> **Warning**
>
> Outbound bridging is not currently implemented

5.7.2. Web Services Transactions

WS includes two transaction models referred to as WSAT and WSBA. WSAT integration with JTA is documented in the Transaction Bridging section of the product guide (http://narayana.io/docs/product/index.html#txbridge). By using this bridge in conjunction with the RESTAT JTA bridge full interoperability between RESTAT and WSAT can be realised.

> **Note**
>
> RESTAT outbound bridging is not currently supported so interoperability is one way only.

Chapter 6. STM

In this chapter we shall look at the Software Transactional Memory (STM) implementation that ships as part of Narayana. We won't go into the theoretical details behind STM as they would take up an entire book and there are sufficint resources available for the interested reader to find out themselves. But suffice it to say that STM offers an approach to developing transactional applications in a highly concurrent environment with some of the same characteristics of ACID transactions. Typically though, the Durability property is relaxed (removed) within STM implementations.

The Narayana STM implementation builds on the Transactional Objects for Java (TXOJ) framework which has offered building blocks for the construction of transactional objects via inheritence. The interested reader should look at the text on TXOJ within the ArjunaCore documentation for more in depth details. However, within TXOJ an application class can inherit from the LockManager class to obtain persistence (D) and concurrency (I), whilst at the same time having the flexibility to change some of these capabilities. For example, an object could be volatile, i.e., no durability, and yet still maintain the other transactional properties.

If you look at the abilities that TXOJ offers to developers then it shares many aspects with STM. However, the downside is that developers need to modify their classes through class inheritence (something which is not always possible), add suitable extension methods (for saving and restoring state), set locks etc. None of this is entirely unreasonable, but it represents a barrier to some and hence is one of the reasons we decided to provide a separate STM implementation.

6.1. An STM Example

In order to illustrate the Narayana STM implementation we shall use a worked example throughout the rest of this chapter. We'll make it simple to start with, just a atomic integer that supports set, get and increment methods:

```
public interface Atomic
{
   public void incr (int value) throws Exception;
   public void set (int value) throws Exception;
   public int get () throws Exception;
}
```

We'll throw exceptions from each method just in case, but obviously you could just as easily catch any problems which occur and return booleans or some other indicator from the increment and set methods.

In this example we'll next create an implementation class:

```
public class ExampleInteger implements Atomic
{
   public int get () throws Exception
   {
       return state;
   }

   public void set (int value) throws Exception
   {
       state = value;
   }

   public void incr (int value) throws Exception
   {
       state += value;
   }

   private int state;
}
```

The implementation is pretty straightforward and we won't go into it here. However, so far apart from inheriting from our Atomic interface there's nothing to call this implementation out as being atomic. That's because we haven't actually done anything STM related to the code yet.

Now let's start to modify it by adding in STM specific elements.

> **Note**
>
> All class scope annotations should be applied to the interface whereas method scope annotations should be applied to the implementation class.

Let's start by looking at the Atomic interface. First of all any transactional objects must be instrumented as such for the underyling STM implementation to be able to differentiate them from non-transactional objects. To do that you use the Transactional annotation on the class. Next we need to ensure that our transactional object(s) is free from conflicts when used in a concurrent environment, so we have to add information about the type of operation, i.e., whether or not the method modifies the state of the object. You do this using either the ReadLock or WriteLock annotations.

> **Note**
>
> If you do not add locking annotations to the methods on your Transactional interface then Narayana will default to assuming they all potentially modify the object's state.

At this stage we end up with a modified interface:

```
@Transactional
public interface Atomic
{
   public void incr (int value) throws Exception;
   public void set (int value) throws Exception;
   public int get () throws Exception;
}
```

And class:

```
public class ExampleInteger implements Atomic
{
   @ReadLock
   public int get () throws Exception
   {
       return state;
   }

   @WriteLock
   public void set (int value) throws Exception
   {
       state = value;
   }
```

```
    @WriteLock
    public void incr (int value) throws Exception
    {
        state += value;
    }

    private int state;
}
```

As you can see, these are fairly straightfoward (and hopefully intuitive) changes to make. Everything else is defaulted, though will we will discuss other annotations later once we go beyond the basic example.

