Actor Bug: Concurrency – Operation Order

Much like message ordering, there are sometimes issues that occur as a result of the order of a job being carried out on a system. This should be considered while developing a system as most concurrent systems are non-deterministic in nature.

SQUBS found itself with an operational order bug in which an actor monitoring system failed to wait for the actors it was monitoring to begin functions. The ActorMonitorSpec would assume that each actor in the system was active as soon as it was able to send them messages. This would cause the system to hang if those actors were not yet functional. This was solved by utilizing Akka futures which will send non-blocking message requests that can be checked on when appropriate, such as when the expected actors have begun operations. This is a subtle difference to the Communication Message Order characteristic in that this is a matter of when the system is operating in conjunction with other parts of the system rather than what order sent messages are received.

As can be seen below, the case of acting before the system is ready was solved by using futures. Each of the necessary actor systems were queried with a future and then the ActorMonitorSpec class would wait on the sequence of those 4 futures replying. Receiving a reply means those actors were no active and the actor monitor could move forward with it's operations.

Link: https://github.com/paypal/squbs/issues/29

Code:

squbs/actormonitor/ActorMonitorSpec.scala

```
 class ActorMonitorSpec extends TestKit(ActorMonitorSpec.boot.actorSystem) with ImplicitSender
                                with WordSpecLike with Matchers with BeforeAndAfterAll
-                               with AsyncAssertions {
+                               with AsyncAssertions with LazyLogging {

   implicit val timeout: akka.util.Timeout = Timeout(1 seconds)
   implicit val ec = system.dispatcher
+  override def beforeAll() {
+    // Make sure all actors are indeed alive.
+    val idFuture1 = (system.actorSelection("/user/TestCube/TestActor") ? Identify(None)).mapTo[ActorIdentity]
+    val idFuture2 = (system.actorSelection("/user/TestCube/TestActorWithRoute") ? Identify(None)).mapTo[ActorIdentity]
+    val idFuture3 = (system.actorSelection("/user/TestCube/TestActorWithRoute/$a") ? Identify(None)).mapTo[ActorIdentity]
+    val idFuture4 = (system.actorSelection("/user/TestCube/TestActor1") ? Identify(None)).mapTo[ActorIdentity]
+    val futures = Future.sequence(Seq(idFuture1, idFuture2, idFuture3, idFuture4))
+    val idList = Await.result(futures, 2 seconds)
+    idList foreach {
```

```
+      case ActorIdentity(_, Some(actor)) => logger.info(s"beforeAll identity: $actor")
+      case other => logger.warn(s"beforeAll invalid identity: $other")
+    }
+
+    import ActorMonitorSpec._
+    val cfgBeanCount = getActorMonitorConfigBean("Count").toString.toInt
+    cfgBeanCount should be > 11
+  }
```