Practical Concurrent and Parallel Programming XIII

Message Passing II

Raúl Pardo

IT UNIVERSITY OF COPENHAGEN

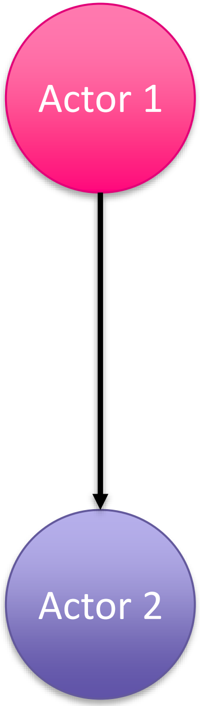
Agenda

- Actors model (revisited)
  - Primer
- Dynamic topology
- Fault-tolerance
- Adaptive load balancing
- Examination

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

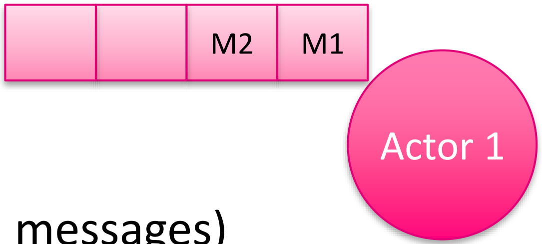
- An actor can be seen as a sequential unit of computation
  - Although, formally, the model allows for parallelism within the actor, one can safely assume that there are not concurrency issues within the actor.
  - You can think of an actor as a thread

- 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



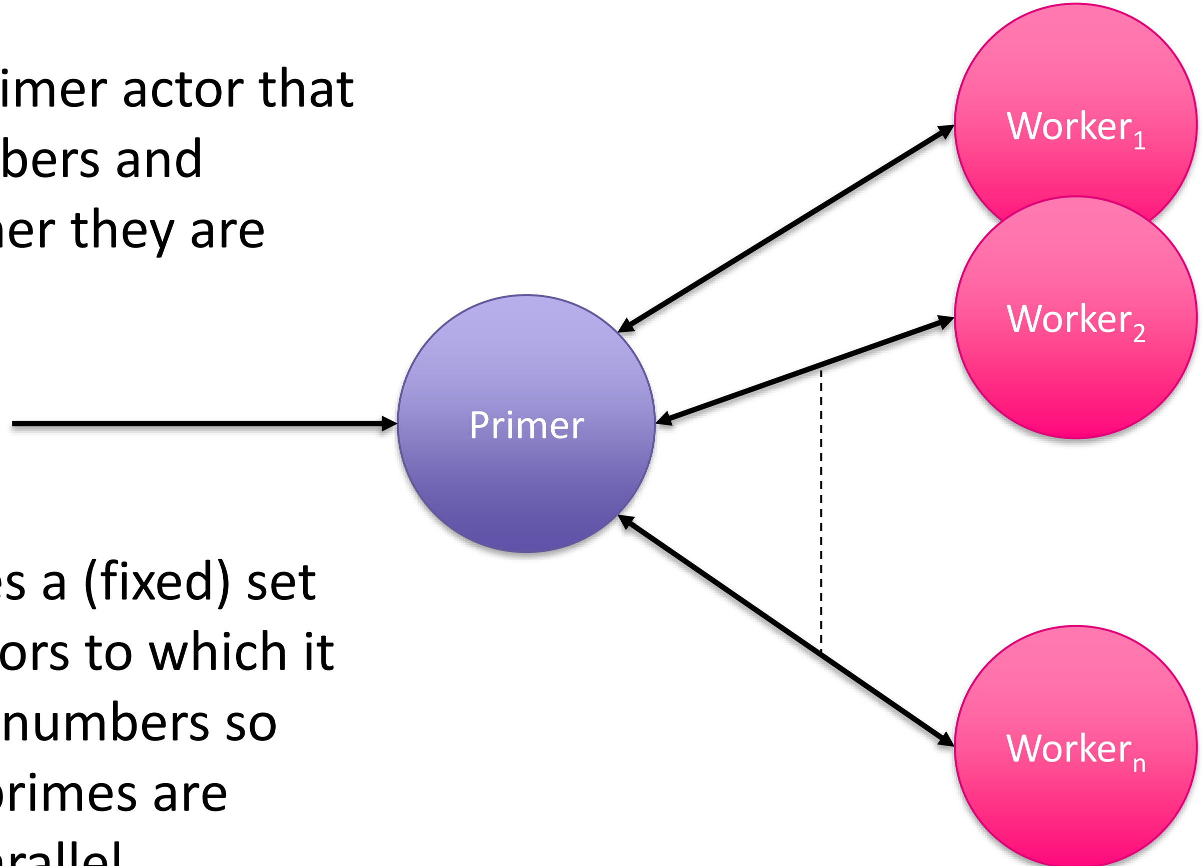
Erlang actors

- There is a one-to-one correspondence of the basic actor operations and concepts in Erlang

Actors Model	Erlang
Actor	Module
Mailbox Address	Process identifier (PID)
Message	Erlang term (typically an atom or tuple)
State	Erlang term (typically a record)
Behaviour	loop()
Create actor	spawn
Send message	PID ! Message
Receive message	receive … end

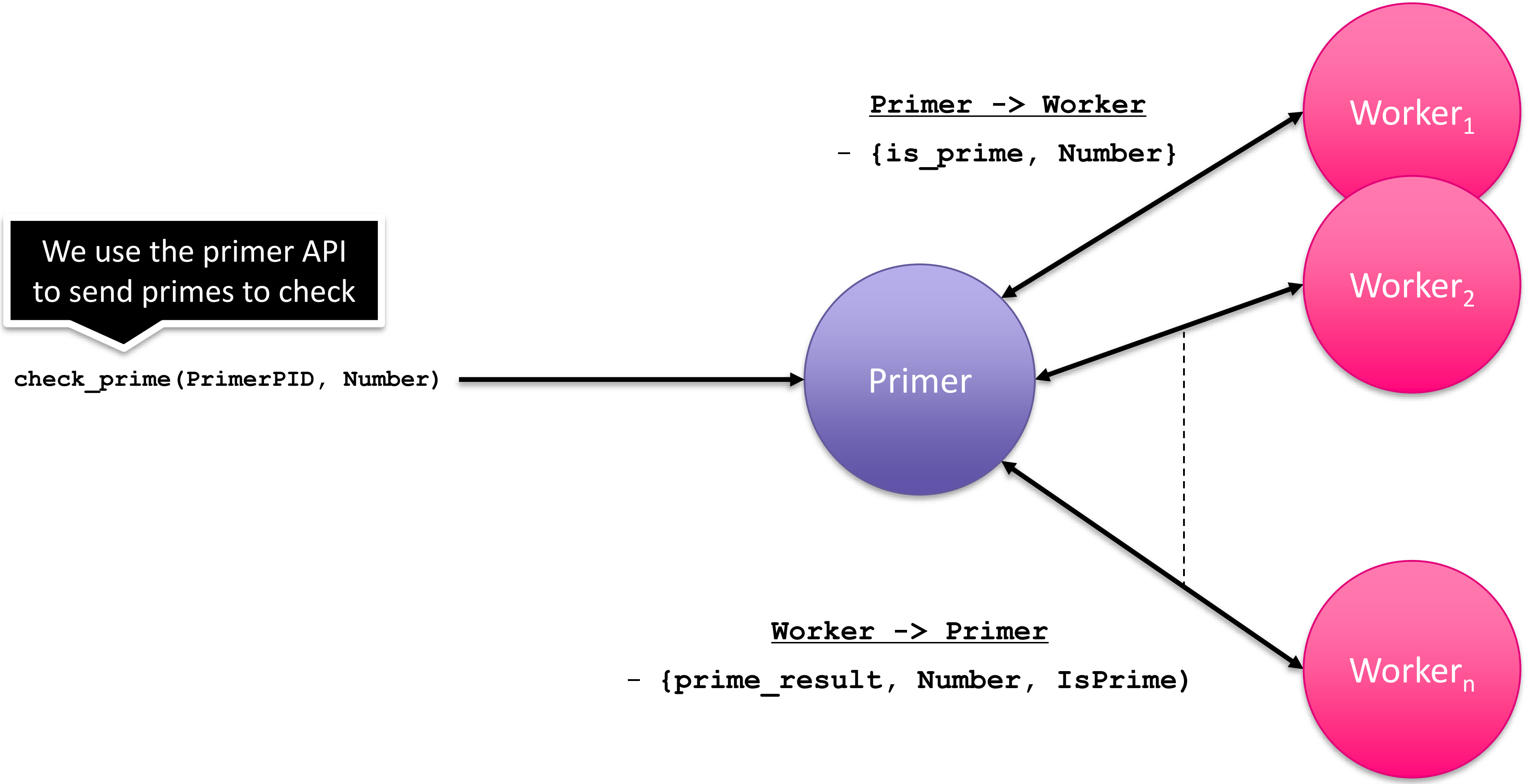
Primer

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

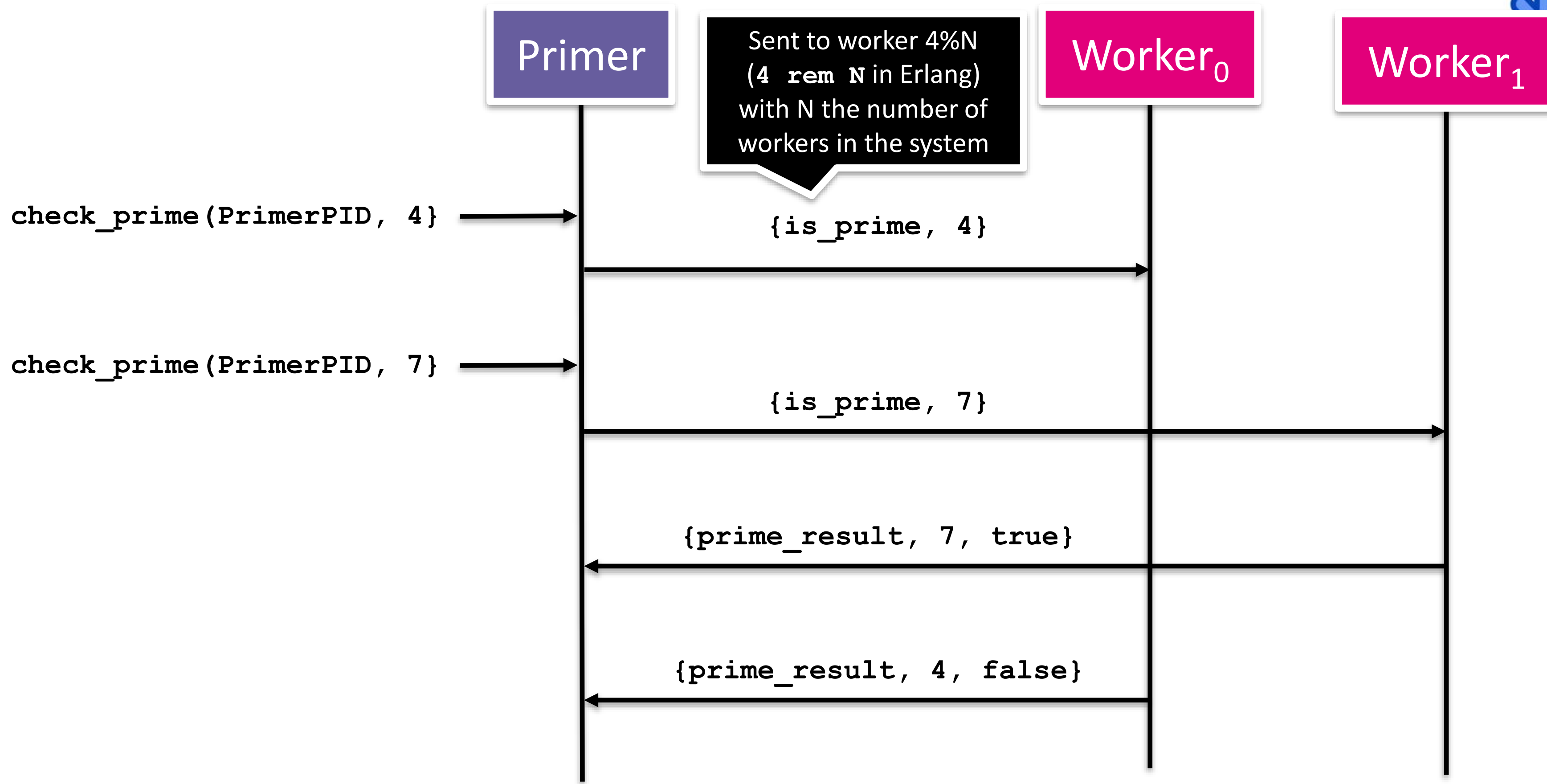


- The actor uses a (fixed) set of worker actors to which it forwards the numbers so that several primes are checked in parallel

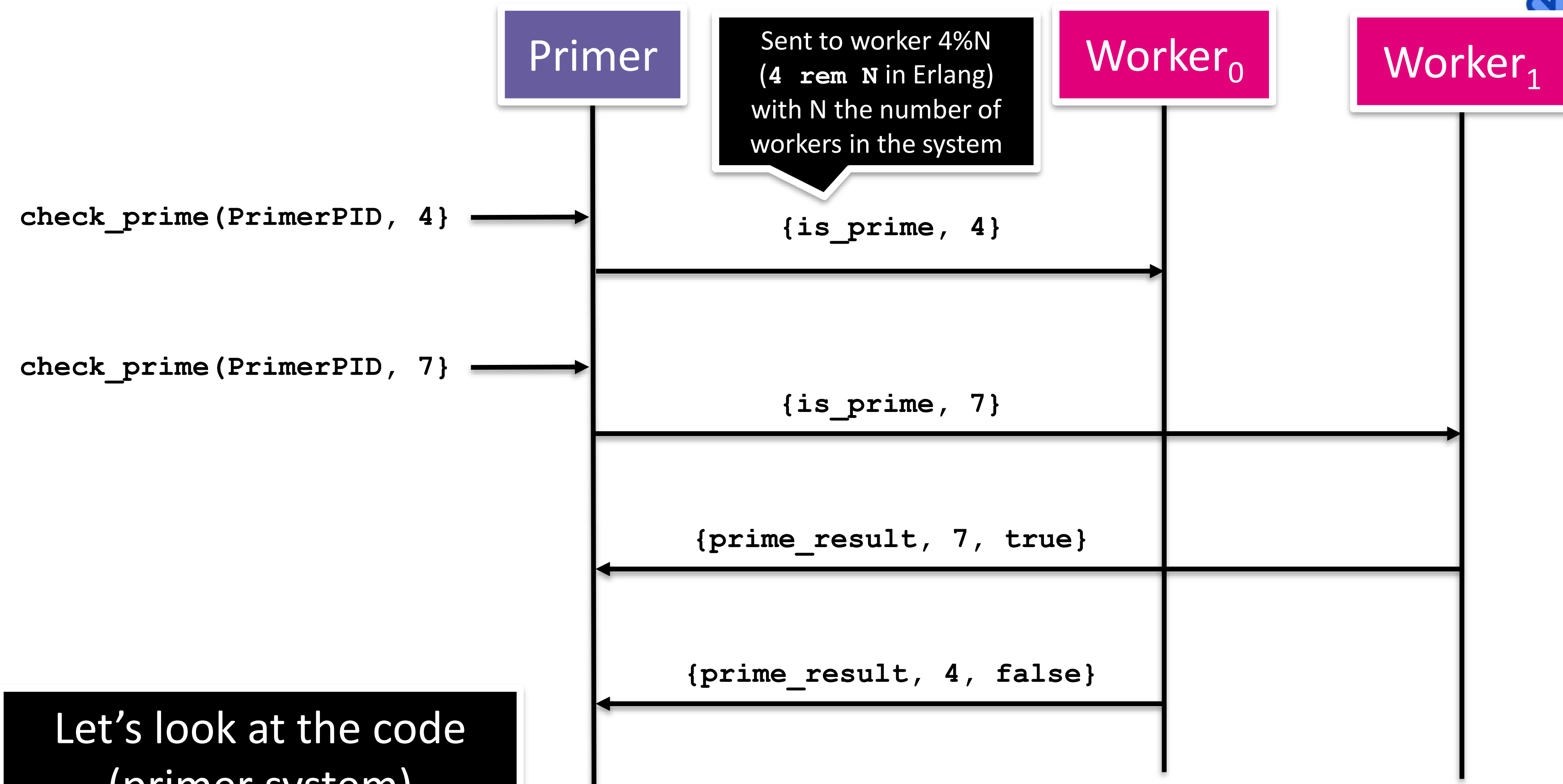
Primer


We use the primer API to send primes to check
check_prime(PrimerPID, Number)
Primer -> Worker
- {is_prime, Number}
Primer
Worker1
Worker2
Workern
Worker -> Primer
- {prime_result, Number, IsPrime)

