Practical Concurrent and Parallel Programming XII

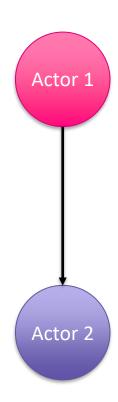# Message Passing II

Raúl Pardo

Agenda

- Actors model (revisited)
  - Primer
- Dynamic topology
- Fault-tolerance
  - Supervision
- Adaptive load balancing
  - Scatter-Gather
- Changing behaviour

What is an Actor? (Bird's eye, revisited)

· 3

- An actor can be seen as a sequential unit of computation
  - Although, formally, the model allows for parallelism within the actor, one can safely assume that there are not concurrency issues within the actor.
  - You can think of an actor as a thread

- Actors can send messages to other actors

Actor 1

Actor 2

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

Actor – Specification (revisited)

- An actor is an abstraction of a thread (intuitively)

- An actors can only execute any of these 4 actions
    1. _Receive messages from other actors_
    2. _Send asynchronous messages_ to other actors
    3. _Create new actors_
    4. _Change its behaviour_ (local state and/or message handlers)

- Actors _do not share memory_
    - They only have access to:
    – Their _local state_ (local memory)
    – Their _mailbox_ (multiset of fixed size with received messages)
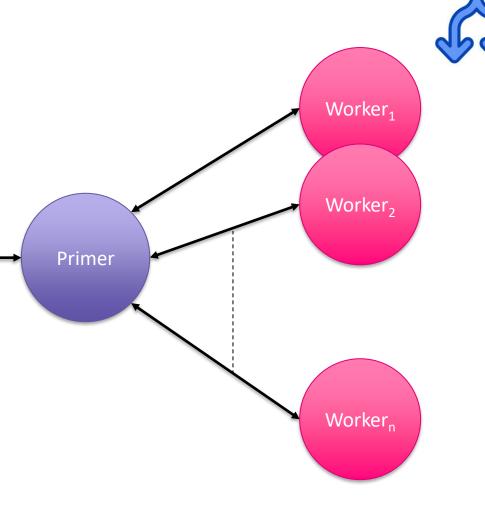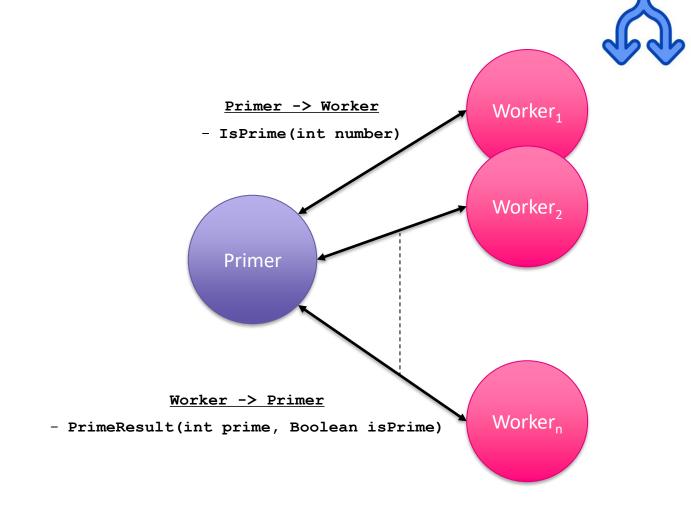   – By default, the mailbox is of unbounded size

		M2	M1

Actor 1

- There is a one-to-one correspondence of the basic actor operations and the Akka API
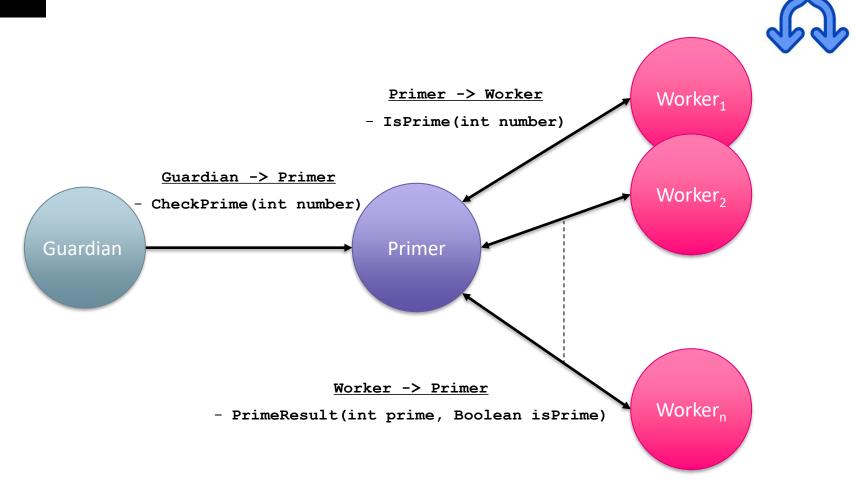
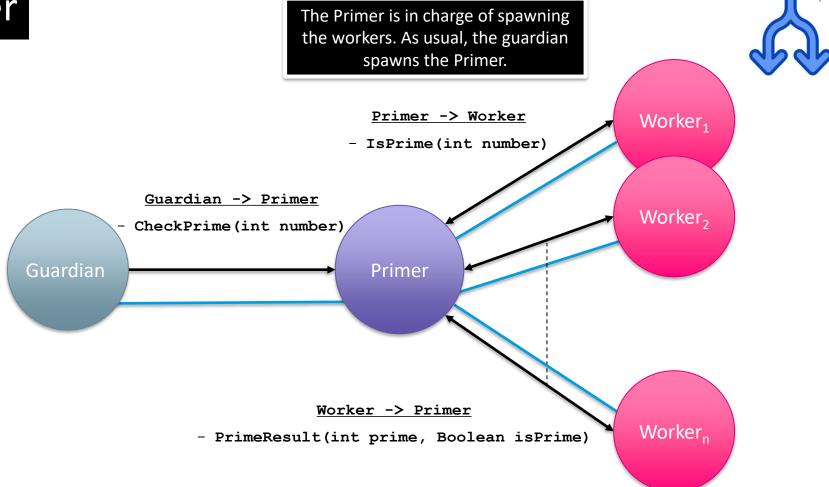| Actors Model	Akka
Actor	Actor class (AbstractBehaviour)
Mailbox Address	Reference to Actor class
Message	Message static final class
State	Actor class local attributes
Behaviour	Handler functions in the Actor class
Create actor	API function
Send message	API function
Receive message	Message handler builder (from API)

Primer

- Consider a Primer actor that receives numbers and checks whether they are prime
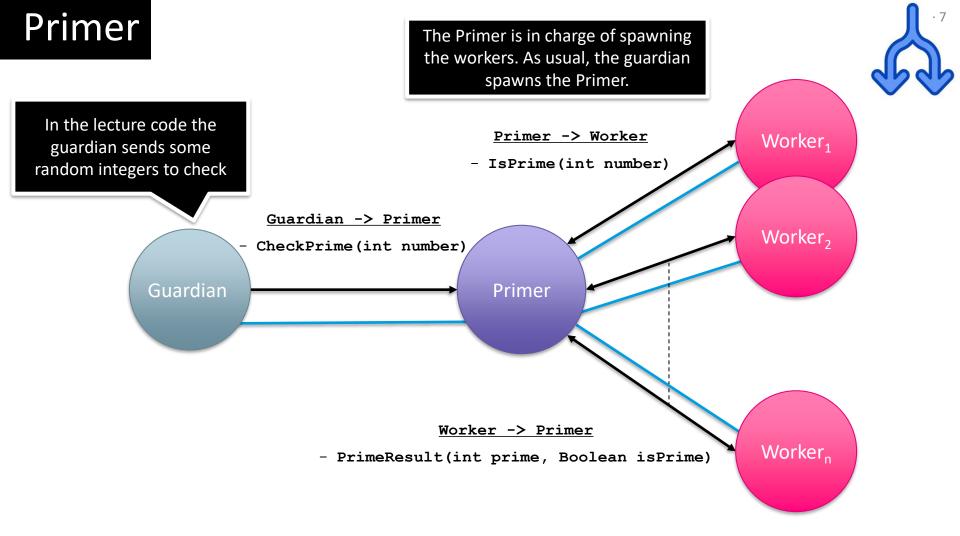
- The actor uses a (fixed) set of worker actors to which it forwards the numbers so that several primes are checked in parallel

