

Scala Actors

Scalable Multithreading on the JVM

Philipp Haller

Ph.D. candidate

Programming Methods Lab

EPFL, Lausanne, Switzerland

The free lunch is over!

- Software is concurrent
 - Interactive applications
 - Web services
 - Distributed software
- Hardware is concurrent
 - Hyper-threading
 - Multi-cores, Many-cores
 - Grid computing

Concurrency on the JVM

Threads and locks (`synchronized`):

- Error-prone [Ousterhout96]
 - races vs. deadlock, not composable
- Correct solutions often *don't scale*
 - memory consumption, lock contention
- Debugging and testing is hard
 - hard to reproduce executions (non-determinism)

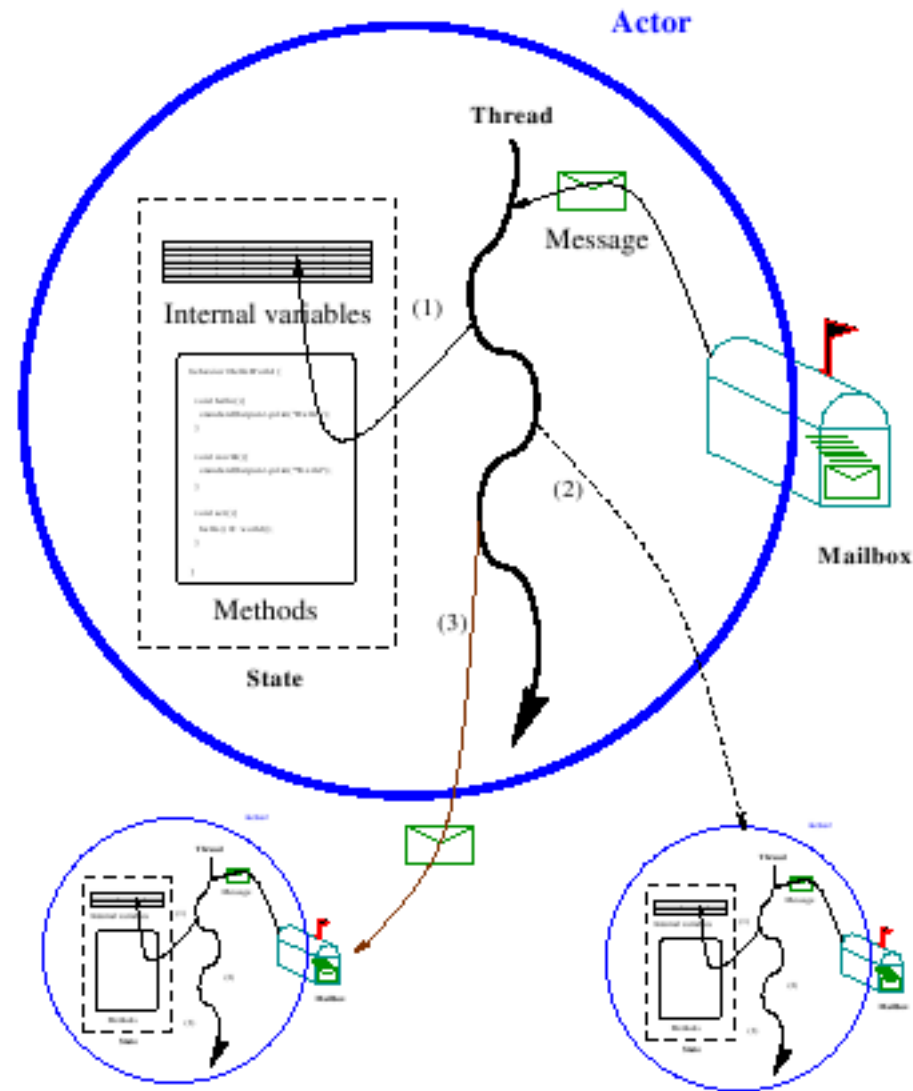
Outline

- Overview of Scala Actors
- Example
- Event-based Actors
- The *lift* Web Framework
- Performance
- Conclusion

Scala Actors

- Light-weight concurrent processes
 - *Actor* model (Erlang-style processes)
 - Asynchronous message passing
 - Expressive pattern matching
- Unify threads and events (efficient, scalable)
- Automatically mapped to multiple JVM threads
 - leverage multi-core processors
- No inversion of control

What is an actor?