> **Note**
>
> We are contemplating allowing method annotations to be applied on the interface and then overridden on the implementation class. For now if you follow the above conventions you will continue to be compatible if this change is eventually supported. https://issues.jboss.org/browse/JBTM-2172

Now we have a transactional class, by virtue of its dependency on the Atomic interface, how we we go about creating instances of the corresponding transactional object and use it (them) within transactions?

```
Container<Atomic> theContainer = new Container<Atomic>();    Example
Atomic obj = theContainer.create(basic);
AtomicAction a = new AtomicAction();

a.begin();

obj.set(1234);

a.commit();

if (obj.get() == 1234)
   System.out.println("State changed ok!");
else
   System.out.println("State not changed!");
a = new AtomicAction();

a.begin();

obj.change(1);

a.abort();

if (obj.get() == 1234)
   System.out.println("State reverted to 1234!");
else
   System.out.println("State is wrong!");
```

For clarity we've removed some of the error checking code in the above example, but let's walk through exactly what is going on.

> **Note**
>
> Some of the discussions around AtomicAction etc. are deliberately brief here because you can find more information in the relevant ArjunaCore documentation.

First we need to create an STM Container: this is the entity which represents the transactional memory within which each object will be maintained. We need to tell each Container about the type of objects for which it will be responsible. Then we create an instance of our ExampleInteger. However, we can't use it directly because at this stage its operations aren't being monitored by the Container. Therefore, we pass the instance to the Container and obtain a reference to an Atomic object through which we can operate on the STM object.

At this point if we called the operations such as incr on the Atomic instance we wouldn't see any difference in behaviour: there are no transactions in flight to help provide the

necessary properties. Let's change that by creating an AtomicAction (transaction) and starting it. Now when we operate on the STM object all of the operations, such as set, will be performed within the scope of that transaction because it is associated with the thread of control. At this point if we commit the transaction object the state changes will be made permanent (well not quite, but that's a different story and one you can see when we discuss the Container in more detail later.)

The rest of the example code simply repeats the above, except this time instead of committing the transaction we roll it back. What happens in this case is that any state changes which were performed within the scope of the transaction are automatically undone and we get back the state of the object(s) as it existed prior to the operations being performed.

Pretty simple and not too much additional work on the part of the developer. Most of the ways in which you will use the Narayana STM implementation come down to similar approaches to what we've seen in the example. Where things may differ are in the various advanced options available to the developer. We'll discuss those next as we look at all of the user classes and annotations that are available.

> **Note**
>
> All of the classes, interfaces and annotations that you should be using can be located within the org.jboss.stm and org.jboss.stm.annotations packages. All other classes etc. located within org.jboss.stm.internal are private implementation specific aspects of the framework and subject to change without warning.

6.2. Annotations

The following annotations are available for use on STM interfaces or classes.

@Transactional: Used on the interface. Defines that implementations of the interface are to be managed within a transactional container. Unless specified using other annotations, all public methods will be assumed to modify the state of the object, i.e., require write locks. All state variables will be saved and restored unless marked explicitly using the @State annotation or SaveState/RestoreState. This assumes currently that all state modification and locking occurs through public methods, which means that even if there are private, protected or package scope methods that would change the state, they will not be tracked. Therefore, the implementation class should not modify state unless by calling its own public methods. All methods should either be invoked within a transactional context or have the Nested annotation applied, wherein the system will automatically create a new transaction when the method is invoked.

@Optimistic: Used on the interface. Specifies that the framework should use optimistic concurrency control for managing interactions on the instances. This may mean that a transaction is forced to abort at the end due to conflicting updates made by other users. The default is @Pessimistic.

@Pessimistic. Used on the interface. Specifies that pessimistic concurrency control should be used. This means that a read or write operation may block or be rejected if another user is manipulating the same object in a conflicting manner. If no other annotation appears to override this, then pessimistic is the default for a transactional object.

@Nested: Used on the interface or class. Defines that the container will create a new transaction for each method invocation, regardless of whether there is already a transaction associated with the caller. These transactions will then either be top-level transactions or nested automatically depending upon the context within which they are created.

@NestedTopLevel: Used on the interface or class. Defines that the container will create a new transaction for each method invocation, regardless of whether there is already a transaction associated with the caller. These transactions will always be top-level transactions even if there is a transaction already associated with the invoking thread.

@ReadLock: Used on the class method. The framework will grab a read lock when the method is invoked.

