Guide To *Software Transactional Memory* (**STM**)

Russell Winder

Version 1.0, 2015-10-01

Table of Contents

`Software Transactional Memory` (**STM**) gives developers transactional semantics for accessing in-memory data. This is similar to database concepts.

When multiple threads share data in memory, by marking blocks of code as transactional (atomic), the developer delegates the responsibility for data consistency to the **STM** engine. **GPars** leverages the Multiverse **STM** engine.

Running A Piece of Code Atomically

When using **STM**, developers organize their code into transactions. A transaction is a piece of code, which is executed **atomically** - either **1)** all the code is run or **2)** none at all.

The data used by the transactional code remains **consistent** irrespective of whether the transaction finishes normally or abruptly. While running inside a transaction, the code is given an illusion of being **isolated** from other concurrently running transactions so that changes to data in one transaction are not visible in the other ones until the transactions commit. This gives us the **ACI** part of the **ACID** characteristics of database transactions. The **durability** transactional aspect so typical for databases, is not typically mandated for **Stm**.

GPars allows developers to specify transaction boundaries by using the *atomic* closures.

Sample of **ACI** *Transaction Boundaries*

```
import groovyx.gpars.stm.GParsStm
import org.multiverse.api.references.TxnInteger
import static org.multiverse.api.StmUtils.newTxnInteger

public class Account {
    private final TxnInteger amount = newTxnInteger(0);

    public void transfer(final int a) {
        GParsStm.atomic {
            amount.increment(a);
        }
    }

    public int getCurrentAmount() {
        GParsStm.atomicWithInt {
            amount.get();
        }
    }
}
```

There are several types of *atomic* closures, each for a different type of return value:

- *atomic* - returning *Object*
- *atomicWithInt* - returning *int*
- *atomicWithLong* - returning *long*
- *atomicWithBoolean* - returning *boolean*
- *atomicWithDouble* - returning *double*
- *atomicWithVoid* - no return value

Multiverse, by default, uses an optimistic locking strategy and automatically rolls back and retries colliding transactions.

Developers should refrain from irreversible actions (e.g. writing to the console, sending e-mails, launching a missile, etc.) in their transactional code. To increase flexibility, the default **Multiverse** settings can be customized through custom *atomic blocks* .

Customizing the Transactional Properties

Frequently it's desirable to specify different values for some of the transaction properties (e.g. read-only transactions, locking strategy, isolation level, etc.). The *createAtomicBlock* method will create a new *AtomicBlock* configured with the supplied values:

Create an **AtomicBlock** *with Custom Parameters*

```
import groovyx.gpars.stm.GParsStm
import org.multiverse.api.AtomicBlock
import org.multiverse.api.PropagationLevel

final TxnExecutor block = GParsStm.createTxnExecutor(maxRetries: 3000, familyName:
'Custom', PropagationLevel: PropagationLevel.Requires, interruptible: false)
assert GParsStm.atomicWithBoolean(block) {
    true
}
```

The customized *AtomicBlock* can then be used to create transactions using the specified settings.

AtomicBlock instances are thread-safe and can be freely reused among threads and transactions

Using the *Transaction* Object

Atomic closures use the current *Transaction* as a parameter. The *Txn* object handle for a transaction can be used to manually control the transaction. This is illustrated in the example below, where we use the *retry()* method to block the current transaction until the counter reaches the desired value:

A Sample

```groovy
import groovyx.gpars.stm.GParsStm
import org.multiverse.api.PropagationLevel
import org.multiverse.api.TxnExecutor

import static org.multiverse.api.StmUtils.newTxnInteger

final TxnExecutor block = GParsStm.createTxnExecutor(maxRetries: 3000, familyName:
'Custom', PropagationLevel: PropagationLevel.Requires, interruptible: false)

def counter = newTxnInteger(0)
final int max = 100

Thread.start {
    while (counter.atomicGet() < max) {
        counter.atomicIncrementAndGet(1)
        sleep 10
    }
}

assert max + 1 == GParsStm.atomicWithInt(block) { tx ->
    if (counter.get() == max) return counter.get() + 1
    tx.retry()
}
```

Data Structures

You might have noticed in previous examples that we use dedicated data structures to hold values. The fact is that normal **Java** classes do not support transactions and thus cannot be used directly, since **Multiverse** would not be able to share them safely among concurrent transactions, commit them nor roll them back.

normal **Java** classes do not support transactions

We need to use data that knows about transactions:

- TxnIntRef
- TxnLongRef
- TxnBooleanRef
- TxnDoubleRef
- TxnRef

You typically create these through the factory methods of the *org.multiverse.api.StmUtils* class.

More Information

We decided not to duplicate the information that was already available on the **Multiverse** website.

Unfortunately with the closure of Codehaus, that website is longer available. You may try to gather more information from the Multiverse source code.

As we are unclear about the future of the **Multiverse** project, we will consider using a different **STM** implementation in a future **GPars 2.0**.