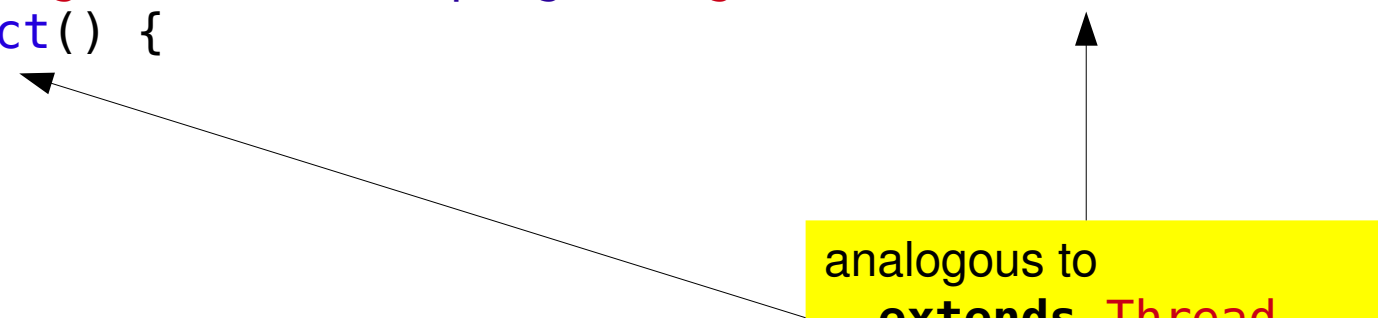
Actors in a Nutshell

- Actors encapsulate state and behavior (like objects)
- Actors are logically active (unlike most objects)
- Actors communicate through asynchronous message passing (*non-blocking* send, *blocking* receive)

Example

```
class Ping(count: int, pong: Pong) extends Actor {  
  def act() {  
  }  
}
```

analogous to
... extends Thread
... def run() { ... }



Message Send/Receive

```
class Ping(count: int, pong: Pong) extends Actor {  
  def act() {  
  
    pong ! 'Ping  
  
    receive {  
      case 'Pong =>  
  
    }  
  }  
}
```

The “Ping” Actor

```
class Ping(count: int, pong: Pong) extends Actor {
  def act() {
    var pingsLeft = count - 1
    pong ! 'Ping
    while (true) {
      receive {
        case 'Pong =>
          Console.println("Ping: 'Pong received")
          if (pingsLeft > 0) {
            Thread.sleep(500)
            pong ! 'Ping
            pingsLeft -= 1
          } else {
            pong ! 'Stop
            exit()
          }
        }
      }
    }
  }
}
```

Library Features

```
val pong = actor {  
  while (true) {  
    receive {  
      case 'Ping =>  
        Console.println("Pong: 'Ping  
        Thread.sleep(500)  
        sender ! 'Pong  
      case 'Stop =>  
        exit()  
    }  
  }  
}
```

“Inline” definition
of actors:

```
actor {  
  ...  
}
```

Implicit sender Ids: `sender`

Futures

- Invoke asynchronous operation, returning a *future* (a place-holder for the reply) used to
 - wait for reply (blocking)
 - test whether reply available (non-blocking)

```
abstract class Future[T] extends Function0[T] {  
  def isSet: boolean  
}
```

```
trait Function0[+R] extends AnyRef {  
  def apply(): R  
}
```

Futures: Examples

```
val ft = a !! Msg // send message, ft is a future
...
val res = ft()    // await future ft

val ft1 = a !! Msg
val ft2 = b !! Msg
val ft3 = c !! Msg
...
val results = awaitAll(500, ft1, ft2, ft3)
// returns a `List[Option[Any]]' holding the results

val res = awaitEither(ft1, ft2)

val ft = future { // define ad-hoc future
  ...
}
```

