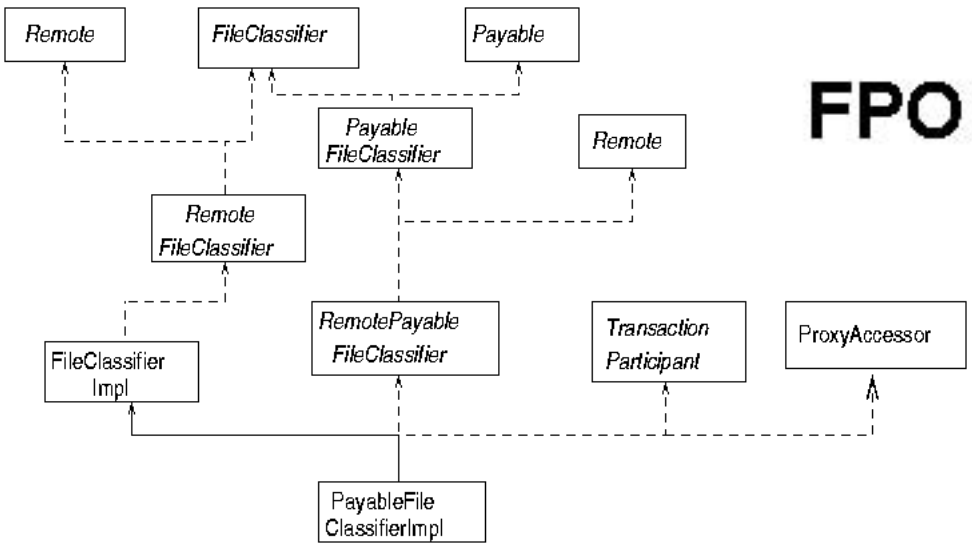


Figure 23-3. *Class diagram for a transaction participant*

A new element in this hierarchy is the interface `Payable`:

```
package common;
import java.io.Serializable;
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 this interface is the `PayableFileClassifier` interface, which will be used by the client to search for the service:

```
package common;
/**
 * PayableFileClassifier.java
 */
public interface PayableFileClassifier extends FileClassifier, Payable {

} // PayableFileClassifier
```

with a simple extension to the remote form:

```
package txn;
import common.PayableFileClassifier;
import java.rmi.Remote;
/**
 * RemotePayableFileClassifier.java
 */
public interface RemotePayableFileClassifier extends PayableFileClassifier, Remote {

} // RemotePayableFileClasssifier
```

The implementation of the `RemotePayableFileClassifier` joins the transaction when `credit()` is called. The implementation object is passed the transaction manager as one parameter to this call. It then finds an `Accounts` service from a known location (e.g., using unicast lookup), registers the money transfer, and then performs the service. There is no real state information kept by this implementation that is 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 may be called by a transaction manager as an optimization, and the version given in this example follows the specification given in the Jini transaction document.

When the implementation object joins the transaction, it must pass an object that the transaction can make calls on. Since the transaction manager is running remotely, this means that the object passed to it must be a *proxy*, which in turn means that the implementation must prepare a proxy and pass it to the transaction manager. On the other hand, the server that contains the service object needs to have a proxy to register the service. You have seen this a few times before: the service implements `ProxyAccessor`, which allows the server to get the proxy from the service.

The service implementation is as follows:

```
package txn;
import common.MIMEType;
import common.Accounts;
import rmi.FileClassifierImpl;
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;
import net.jini.export.ProxyAccessor;
import net.jini.export.*;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.tcp.TcpServerEndpoint;
/**
 * PayableFileClassifierImpl.java
 */
public class PayableFileClassifierImpl extends FileClassifierImpl
    implements RemotePayableFileClassifier, TransactionParticipant, ProxyAccessor {
    protected TransactionManager mgr = null;
    protected Accounts accts = null;
    protected long crashCount = 0; // ???
    protected long cost = 10;
    protected final long myID = 54321;
    protected  TransactionParticipant proxy;
    public PayableFileClassifierImpl() throws java.rmi.RemoteException {
        super();
        System.setSecurityManager(new RMISecurityManager());
        try {
             Exporter exporter = new BasicJeriExporter(TcpServerEndpoint.getIn-
stance(0),
                                                  new BasicILFactory());
             proxy = (TransactionParticipant) exporter.export(this);
        } catch (Exception e) {

        }

    }
```

```
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, proxy, 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;
```

```
    }
    public Object getProxy() {
        return proxy;
    }
} // PayableFileClassifierImpl
```

The server for this implementation uses the variation that it gets the service proxy from the service. Code to do this was illustrated in the server in Chapter 16 and is not repeated here.

AccountsImpl

Let's assume that all accounts in this example are managed by a single `Accounts` service that knows about all accounts via a `long` identifier. These accounts should be stored in a permanent form, there should be proper crash recovery mechanisms, and so on. For simplicity, we will just have a hash table of accounts, with uncommitted transactions kept in a "pending" list. When a commit occurs, the pending transaction takes place.

The `Accounts` service joins the transaction when `creditDebit()` is called. It is passed the transaction manager as a parameter in this call. Figure 23-4 shows the `AccountsImpl` class diagram.

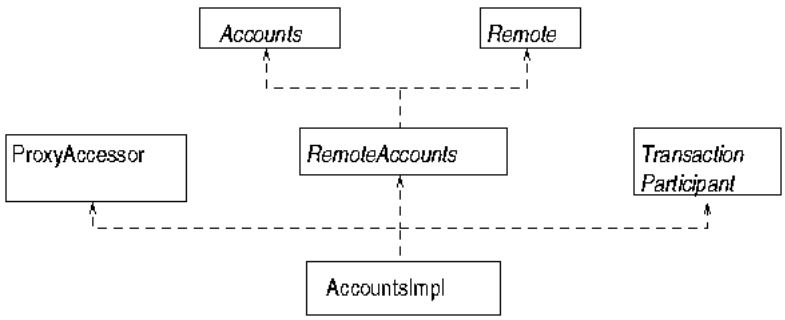


Figure 23-4. *Class diagram for Accounts*

The `Accounts` interface is