Worker$_1$

Worker$_2$

Primer

Worker$_n$

Primer -> Worker

- **IsPrime(int number)**

Worker$_1$

Worker$_2$

Primer

Worker -> Primer

- **PrimeResult(int prime, Boolean isPrime)**

Worker$_n$

Primer -> Worker
- **IsPrime(int number)**

Worker$_1$

Worker$_2$

Guardian -> Primer
- **CheckPrime(int number)**

Guardian

Primer

Worker -> Primer
- **PrimeResult(int prime, Boolean isPrime)**

Worker$_n$

Primer

The Primer is in charge of spawning the workers. As usual, the guardian spawns the Primer.

Primer -> Worker

– `IsPrime(int number)`

Guardian -> Primer

– `CheckPrime(int number)`

Guardian

Primer

Worker$_1$

Worker$_2$

Worker$_n$

Worker -> Primer

– `PrimeResult(int prime, Boolean isPrime)`

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

Primer

The Primer is in charge of spawning the workers. As usual, the guardian spawns the Primer.

In the lecture code the guardian sends some random integers to check

Primer -> Worker

– **IsPrime(int number)**

Worker$_1$

Guardian -> Primer

– **CheckPrime(int number)**

Worker$_2$

Guardian

Primer

Worker -> Primer

– **PrimeResult(int prime, Boolean isPrime)**

Worker$_n$

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

Primer – execution example

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023
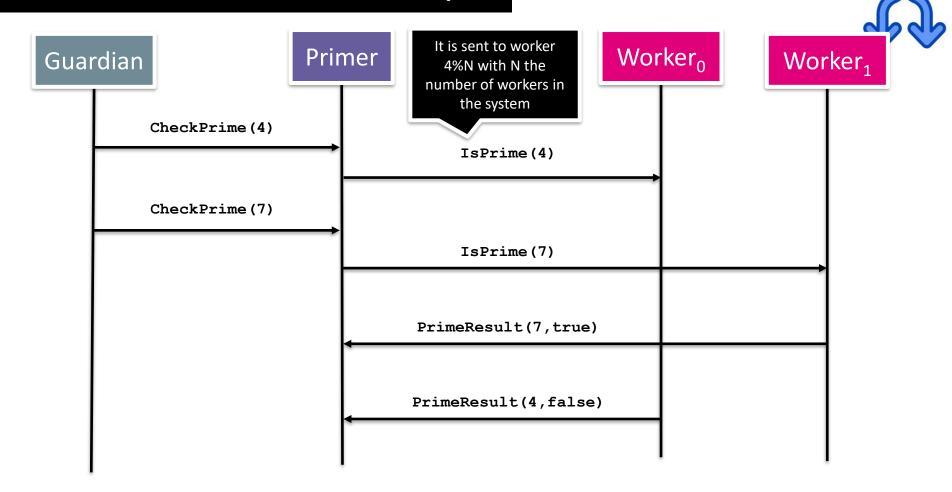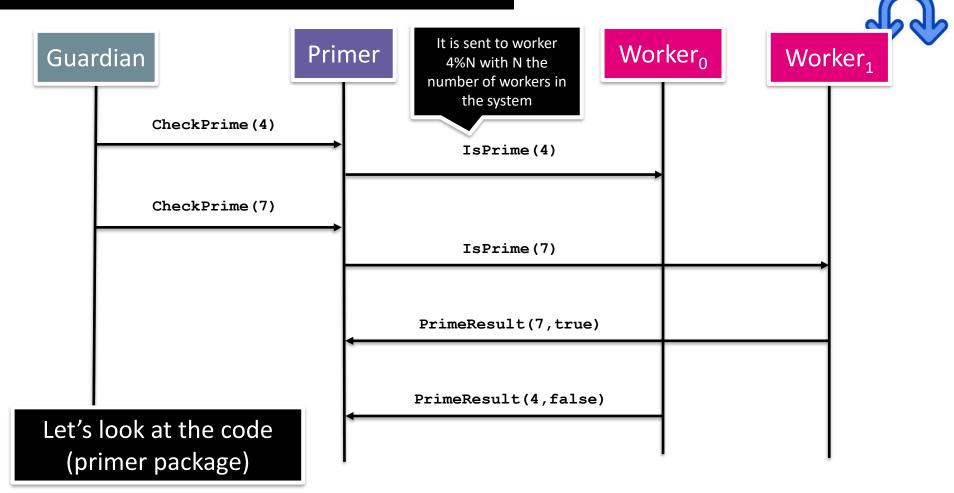
Primer – Printing order

- Note that the printing order of the results does not correspond to the order of sending the requests

```
pardo@pardo-work:week10lecture$ gradle run -PmainClass=primer.Main

> Task :app:run
[primer_system-akka.actor.default-dispatcher-3] INFO akka.event.slf4j.Slf4jLogger - Slf4jLogger started
>>> Press ENTER to exit <<<
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Server and workers started
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 21098598 is prime by worker worker_19
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 1001439 is prime by worker worker_20
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 47257026 is prime by worker worker_7
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 40857223 is prime by worker worker_4
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 10667083 is prime by worker worker_4
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 1001439 is not prime. [1/5]
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 21098598 is not prime. [2/5]
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 47257026 is not prime. [3/5]
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 40857223 is not prime. [4/5]
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 10667083 is not prime. [5/5]
```

Primer – Printing order

- Note that the printing order of the results does not correspond to the order of sending the requests

```
pardo@pardo-work:week10lecture$ gradle run -PmainClass=primer.Main

> Task :app:run
[primer_system-akka.actor.default-dispatcher-3] INFO akka.event.slf4j.Slf4jLogger - Slf4jLogger started
>>> Press ENTER to exit <<<
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Server and workers started
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 21098598 is prime by worker worker_19
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 1001439 is prime by worker worker_20
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 47257026 is prime by worker worker_7
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 40857223 is prime by worker worker_4
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 10667083 is prime by worker worker_4
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 1001439 is not prime. [1/5]
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 21098598 is not prime. [2/5]
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 47257026 is not prime. [3/5]
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 40857223 is not prime. [4/5]
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 10667083 is not prime. [5/5]
```
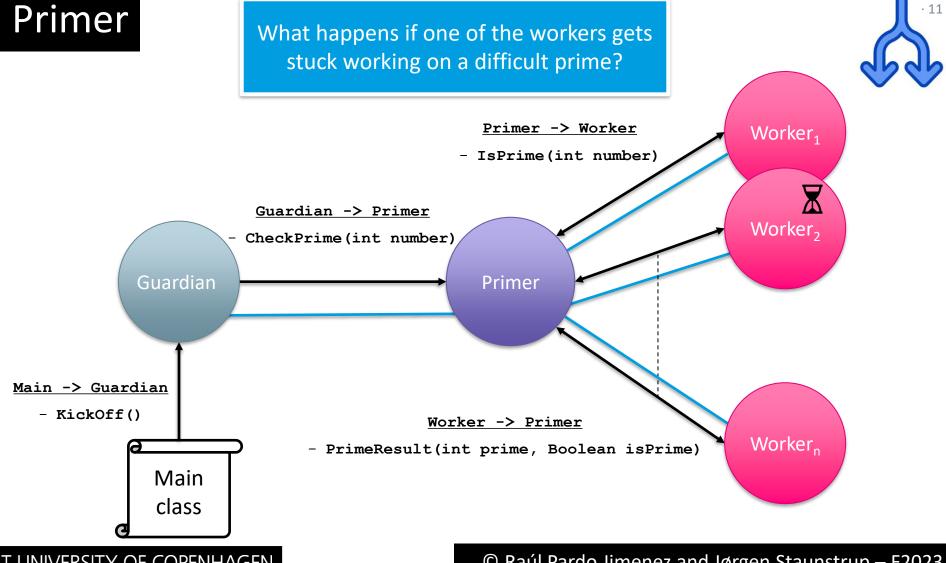
How can this ordering happen?