More Library Features

- `receiveWithin(timeout)`
- Channels (type-safe communication)
- Java threads are Actors (automatically)
- Linking Actors (monitoring)
- Pluggable schedulers
- Remote Actors
 - over TCP, JXTA not yet released

Event-based Actors

- Do not consume a thread
- Very light-weight representation at run-time
 - closure object (similar to a Runnable)
- Use `react` instead of `receive`
- Restriction:
 - call to `react` does not return
 - at the end: `exit` or call rest of computation
 - shortcuts for sequence and looping

The “Ping” Actor - Event-based

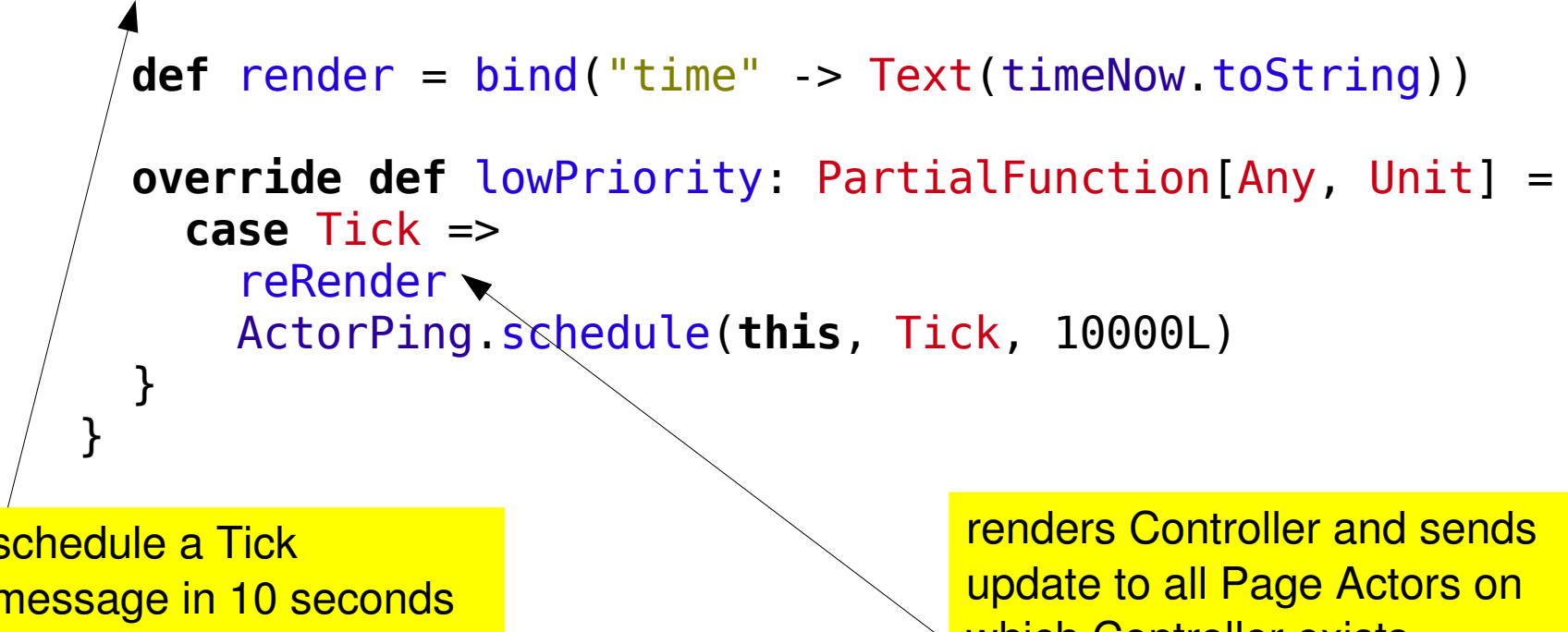
```
class Ping(count: int, pong: Pong) extends Actor {
  def act() {
    var pingsLeft = count - 1
    pong ! 'Ping
    loop {
      react {
        case 'Pong =>
          Console.println("Ping: 'Pong received")
          if (pingsLeft > 0) {
            Thread.sleep(500)
            pong ! 'Ping
            pingsLeft -= 1
          } else {
            pong ! 'Stop
            exit()
          }
        }
      }
    }
  }
}
```


