UNIVERSITÄT BERN

11. Actors

Haidar Osman

Motivations

- Mutable shared state is the source of concurrency problems.
- Mixing concurrency logic with business logic makes the code harder to maintain.
- There is an increased need for high performant systems.
  - For instance, big data applications
- Distributing a standalone application is not a straightforward process.
  - You have to deal with different protocols for establishing inter-machine communication.

What are actors?

- The Actors Model is “another” model for expressing computational problems.
  - It is equivalent to Turing Machine and Lambda Calculus
- Actors are not new. It is a solid and scientifically robust idea (Carl Hewitt 1973).

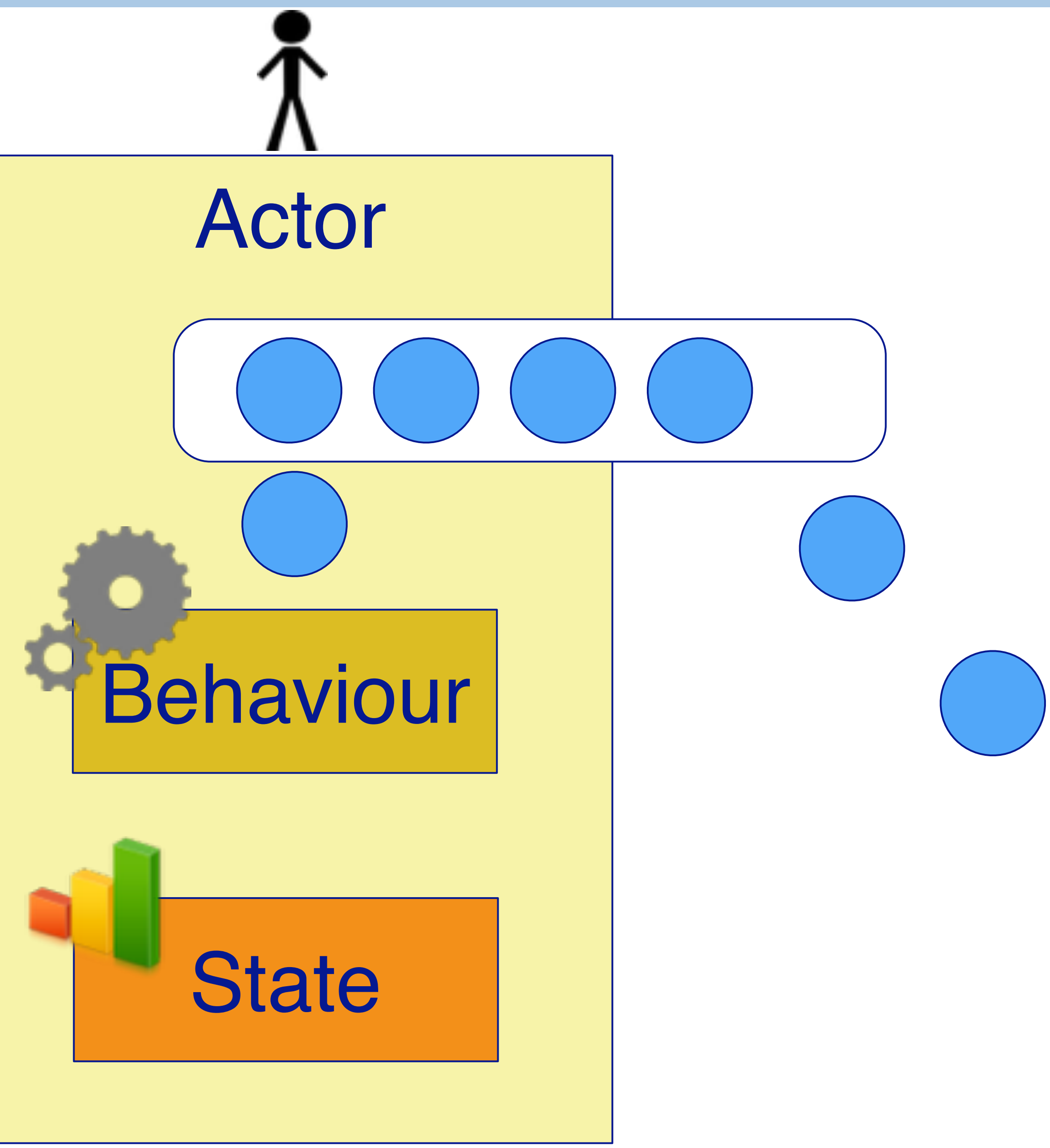
What are actors?