- Note that the printing order of the results does not correspond to the order of sending the requests

```
pardo@pardo-work:week10lecture$ gradle run -PmainClass=primer.Main

> Task :app:run
[primer_system-akka.actor.default-dispatcher-3] INFO akka.event.slf4j.Slf4jLogger - Slf4jLogger started
>>> Press ENTER to exit <<<
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Server and workers started
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 21098598 is prime by worker worker_19
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 1001439 is prime by worker worker_20
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 47257026 is prime by worker worker_7
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 40857223 is prime by worker worker_4
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Cheking whether 10667083 is prime by worker worker_4
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 1001439 is not prime. [1/5]
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 21098598 is not prime. [2/5]
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 47257026 is not prime. [3/5]
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 40857223 is not prime. [4/5]
[primer_system-akka.actor.default-dispatcher-3] INFO primer.Primer - primer_server: Number 10667083 is not prime. [5/5]
```

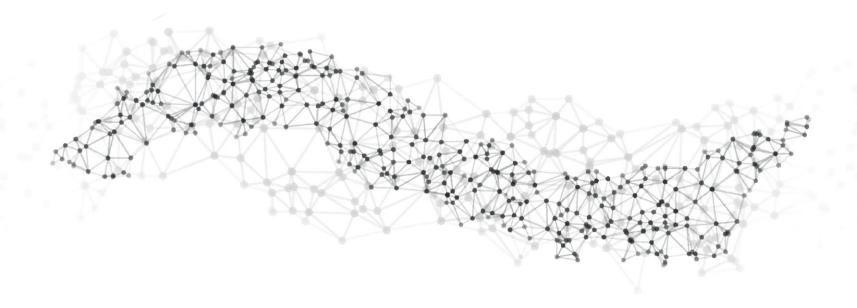How would you change the system to print the results in the same order as they arrived?

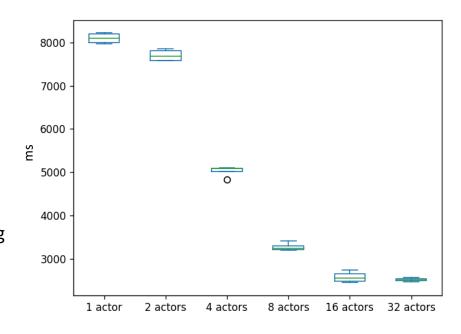© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

Actors systems with _dynamic_ topology

- The Actors model encourages creating many actors that perform small tasks and communicate with each other

- As usual, performance depends on the hardware

- These are the results of running the primer to check 1 million numbers between 1 billion and Integer.MAX_VALUE
  - Not very strong statistics (4 runs for each number of actors)

- Akka implements actors systems using ForkJoinPools

- However, actor systems can be distributed among many JVMs and computers
  - We are not limited to a single computer throughput
  - See Akka cluster

That said, distributing computation among actors makes it easy to implement fault-tolerant systems and adaptive load-balancing
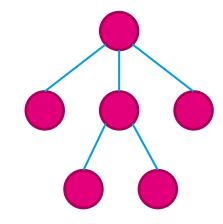
- Akka implements actors systems using ForkJoinPools

- However, actor systems can be distributed among many JVMs and computers
  - We are not limited to a single computer throughput
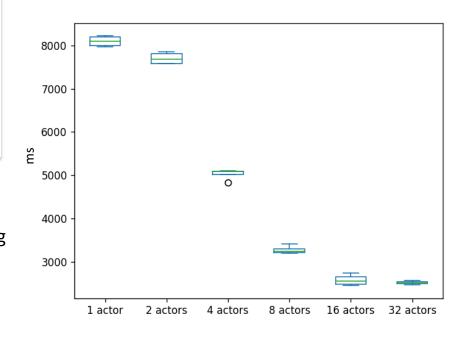 - See Akka cluster

- We use the term *topology* to refer to the parental structure of actors in the system

  - In Akka, this structure is a tree, and it is called a *hierarchy*

- The systems we have seen so far feature a *static* topology

  - All the actors in the system are spawned during initialization

Solid lines and actors represent elements that are created during initialization and never change

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023
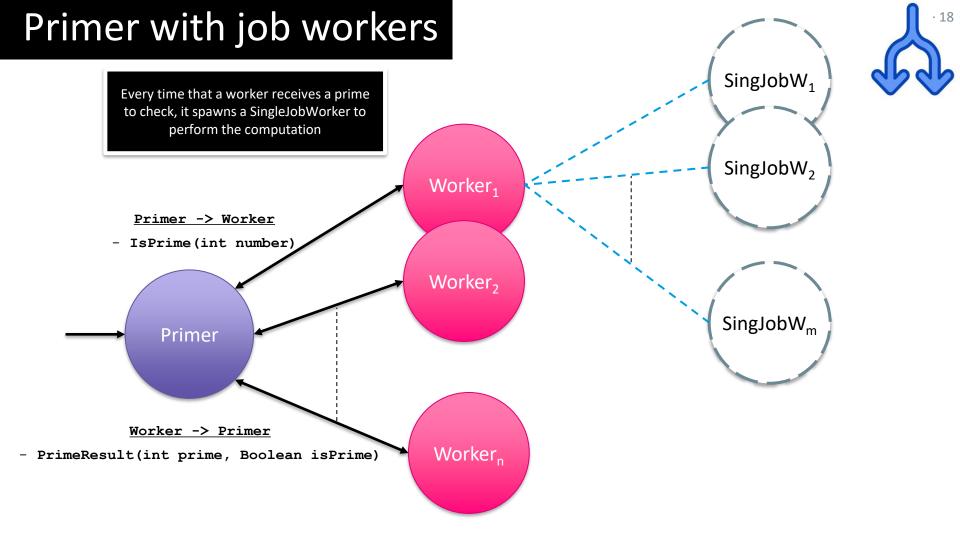
- Actor systems with static topology may not exploit computational resources effectively
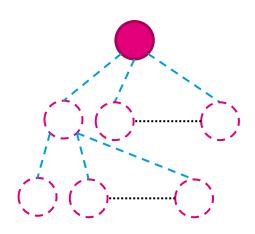 - As we saw, the system may slow down if some actors are consuming excessive computational resources
  - Actors may also crash, and the system should be able to recover from this (fault-tolerance)

- The advantages of the actors model are better exploited when the system can adaptively decide the number of workers

- Actors should be seen as *nice co-workers*
  - *A group of computational resources that collaborate to achieve a common goal*

Dashed lines and actors represent elements that may be created dynamically (on-demand, after initialization)