The *lift* Web Framework

- 3rd party web framework
- Compatible with any 2.4 servlet engine
- Multi-threaded, scalable
- Scala Actors used for critical parts
 - Session management (Session Actors)
 - Dynamic content (Controller Actors)
 - update asynchronously, send updates to Page Actors
 - Page Actor updates packaged as DOM-modifying JavaScript sent back to browser

lift: Actor Example

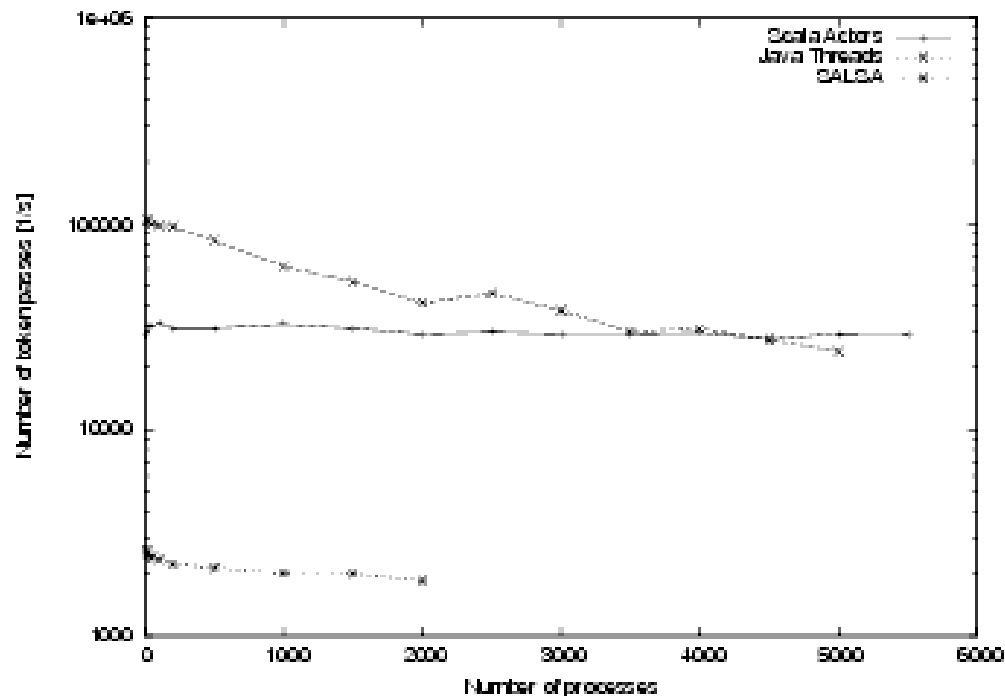
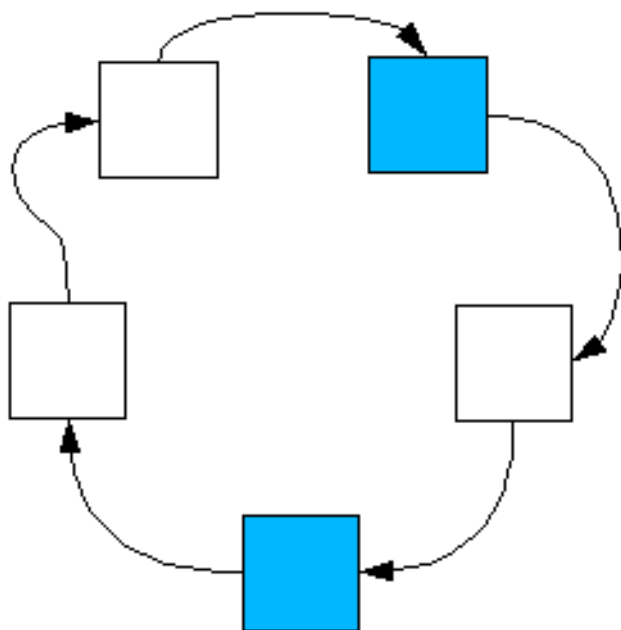
```
class Clock extends ControllerActor {  
  ActorPing.schedule(this, Tick, 10000L)  
  
  def render = bind("time" -> Text(timeNow.toString))  
  
  override def lowPriority: PartialFunction[Any, Unit] = {  
    case Tick =>  
      reRender  
      ActorPing.schedule(this, Tick, 10000L)  
  }  
}
```



schedule a Tick
message in 10 seconds

renders Controller and sends
update to all Page Actors on
which Controller exists