Primer – execution example



Primer – execution example



Let's look at the code (primer system)

Primer – Printing order

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

```
1> primer:check_primes(7, 2, 100000000).
The number 59934504 is not prime [1]
[{1,59934504},
 {2,32063853},
 {3,57613610},
 {4,87902431},
 {5,58920555},
 {6,20468351},
 {7,31784057}]
The number 57613610 is not prime [2]
The number 32063853 is not prime [3]
The number 87902431 is not prime [4]
The number 58920555 is not prime [5]
The number 20468351 is not prime [6]
The number 31784057 is prime [7]
```

Primer – Printing order

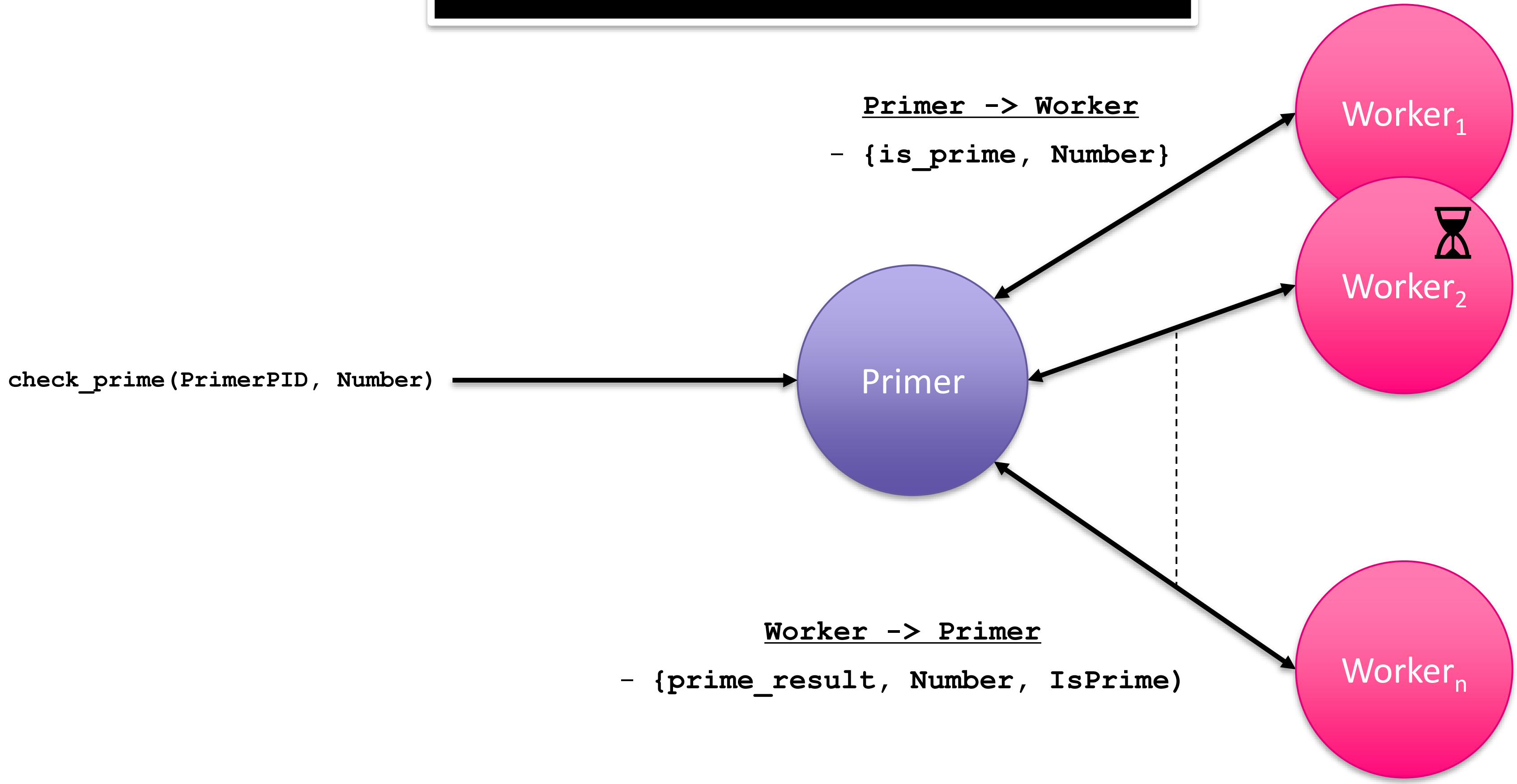
- Note that the printing order of the results does not correspond to the order of sending the requests

```
1> primer:check_primes(7, 2, 100000000).
The number 59934504 is not prime [1]
[{1,59934504},
 {2,32063853},
 {3,57613610},
 {4,87902431},
 {5,58920555},
 {6,20468351},
 {7,31784057}]
The number 57613610 is not prime [2]
The number 32063853 is not prime [3]
The number 87902431 is not prime [4]
The number 58920555 is not prime [5]
The number 20468351 is not prime [6]
The number 31784057 is prime [7]
```

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

Primer

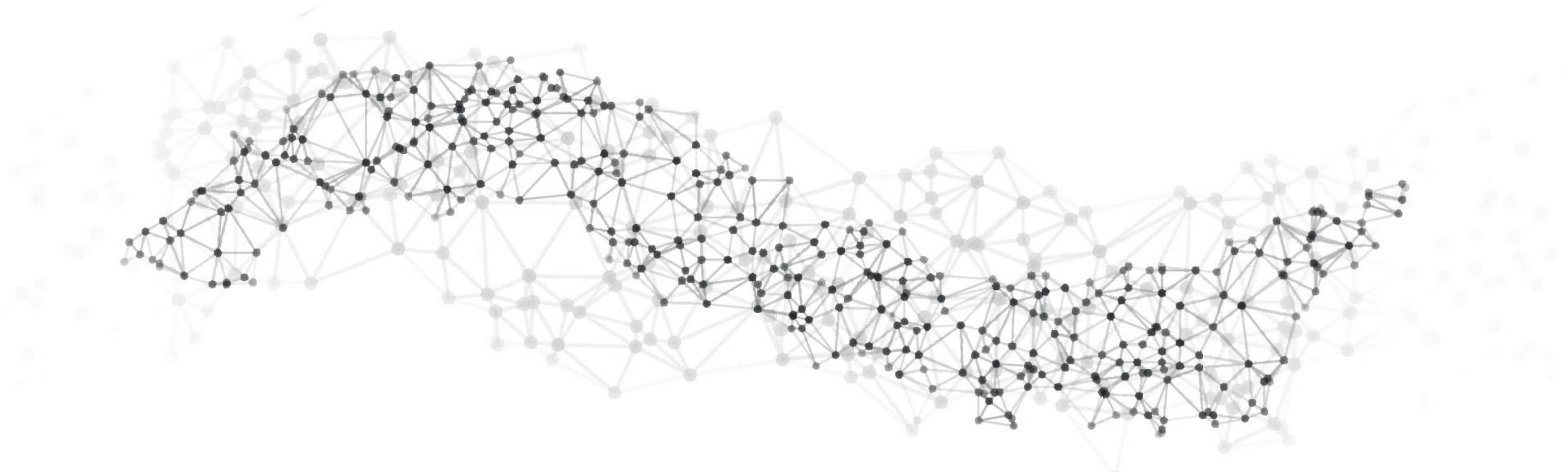
What happens if one of the workers gets stuck working on a difficult prime?



Actors systems with *dynamic* topology

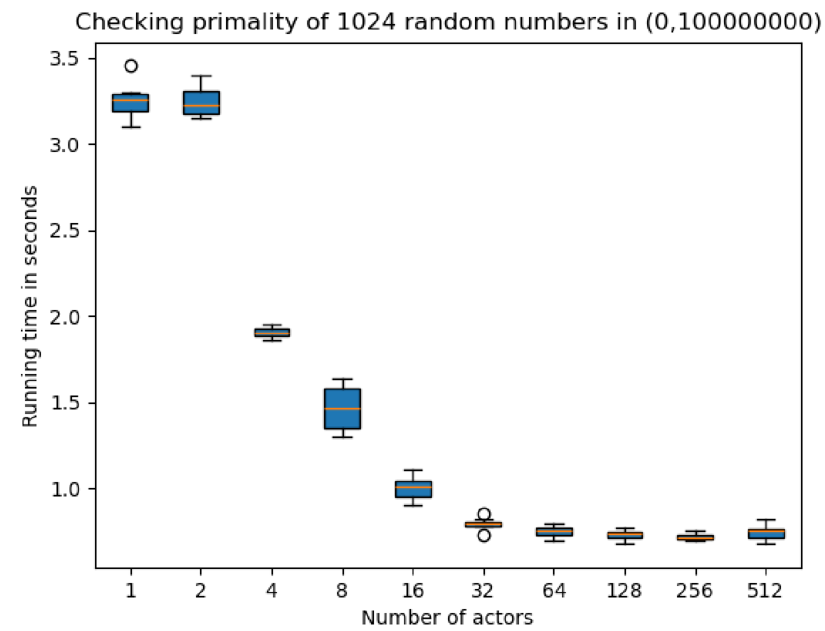
Don’t be shy, spawn actors!

- The Actor 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 system to check 1024 numbers between 0 and 100000000
  - Not very strong statistics (8 runs for each number of actors)
- Erlang implements processes efficiently
  - *“Erlang's processes take about 300 words of memory each and can be created in a matter of microseconds—not something doable on major operating systems these days.”* [Learn You Some Erlang for great good! Chapter: The Hitchhiker's Guide to Concurrency]
- However, actor systems can be distributed among many computers (Erlang shell nodes in the case of Erlang) and computers
  - We are not limited to a single computer throughput
  - See distributed Erlang in Learn You Some Erlang for Great Good

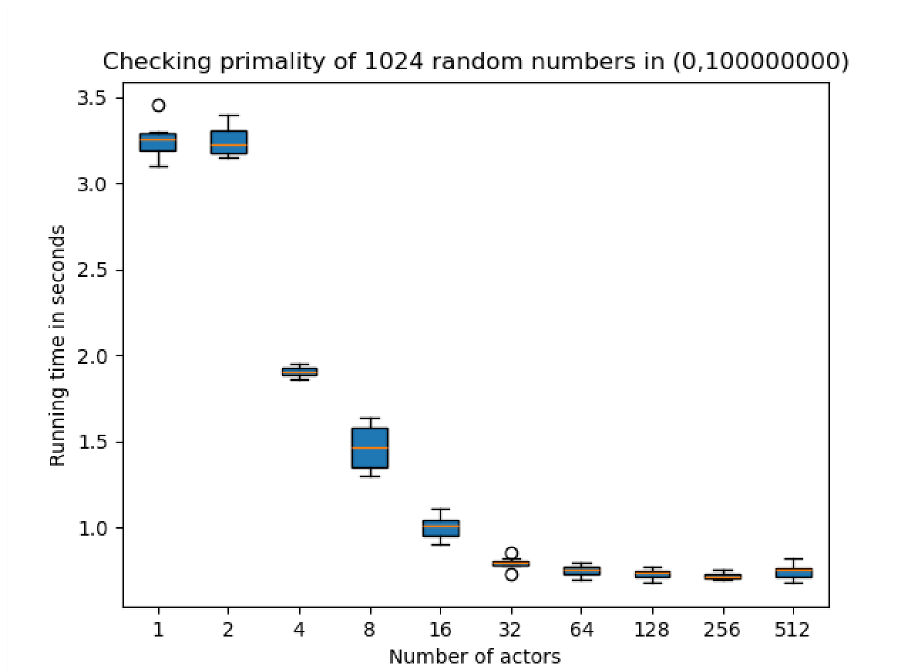


Do more actors improve performance?

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

microseconds—not something doable on major operating systems these days."
[Learn You Some Erlang for great good! Chapter: The Hitchhiker's Guide to Concurrency]

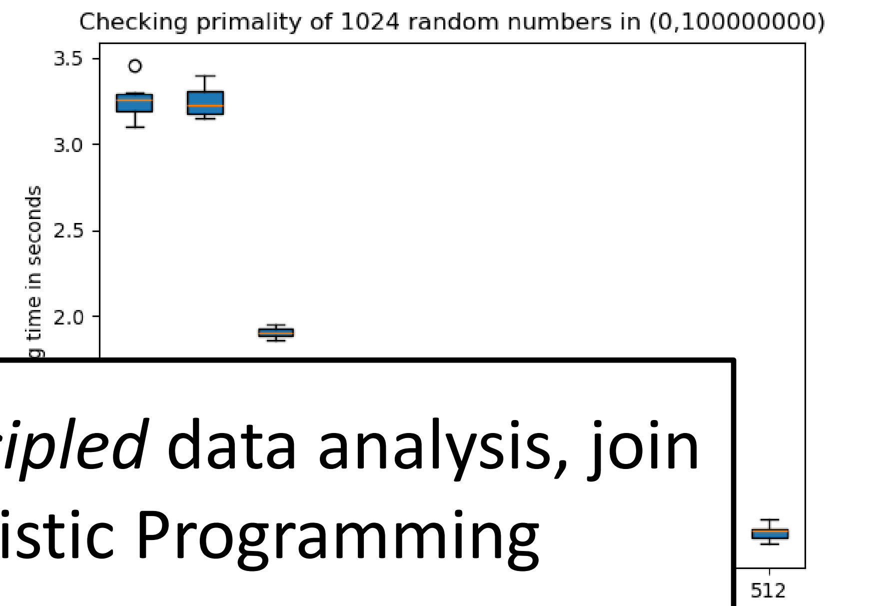
- However, actor systems can be distributed among many computers (Erlang shell nodes in the case of Erlang) and computers
  - We are not limited to a single computer throughput
  - See distributed Erlang in Learn You Some Erlang for Great Good



Do more actors improve performance?

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

microseconds—not something doable on major operating systems these days.”