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

- To avoid excessive delays by primes that are difficult to check, we extend the system with dedicated actors whose only task is to check the prime

- After these dedicated actors have finished the computation they report the result and terminate the execution

Primer with job workers

Every time that a worker receives a prime to check, it spawns a SingleJobWorker to perform the computation

Primer -> Worker

- **IsPrime(int number)**

Primer

Worker$_1$

Worker$_2$

Worker$_n$

SingJobW$_1$

SingJobW$_2$

SingJobW$_m$

Worker -> Primer

- **PrimeResult(int prime, Boolean isPrime)**

Primer with job workers

```
Worker -> SingJobW
- IsPrime(int number,
          ActorRef server)
```

After begin spawned, the job is forwarded to the single job worker

SingJobW$_1$

SingJobW$_2$

SingJobW$_m$

Worker$_1$

Worker$_2$

Worker$_n$

```
Primer -> Worker
- IsPrime(int number)
```

Primer

```
Worker -> Primer
- PrimeResult(int prime, Boolean isPrime)
```

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

Primer with job workers

After begin spawned, the job is forward to the sing job worker

Worker -> SingJobW

- **IsPrime(int number,**
   ActorRef server)

Primer -> Worker

- **IsPrime(int number)**

SingJobW -> Primer

- **PrimeResult(int prime,**
   Boolean isPrime)

Worker -> Primer

- **PrimeResult(int prime, Boolean isPrime)**

The single job worker sends the result to the primer directly. Note that the message from worker to primer is unnecessary now.

Primer with job workers

Worker -> SingJobW
- **IsPrime(int number,**
  ActorRef server)

After begin spawned, the job is forward to the sing job worker

SingJobW₁

SingJobW₂

Worker₁

Primer -> Worker
- **IsPrime(int number)**

Would there be any problem if we send the message to the parent worker instead? (and it forwards it to the primer?)

Worker₂

SingJobWₘ

Primer

SingJobW -> Primer
- **PrimeResult(int prime,**
  Boolean isPrime)

Worker -> Primer
- PrimeResult(int prime, Boolean isPrime)

Workerₙ

The single job worker sends the result to the primer directly. Note that the message from worker to primer is unnecessary now.

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

Primer with job workers

```
Worker -> SingJobW
- IsPrime(int number,
         ActorRef server)
```

SingJobW$_1$

SingJobW$_2$

After begin spawned, the job is forward to the sing job worker

Worker$_1$

SingJobW$_m$

```
Primer -> Worker
- IsPrime(int number)
```

Worker$_2$

Primer

```
SingJobW -> Primer
- PrimeResult(int prime,
             Boolean isPrime)
```

Worker$_n$

After finishing the computation the single job worker terminates the execution. The symbol ⏻ means normal termination
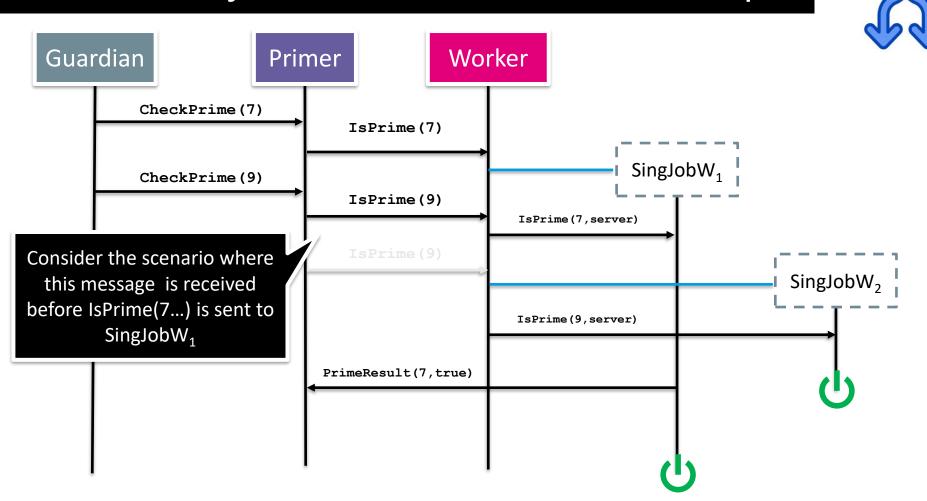
© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

- The Akka API has a specific method to shutdown an actor
  - `Behaviours.stopped()`

- This behaviour can be used as the return object of a message handler
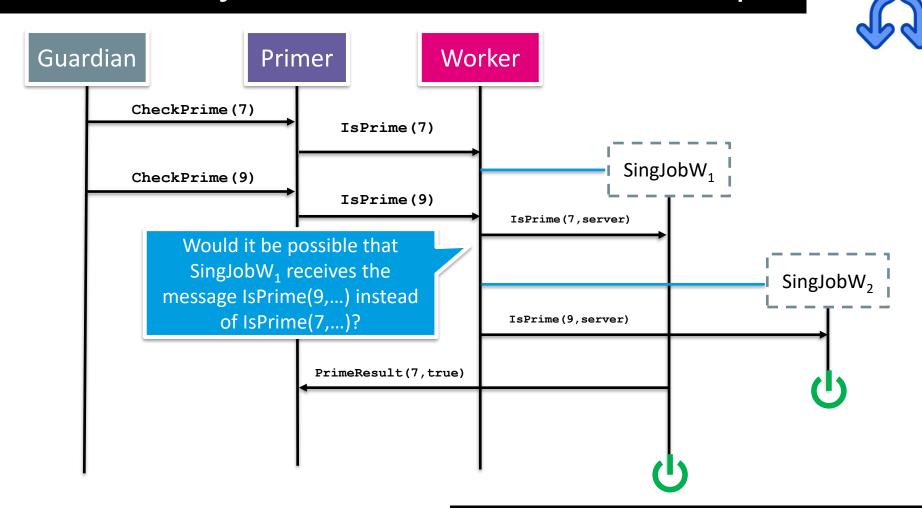
```
public Behavior<IsPrime> onIsPrime(IsPrime msg) {
    msg.server.tell(new Primer.PrimeResult(msg.number, isPrime(msg.number)));
    return Behaviors.stopped();
}
```
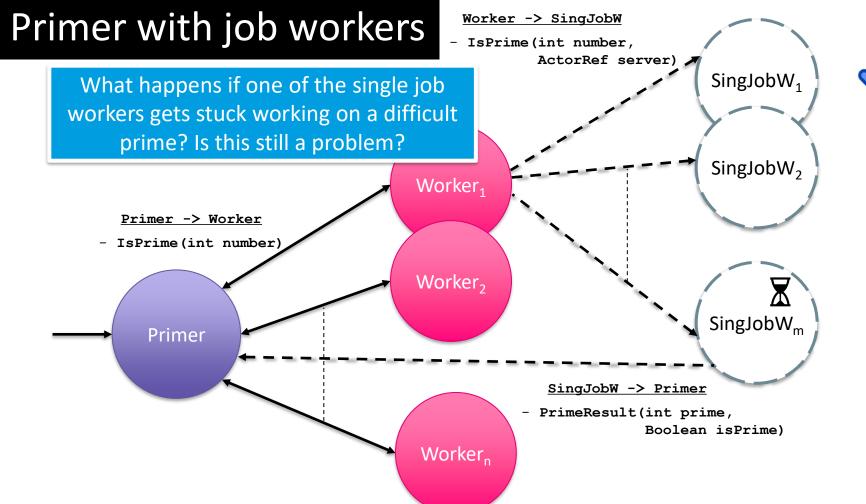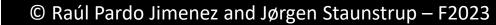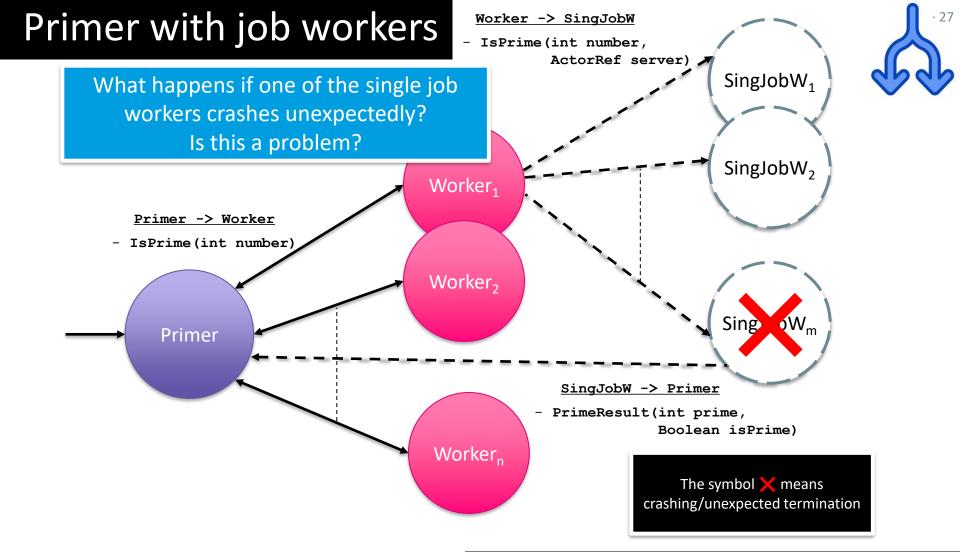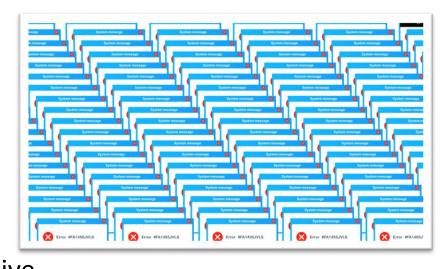
© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

