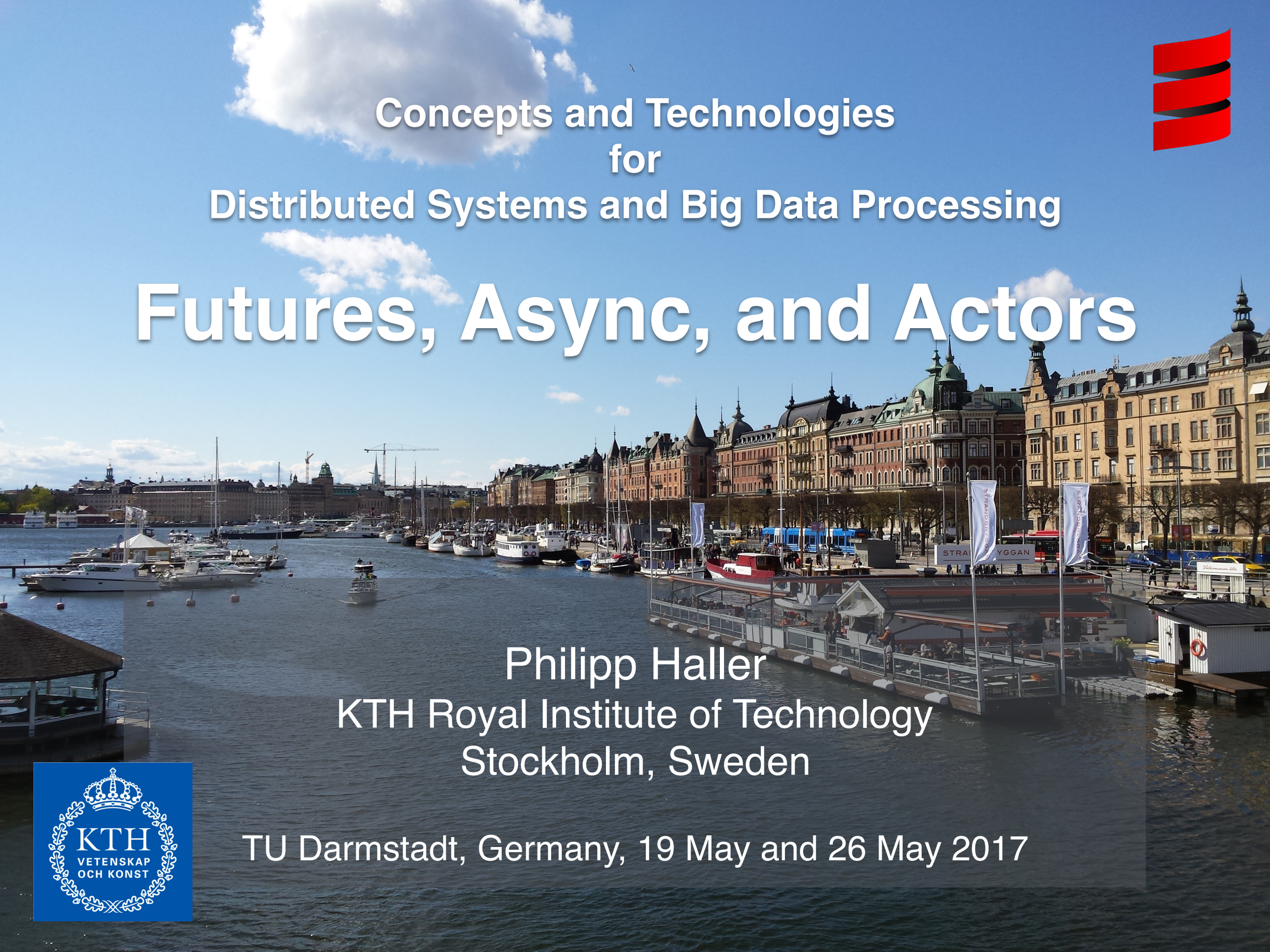

Concepts and Technologies
for
Distributed Systems and Big Data Processing

Futures, Async, and Actors

Philipp Haller
KTH Royal Institute of Technology
Stockholm, Sweden

TU Darmstadt, Germany, 19 May and 26 May 2017

About Myself

- 2006 Dipl.-Inform.
  Karlsruhe Institute of Technology (KIT), Germany
- 2010 Ph.D. in Computer Science
  Swiss Federal Institute of Technology Lausanne (EPFL), Switzerland
- 2011–2012 Postdoctoral Fellow
  Stanford University, USA, and EPFL, Switzerland
- 2012–2014 Consultant and software engineer
  Typesafe, Inc.
- 2014—present Assistant Professor of Computer Science
  KTH Royal Institute of Technology, Sweden

Programming a Concurrent World

- How to compose programs handling
  - ***asynchronous events?***
  - ***streams*** of asynchronous events?
  - ***distributed*** events?

➟ Programming abstractions for concurrency!

Overview

- Futures and promises
- Async/await
- Actors

Why a Growable Language for Concurrency?

- Concurrency not a solved problem ➟ development of new programming models

- Futures, promises
- Async/await
- STM
- Agents
- Actors
- Join-calculus
- Reactive streams
- CSP
- CML

Which one is going to "win"?

Background

- Authored or co-authored:
  - Scala Actors (2006)
  - Scala futures and promise
  - Scala Async (2013)
- Contributed to Akka (Typesa
- Akka.js project (2014)

Other proposals and research projects:

- Scala Joins (2008)
- FlowPools (2012)
- Spores (safer closures)
- Capabilities and uniqueness
- …

Scala Primer

- Local variables:

```
val x = fun(arg)  // type inference
```

- Collections:

```
val list = List(1, 2, 3)  // list: List[Int]
```

- Functions:
  - `{ param => fun(param) }`
  - `(param: T) => fun(param)`
  - Function type: `T => S` or `(T, S) => U`

Scala Primer (2)

- Methods: `def meth(x: T, y: S): R = { .. }`
- Classes: `class C extends D { .. }`
- Generics:
  - `class C[T] { var fld: T = _ ; .. }`
  - `def convert[T](obj: T): Result = ..`

Scala Primer (3)

- Case classes and pattern matching:
  - `case class Person(name: String, age: Int)`
