Practical Concurrent and Parallel Programming XII

Message Passing II

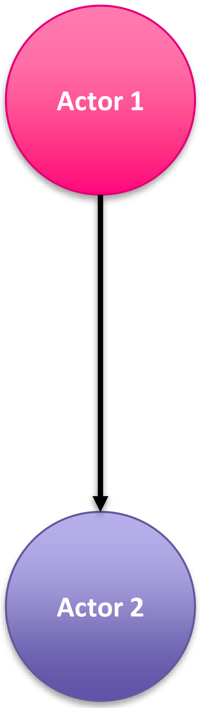
Raúl Pardo

Agenda

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

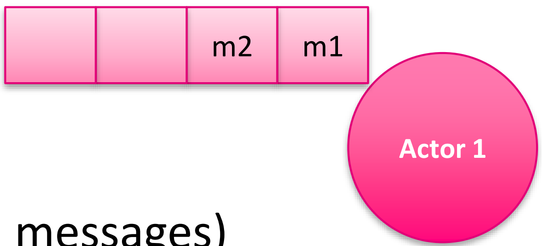
What is an Actor? (Bird's eye, revisited)

- 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 – 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



Producer-consumer problem | Intuition

- Perhaps more intuitive example

Consumers

Producers

Shared data structure of fixed size

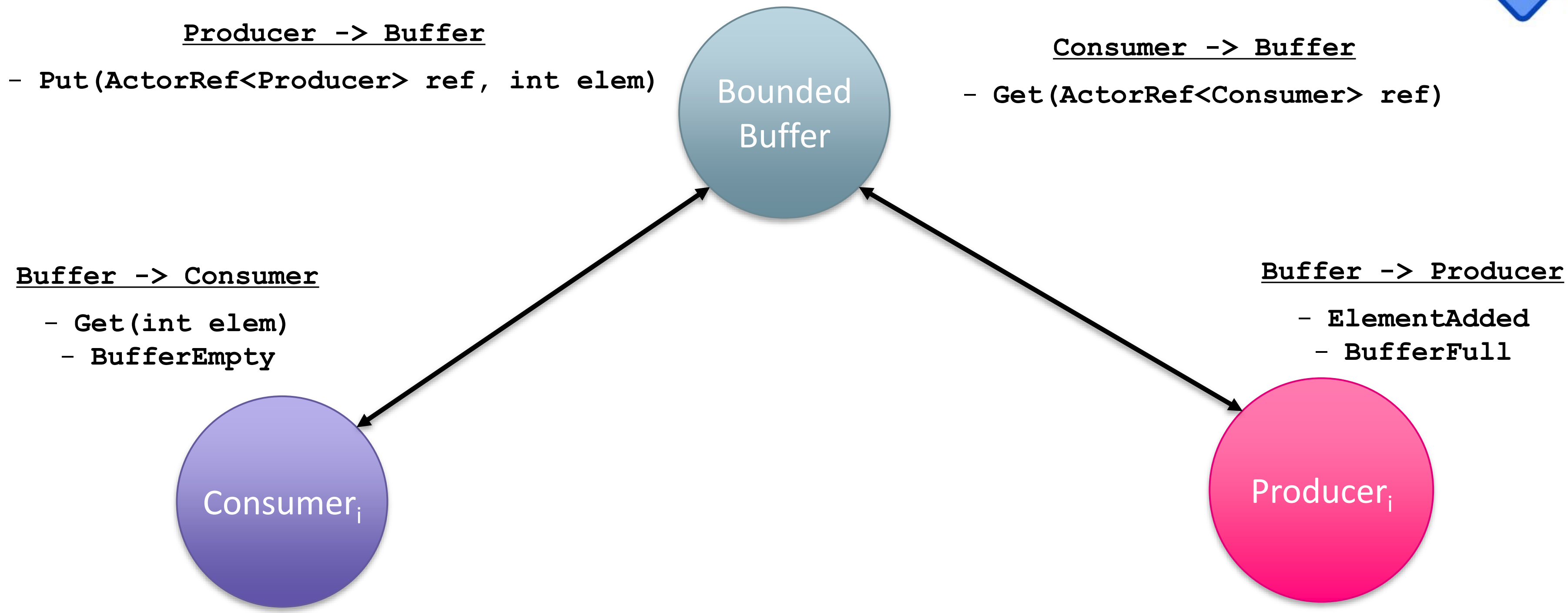
Producer-consumer problem | Intuition

Consumers

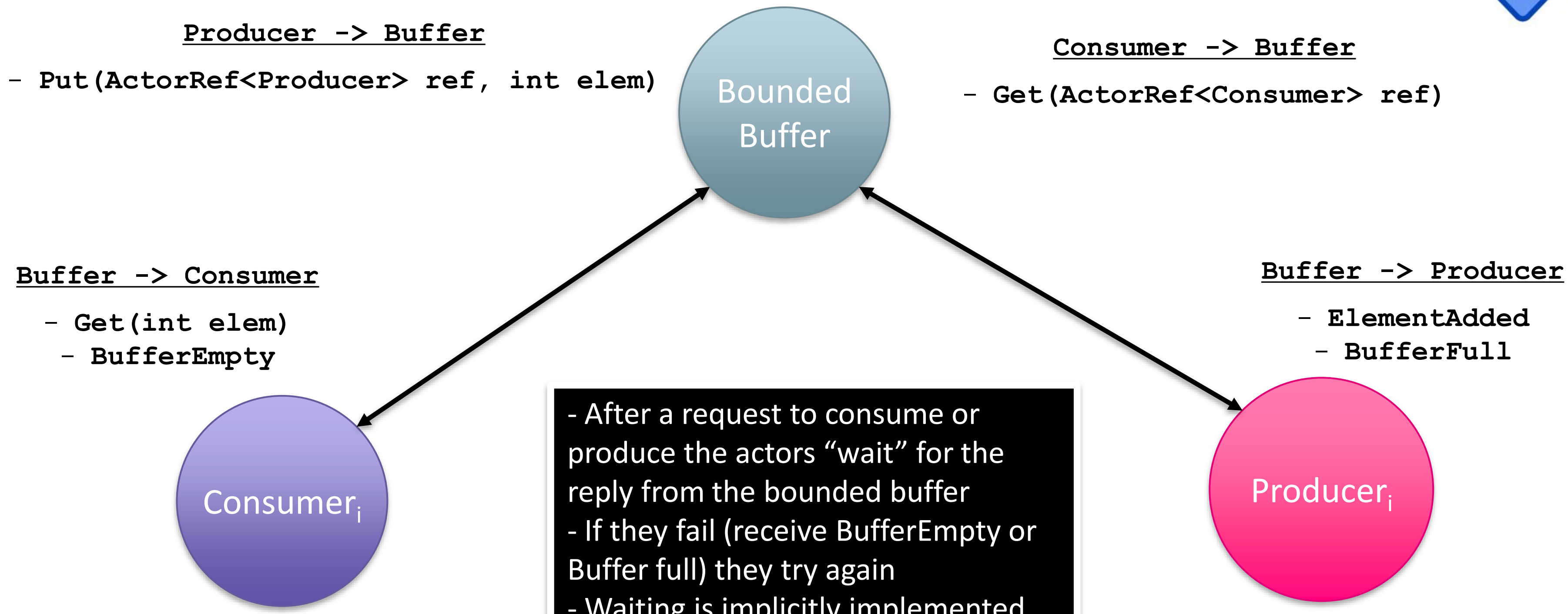
Producers

Shared data structure of fixed size

Bounded Buffer with Actors

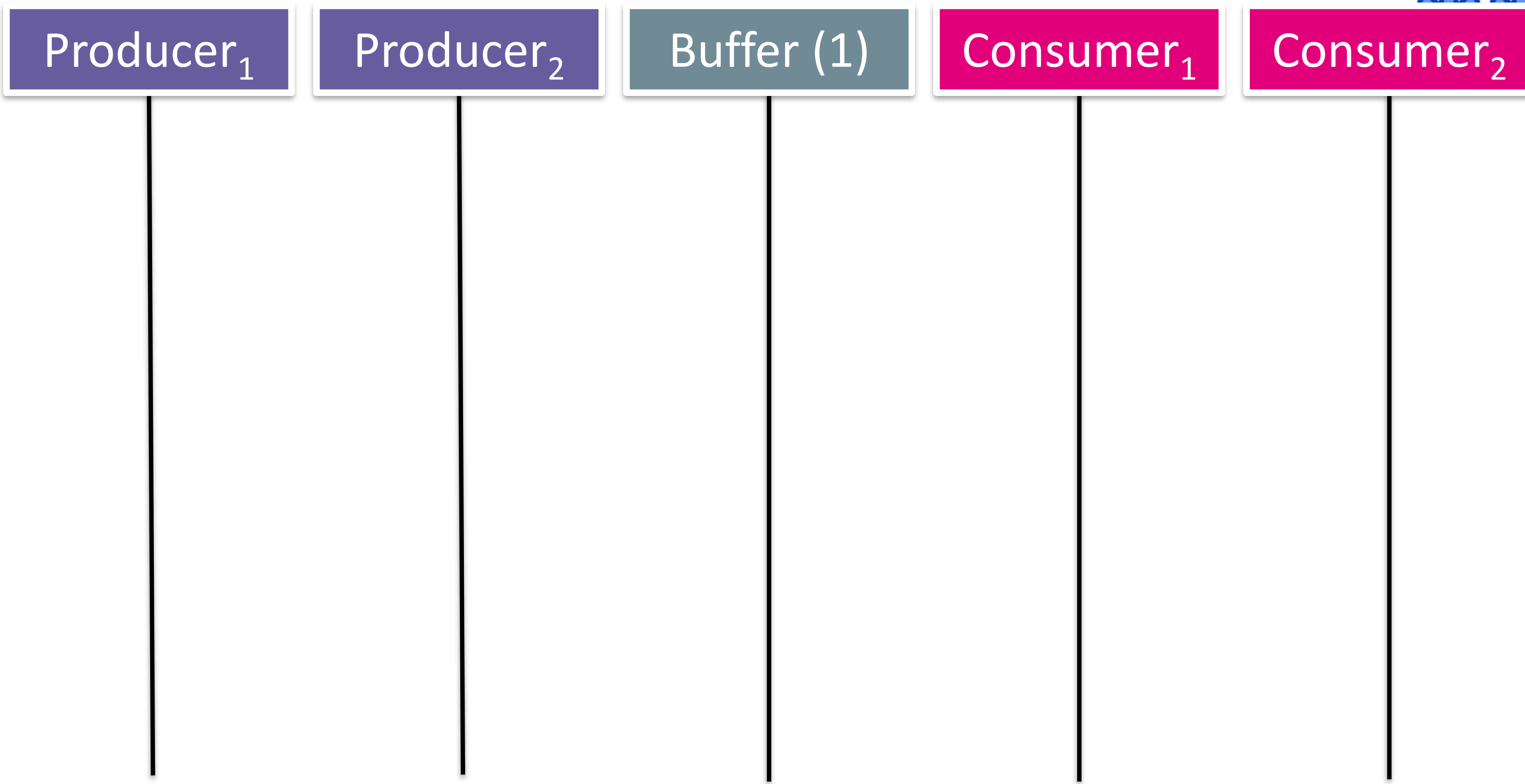


Bounded Buffer with Actors

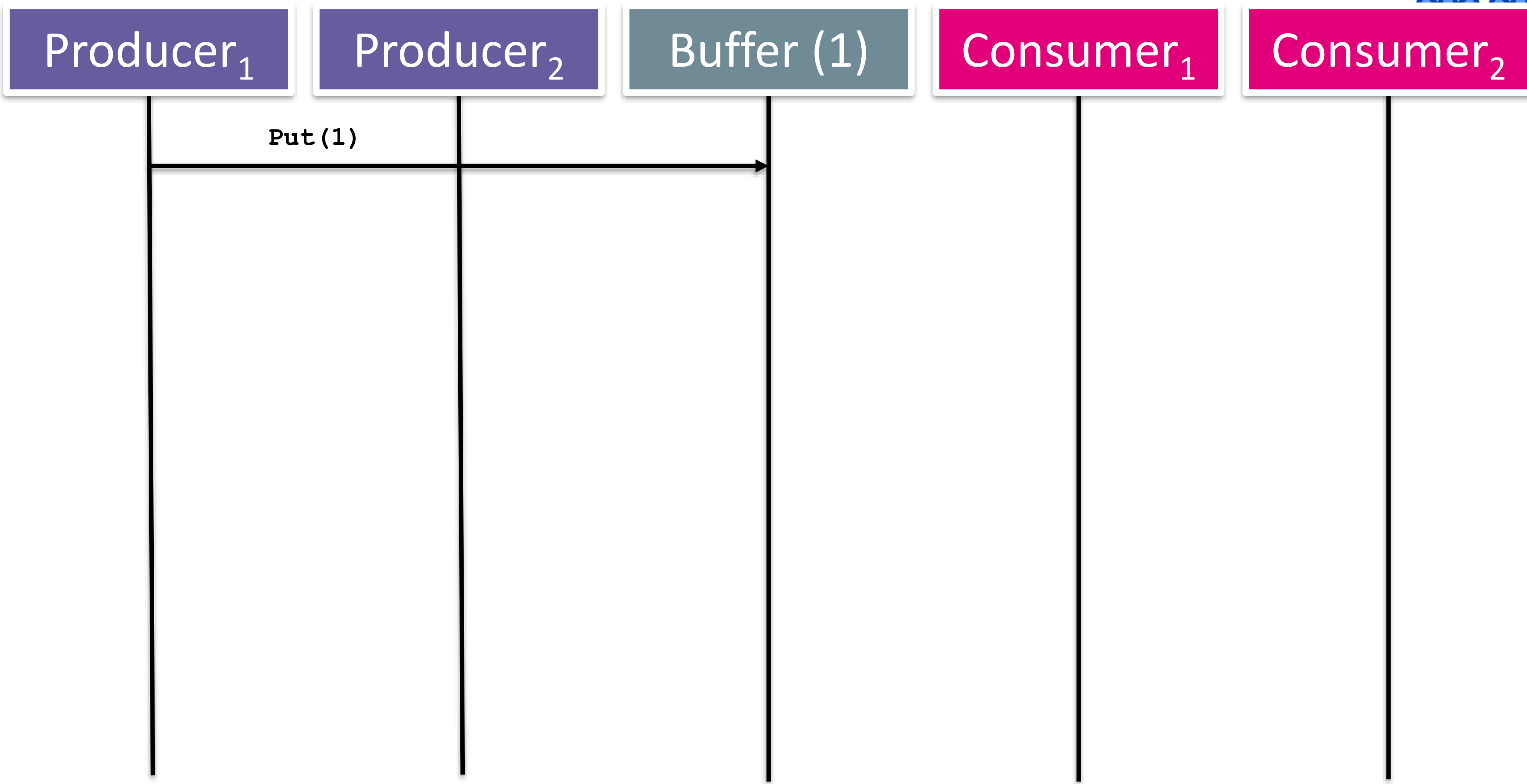


- After a request to consume or produce the actors “wait” for the reply from the bounded buffer
- If they fail (receive BufferEmpty or Buffer full) they try again
- Waiting is implicitly implemented as producers and consumers wait for the answer of the bounded buffer, and the buffer replies with error only if the actors cannot make progress

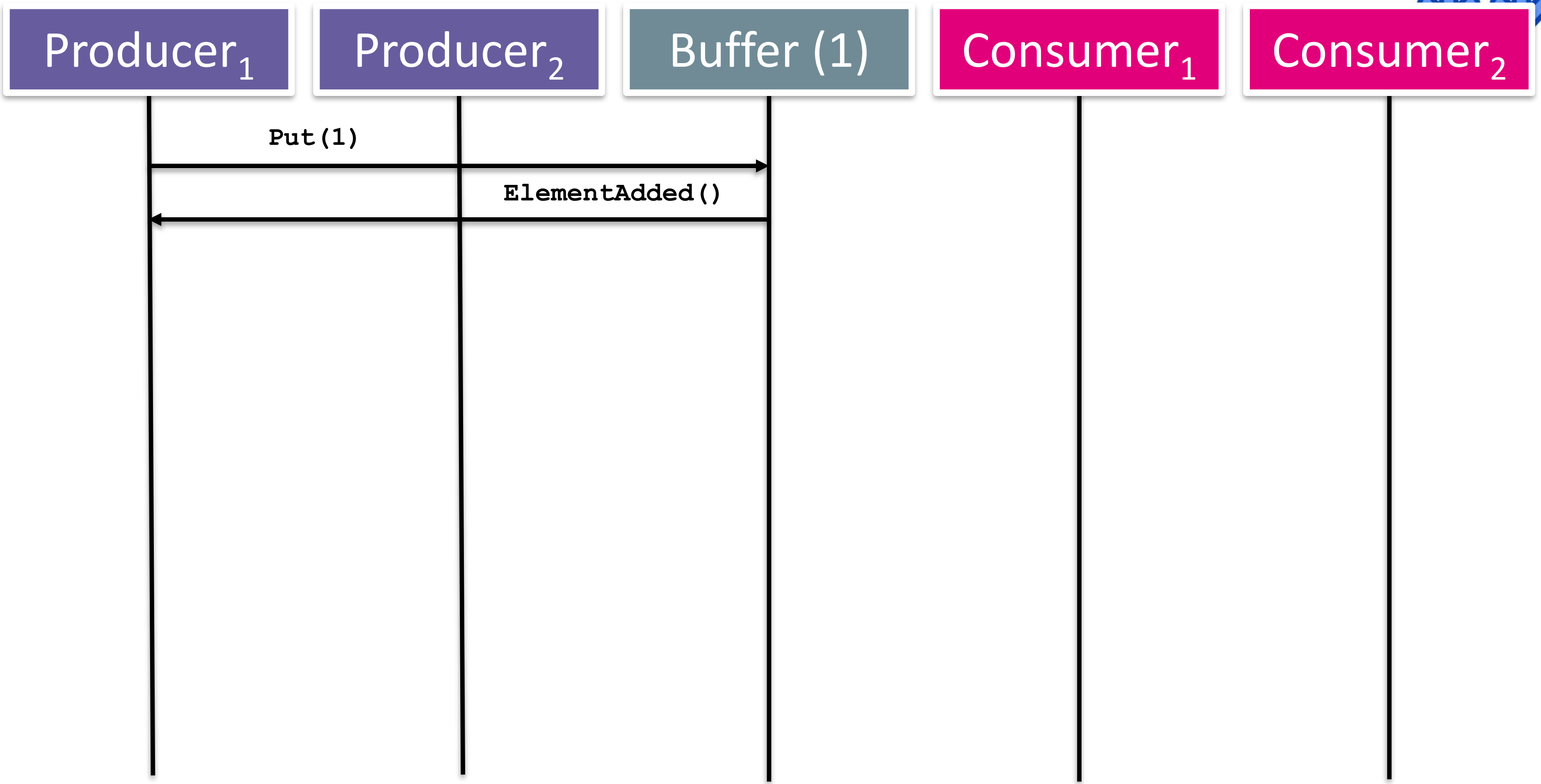
Bounded Buffer (size 1) – execution example



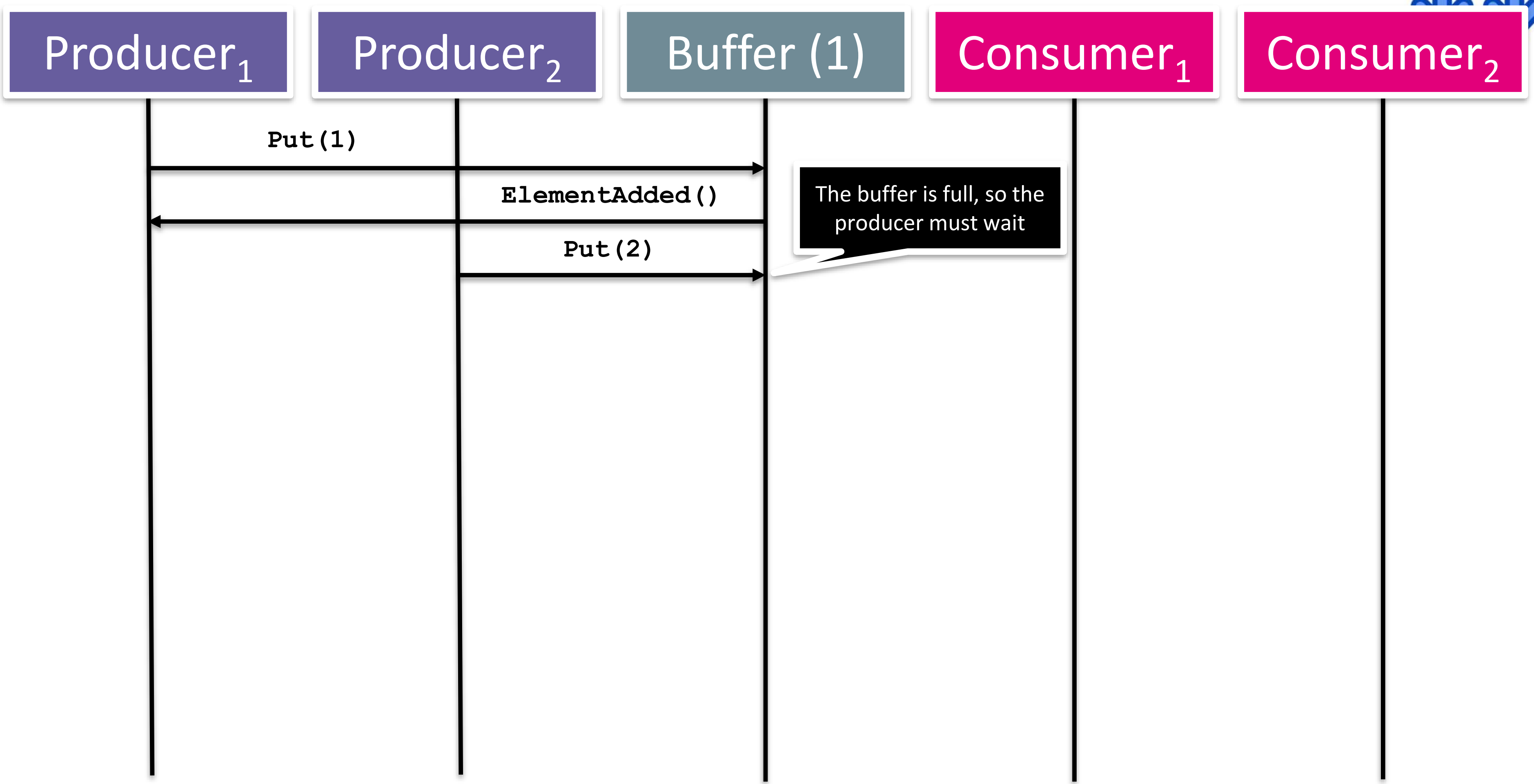
Bounded Buffer (size 1) – execution example