Primer with job workers – execution example

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

Primer with job workers

What happens if one of the single job workers gets stuck working on a difficult prime? Is this still a problem?

Worker -> SingJobW
- `IsPrime(int number, ActorRef server)`

Primer -> Worker
- `IsPrime(int number)`

SingJobW_1

SingJobW_2

SingJobW_m

Worker_1

Worker_2

Worker_n

Primer

SingJobW -> Primer
- `PrimeResult(int prime, Boolean isPrime)`

Primer with job workers

What happens if one of the single job workers crashes unexpectedly?
Is this a problem?

Primer -> Worker

- `IsPrime(int number)`

Worker -> SingJobW

- `IsPrime(int number,`
            `ActorRef server)`

SingJobW -> Primer

- `PrimeResult(int prime,`
              `Boolean isPrime)`

Primer

$Worker_1$

$Worker_2$

$Worker_n$

$SingJobW_1$

$SingJobW_2$

$SingJobW_m$

The symbol ✖ means crashing/unexpected termination

Fault-tolerance in Akka

- Actor libraries and programming languages encourage a _let it crash_ programming model

- Do not put a lot of effort ensuring that actors never crash
  - Assume that things will fail

- Develop actors systems ensuring that if an actors crashes the system can recover

- Specially, useful in distributed systems when you cannot predict what type of message you will receive

- Akka implements supervision mechanisms to react to failures

- Children may inform their parents when they terminate or fail

- Actors may use the function `watch(ActorRef<T> actor)` to supervise their children

- If an actor is being supervised by a (parent) watcher, it sends to the watcher

  - A `ChildFailed` signal, if it crashed due to an exception
  - A `Terminated` signal, if it terminates normally

But ChilFailed extends from Terminated

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

- A *signal* can be seen as a message that is automatically sent by Akka
- For a watcher to handle signals, it *must* have a message handler with `onSignal(Signal.class, Function f)`
- The message send by the signal contains a reference to the sender actor. It can be accessed with `getRef()`

```java
/* --- Message handling --------------------------- */
@Override
public Receive<T> createReceive() {
    return newReceiveBuilder()
     .onMessage(Message.class, this::onMessage)
     // Here order matters `ChildFailed extends Terminated`
     .onSignal(ChildFailed.class, this::onChildFailed)
     .onSignal(Terminated.class, this::onTerminated)
     .build();
}
```

Note I: When processing a message/signal, the message handler picks the handler that first matches the class of the message

Note II: Since ChildFailed extends Terminated, if onSignal(Terminated,…) appears before onSignal(ChildFailed,…), when a ChildFailed signal arrives, the latter onSignal will not be triggered
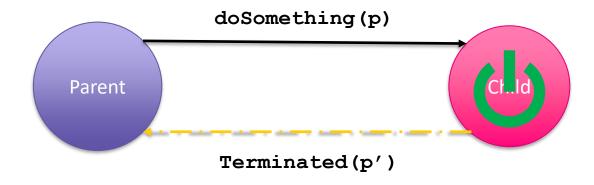
Actor supervision (graphically)

Actor supervision (graphically)

doSomething(p)
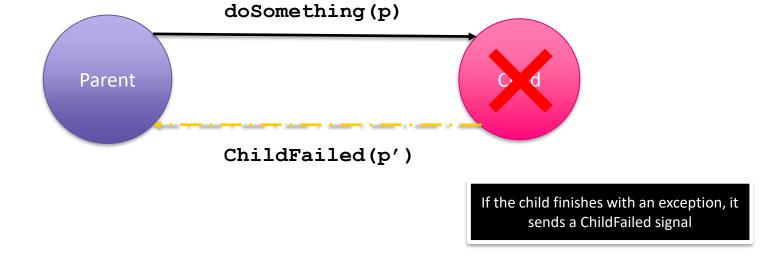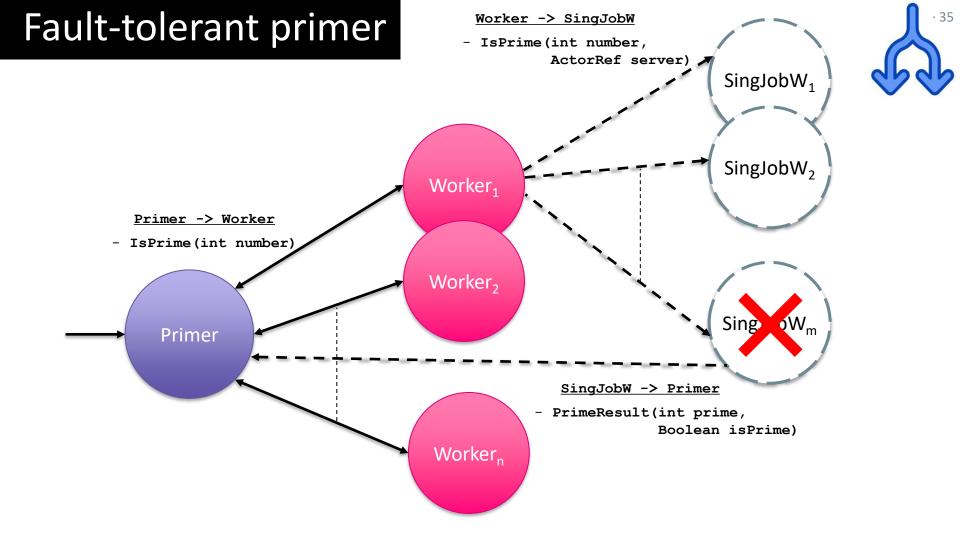
Parent

