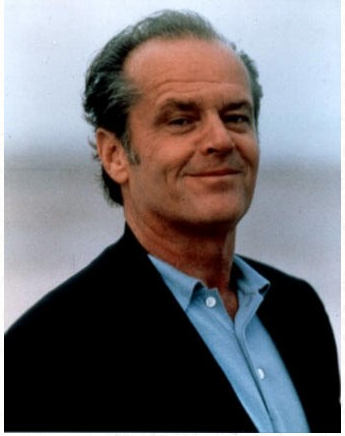
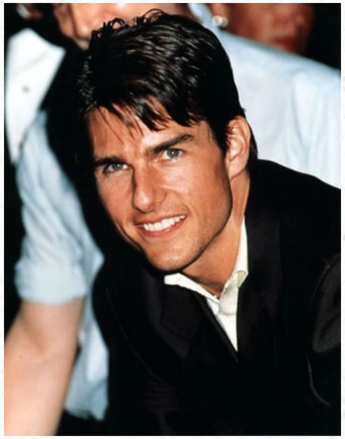
Scaling Up and Out with Actors

St. Louis Java Users Group
June 9th, 2011

Tim Dalton
Senior Software Engineer
Object Computing Inc.



Actors ?



What are actors?

Actors provides simpler unified model consisting of:

- Isolated computing entities (Actors)
  - “Share nothing”
  - Nothing to synchronize
- Asynchronous message passing
  - Immutable messages
  - Actors have a mailbox / queue for messages

Where does it come from ?

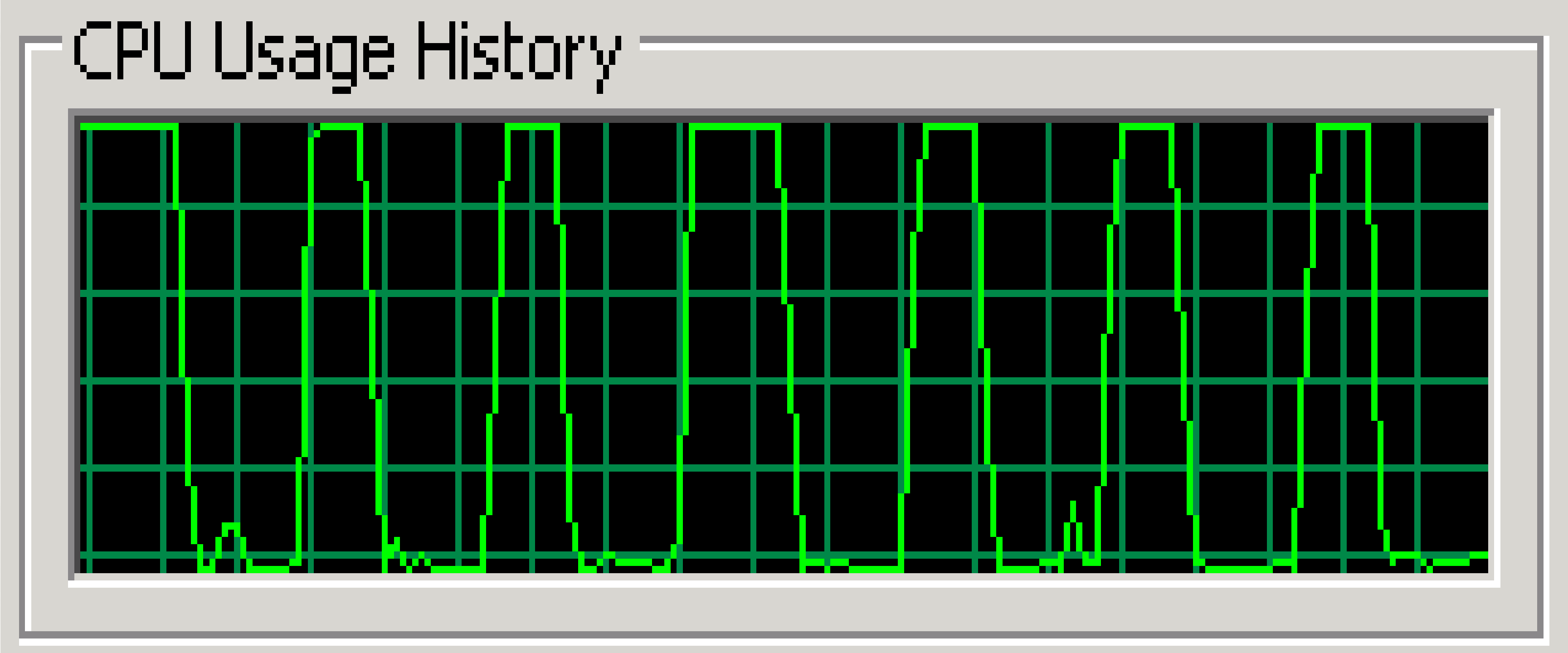
- First defined in paper by Carl Hewitt in 1973

- Popularized by Erlang OTP (Open Telecom Platform)