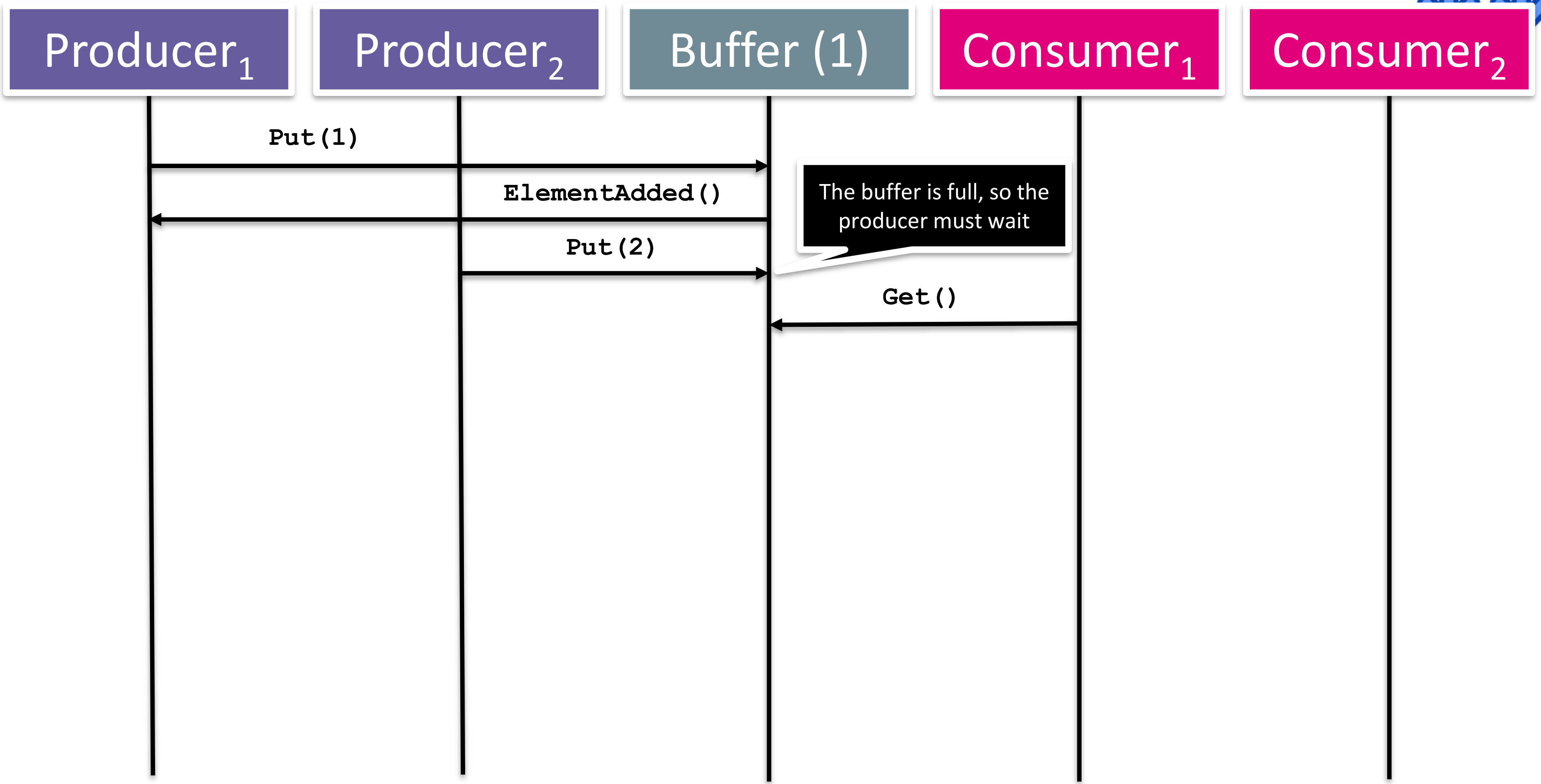
Bounded Buffer (size 1) – execution example



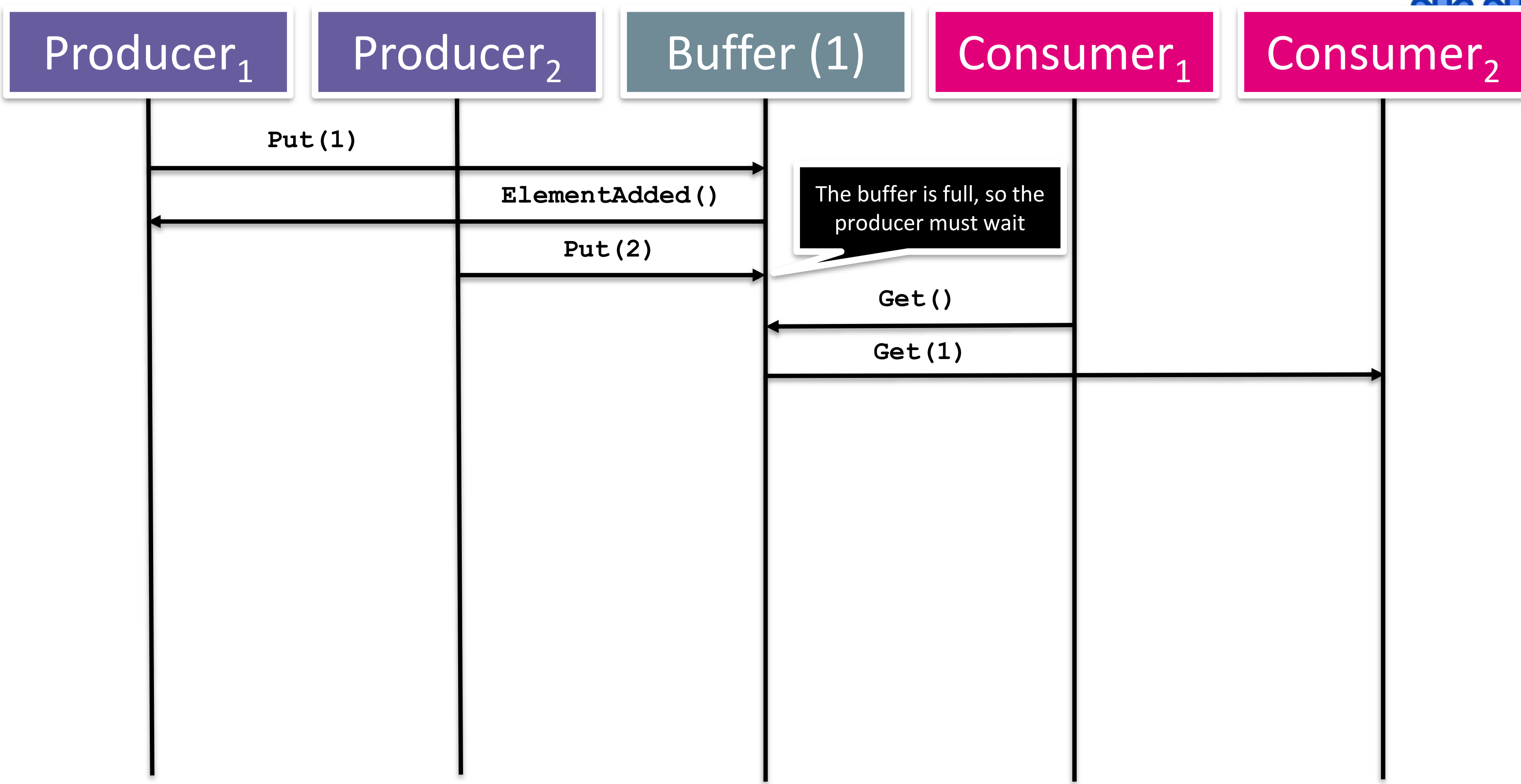
Bounded Buffer (size 1) – execution example



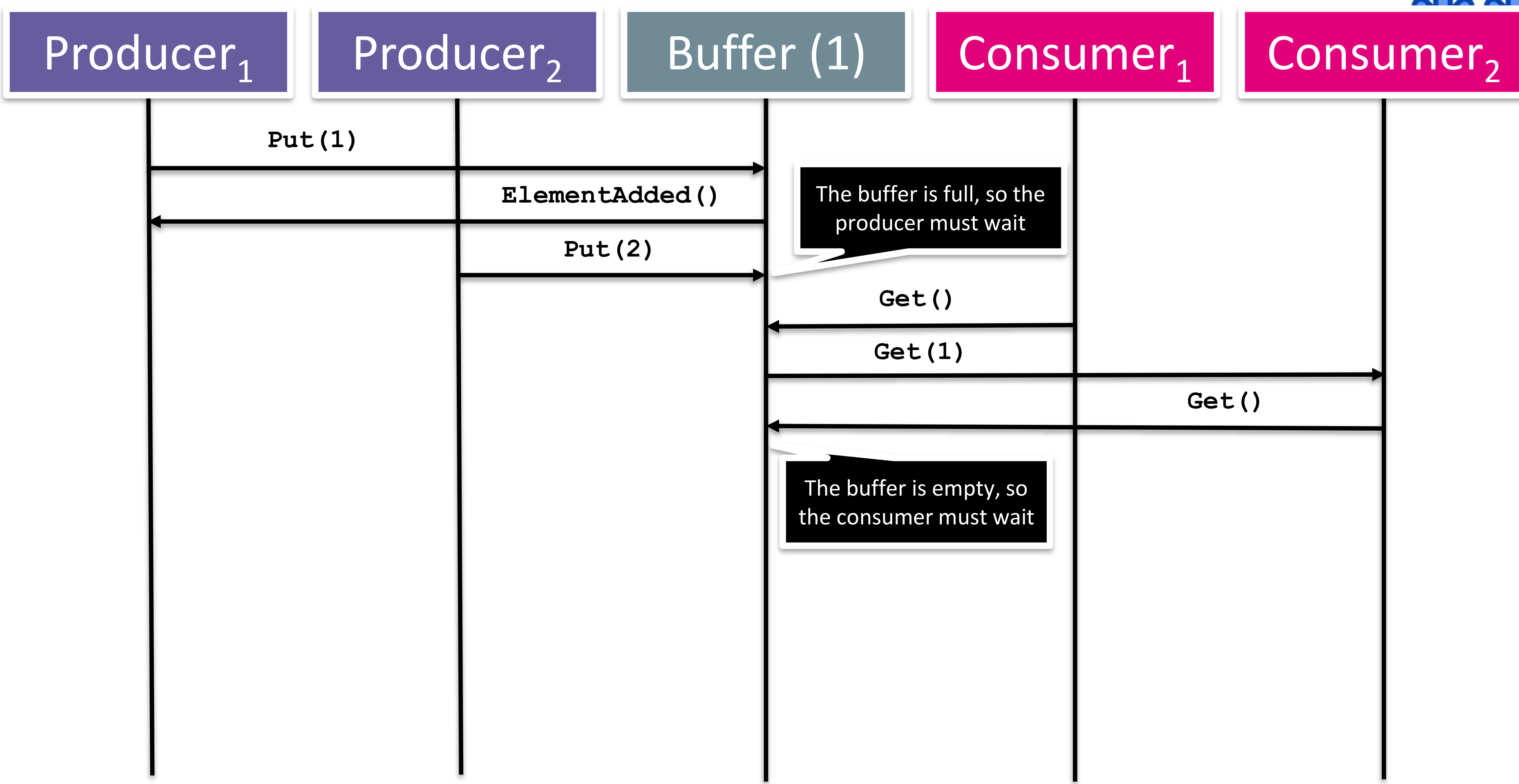
Bounded Buffer (size 1) – execution example



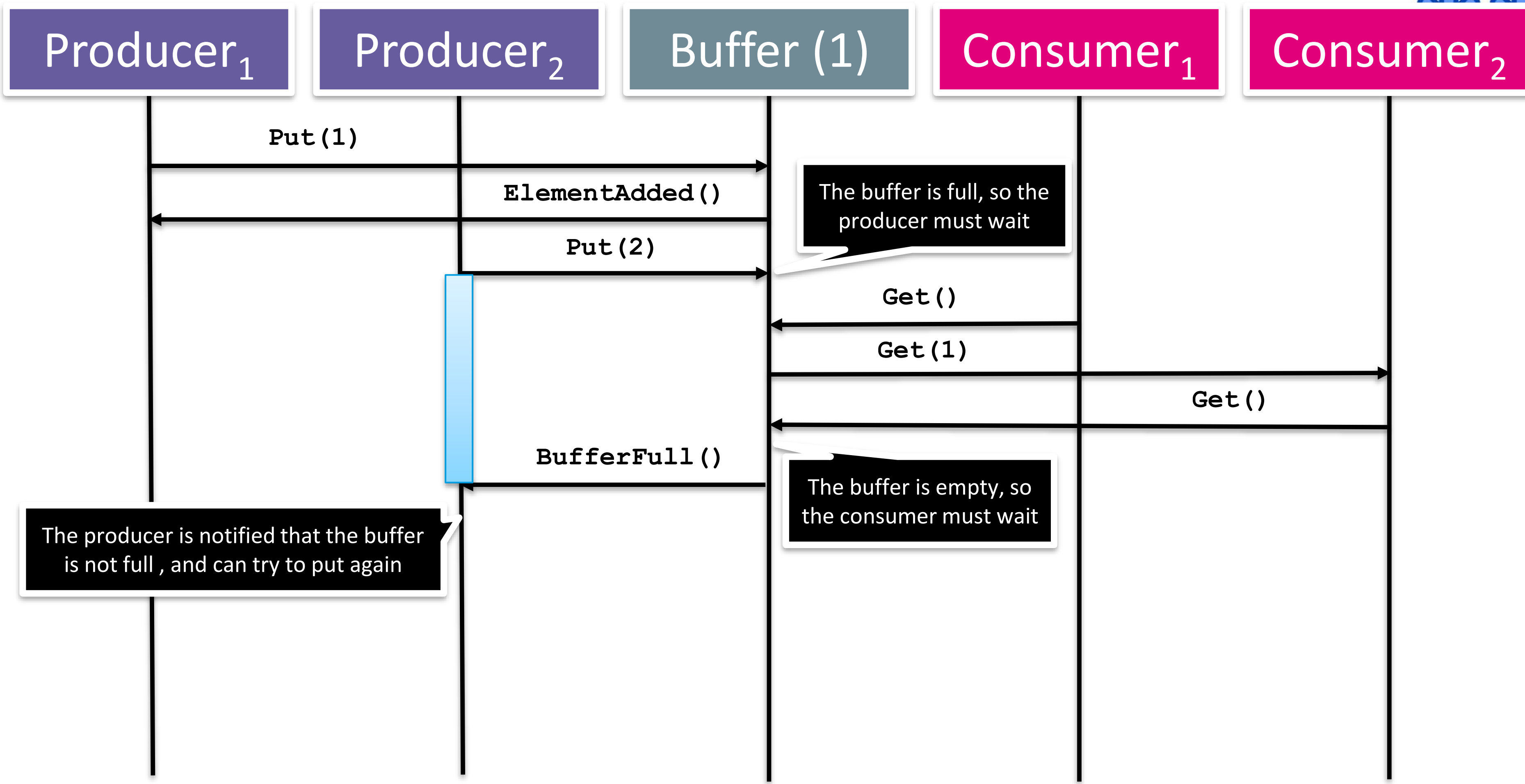
Bounded Buffer (size 1) – execution example



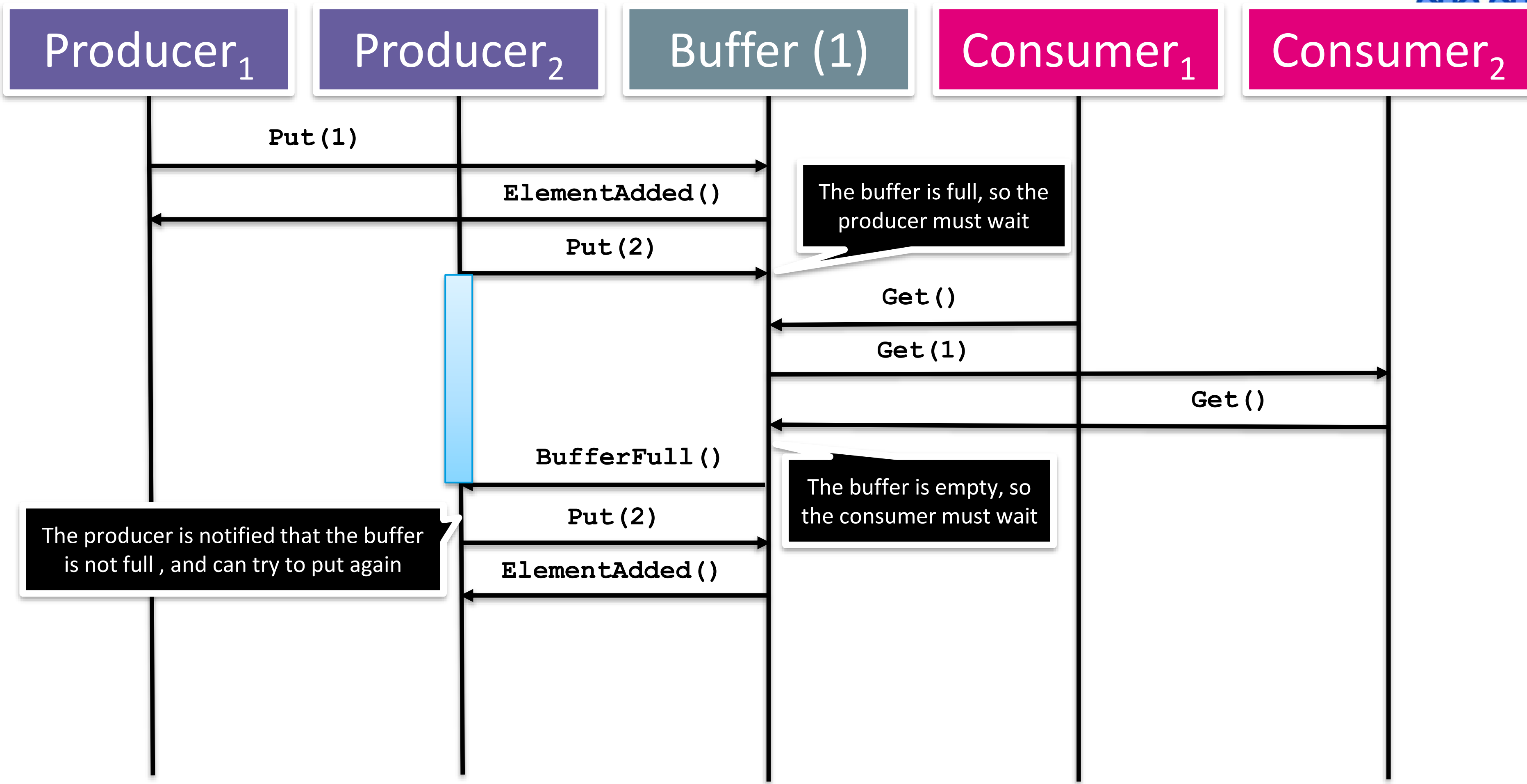
Bounded Buffer (size 1) – execution example



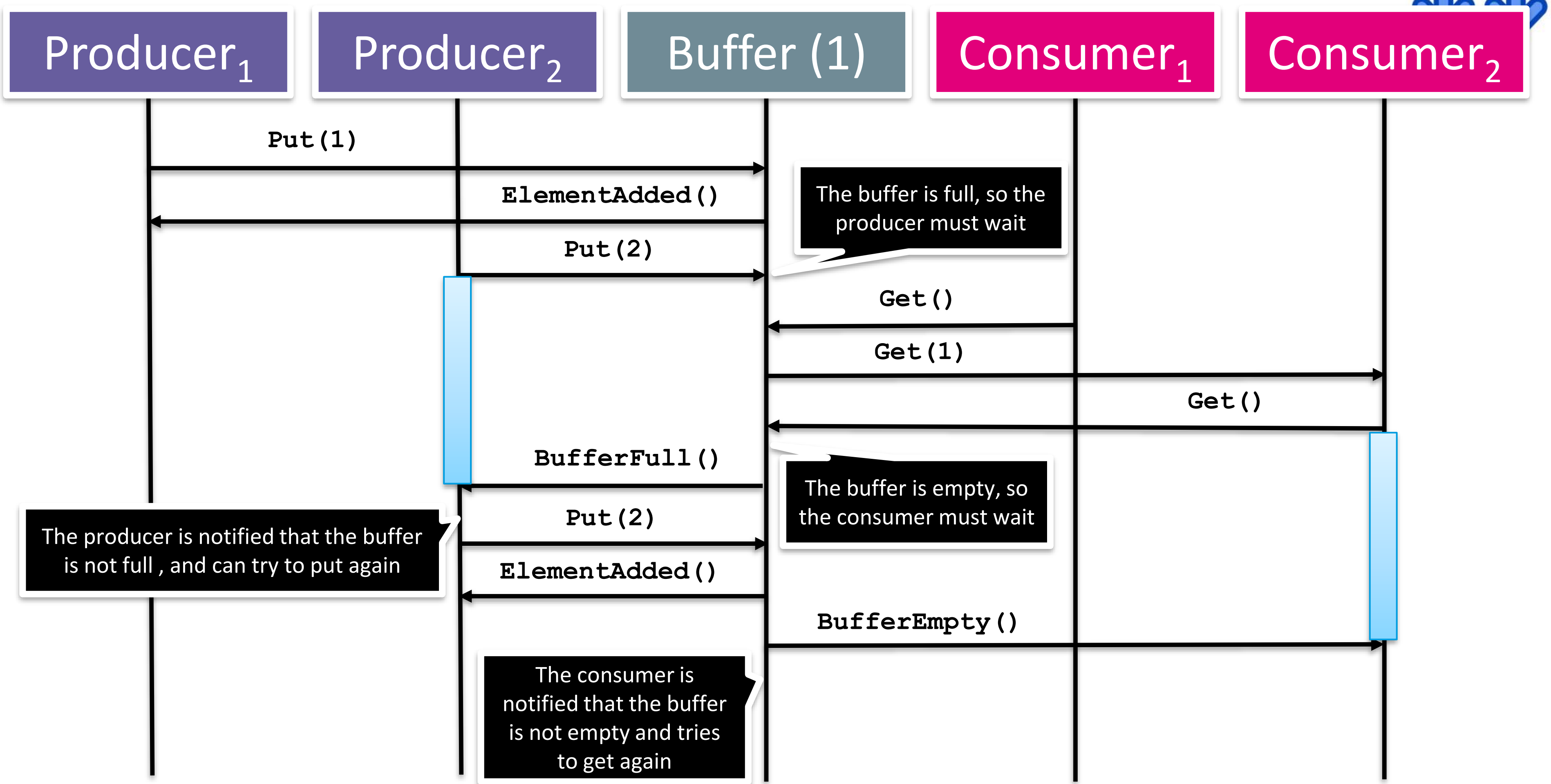
Bounded Buffer (size 1) – execution example