@WriteLock: Used on the class method. The framework will grab a write lock then the method is invoked.

@LockFree: Used on the class method. No locks will be obtained on this method, though any transaction context will still be on the thread when the method is invoked.

@TransactionFree: Used on the class method. This means that the method is not transactional, so no context will exist on the thread or locks acquired/released when the method is invoked.

@Timeout: Used on the class method. If pessimistic concurrency control is being used then a conflict will immediately cause the operation to fail and the application can do something else. If instead the developer wants the system to retry getting the lock before returning, then this annotation defines the time between each retry attempt in milliseconds.

@Retry: Used on the class method. If pessimistic concurrency control is being used then a conflict will immediately cause the operation to fail and the application can do something else. If instead the developer wants the system to retry getting the lock before returning, then this annotation defines the number of retry attempts.

@State: Used on the class member variables to define which state will be saved and restored by the transaction system. By default, all member variables (non-static, non-volatile) will be saved.

@NotState: Used on the class member variables to define which state to ignore when saving/restoring instance data. Note that any member variable that is not annotated with NotState will be saved and restored by the transaction system, irrespective of whether or not it has the State annotation. You should use these annotations cautiously because if you limit the state which is saved (and hence restored) you may allow dirty data to cross transaction boundaries.

@SaveState: Used on the class method to define the specific save_state method for the class. This is used in preference to any @State indications on the class state. This is the case no matter where in the class hierarchy it occurs. So if you have a base class that uses save/restore methods the inherited classes must have them too if their state is to be durable. In future we may save/restore specifically for each class in the inheritance hierarchy.

@RestoreState: Used on the class method to define the specific restore_state method for the class. This is used in preference to any @State indications on the class state.

6.3. Containers, Volatility and Durability

By default objects created within STM do not possess the Durable aspect of traditional ACID transactions, i.e., they are volatile instances. This has an obvious performance benefit since there is no disk or replicated in-memory data store involved. However, it has disadvantages. If the objects are Pessimitic or Optimistic then they can be shared between threads in the same address space (JVM instance). At the time of writing Optimistic objects cannot be shared between address spaces.

Most of the time you will want to create volatile STM objects, with the option of using optimistic of pessimistic concurrency control really down to the type of application you are developing. As such you use of Containers will be very similar to that which we have seen already:

```
TestContainer<Sample> theContainer = new TestContainer<Sample>();
SampleLockable tester = new SampleLockable();
Sample proxy = theContainer.enlist(tester);
```

However, the Container class has a number of extensibility options available for the more advanced user and requirements, which we shall discuss in the rest of this section.

By default when you create a Container it is used to manage volatile objects. In STM language we call these objects recoverable due to the fact their state can be recovered in the event of a transaction rolling back, but not if there is a crash. The Container therefore supports two types:

```
public enum TYPE { RECOVERABLE, PERSISTENT };
```

You can therefore use the TYPE constructore to create a Container of either type. You can always determine the type of a Container later by calling the type() method.

All Containers can be named with a String. We recommend uniquely naming your Container instances and in fact if you do not give your Container a name when it is created using the default constructure then the system will assign a unique name (an instance of a Narayana Uid). If you want to give you Container a name then you can use the constructor that takes a String and you can get the name of any Container instance by calling the name() method. The default type of a Container is RECOVERABLE.

The Container also supports two sharing models for objects created:

```
public enum MODEL { SHARED, EXCLUSIVE };
```

SHARED means the instance may be used within multiple processes. It must be PERSISTENT too; if not then the framework. EXCLUSIVE means that the instance will only be used within a single JVM, though it can be PERSISTENT or RECOVERABLE. You can get the model used by your container by calling the model() method. The default model for a Container is EXCLUSIVE.

Given the above information, you should now be able to understand what the various constructors of the Container class do, since they provide the ability to modify the behaviour of any created instance through combinations of the above three parameters. Where a given parameter is not available in a specific constructor, the default value discussed previously is used.