  - ```
    val isAdult =
        p match { case Person(_, a) => a >= 18
                  case Alien(_, _) => false }
    ```
    ```

Example

- Common task:
  - Convert object to JSON
  - Send HTTP request containing JSON

```
import scala.util.parsing.json._

def convert[T](obj: T): Future[JSONType]
def sendReq(json: JSONType): Future[JSONType]
```

Latency numbers every programmer should know

L1 cache reference	0.5ns	
Branch mispredict	5ns	
L2 cache reference	7ns	
Mutex lock/unlock	25ns	
Main memory reference	100ns	
Compress 1K bytes with Zippy	3,000ns	= 3**µs**
Send 2K bytes over 1Gbps network	20,000ns	= 20**µs**
SSD random read	150,000ns	= 150**µs**
Read 1 MB sequentially from memory	250,000ns	= 250**µs**
Roundtrip within same datacenter	500,000ns	= 0.5**ms**
Read 1MB sequentially from SSD	1,000,000ns	= 1**ms**
Disk seek	10,000,000ns	= 10**ms**
Read 1MB sequentially from disk	20,000,000ns	= 20**ms**
Send packet US → Europe → US	150,000,000ns	= 150**ms**

Original compilation by Peter Norvig, w/ contributions by Joe Hellerstein & Erik Meijer

Latency numbers: humanized!

Seconds:

L1 cache reference	0.5 s	One heart beat
Branch mispredict	5 s	Yawn
L2 cache reference	7 s	Long yawn
Mutex lock/unlock	25 s	Making a coffee

Minutes:

Main memory reference	100 s	Brushing your teeth
Compress 1KB with Zippy	50 min	One episode of a TV show

Latency numbers: humanized!

Hours:

Send 2KB over 1 Gbps network	5.5 hr	From lunch to end of work day

Days:

SSD random read	1.7 days	A normal weekend
Read 1MB sequentially from memory	2.9 days	A long weekend
Round trip within same datacenter	5.8 days	A medium vacation
Read 1MB sequentially from SSD	11.6 days	Waiting almost 2 weeks for a delivery

Latency numbers: humanized!

Months:

Disk seek	16.5 weeks	A semester at university
Read 1MB sequentially from disk	7.8 months	Almost producing a new human being
The above 2 together	1 year	

Years:

Send packet US → Europe → US	4.8 years	Average time it takes to complete a bachelor's degree

Callbacks

- How to respond to ***asynchronous completion event?***

➠ Register callback