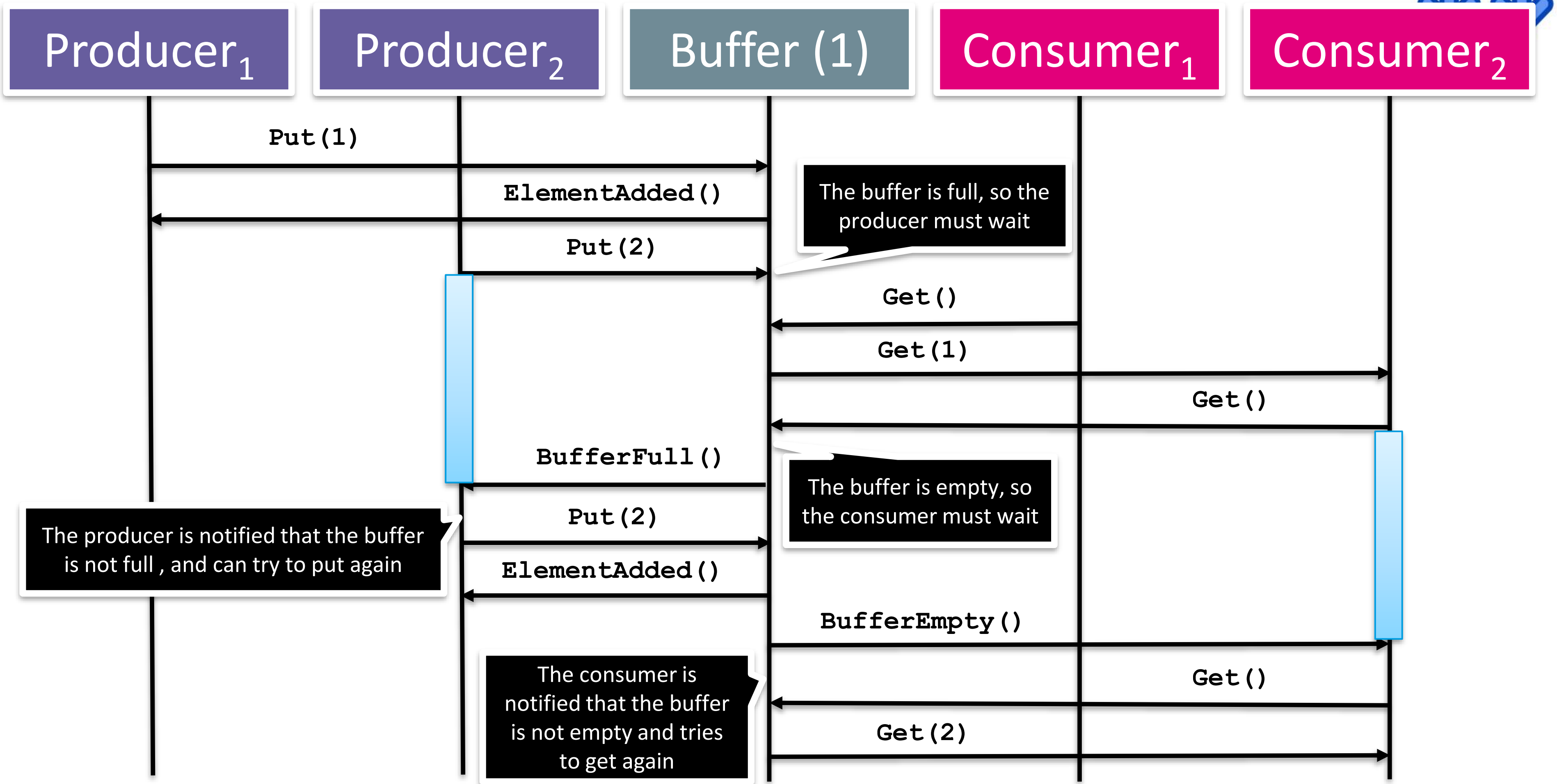
Bounded Buffer (size 1) – execution example



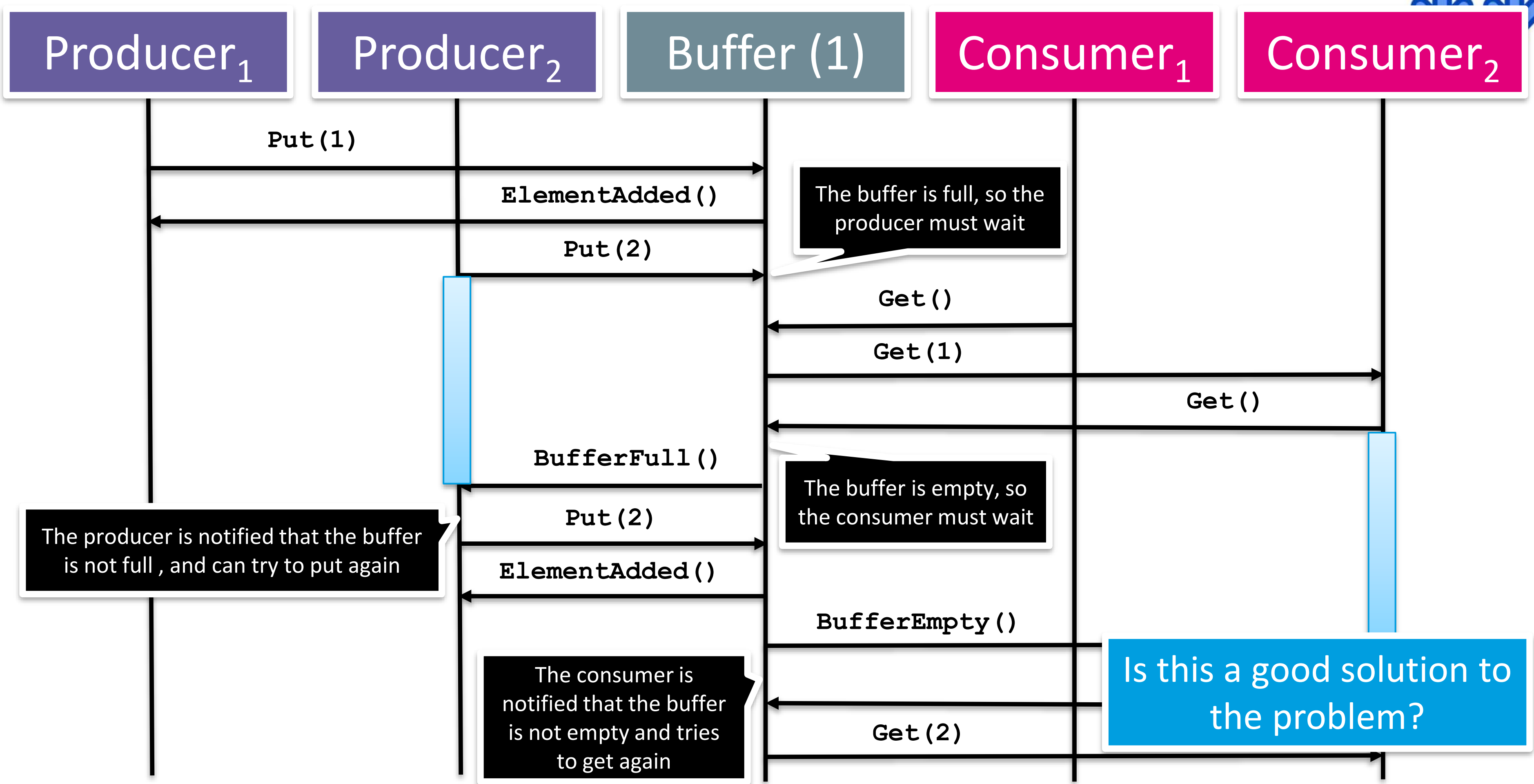
Bounded Buffer (size 1) – execution example



Bounded Buffer (size 1) – execution example

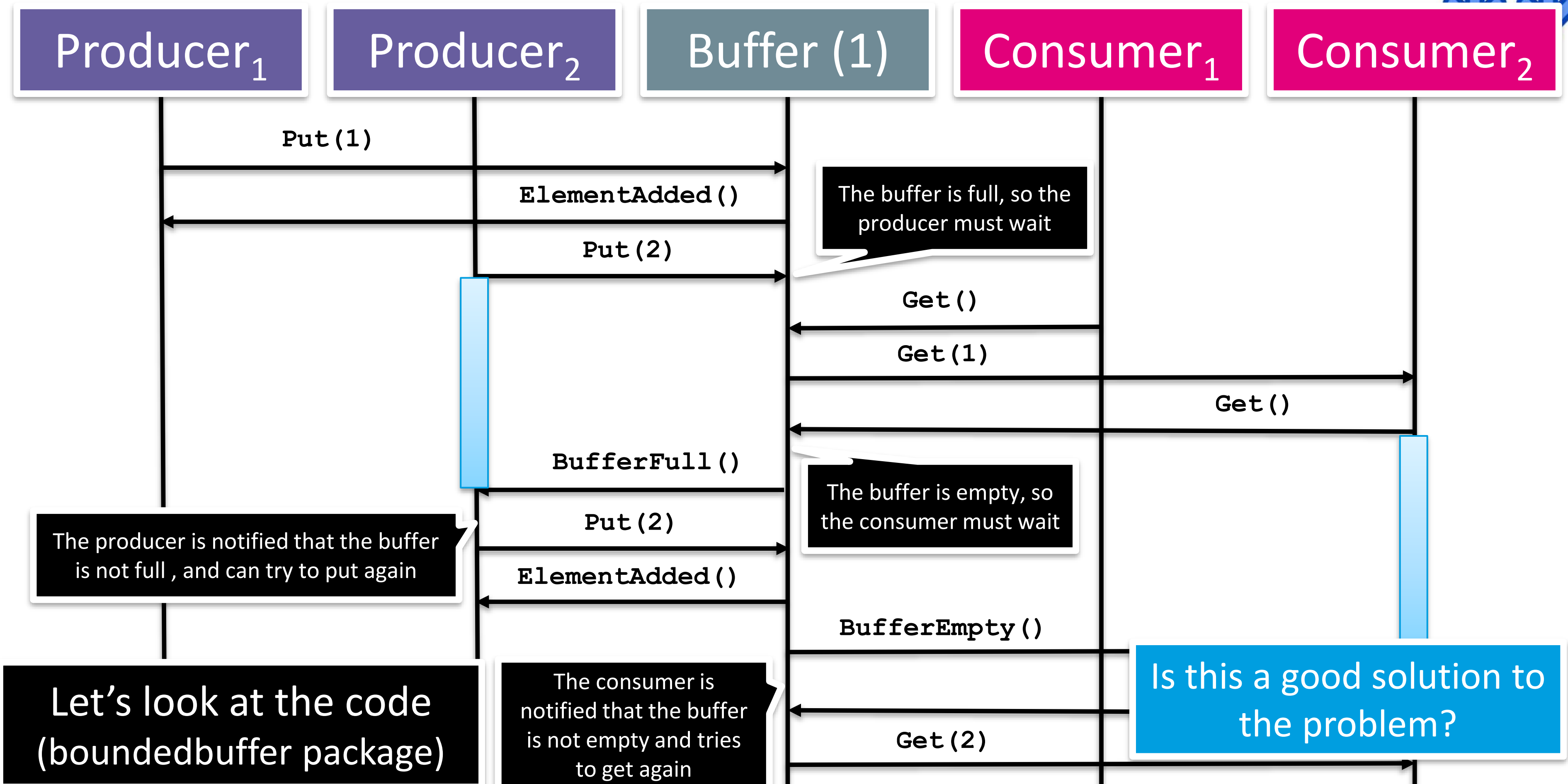


Bounded Buffer (size 1) – execution example



Is this a good solution to the problem?

Bounded Buffer (size 1) – execution example



Let's look at the code (boundedbuffer package)

Is this a good solution to the problem?

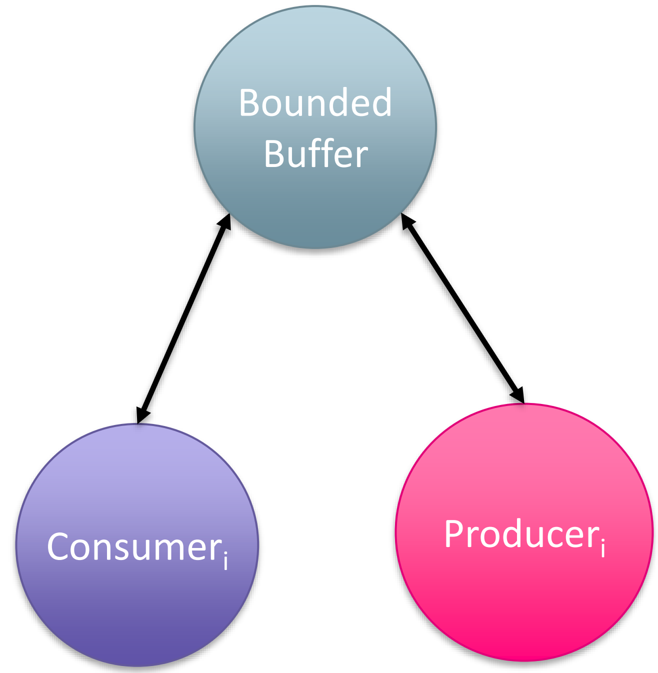
Bounded Buffer (interesting) executions

Assuming FIFO mailboxes
(Akka's default)

- Consider this execution

1. Producer1 sends put(1) to the buffer
2. Consumer1 sends get() to the buffer
3. ...

Is it guaranteed that Consumer 1 will get the produced element?

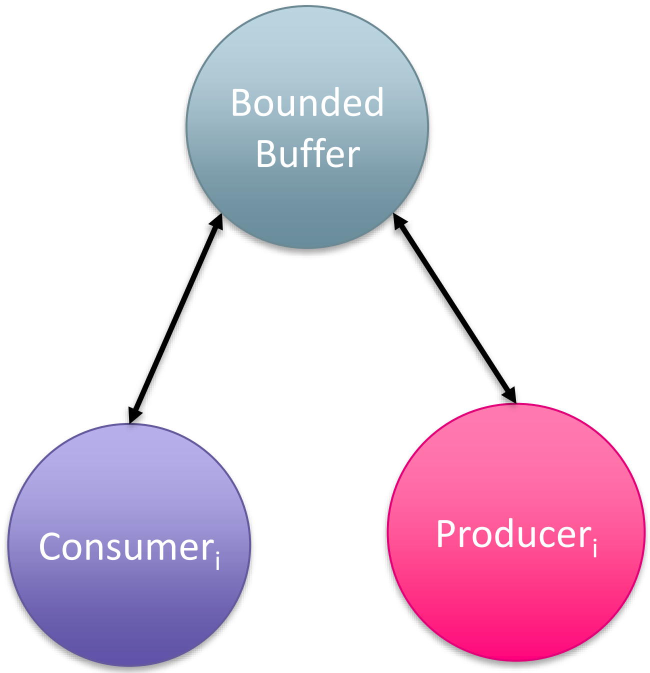


Bounded Buffer (interesting) executions

Assuming FIFO mailboxes
(Akka's default)

- Consider this execution

1. Producer1 sends put(1) to the buffer
2. Producer2 sends put(2) to the buffer
3. Consumer1 sends get() to the buffer
4. …



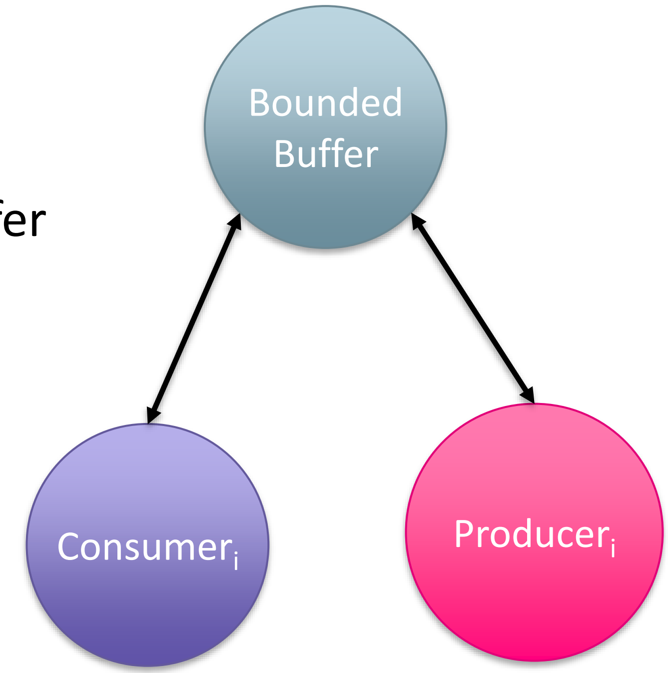
Is it guaranteed that Consumer1 will get either 1 or 2?

Bounded Buffer (interesting) executions

Assuming FIFO mailboxes
(Akka's default)

- Consider this execution

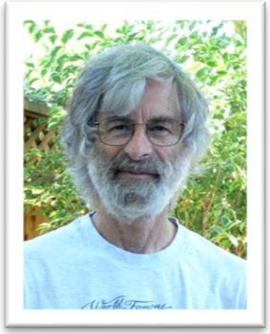
1. Producer1 sends put(1) to the buffer
2. Producer2 sends put(2) to the buffer
3. Producer1 receives ElementAdded() from the buffer
4. Consumer1 sends get() to the buffer
5. …



Is it guaranteed that Consumer 1 will get the produced element?

Happened-before in distributed systems

- Note that in the previous questions the behaviour of the systems depends on the reception of messages



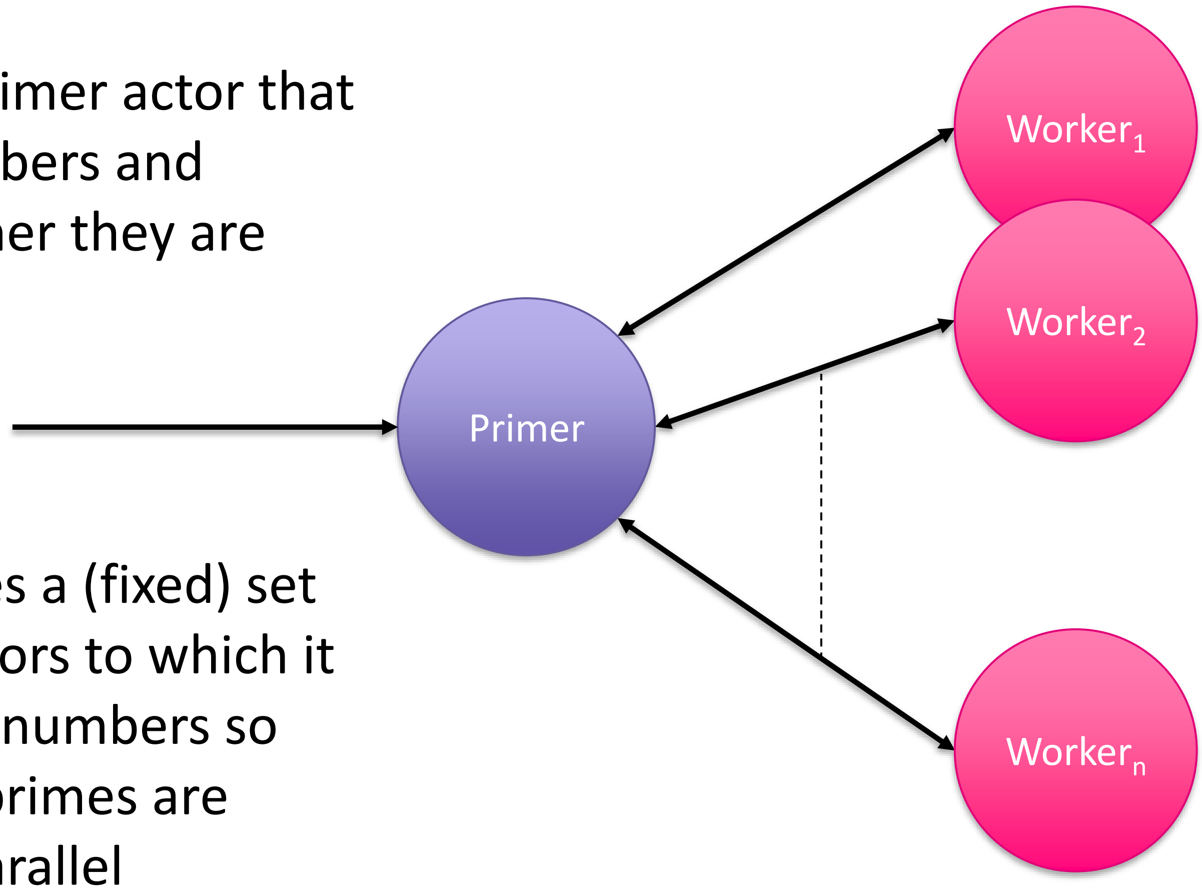
- Thus, the happened-before relation defined by Lamport is useful in reasoning about actor systems
  - An action *a* happens-before an action *b* if they belong to the same actor and *a* was executed before *b*
  - A send(m) action happens-before its corresponding receive(m)

- Note the similarity with the happens-before relation of the Java memory model
  - We reason about message exchange instead of locking (inherent coordination problems remain, i.e., "semantic" deadlock & starvation)
  - Visibility issues disappear as actors only access local memory

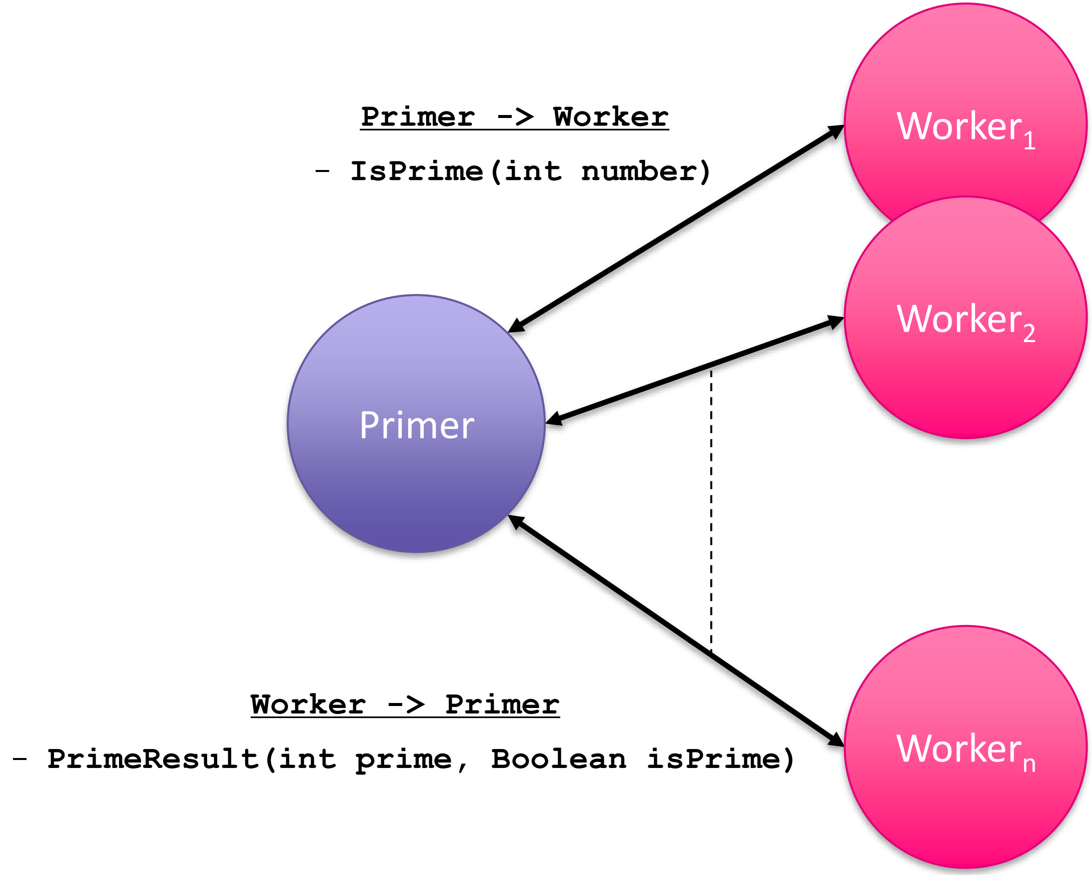
Primer

- Consider a Primer actor that receives numbers and checks whether they are prime