Actors are fundamental units of computation that embody:

- Processing (behaviour)
- Storage (state)
- Communication (messages)

When an actor receives a message, it can:

- create more actors
- send messages to other actors
- designate what to do with the next message



Rules of bare-metal actors

> Every actor has an address.

> An actor can process one message at a time.

> Everything in an actor system is an actor.

> Message delivery is *best-effort*

> There are no guarantees that messages are received in the same order they are sent.

> Messages to actors should always be immutable to avoid shared state.

Example 1 - Simple Actor Definition

```
Class Counter extends Actor{
  var count=0
  def receive = {
    case “incr” => count+=1
    case (“get”, sender:ActorRef) => sender ! count
  }
}
```

Example 1 - Simple Actor Definition

```
Class Counter extends Actor{
  var count=0
  def receive = {
    case “incr” => count+=1
    case (“get”, sender:ActorRef) => sender ! count
  }
}
```

Example 1 - Actor Trait and ActorRef Class

```
Trait Actor{
  implicit val self: ActorRef
 . . .
}
```

```
Abstract class ActorRef{
  def !(msg: Any)(implicit sender:ActorRef= Actor.noSender): Unit
  def tell(msg:Any, sender:ActorRef)=this.!(msg)(sender)
  . . .
}
```

Example 1 - Simplifying Actor Definition

```
Class Counter extends Actor{
  var count=0
  def receive = {
    case "incr" => count+=1
    case "get"  => sender ! count
  }
}
```

```
Trait Actor{
  implicit val self: ActorRef
 . . .
}
```

```
Abstract class ActorRef{
  def !(msg: Any)(implicit sender:ActorRef= Actor.noSender): Unit
  def tell(msg:Any, sender:ActorRef)=this.!(msg)(sender)
  . . .
}
```

Example 1 - We Still Have State

```
Class Counter extends Actor{
  var count=0
  def receive = {
    case "incr" => count+=1
    case "get"  => sender ! count
  }
}
```

```
Trait Actor{
  implicit val self: ActorRef
 . . .
}
```

```
Abstract class ActorRef{
  def !(msg: Any)(implicit sender:ActorRef= Actor.noSender): Unit
  def tell(msg:Any, sender:ActorRef)=this.!(msg)(sender)
  . . .
}
```

Example 1 - ActorContext

```
Trait Actor{
  implicit val context: ActorContext
  . . .
}
```

```
Trait ActorContext{
  def become(behaviour:Receive, discardOld:Boolean=true): Unit
  def unbecome(): Unit
  . . .
}
```

Example 1 - Stateful Actor Without Explicit State

```
Class Counter extends Actor{
 def counter(n:Int):Receive={
  case "incr" => context.become(counter(n+1))
  case "get"  => sender ! n
 }
 def receive = counter(0)
}
```

```
Trait Actor{
  implicit val context: ActorContext
  . . .
}
```

```
Trait ActorContext{
  def become(behaviour:Receive, discardOld:Boolean=true): Unit
  def unbecome(): Unit
  . . .
}
```

Example 1 - Some More of The ActorContext

```
Trait ActorContext{
  def become(behaviour:Receive, discardOld:Boolean=true): Unit
  def unbecome(): Unit

  def actorOf(p:Props, name:String): ActorRef
  def stop(a:ActorRef): Unit
  . . .
}
```

Example 1 - Putting It Together

```
Class Application extends Actor{
 val counter= context.actorOf(Props[Counter], “counter”)
 counter ! “incr”
 counter ! “incr”
 counter ! “get”
 def receive = {
  case count:Int =>
       println(“count is ”+ count)
       context.stop(self)
 }
}
```