```
val person = Person(“Tim”, 25)

val fut: Future[JSONType] = convert(person)

fut.foreach { json =>
  val resp: Future[JSONType] = sendReq(json)
  ..
}
```

Exceptions

- Serialization to JSON may fail at runtime
  - Closure passed to `foreach` not executed in this case
  - How to handle asynchronous exceptions?

```
val fut: Future[JSONType] = convert(person)

fut.onComplete {
  case Success(json) =>
    val resp: Future[JSONType] = sendReq(json)
  case Failure(e) =>
    e.printStackTrace()
}
```

Partial Functions

```
{
  case Success(json) => ..
  case Failure(e) => ..
}
```

... creates an instance of PartialFunction[T, R]:

```
val pf: PartialFunction[Try[JSONType], Any] = {
  case Success(json) => ..
  case Failure(e) => ..
}
```

Type of Partial Functions

type PartialFunction[A, B]

a subtype of Function1[A, B]

Actually: **trait** rather than **abstract class**

Simplified!

```
abstract class Function1[A, B] {
  def apply(x: A): B
  ..
}

abstract class PartialFunction[A, B] extends Function1[A, B] {
  def isDefinedAt(x: A): Boolean
  def orElse[A1 <: A, B1 >: B]
        (that: PartialFunction[A1, B1]): PartialFunction[A1, B1]
  ..
}
```

Success and Failure

```
package scala.util

sealed abstract class Try[+T]

final case class Success[+T](v: T) extends Try[T]

final case class Failure[+T](e: Throwable)
  extends Try[T]
```

Nested Exceptions

➟ Exception handling tedious and not compositional:

```
val fut: Future[JSONType] = convert(person)

fut.onComplete {
  case Success(json) =>
    val resp: Future[JSONType] = sendReq(json)
    resp.onComplete {
      case Success(jsonResp) => .. // happy path
      case Failure(e1) =>
        e1.printStackTrace(); ???
    }
  case Failure(e2) =>
    e2.printStackTrace(); ???
}
```

Failed Futures

- Future[T] is completed with Try[T], i.e., with success or failure
- Combinators enable compositional failure handling
- Example:

```
val resp: Future[JSONType] = sendReq(json)
val processed = resp.map { jsonResp =>
  .. // happy path
}
```

Encapsulates failure

Map Combinator

- Creates a new future by ***applying a function*** to the successful result of the receiver future
- If the ***function application*** results in an ***uncaught exception e*** then the new future is completed with ***e***
- If the ***receiver future*** is completed with an ***exception e*** then the new future is also completed with ***e***

```
abstract class Future[+T] extends Awaitable[T] {

  def map[S](f: T => S)(implicit ..): Future[S]

  // ..
}
```

Future Composition

```
val fut: Future[JSONType] = convert(person)

val processed = fut.map { json =>
  val resp: Future[JSONType] = sendReq(json)
  resp.map { jsonResp =>
    ... // happy path
  }
}
```

Encapsulates *all failures*

Problem: `processed` has type `Future[Future[T]]`

Future Pipelining

```
val fut: Future[JSONType] = convert(person)

val processed = fut.flatMap { json =>
  val resp: Future[JSONType] = sendReq(json)
  resp.map { jsonResp =>
    .. // happy path
  }
}
```

Future pipelining: the result of the inner future (result of `map`) determines the result of the outer future (`processed`)

FlatMap Combinator

- Creates a new future by ***applying a function*** to the successful result of the receiver future
- The ***future result*** of the function application ***determines*** the result of the new future
- If the ***function application*** results in an ***uncaught exception e*** then the new future is completed with ***e***
- If the ***receiver future*** is completed with an ***exception e*** then the new future is also completed with ***e***

