ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Latency as an Effect

Principles of Reactive Programming

Erik Meijer

The Four Essential Effects In Programming

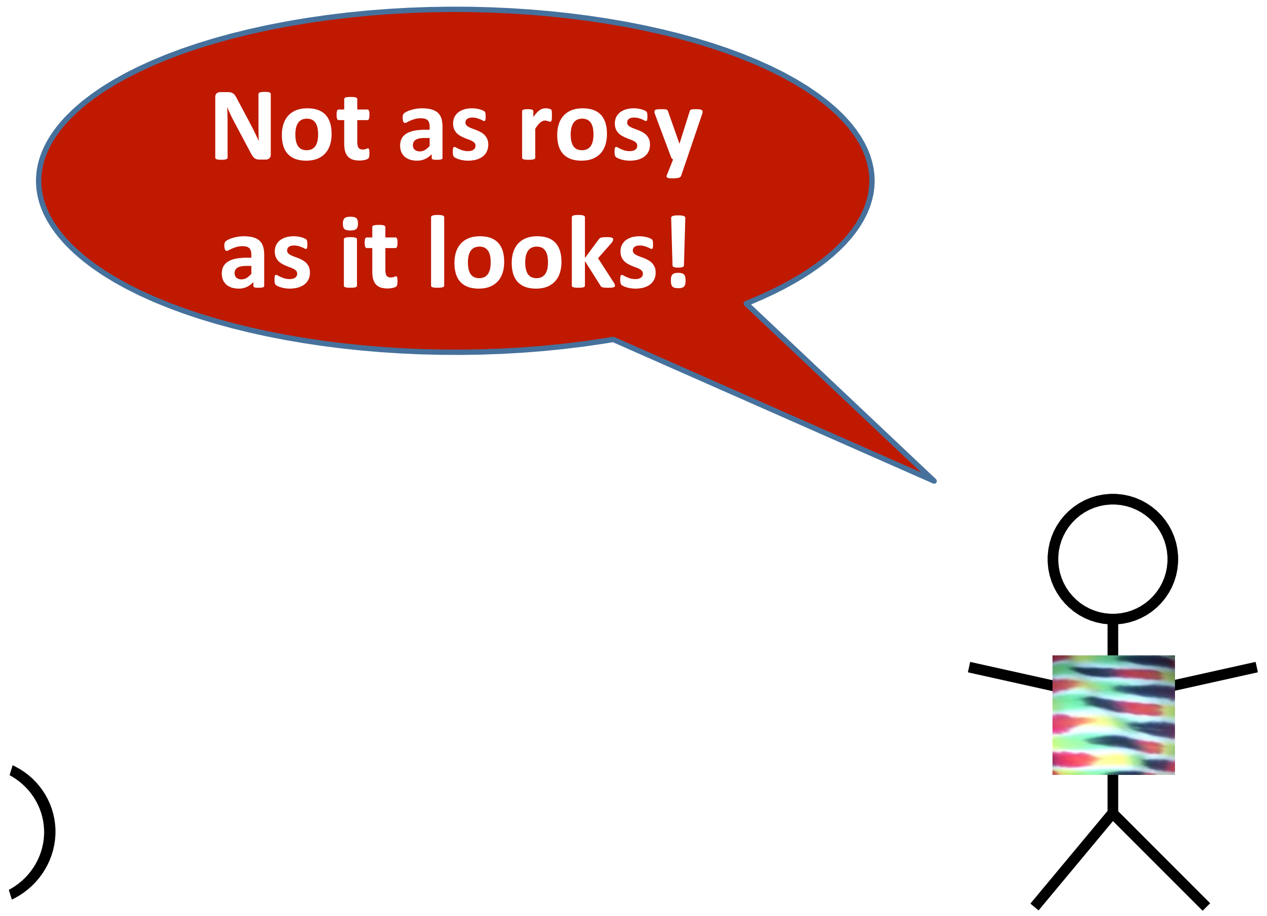
	One	Many
Synchronous	`T/Try[T]`	`Iterable[T]`
Asynchronous	`Future[T]`	`Observable[T]`