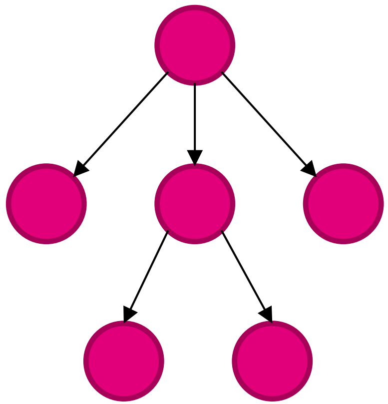


If you are interested in *principled* data analysis, join our course on Probabilistic Programming

Probabilistic Programming (Spring 2025)

Topology (Actor hierarchy)

- We use the term *topology* to refer to the structure of the actor system in terms of number of actors and communication
- The systems we have seen so far feature a *static* topology
  - All the actors in the system are spawned during initialization



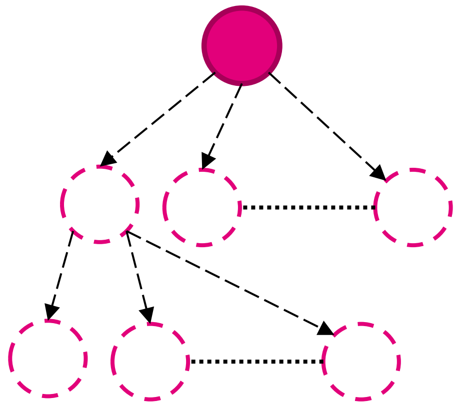
Solid lines and actors represent elements that are created during initialization and never change

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 actor 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)

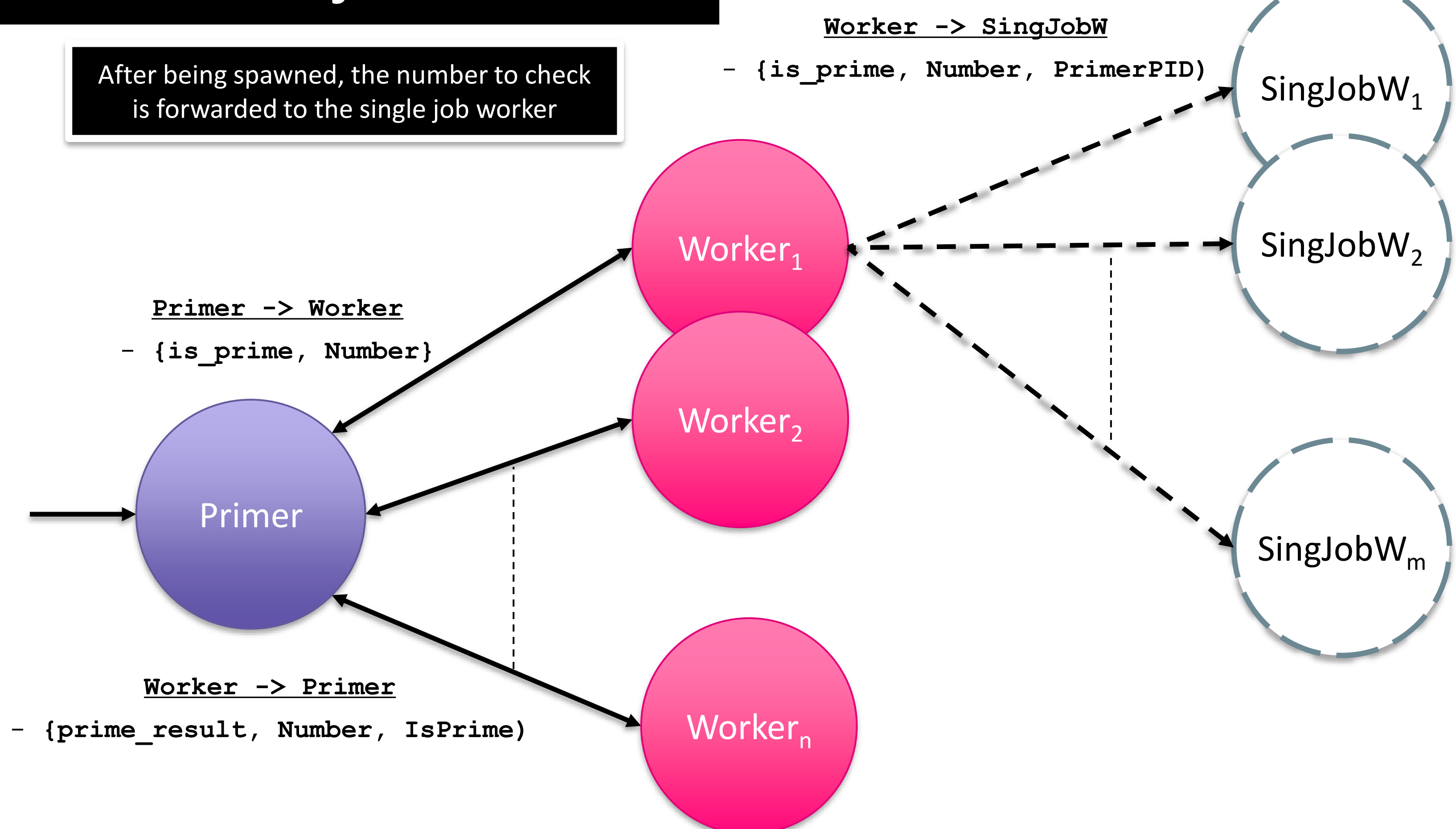
Primer with job workers

- 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

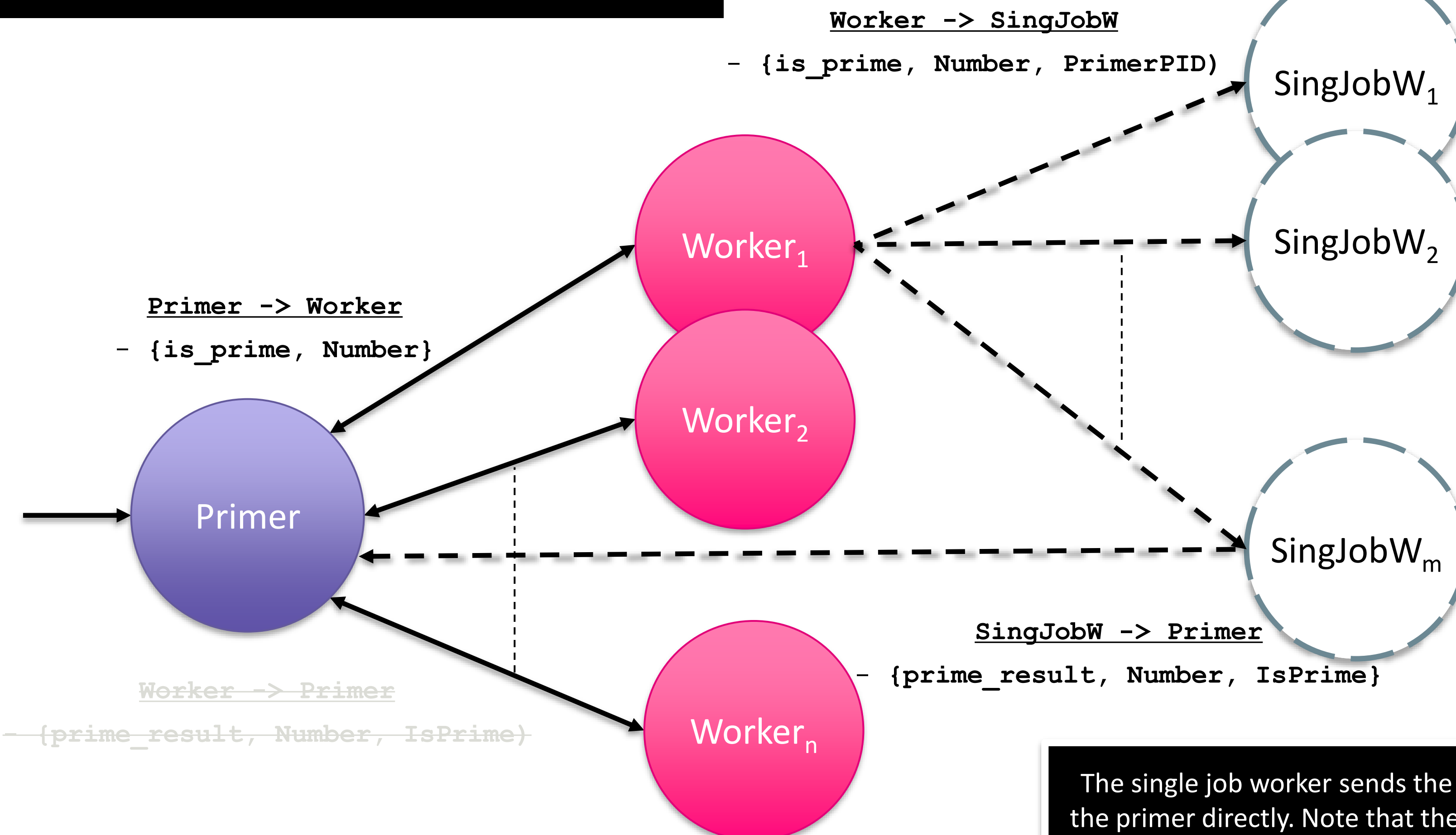
- After these dedicated actors have finished the computation, they report the result and terminate the execution

Primer with job workers

After being spawned, the number to check is forwarded to the single job worker

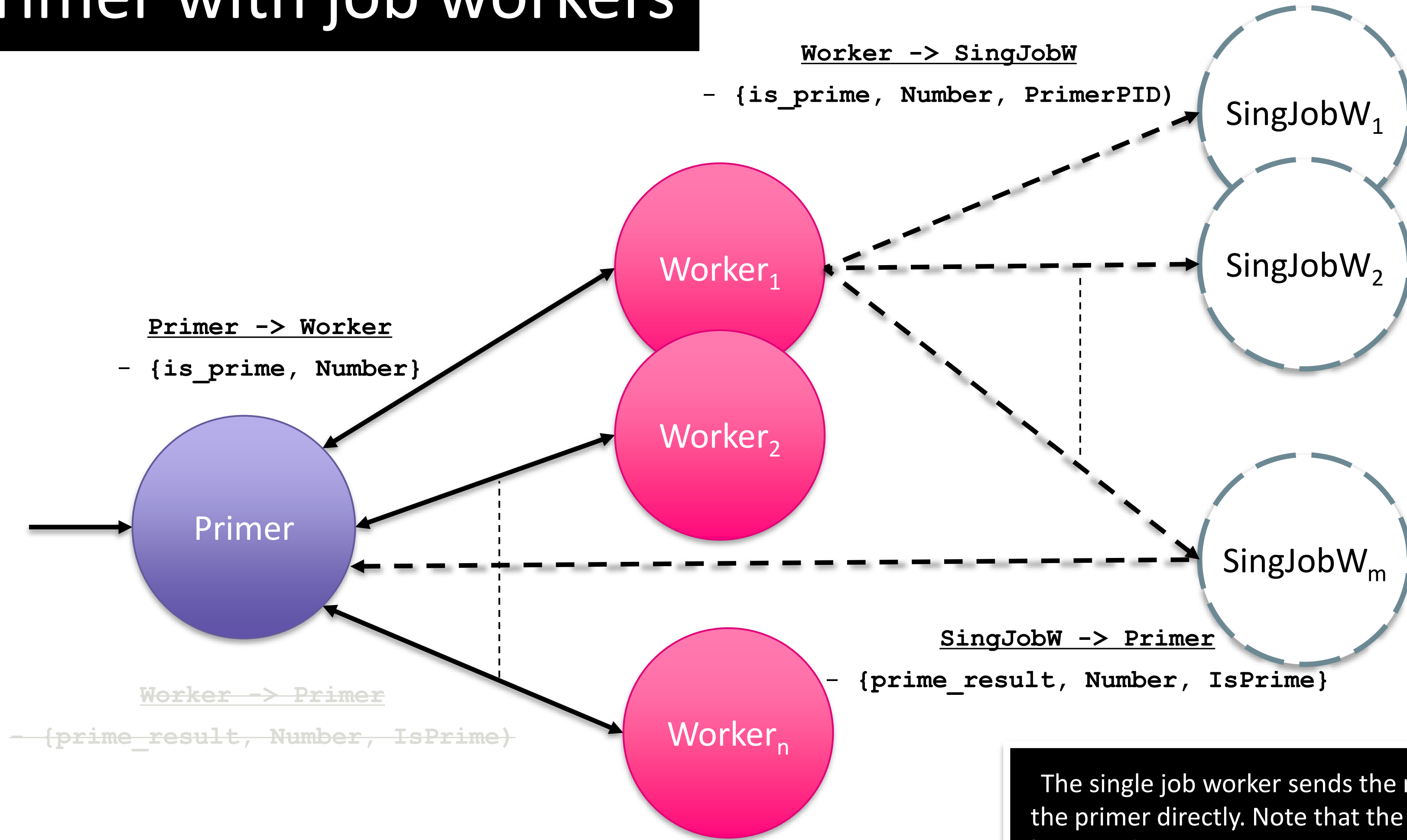


Primer with job workers



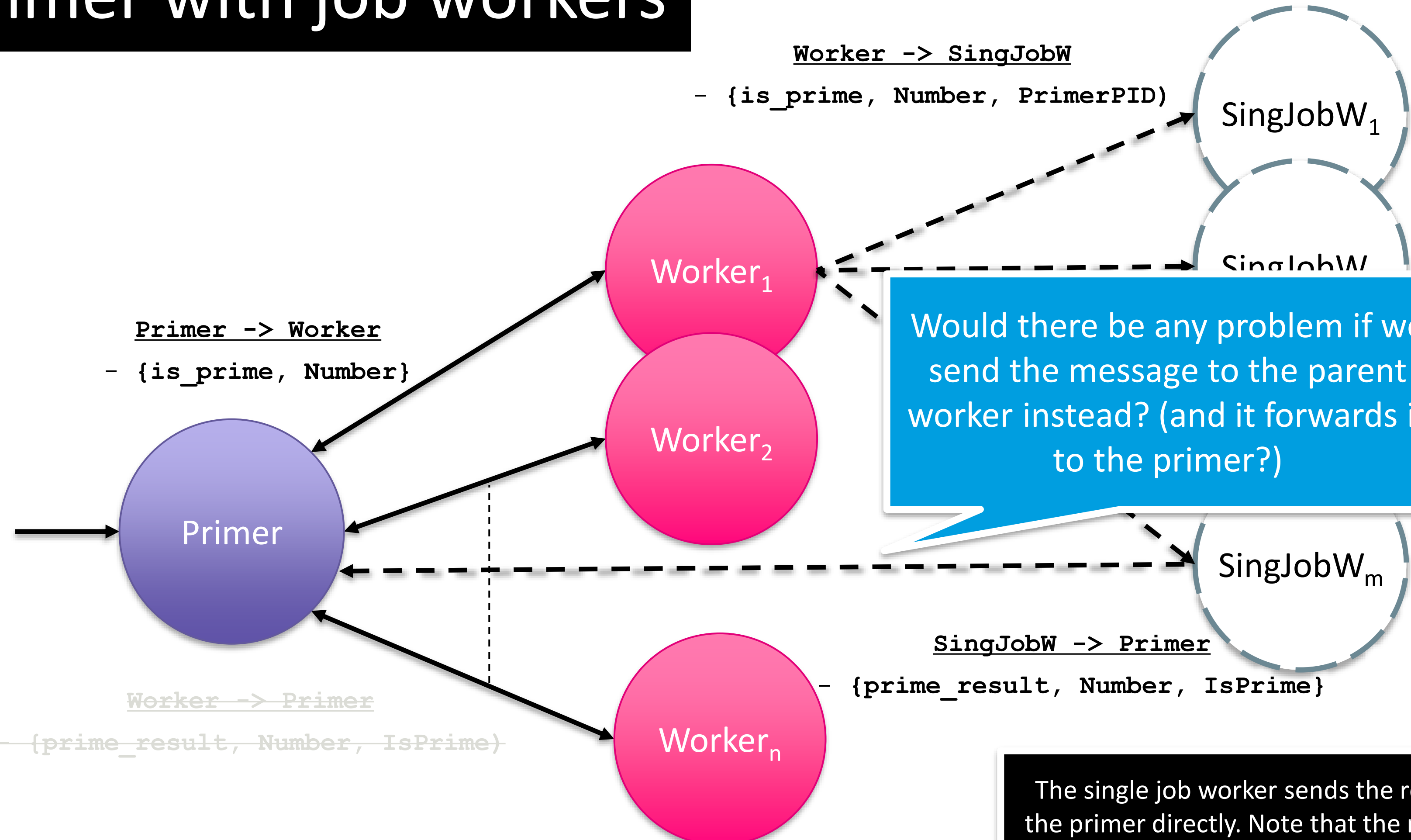
The single job worker sends the result to the primer directly. Note that the message from worker to primer is unnecessary now.

