Advanced Actor Model

CSCI 5828: Foundations of Software Engineering
Lecture 25 — 04/17/2012

© Kenneth M. Anderson, 2012

Goals

- Review the material in Chapter 8 of the Concurrency textbook
  - that deals with the use of TypedActors in Akka
  - and mixing
    - the Actor model with
    - the Software Transactional Memory model
  - in the same program, given that Akka implements both!

Review: The Actor Model (I)

- The Actor Model of Concurrency adopts the approach of
  - isolated mutability
- We have individual actors that act independently of one another
  - Each one is allowed its own set of mutable variables
  - Each actor has access only to its variables
    - A can't access B's mutable variables
  - Akka enforces this by never giving us a direct reference to an actor
    - We only ever deal with ActorRefs

© Kenneth M. Anderson, 2012

Review: The Actor Model (II)

- Each actor has a queue and sits waiting for messages to arrive
  - Work is performed by having actors pass messages to each other
    - thus triggering an actor to perform work while processing the message
- Only immutable values get passed from one actor to another
  - no chance for race conditions, no chance for visibility problems, etc.
- Actors are independent of threads
  - multiple threads can handle the execution of a single actor
  - multiple actors might be run by a single thread
- We do not have to worry about assigning actors to threads as well
  - As we saw, Akka did a good job of distributing actors across threads (and cores) for both IO intensive and compute intensive applications!

© Kenneth M. Anderson, 2012

Untyped Actors (I)

- In our previous lecture, we looked at the use of “untyped actors”
  - In Akka, that means that from an API standpoint
    - one actor looks like another
    - If I had three different classes (A, B, and C) implement UntypedActor, they would all have the same interface at run-time
      - That is, they would all have a runtime type of ActorRef and they would all respond (basically) to the ask() and tell() methods
  - The difference, of course, is that each would handle different types of messages
    - for instance in the FileSizer example, we had messages types for the SizeCollector and others for the FileProcessor

Untyped Actors (II)

- With the need for creating your own message types for UntypedActors
  - we saw classes like this in the FileSizer example
    - class FileSize {
      - public final long size;
      - public FileSize(final long fileSize) { size = fileSize; }
    - }
  - you would then create instances like this
    - new FileSize(size)
  - and access them like this
    - ((FileSize)(message)).size;

Hmm. That's "less than ideal"

Untyped Actors (III)

- The problem?
  - We’re losing our ability to rely on OO principles in our design
    - just to take advantage of a new approach to concurrency
  - We’re now surfacing the details of messages
    - in a way that is unfamiliar and awkward
  - We have to do a bunch of runtime checks on generic instances to handle all the different message types
    - “if (message instanceof FileSize) {” is a “bad smell” in OO programming

© Kenneth M. Anderson, 2012

TypedActors

- Akka's TypedActors attempts to address this situation
    - You can design your own class for your actor following good OO principles
    - You can get a handle to an instance of your class
        - (You still use an Akka factory to create that instance, however)
    - You can then call your actor's methods as you would normally
        - behind the scenes, those method calls
            - get intercepted and
            - converted to asynchronous nonblocking messages passed to the actor in the same manner that we saw with UntypedActors

Creating a TypedActor

- To create a typed actor, we need to create two things
  - An interface that declares the methods of our TypedActor
  - An subclass of TypedActor that implements the above interface
- To instantiate a typed actor, we pass our interface and our subclass to an Akka factory and get back an instance of our TypedActor subclass
  - final Foo f = TypedActor.newInstance(Foo.class, FooImpl.class);

Using a Typed Actor

- If Foo defined a method: void addBar(Bar b)
  - then a call to f.addBar(b) would be converted
    - to passing a message containing the immutable value “b”
    - to the actor pointed at by f using Akka’s tell() method
- If Foo defined a method: int numberOfBars();
  - then a call to “int count = f.numberOfBars” would be converted
    - to passing an empty message
    - to the actor pointed at by f using Akka’s ask() method
    - The use of Future to retrieve the result is handled automatically

Modifying AKKA_JARS

- To make use of TypedActors, we need to add additional jars to our AKKA_JARS environment variable
    - export AKKA_JARS="$AKKA_JARS:$AKKA_HOME/lib/akka/akka-typed-actor-1.3.1.jar"
- and
    - export AKKA_JARS="$AKKA_JARS:$AKKA_HOME/lib/akka/aspectwerkz-2.2.3.jar"
- In other words, you need to add akka-typed-actor-1.3.1.jar and aspectwerkz-2.2.3.jar to your classpath

Increment Lives Again! (Groan)

- Let's implement our increment program one more time
  - This time using a typed actor
- First, we need an interface

```
public interface Counter {
  void increment(final int delta);
  int getCount();
}
```

Increment (II)

- Second, we need an implementation

```
import akka.actor.TypedActor;
public class CounterImpl extends TypedActor implements Counter {
  public int count = 0;
  public int getCount() {
    return count;
  }
  public void increment(final int delta) {
    count += delta;
  }
}
```

© Kenneth M. Anderson, 2012

Increment (III)

- Third, we need an interface and implementation for Drone

```
public interface Drone {
  void go();
}
import akka.actor.TypedActor;
public class DroneImpl extends TypedActor implements Drone {
  private final Counter counter;
  public DroneImpl(final Counter counter) {
    this.counter = counter;
  }
  public void go() {
    for (int i = 0; i < 5; i++) {
      counter.increment(1);
    }
  }
}
```

Increment (IV)