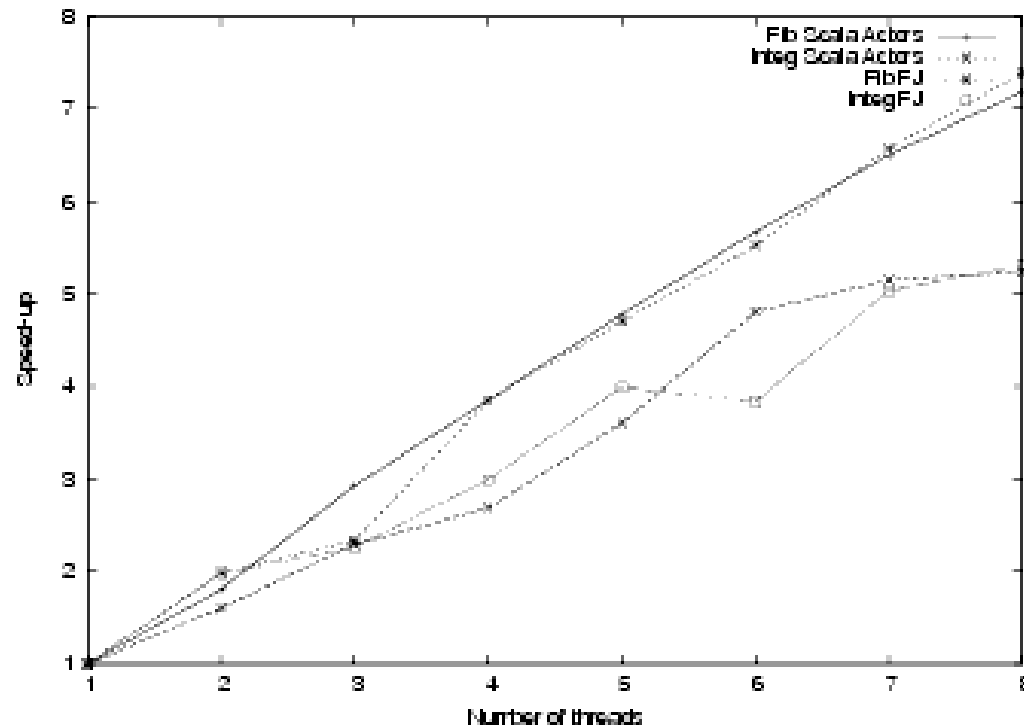
Performance



Token passes per sec. in ring of processes

- Java Threads: only up to 5000 threads, throughput breaks in
- Scala Actors: constant throughput, up to 1.200.000 actors

Scalability on Multi-Cores



- Micro-benchmarks run on 4-way dual-core Opteron machine (8 cores total)
- Compared to Doug Lea's FJTask framework for Java

Gentoo : Intel® Pentium® 4 Computer Language Benchmarks Game

Frequently Asked Questions

Compare the performance of **Scala** programs against some other language implementation, or check the Scala [CPU time and Memory use measurements](#).

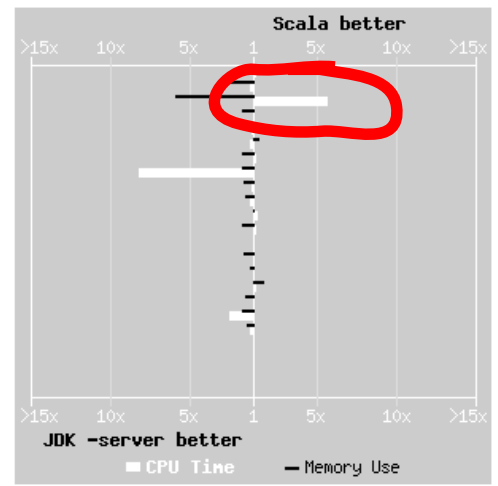
For more information about the Scala implementation we measured see [about Scala](#).