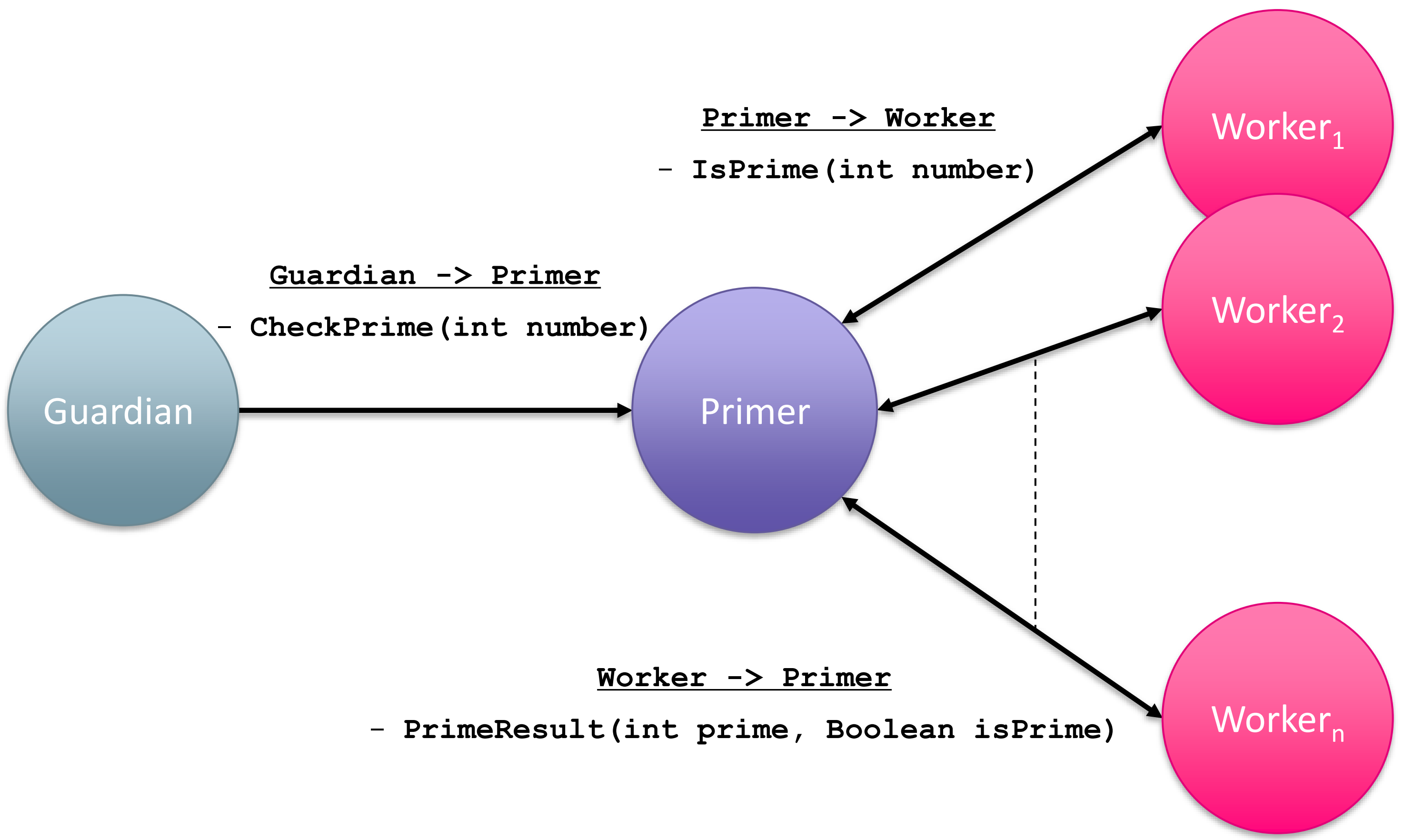
- The actor uses a (fixed) set of worker actors to which it forwards the numbers so that several primes are checked in parallel



Primer


Primer -> Worker
- IsPrime(int number)
Worker₁
Worker₂
Primer
Workerₙ
Worker -> Primer
- PrimeResult(int prime, Boolean isPrime)

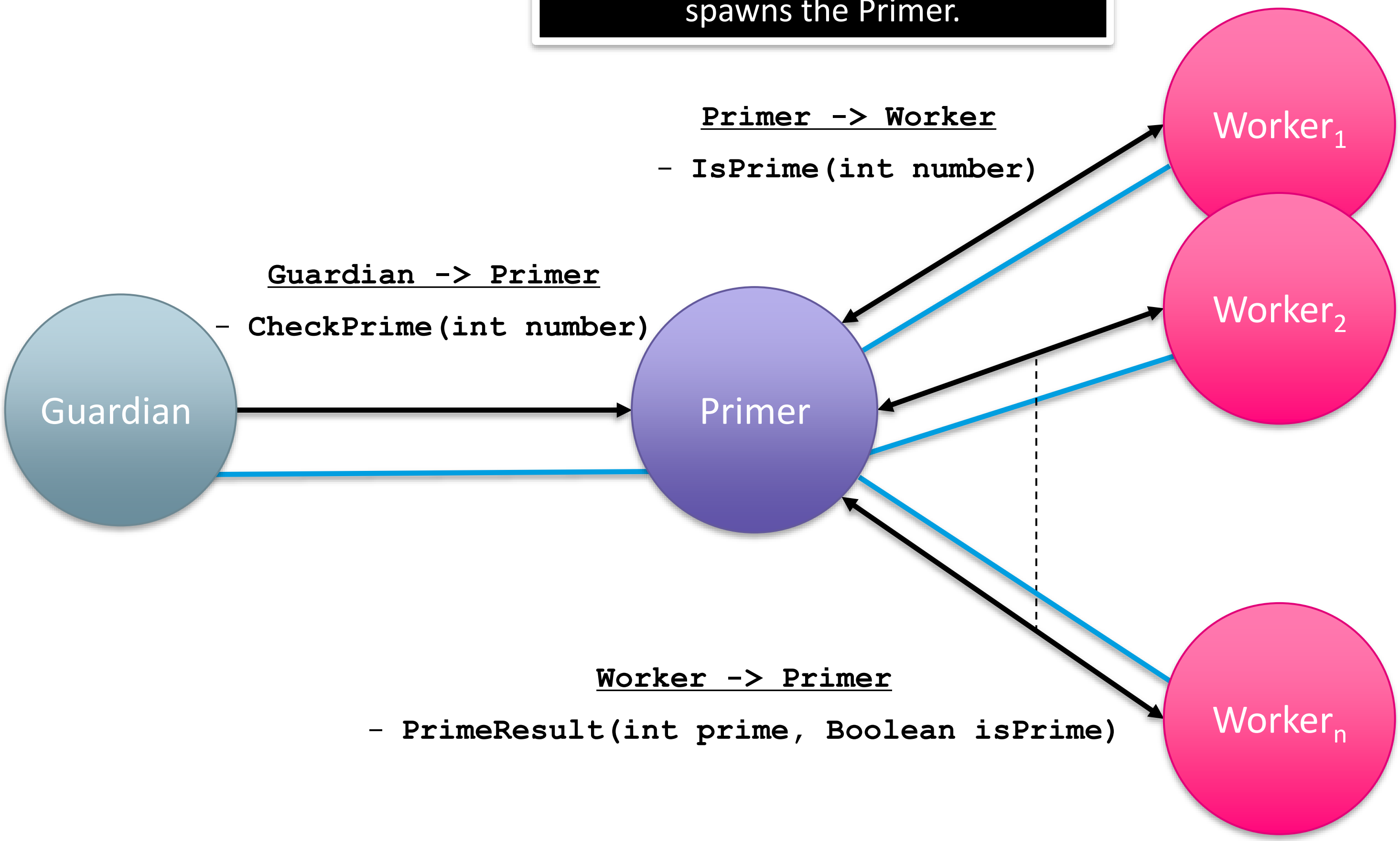
Primer



Primer -> Worker
- IsPrime(int number)
Guardian -> Primer
- CheckPrime(int number)
Worker -> Primer
- PrimeResult(int prime, Boolean isPrime)
Guardian
Primer
Worker$_1$
Worker$_2$
Worker$_n$

Primer

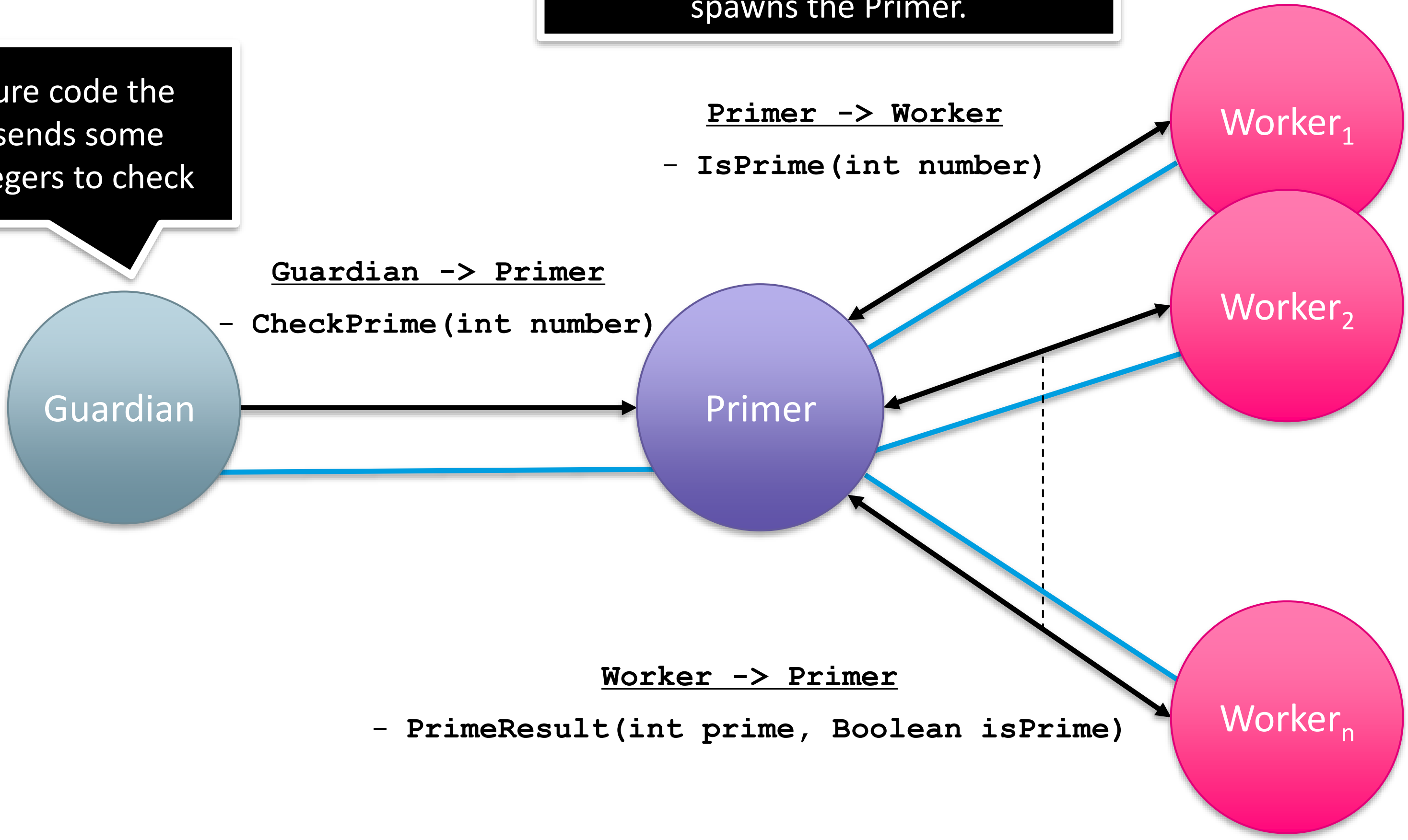
The Primer is in charge of spawning the workers. As usual, the guardian spawns the Primer.



Primer

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

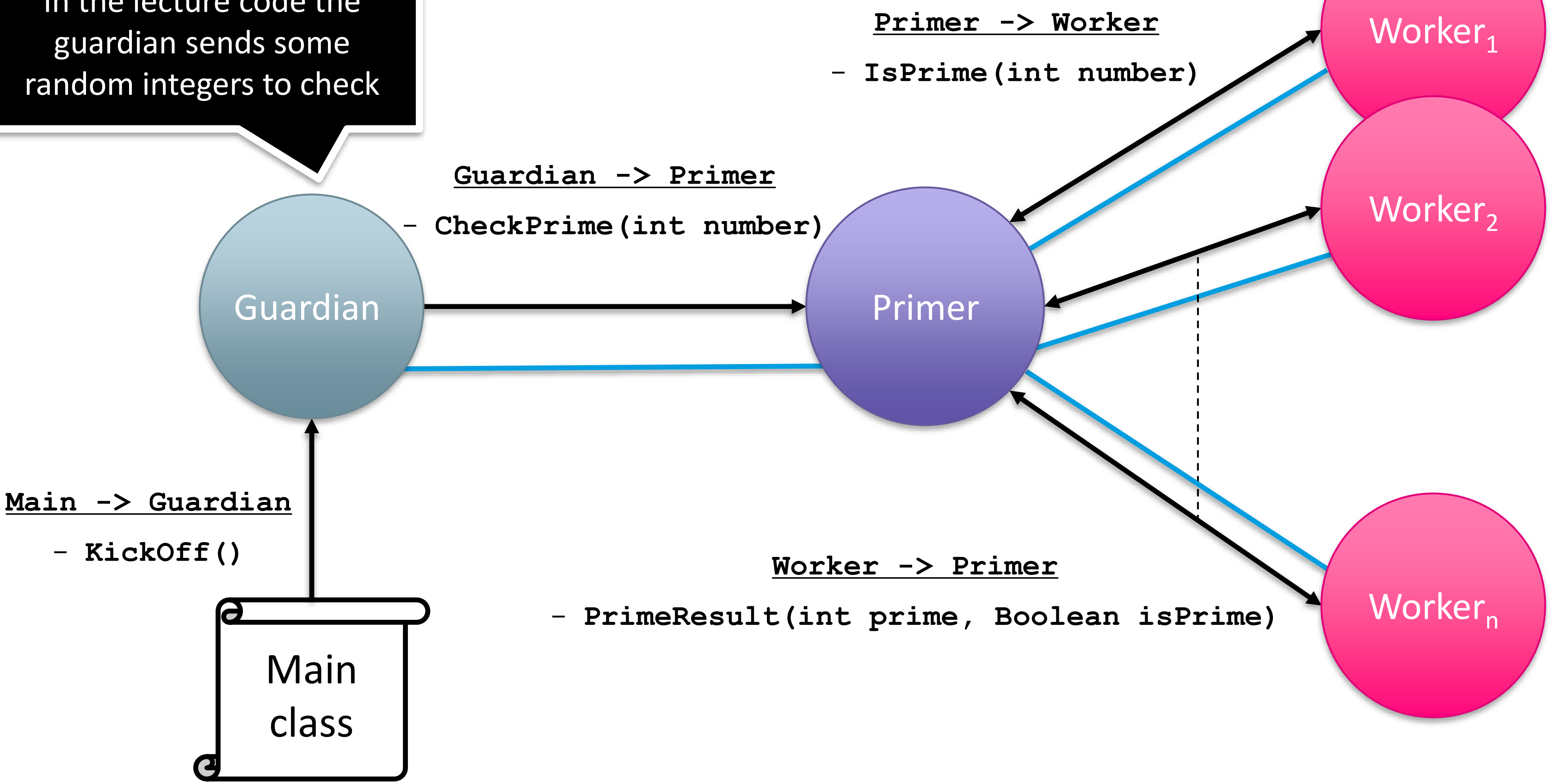
In the lecture code the guardian sends some random integers to check



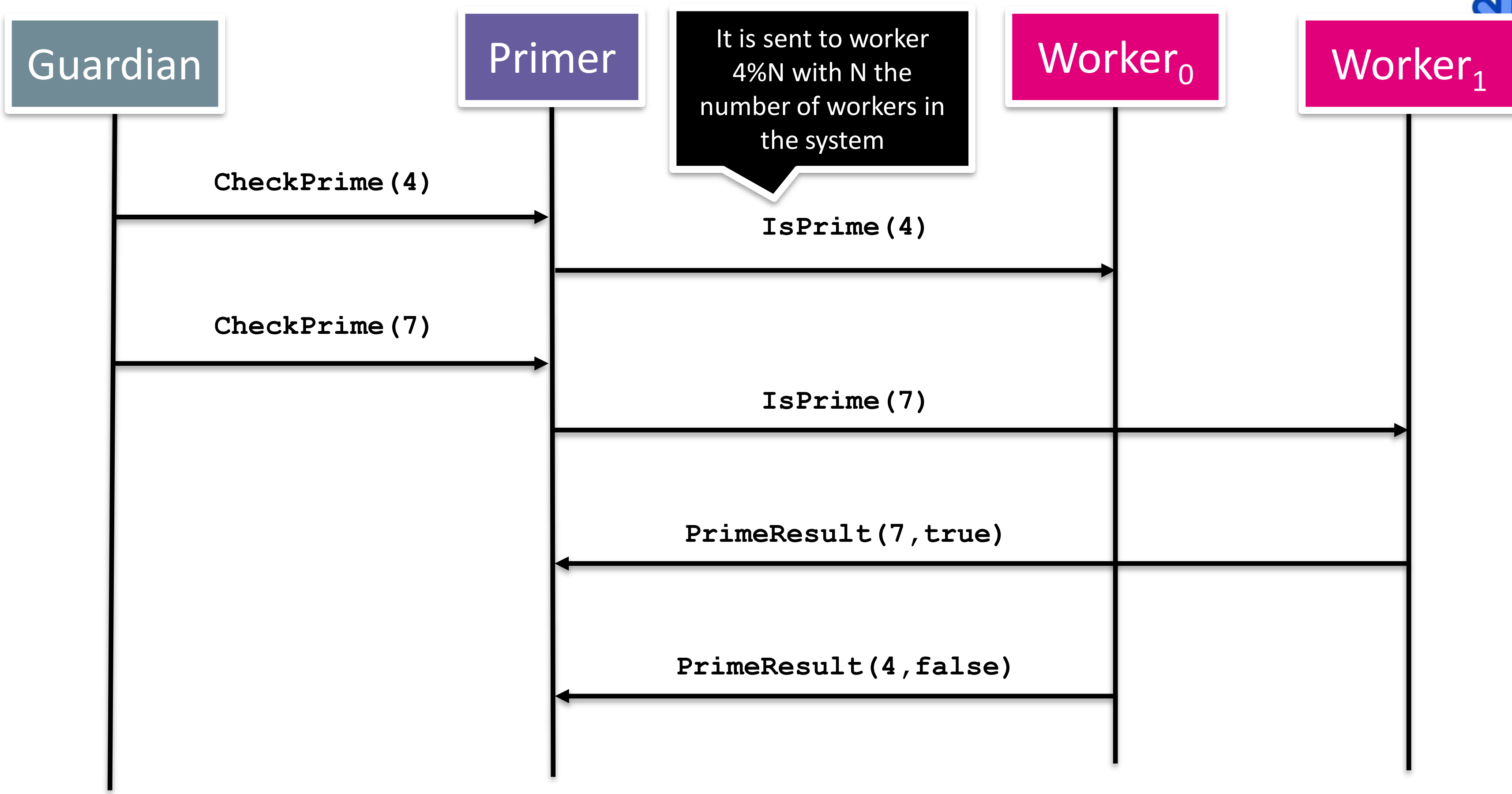
Primer

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

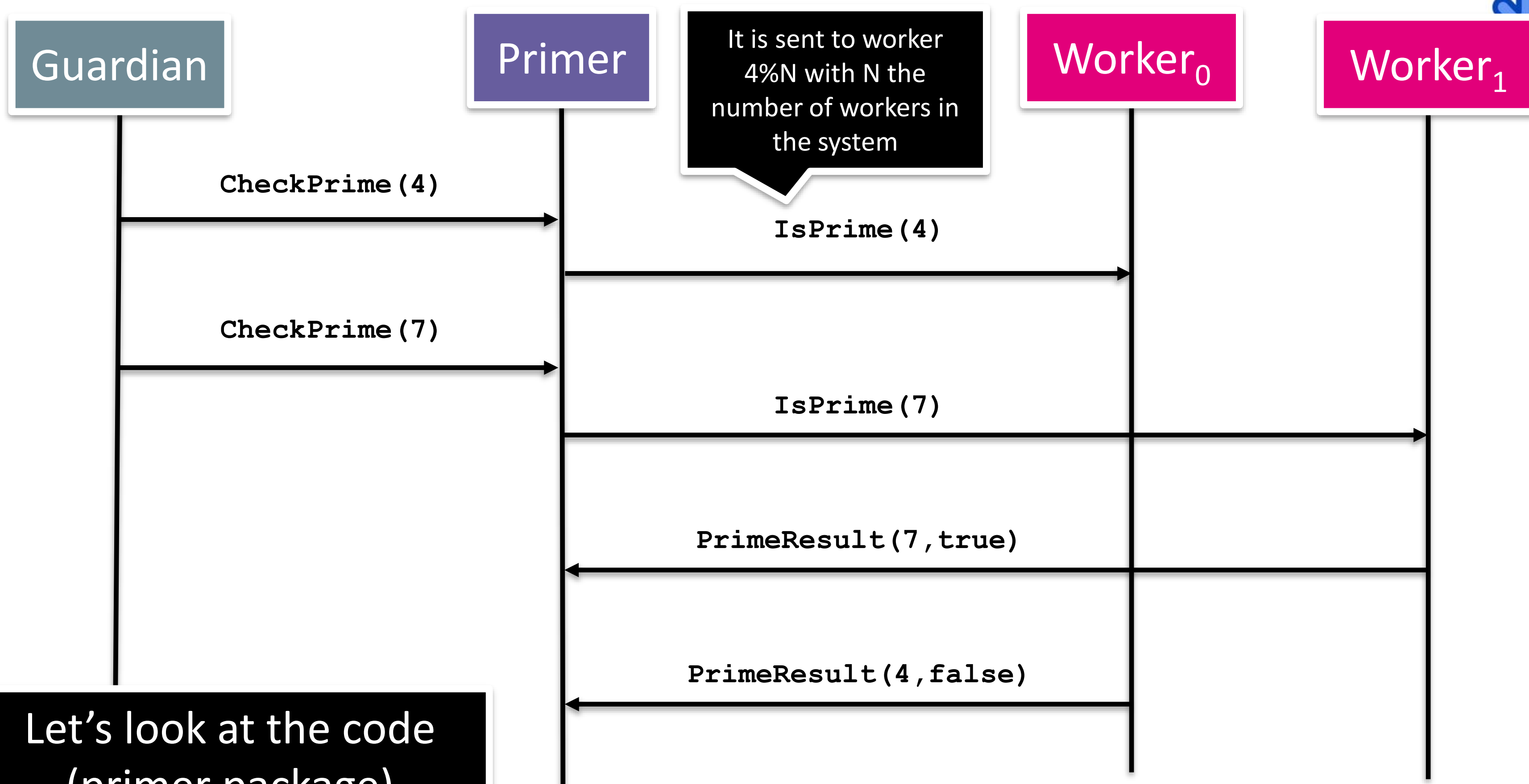
In the lecture code the guardian sends some random integers to check



Primer – execution example



Primer – execution example



Let's look at the code (primer package)

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?