- Finally, we need a program that creates the counter object and the drones and invokes them
- Here's the code to create the Counter object
    - `final Counter counter = TypedActor.newInstance(Counter.class, CounterImpl.class);`
- We don't know what the actual type is for the instance passed back from newInstance().
    - Here's the important thing: We don't care!
    - As long as it responds to the Counter interface, that's all we need!
- **DEMO**

Returning to Energy Source, Step One

- The book's example for typed actors returns to the Energy Source example
  - An interface for the energy source is defined
    - `public interface EnergySource {`
      - `long getUnitsAvailable();`
      - `long getUsageCount();`
      - `void useEnergy(final long units);`
    - `}`
- The energy source is then implemented as a typed actor
- The main program shows that typed actors behave like untyped actors with respect to threads
  - different threads might be used for different messages sent to the same actor; **DEMO**

Energy Source, Step Two: I'll do it my way

- The second part of the energy source example in the book was very complicated
    - It relies on a language feature of Scala known as traits
    - Traits have a weird manifestation inside of Java
        - and I decided not to try to teach Scala in this class
        - So, I'm going to skip the book's implementation and
            - try to implement it another way
- Basically, we need a replenish() method on the energy source and a way to invoke the method on the energy source automatically
    - We'll create a service that can handle this for us
        - our main program will create the energy source and then pass it to this service; the service will use a Timer to invoke replenish each second

Combining the Actor Model with STM

- The Actor Model is designed for problem domains where
  - concurrent tasks can run independently from one another
  - and communication via asynchronous messages is enough to coordinate between tasks
- The actor model does not provide a means for managing consensus across tasks (or actors)
  - We may want the actions of two or more actors to all succeed or all fail collectively (which sounds like a transaction)
- Indeed, such behavior is possible by combining the actor model with STM

© Kenneth M. Anderson, 2012

Returning to the Account Transfer Example

- The account transfer example is sufficient to show why you will sometimes need to combine both of these new concurrency models
  - Account objects can be implemented as actors
    - withdrawals and deposits needs to occur in a consistent fashion
      - single-threaded actors with isolated mutability are perfect for that
  - A transfer between accounts, however, requires coordination between the two actors that manage the two accounts
    - if the deposit action succeeds but the withdrawal fails, we need to be able to roll back the deposit
    - both actor actions need to succeed or they both need to fail

Transactors (I)

- Akka provides several ways to mix actors and STM
  - We'll first look at Akka's transactional actors, known as transactors
- Transactors act like UntypedActors but
  - rather than use the onReceive() message to handle messages
  - they use the method atomically()
    - and so handling a message occurs inside of a transaction
    - which means it can update the value of a Ref without causing an error
    - changes to Ref's will roll back if the transaction fails
  - atomically() otherwise acts like onReceive()
    - each message is handled one at a time

© Kenneth M. Anderson, 2012

Transactors (II)

- To include other actors in on a single transaction
  - we implement Transactor's coordinate() method
  - In that method, we identify the other methods and send messages to them
- They will process those messages as part of the transaction created by the calling Transactor
  - if any of the sub-actors fail, it will cause the entire transaction to roll back
- A Transactor that implements coordinate() is still required to implement atomically(); if so, it may do work in tandem with the sub-actors
  - if it's actions fail, the transaction will roll back negating the actions of the sub-actors

© Kenneth M. Anderson, 2012

Account Transfer Example

- Defines two Transactors
  - Account and AccountService
- Defines five messages
  - Deposit, Withdraw, FetchBalance, Balance, Transfer
- Account handles the first four messages
- AccountService only handles Transfer
- The main program demonstrates both successful and failed transfers

- **DEMO**

© Kenneth M. Anderson, 2012

Coordinating Typed Actors

- The code becomes much simpler when coordinating typed actors
  - All we need to do is to add the java annotation @Coordinated to any methods in the interface of our typed actor
    - The only limitation is that the method have a return type of void
  - Then to ensure that method calls on typed actors are coordinated (i.e., they run in a transaction and either all succeed or all fail) is to call those methods inside of a coordinate() method call
- That's it
  - The account service example is much more concise than the previous version: withdraw and deposit are marked as coordinated; the transfer service calls those methods inside of a call to coordinate(); simple! **DEMO**

© Kenneth M. Anderson, 2012

Remote Actors

- Akka has functionality to allow for actors to be in separate processes
  - It provides a registry for locating remote actors and sending them messages
- Other than a few changes to indicate that an actor you want to send a message to is in some other process, the interactions with remote actors are the same
- See the book for details
  - Note: you need to add additional jars to your AKKA_JARS environment variables to get the remote example to run

Actor Limitations

- The Actor model has a few limitations
  - You need to make sure that messages are immutable
    - Otherwise, you can see shared mutable effects (i.e. instability, race conditions, etc.) creep into actor model programs
  - Actors can starve
    - if an actor fails, then it will not send out its messages
    - if some other actor is waiting for those messages
      - it will sit there and do nothing
  - Actors can deadlock
    - if you design two actors to wait for messages from each other: BOOM!

© Kenneth M. Anderson, 2012

Summary

- The actor model is a very powerful model that is scalable and efficient
  - We get to write code that makes use of isolated mutability
    - allowing us to change the value of a variable in a well understood way
  - We handle one message at a time; message passing and the queues are all handled for us; we just send messages
  - Asynchronous, one-way messages can be sent very efficiently
    - helping to enable high concurrency between agents
  - Agents can share a pool of threads
    - the thread used by a thread can change from message to message
- Today, we saw how to use
  - typed actors for more concise, understandable actor model code
  - how to mix actors (both typed and untyped) with the STM

Coming Up Next

- Lecture 26: Creating Agile Software
- Lecture 27: TBD