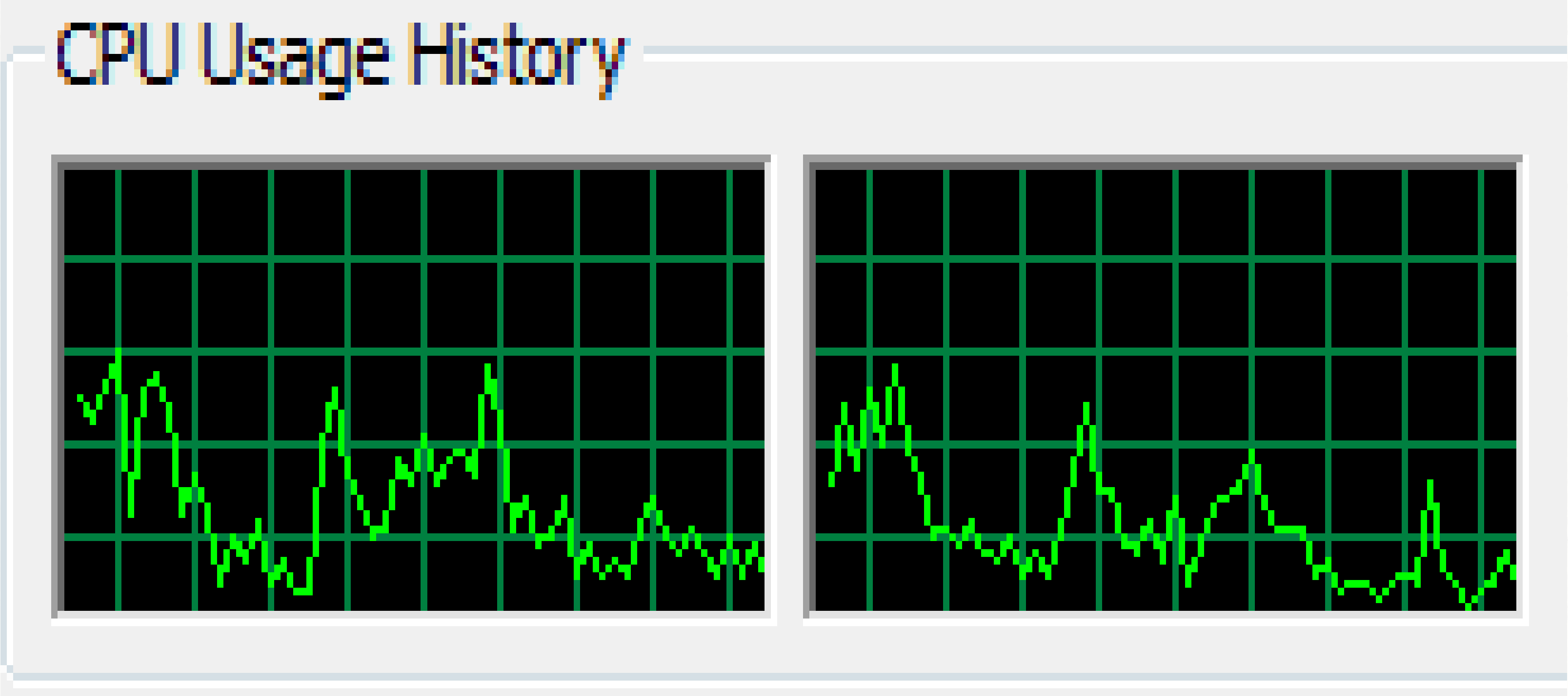
- Message passing closer to the original intent for Object Orientation
  - According to Alan Kay:

    "OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things"

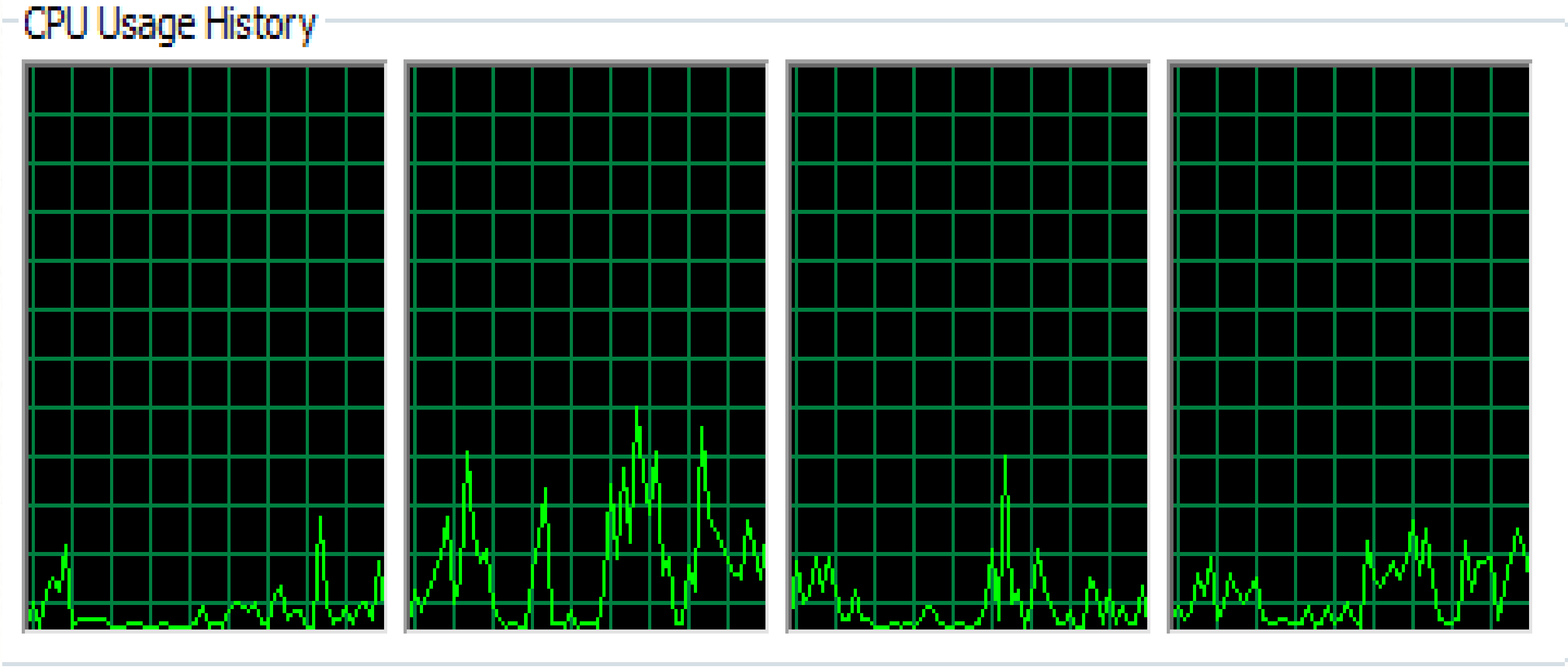
Why actors ? Because this:



Became this



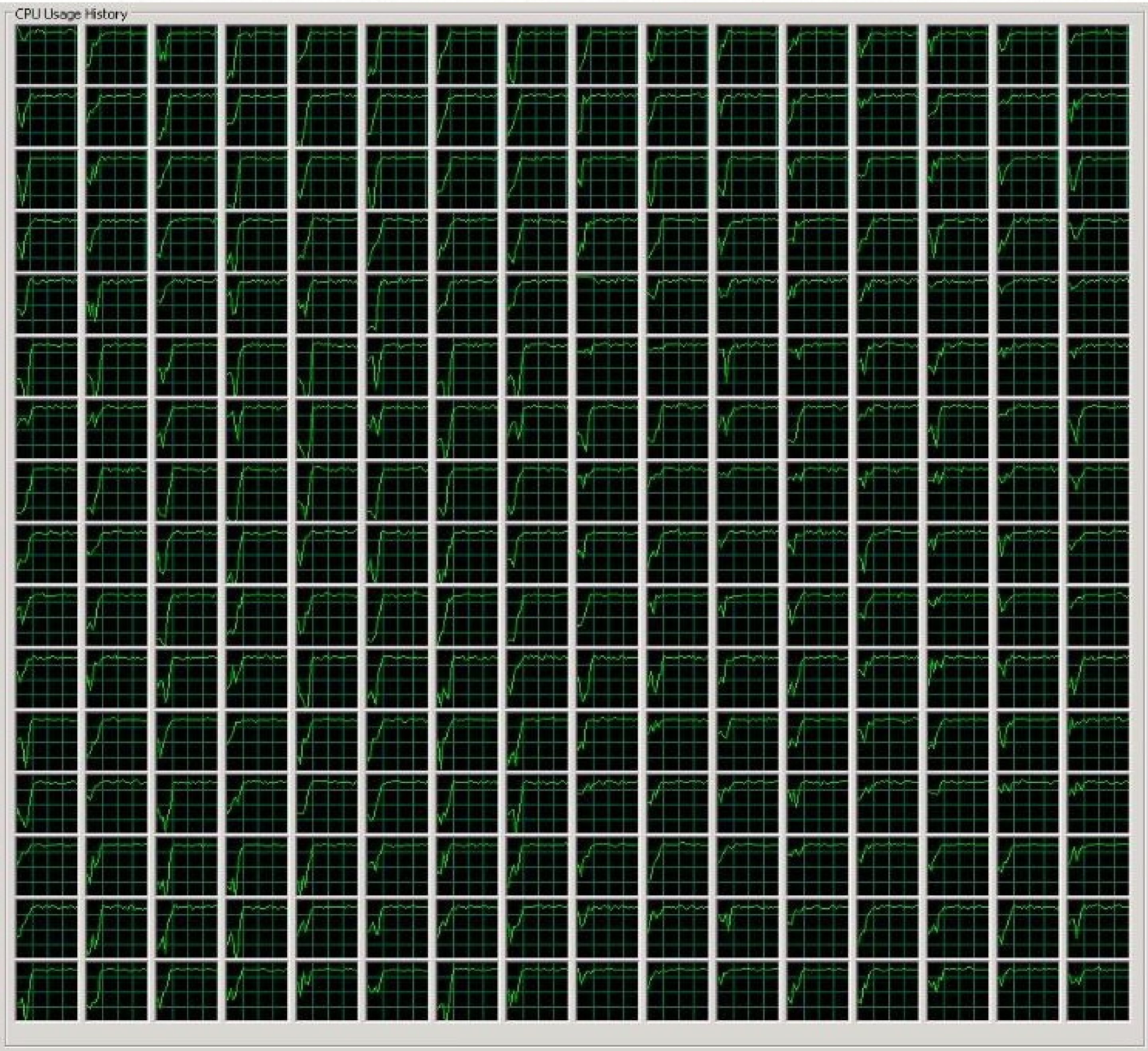
Became this



CPU Usage History

Will soon become ...

THIS



Sometimes this is needed

Not a silver bullet...

… or pixie dust

Another tool in our “Concurrency Toolbox”

What is Akka?

Jonas Bonér

- AspectWerkz
- JRockit
- Terracotta
- Typesafe
  - Founded with Martin Oderksy (May 2011)

Named project “Akka” after mountains in Sweden

Viktor Klang

Speaking at Strange Loop 2011 !

Why Akka?

- Scale up
  - Akka actors are lightweight and many thousands can be created on an average system
  - Threads are heavyweight and context switching is relatively slow
- Scale out
  - Actor can be "remoted" without code changes
- Fault-tolerance and error handling

A simple actor

```
import akka.actor.UntypedActor;

public class ToStringActor extends UntypedActor {
    @Override
    public void onReceive(Object message) {
       System.out.println(message.toString());
    }
}
```

A simple actor

```
import akka.actor.{Actor}

class ToStringActor extends Actor {
  override def receive = {
    case message:Any =>
      println(message.toString)
    }
}
```

Nothing prevents this

```
(ns tfd.clojurefun.ClojureActor
  (:gen-class
      :extends akka.actor.UntypedActor
    )
  )

(defn -onReceive [this msg]
   (println (.toString msg))
)
```

Sending messages

```
ActorRef toString =
   Actors.actorOf(ToStringActor.class).start();

toString.sendOneWay(42);
42

toString.sendOneWay(3.14159);
3.14159

toString.sendOneWay(true);
true
```

Sending messages

```
val toString =
     Actor.actorOf(classOf[ToStringActor]).start

toString ! 42 // same as toString.!(42)
42

ToString ! 3.14159
3.14159

toString ! true
true
```

Replying to messages

```
public class DoubleItActor extends UntypedActor {
    @Override
    public void onReceive(Object message) {
        getContext().replyUnsafe(
           message.toString() +
           message.toString());
    }
}
```

Or:

```
        getContext().replySafe(
```

Replying to messages

```
class DoubleItActor extends Actor {
  override def receive = {
    case message:Any =>
      self.reply(
          message.toString + message.toString)
  }
}
```

Or:

```
      self.reply_?
```

Send with reply

```
ActorRef doubleIt =
   Actors.actorOf(DoubleItActor.class).start();

System.out.println(
              doubleIt.sendRequestReply("foo"));
foofoo

System.out.println(
           doubleIt.sendRequestReplyFuture("bar")
                 .get());
barbar
System.out.println(
           doubleIt.sendRequestReplyFuture("bar")
                 .await().result());
Some(barbar)
```