Primer – Printing order

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

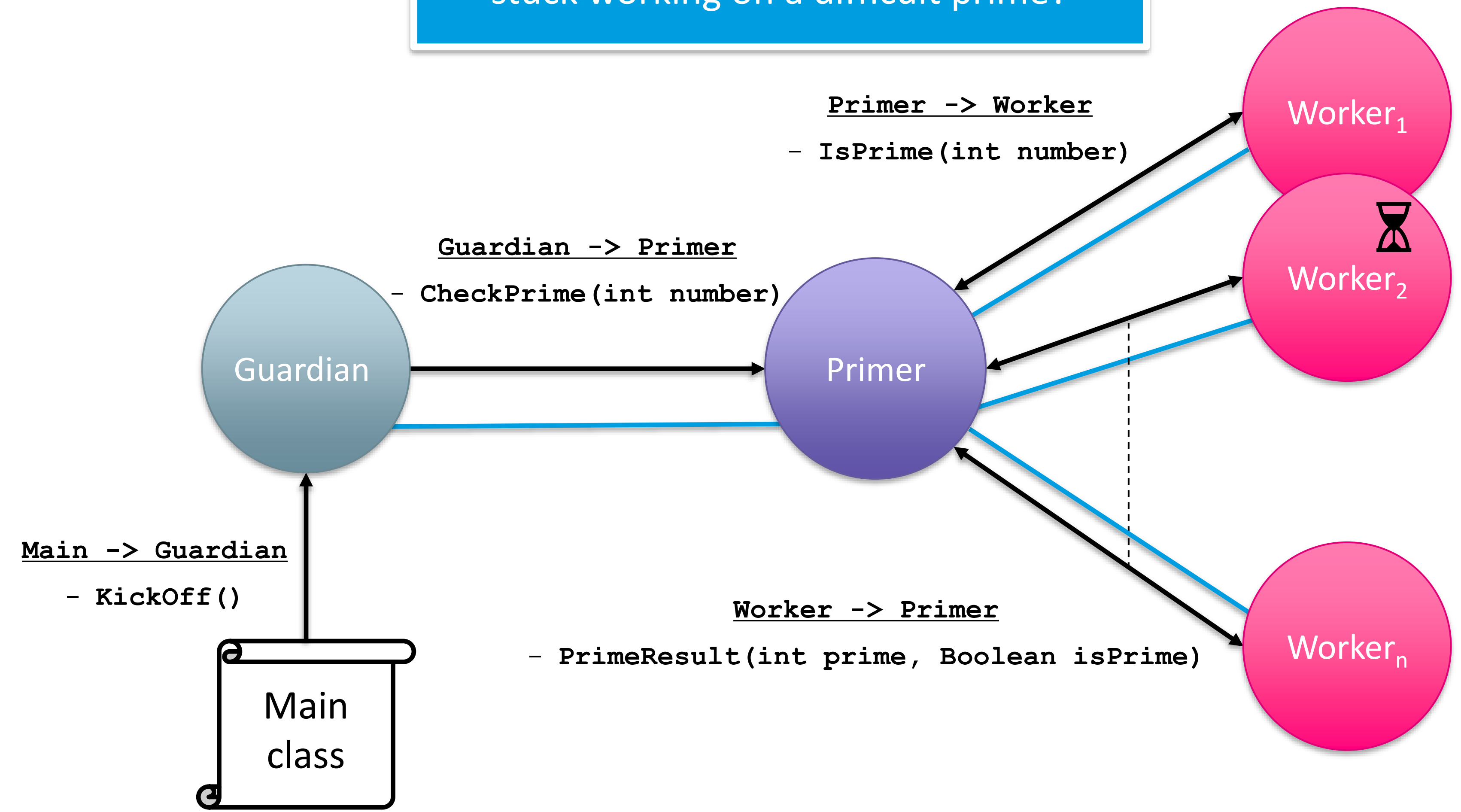
```
pardo@pardo-work:week10lecture$ gradle run -PmainClass=primer.Main

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

Primer

What happens if one of the workers gets stuck working on a difficult prime?



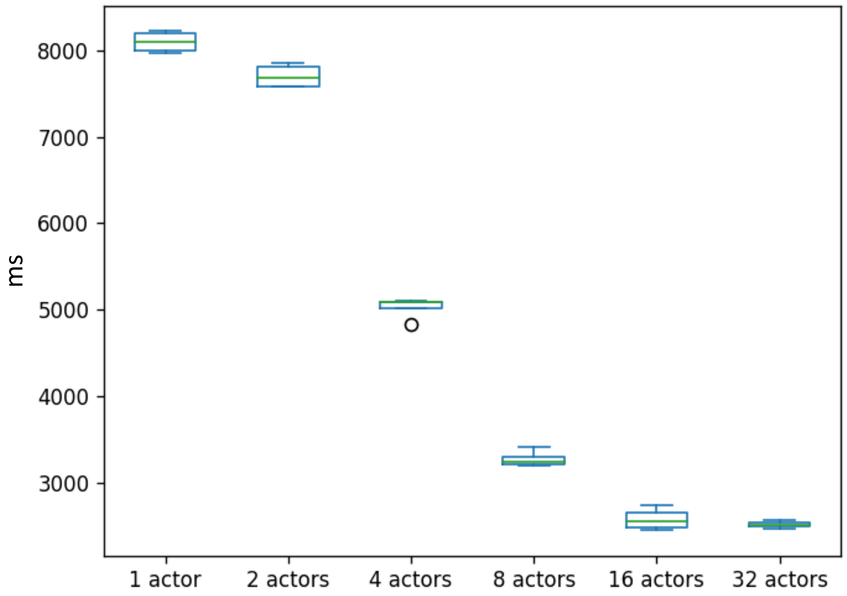
Actors systems with *dynamic* topology

Don’t be shy, spawn actors!

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

Do more actors improve performance?

- 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 a ForkJoinPool (a version of the ThreadPool, which is more efficient for tasks with low dependencies)
- However, actor systems can be distributed among many JVMs and computers
    - We are not limited to a single computer throughput
    - See Akka cluster

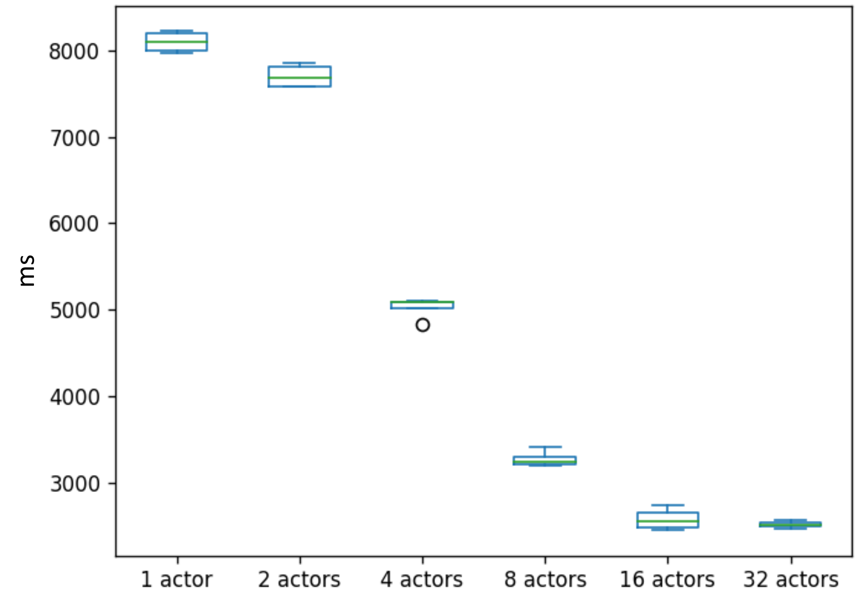


Do more actors improve performance?

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

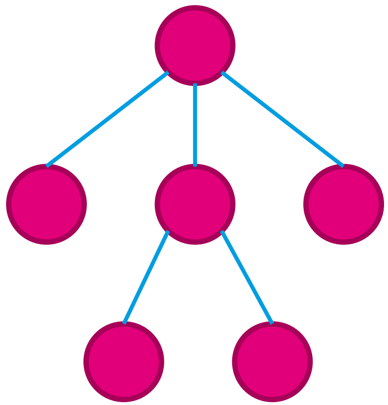
- Akka implements actors systems using a ForkJoinPool (a version of the ThreadPool, which is more efficient for tasks with low dependencies)

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



Topology (Actor hierarchy)

- 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

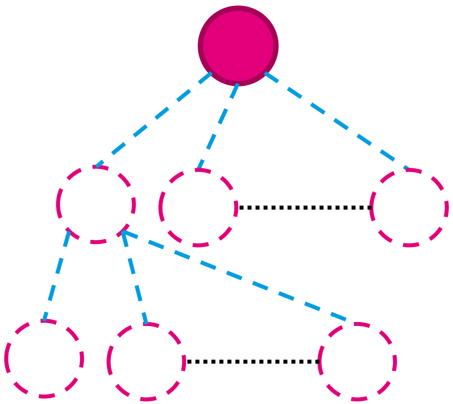
Solid lines and actors represent elements that are created during initialization and never change

Dynamic topology

- 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)

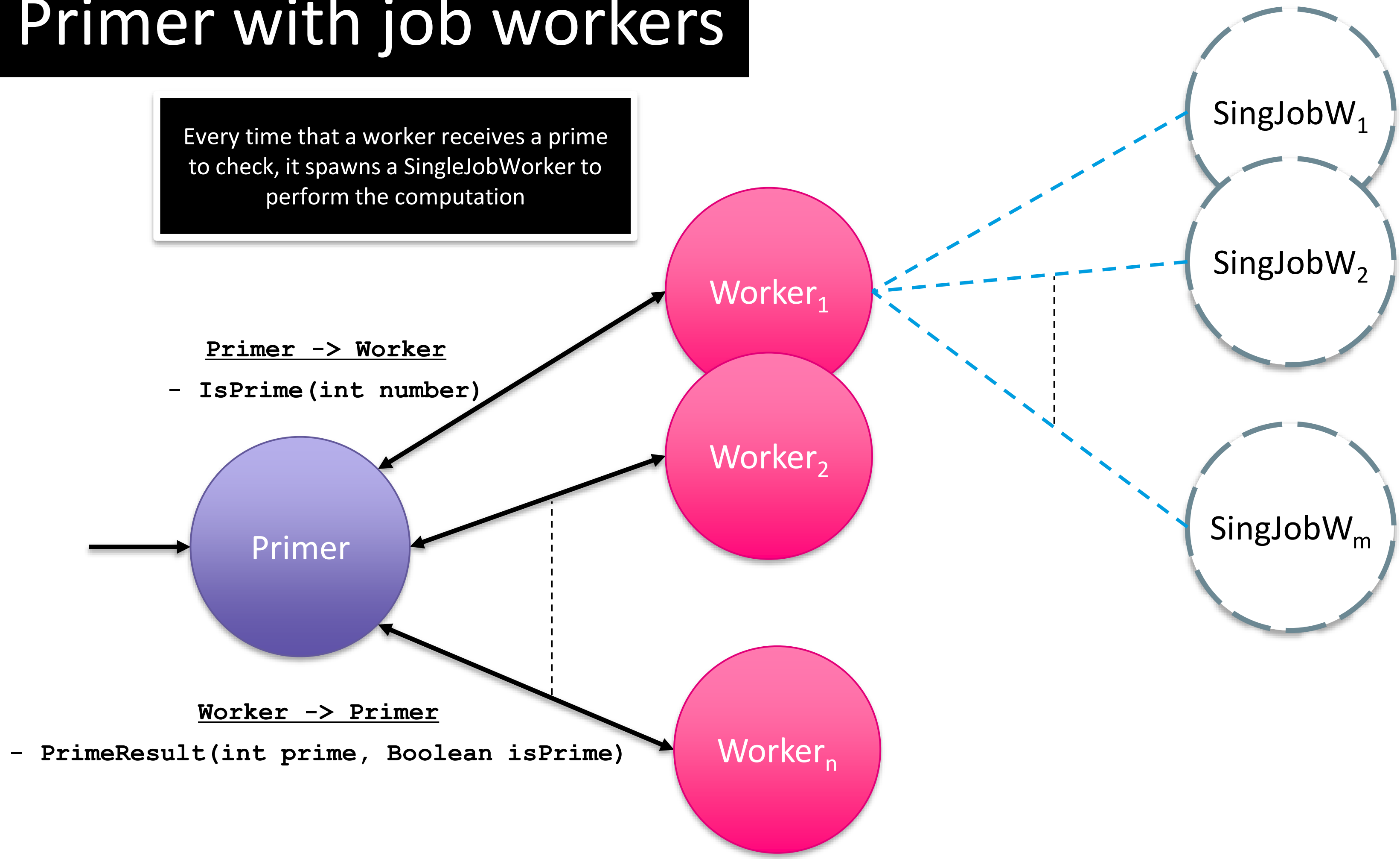
Primer with job workers

- 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

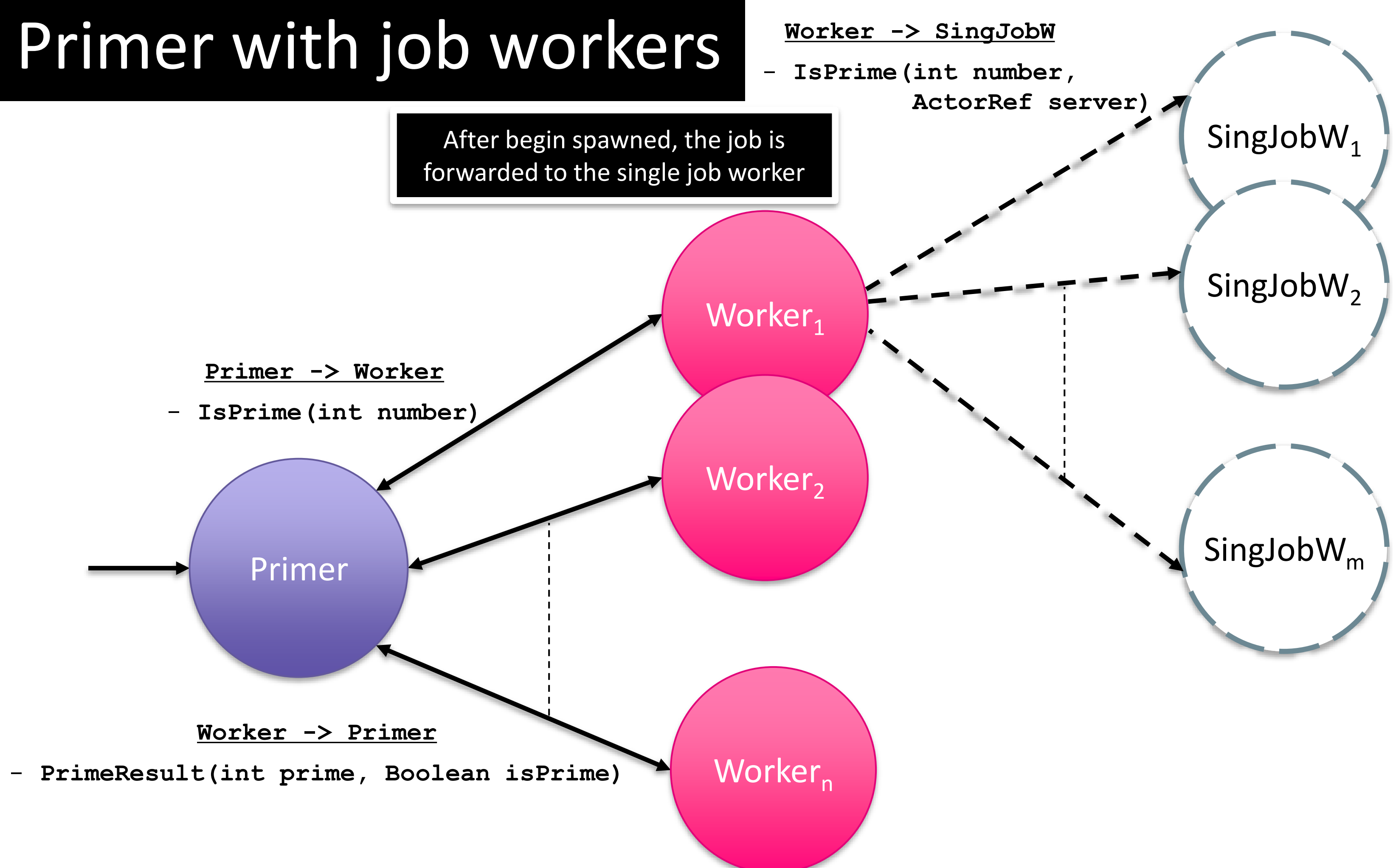
- After these dedicated actors have finished the computation they report the result and terminate the execution

Primer with job workers

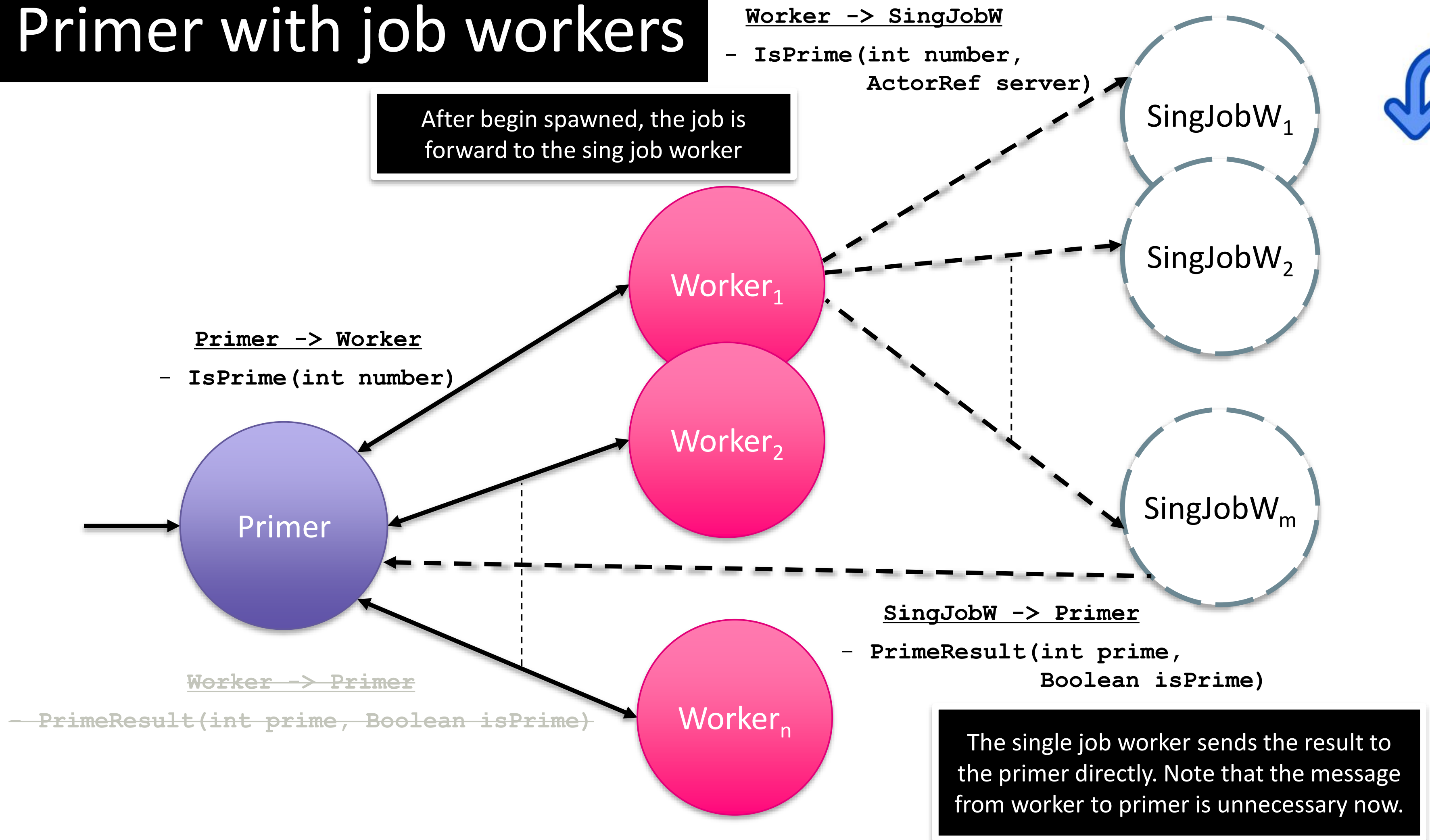
Every time that a worker receives a prime to check, it spawns a SingleJobWorker to perform the computation



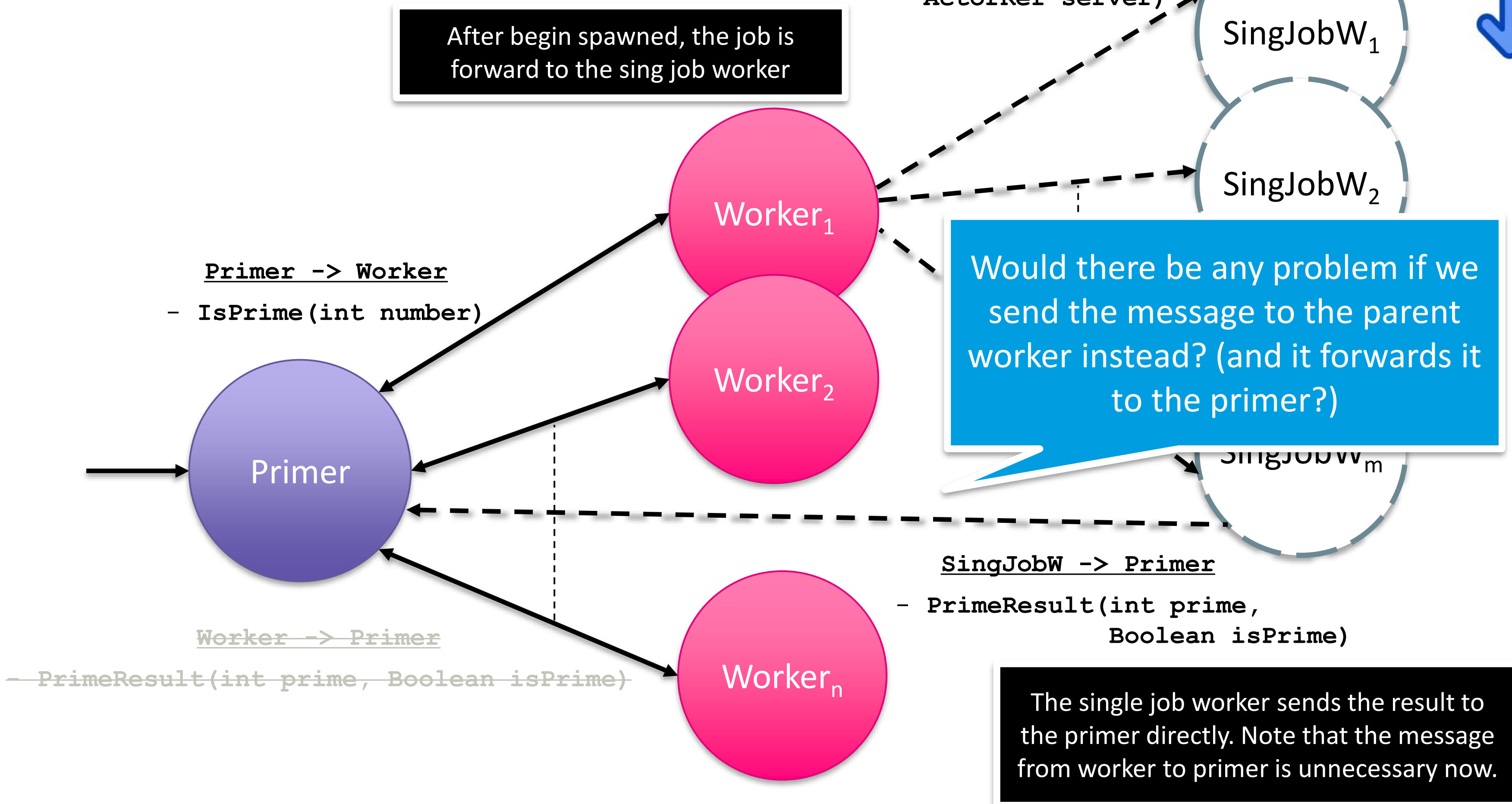
Primer with job workers



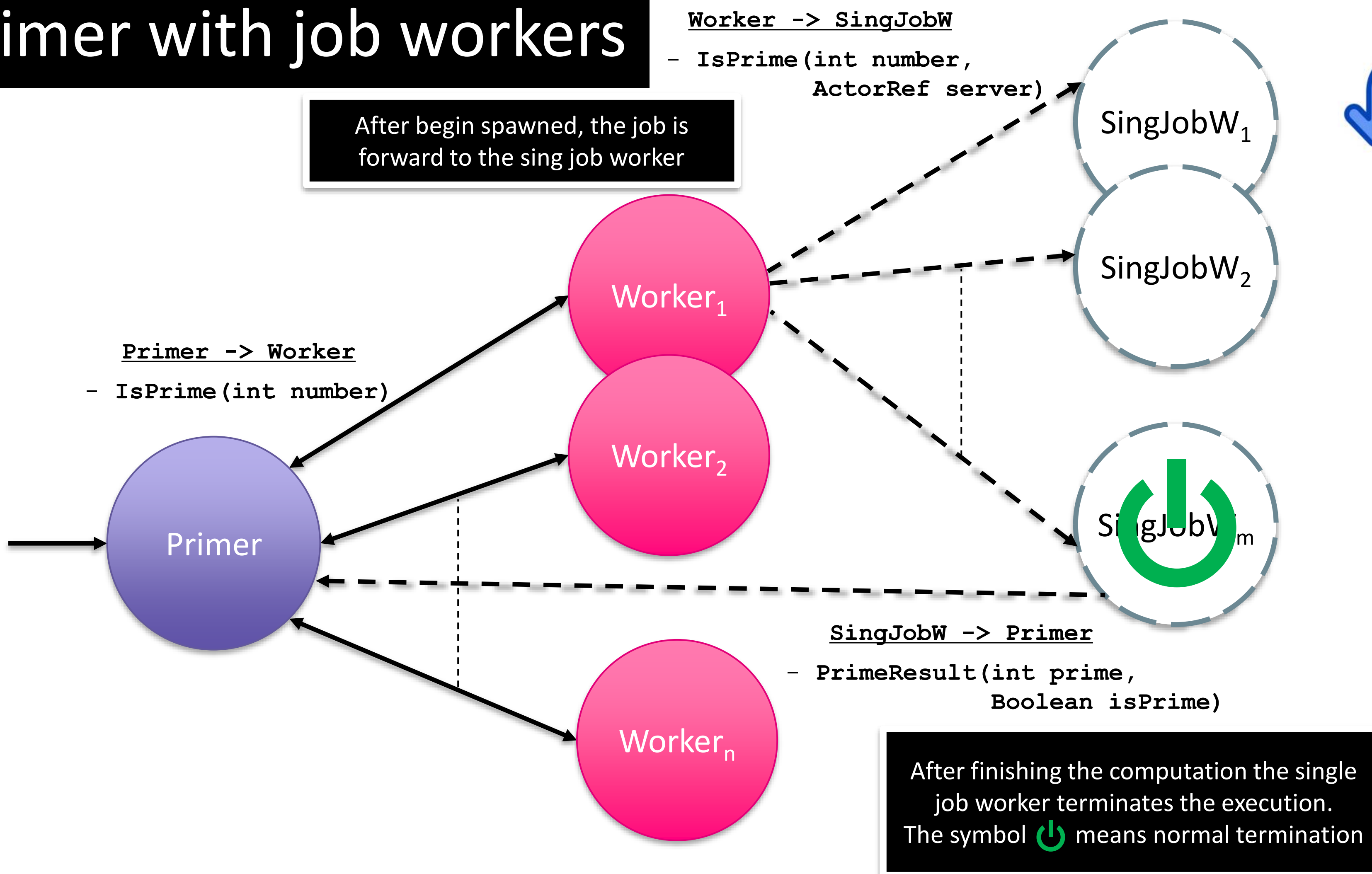
Primer with job workers



Primer with job workers



Primer with job workers



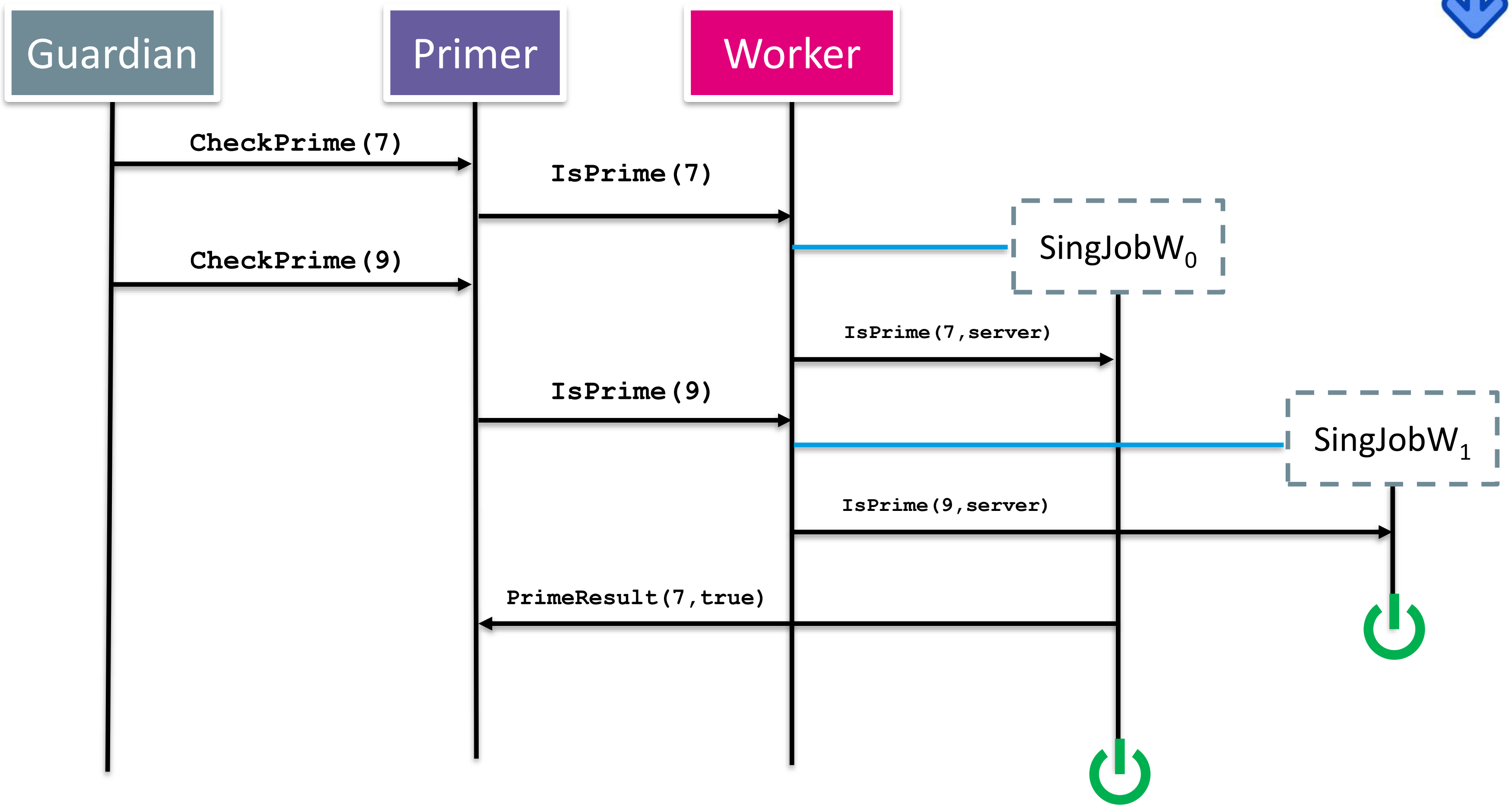
How to shutdown actors in Akka

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

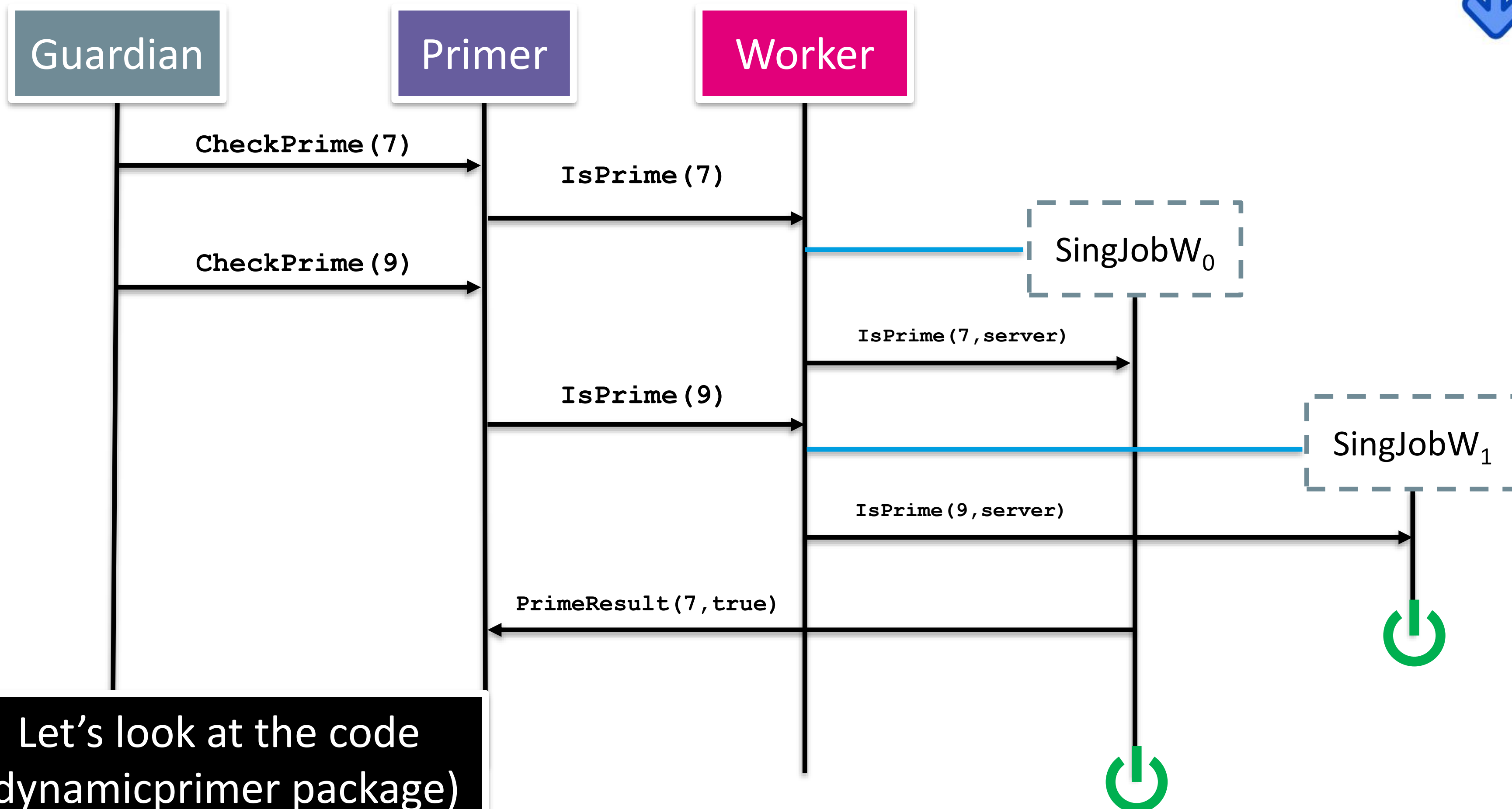
- This behaviour can be used as the return object fo a message handler