A simple adventure game ....

```
trait Adventure {
  def collectCoins(): List[Coin]
  def buyTreasure(coins: List[Coin]): Treasure
}


val adventure = Adventure()
val coins = adventure.collectCoins()
val treasure = adventure.buyTreasure(coins)
```

is very similar to a simple network stack

```
trait Socket {
  def readFromMemory(): Array[Byte]
  def sendToEurope(packet: Array[Byte]): Array[Byte]
}
```




```
val socket = Socket()
val packet = socket.readFromMemory()
val confirmation = socket.sendToEurope(packet)
```

Timings for various operations on a typical PC on human scale

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

http://norvig.com/21-days.html#answers

Sequential composition of actions that take time

```
val socket = Socket()
val packet = socket.readFromMemory()
// block for 50,000 ns
// only continue if there is no exception
val confirmation = socket.sendToEurope(packet)
// block for 150,000,000 ns
// only continue if there is no exception
```

Sequential composition of actions

Lets translate this into human terms.

1 nanosecond

$\longrightarrow$

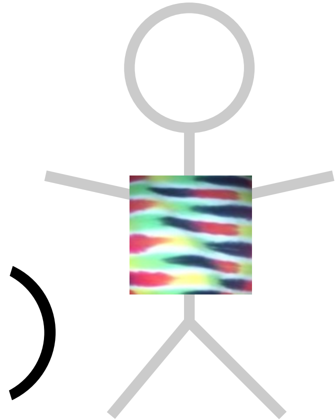
1 second (then hours/days/months/years)

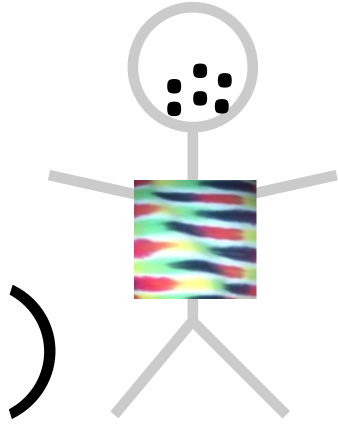
Timings for various operations on a typical PC on human scale

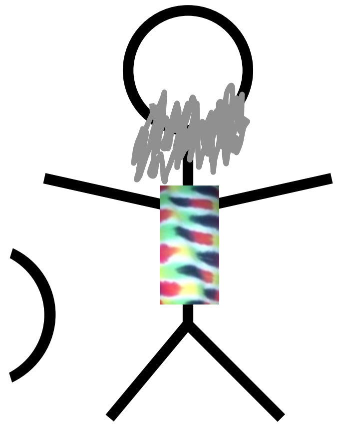
execute typical instruction	1 second
fetch from L1 cache memory	0.5 seconds
branch misprediction	5 seconds
fetch from L2 cache memory	7 seconds
Mutex lock/unlock	½ minute
fetch from main memory	1½ minutes
send 2K bytes over 1Gbps network	5½ hours
read 1MB sequentially from memory	3 days
fetch from new disk location (seek)	13 weeks
read 1MB sequentially from disk	6½ months
send packet US to Europe and back	5 years

Sequential composition of actions

```
val socket = Socket()
val packet = socket.readFromMemory()
// block for 3 days
// only continue if there is no exception
val confirmation = socket.sendToEurope(packet)
// block for 5 years
// only continue if there is no exception
```