Send with reply

```
val doubleIt =
   Actors.actorOf(classOf[DoubleItActor]).start

println(doubleIt !! "foo")
Some(foofoo)

println((doubleIt !!! "bar").get)
barbar

println((doubleIt !!! "bar").await.result)
Some(barbar)

println((doubleIt !!! "meh")
   .map((x:String) => x.length).await.result)
Some(6)
```

Less painful in Java 8 ?

```
 System.out.println(
   doubleIt.sendRequestReplyFuture("meh")
      .map(new Function<String,Integer>() {
        public Integer apply(String param) {
           return param.length();
        }
 }).await().result());
Some(6)
```

Futures in Akka 1.1

```
val upperIt = actorOf(classOf[ToUpperActor]).start

val doubleFoo = doubleIt !!! "foo"
val upperBar = upperIt !!! "bar"

println(
   (for (
      x:String <- doubleFoo;
      y:String <- upperBar
    ) yield (x + y)).await.result
)
```

```
Some(foofooBAR)
```

What if one way message is replied to?

```
ActorRef toString = actorOf(ToStringActor.class)
                              .start();

doubleIt.sendOneWay("foobar", toString);
```

foobarfoobar

```
val toString = Actors.actorOf(classOf[ToStringActor]).start

(double ! sendOneWay)(toString)
```

foobarfoobar

Anonymous actors

```
val fooActor = Actor.actorOf(
   new Actor {
      def receive = {
         case x => println("Foo: " + x)
      }
   }).start
```

Anonymous actors

Unfortunately this does not work:

```
ActorRef act = Actors.actorOf(new UntypedActor() {

   public void onReceive(Object message) {
     System.out.println("Received : "
                                       + message);
   }
}).start();
```

Anonymous actors

A factory is needed:

```
ActorRef act = actorOf(new UntypedActorFactory() {
   public UntypedActor create() {
     return new UntypedActor() {
        public void onReceive(Object message) {
           System.out.println("Received : "
                                      + message);
        }
     };
   }
}).start();
```

No built-in pattern matching in Java

(Author's idea, not part of Akka)

```
public void onReceive(Object message) {
    Matchable matchable = new Matchable(message);
    for (Integer value :  matchable.matchesInteger(equalTo(42))) {
        System.out.println(value + " is the answer !");
    }
    for (Integer value : matchable.matchesInteger()) {
        System.out.println(value +
                             " is a number, but not the answer.");
    }
    for (Object value : matchable) {
        System.out.println(value + " is not even close.");
    }
}
```

- Matchable is an Iterable that can only be iterated once

- Idea inspired similar idea in Play! framework

- Uses Hamcrest matchers

Scala equivalent

```
def receive = {
    case value:Integer if (value == 42) =>
                     println(value + " is the answer !")
    case value:Integer =>
                      println(value +
                        " is a number, but not the answer.")
     case value:Any => println(value + " is not even close.")
}
```

Typed actors

```
public interface Counter {
    public void increment();

    public Integer getValue();

    public Future<Integer> getFutureValue();
}
```

Typed actors

```
public class CounterActor extends TypedActor
   implements Counter {
    private Integer value = 0;

    public void increment() {
        try { Thread.sleep(1000); }
                       catch (InterruptedException ex) { }
        value++;
    }

    public Integer getValue() {
        return value;
    }

    public Future<Integer> getFutureValue() {
        return future(value);
    }
}
```

Typed actors

```
Counter counter = TypedActor
   .newInstance(Counter.class,
                   CounterActor.class, 2000);

counter.increment();

System.out.println(counter.getValue());
1

counter.increment();

System.out.println(
           counter.getFutureValue().get());
2
```

Server managed remote actors

On Remote Server:

```
RemoteServerModule server =
                remote().start("localhost", 2553);

server.register("double-it",
                     actorOf(DoubleItActor.class));
server.register("to-string",
                     actorOf(ToStringActor.class));
```

On Client:

```
ActorRef doubleIt = remote()
         .actorFor("double-it", "localhost", 2553);
ActorRef toString = remote()
         .actorFor("to-string", "localhost", 2553);

doubleIt.sendOneWay("meh", toString);
```

On Remote Server:

```
mehmeh
```

Client managed remote actors

```
ActorRef remoteToString = remote().actorOf(
      ToStringActor.class, "localhost", 2553).start();

ActorRef localDoubleIt =
          Actors.actorOf(DoubleItActor.class).start();

remoteToString.sendOneWay("foo");

localDoubleIt.sendOneWay("bar", remoteToString);
```

On Remote Server:

```
foo
barbar
```

Deprecated as of Akka 1.1

Event based dispatching

- Default dispatcher for Akka
  - `Dispatchers.globalExecutorBasedEventDrivenDispatcher`
- Many actors per thread
- Actors should not block
- Akka can spawn extra threads if needed
  - Up to a max number

Priority event based dispatcher

```
val act = actorOf(new Actor {
  def receive = { case x => println("Received : " + x)
}})

act.dispatcher =
    new PriorityExecutorBasedEventDrivenDispatcher("foo",
PriorityGenerator {
      case _:String => 0
      case x:Int => x
      case _ => 50
})

  act.start.dispatcher.suspend(act)

  act ! 1.0
  act ! "foo"
  (0 to 9).map { x:Int => act ! (x * 10) }
  act ! 2.0
  act ! "bar"
```

Priority event based dispatcher

```
Received : foo
Received : 0
Received : bar
Received : 10
Received : 20
Received : 30
Received : 40
Received : 1.0
Received : 50
Received : 2.0
Received : 60
Received : 70
Received : 80
Received : 90
```

Work stealing event dispatcher

```
public class FooActor extends UntypedActor {
    public static MessageDispatcher dispatcher = Dispatchers
    .newExecutorBasedEventDrivenWorkStealingDispatcher("foobar", 5)
            .build();

    private static int currentId = 0;

    private final int instanceId;

    private int count = 0;

    public FooActor() {
        getContext().setDispatcher(dispatcher);
        instanceId = currentId++;
    }
```