Ch..ld

Terminated(p')

The dashed-dotted yellow arrow indicates the sending of a signal. These are sent automatically by Akka as part of the supervision functionality. If the child finishes normally, it sends a Terminated signal.
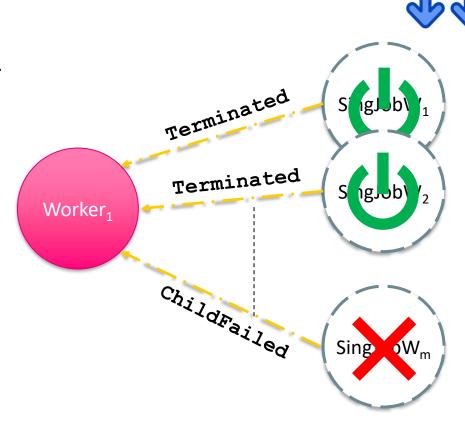
doSomething(p)

Parent

Child

ChildFailed(p')

If the child finishes with an exception, it sends a ChildFailed signal

Fault-tolerant primer

Worker -> SingJobW

- `IsPrime(int number,`
  `        ActorRef server)`

SingJobW$_1$

SingJobW$_2$

SingJobW$_m$

Worker$_1$

Worker$_2$

Worker$_n$

Primer -> Worker

- `IsPrime(int number)`

Primer

SingJobW -> Primer

- `PrimeResult(int prime,`
  `           Boolean isPrime)`

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

Fault-tolerant primer

- We extend the primer to handle the case when a single job worker fails

- To this end, the worker needs to:
1. Watch all the actors it spawns
2. Handle ChildFailed signals
   - The handler spawns a new worker and sends the number again to check whether it is prime
3. Handle Terminated signals
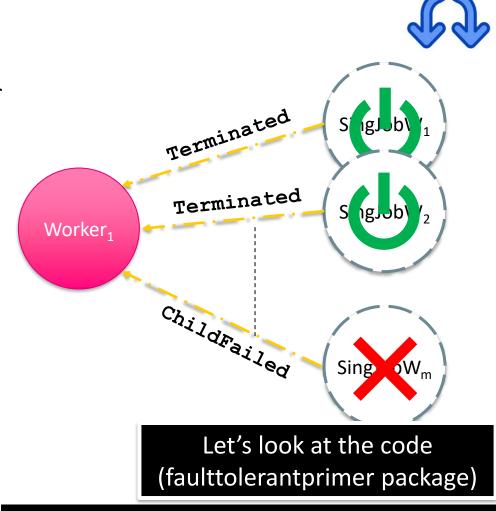   - No more computation needed, we can mark the number as checked

$SingJobW_1$

$SingJobW_2$

$Worker_1$

$SingJobW_m$

Terminated

Terminated

ChildFailed

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

- We extend the primer to handle the case when a single job worker fails

- To this end, the worker needs to:
1. Watch all the actors it spawns
2. Handle ChildFailed signals
   - The handler spawns a new worker and sends the number again to check whether it is prime
3. Handle Terminated signals
   - No more computation needed, we can mark the number as checked

We must extend the actor's state to keep track of what number is computed by what actor

$Worker_1$

$SingJobW_1$ — Terminated

$SingJobW_2$ — Terminated

$SingJobW_m$ — ChildFailed

Fault-tolerant primer

- We extend the primer to handle the case when a single job worker fails

- To this end, the worker needs to:
1. Watch all the actors it spawns
2. Handle ChildFailed signals
   - The handler spawns a new worker and sends the number again to check whether it is prime
3. Handle Terminated signals
   - No more computation needed, we can mark the number as checked

We must extend the actor's state to keep track of what number is computed by what actor

Worker$_1$

SingJobW$_1$ Terminated

SingJobW$_2$ Terminated

SingJobW$_m$ ChildFailed

Let's look at the code
(faulttolerantprimer package)

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

Adaptive load balancing

Adaptive load balancing

- *Load balancing* refers to the process of distributing a set of tasks over a set of resources (computing units), with the aim of making their overall processing more efficient.   [Wikipedia]

- In the (static) primer system, we indiscriminately spawned processes to perform tasks
  - This may cause sending tasks to busy workers while other idle workers could be processing them

- There exists some patterns that aim at distributing computation fairly among actors.
  - For instance, the scatter-gather pattern
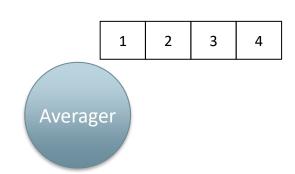
- Scatter-Gather is a common design patter in distributed systems that can be easily implemented with actors

- Typically, the level of scattering (i.e., number of spawned actors) depends on the size of the problem to solve (dynamic load balancing)
  - But it can also be limited by other factors, e.g., CPU or memory usage

- A scatter-gather systems contains two main type of actors
  - Scatterer: if possible, it splits computation in smaller units. Otherwise, it may perform a processing step in the atomic piece of data and send it to a gatherer
  - Gatherer: Receives pieces of data from scatterers or gatherers and combines them into a single piece of data. Then, if it is not the highest-level gatherer, it sends the result to a gatherer
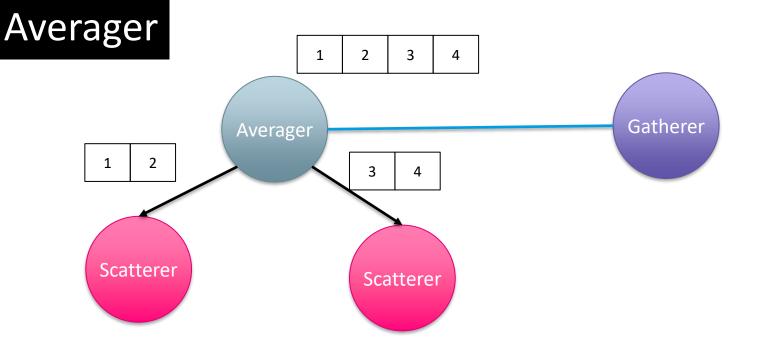
Average and scatter-gather

- A problem for which this pattern is suitable is computing the average of a list of numbers

- Given a set of natural numbers $a_1, a_2, ..., a_n$, the average is $\frac{1}{n}\sum_i a_i$

  - Note that this is equivalent to $\sum_i \frac{a_i}{n}$

- In a nutshell, we can have scatterer actors splitting computation and computing each factor $\frac{a_i}{n}$, and gatherers summing up the results
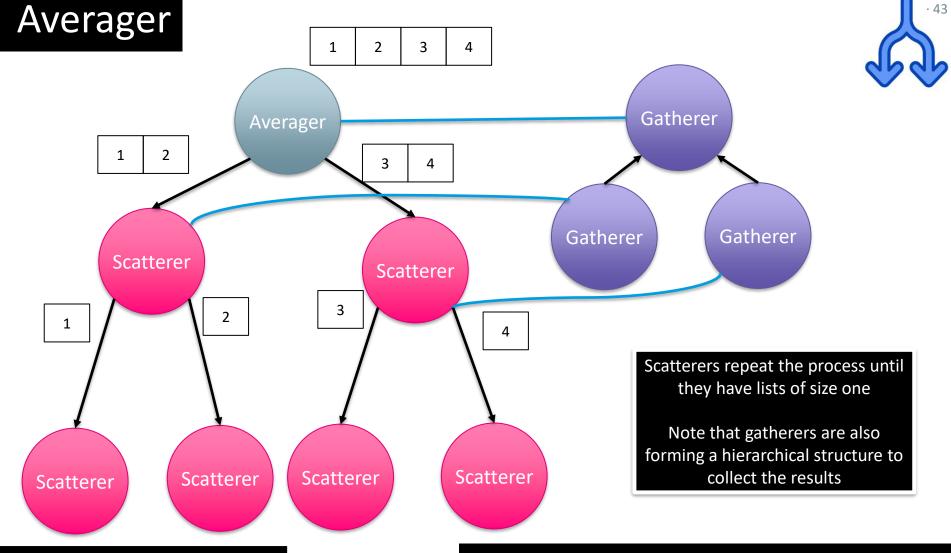
Averager