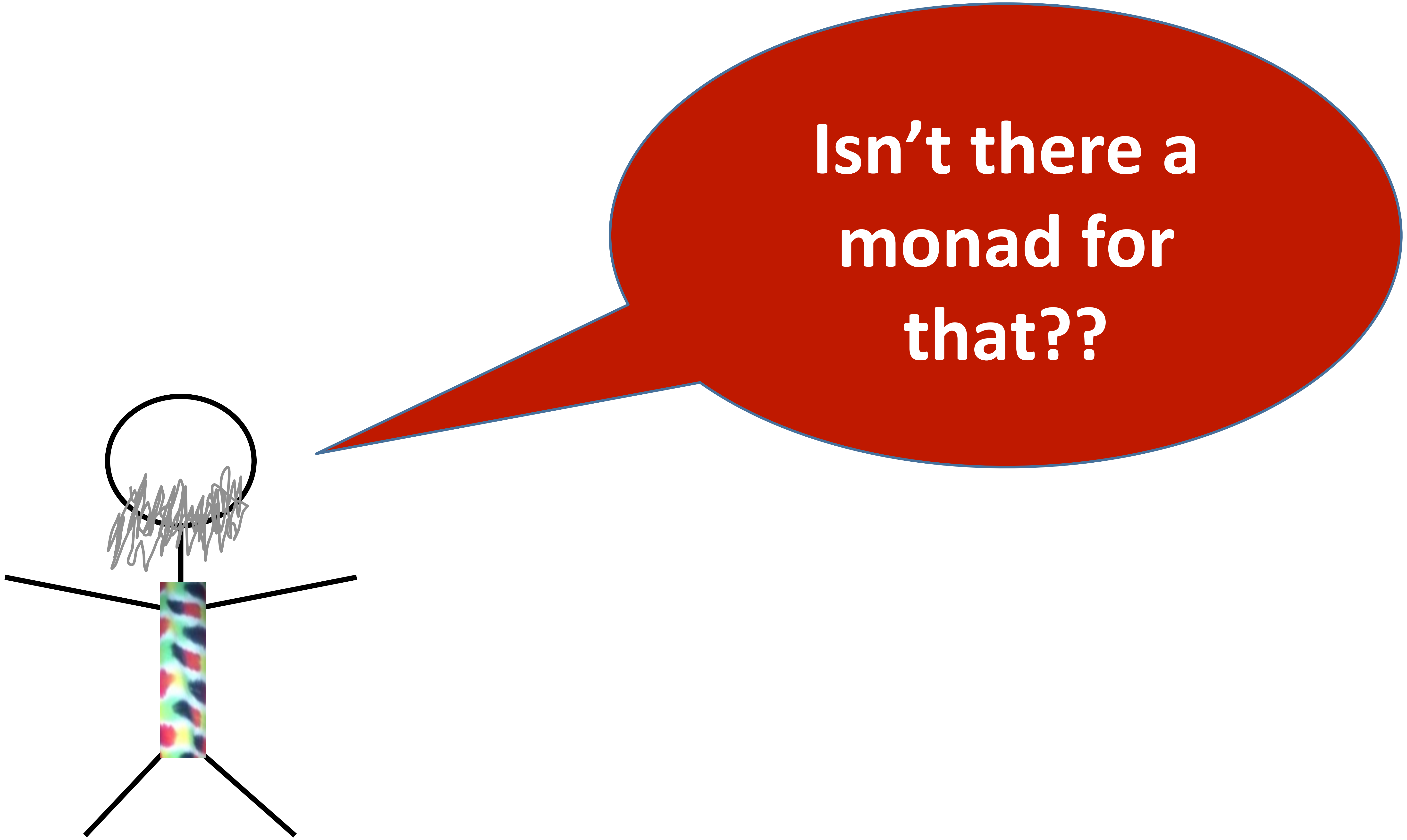


Sequential composition of actions

12 months to walk coast-to-coast
3 months to swim across the Atlantic
3 months to swim back
12 months to walk back

Humans are twice as fast as computers!

Sequential composition of actions that take time and fail



Monads guide you through the happy path

`Future[T]`

A monad that handles exceptions and **latency**.

Futures asynchronously notify consumers

```
import scala.concurrent._
import scala.concurrent.ExecutionContext.Implicits.global

trait Future[T] {
  def onComplete(callback: Try[T] ⇒ Unit)
    (implicit executor: ExecutionContext): Unit
}
```