Work stealing event dispatcher

```
@Override
public void onReceive(Object message) {
   System.out.printf(
      "Foo %d processed : %s (count = %d) on Thread : %s\n",
           InstanceId,
           message.toString(),
           ++count,
           Thread.currentThread().getName()
   );
    try { Thread.sleep(instanceId * 50 + 50); }
                       catch (InterruptedException ex) {  }
}
```

```
Foo 0 processed : 0 (count = 1) on Thread : akka:event-
driven:dispatcher:foobar-1
Foo 1 processed : 1 (count = 1) on Thread : akka:event-
driven:dispatcher:foobar-2
Foo 0 processed : 2 (count = 2) on Thread : akka:event-
driven:dispatcher:foobar-3
Foo 1 processed : 3 (count = 2) on Thread : akka:event-
driven:dispatcher:foobar-4
Foo 0 processed : 4 (count = 3) on Thread : akka:event-
driven:dispatcher:foobar-5
Foo 1 processed : 5 (count = 3) on Thread : akka:event-
driven:dispatcher:foobar-6
Foo 0 processed : 6 (count = 4) on Thread : akka:event-
driven:dispatcher:foobar-7
Foo 0 processed : 7 (count = 5) on Thread : akka:event-
driven:dispatcher:foobar-8
Foo 1 processed : 8 (count = 4) on Thread : akka:event-
driven:dispatcher:foobar-9
Foo 0 processed : 9 (count = 6) on Thread : akka:event-
driven:dispatcher:foobar-10
```

```
Foo 1 processed : 93 (count = 40) on Thread : akka:event-driven:dispatcher:foobar-15
Foo 0 processed : 94 (count = 55) on Thread : akka:event-driven:dispatcher:foobar-13
Foo 0 processed : 95 (count = 56) on Thread : akka:event-driven:dispatcher:foobar-13
Foo 0 processed : 96 (count = 57) on Thread : akka:event-driven:dispatcher:foobar-16
Foo 1 processed : 97 (count = 41) on Thread : akka:event-driven:dispatcher:foobar-14
Foo 0 processed : 98 (count = 58) on Thread : akka:event-driven:dispatcher:foobar-4
Foo 1 processed : 99 (count = 42) on Thread : akka:event-driven:dispatcher:foobar-4
```

Thread based dispatcher

```
getContext().setDispatcher(

Dispatchers.newThreadBasedDispatcher(getContext()));.

self.dispatcher =
            Dispatchers.newThreadBasedDispatcher(self)
```

- Worse scalability and performance
- Good for “daemon” actors
    - Low frequency messages
    - Blocking

Routing and load balancing

```
InfiniteIterator<ActorRef> iter =
   new CyclicIterator<ActorRef>(Arrays.asList(
            actorOf(FooActor.class).start(),
            actorOf(BarActor.class).start()
   ));

for (Integer value:Arrays.asList(4, 8, 15, 16, 23, 42))
{
   iter.next().sendOneWay(value);
}
```

```
Bar: 8
Foo: 4
Bar: 16
Foo: 15
Bar: 42
Foo: 23
```

Routing and load balancing

```
ActorRef router = actorOf(new UntypedActorFactory
    public UntypedActor create() {
       return new UntypedLoadBalancer() {
         private InfiniteIterator iterator =
            new CyclicIterator<ActorRef>(Arrays.asList(
               actorOf(FooActor.class).start(),
               actorOf(BarActor.class).start()));

         public InfiniteIterator<ActorRef> seq() {
            return iterator;
         }
      };
}}).start();

for (Integer value :
               Arrays.asList(4, 8, 15, 16, 23, 42)) {
   router.sendOneWay(value);
}
```

Routing and load balancing

```
class SleepyCounterActor(name:String, sleepyTime:Int)
  extends Actor
{
  var count = 0

  def receive = {
    case x => {
      Thread.sleep(sleepyTime)
      count += 1;
      printf("%s received '%s' count = %d\n",
                                     name, x, count) }
}
```

Routing and load balancing

```
val iter = new
  SmallestMailboxFirstIterator(List(
    actorOf(new SleepyCounterActor("Foo",  1)).start,
    actorOf(new SleepyCounterActor("Bar", 50)).start))

(1 to 15).foreach { x:Int =>
   Thread.sleep(2)
   iter.next ! x
}
```

Routing and load balancing

```
val balancer = loadBalancerActor(new
      SmallestMailboxFirstIterator(List(
        actorOf(new SleepyCounterActor("Foo", 1)).start,
        actorOf(new SleepyCounterActor("Bar", 50)).start)
      )
    )

(1 to 15).foreach { x:Int =>
   Thread.sleep(2)
   balancer ! x
}
```

Routing and Load Balancing

```
Foo received '3' count = 1
Foo received '4' count = 2
Foo received '6' count = 3
Foo received '8' count = 4
Foo received '9' count = 5
Foo received '10' count = 6
Foo received '11' count = 7
Foo received '12' count = 8
Foo received '13' count = 9
Foo received '14' count = 10
Foo received '15' count = 11
Bar received '1' count = 1
Bar received '2' count = 2
Bar received '5' count = 3
Bar received '7' count = 4
```

Supervision and fault tolerance

```
class DivideInto10Actor extends Actor {
  def receive = { case x:Int =>  self.reply_?(10 / x) }

  override def preStart {
    print("DivideInto10Actor.preStart()\n")
  }

  override def postStop {
    print("DivideInto10Actor.postStop()\n")
  }

  override def preRestart(reason: Throwable) {
    printf("DivideInto10Actor.preRestart(%s)\n", reason.getMessage)
  }

  override def postRestart(reason: Throwable) {
    printf("DivideInto10Actor.postRestart(%s)\n", reason.getMessage)
  }
}
```

Supervision and fault tolerance

```
val supervisedActors = Array(
    actorOf(classOf[DivideInto10Actor]),
    actorOf(classOf[DivideInto10Actor])
)

val supervisor =  Supervisor(
    SupervisorConfig(
      OneForOneStrategy(List(classOf[Exception]), 3, 1000),
        List(
          Supervise(supervisedActors(0), Permanent),
          Supervise(supervisedActors(1), Temporary)
)))

DivideInto10Actor.preStart()
DivideInto10Actor.preStart()
```

Supervision and fault tolerance

```
supervisedActors(0) ! 0 // Permanent

DivideInto10Actor.postStop()
java.lang.ArithmeticException: / by zero
...
DivideInto10Actor.preRestart(/ by zero)
DivideInto10Actor.preStart()
DivideInto10Actor.postRestart(/ by zero)

supervisedActors(1) ! 0 // Temporary

DivideInto10Actor.postStop()
java.lang.ArithmeticException: / by zero
...
DivideInto10Actor.preRestart(/ by zero)
DivideInto10Actor.preStart()
DivideInto10Actor.postRestart(/ by zero)
```

