Akka:
Simpler Concurrency, Scalability & Fault-tolerance through Actors

Jonas Bonér
Scalable Solutions
jonas@jonasboner.com
twitter: @jboner
The problem

It is way too hard to build:

1. correct highly concurrent systems
2. truly scalable systems
3. fault-tolerant systems that self-heals

...using “state-of-the-art” tools
Vision

Simpler

- Concurrency
- Scalability
- Fault-tolerance

Through one single unified
- Programming model
- Runtime service
Manage system overload
Scale up & Scale out
Replicate and distribute for fault-tolerance
Automatic & adaptive load balancing
Introducing akka

Concurrency  Scalability  Fault-tolerance

Actors  STM  Agents  Dataflow

Distributed  Secure  Persistent  Open Source  Clustered
Architecture

Core Modules

Fault-tolerance
- Local Actor Supervision
- Remote Actor Supervision

Scalability
- Client Managed Remote Actors
- Server Managed Remote Actors
- Cluster Membership

Concurrency
- Actors
- STM
- Agents
- Dataflow
Architecture

Enterprise Modules

Microkernel
PubSub
REST
Comet
Security
JTA
Spring integration
Guice integration
Camel integration
AMQP integration

Fault-tolerance
Local Actor Supervision
Remote Actor Supervision

Scalability
Client Managed Remote Actors
Server Managed Remote Actors
Cluster Membership

Concurrency
Actors
STM
Agents
Dataflow

Persistence
Cassandra backend
MongoDB backend
Redis backend

Monitoring
Management
Dashboard
JMX
SNMP
Clustering
Provisioning
(OSGi-based)
Actors
one tool in the toolbox
Akka Actors
Actor Model of Concurrency

- Implements Message-Passing Concurrency
- Share NOTHING
- Isolated lightweight processes
- Communicates through messages
- Asynchronous and non-blocking
- Each actor has a mailbox (message queue)
Actor Model of Concurrency

• Easier to reason about
• Raised abstraction level
• Easier to avoid
  – Race conditions
  – Deadlocks
  – Starvation
  – Live locks
Two different models

• **Thread**-based
• **Event**-based
  – *Very* lightweight (600 bytes per actor)
  – Can easily create **millions** on a single workstation (13 million on 8 G RAM)
  – Does not consume a thread
case object Tick

class Counter extends Actor {
  private var counter = 0

  def receive = {
    case Tick =>
      counter += 1
      println(counter)
  }
}

Actors
Create Actors

import Actor._

val counter = actorOf[Counter]

counter is an ActorRef
val actor = actorOf(new MyActor(..))

create actor with constructor arguments
Start actors

val counter = actorOf[Counter]
counter.start
Start actors

```scala
val counter = actorOf[Counter].start
```
val counter = actorOf[Counter].start
counter.stop
class MyActor extends Actor {

  override def preStart = {
    ...
    // called after ‘start’
  }

  override def postStop = {
    ...
    // called before ‘stop’
  }
}

init & shutdown callbacks
class RecursiveActor extends Actor {
  private var counter = 0
  self.id = "service:recursive"

  def receive = {
    case Tick =>
      counter += 1
      self ! Tick
  }
}

the self reference
val worker = actor {
    case Work(fn) => fn()
}
Send: !

counter ! Tick

fire-forget
Send: !

counter.sendOneWay(Tick)

fire-forget
Send: !!

val result = (actor !! Message).as[String]

uses Future under the hood (with time-out)
Send: !!

```scala
val result = counter.sendRequestReply(Tick)
```

uses Future under the hood (with time-out)
Send: !!!

```scala
// returns a future
val future = actor !!! Message
future.await
val result = future.get

...
Futures.awaitOne(List(fut1, fut2, ...))
Futures.awaitAll(List(fut1, fut2, ...))
```

returns the Future directly
val result = counter.sendRequestReplyFuture(Tick)

returns the Future directly
class SomeActor extends Actor {
    def receive = {
        case User(name) =>
            // use reply
            self.reply("Hi " + name)
    }
}
class SomeActor extends Actor {
  def receive = {
    case User(name) =>
      // store away the sender
      // to use later or
      // somewhere else
      ... = self.sender
  }
}
class SomeActor extends Actor {
  def receive = {
    case User(name) =>
      // store away the sender future
      // to resolve later or
      // somewhere else
      ... = self.senderFuture
  }
}
// define the case class
case class Register(user: User)