- all benchmarks - Scala

Compare to: Java JDK -server Show

Are the Scala programs better?

For each of one our benchmarks, a white bar shows which language implementation had the faster program, and a black bar shows which used the least memory.



How many times better?

How many times *faster or smaller* are the **Scala** programs than the corresponding Java JDK -server programs?

Program & Logs	Scala x times better		
	Faster	Smaller: Memory Use	Smaller: GZip Bytes
binary-trees	1.1	-1.4	1.1
chameneos	-1.2	-3.0	1.1
cheap-concurrency	5.6	-5.8	1.4
fannkuch	1.0	-1.6	1.2
fasta	-1.1	-1.6	1.0
k-nucleotide	-1.2	1.3	1.0
mandelbrot	1.1	-1.6	1.1
meteor-contest (new)	-8.1	-1.7	1.7
n-body	-1.0	-1.6	
nsieve	-1.1	-1.4	1.2

Summary: Performance

- Millions of actors, constant throughput
- Scalability on multi-cores *without changes in program*
- Real-life experience with *lift* web framework¹⁾:



different computers, they can share a shopping cart.) Controllers are based on Scala Actors. Each controller consumes about 200 bytes plus whatever state the controller keeps around (let's say 2K of state per Controller for a shopping cart.) This means that 10,000 active controllers would consume about 20MB of RAM, or about 1/2 of the RAM used by a single Rails instance. Put another way, keeping state in memory scales.

1) <http://blog.lostlake.org/index.php?/archives/46-Some-more-Rails-to-lift-code-examples.html>

Scala Actors: Take Home

- Multi-threading on the JVM made *easier* and *more scalable*
- Used in real-world frameworks
- Included in Scala standard library
- Documentation/Tutorial at <http://lamp.epfl.ch/~phaller/>
- Try it out: <http://scala-lang.org>
- Send me mail: philipp.haller@epfl.ch

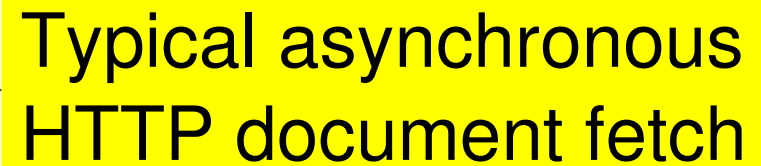
Asynchronous Web Services

- Trend towards rich, responsive web applications
 - e.g. Gmail, Google calendar
 - technologies such as AJAX
- Responsiveness, performance, scalability
- Asynchronicity is key