6.4. Sharing STM Objects

Once a Container is created, you can use the create() method to create objects (handles) within the STM. As shown in the previous example, you pass in an unmodified (with the possible exception of annotations) class instance which corresponds to the interface type given to the Container when it was created and the Container will return a reference to an instance of the same type:

```
Sample1 obj1 = theContainer.create(new Sample1Imple(10));
```

All objects thus created are uniquely identified by the system. You can obtain their identifier (an instance of the Uid class) at any time by calling the getIdentifier method of the corresponding Container:

```
Uid id = theContainer.getIdentifier(obj1)
```

This can be useful for debugging purposes. However, it can also be useful if you want to create a duplicate handle to the object for another thread to use. This is not strictly necessary when using the default Pessimistic concurrency control, but is a requirement when using Optimistic (MVCC) (see relevant section).

> **Warning**
>
> Do not share the same reference for an Optimistic object with multiple threads. You must use the clone() operation for each thread.

There are two variants of the clone() operation. Both of them require an empty instance of the original non-STM class to clone the data in to (this does not actually happen for Pessimistic instances, but is still required at present for uniformity):

```
public synchronized T clone (T member, T proxy)
```

This version requires a reference to the STM object that is being cloned as the second parameter:

```
Sample1 obj2 = theContainer.clone(new Sample1Imple(), obj1);
```

The second version is similar:

```
public synchronized T clone (T member, Uid id)
```

This time instead of a reference you can provide the object's identifier:

```
Sample1 obj2 = theContainer.clone(new Sample1Imple(), theContainer.g
```

You are free to use either clone() operation depending upon what information your program has available.

6.5. State Management

Earlier in this chapter we discussed how you can instrument your implementation class member variables with the State and NotState annotations to indicate what state should be saved and restored by the transaction system. In some situations you may want even more control over this process and this is where the @SaveState and @RestoreState

annotations come in. These annotations let you define a method which will be called when the system needs to save your objec's state and likewise when it needs to restore it.

> **Note**
>
> You must use SaveState and RestoreState annotations together, i.e., you cannot just define one without the other.

Your methods can be called whatever you want but they must have the following signatures.

@SaveState

```
public void save_state (OutputObjectState os) throws IOException
```

@RestoreState

```
public void restore_state (InputObjectState os) throws IOException
```

Each operation is then given complete control over which state variables are saved and restored at the appropriate time. Any state-related annotations on member instance variables are ignored by the framework so you must ensure that all state which can be modified within the scope of a transaction must be saved and restored if you want it to be manipulated appropriately by the transaction.

For instance, look at the following example:

```
public class DummyImple implements Dummy
{
        public DummyImple ()
        {
            _isNotState = false;
            _saved = 1234;
        }

        @ReadLock
        public int getInt ()
        {
            return _saved;
        }

        @WriteLock
        public void setInt (int value)
        {
            _saved = value;
        }

        @ReadLock
        public boolean getBoolean ()
        {
            return _isNotState;
        }

        @WriteLock
        public void setBoolean (boolean value)
        {
            _isNotState = value;
        }

        @SaveState
        public void save_state (OutputObjectState os) throws IOExcep
        {
           os.packInt(_saved);
        }

        @RestoreState
        public void restore_state (InputObjectState os) throws IOExc
        {
            _saved = os.unpackInt();
        }

        public int _saved;
        public boolean _isNotState;
}
```

In this example, only the int member variable is saved and restored. This means that any changes made to the other member variable(s) within the scope of any transaction, in this case the boolean, will not be undone in the event the transaction(s) rolls back.

> **Warning**
>
> Use the SaveState and RestoreState annotations with care as you could cause dirty data to be visible between transactions if you do not save and restore all of the necessary state.

6.6. Optimistic Concurrency Control

Per object concurrency control is done through locks and type specific concurrency control is available. You can define locks on a per object and per method basis, and combined with nested transactions this provides for a flexible way of structuring applications that would typically not block threads unless there is really high contention. All but the @Transactional annotation are optional, with sensible defaults taken for everything else including locks and state.

However, the locking strategy we had originally was pessimistic. Most transaction systems utilize what is commonly referred to as pessimistic concurrency control mechanisms: in essence, whenever a data structure or other transactional resource is accessed, a lock is obtained on it as described earlier. This lock will remain held on that resource for the duration of the transaction and the benefit of this is that other users will not be able to modify (and possibly not even observe) the resource until the holding transaction has terminated. There are a number of disadvantages of this style: (i) the overhead of acquiring and maintaining concurrency control information in an environment where conflict or data sharing is not high, (ii) deadlocks may occur, where one user waits for another to release a lock not realizing that that user is waiting for the release of a lock held by the first.