Primer with job workers



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

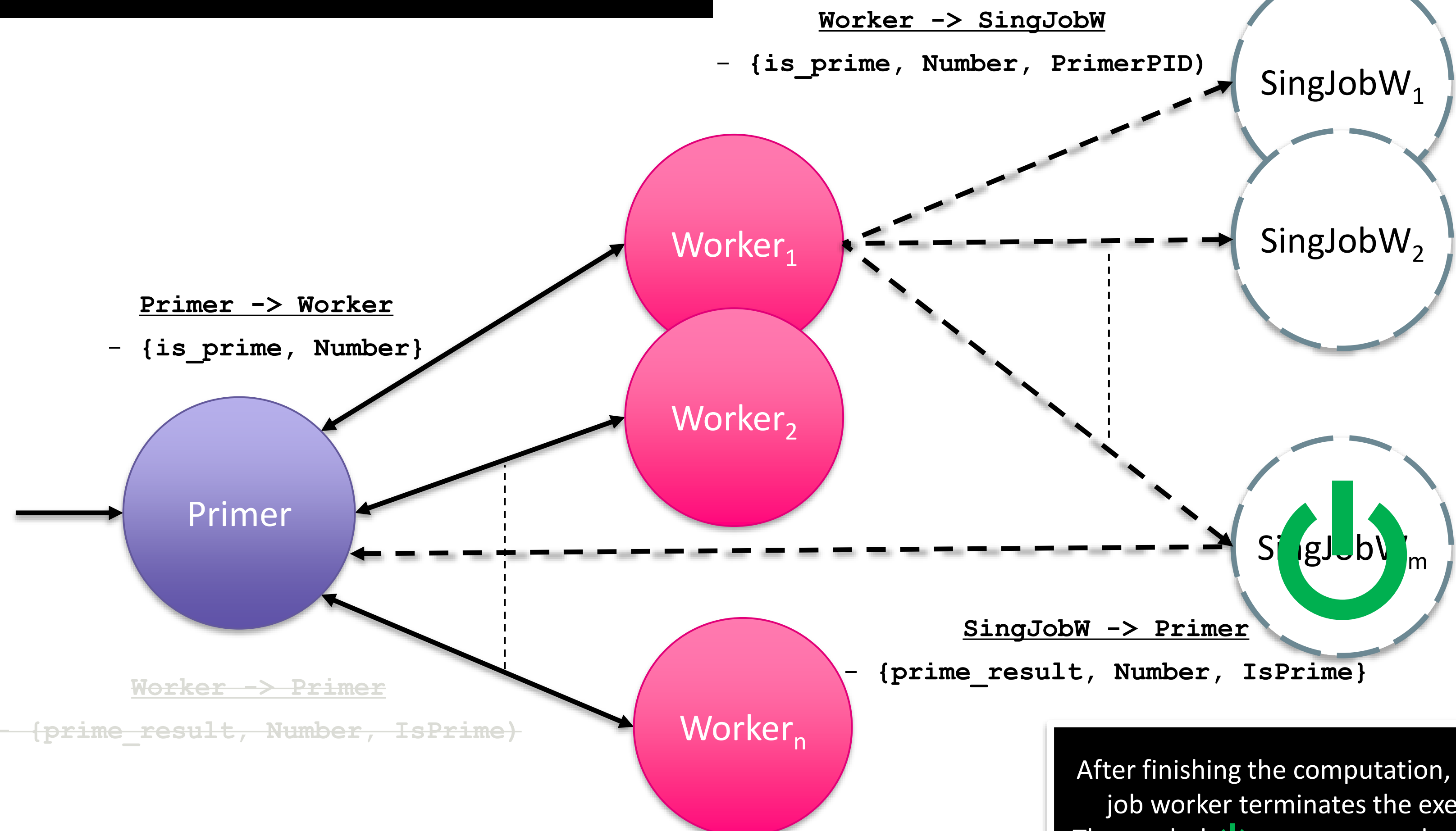
Primer with job workers



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

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



After finishing the computation, the single job worker terminates the execution. The symbol ⏻ means normal termination

How to (normally) shutdown actors in Erlang

- Not implementing a loop function, make sure that the actor simply executes finite number of instructions

single_job_worker.erl

```
no_loop() ->
    receive
        {check_prime, Number, PrimerPID} ->
            PrimerPID ! {prime_result, Number, is_prime_naive(Number)}
    end.
```

- Breaking the loop function, i.e., in the case when you want to terminate normally do not make a recursive call to loop

How to (normally) shutdown actors in Erlang

- Not implementing a loop function, make sure that the actor simply executes finite number of instructions

single_job_worker.erl

```
no_loop() ->
    receive
        {check_prime, Number, PrimerPID} ->
            PrimerPID ! {prime_result, Number, is_prime_naive(Number)}
    end.
```

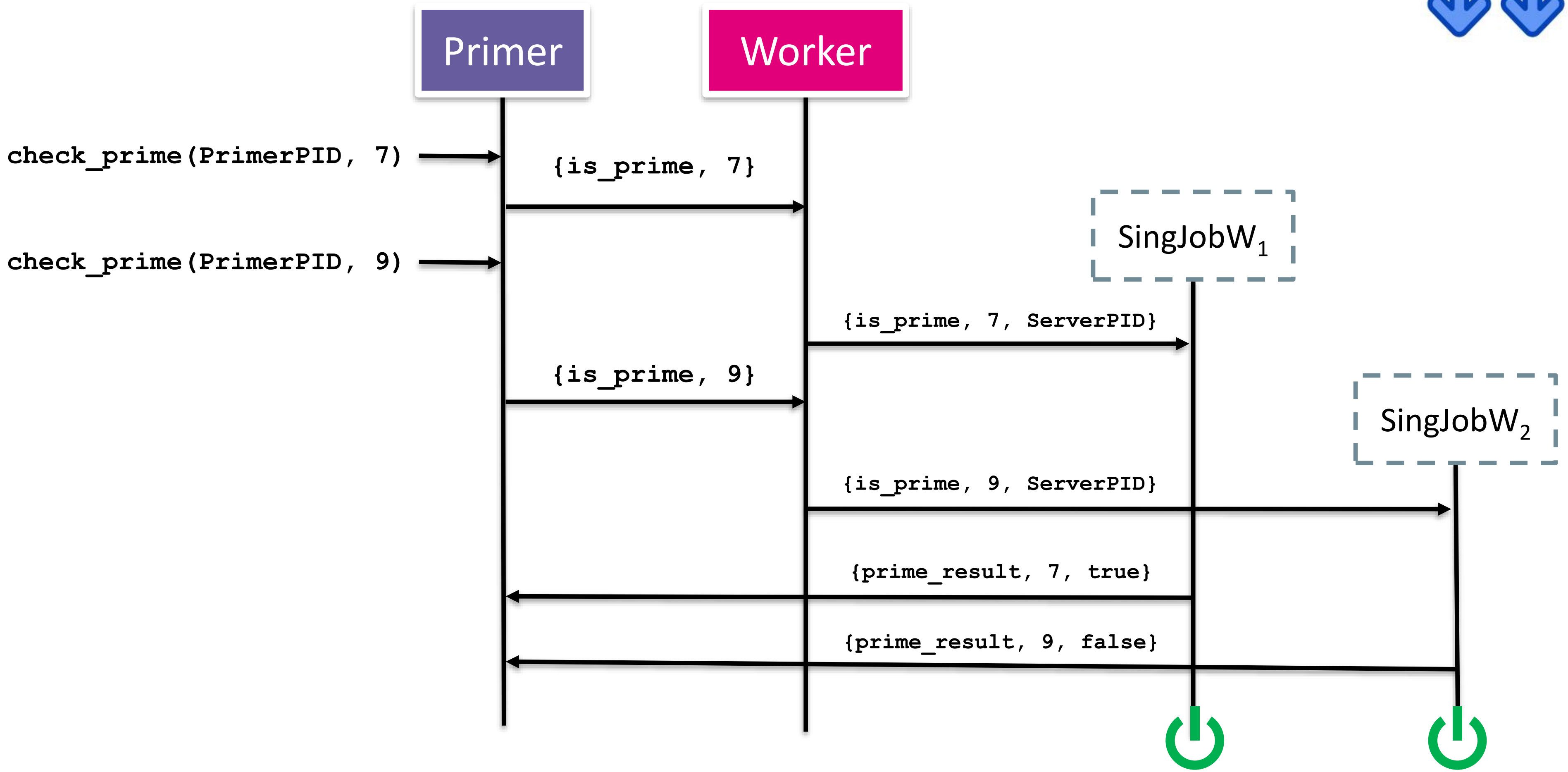
- Breaking the loop function, i.e., in the case when you want to terminate normally do not make a recursive call to loop

- Explicitly calling for a normal exit **`exit(normal)`**

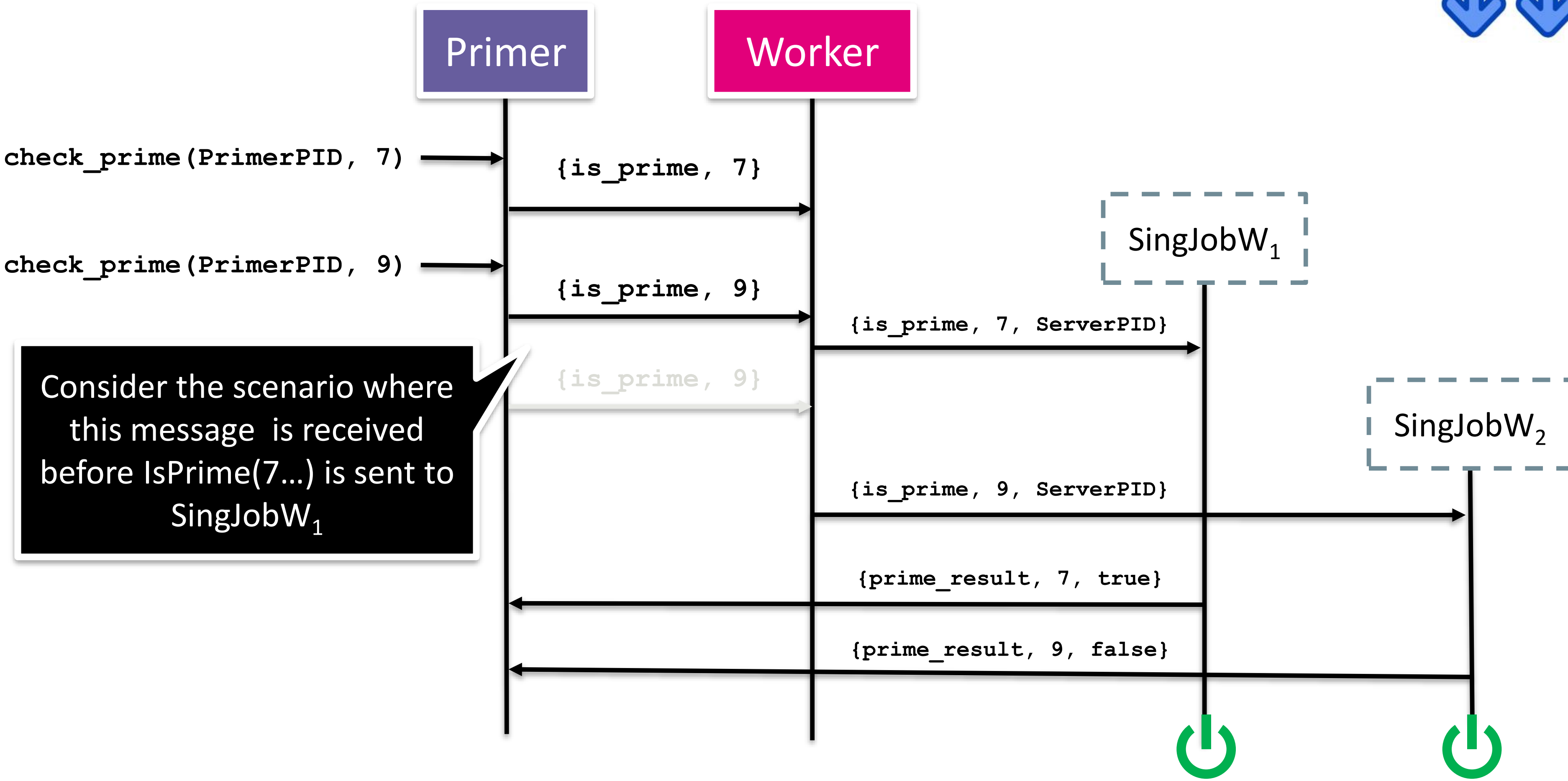
worker.erl (exercises)

```
loop(State) ->
    receive
        {compute, SenderPID, Task} ->
            handle_compute(SenderPID, Task, State);
        stop ->
            exit(normal)
    end.
```

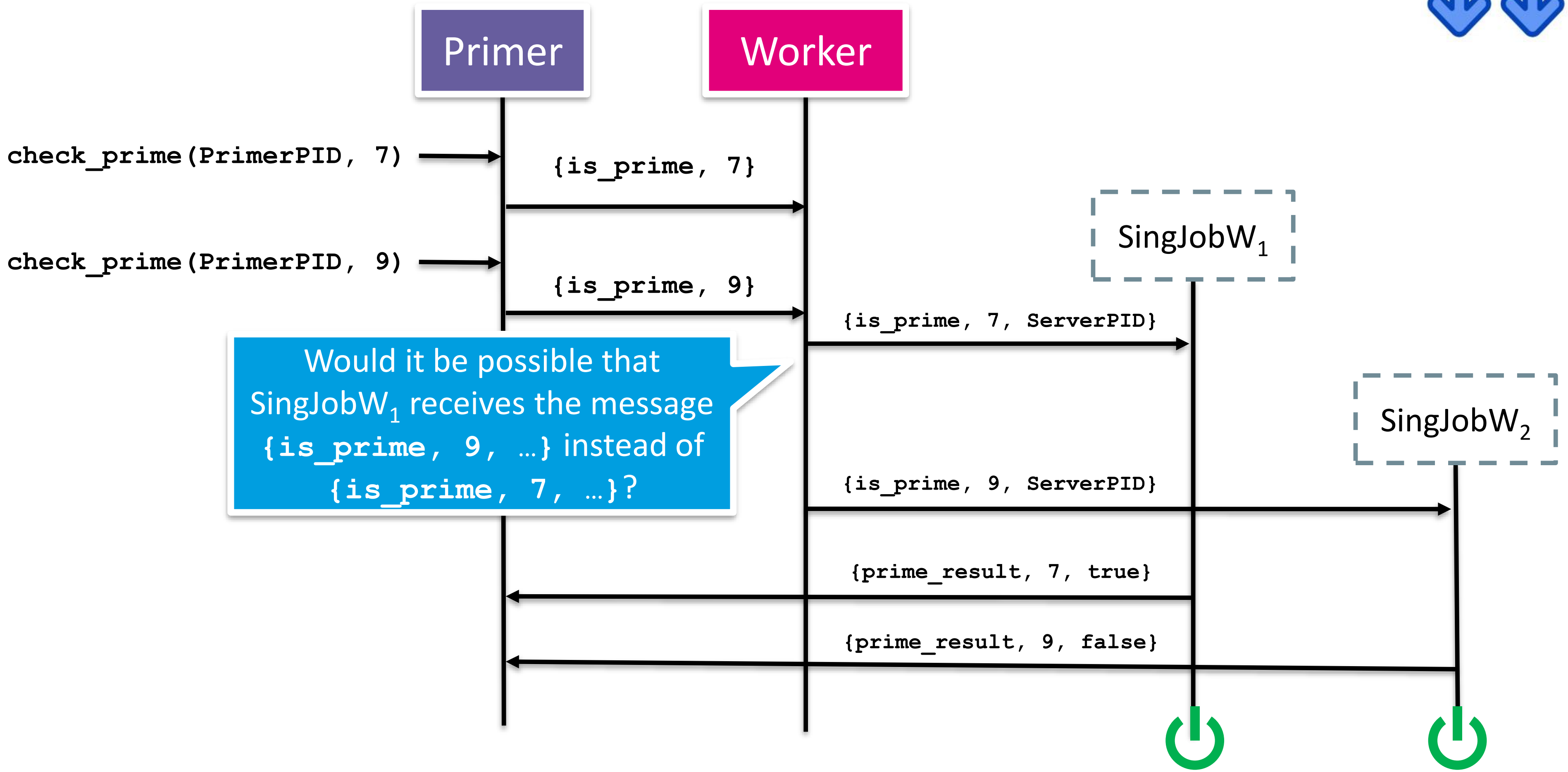
Primer with job workers – execution example