Example 2 - Approximating π using Leibniz formula

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} \quad = \quad \pi / 4$$

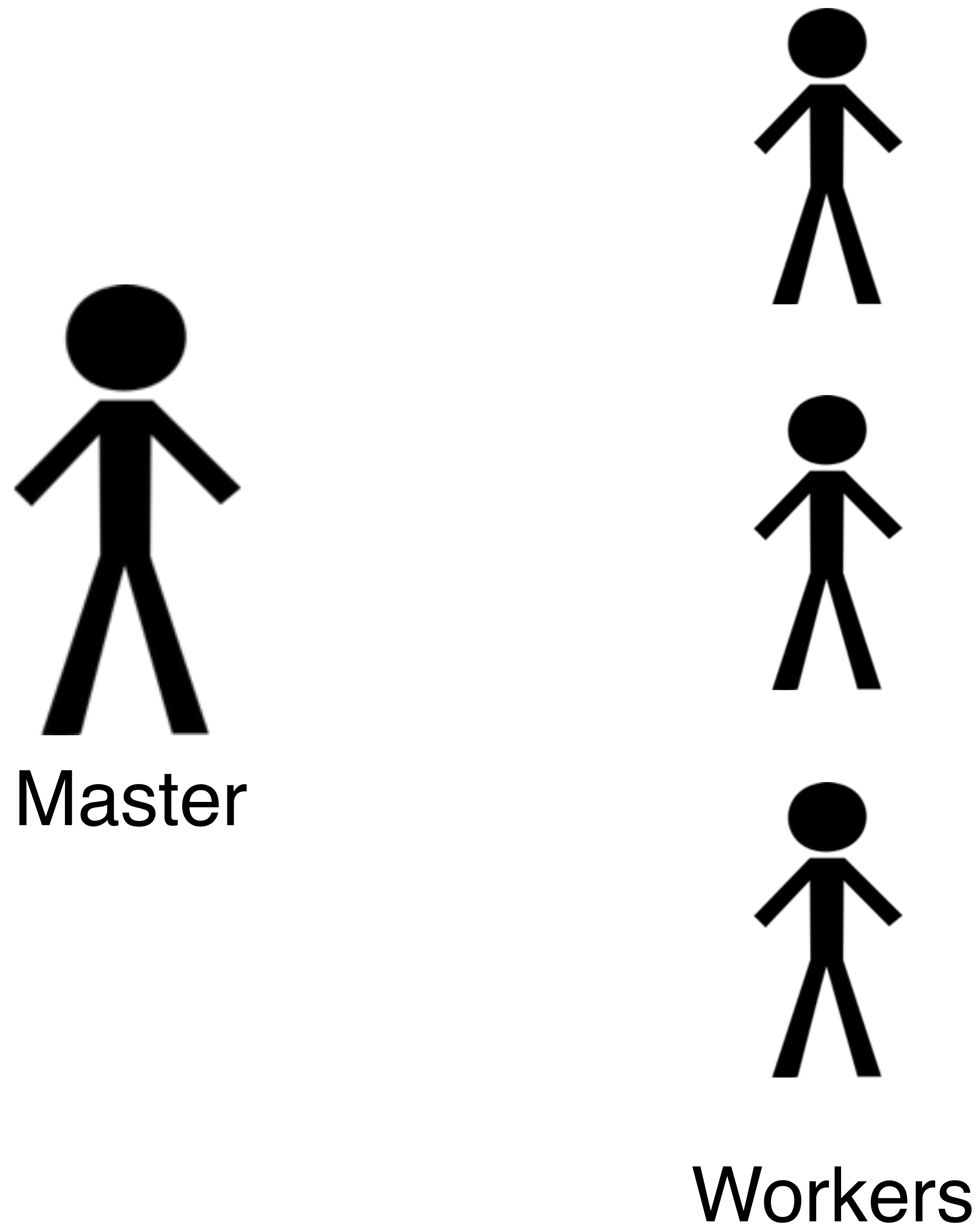
$$\pi = 4 * (1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13 - 1/15 + \ldots)$$

Example 2 - Approximating π using Leibniz formula

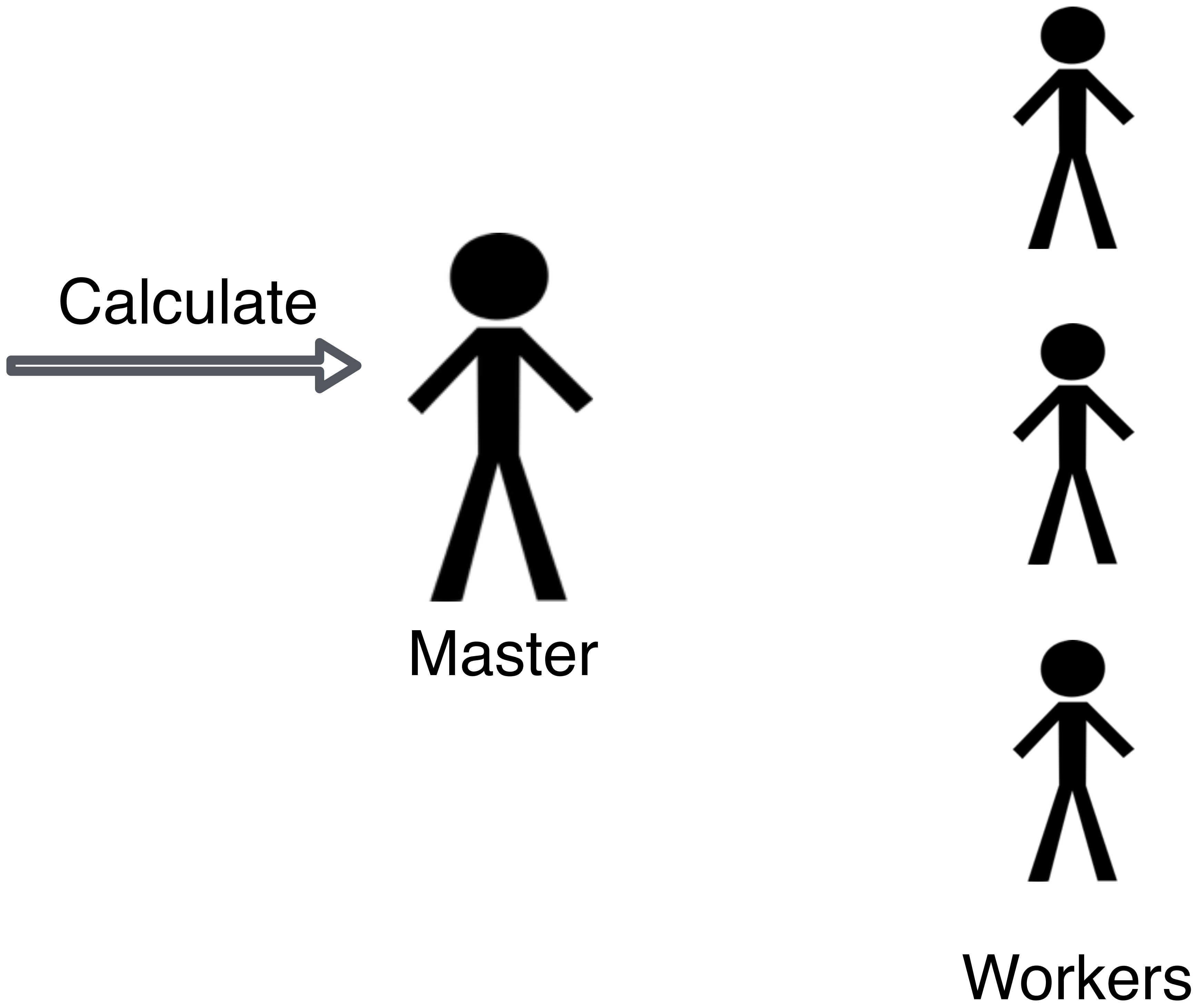
$$\sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = \pi / 4$$

$$\pi = 4 * (1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13 - 1/15 + \ldots)$$

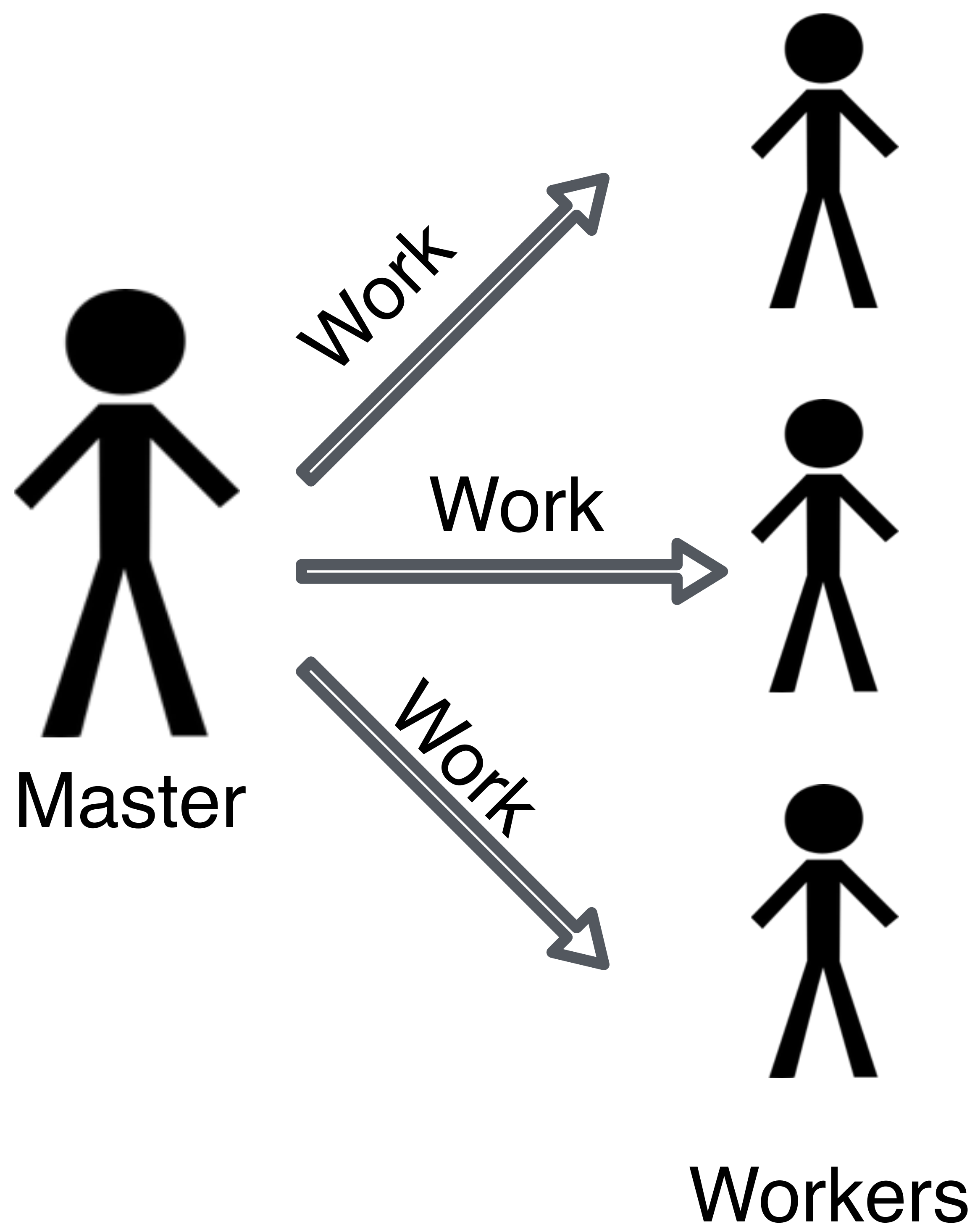
Example 2 - Designing The Actor System



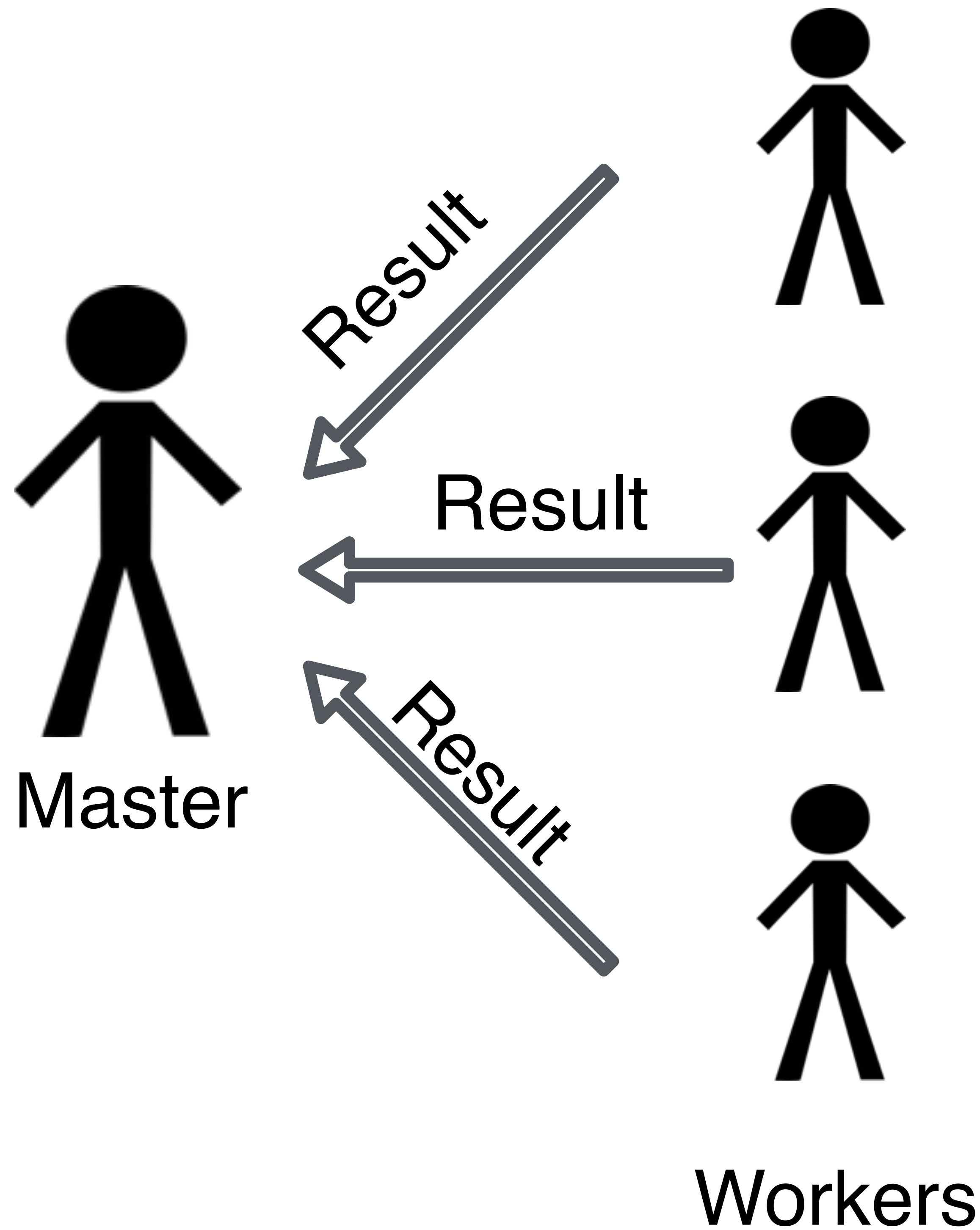
Example 2 - Designing The Actor System



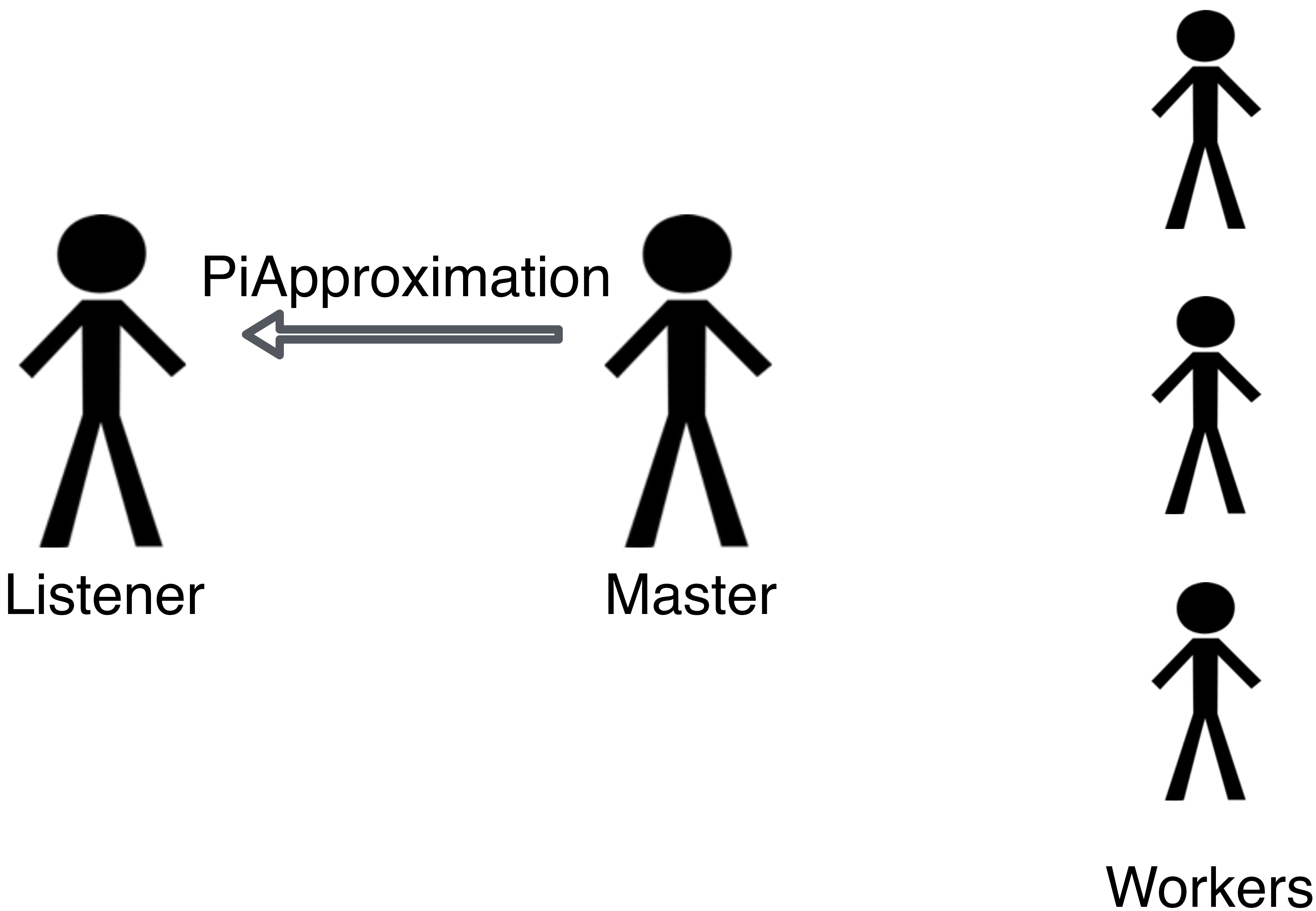
Example 2 - Designing The Actor System



Example 2 - Designing The Actor System



Example 2 - Designing The Actor System

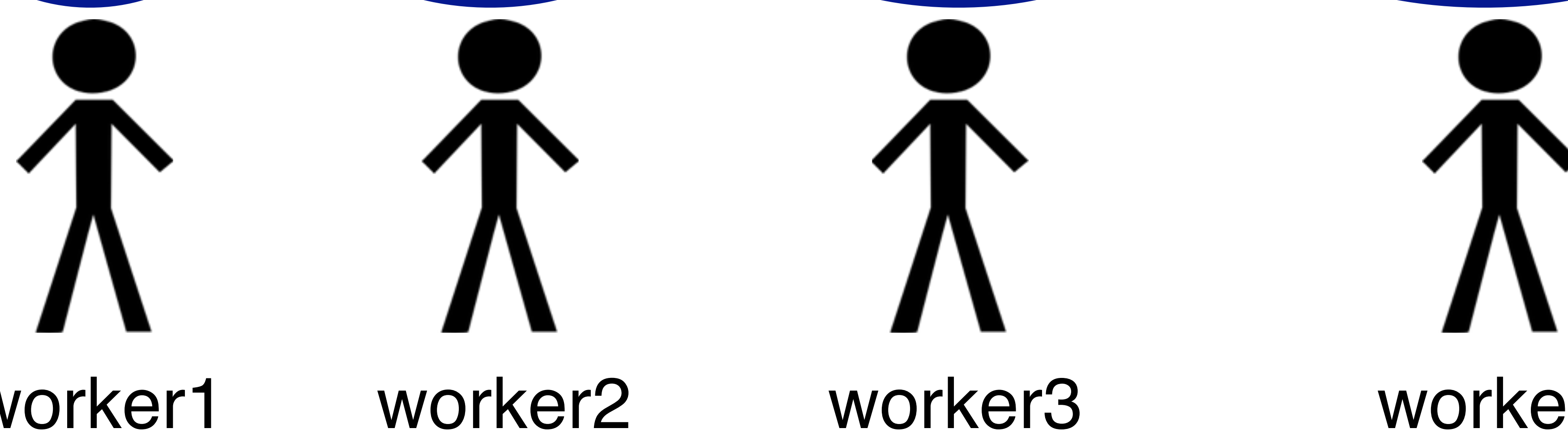


Example 2 - Approximating π using Leibniz formula

$$\pi = 4 * (1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13 - 1/15 + \ldots)$$

Example 2 - Approximating π using Leibniz formula

$$\pi = 4 * (1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13 - 1/15 + \ldots)$$

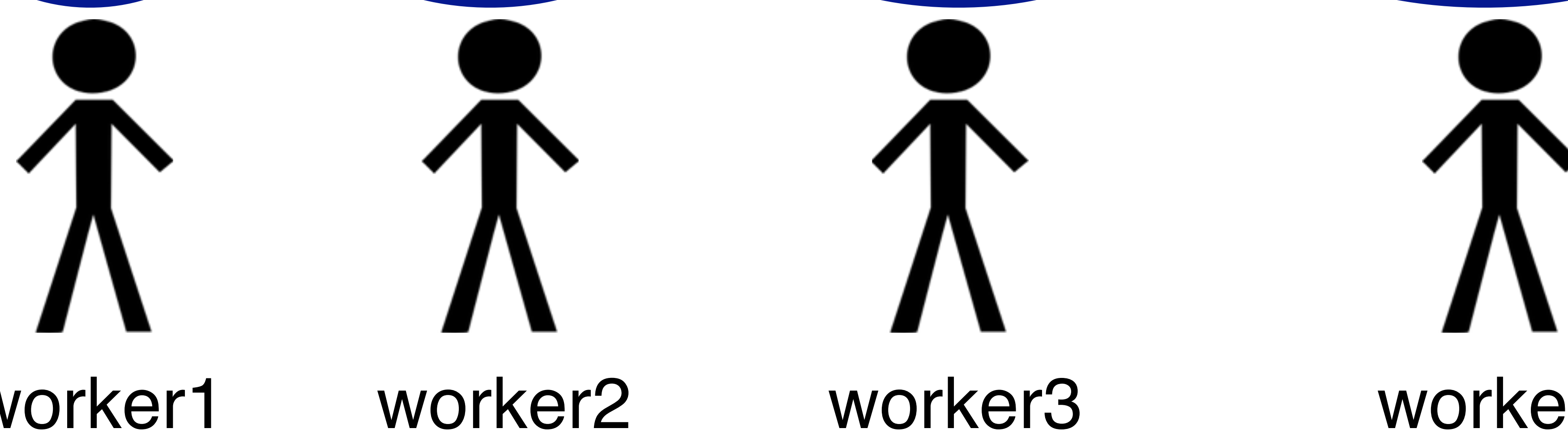


Example 2 - Approximating π using Leibniz formula

System configurations:

- Number of workers
- Number of messages
- Number of elements per message

$$\pi = 4 * (1 - 1/3 + 1/5 - 1/7 + 1/9 - 1/11 + 1/13 - 1/15 + \ldots)$$



Example 2 - Messages

```scala
sealed trait PiMessage
case object Calculate extends PiMessage
case class Work(start: Int, nrOfElements: Int) extends PiMessage
case class Result(value: Double) extends PiMessage
case class PiApproximation(pi: Double, duration: Duration)
```

Example 2 - Worker

```scala
class Worker extends Actor {

  def calculatePiFor(start: Int, nrOfElements: Int): Double ={
    var acc = 0.0
    for (i <- start until (start + nrOfElements))
      acc += 4.0 * (1 - (i % 2) * 2) / (2 * i + 1)
    acc
  }

  def receive = {
    case Work(start, nrOfElements)=>
      sender ! Result(calculatePiFor(start, nrOfElements))
  }
}
```

Example 2 - Master 1/2

```scala
class Master(nrOfWorkers: Int,
             nrOfMessages: Int,
             nrOfElements: Int,
             listener: ActorRef) extends Actor {

  var pi: Double = 0
  var nrOfResults: Int = 0
  var start: Long = _
  val workerRouter = context.actorOf(
    Props[Worker].withRouter(RoundRobinRouter(nrOfWorkers)),
    name = "workerRouter")
```

Example 2 - Master 2/2

```scala
def receive = {
  // handle messages ...
  case Calculate =>
    start= System.currentTimeMillis
    for (i <- 0 until nrOfMessages)
      workerRouter ! Work(i * nrOfElements, nrOfElements)
  case Result(value) =>
    pi += value
    nrOfResults += 1
    if (nrOfResults == nrOfMessages) {
      // Send the result to the listener
      val currentTime=System.currentTimeMillis
      val executionTime=Duration.create(currentTime - start, TimeUnit.MILLISECONDS)
      listener ! PiApproximation(pi, executionTime)
      // Stops this actor and all its supervised children
      context.stop(self)
    }
}
```

Example 2 - Listener

```
class Listener extends Actor {
  def receive = {
    case PiApproximation(pi, duration) ⇒
      println("\n\tPi approximation: \t\t%s\n\tCalculation time: \t%s"
        .format(pi, duration))
      context.system.shutdown()
  }
}
```

Example 2 - Putting It Together

```scala
object Pi extends App {

  calculate(nrOfWorkers = 4, nrOfElements = 10000, nrOfMessages = 10000)

  // actors and messages ...

  def calculate(nrOfWorkers: Int, nrOfElements: Int, nrOfMessages: Int) {
    // Create an Akka system
    val system = ActorSystem("PiSystem")

    // create the result listener, which will print the result and shutdown the system
    val listener = system.actorOf(Props[Listener], name = "listener")

    // create the master
    val master = system.actorOf(Props(new Master(
      nrOfWorkers, nrOfMessages, nrOfElements, listener)),
      name = "master")

    // start the calculation
    master ! Calculate
  }
}
```

With actors …

> It is easier to build scalable software
  - Scalability is a deployment problem (Scaling up and out)

> It is a convenient abstraction for modeling
  - Similarly to OOP

> Fault tolerance can be implemented on top
  - Actors can fail and get replaced at run time

References

>The original actor paper [Hewitt73]:

"[A universal modular ACTOR formalism for artificial intelligence]"

> Hewitt explaining actors:

https://channel9.msdn.com/Shows/Going+Deep/Hewitt-Meijer-and-Szyperski-The-Actor-Model-everything-you-wanted-to-know-but-were-afraid-to-ask

> Scala Actors Course:

Part1: https://www.youtube.com/watch?v=SacX6bjQNRM

Part2: https://www.youtube.com/watch?v=6q7-EzcY7aA

Part3: https://www.youtube.com/watch?v=j25qQxM1F6k

>Akka Tutorial:

http://doc.akka.io/docs/akka/2.0.2/intro/getting-started-first-scala.html

>Scaling up and out with actors:

https://www.youtube.com/watch?v=3jbqTxstIC4