// create and send a new case class message
actor ! Register(user)

// tuples
actor ! (username, password)

// lists
actor ! List("bill", "bob", "alice")
val actors = ActorRegistry.actors
val actors = ActorRegistry.actorsFor(TYPE)
val actors = ActorRegistry.actorsFor(id)
val actor = ActorRegistry.actorFor(uuid)
ActorRegistry.foreach(fn)
ActorRegistry.shutdownAll
TypedActor
class CounterImpl extends TypedActor with Counter {

    private var counter = 0;

    def count = {
        counter += 1
        println(counter)
    }
}
Create Typed Actor

val counter = TypedActor.newInstance(classOf[Counter], classof[CounterImpl])
Send message

counter.count

fire-forget
val hits = counter.getNrOfHits

uses Future under the hood (with time-out)
class PingImpl extends TypedActor
  with Ping {
  def hit(count: Int) = {
    val pong = getContext
      .getSender.asInstanceOf[Pong]
    pong.hit(count += 1)
  }
}
akka {
    version = "0.10"
    time-unit = "seconds"
    actor {
        timeout = 5
        throughput = 5
    }
}
Scalability Benchmark
Simple Trading system

- Synchronous Scala version
- Scala Library Actors 2.8.0
  - Fire-forget
  - Request-reply (Futures)
- Akka
  - Fire-forget (Hawt dispatcher)
  - Fire-forget (default dispatcher)
  - Request-reply

Run it yourself:
http://github.com/patriknw/akka-sample-trading
Agents

yet another tool in the toolbox
val agent = Agent(5)

// send function asynchronously
agent send (_ + 1)

val result = agent() // deref
...
...
// use result

agent.close

Agents

Cooperates with STM
Akka Dispatchers
Dispatchers

- Executor-based Dispatcher
- Executor-based Work-stealing Dispatcher
- Hawt Dispatcher
- Thread-based Dispatcher
object Dispatchers {
    object globalHawtDispatcher extends HawtDispatcher
    ...

    def newExecutorBasedEventDrivenDispatcher(name: String)
    def newExecutorBasedEventDrivenWorkStealingDispatcher(name: String)
    def newHawtDispatcher(aggregate: Boolean)
    ...
}

Dispatchers
Set dispatcher

class MyActor extends Actor {
    self.dispatcher = Dispatchers
        .newThreadBasedDispatcher(self)
    ...
}

actordispatcher = dispatcher // before started
Let it crash
fault-tolerance
Influenced by Erlang
9
nines
OneForOne
fault handling strategy
OneForOne
fault handling strategy
OneForOne

fault handling strategy
OneForOne
fault handling strategy
OneForOne
fault handling strategy
OneForOne
fault handling strategy
OneForOne
fault handling strategy
OneForOne
fault handling strategy
AllForOne
fault handling strategy
AllForOne
fault handling strategy
AllForOne
fault handling strategy
AllForOne
fault handling strategy
AllForOne
fault handling strategy
Supervisor hierarchies
Supervisor hierarchies
Supervisor hierarchies
Supervisor hierarchies
Supervisor hierarchies
Supervisor hierarchies
Fault handlers

\textbf{AllForOne}Strategy(\textit{maxNrOfRetries}, within\textit{TimeRange})

\textbf{OneForOne}Strategy(\textit{maxNrOfRetries}, within\textit{TimeRange})
Linking

`link(actor)`  
`unlink(actor)`  

`startLink(actor)`  
`spawnLink[MyActor]`
trapExit = List(
    classOf[ServiceException],
    classOf[PersistenceException])
class Supervisor extends Actor {
    trapExit = List(classOf[Throwable])
    faultHandler =
        Some(OneForOneStrategy(5, 5000))

    def receive = {
        case Register(actor) =>
            link(actor)
    }
}
class FaultTolerantService extends Actor {
    ...
    override def preRestart(reason: Throwable) = {
        ...
        // clean up before restart
    }
    override def postRestart(reason: Throwable) = {
        ...
        // init after restart
    }
}
Remote Actors
// use host & port in config
RemoteNode.start