1	2	3	4

Averager

Consider a system that computes the average of a list of numbers

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

Averager

1	2	3	4

Averager ———————— Gatherer
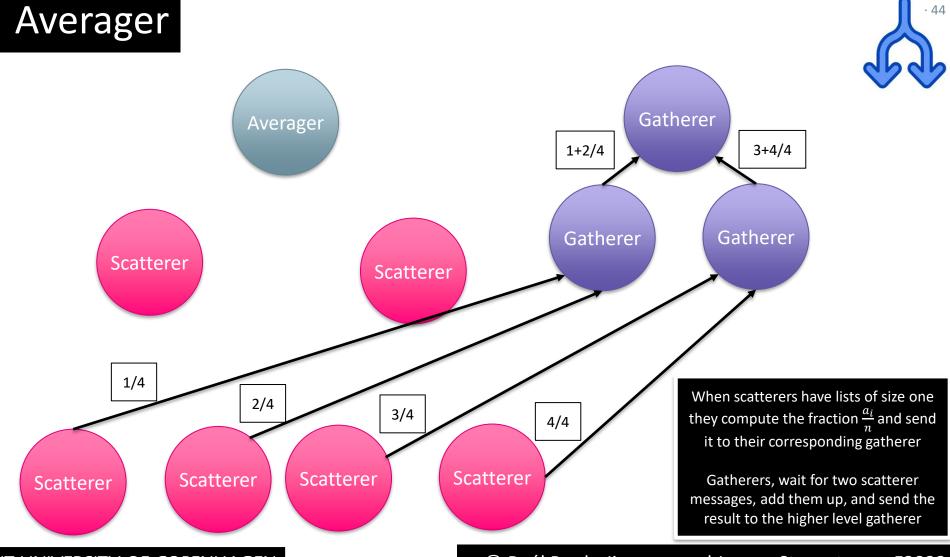
1	2

3	4

Scatterer

Scatterer

In the first step, we split the computation into two sublists, and assign them to separate scatterer workers

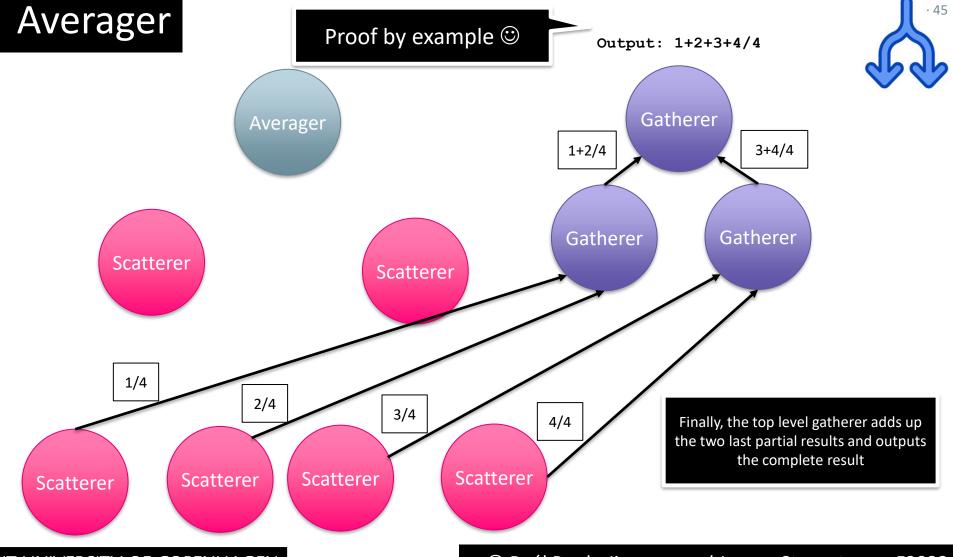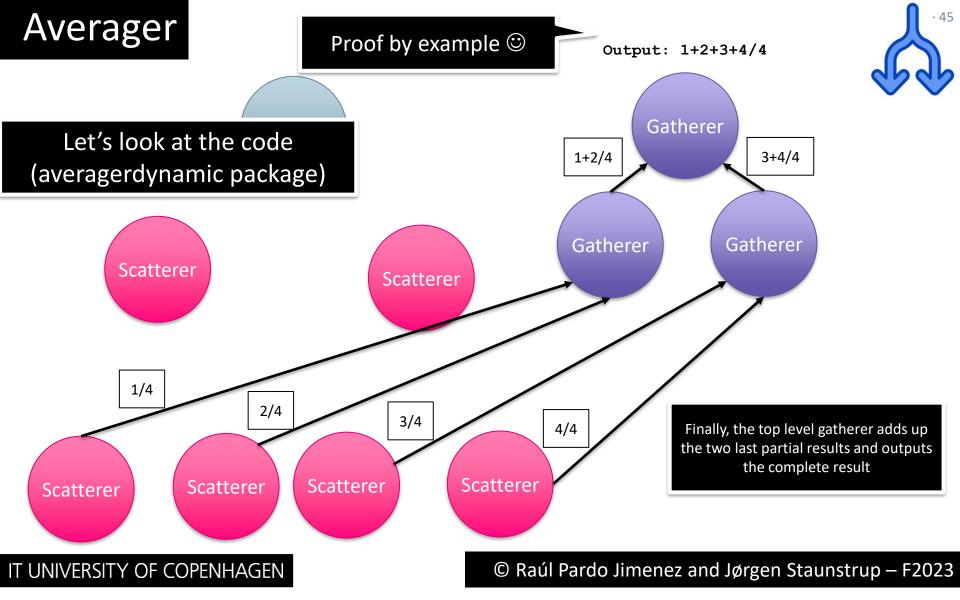Also, we spawn a gatherer worker that will receive merge the average of each sublist

Averager

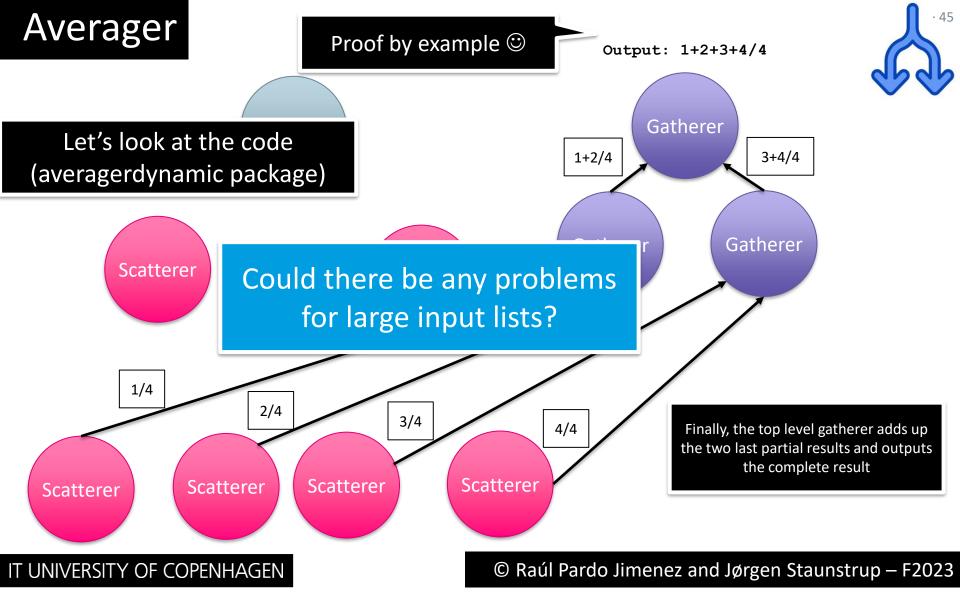Scatterers repeat the process until they have lists of size one

