More Software Transactional Memory
Goals

• Complete our review of the material in Chapter 6 of our concurrency textbook
  • Examine more in depth examples
    • using STM
    • in Java
    • via the Akka framework
Last Time

• Introduced notion of software transactional memory
  • Approach to concurrency based on the use of transactions
    • to update identities (or refs) that have a mutable association with an immutable value
      • at any one point in time, the ref has one and only one value
      • in a transaction, we can change the ref’s association to a different immutable value
  • This approach can achieve better utilization of cores than traditional lock-based/synchronization-based approaches to concurrency because it employs an optimistic locking approach in which a thread encounters overhead (unnecessary work) when a write contention occurs
Picking Up Where We Left Off: Nested Transactions

• Nested transactions occur
  • when a method executing inside of a transaction
  • calls another method that starts a new transaction

• Akka can be configured to handle nested transactions in various ways
  • the default is that changes made by inner transactions are not committed
    • until the outer transaction is committed
  • thus the changes made by the inner transactions are local to the outer transaction
    • all such changes will be rolled back as a group if the outer transaction has to be retried
Example: Transferring Money Between Accounts

• We return to an example we saw back in Chapter 4
  • transferring money back between bank accounts
• This situation is ideal for nested transactions
  • the outer transaction is the transfer in total
    • the inner transactions are
      • the withdrawal from one account
      • the deposit into a second account
• The Chapter 4 version that used a Lock to implement the transfer
  • the STM version is more concise and has no locks; DEMO
Configuring Transactions

• Akka provides a way to configure transactions programmatically
  
  • by use of a TransactionFactory class

  • An instance of this class can be passed to an instance of the Atomic<T> class to configure properties of the transaction that it creates

  • A TransactionFactoryBuilder is used to create an instance of TransactionFactory

    • the book shows how to make a transaction “read only” but the documentation to TransactionFactoryBuilder reveals methods for setting whether a transaction is interruptible, how many times it can be retried, what its timeout is if blocked, whether it CAN be blocked, etc.

• The example creates a read only transaction and then tries to change a ref; DEMO
Blocking Transactions

• If we have a transaction that fails because the value of one of its refs is in a state that prevents the transaction’s logic from doing its job

  • For instance, withdrawing $500 from an account that has only $200

• Akka will allow a transaction to enter a queue to be retried but to wait (block) until the ref it depends on has been changed

  • You need to configure the transaction to enable blocking and you need to specify how long you are willing to wait

  • Then, within the transaction, you check the value of the ref that you depend on and if you can’t do your job, you call retry()

  • Your transaction will then be blocked until it can make progress

• The example involves getting cups of coffee from a coffee pot that will be refilled on a periodic basis; some transactions will block between refills; DEMO
Transaction Event Handlers

- Akka provides a means for executing code when
  - a transaction succeeds (i.e. commits successfully)
  - or when a transaction fails (i.e. is rolled back)
- Within our atomically() method, we first configure our event handlers by
  - calling deferred() and passing in an instance of Runnable containing the code that should execute when our transaction succeeds
  - calling compensating() and passing in an instance of Runnable containing the code that should execute when our transaction fails
- Note: this code will run in a separate thread and the code in compensating() may run multiple times once for each time its associated transaction fails
- Design Accordingly! DEMO
Dealing with Non-Primitive Values (I)

• The examples so far have all associated primitive values with our refs
  • But applications are much more complex and application-specific classes and their instances will be needed as well
    • If so, these classes need to be made immutable
      • The class needs to be declared final
      • All instance variables need to be marked as final
        • And, all of their values need to be immutable
      • When a change is made, we make a copy; no mutable state!
    • The problem of course is we need to be smart about how we do this; inefficient copying can lead to too much memory being used
Dealing with Non-Primitive Values (II)

• In addition to using immutable application-specific classes
  • we must also make sure that when we need to use a collection class
    • that it is implemented to support immutability via persistent data structures
  • Akka provides access to two persistent collection classes in Java
    • TransactionalVector and TransactionalMap
• These classes behave like arrays and hash tables but honor Akka’s transaction semantics
  • You can make as many changes as you need to them in a transaction
    • if the transaction fails, all of the changes are discarded; DEMO
Dealing with Write Skew

- As we saw in lecture 19, STM can fall prey to write skew
  - The situation where two transactions can meet application properties in isolation but violate an application property globally after both of their effects are applied
    - The example we looked at concerned withdrawals on checking and savings accounts in which the sum of their balances must always be greater than or equal to $1000
  - Akka supports the ability to avoid write skew by triggering transaction rollback when any ref accessed by a transaction is updated by some other transaction (regardless of whether we update the ref or not)
    - You just need to configure it via the TransactionFactory; Demo
Limitations (I)

• STM has a number of properties to make it attractive as an alternative means of designing concurrent software systems with shared mutability

  • But, it does have some limitations

• In particular

  • STM is ideal for those applications where write contention happens rarely

  • If your application will have lots of threads changing the same identity, then STM is not the best fit

    • The book demonstrates this by revisiting the FileSize application again

      • It spawns too many threads all updating the same refs

        • any significant directory hierarchy causes the program to fail
Limitations (II)

- The book, The Joy of Clojure, identifies two additional limitations
  - IO cannot be performed during a transaction
  - Transactions need to be short
- The reason?
  - IO operations are not idempotent
    - Each time you perform an IO operation, you can get a different result
    - Thus, if you have an IO operation in your transaction and the transaction fails then the transaction is going to be retried and the IO operation will be invoked again
  - Long transactions have a high risk of failure; will get stuck in retry loop
Summary

• STM is an alternative approach to concurrency with major benefits
  • Provides maximum concurrency via lock-free concurrent programming model organized around transactions
    • Changes to shared mutable state only happen in transactions
    • No race conditions due to transaction semantics; no visibility problems
    • With no locks, deadlock and livelock are eliminated
  • It does have limitations
    • Application must have minimal write contention
    • No IO during transactions
    • No long transactions
Coming Up Next

• **SPRING BREAK!!!**

• Lecture 21: Agile Project Execution