Supervision and fault tolerance

With OneForOneStrategy:

```
SupervisedActors(0) ! 0 // Permanent

DivideInto10Actor.preRestart(/ by zero)
java.lang.ArithmeticException: / by zero
...
DivideInto10Actor.preStart()
DivideInto10Actor.postRestart(/ by zero)


SupervisedActors(1) ! 0 // Temporary

DivideInto10Actor.postStop()
java.lang.ArithmeticException: / by zero
```

Supervision and fault tolerance

```
val supervisor =  Supervisor(
    SupervisorConfig(
      OneForOneStrategy(List(classOf[Exception]), 3, 1000),
        Nil
  ))

  supervisedActors(0).setLifeCycle(Permanent)
  supervisedActors(0).start
  supervisor.link(supervisedActors(0))

  supervisedActors(1).setLifeCycle(Temporary)
  supervisedActors(1).start
  supervisor.link(supervisedActors(1))
```

Supervision and fault tolerance

```
val supervisor =  Supervisor(
    SupervisorConfig(
      OneForOneStrategy(List(classOf[Exception]), 3, 1000),
        Nil
  ))

  supervisedActors(0).setLifeCycle(Permanent)
  supervisor.startLink(supervisedActors(0))

  supervisedActors(1).setLifeCycle(Temporary)
  supervisor.startLink(supervisedActors(1))
```

Supervision and fault tolerance

```
class CustomSupervisor extends Actor {
  self.faultHandler = OneForOneStrategy(
                List(classOf[Throwable]), 5, 5000)

  def receive = {
    case x => println("CustomSupervisor received : " + x)
  }
}


val supervisor = actorOf(classOf[CustomSupervisor]).start

supervisedActors(0).setLifeCycle(Permanent)
supervisor.startLink(supervisedActors(0))

supervisedActors(1).setLifeCycle(Temporary)
supervisor.startLink(supervisedActors(1))
```

Scheduler

```
Scheduler.schedule(
     receiverActor,
     messageToBeSent,
     initialDelayBeforeSending,
     delayBetweenMessages,
     timeUnit)

Scheduler.scheduleOne(
     receiverActor,
     messageToBeSent,
     delayUntilSend,
     timeUnit)
```

Scheduler

```
Scheduler.schedule(
                toString,
                "ping",
                 0,
                 5,
                 TimeUnit.SECONDS
);

Scheduler.scheduleOnce(
                toString,
                "pong",
                15,
                TimeUnit.SECONDS
);
```

Declarative configuration

akka.conf:

```
akka {
  version = "1.1"
  enabled-modules = ["camel", "http"]

  time-unit = "seconds"

  event-handlers = ["akka.event.EventHandler$DefaultListener"]
  event-handler-level = "INFO"
...
actor {
  timeout = 5
  serialize-messages = off
  throughput = 5
  ...
```

Declarative configuration

```
    default-dispatcher {
      type = "GlobalExecutorBasedEventDriven"
      keep-alive-time = 60
      core-pool-size-factor = 1.0
      max-pool-size-factor  = 4.0
      mailbox-capacity = -1
      rejection-policy = "caller-runs"
      throughput = 5
...
```

Data Flow concurrency

```
val x, y, z = Promise[Int]

   flow {
      z << x() + y()
      println("z = " + z())
   }

   flow { x << 40 }

   flow { y << 2 }
}
```

```
z = 42
```

Back to (the) Future

```
val future = Future {
   Thread.sleep(1000)
   42
 }

 val doubleItActor = actorOf(new Actor {
     def receive = {
       case x:Int => Thread.sleep(1000); self.reply(x * 2)
       case _ => self.reply(0)
     }
   }
 ).start

 val actorReply:Future[Int] =
               doubleItActor !!! 73

 val promise = Promise[Int]
```

Back to (the) Future

```
flow { promise << 23 }

val totalFuture = for {
    x <- future
    y <- actorReply
    c <- promise
} yield ( x + y + c )

println(totalFuture.get)
```

```
211
```

Back to (the) Future

```
flow {
   println (future() + promise() + actorReply())
}
```

```
211
```