RemoteNode.start("localhost", 9999)

Scalable implementation based on NIO (Netty) & Protobuf
Two types of remote actors

**Client managed**

**Server managed**
Client-managed supervision works across nodes

// methods in Actor class

spawnRemote[MyActor](host, port)

spawnLinkRemote[MyActor](host, port)

startLinkRemote(actor, host, port)
Client-managed moves actor to server client manages through proxy

```scala
def spawnLinkRemote[MyActor](actorName: String, actorId: Int, actorKey: Int) = async { 
  spawnRemote(MyActor((), actorName, actorId, actorKey)) 
}

val actorProxy = spawnLinkRemote[MyActor](
  "darkstar",
  9999)
actorProxy ! message
```
Server-managed
register and manage actor on server
client gets “dumb” proxy handle

```java
RemoteNode.register("service:id", actorOf[MyService])
```

server part
val handle = RemoteClient.actorFor(“service:id”, “darkstar”, 9999)

handle ! message

client part
Cluster Membership

Cluster.\texttt{relayMessage}(
  \texttt{classOf[TypeOfActor], message})

\textbf{for (endpoint <- Cluster)}
  \texttt{spawnRemote[TypeOfActor] (}
    \texttt{endpoint.host, endpoint.port})
STM
yet another tool in the toolbox
What is STM?
STM: overview

• See the memory (heap and stack) as a transactional dataset
• Similar to a database
  • begin
  • commit
  • abort/rollback
• Transactions are retrieved automatically upon collision
• Rolls back the memory on abort
Managed References

- Separates **Identity** from **Value**
  - Values are **immutable**
  - **Identity** (Ref) holds **Values**
- Change is a function
- Compare-and-swap (CAS)
- Abstraction of time
- Must be used **within a transaction**
import se.scalablesolutions.akka.stm.local._

atomic {
  ...
  ...
  atomic {
    ... // transactions compose!!!
  }
}
Managed References

Typical OO: direct access to mutable objects

Managed Reference: separates Identity & Value

Copyright Rich Hickey 2009
Managed References

• Separates Identity from Value
  - Values are immutable
  - Identity (Ref) holds Values
• Change is a function
• Compare-and-swap (CAS)
• Abstraction of time
• Must be used within a transaction
import se.scalablesolutions.akka.stm.local._

// giving an initial value
val ref = Ref(0)

// specifying a type but no initial value
val ref = Ref[Int]
Managed References

```python
val ref = Ref(0)

atomic {
    ref.set(5)
}
// -> 0

atomic {
    ref.get
}
// -> 5
```
Managed References

```scala
val ref = Ref(0)

atomic {
    ref alter (_, + 5)
}
// -> 5

val inc = (i: Int) => i + 1

atomic {
    ref alter inc
}
// -> 6
```
val ref = Ref(1)
val anotherRef = Ref(3)

atomic {
  for {
    value1 <- ref
    value2 <- anotherRef
  } yield (value1 + value2)
}
// -> Ref(4)

val emptyRef = Ref[Int]

atomic {
  for {
    value1 <- ref
    value2 <- emptyRef
  } yield (value1 + value2)
}
// -> Ref[Int]
Transactional datastructures

// using initial values
val map = TransactionalMap("bill" -> User("bill"))
val vector = TransactionalVector(Address("somewhere"))

// specifying types
val map = TransactionalMap[String, User]
val vector = TransactionalVector[Address]
atomic {
    deferred {
        // executes when transaction commits
    }
    compensating {
        // executes when transaction aborts
    }
}
Actors + STM = Transactors
class UserRegistry extends Transactor {

    private lazy val storage = TransactionalMap[String, User]()

    def receive = {
        case NewUser(user) =>
            storage + (user.name -> user)
        ...
    }
}
Transactors
Transactors

Start transaction
Transactors

Start transaction

Send message
Transactors

Start transaction

Send message
Transactors

- Start transaction
- Send message
- Update state within transaction
Transactors

Start transaction
Send message
Update state within transaction
Transactors

Start transaction

Send message

Update state within transaction
Transactors

- Start transaction
- Send message
- Update state within transaction
Transactors

- Start transaction
- Send message
- Update state within transaction
Transactors

Start transaction

Send message

Update state within transaction
Transactors

Start transaction

Send message

Update state within transaction

Transaction fails
**Transactors**

- Start transaction
- Send message
- Update state within transaction
- Transaction fails
Transactors
Transactors
Transactors
Transactors
Transactors
Transactors
Transactors