```
public Behavior<IsPrime> onIsPrime(IsPrime msg) {
    msg.server.tell(new Primer.PrimeResult(msg.number, isPrime(msg.number)));
    return Behaviors.stopped();
}
```

Primer with job workers – execution example

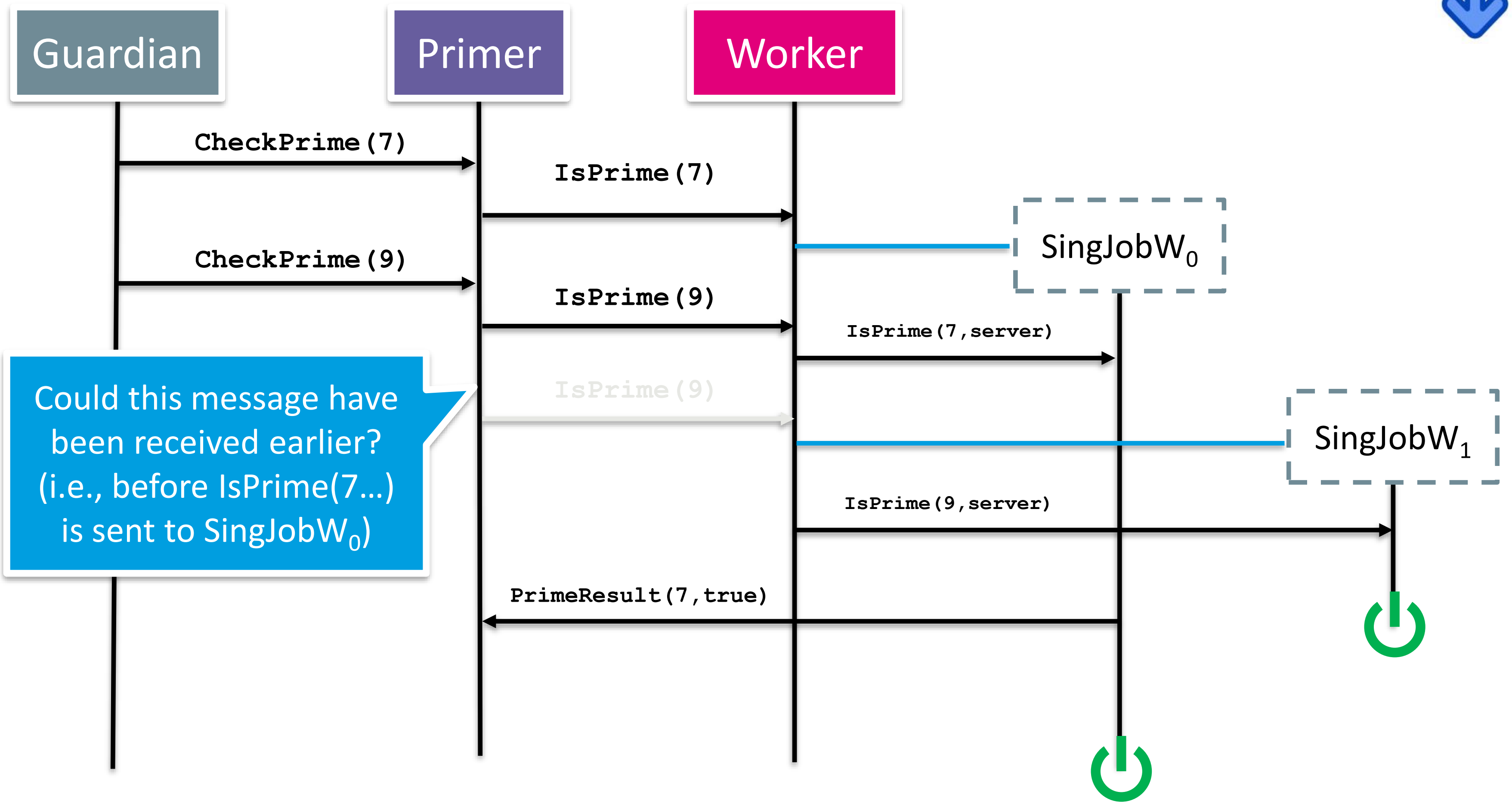


Primer with job workers – execution example

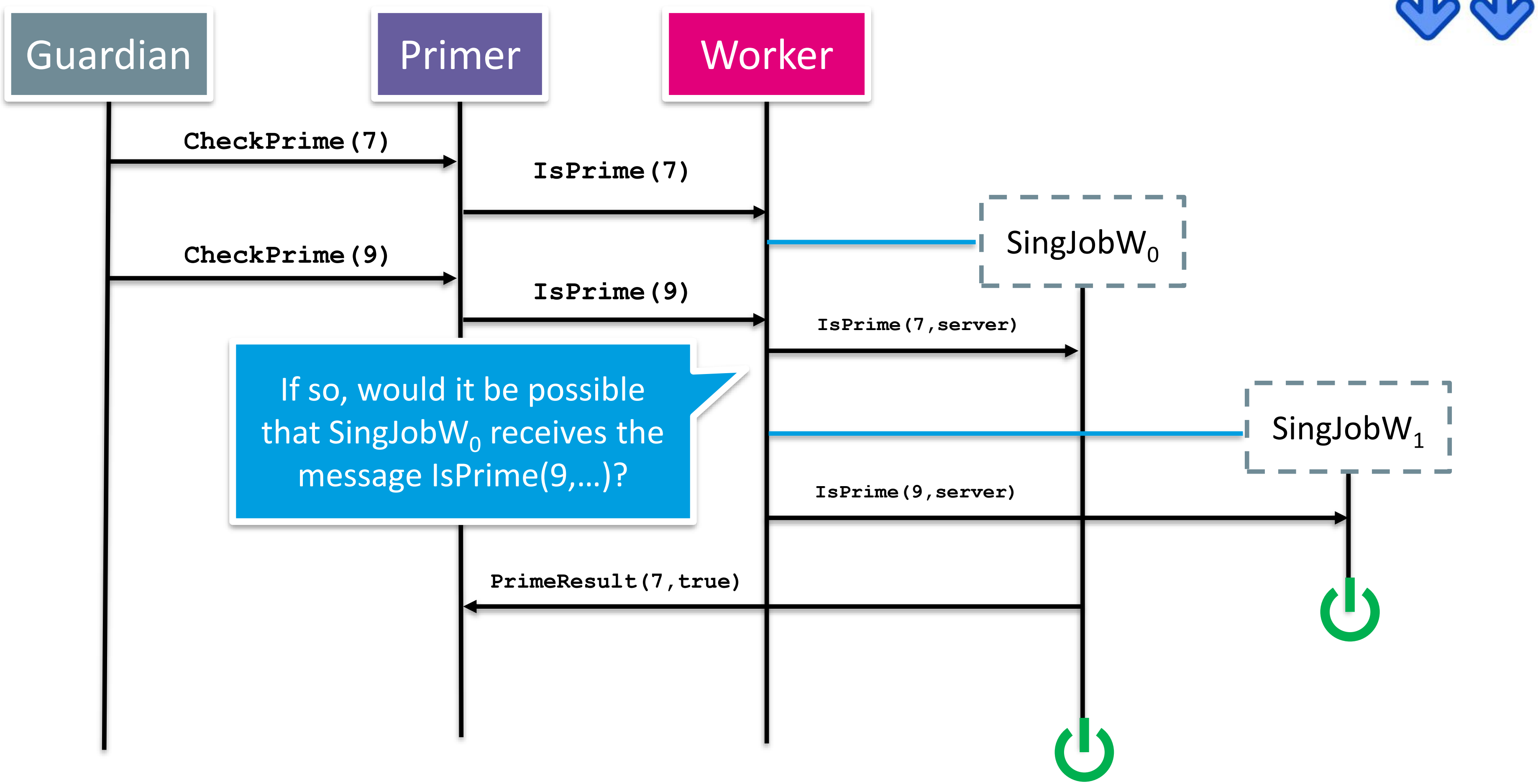


Let’s look at the code
(dynamicprimer package)

Primer with job workers – execution example



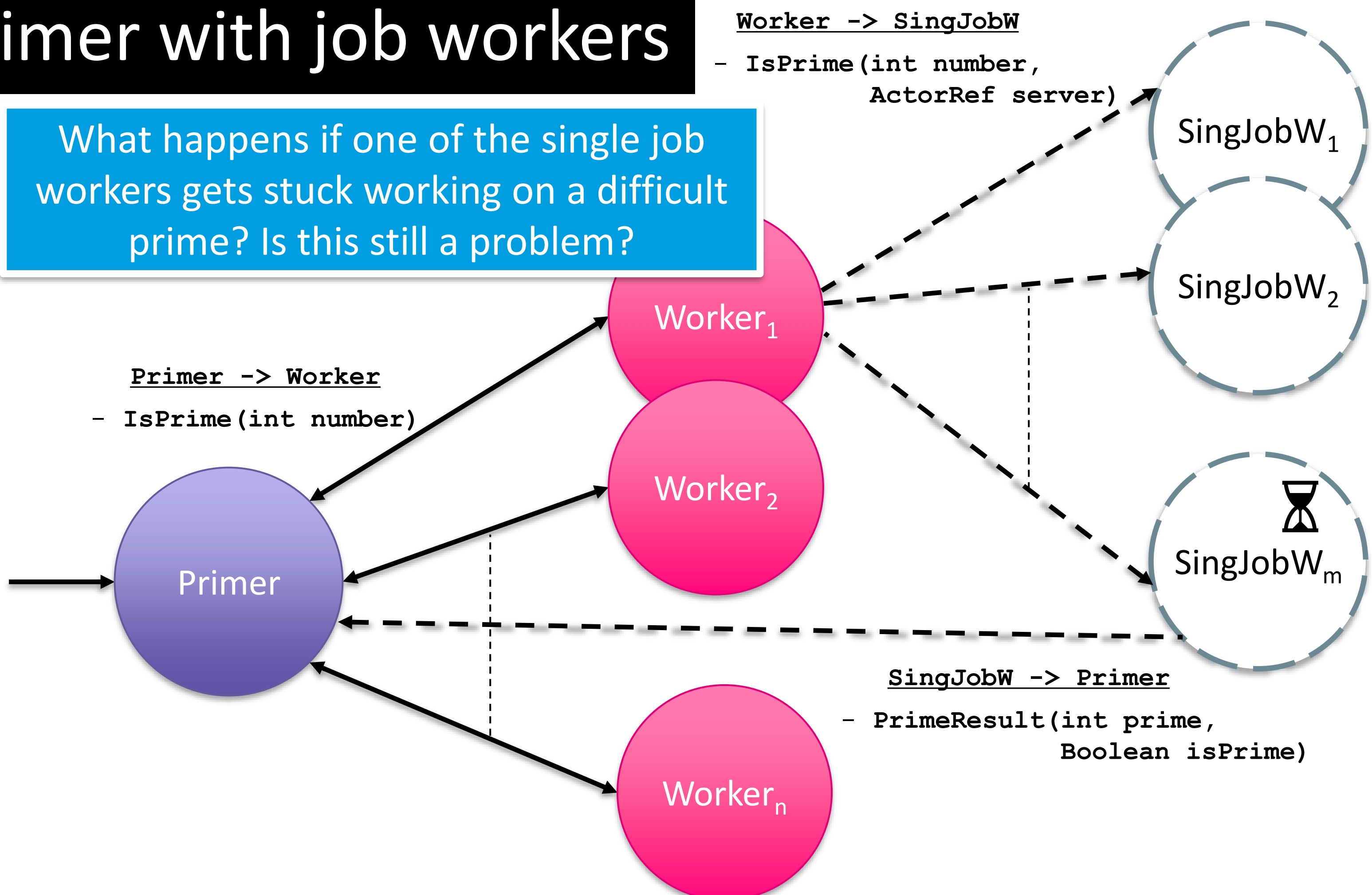
Primer with job workers – execution example



If so, would it be possible that SingJobW$_0$ receives the message IsPrime(9,...)?

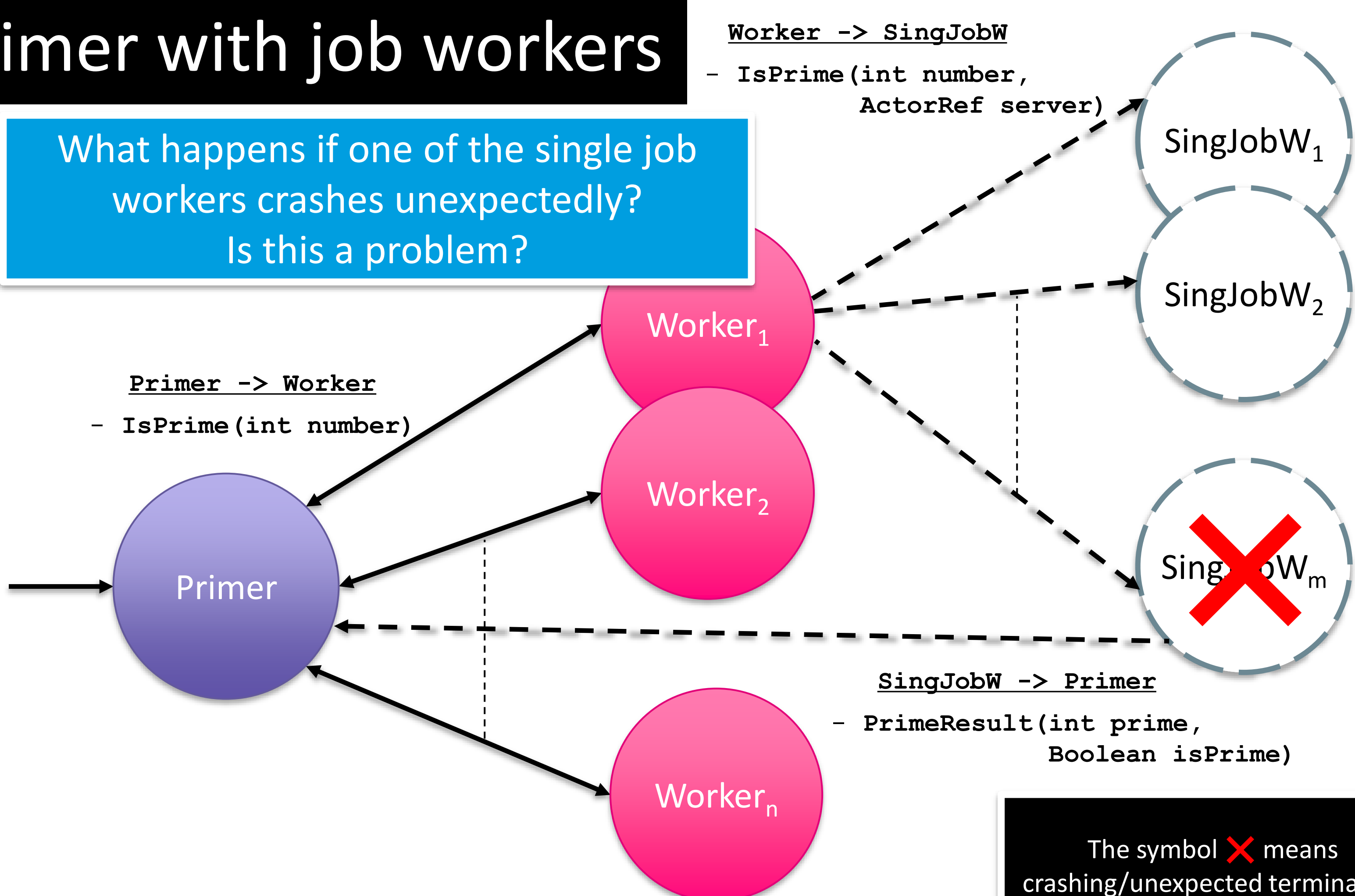
Primer with job workers

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



Primer with job workers

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

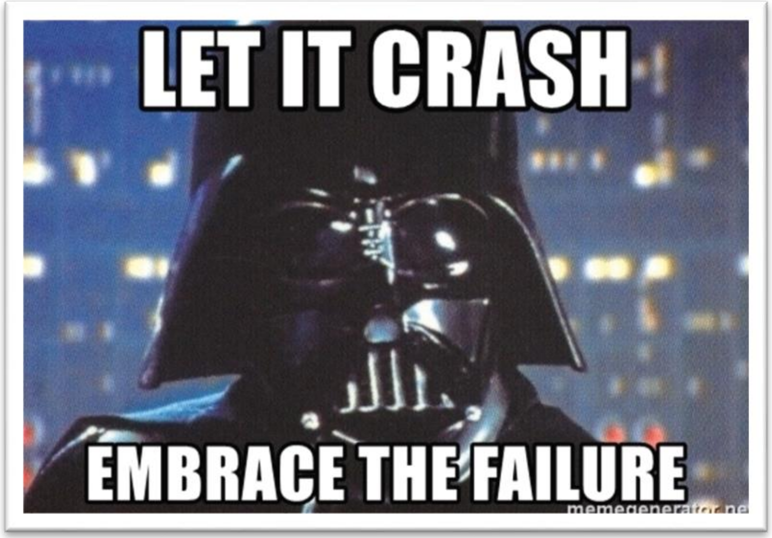


The symbol ❌ means crashing/unexpected termination

Fault-tolerance in Akka

Let it crash! model

- 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



Supervision in Akka

- 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

Handling signals in Akka

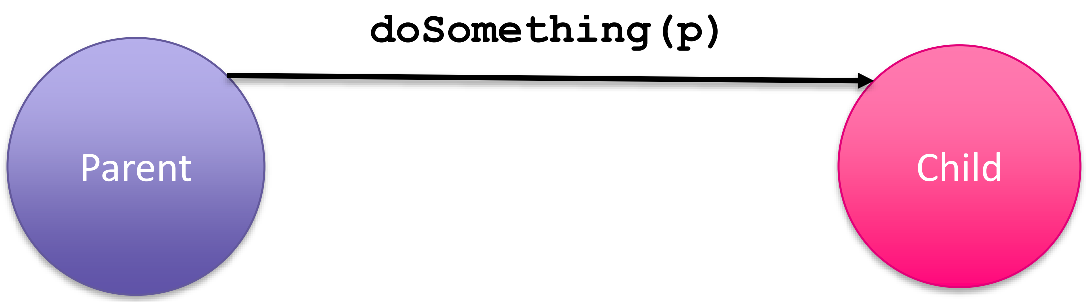
- A *signal* can be seen as a message that is automatically sent by Akka
- For a watcher to handle signals, it *must* have a message handler with **`onSignal(Signal.class, Function f)`**
- The message send by the signal contains a reference to the sender actor. It can be accessed with `getRef()`.