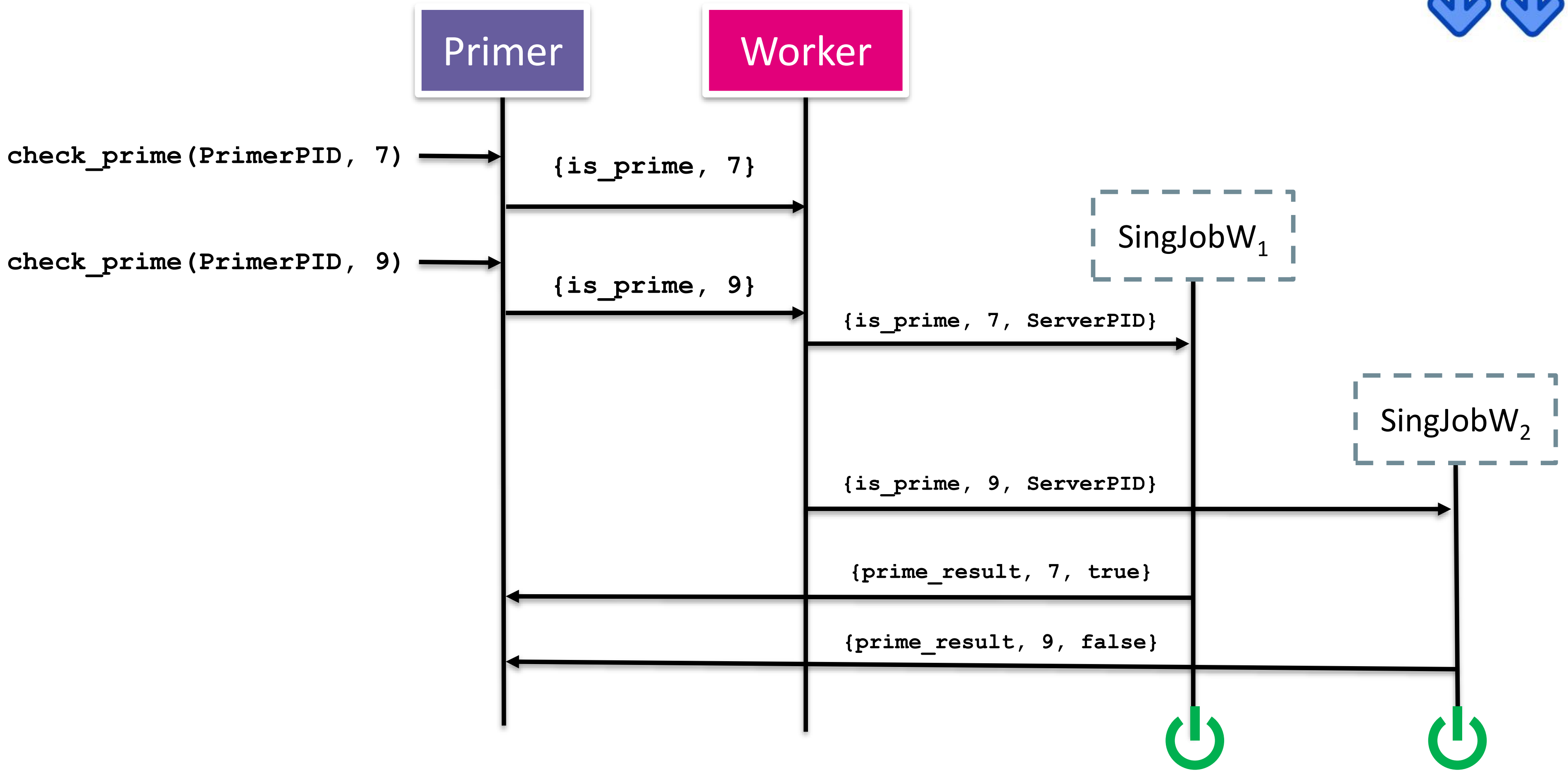
Primer with job workers – execution example



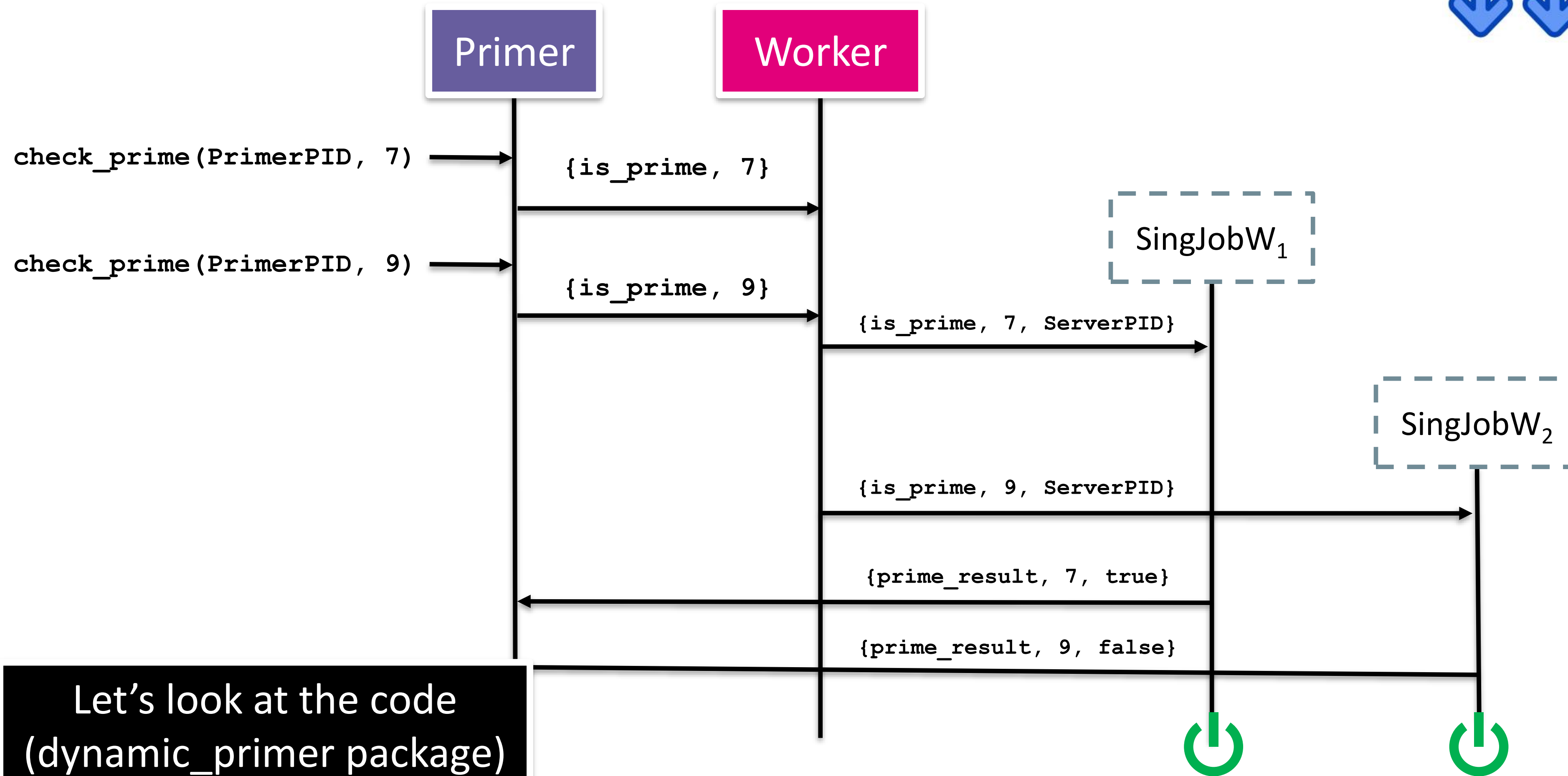
Primer with job workers – execution example



Primer with job workers – execution example



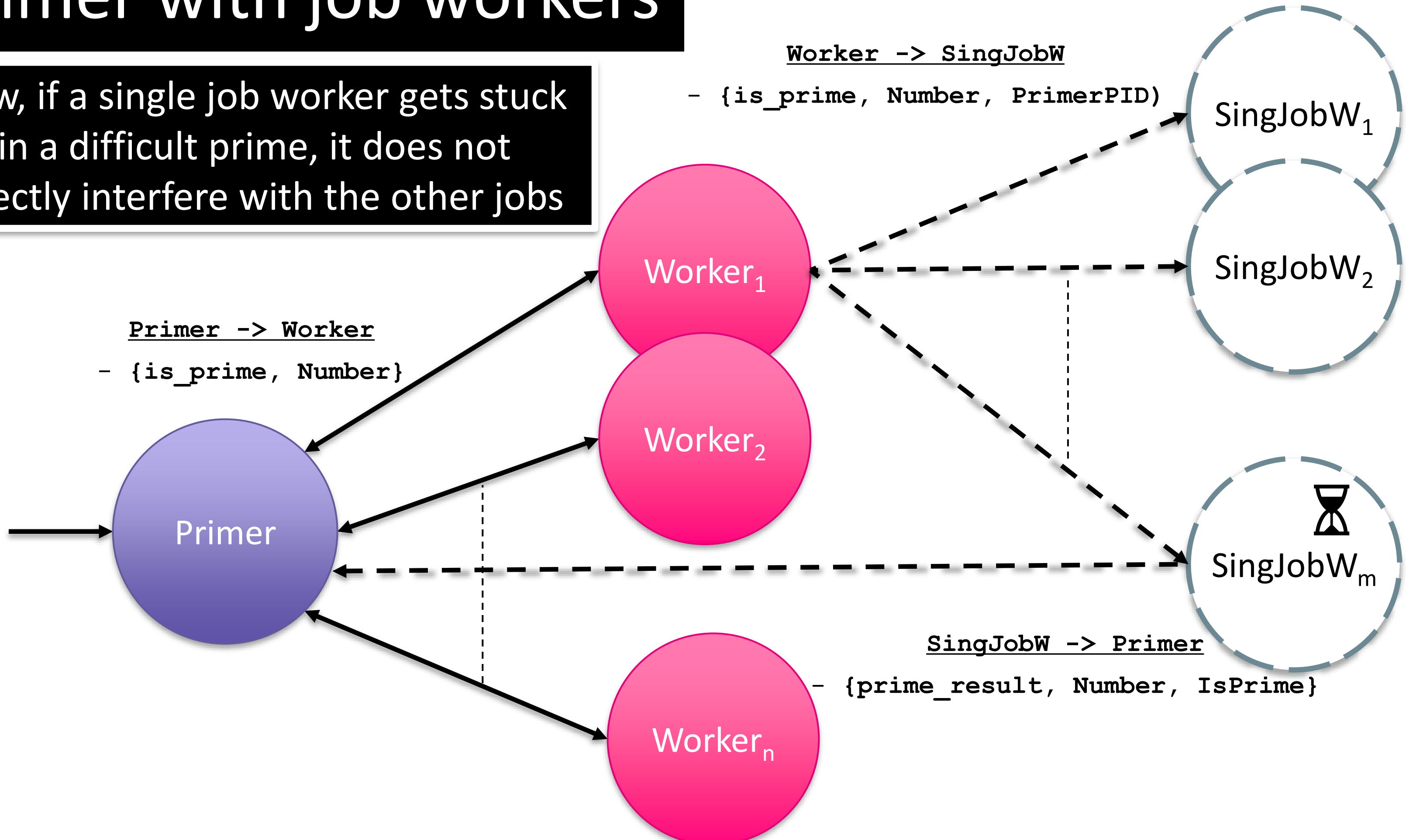
Primer with job workers – execution example



Let's look at the code
(dynamic_primer package)

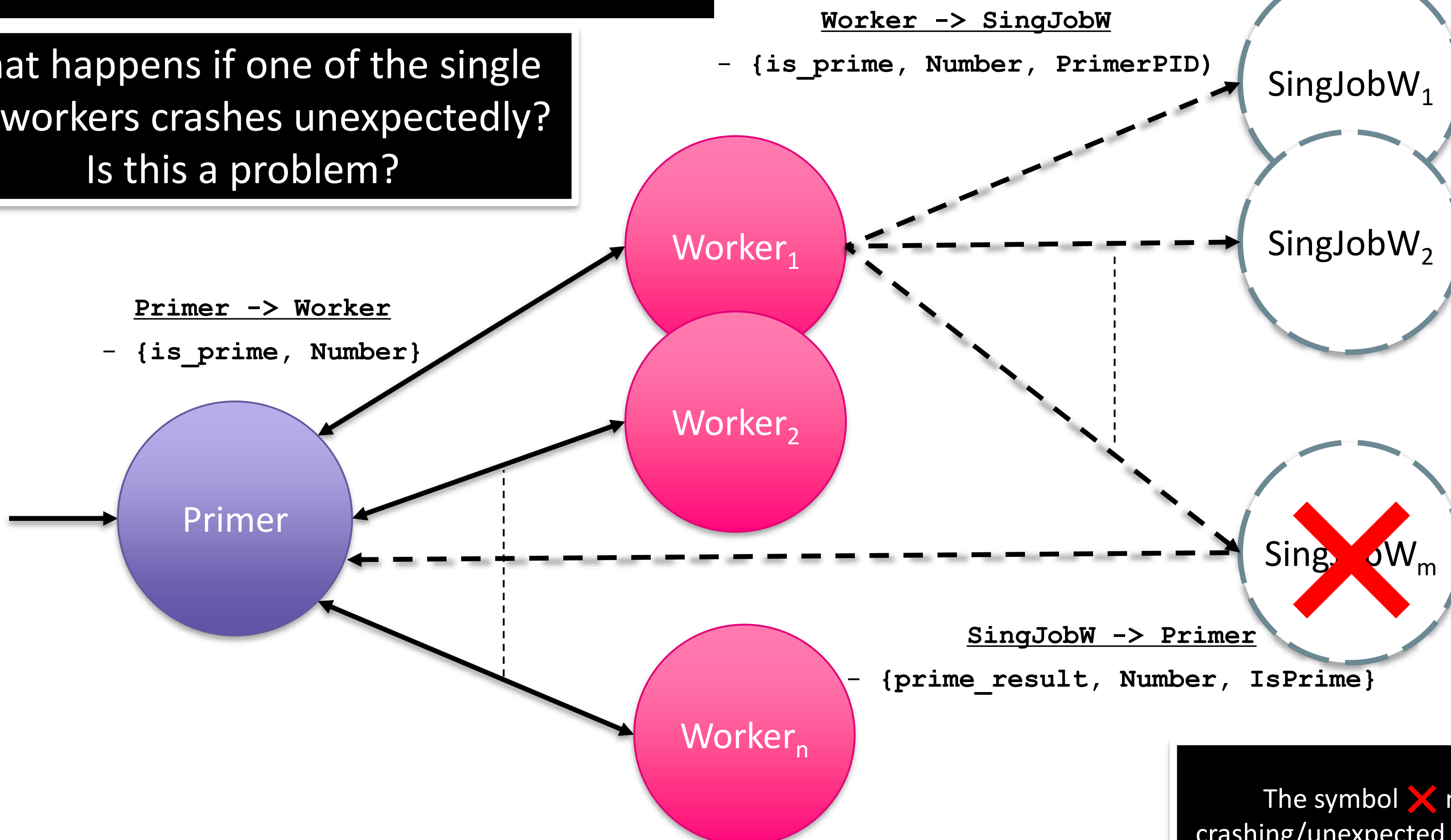
Primer with job workers

Now, if a single job worker gets stuck in a difficult prime, it does not directly interfere with the other jobs



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 Erlang

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 actor systems ensuring that if an actor crashes the system can recover

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

Fault-tolerance in Erlang

There is also a **`spawn_monitor`** function that spawns a process and monitors it atomically.