The obvious alternative to this approach is optimistic or MVCC. Therefore, optimistic concurrency control assumes that conflicts are not high and tries to ensure locks are held only for brief periods of time: essentially locks are only acquired at the end of the transaction when it is about to terminate. This kind of concurrency control requires a means to detect if an update to a resource does conflict with any updates that may have occurred in the interim and how to recover from such conflicts. Typically detection will happen using timestamps, whereby the system takes a snapshot of the timestamps associated with resources it is about to use or modify and compares them with the timestamps available when the transaction commits.

As discussed previously, there are two annotations: @Optimistic and @Pessimistic, with Pessimistic being the default, i.e., if no annotation is present, then the STM framework will assume you want pessimistic concurrency control. These are defined on a per interface basis and define the type of concurrency control implementation that is used whenever locks are needed.

```
@Transactional
@Optimistic
public class SampleLockable implements Sample
{
   public SampleLockable (int init)
   {
      _isState = init;
   }

   @ReadLock
   public int value ()
   {
      return _isState;
   }

   @WriteLock
   public void increment ()
   {
      _isState++;
   }

   @WriteLock
   public void decrement ()
   {
      _isState--;
```

```
    }

    @State
    private int _isState;
}
```

And that's it. No other changes are needed to the interface or to the implementation. However, at present there is a subtle change in the way in which you create your objects. Recall how that was done previously and then compare it with the style necessary when using optimistic concurrency control:

```
Container theContainer = new Container();
Sample obj1 = theContainer.create(new SampleLockable(10));
Sample obj2 = theContainer.clone(new SampleLockable(10),obj1);
```

In the original pessimistic approach the instance obj1 can be shared between any number of threads and the STM implementation will ensure that the state is manipulated consistently and safely. However, with optimistic concurrency we need to have one instance of the state per thread. So in the above code we first create the object (obj1) and then we create a copy of it (obj2), passing a reference to the original to the container.

> **Warning**
>
> Remember that the same reference to Optimistic (MVCC) objects cannot be shared between different threads: you must use the clone() operation on the corresponding Container for each thread which wishes to use the object.

6.7. A Typical Use Case

In this chapter we have considered all of the publicly available interfaces and classes for the STM framework within Narayana. There is deliberately a lot of flexibility on offer but much of it will only be needed by more advanced users and use cases. In this section we shall consider the most typical way in which we believe users will want to use the STM implementation. Let's consider the interface first:

```
@Transactional
public interface Sample
{
       public void increment ();
       public void decrement ();

       public int value ();
}
```

Whilst MVCC (optimistic concurrency control) is available, it is most useful in environments with a high degree of contention. Even then, with the ability to control the timeout and retry values of the locking used by the pessimistic concurrency control option, the surety of making progress in a longer running transaction and not being forced to roll back later can be an advantage. Therefore, pessimistic (the default) is probably the approach you will want to take initially.

Now let's look at the implementation class:

```
public class MyExample implements Sample
{
        public MyExample ()
        {
            this(0);
        }

        public MyExample (int init)
        {
            _isState = init;
        }

        @ReadLock
        public int value ()
        {
            return _isState;
        }
```

```
        @WriteLock
        public void increment ()
        {
            _isState++;
        }

        @WriteLock
        public void decrement ()
        {
            _isState--;
        }

        private int _isState;
}
```

By this point it should look fairly straightforward. We've kept it simple deliberately, but it can be as complex as your application requires. There are no nested transactions at work here, but you can easily add them using the Nested annotation. Remember that they give you improved modularity as well as the ability to better control failures.

Because STM implementations typically relax or remove the durability aspect, you are more likely to want to create volatile objects, i.e., objects that do not survive the crash and repair of the JVM on which they are created. Therefore, you should use the default Container constructor, unless you want to control the name of the instance and in which case you can pass in an arbitrary string. Then all that is left is the creation and manipulation of AtomicActions as you invoke the relevant methods on your object(s).

```
MyExample ex = new MyExample(10);
Container<Sample> theContainer = new Container<Sample>();    Sample
AtomicAction act = new AtomicAction();

act.begin();

obj1.increment();

act.commit();
```

Chapter 7. BlackTie

7.1. Standards

7.1.1. X/Open

BlackTie implements the following standards: