Hazard Reduction in Scala Actors

Marc Stiegler

HP Laboratories
HPL-2009-148

Keyword(s):
Parallel programming, concurrency, SCALA, message passing

Abstract:
We identify four fine grain data race hazards in the Scala Actors architecture. These hazards are data races that the compiler cannot catch and that the programmer is likely to cause by implementing code in haste to meet a deadline. We present the Actress wrapper for the Actor system which mitigates the risks with the three most dangerous hazards. These hazards are mitigated by a combination of 1. Enabling static type checking support and 2. modifying programming affordances to make the easiest way to implement concurrency the safest way.

External Posting Date: July 6, 2009 [Fulltext] Approved for External Publication
Internal Posting Date: July 6, 2009 [Fulltext]

Hazard Reduction in Scala Actors

Marc Stiegler

HP Labs

marc.d.stiegler@hp.com

Abstract

We identify four fine grain data race hazards in the Scala Actors architecture. These hazards are data races that the compiler cannot catch and that the programmer is likely to cause by implementing code in haste to meet a deadline. We present the Actress wrapper for the Actor system which mitigates the risks with the three most dangerous hazards. These hazards are mitigated by a combination of

- *Enabling static type checking support and*
- *modifying programming affordances to make the easiest way to implement concurrency the safest way.*

Problem

The Scala Actors system for implementing concurrency was inspired by the Erlang model of concurrent computation [Vermeersch09]. The original Erlang model is free of fine-grain data races and reduces the risk of deadlock compared to lock-based models of computation. The Scala adaptation shares the deadlock risk reduction of Erlang. However, while the risk of fine-grain data race is mitigated compared to lock-based models, the Scala implementation of actors includes 4 data race hazards, i.e., there are four ways in which the Scala programmer suffering a tight schedule or momentary lapse of caution can accidentally introduce races. These races are especially hazardous because they may only cause failures after field deployment, and they may be difficult to reproduce and debug. These four hazards all rest on the following difference between the Scala and Erlang: it is possible in Scala to accidentally share mutable state across multiple actors and therefore across multiple threads.

Here we list the 4 hazards. We have ordered them according to our best assessment of the risk, from least dangerous to most dangerous:

- Mutable state may be included in the definition of a singleton that is accessed by more than one actor. We consider the danger from this hazard to be smallest because such singleton mutable state effectively represents a global variable in the Scala context, and mutable global state is generally recognized as modularity-breaking poor programming style for reasons besides concurrency. Many organizations have programming guidelines prohibiting such global mutables, and software reviewers are particularly keen in criticizing such usage. In addition, Scala programming examples in popular texts [Odersky08] never demonstrate usage of mutable singletons, presumably for all of these reasons, and programmers generally understand that they can write programs successfully without them.

  Example: object MutableGlobal {var x = 0}

- A mutable object may be passed as an argument to the constructor for an Actor subclass. We consider this to be a relatively less dangerous hazard because the hazard can be caught by inspecting a single line of code, namely, the constructor's parameter list.

  Example: Class Racer(data: Array[Int]) extends Actor { ... }

- A mutable object may be passed between actors using the send (!) message. Standard documentation warns against this [Odersky08a], but there is no compiler or library support to assist the programmer in self defense. Multiple lines of code must be examined to identify such an error (namely, all the case

patterns in the react/receive block of the actor), making it less likely that a casual observation can note a problem.

Example: racer ! ("race", myArray)

- A mutable object may be captured by an actor created inline using the actor{} closure. This is a subtle error that is quite easy to make. Consider the following 2 code snippets: one is safe, and the other has a latent data race waiting to be exercised, though both work fine in simple tests:

```
/* safe version */
def parent = actor {
   def child = {
      var x = 0
      ....
      /* inside react */ x+= 1
      …
   }
}

/* unsafe version */
def parent = actor {
   var x = 0
   def child = actor {
      …
      /* inside react */ x += 1
      …
   }
   ….
}
```

Proving that an actor has not captured mutable state from its enclosing scope (which runs on a separate thread) requires identifying every variable that is captured during construction, which can require an examination of most of the file in which the actor is created. Even if the first version of the code avoids data race, every successive maintenance upgrade of the code in the file offers an opportunity to silently introduce such a race. Further, this style of creation for actors is convenient and is often used in the sample code for papers and examples [Haller07]. Consequently, this hazard will be encountered frequently. For all these reasons, for large scalable code systems, we consider this to be the greatest hazard.

Data Race Mitigation with Actresses

Actresses use two techniques to mitigate hazards 2, 3, and 4 (the three most dangerous hazards). First, Actresses introduces the extensible Sendable collection of case classes for argument passing, which enables Scala's static type checking system to assist the programmer in detecting data races at compile time. Second, Actresses changes the affordances of actress construction so that the easiest way to construct an actress is the way that captures no mutable shared state.

The abstract Sendable class and its case classes, which underly Actress's ability to enable some static type checking support, can be seen in the sources in the Appendix.

The important features of the Sendable class are:

- There are 4 general categories of objects in the Sendable cases:

    - basic immutable objects, such as Strings and Ints
    - other Actresses, which can only be accessed via their concurrency-safe send (!) message
    - data structures that have been explicitly designed to be thread-safe, such as Java's ConcurrentHashMap. Including these structures that are separately designed to be thread-safe is important to the Scala Actors goal of achieving maximum performance in shared-memory environments where the queuing and dequeuing operations for message passing may be a noticeable overhead.
    - immutable collections of Sendables (such as the ListS case class)

- The Sendable class is extensible with additional cases. So programmers can make other Sendable cases for immutable arrays as well as immutable lists, and can use other threadsafe collections in addition to the hashmap in the basic Sendable collection. It is also useful for creating "method name" classes: if a ping-pong actress needed to respond to a parameterless Ping message, one could create

  ```
  case class Ping() extends Sendable
  ```

  The extensibility also enables the programmer to address a performance risk with actresses. For example, sending a List[Int] in actress using the basic Sendable cases would require mapping the List[Int] into a ListS[IntS] before transmission, and might require mapping back from ListS[IntS] to List[Int] again inside the actress. If this code is performance sensitive and the list can be large, it would make sense to create a new Sendable class

  ```
  case class ListIntS(listInt: List[Int])
  ```

  that would require no such transformations.

  Making the Sendable class extensible introduces a hazard -- the programmer can create unsafe cases -- but such an explicitly constructed special class whose purpose is to assert safety increases the chances that even the harried programmer will spot a problem before he creates it. It also makes it easier for the reviewer to do a concurrency safety inspection simply by examining the Sendable cases.

- Raw values can be easily extracted from a Sendable via pattern matching. In the PingPong example at the end of this report, the Pinger actress receives a Sendable at construction time that consists of a list with a msg string and a ping count integer. The arguments are constructed like this:

  ```
  new Pinger(ListS(List(StrS("ping"), IntS(10))))
  ```

  and extracted like this:

  ```
  val (msg, numPings) = args match {
    case ListS(StrS(amsg) :: IntS(anum):: Nil) => (amsg,anum)
    case  _ => ("bad args", 1)
  }
  ```

In Actress, the send (!) operator only accepts Sendable arguments, mitigating risk (3). Similarly, the Actress abstract class from which actresses are derived requires a Sendable argument list in its constructor. As a consequence, the quickest and easiest way to initialize an actress is to pass it a Sendable list of arguments, mitigating risk (2).

Risk 4, the risk of accidentally capturing mutable state from an actor{} closure, is mitigated by putting the natural and easy place to create the actor in a safe place. During actress construction, the underlying actor is created by invoking a protected abstract method

make() :Actor

The easiest way to implement the actor is to implement this method with an actor{} closure:

protected make() :Actor = actor { ... }

Such an actor will capture only the safe Sendable state from the constructor parameter.

Issues

While Actresses mitigates the risk of accidentally violating the implicit contract of the actors library, it does not eliminate these risks. Also, there is a syntactic burden in using actresses: the programmer must construct and deconstruct the Sendable arguments to both the actress constructor and the receive/react pattern match inside the actor.

Conclusion

4 data race hazards in scala actors are identified. The Actress wrapper for the actor system is introduced, which uses 2 techniques to mitigate the 3 most dangerous of these hazards. First we introduce the Sendable case classes, which enable compiler support in detecting hazardous message sends and hazardous object constructors. Second, the actress abstract class makes it attractive to write the underlying actor{} closure as part of a protected method that will capture no sharable mutable state. While these enhancements do not guarantee an elimination of data races, these enhancements work together to make the easy way of creating actresses the safe way.

References

[Vermeersch09]: http://ruben.savanne.be/articles/concurrency-in-erlang-scala
[Odersky08]: Odersky, Spoon, Venners, Programming in Scala, Artima Press, 2008.
[Odersky08a]: Odersky, Spoon, Venners, Programming in Scala, Artima Press, 2008, Chapter 30.
[Haller07]: Haller and Odersky, Actors that Unify Threads and Events, Proc. COORDINATION 2007.

Appendix: Actress Package and Example Program (PingPong)

Note: hp.actress.Sendable and hp.actress.Actress embody the actress library. hp.actress.Ping and hp.actress.PingPongMain embody the example.

```
package hp.actress
import java.util.concurrent.ConcurrentHashMap
abstract class Sendable
  case class IntS(x: Int) extends Sendable
  case class BoolS(b: Boolean)  extends Sendable
  case class StrS(s: String) extends Sendable
  case class ActS(a: Actress) extends Sendable
  case class ListS(l: List[Sendable]) extends Sendable
  case class HashS(h: ConcurrentHashMap[Sendable, Sendable]) extends Sendable

  --------

package hp.actress
import scala.actors.Actor
```

```
abstract class Actress (initialState: Sendable) {
      protected def make() :Actor
      private val performer = this.make()
      def ! (msg: Sendable): Unit = {performer ! msg}
}

---------------

package hp.actress

import scala.actors.Actor._
import scala.actors._

class Pinger (args: Sendable) extends Actress(args) {
      protected def make(): Actor = actor {
        val (msg, numPings) = args match {
          case ListS(StrS(amsg) :: IntS(anum):: Nil) => (amsg,anum)
          case  _ => ("bad args", 1)
        }
        var pingCount = 0
        loop {
          react {
            case ActS(partner) =>
              println(msg)
                partner ! ActS(this)
                pingCount += 1
                if (pingCount >= numPings) {
                  println("actress finished");
                  exit
                }
            case bad => println("bad msg" + bad)
          }
        }
    }
}

------------
```

```
package hp.actress
import scala.actors.Actor
object PingPongMain {

      def main(args: Array[String]) = {
        val numPings = IntS(10)
        val pinger = new Pinger(ListS(List(StrS("ping"), numPings)))
        val ponger = new Pinger(ListS(List(StrS("pong"), numPings)))
        pinger ! ActS(ponger)
      }
}
```