Back to (the) Future

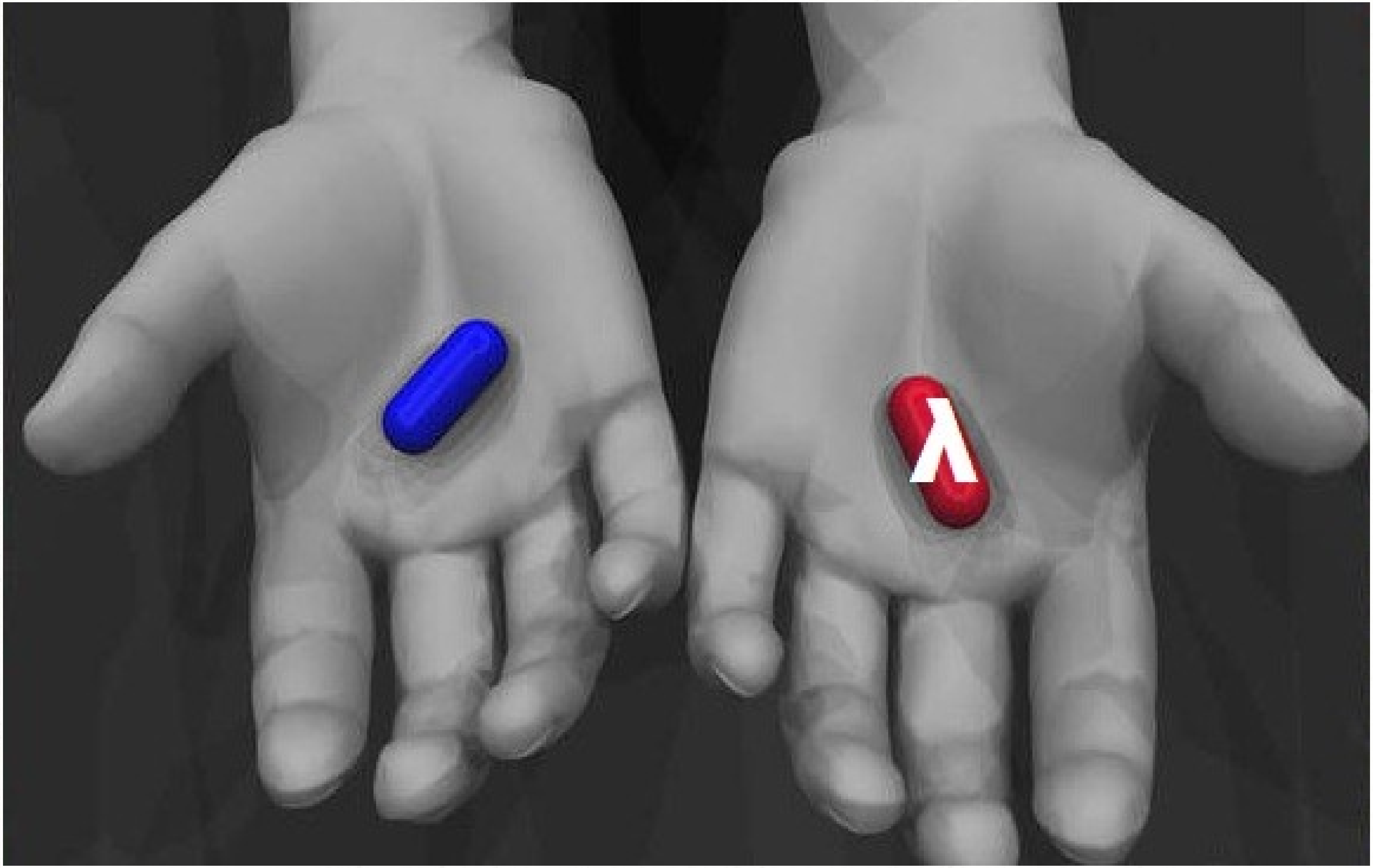
```
val resultList = Future.sequence(
           List(future, actorReply, promise)).get

println (resultList)

List(42, 146, 23)

println (resultList.reduce(_ + _))

211
```


λ

Map-Reduce (like) with Akka (finding anagrams)

Stateful behaviors

```
val doubleIt = actorOf(classOf[DoubleItActor])
                                        .start
println(doubleIt !! "foo")
Some(foofoo)

doubleIt ! HotSwap ( self => {
   case message => self.reply(
   message.toString + message.toString + " hotswapped")
})

println(doubleIt !! "bar")
Some(barbar hotswapped)

doubleIt ! RevertHotSwap

println(doubleIt !! "meh")
Some(mehmeh)
```

Stateful behaviors

```
class FooActor extends UntypedActor {

   public void onReceive(Object message) {
      if ("foo".equals(message)) {
         become(BAR_BEHAVIOR);
      } else {
         getContext().replySafe("foo");
      }
}
```

Stateful behaviors

```
private final Procedure<Object> BAR_BEHAVIOR = new
   Procedure<Object>()  {
      @Override
      public void apply(Object param) {
         getContext().replySafe("bar");
         unbecome();
      }
};
```

Stateful behaviors

```
System.out.println(actorRef.sendRequestReply("bar"));

actorRef.sendOneWay("foo");

System.out.println(actorRef.sendRequestReply("bar"));


System.out.println(actorRef.sendRequestReply("bar"));
```

```
foo
bar
foo
```

Finite state machine in an Actor (vending machine)

Software Transactional Memory: Refs

- Popularized by Clojure on the JVM
- A form on optimistic locking for memory
- ACI (Atomic, Consistent, Isolated)
  - Not Durable
- Built on Multiverse (http://multiverse.codehaus.org)

Refs

```
val ref1 = new Ref(4)
val ref2 = new Ref(4)

println("ref1 = " + ref1.get)
println("ref2 = " + ref2.get)

try {
  atomic {
    ref1.swap(12)
    ref2.alter(_ * 2)
    ref1.alter(_ / 0)
  }
} catch { case ex:Exception => println(ex.getMessage) }

println("ref1 = " + ref1.get)
println("ref2 = " + ref2.get)
```

```
ref1 = 4
ref2 = 4
/ by zero
ref1 = 4
ref2 = 4
```

Refs

```
val ref1 = new Ref(4)
val ref2 = new Ref(4)

println("ref1 = " + ref1.get)
println("ref2 = " + ref2.get)

try {
    ref1.swap(12)
    ref2.alter(_ * 2)
    ref1.alter(_ / 0)
} catch { case ex:Exception => println(ex.getMessage) }

println("ref1 = " + ref1.get)
println("ref2 = " + ref2.get)
```

```
ref1 = 4
ref2 = 4
/ by zero
ref1 = 12
ref2 = 8
```

Refs

```
final Ref<Integer> ref1 = new Ref(4);
final Ref<Integer> ref2 = new Ref(4);

new Atomic<Integer>() {
    public Integer atomically() {
        ref1.set(8);
        ref2.set(12);
        return (Integer) ref1.get() +
                (Integer) ref2.get();
    }
}.execute());
```

Transactional datastructues

- Akka provides two data structures supporting STM
  - TransactionalMap
  - TransactionalVector
- Both are “persistent” and work with atomic blocks

Transactional datastructues

```
val tranMap = TransactionalMap[String, Int]("Foo" -> 23)
val tranVector = TransactionalVector(2.0, 4.0, 6.0)
try {
    atomic {
        tranMap += ("Bar" -> 42)
        tranVector.add(8.0)

        println(tranMap)
        println(tranVector)
        1 / 0
      }
} catch { case ex:Any => println(ex) }
    atomic { println(tranMap); println(tranVector) }
Map(Bar -> 42, Foo -> 23)
TransactionalVector(2.0, 4.0, 6.0, 8.0)
java.lang.ArithmeticException: / by zero
Map(Foo -> 23)
TransactionalVector(2.0, 4.0, 6.0)
```