```
/**
 * 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 the implementation is

```
/**
 * 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.util.Hashtable;
import net.jini.export.ProxyAccessor;
import net.jini.export.*;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.tcp.TcpServerEndpoint;
// debug
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;
// end debug
public class AccountsImpl
    implements RemoteAccounts, TransactionParticipant, ProxyAccessor {

    protected long crashCount = 0; // value??
    protected Hashtable accountBalances = new Hashtable();
    protected Hashtable pendingCreditDebit = new Hashtable();
    protected  TransactionParticipant proxy;
    public AccountsImpl() throws java.rmi.RemoteException {
        try {
             Exporter exporter = new BasicJeriExporter(TcpServerEndpoint.getIn-
stance(0),
                                                       new BasicILFactory());
             proxy = (TransactionParticipant) exporter.export(this);
        } catch (Exception e) {

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

        try {
            System.out.println("Trying to join");
            mgr.join(transactionID, proxy, 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));
    }
    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) {
        }
    }
    public Object getProxy() {
```

```
        return proxy;
    }
} // AccountsImpl
```

The server for this implementation is standard and so its code is omitted.

Client

The final component in this example is the client that starts the transaction. The simplest code for the client would just use the blocking `lookup()` method of `ClientLookupManager` to find first the service and then the transaction manager. We use the longer way to show various other 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 `UnicastRemoteObject.exportObject`. 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 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;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.discovery.LookupDiscoveryManager;
import net.jini.lookup.ServiceDiscoveryManager;
import net.jini.core.lookup.ServiceItem;
import net.jini.lease.LeaseRenewalManager;
import net.jini.export.*;
import net.jini.jeri.BasicJeriExporter;
import net.jini.jeri.BasicILFactory;
import net.jini.jeri.tcp.TcpServerEndpoint;
import java.rmi.server.ExportException;
/**
 * TestTxn.java
 */
public class TestTxn implements TransactionParticipant {
```

```
private static final long WAITFOR = 100000L;
long crashCount = 0;
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());
    classifier = findClassifier();
    long cost = 0;
    try {
        cost = classifier.getCost();
    } catch(java.rmi.RemoteException e) {
        e.printStackTrace();
    }
    if (cost > 20) {
        System.out.println("Costs too much: " + cost);
        classifier = null;
    }

    mgr = findTxnMgr();
    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");
    // we need to give a proxy to the transaction mgr
```

```
Exporter exporter = new BasicJeriExporter(TcpServerEndpoint.getInstance(0),
                                          new BasicILFactory());
// export an object of this class
TransactionParticipant proxy = null;
try {
    proxy = (TransactionParticipant) exporter.export(this);
} catch (ExportException e) {
    e.printStackTrace();
    System.exit(1);
}
try {
    mgr.join(transactionID, proxy, 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");

    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 PayableFileClassifier findClassifier() {
    ServiceDiscoveryManager clientMgr = null;
    try {
        LookupDiscoveryManager mgr =
            new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                       null, // unicast locators
                                       null); // DiscoveryListener
        clientMgr = new ServiceDiscoveryManager(mgr,
                                            new LeaseRenewalManager());
    } catch(Exception e) {
        e.printStackTrace();
        System.exit(1);
    }

    Class [] classes = new Class[] {PayableFileClassifier.class};
    ServiceTemplate template = new ServiceTemplate(null, classes,
                                                   null);
    ServiceItem item = null;
    // Try to find the service, blocking until timeout if necessary
    try {
        item = clientMgr.lookup(template,
                                null, // no filter
                                WAITFOR); // timeout
    } catch(Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
    if (item == null) {
        // couldn't find a service in time
        System.out.println("no service");
        System.exit(1);
```

```
        }
        // Get the service
        PayableFileClassifier classifier = (PayableFileClassifier) item.service;
        if (classifier == null) {
            System.out.println("Classifier null");
            System.exit(1);
        }
        return classifier;
    }
    public TransactionManager findTxnMgr() {
        ServiceDiscoveryManager clientMgr = null;
        try {
            LookupDiscoveryManager mgr =
                new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS,
                                           null, // unicast locators
                                           null); // DiscoveryListener
            clientMgr = new ServiceDiscoveryManager(mgr,
                                                new LeaseRenewalManager());
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }

        Class [] classes = new Class[] {TransactionManager.class};
        ServiceTemplate template = new ServiceTemplate(null, classes,
                                                       null);
        ServiceItem item = null;
        // Try to find the service, blocking until timeout if necessary
        try {
            item = clientMgr.lookup(template,
                                    null, // no filter
                                    WAITFOR); // timeout
        } catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
        if (item == null) {
            // couldn't find a service in time
            System.out.println("no service");
            System.exit(1);
        }
        // Get the service
        TransactionManager mgr = (TransactionManager) item.service;
        if (mgr == null) {
            System.out.println("Mgr null");
            System.exit(1);
        }
```

```
        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;
    }
} // 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 semantics details left to the clients and services. The Jini distribution supplies a transaction manager that can be used to assist the process.