Transaction automatically retried
class Transferer extends Actor {
    implicit val txFactory = TransactionFactory(
        blockingAllowed = true, trackReads = true, timeout = 60 seconds)

    def receive = {
        case Transfer(from, to, amount) =>
            atomic {
                if (from.get < amount) {
                    log.info("not enough money - retrying")
                    retry
                }
                log.info("transferring")
                from.alter(_ - amount)
                to.alter(_ + amount)
            }
    }
}
atomic {
  either {
    if (left.get < amount) {
      log.info("not enough on left - retrying")
      retry
    }
    log.info("going left")
  } orElse {
    if (right.get < amount) {
      log.info("not enough on right - retrying")
      retry
    }
    log.info("going right")
  }
}
STM: config

```scala
akka {
  stm {
    max-retries = 1000
    timeout = 10
    write-skew = true
    blocking-allowed = false
    interruptible = false
    speculative = true
    quick-release = true
    propagation = requires
    trace-level = none
    hooks = true
    jta-aware = off
  }
}
```
Modules
Akka Persistence
STM gives us

Atomic
Consistent
Isolated
Persistence module turns STM into

Atomic, Consistent, Isolated, Durable
Akka Persistence API

```scala
// transactional Cassandra-backed Map
val map = CassandraStorage.newMap

// transactional Redis-backed Vector
val vector = RedisStorage.newVector

// transactional Mongo-backed Ref
val ref = MongoStorage.newRef
```
For Redis only (so far)

```scala
val queue: PersistentQueue[ElementType] = RedisStorage.newQueue

val set: PersistentSortedSet[ElementType] = RedisStorage.newSortedSet
```
Akka Spring
Spring integration

```xml
<beans>
  <akka:typed-actor
    id="myActiveObject"
    interface="com.biz.MyPOJO"
    implementation="com.biz.MyPOJO"
    transactional="true"
    timeout="1000" />
...
</beans>
```
Spring integration

```xml
<akka:supervision id="my-supervisor">
  <akka:restart-strategy failover="AllForOne"
    retries="3"
    timerange="1000">
    <akka:trap-exits>
      <akka:trap-exit>java.io.IOException</akka:trap-exit>
    </akka:trap-exits>
  </akka:restart-strategy>
  <akka:typed-actors>
    <akka:typed-actor interface="com.biz.MyPOJO"
      implementation="com.biz.MyPOJOImpl"
      lifecycle="permanent"
      timeout="1000"/>
  </akka:typed-actors>
</akka:supervision>
```
Akka Camel
class MyConsumer extends Actor with Consumer {
  def endpointUri = "file:data/input"

  def receive = {
    case msg: Message =>
      log.info("received %s" format msg.bodyAs(classOf[String]))
  }
}
class MyConsumer extends Actor with Consumer {
  def endpointUri =
    "jetty:http://0.0.0.0:8877/camel/test"

  def receive = {
    case msg: Message =>
      reply("Hello %s" format
        msg.bodyAs(classOf[String]))
  }
}
class CometProducer
  extends Actor with Producer {

  def endpointUri = "cometd://localhost:8111/test"

  def receive = produce // use default impl
}

Camel: producer
val producer = actorOf[CometProducer].start

val time = "Current time: " + new Date
producer ! time
Akka HotSwap
actor ! HotSwap({
  // new body
  case Ping =>
  ...
  case Pong =>
  ...
})
self.become({
  // new body
  case Ping =>
  ...
  case Pong =>
  ...
})
How to run it?

- Deploy as dependency JAR in WEB-INF/lib etc.
- Run as stand-alone microkernel
- OSGi-enabled; drop in any OSGi container (Spring DM server, Karaf etc.)
Akka Kernel
Start Kernel

```
java -jar akka-1.0-SNAPSHOT.jar \
-Dakka.config=<path>/akka.conf
```
...and much much more

REST
PubSub
FSM
Comet
OSGi
Security
Web
Guice
Learn more

http://akkasource.org
EOF