Note that gatherers are also forming a hierarchical structure to collect the results

Averager

Averager

Gatherer

1+2/4

3+4/4

Gatherer

Gatherer

Scatterer

Scatterer

1/4

2/4

3/4

4/4

Scatterer

Scatterer

Scatterer

Scatterer

When scatterers have lists of size one they compute the fraction $\frac{a_i}{n}$ and send it to their corresponding gatherer

Gatherers, wait for two scatterer messages, add them up, and send the result to the higher level gatherer

Averager

Proof by example ☺

`Output: 1+2+3+4/4`

Averager

Gatherer

1+2/4   3+4/4

Gatherer   Gatherer

Scatterer

Scatterer

1/4

2/4

3/4

4/4

Finally, the top level gatherer adds up the two last partial results and outputs the complete result

Scatterer   Scatterer   Scatterer   Scatterer

- The size of the problem does not necessarily need to determine the distribution of computation

- One may have HW restrictions
  - As we saw, actors systems running in a single machine may not scale well beyond the number of processors

- Another example of adaptive load balancing are elastic systems
  - Elastic systems try to keep a number of active actors proportional to the workload
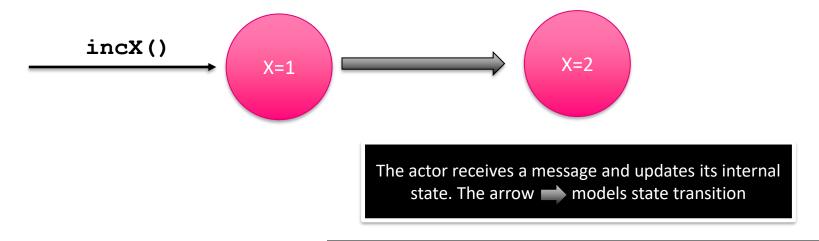 - Several exercises for this week target implementing an elastic server
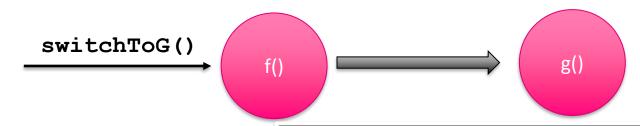
Changing behaviour

- The actors model states that an actor can *"Change its behaviour (local state and/or message handlers)"*

- So far we have considered change in behaviour as change in state

incX()

X=1

X=2

The actor receives a message and updates its internal state. The arrow ➡ models state transition

Actors with changing behaviour

- The actors model states that an actor can *"Change its behaviour (local state and/or <u>message handlers</u>)"*

- However, actors may also change the *functions* to process messages (i.e., message handlers)

switchToG() → f() → g()

In this example, the actor that was executing f() when a message comes, now executes g(). The new function g() could be completely different, e.g., changing how messages are processed or even waiting for different type of messages!

- In Akka, we can change the behaviour of an actor by defining a function that returns the new behaviour

  - Like in the behaviour defined in `createReceive()`

- In fact, we have already done this when we return `Behaviors.stopped()` to terminate an actor

- The packages averagerdynamic and averagerbehavior implement the same system, but the latter uses changing behaviour

- In averagerdynamic, we use a Boolean variable (`receivedFirstNumber`) to determine whether we have received the first or second GathererCommand message

- In averagebehavior, we define a new behaviour (`waitForSecond`) to which the actor transitions after receiving the first GathereCommand message
  - In this way we can do without the Boolean variable mentioned above

- The packages averagerdynamic and averagerbehavior implement the same system, but the latter uses changing behaviour

- In averagerdynamic, we use a Boolean variable (`receivedFirstNumber`) to determine whether we have received the first or second GathererCommand message

- In averagebehavior, we define a new behaviour (`waitForSecond`) to which the actor transitions after receiving the first GathereCommand message
  - In this way we can do without the Boolean variable mentioned above

Let's look at the code
(averagerbehaviors package)

The actors model has natural mapping in distributed systems

Computer 1   Computer 2   Computer 3

Application A

Application B

Middleware (Distributed system layer)

Local OS 1   Local OS 2   Local OS 3

Network
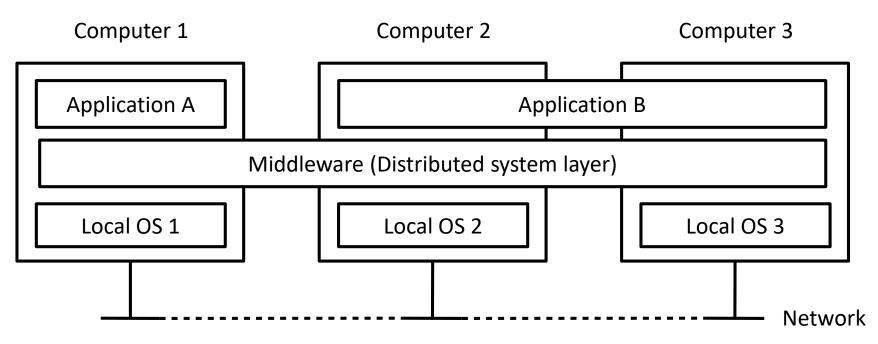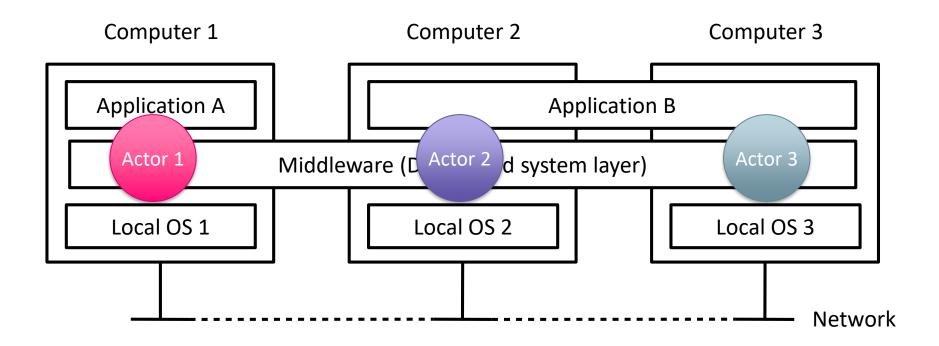
Figure taken from -> Distributed Systems: Principles and Paradigms. Andrew S. Tanenbaum and Maarten Van Steen. 2007.
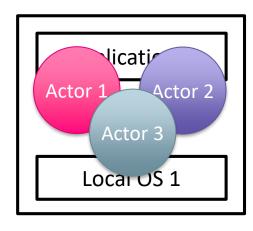
The actors model has natural mapping in distributed systems

The actors model is applicable in a single computer as well

Computer 1

In this course, we focus on this type of actor system

- Actors model (revisited)
  - Primer
- Dynamic topology
- Fault-tolerance
  - Supervision
- Adaptive load balancing
  - Scatter-Gather
- Changing behaviour