Dining Philosophers with STM

```
Philosopher 3 wants to ponder
Philosopher 4 sees forks available
Philosopher 1 sees forks available
Philosopher 4 eats
Philosopher 1 eats
Philosopher 2 sees forks available
Philosopher 2 can't get forks
Philosopher 2 can't get forks
Philosopher 5 can't get forks
```

Agents

```
val agent = new Agent(5)

val times2 = { x:Int => x * 2 }

agent.send(times2)
agent.send(_ - 1)
agent.send(_ * 3)
println(agent.get)
5

Thread.sleep(1000)
println(agent.get)
27
```

Agents

```
Agent<Integer> agent = new Agent<Integer>(5);

AbstractFunction1<Integer, Integer> times2 =
     new AbstractFunction1<Integer, Integer>() {
         public final Integer apply(Integer x) {
                return x * 2;
         }
     };
agent.send(times2);
agent.send(times2);
5
System.out.println(agent.get());
try { Thread.sleep(1000); }
              catch (InterruptedException iex) { }
System.out.println(agent.get());
20
```

Coordinated transactions

```
class CoordinatedAccountActor extends Actor {
  private val balance:Ref[Int] = Ref(0)

  def doReceive:Receive = {
    case Withdraw(x) if x > balance.get =>
                            throw new Exception("Overdrawn")
    case Withdraw(x) if x > 0           =>
                            balance.alter(_ - x)
    case Deposit(x)  if x > 0           =>
                            balance.alter(_ + x)
    case PrintBalance                   =>
                            println("Balance = " + balance.get)
  }

  def receive = {
    case coordinated @ Coordinated(action) => coordinated atomic {
      doReceive(action)
    }
    case other:Any => doReceive(other)
  }
}
```

Coordinated Transactions

```
account1 ! Deposit(100)
account2 ! Deposit(50)

val coordinated = Coordinated()

try {
   coordinated atomic {
     account1 ! coordinated.coordinate(Deposit(75))
     account2 ! coordinated.coordinate(Withdraw(75))
   }
} catch {
  case ex:Exception => println(ex.getMessage)
}
org.multiverse.templates.InvisibleCheckedException:
java.lang.Exception: Overdrawn

account1 ! PrintBalance
Balance = 100
account2 ! PrintBalance
Balance = 50
```

Transactors

```
class AccountTransactor extends Transactor {
  private val balance:Ref[Int] = Ref(0)

  override def atomically = {
    case Withdraw(x) if x > balance.get =>
                       throw new Exception("Overdrawn")
    case Withdraw(x) if x > 0           =>
                       balance.alter(_ - x)
    case Deposit(x)  if x > 0           =>
                       balance.alter(_ + x)
    case PrintBalance                   =>
                       println("Balance = " + balance.get)
  }
}
```

Coordinated typed actors

```
public interface BankAccount {
    @Coordinated public void withdraw(int amount);
    @Coordinated public void deposit(int amount);
    public int getBalance();
}
```

Coordinated typed actors

```
public class BankAccountTypedActor extends TypedActor
   implements BankAccount {
    private Ref<Integer> balance = new Ref<Integer>(0);

    public void withdraw(int amount) {
        if (amount > 0) {
            if (amount > balance.get()) {
                throw new RuntimeException("Overdrawn");
            } else {
                balance.set(balance.get() - amount);
            }
        }
    }

    public void deposit(int amount) {
        if (amount > 0) { balance.set(balance.get() + amount); }
    }

    public int getBalance() {
        return balance.get();
    }
}
```

Coordinated typed actors

```
public static void transferUnsafe(
    BankAccount fromAccount,
    BankAccount toAccount,
    int amount)
{
    toAccount.deposit(amount);
    fromAccount.withdraw(amount);
}

public static void transferSafe(
    final BankAccount fromAccount,
    final BankAccount toAccount,
    final int amount)
{
    Coordination.coordinate(true, new Atomically() {
        public void atomically() {
            toAccount.deposit(amount);
            fromAccount.withdraw(amount);
        }
    });
}
```

Coordinated typed actors

```
BankAccount account1 = (BankAccount) TypedActor
 .newInstance(BankAccount.class, BankAccountTypedActor.class
BankAccount account2 = (BankAccount) TypedActor
 .newInstance(BankAccount.class, BankAccountTypedActor.class, 1000);

account1.deposit(100);
account2.deposit(50);

try {
   transferSafe(account1, account2, 150);
} catch (Exception ex) {
    System.err.println(ex.getMessage());
}

System.out.println(account1.getBalance());
100
System.out.println(account2.getBalance());
50
java.lang.RuntimeException: Overdrawn
```

Coordinated typed actors

```
BankAccount account1 = (BankAccount) TypedActor
 .newInstance(BankAccount.class, BankAccountTypedActor.class,
BankAccount account2 = (BankAccount) TypedActor
 .newInstance(BankAccount.class, BankAccountTypedActor.class, 1000);

account1.deposit(100);
account2.deposit(50);

try {
    transferUnsafe(account1, account2, 150);
} catch (Exception ex) {
    System.err.println(ex.getMessage());
}

System.out.println(account1.getBalance());
100
System.out.println(account2.getBalance());
200
java.lang.RuntimeException: Overdrawn
```

Integration with Spring

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:akka="http://akka.io/schema/akka"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-
beans-3.0.xsd
       http://akka.io/schema/akka
       http://akka.io/akka-1.1.xsd">

    <akka:untyped-actor id="toStringActor"
          implementation="tfd.akkatest.java.ToStringActor"
          scope="singleton"
          autostart="true"/>
</beans>
```

Spring supports:

- Typed Actors
- Remote Actors
- Dispatchers
- Supervisors
- Camel endpoints

Other Modules

- Camel
- HTTP
  - JAX-RS (JSR-311)
  - Expose Actors as REST services
- AMQP
- Guice
- And more !

Next for Akka

- Akka 2.0
  - Clustering support
- Cloudy Akka (Atmos)
  - Commercial product
  - Monitoring and Management

    JMX - SNMP
  - Provisioning

    Remote classloading

More Information

- Akka Home (http://akka.io/)
- Typesafe (http://typesafe.com/)
- It's Actors All The Way Down (http://www.dalnefre.com/wp/)
- Pure Danger Tech - Actor Concurrency (http://tech.puredanger.com/presentations/actor-concurrency/)

Thank You !

Questions