- Erlang implements multiple mechanisms for fault-tolerance including process links and monitors
- **In this course, we focus on process <u>monitoring</u>**
- An Erlang process may monitor another process by invoking the function <u>**`monitor(PID)`**</u>
  - PID is the PID of the process to monitor
  - This function returns a reference **`Ref`** that can be used when receiving signals (see below)
- A *signal* can be seen as message that is automatically when processes are monitored
  - In fact, signals are more general than messages. A message is a specific type of signal. You can see a detailed description of Erlang signal in the optional readings [Erlang documentation. Chapter 14. Section 14.6 Signals]
- If an actor monitors another actor, then it can receive the following signals
  - **`{'DOWN', Ref, process, Pid, normal}`**, if the monitored actor terminated normally
  - **`{'DOWN', Ref, process, Pid, Reason}`**, if the monitored actor crashed due to an exception

The reference returned by the call to **`monitor`**

The PID of the process from which the signal comes

The reason for sending the signal. If it terminates normally, it is the atom normal. Otherwise, it is an Erlang term with the cause for sending the signal

Handling signals in Erlang

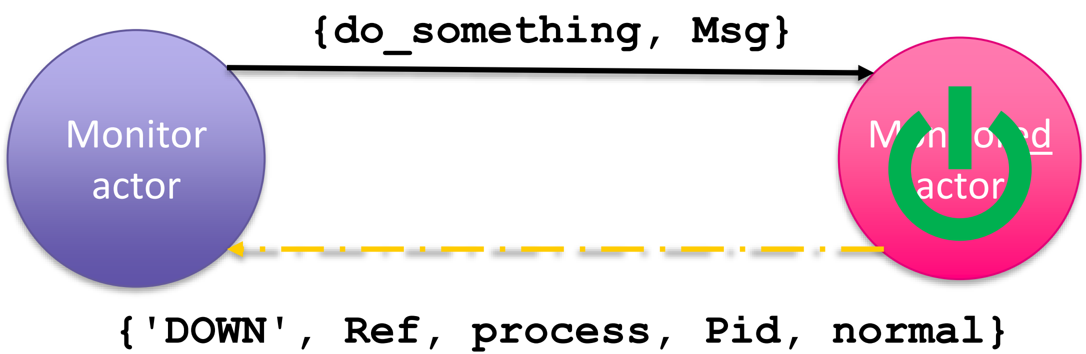
- Simply add a case in the loop function for the signal messages mentioned in the previous slide

```
loop(State) ->
    receive
        {msg1, Msg} ->
            handle_mgs1(Msg, State);
        {msg2, Msg} ->
            handle_mgs1(Msg, State);
        …
        {'DOWN', _Ref, process, PID, normal} ->
            handle_normal_exit(PID, State);
        {'DOWN', _Ref, process, PID, Reason} ->
            handle_exit(PID, Reason, State)
    end.
```

Recall: When processing a message/signal, **`receive`** picks the message that first pattern matches the tuple structure. Due to this, the case for **`normal`** signals is before the more general case for any **`Reason`**.

Actor supervision (graphically)

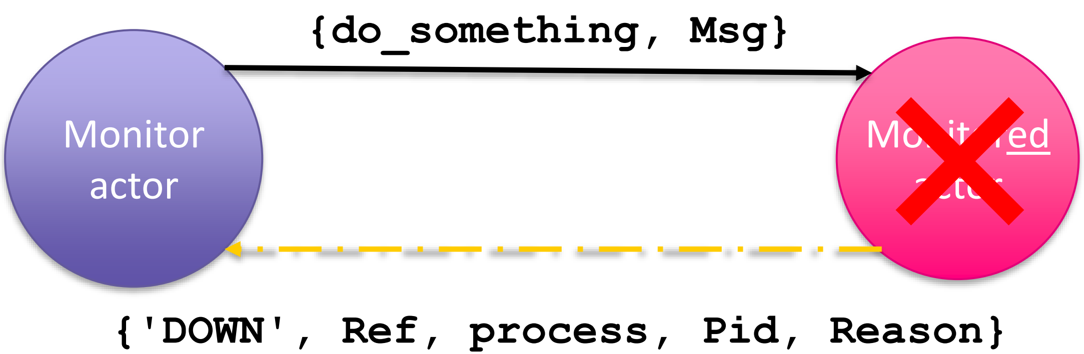
Actor supervision (graphically)



The dashed-dotted yellow arrow indicates the sending of a signal. These are sent automatically by Erlang as part of the supervision functionality.

If the process finishes normally, the mechanism sends the tuple **`{'DOWN', Ref, process, Pid, normal}`**

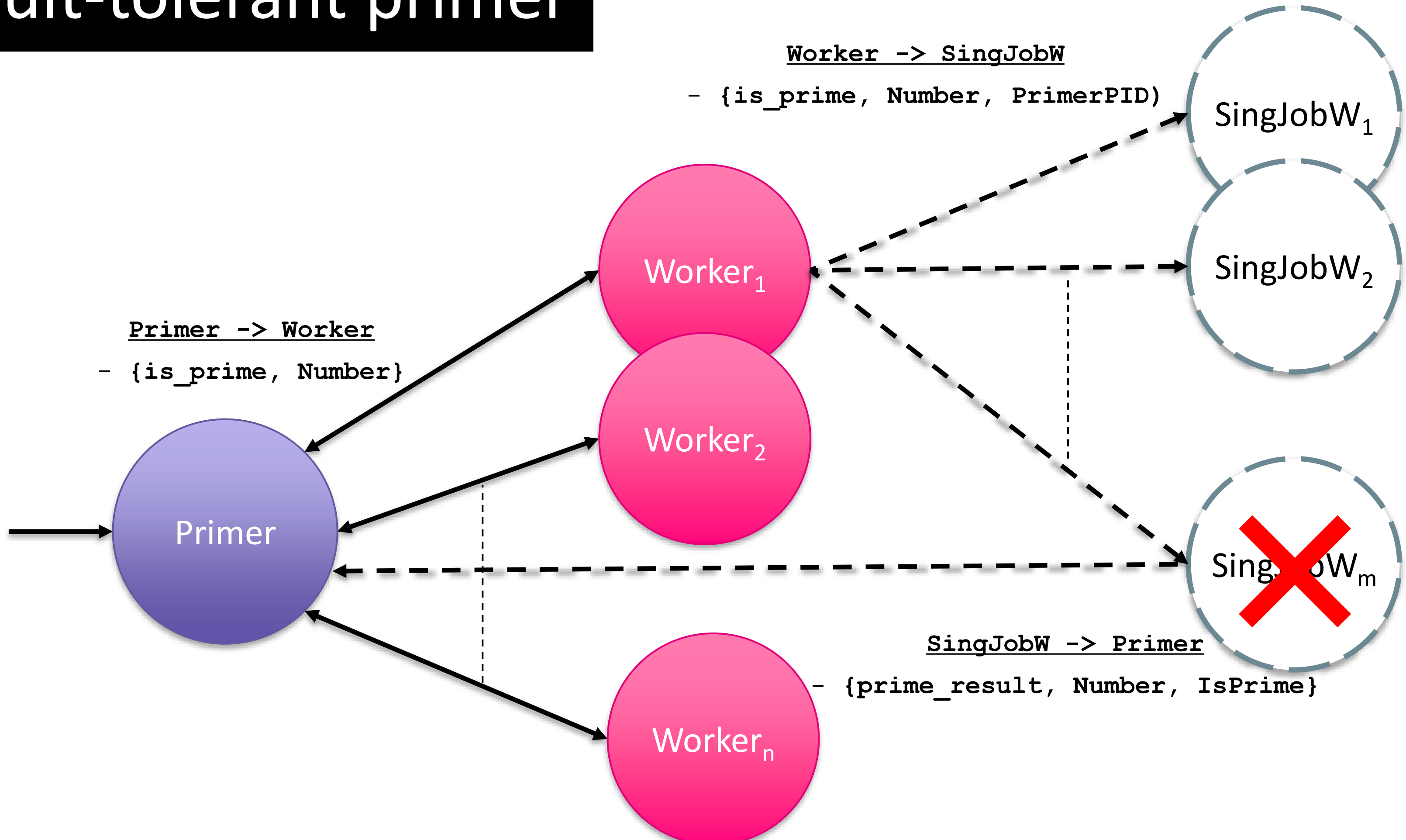
Actor supervision (graphically)



If the process crashes due to exceptions, the mechanism sends the tuple **`{'DOWN', Ref, process, Pid, Reason}`**

`Reason` is an Erlang term containing the information about the cause of the crash

Fault-tolerant primer

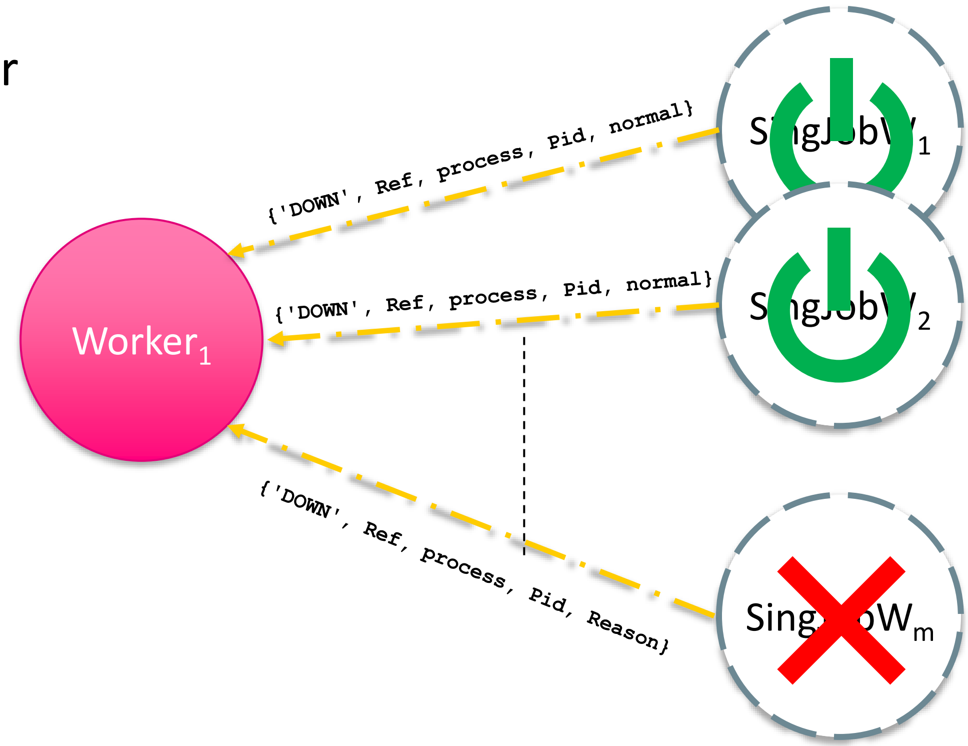


Fault-tolerant primer

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

- To this end, the worker needs to:

1. Monitor all the actors it spawns
2. Handle crash/exception error signals
   - The handler spawns a new worker and sends the number again to check whether it is prime
3. Handle normal termination signals
   - No more computation needed; we can mark the number as checked

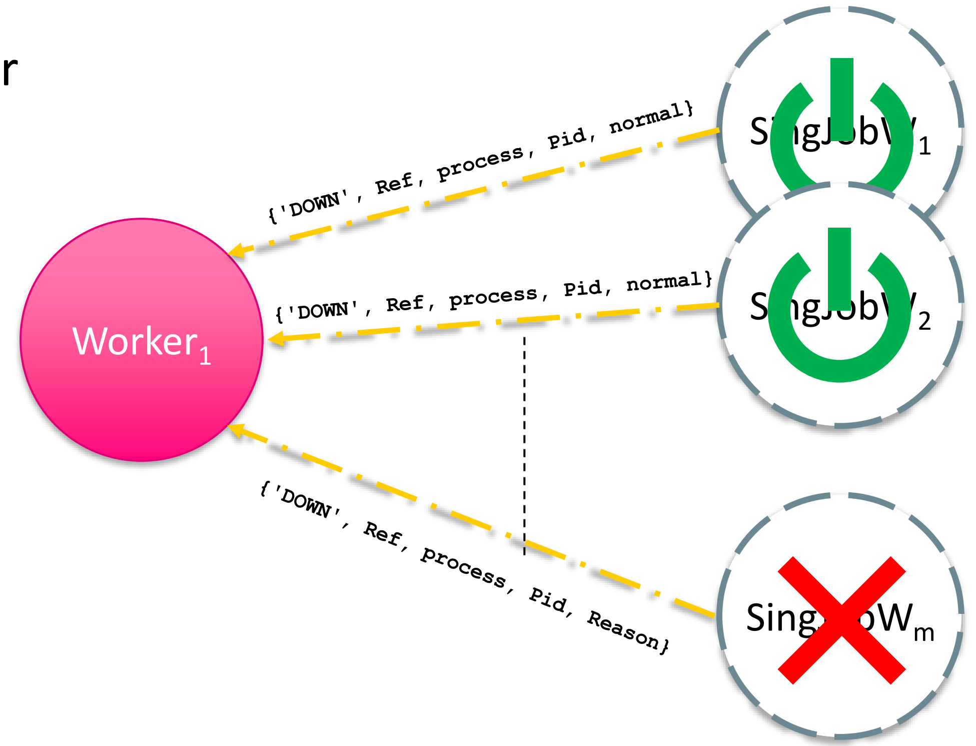


Fault-tolerant primer

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

- To this end, the worker needs to:

1. Monitor all the actors it spawns
2. Handle crash/exception error signals
    - The handler spawns a new worker and sends the number again to check whether it is prime
3. Handle normal termination signals
    - No more computation needed; we can mark the number as checked