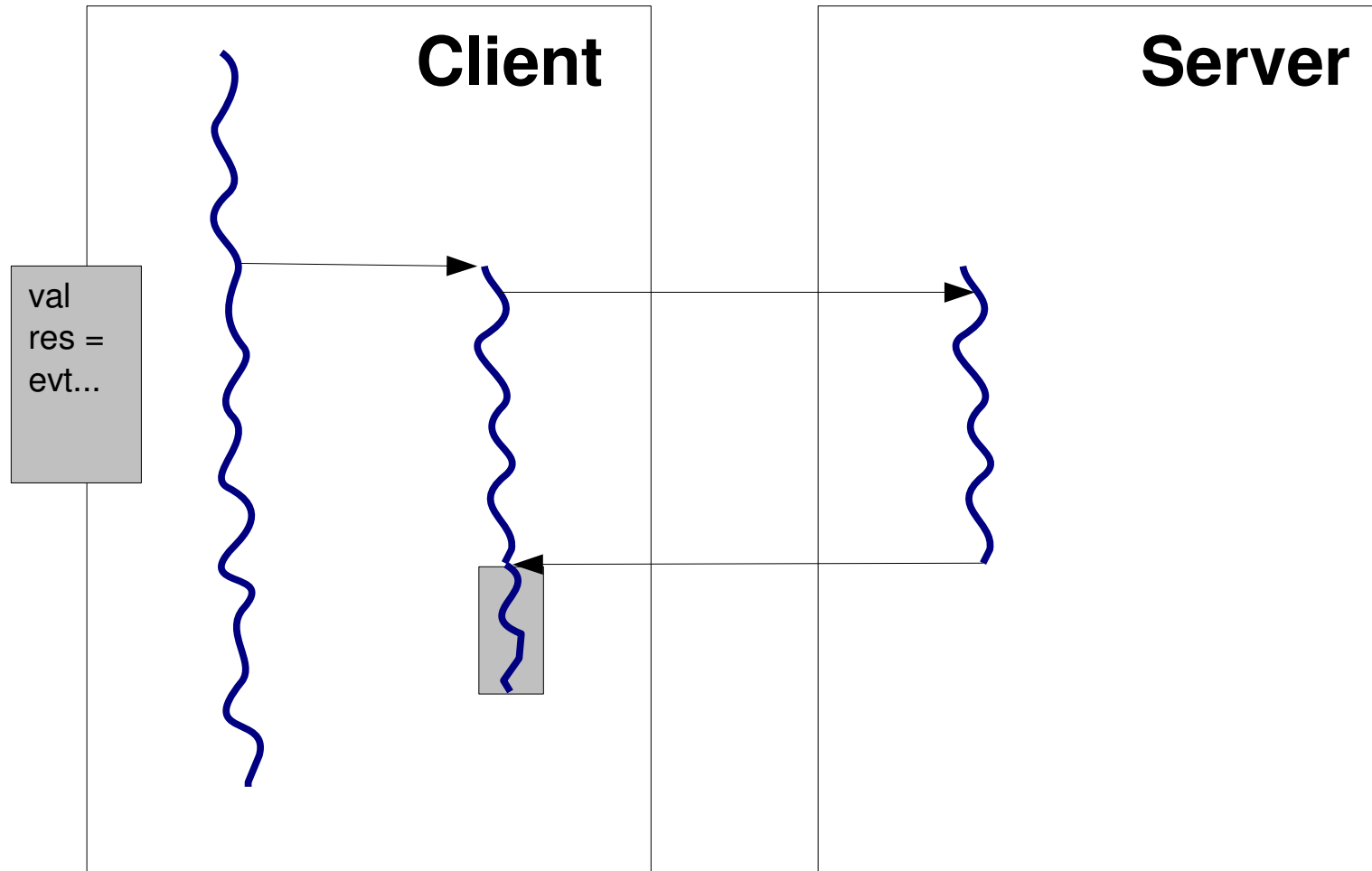
Problem: Asynchronicity is hard

Asynchronicity is hard

```
def httpFetch(queryURL: String) = {  
  val req = new XmlHttpRequest  
  req.addOnReadyStateChangedListener(new PropertyChangeListener() {  
    override def propertyChange(evt: PropertyChangeEvent) {  
      if (evt.getNewValue() == ReadyState.LOADED) {  
        val response = req.getResponseText()  
        httpParseResponse(response)  
      }  
    }  
  })  
  try {  
    req.open(Method.GET, new URL(queryURL))  
    req.send()  
  } catch {  
    case e: Throwable => ...  
  }  
}
```