Futures alternative designs

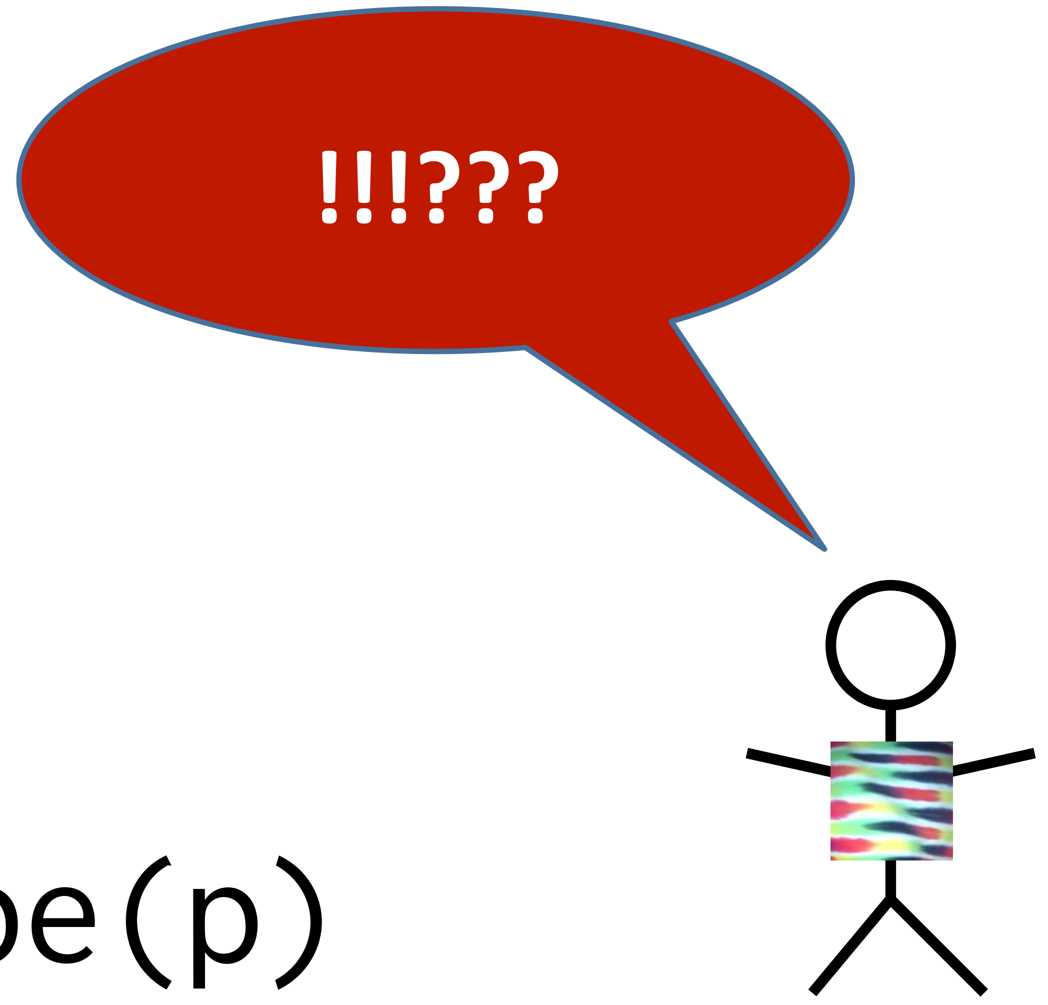
```
trait Future[T] {
  def onComplete
     (success: T ⇒ Unit, failed: Throwable ⇒ Unit): Unit

  def onComplete(callback: Observer[T]): Unit
}

trait Observer[T] {
  def onNext(value: T): Unit
  def onError(error: Throwable): Unit
}
```

Futures asynchronously notify consumers

```
import scala.concurrent._

trait Future[T] {
  def onComplete(callback: Try[T] ⇒ Unit)
    (implicit executor: ExecutionContext): Unit
}

trait Socket {
  def readFromMemory(): Future[Array[Byte]]
  def sendToEurope(packet: Array[Byte]): Future[Array[Byte]]
}
```