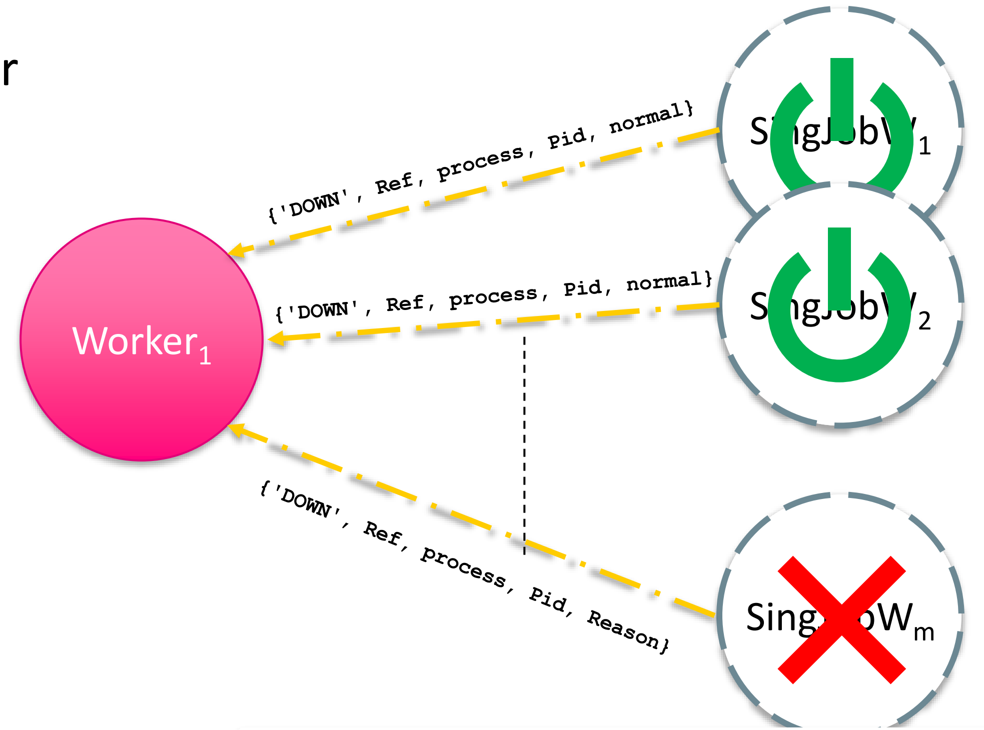
We must extend the actor's state to keep track of what number is computed by what actor

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. Monitor all the actors it spawns
2. Handle crash/exception error signals
   - The handler spawns a new worker and sends the number again to check whether it is prime
3. Handle normal termination 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

Let's look at the code (fault_tolerant_primer 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

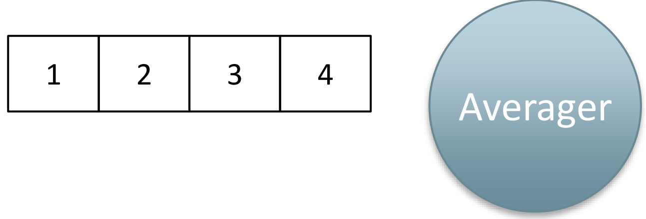
Scatter-Gather systems

- 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 or gatherers and combines them into a single piece of data. Then, it sends the result to a higher level gatherer

Average and scatter-gather

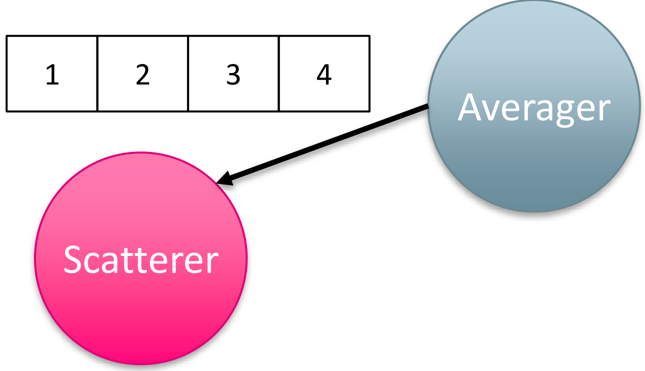
- A problem for which this pattern is suitable is computing the average of a list of numbers
- Given a set of natural numbers $a_1, a_2, \ldots, a_n$, the average is $\frac{1}{n}\sum_i a_i$
    - Note that this is equivalent to $\sum_i \frac{a_i}{n}$
- In a nutshell, we can have scatterer actors splitting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

Averager



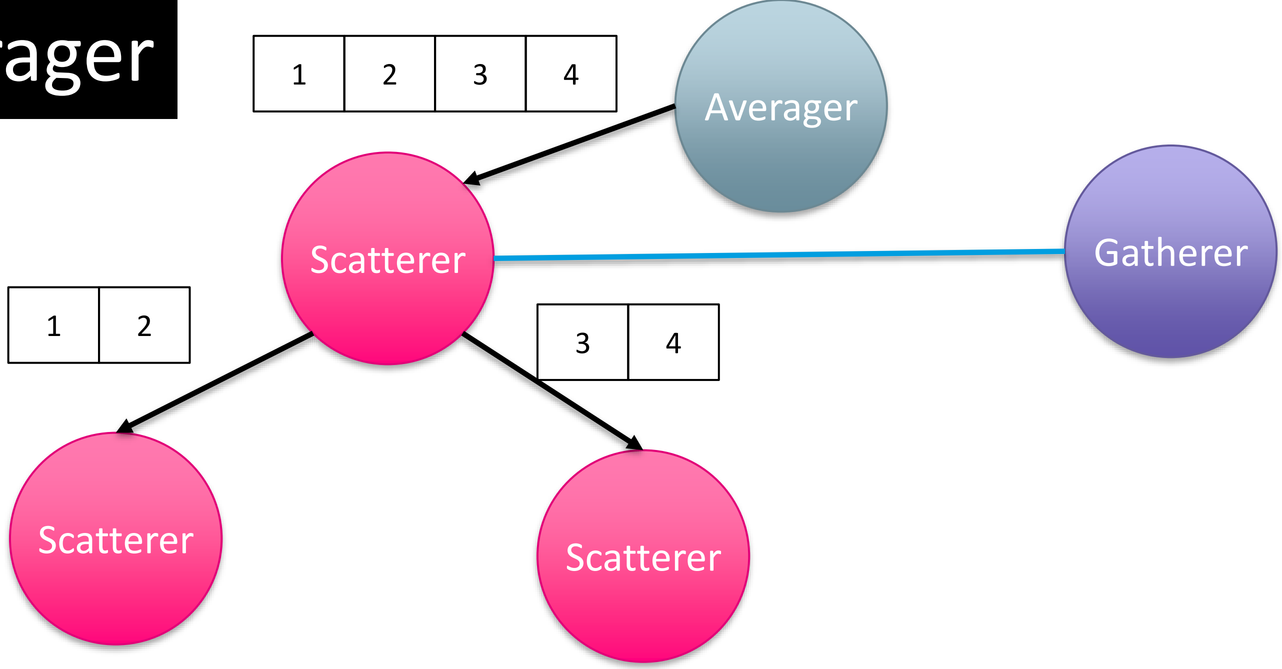
- Consider a system that computes the average of a list of numbers
- The averager above is a process providing an API to the system

Averager



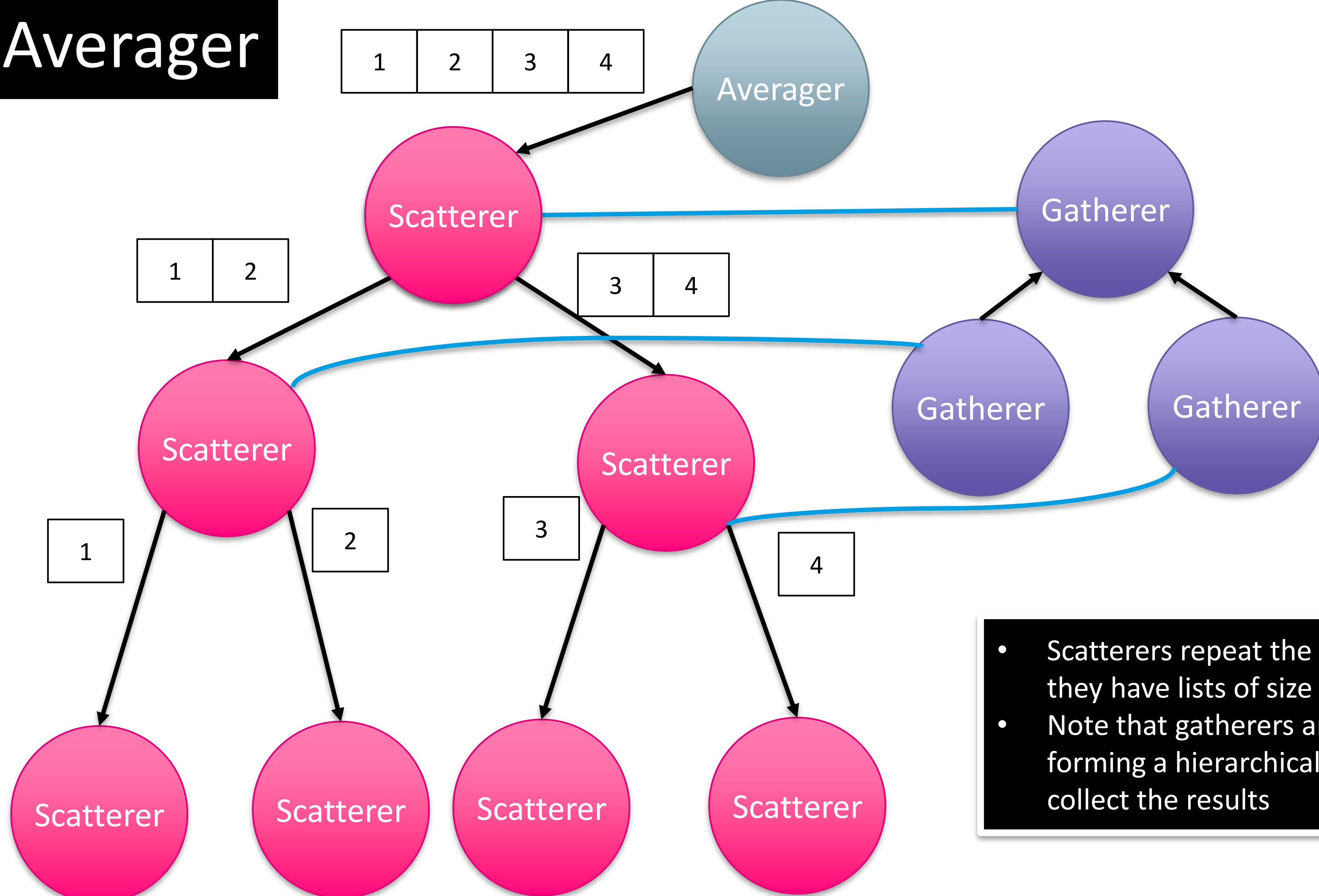
- The averager spawns a scatterer and sends the input list

Averager



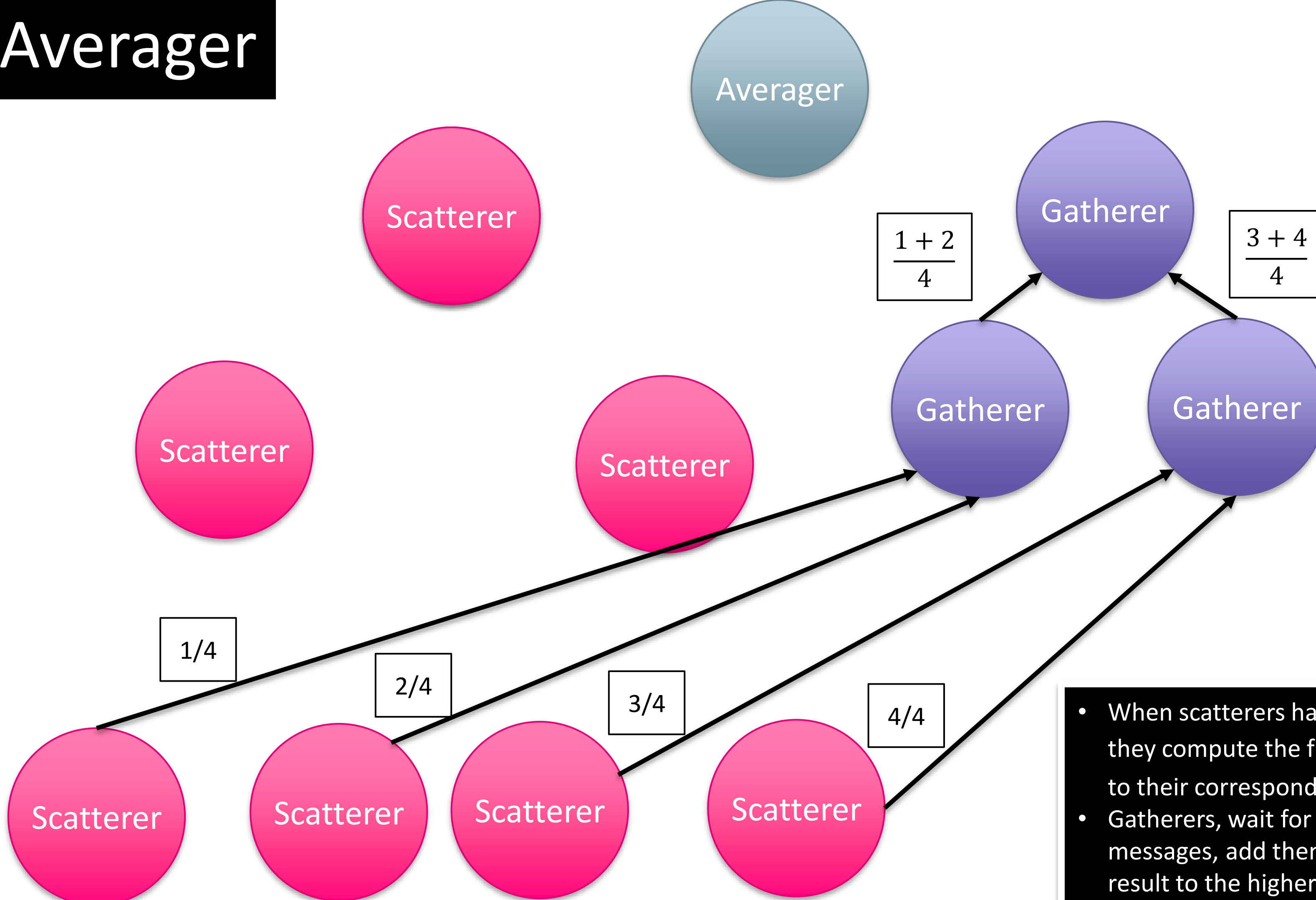
- In the first step, the scatterer splits the computation into two sublists, and assigns them to new scatterer workers
- Also, it spawns a gatherer worker that will receive and merge the average of each sublist (we use the blue line to indicate the actor that spawned the gatherer)