```
/* --- 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();
}
```

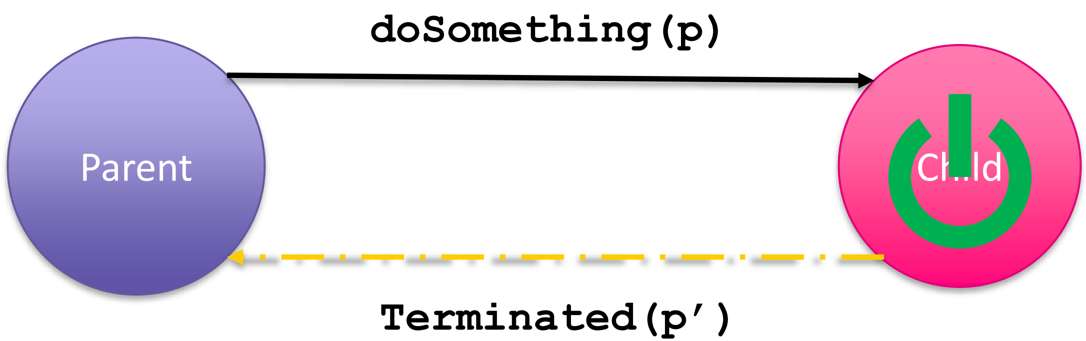
Note I: When processing a message/signal, the message handler picks the handler that first matches the class of the message.

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)

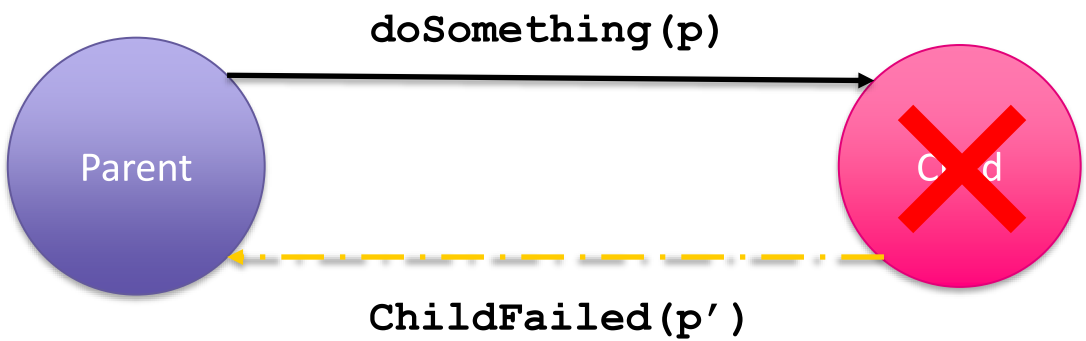


Actor supervision (graphically)



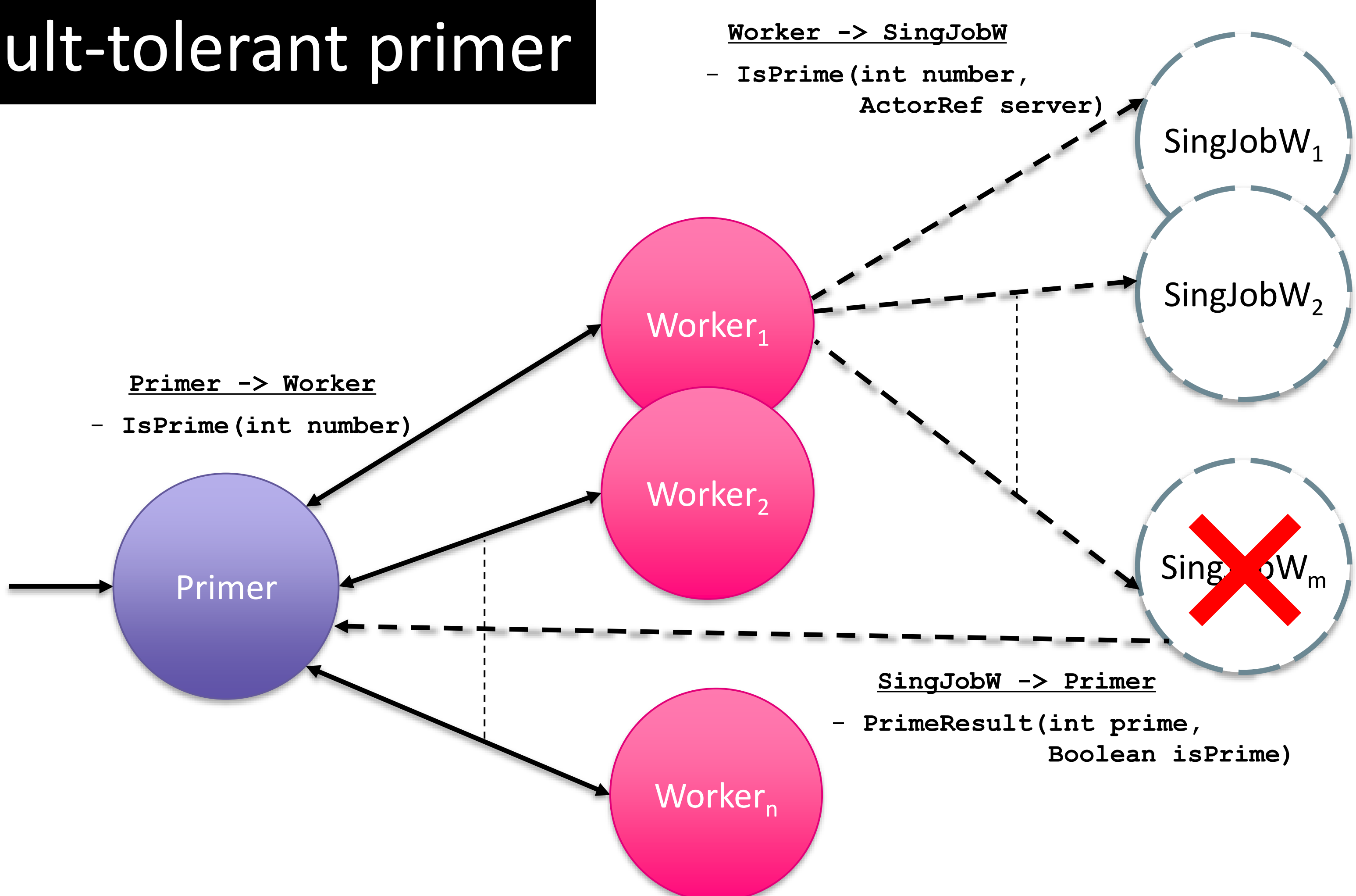
The dashed-dotted yellow arrow indicates the sending of a signal. These are sent automatically by Akka as part of the supervision functionality. If the child finishes normally, it sends a Terminated signal.

Actor supervision (graphically)



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

Fault-tolerant primer

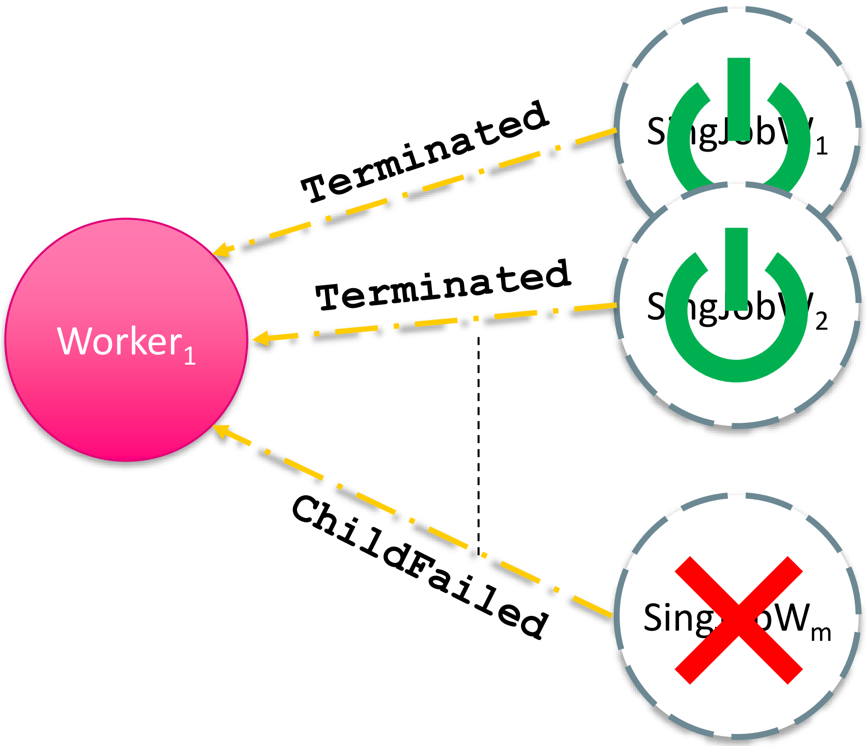


Fault-tolerant primer

- 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



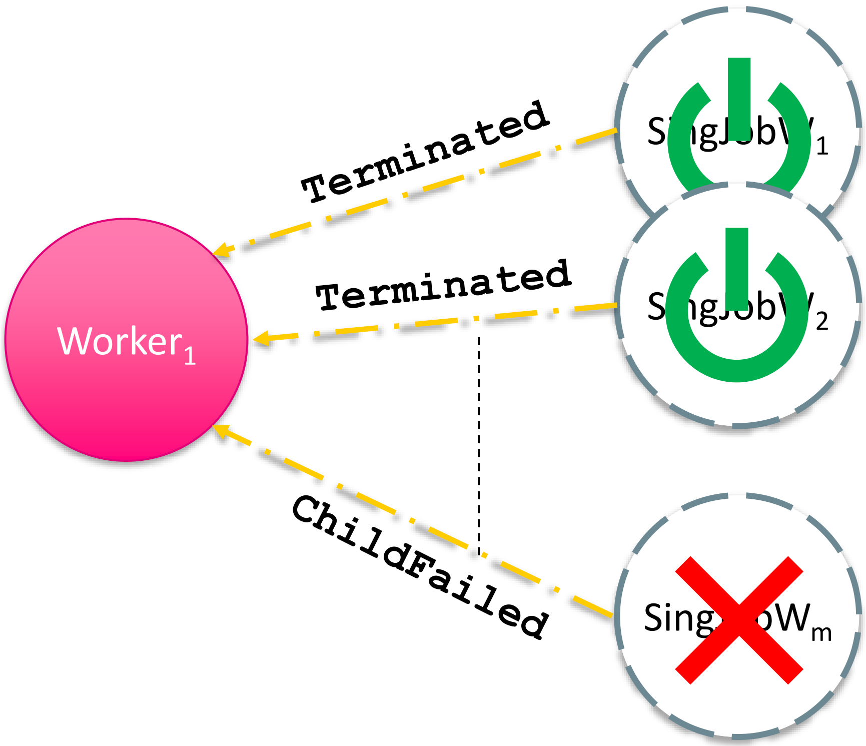
Fault-tolerant primer

- 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

Do we need to extend the state of Worker actors to handle fault-tolerance?

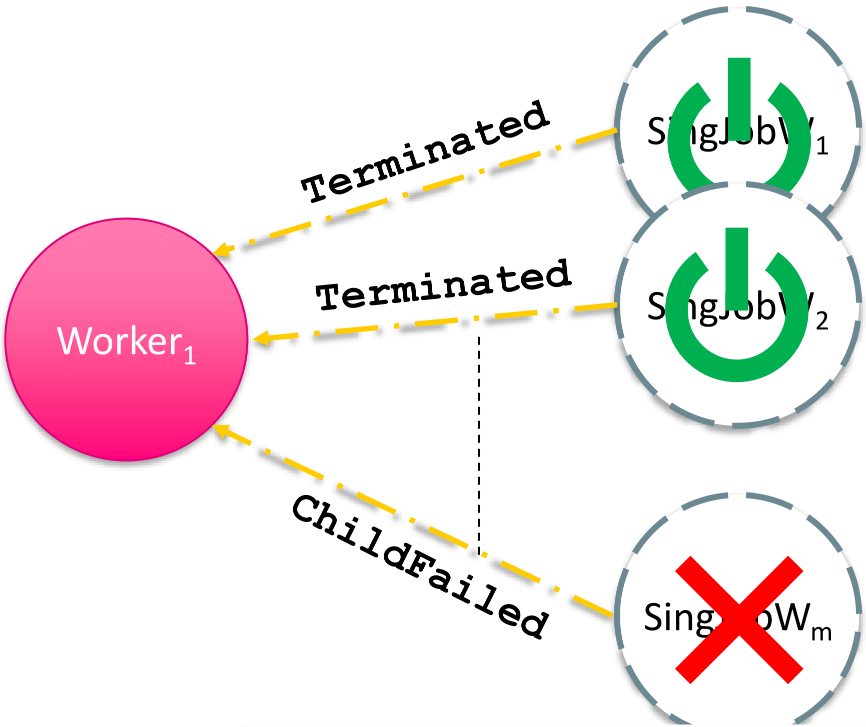


Fault-tolerant primer

- 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



Do we need to extend the state of Worker actors to handle fault-tolerance?

Let's look at the code (faulttolerantprimer package)

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 systems

- Scatter-Gather is a common design patter in distributed systems that can be easily implemented with actors

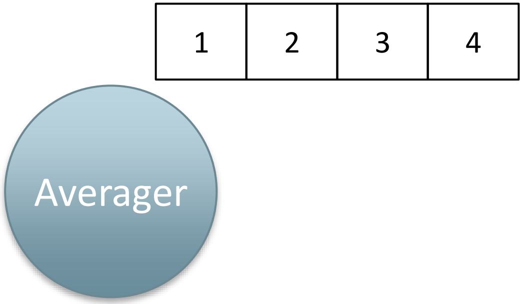
- Typically, the level of scattering (i.e., number of spawned actors) depends on the size of the problem to solve (dynamic load balancing)
  - But it can also be limited by other factors, e.g., CPU or memory usage

- A scatter-gather systems contains two main type of actors
  - <u>Scatterer</u>: 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.
  - <u>Gatherer</u>: Receives pieces of data from scatterers, and combines them into a single piece of data performing

Average and scatter-gather

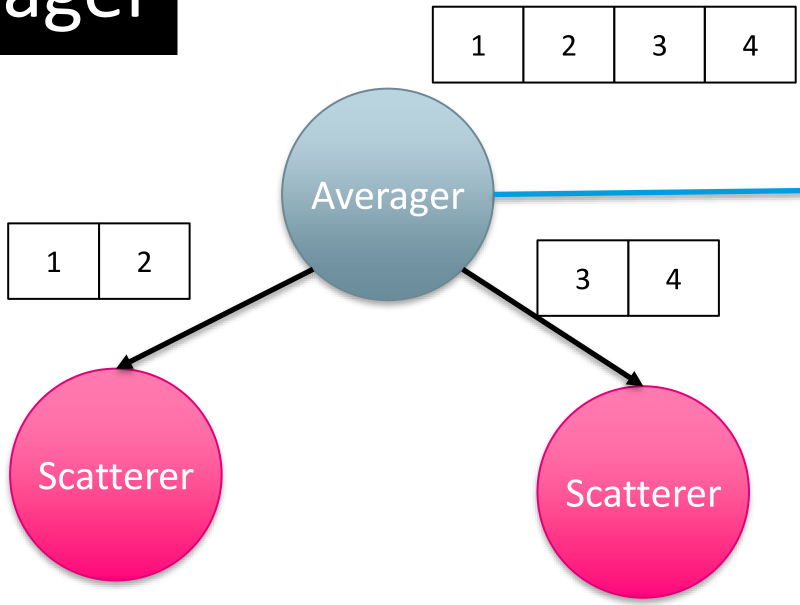
- A problem for which this pattern is suitable is computing the average of a list of numbers
- Given a set of natural numbers $a_1, a_2, \ldots, a_n$, the average is $\frac{1}{n}\sum_i a_i$
  - Note that this is equivalent to $\sum_i \frac{a_i}{n}$
- In a nutshell, we can have scatterer actors splitting computation and computing each factor $\frac{a_i}{n}$, and gatherers summing up the results