```
def flatMap[S](f: T => Future[S])(implicit ..): Future[S]
```

Creating Futures

- Futures are created based on (a) computations, (b) events, or (c) combinations thereof
- Creating computation-based futures:

```
object Future {

  def apply[T](body: => T)(implicit ..): Future[T]

}
```

"Code block" with result type T

Singleton object

"Unrelated" to the singleton object!

Futures: Example

```
val firstGoodDeal = Future {
  usedCars.find(car => isGoodDeal(car))
}
```

Short syntax for:

```
val firstGoodDeal = Future.apply({
  usedCars.find(car =>                    ))
})
```

Type
Future[Option[Car]]

Type inference:

```
val firstGoodDeal = Future.apply[Option[Car]]({
  usedCars.find(car => isGoodDeal(car))
})
```

Creating Futures: Operationally

- Invoking the shown factory method creates a ***task object*** encapsulating the computation
- The task object is ***scheduled for execution*** by an execution context
- An execution context is capable of executing tasks, typically using a ***thread pool***
- Future tasks are submitted to the current ***implicit execution context***

```
def apply[T](body: => T)(implicit
                executor: ExecutionContext): Future[T]
```

Implicit Execution Contexts

Implicit parameter requires selecting a execution context

```
Welcome to Scala 2.12.2 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_..).
Type in expressions for evaluation. Or try :help.

scala> import scala.concurrent._
import scala.concurrent._

scala> val fut = Future { 40 + 2 }

<console>:10: error: Cannot find an implicit ExecutionContext. You might pass
an (implicit ec: ExecutionContext) parameter to your method
or import scala.concurrent.ExecutionContext.Implicits.global.
       val fut = Future { 40 + 2 }
                          ^
```

But...

Execution Contexts

- Interface for asynchronous task executors
- May wrap a `java.util.concurrent.{Executor, ExecutorService}`

Collections of Futures

```
val reqFuts: List[Future[JSONType]] = ..

val smallestRequest: Future[JSONType] =
  Future.sequence(reqFuts).map(
    reqs => selectSmallestRequest(reqs)
  )
```

Promise

Main purpose: create futures for non-lexically-scoped asynchronous code

Example

Function for creating a Future that is completed with ***value*** after ***delay*** milliseconds

```
def after[T](delay: Long, value: T): Future[T]
```

“after”, Version 1

```
def after1[T](delay: Long, value: T) =
  Future {
    Thread.sleep(delay)
    value
  }
```

“after”, Version 1

How does it behave?

```
assert(Runtime.getRuntime()
                .availableProcessors() == 8)

for (_ <- 1 to 8) yield
  after1(1000, true)

val later = after1(1000, true)
```

Quiz: when is “later” completed?

Answer: after either ~1 s or ~2 s (most often)

Promise

```
object Promise {
  def apply[T](): Promise[T]
}
```

```
trait Promise[T] {
  def success(value: T): Promise[T]
  def failure(cause: Throwable): Promise[T]

  def future: Future[T]
}
```

“after”, Version 2

```
def after2[T](delay: Long, value: T) = {
  val promise = Promise[T]()

  timer.schedule(new TimerTask {
    def run(): Unit = promise.success(value)
  }, delay)

  promise.future
}
```

Much better behaved!

Futures and Promises: Conclusion

- Scala enables flexible concurrency abstractions
- Futures: high-level abstraction for asynchronous events and computations
  - Combinators instead of callbacks
- Promises enable integrating futures with any event-driven API

Overview

- Futures, promises
- Async/await
- Actors

What is Async?

- Scala module
  - `"org.scala-lang.modules" %% "scala-async"`
- Purpose: ***simplify programming with futures***
- Scala Improvement Proposal SIP-22
- Releases for Scala 2.10, 2.11, and 2.12
  - See `https://github.com/scala/async/`

What Async Provides

- Future and Promise provide ***types*** and operations for managing ***data flow***
  - Very little support for control flow
- Async complements Future and Promise with constructs to manage ***control flow***

Programming Model

Basis: ***suspendible computations***

- `async { .. }` — ***delimit*** suspendible computation
- `await(future)` — ***suspend*** computation until `future` is completed

Async