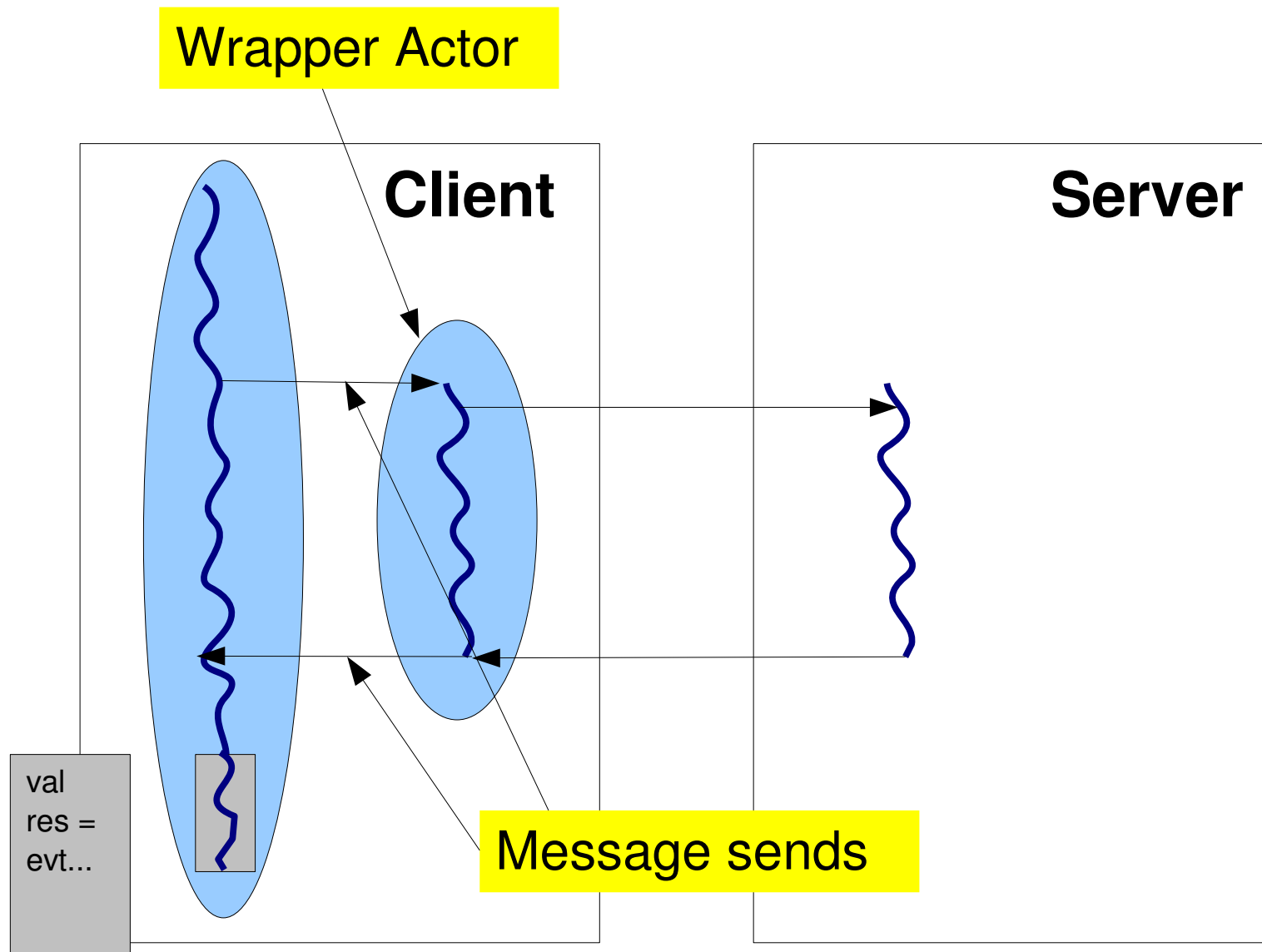


Typical asynchronous
HTTP document fetch



Problems of Inversion of Control

- Hard to understand control-flow
 - reconstruct entire call-graph
- Manual stack management
 - handler code *not* defined where event is handled
 - local variables, parameters etc. not accessible
- Managing resources (files, sockets) becomes *even harder*
 - often long-lived, used in several event handlers
 - when is a missing `close()` a leak?



Avoiding Inversion of Control

Wrapper:

```
val fetcher = actor {  
  loop {  
    react {  
      case HttpFetch(url) =>  
        httpFetch(url)  
    }  
  }  
}
```

Client:

```
fetcher ! HttpFetch("http://www.epfl.ch")  
// do some overlapping computation  
react { // wait for response  
  case Response(content) =>  
    // process response  
}
```