Averager



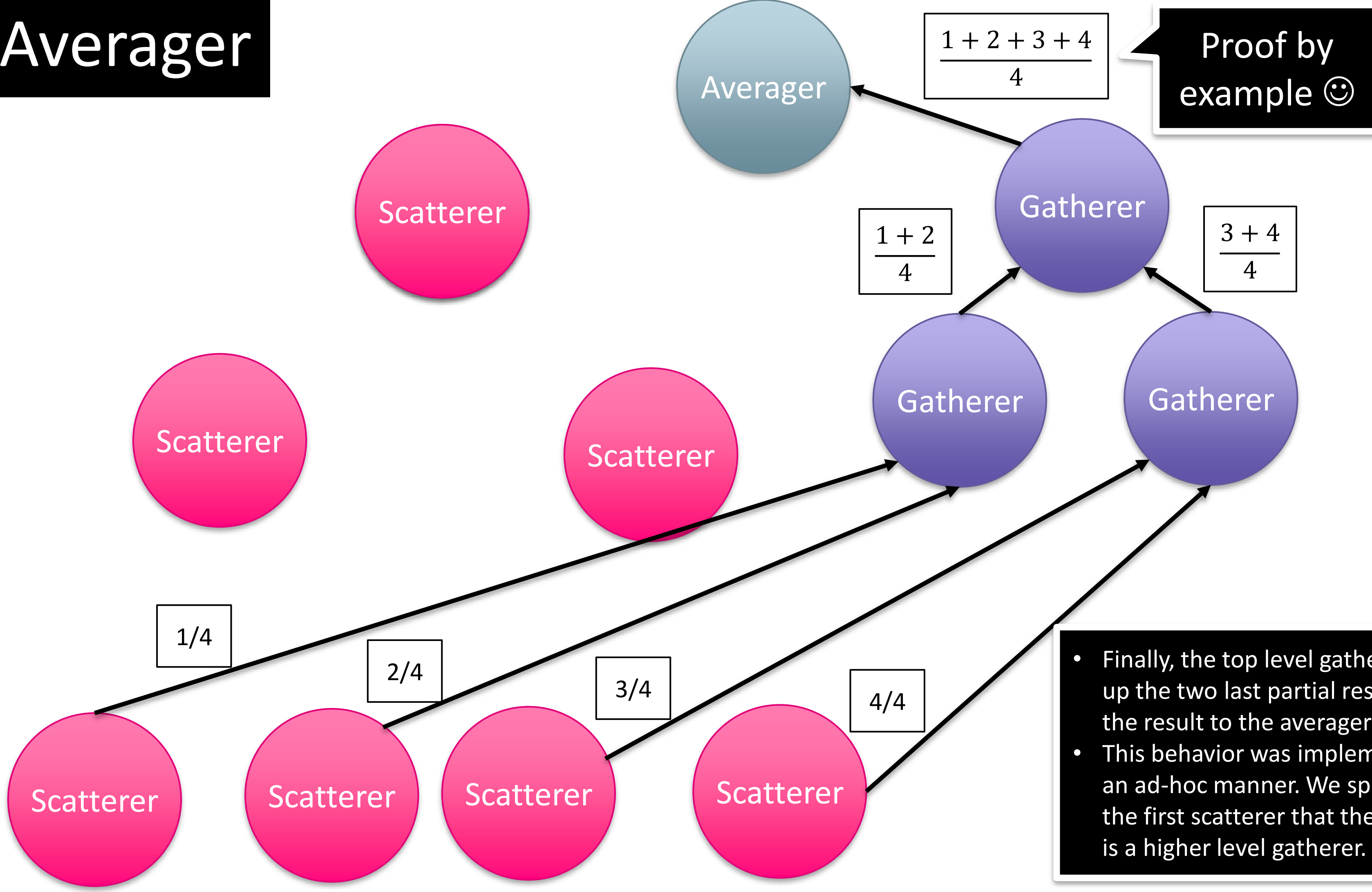
- Scatterers repeat the process until they have lists of size one
- Note that gatherers are also forming a hierarchical structure to collect the results

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

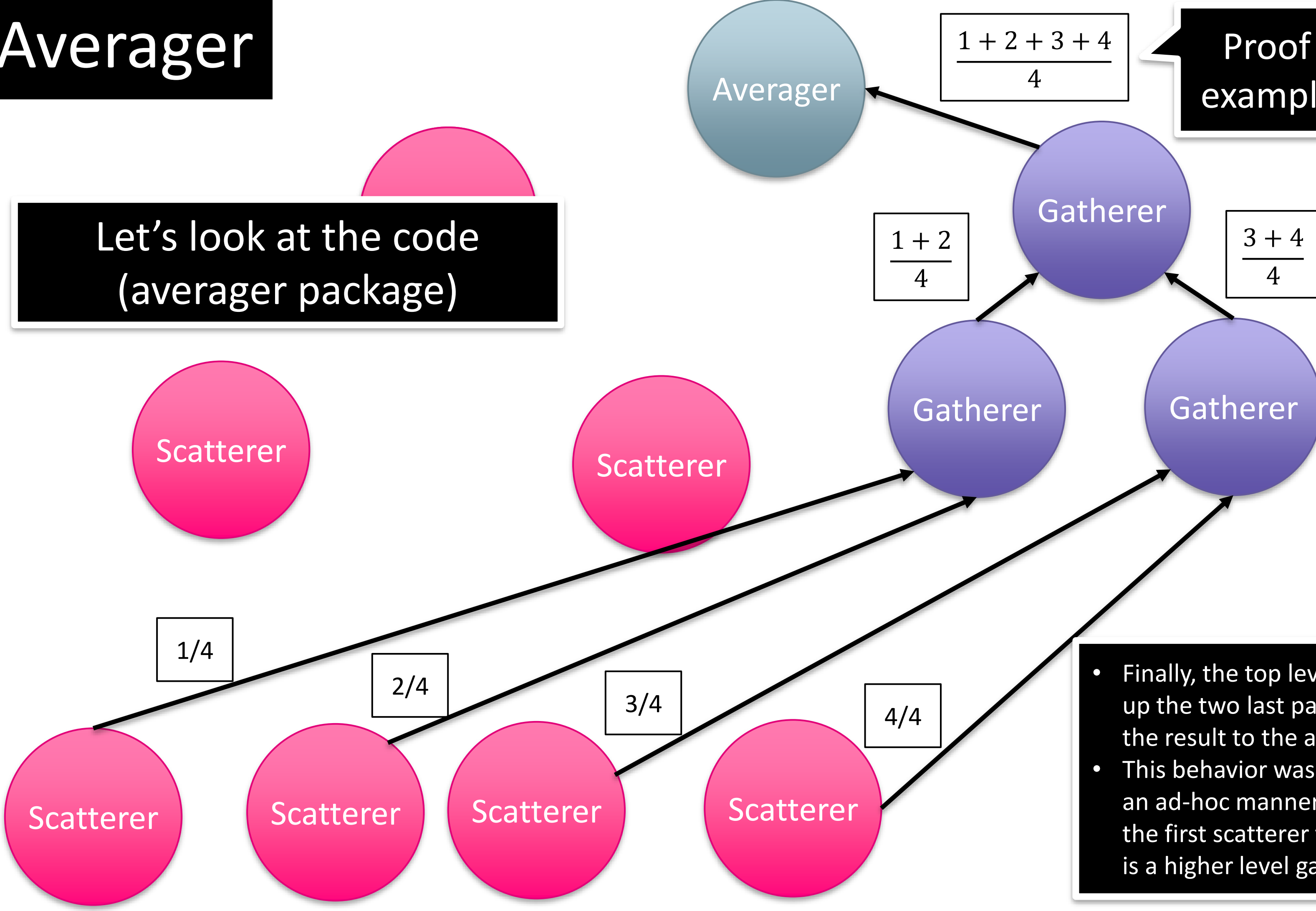


- Finally, the top level gatherer adds up the two last partial results sends the result to the averager process
- This behavior was implemented in an ad-hoc manner. We specified for the first scatterer that the averager is a higher level gatherer.

Averager

Proof by example ☺

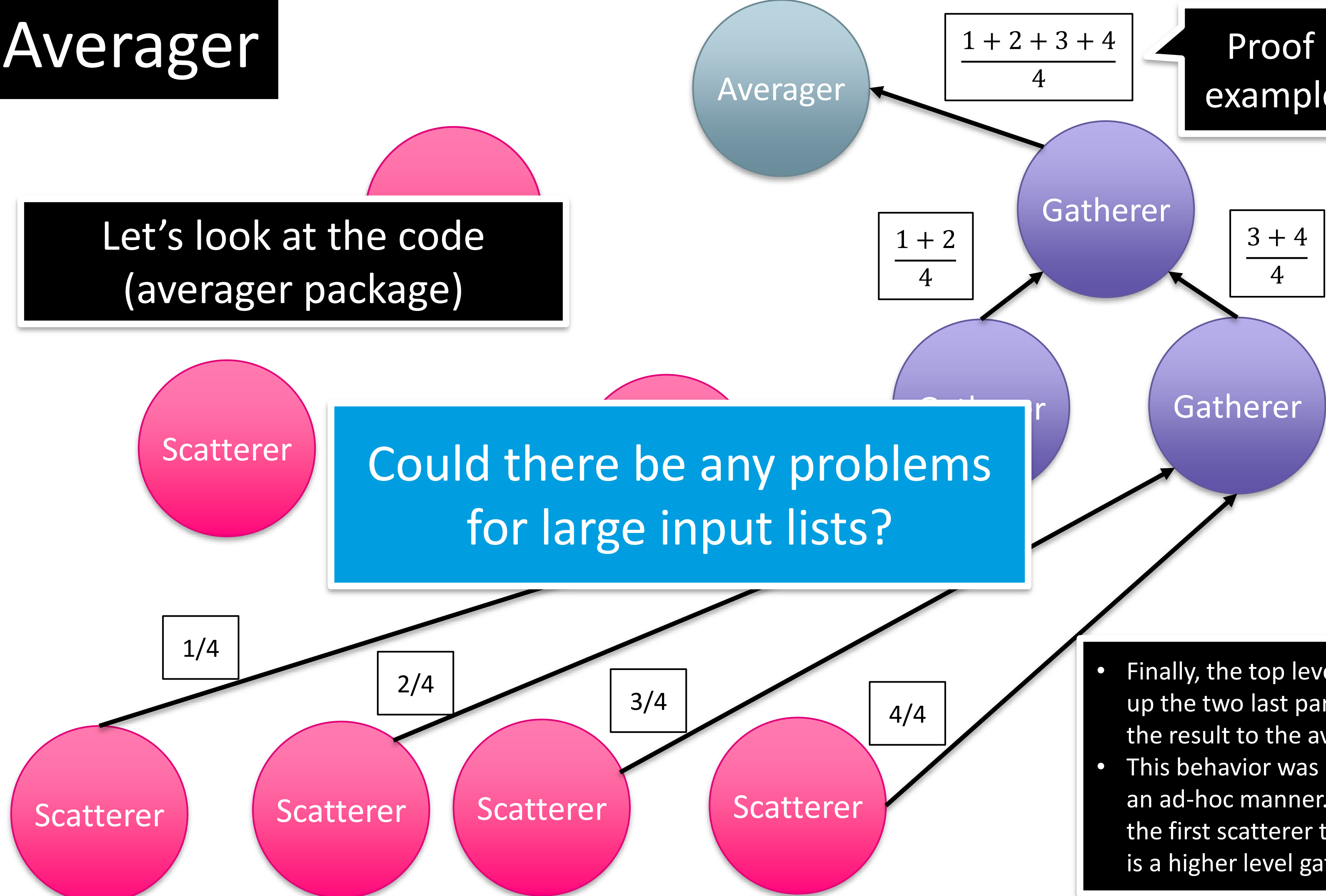
Let's look at the code (averager package)



- Finally, the top level gatherer adds up the two last partial results sends the result to the averager process
- This behavior was implemented in an ad-hoc manner. We specified for the first scatterer that the averager is a higher level gatherer.

Averager

Let’s look at the code (averager package)



Proof by example ☺

Could there be any problems for large input lists?

- Finally, the top level gatherer adds up the two last partial results sends the result to the averager process
- This behavior was implemented in an ad-hoc manner. We specified for the first scatterer that the averager is a higher level gatherer.

Other approaches to load balancing

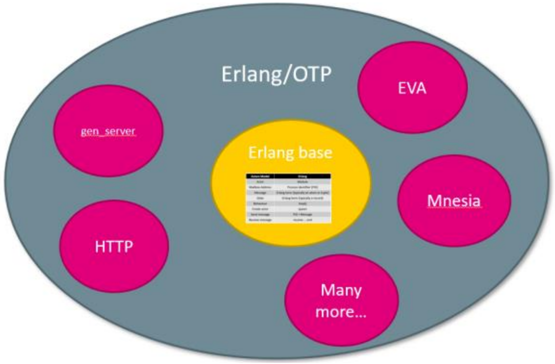
- The size of the problem does not necessarily need to determine the distribution of computation

- One may have HW restrictions
  - As we saw, actor 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 the number of active actors proportional to the workload
  - The exercises for this week target implementing an elastic server

More Erlang please!

- Recall that we have only used a small part of Erlang's functionalities (which, as you might noticed, are extremely powerful)



- If you want to learn more Erlang at ITU:
  - Join the software analysis specialization! In the course "Advanced Programming with Types" you will use Erlang to implement systems and types for verification

Examination

Examination – Material

- The folder **`exam`** in the GitHub repository will contain by Tuesday Nov 26th
  - *The mandatory readings for the exam* (we can ask questions about any of these readings)
  - *Questions for the exam*

  Although *the list is preliminary and subject to change*, you can consider this an *almost final version*

- Please **read the list with mandatory reading and exam questions carefully** and ask for any clarifications/comments
  - **Send questions and/or topics to revisitto Raúl (raup@itu.dk) before Thursday Nov 28th**

- Week 14 will be mostly about addressing your question/comments

- *Questions and answers in the LearnIT forum are not part of the mandatory readings*
  - The Q&A forum will be closed soon after we finish the course

Examination – Preparation

- **Prepare a short presentation for each question**
  - You may find inspiration in this video https://www.youtube.com/watch?v=587aD3tWSGk

- Make a short agenda
  for the answer to each question
  1. Motivation for concept X
  2. Key elements
  3. Challenges/Shortcomings/Alternatives
  4. Code examples
     - **Use code examples from your assignments**

- **Thoroughly study the mandatory readings**

Examination – Process

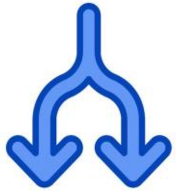
- The exam starts with a question you draw (at random) from the list of questions in GitHub
- Afterwards, the teachers and examiners may ask you anything from the mandatory readings
- While you answer a question, teachers and examiners may ask about specific details related to the question you are answering

Examination – What you can bring to the exam

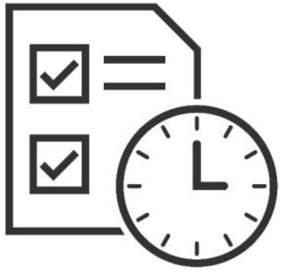
- One A4 paper (optional)
  - With the ***agenda*** for the short presentation you may prepare for each question
  - You cannot write full answers to the questions in this page
  - If we see you are reading from the paper, we will probably switch to other topics

- Your laptop or printout of the code
  - To show code example(s)

Mandatory assignments



- To be eligible for the exam, 5 (or more) mandatory assignments must be approved



- You will get confirmation
  in the feedback for assignment 6
  - *“Your assignments have been approved and you may take the exam”*

- <u>*It is your responsibility to let us know if there are any errors in grading*</u>
  - For instance, missing grades, ungraded assignment, etc.

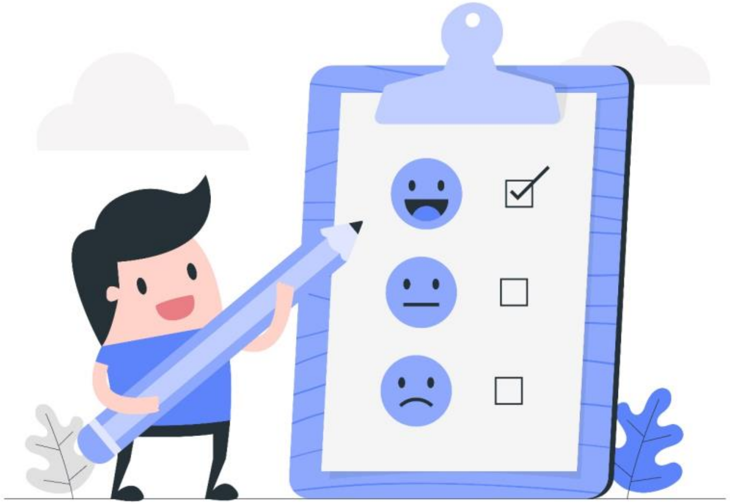
- There will be a final extra deadline on Dec 12th to hand-in assignments that have not yet been approved
  - With no possibility of re-submission and with written feedback

Examination – Dates

- Exam dates:
  - Week 2: January 10, (32 spots)
  - Week 3: January 13, 15, 16, 17 (80 spots)
  - Week 4: January 20, 21, 22, 23, 24 (80 spots)
- If you have constraints (e.g. other exams), please inform Raúl via e-mail (raup@itu.dk) by Dec 12th
  - We cannot guarantee that we will meet all constraints, but we will do our best
  - The more constraints we get, the more difficult it is to meet them
  - Please consider carefully whether your constraint is justified/reasonable
- The final schedule will be available in LearnIT in early January

Course Evaluation Survey

Please participate in the course evaluation



Questions ?