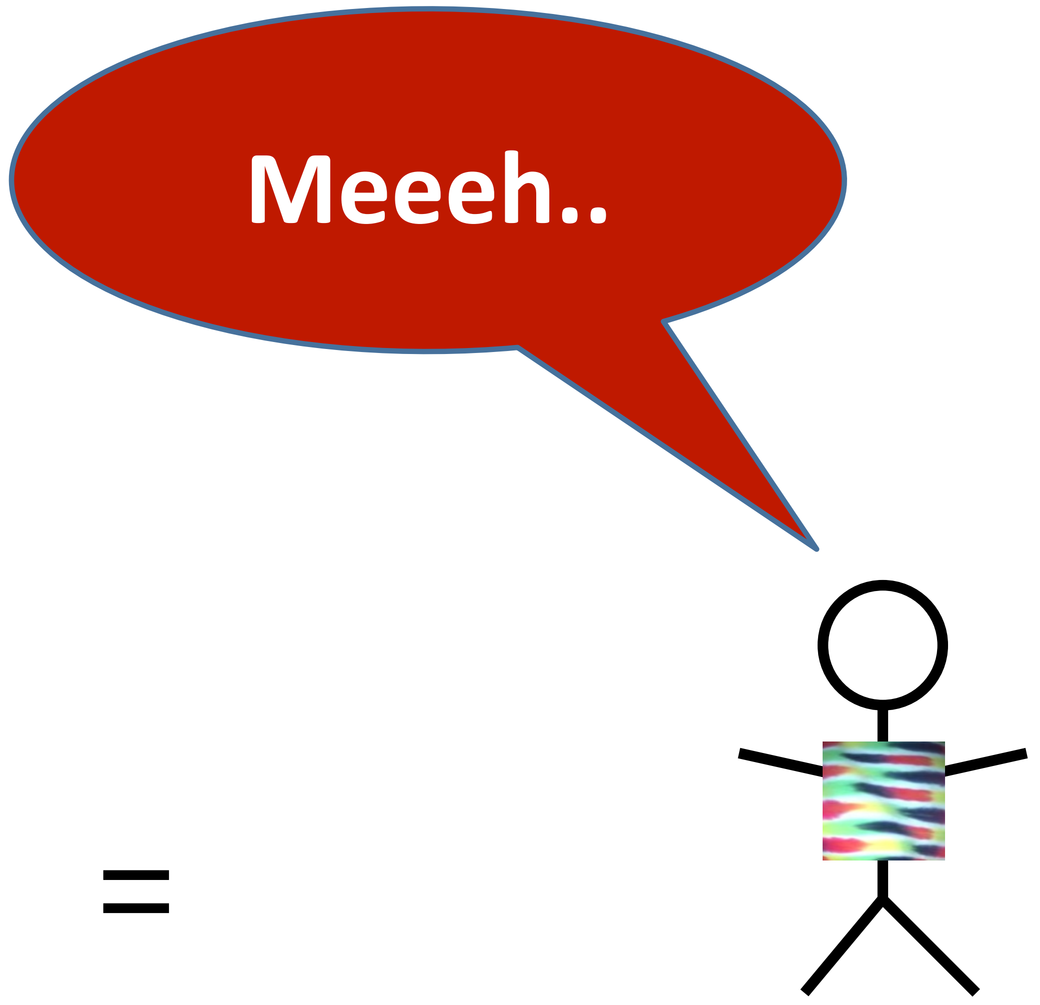
Send packets using futures I

```
val socket = Socket()
val packet: Future[Array[Byte]] =
  socket.readFromMemory()

val confirmation: Future[Array[Byte]] =
  packet onComplete {
    case Success(p) => socket.sendToEurope(p)
    case Failure(t) => …
  }
```



Send packets using futures II

```
val socket = Socket()
val packet: Future[Array[Byte]] =
  socket.readFromMemory()

packet onComplete {
  case Success(p) ⇒ {
    val confirmation: Future[Array[Byte]] =
      socket.sendToEurope(p)
  }
  case Failure(t) => …
}
```



Creating Futures

```
// Starts an asynchronous computation
// and returns a future object to which you
// can subscribe to be notified when the
// future completes

object Future {
  def apply(body: ⇒T)
    (implicit context: ExecutionContext): Future[T]
}
```

Creating Futures

```
import scala.concurrent.ExecutionContext.Implicits.global
import akka.serializer._

val memory = Queue[EMailMessage](
  EMailMessage(from = “Erik”, to = “Roland”),
  EMailMessage(from = “Martin”, to = “Erik”),
  EMailMessage(from = “Roland”, to = “Martin”))

def readFromMemory(): Future[Array[Byte]] = Future {
  val email = queue.dequeue()
  val serializer = serialization.findSerializerFor(email)
  serializer.toBinary(email)
}
```

Quiz

```
import scala.concurrent.ExecutionContext.Implicits.global

val packet: Future[Array[Byte]] = socket.readFromMemory()

packet onSuccess {
  case bs ⇒ socket.sendToEurope(p)
}

packet onSuccess {
  case bs ⇒ socket.sendToEurope(p)
}
```

How many messages are left in the e-mail queue?

a) 3
b) 2
c) 1
d) 0