Averager



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

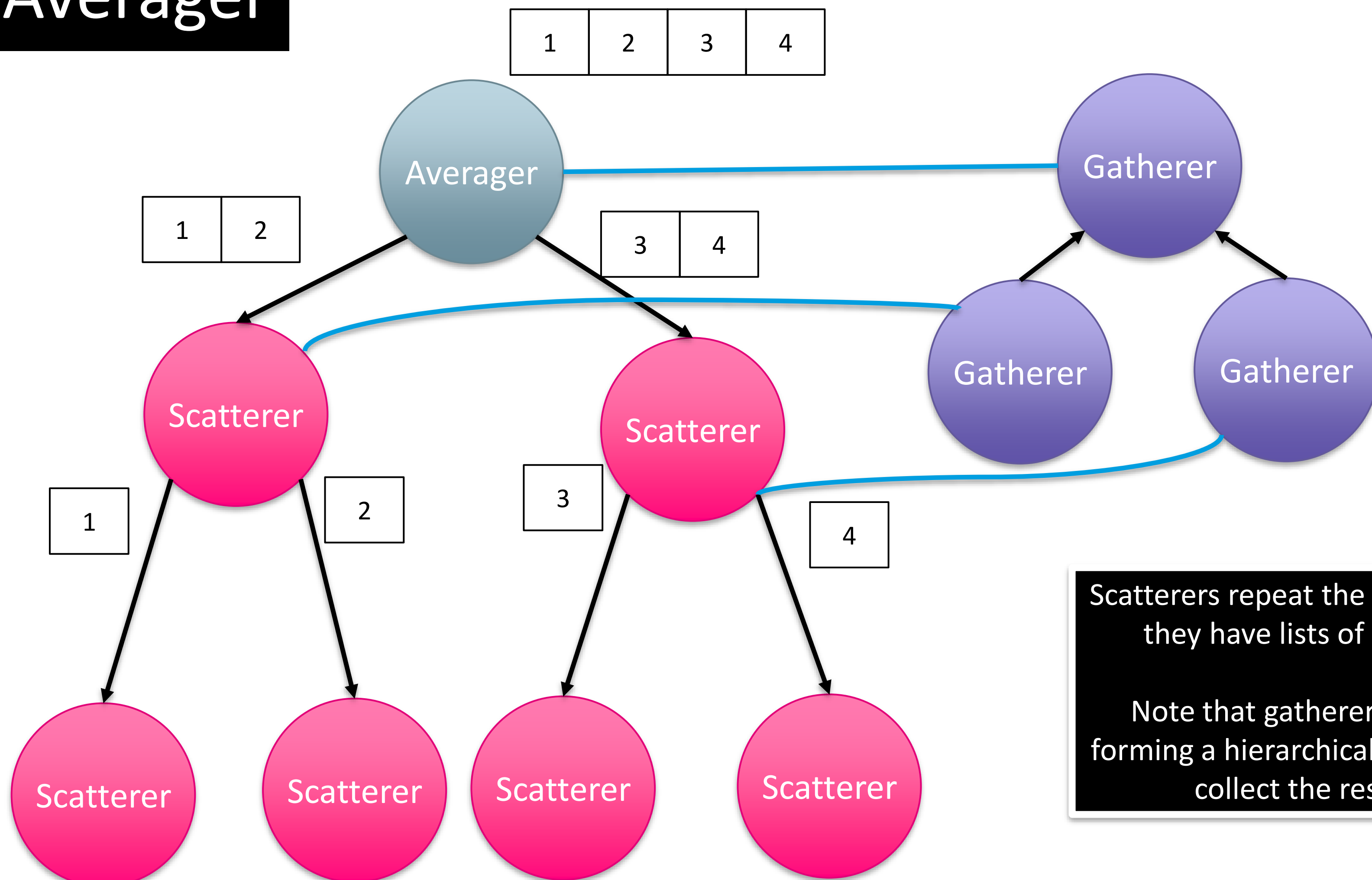
Averager



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

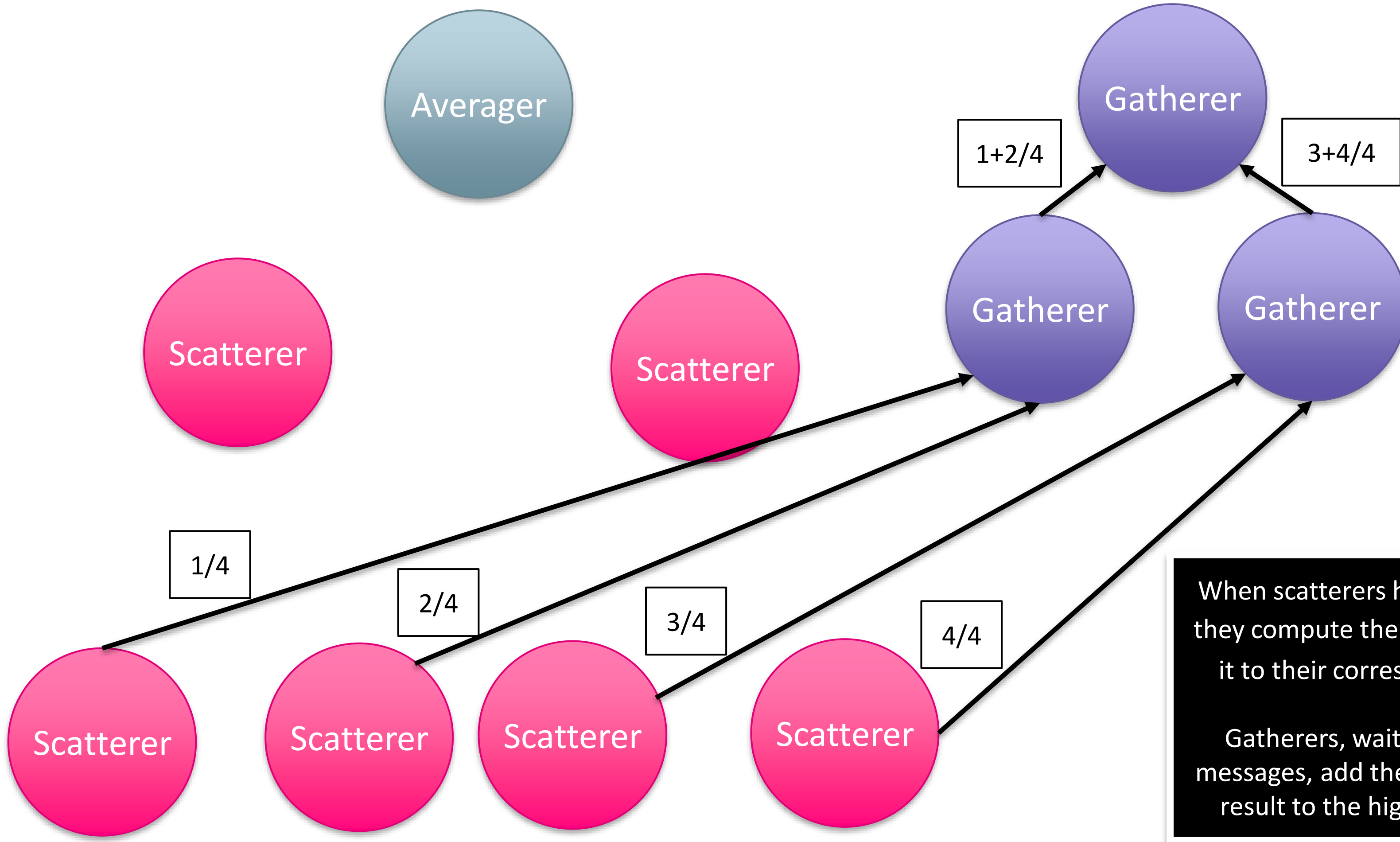
Averager



Scatterers repeat the process until they have lists of size one.

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

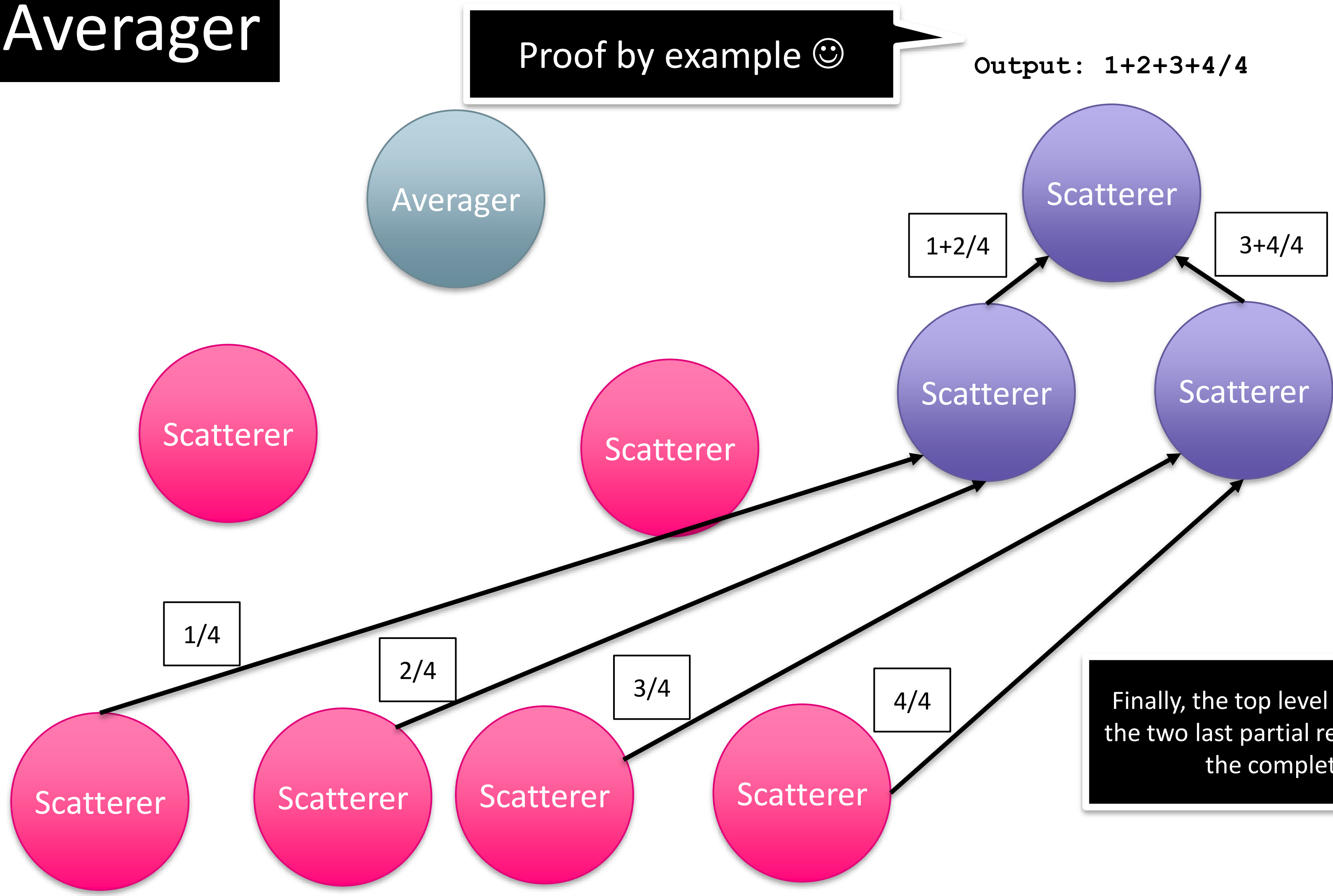
Averager



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

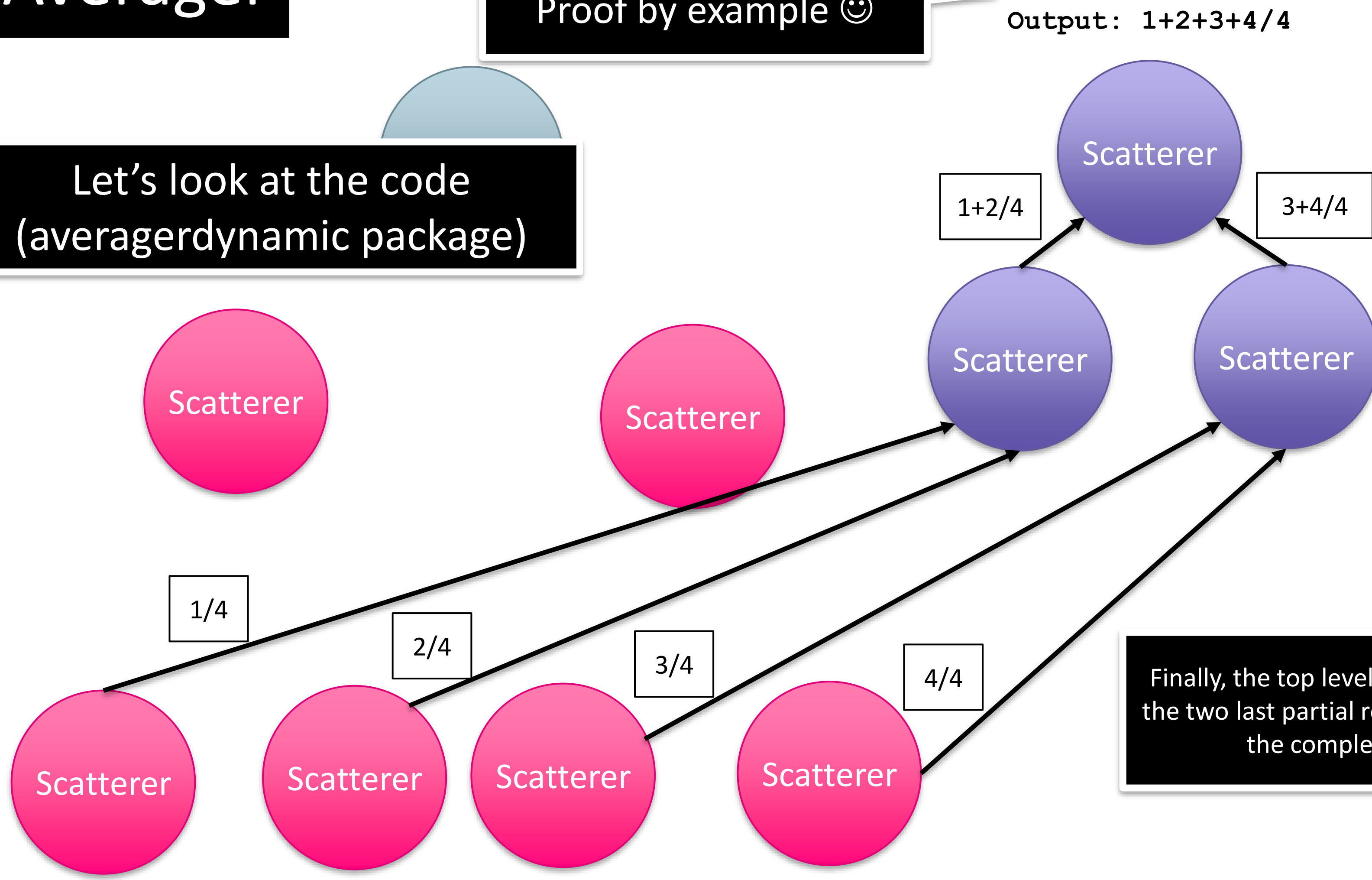


Finally, the top level gatherer adds up the two last partial results and outputs the complete result.

Averager

Proof by example ☺

Let's look at the code
(averagerdynamic package)



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

Other approaches to load balancing

- 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

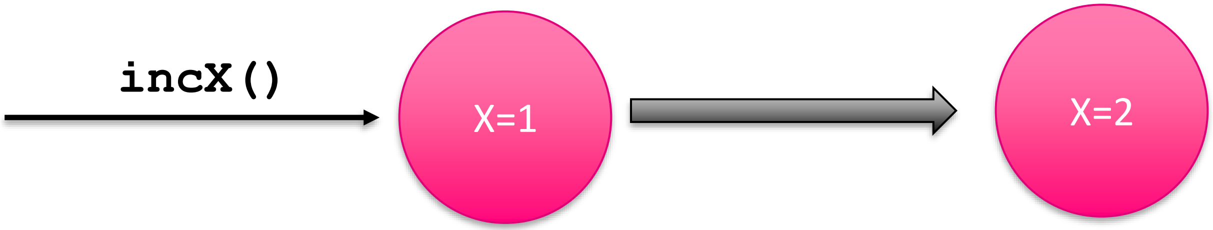
- Another example of adaptive load balancing are elastic systems
  - Elastic systems try to keep a number of active actors proportional to the workload
  - Several exercises for this week target implementing an elastic server

Changing behaviour

Actors with changing behaviour

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

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

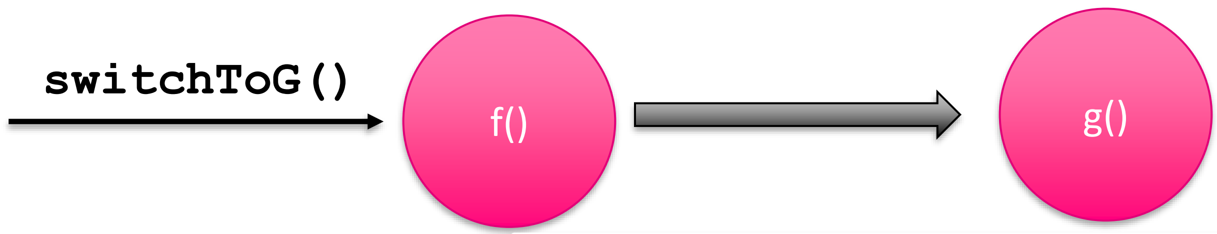


The actor receives a message and updates its internal state. The arrow ➡ models state transition

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)



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!

Changing behaviour in Akka

- 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

Averager with changing behaviour

- 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

Averager with changing behaviour

- 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)

Actors in distributed systems

The actors model has natural mapping in distributed systems

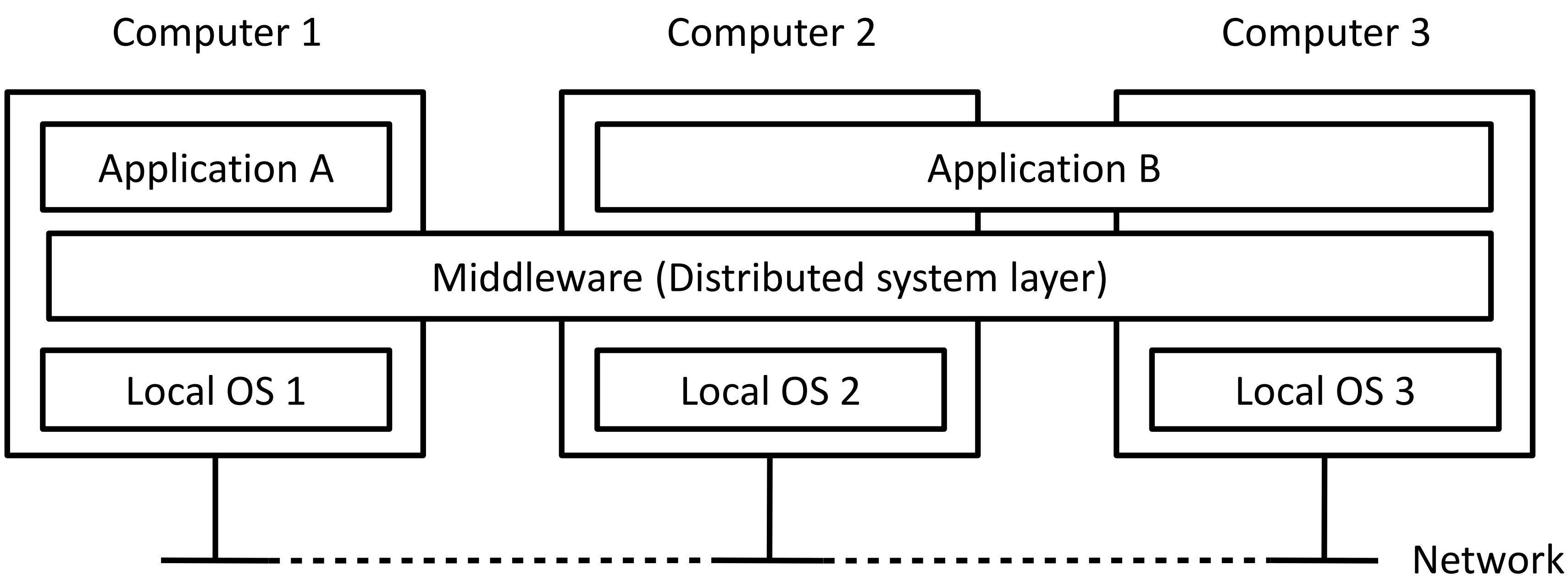
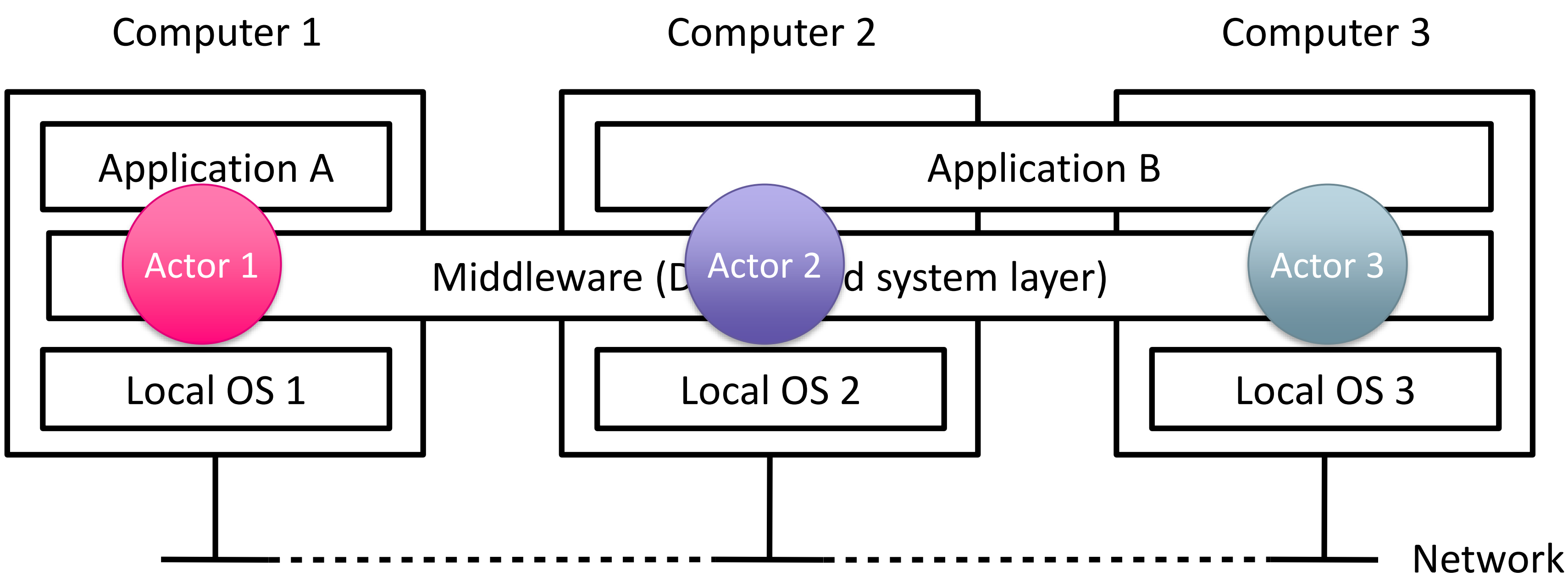


Figure taken from -> Distributed Systems: Principles and Paradigms. Andrew S. Tanenbaum and Maarten Van Steen. 2007.

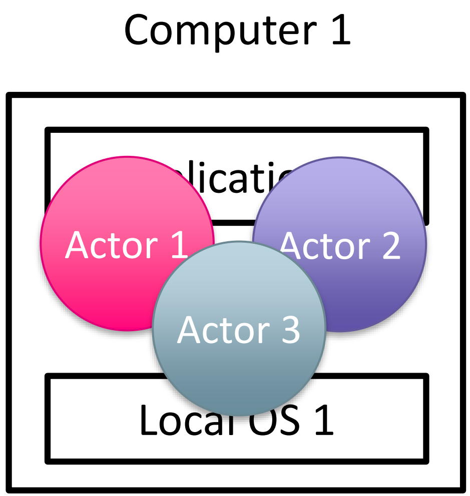
Actors in distributed systems

The actors model has natural mapping in distributed systems



Actors in a single computer

The actors model is applicable in a single computer as well



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

Agenda

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