```
object Async {

  def async[T](body: => T): Future[T]

  def await[T](future: Future[T]): T

}
```

Example

```
val fstGoodDeal: Future[Option[Car]] = ..
val sndGoodDeal: Future[Option[Car]] = ..

val goodCar = async {
  val car1 = await(fstGoodDeal).get
  val car2 = await(sndGoodDeal).get
  if (car1.price < car2.price) car1
  else car2
}
```

Futures vs. Async

- "Futures and Async: When to Use Which?", Scala Days 2014, Berlin
- Video: *https://www.youtube.com/watch?v=TyuPdFDxkro*
- Slides: *https://speakerdeck.com/phaller/futures-and-async-when-to-use-which*

Async in Other Languages

Constructs similar to async/await are found in a number of widely-used languages:

- C#
- Dart (Google)
- Hack (Facebook)
- ECMAScript 7 [1]

[1] *http://tc39.github.io/ecmascript-asyncawait/*

From Futures to Actors

- Limitations of futures:
  - At most one completion event per future
  - Overhead when creating many futures
- How to model distributed systems?

The Actor Model

- ***Model of concurrent computatio*** the "actor" [Hewitt et al. '73]

Related to *active objects*

- Actors = concurrent "processes" communicating via asynchronous messages
- Upon reception of a message, an actor may
  - change its behavior/state
  - send messages to actors (including itself)
  - create new actors
- Fair scheduling
- Decoupling: message sender cannot fail due to receiver

Example

```
class ActorWithTasks(tasks: ...) extends Actor {
  ...

  def receive = {
    case TaskFor(workers) =>
      val from = sender

      val requests = (tasks zip workers).map {
        case (task, worker) => worker ? task
      }

      val allDone = Future.sequence(requests)

      allDone andThen { seq =>
        from ! seq.mkString(",")
      }
  }
}
```

Using ***Akka*** (http://akka.io/)

Anatomy of an Actor (1)

- An actor is an active object with its own behavior
- Actor behavior defined by:
  - subclassing `Actor`
  - implementing `def receive`

```
class ActorWithTasks(tasks: List[Task]) extends Actor {
  def receive = {
    case TaskFor(workers) =>  // send `tasks` to `workers`
    case Stop             =>  // stop `self`
  }
}
```

Anatomy of an Actor (2)

- Exchanged messages should be ***immutable***
  - And ***serializable***, to enable remote messaging
- Message types should implement ***structural equality***
- In Scala: ***case classes*** and ***case objects***

- Enables pattern matching on the receiver side

```
case class  TaskFor(workers: List[ActorRef])
case object Stop
```

Anatomy of an Actor (3)

- Actors are ***isolated***
  - Strong encapsulation of state
- Requires restricting ***access*** and ***creation***
- Separate Actor instance and ActorRef
  - ActorRef public, safe interface to actor

```
val system = ActorSystem(“test-system”)
val actor1: ActorRef = system.actorOf[ActorWithTasks]

actor1 ! TaskFor(List())  // async message send
```

Why Actors?

Reason 1: simplified concurrency

- "Share nothing": strong isola
  ➟ no race conditions

"Macro-step semantics"

- Actors handle at most one message at a time
  ➟ sequential reasoning

- Asynchronous message handling
  ➟ less risk of deadlocks

- No "inversion of control": access to own state and messages in safe, direct way

Why Actors? (cont'd)

Reason 2: actors model reality of distributed systems

- Message sends truly asynchronous
- Message reception not guaranteed
- Non-deterministic message ordering
  - Some implementations preserve message ordering between ***pairs*** of actors

Therefore, actors well-suited as a foundation for distributed systems

Summary

- Concurrency benefits from ***growable languages***
- ***Futures and promises*** a versatile abstraction for single, asynchronous events
  - Supported by ***async/await***
- The ***actor model*** faithfully models distributed systems