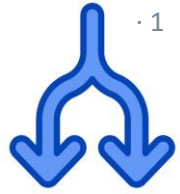
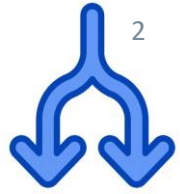


Practical Concurrent and Parallel Programming X

Message Passing I

Raúl Pardo

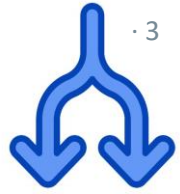
Note about visibility



- Part of the lecture code in lecture 4 was using volatile unnecessarily
  - https://github.itu.dk/jst/PCPP2021-public/blob/de5d21113ef8f79fd0467190e90a243dd07d4d4c/week04/code-lecture/week04lecture/app/src/main/java/lecture04/FairReadWriteMonitor.java#L9

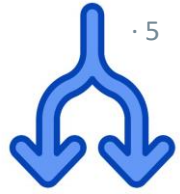
- We were using volatile only to ensure visibility, but it is not necessary due to the definition of happens-before in the Java memory model, in particular,
  - *"The write of the default value (zero, false, or null) to each variable synchronizes-with the first action in every thread."*
  - Since we were initializing integers to zero and Booleans to false, and access after initialization was protected by locks, volatile was unnecessary
  - https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4.4

Agenda



- Problems in shared memory concurrency (revisited)
- Actors
- Akka
- Example systems
  - Printer
  - Broadcaster (observer/observable)
  - Primer

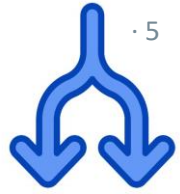
Problems in shared memory concurrency



"Writing thread-safe code is, at its core, about ***managing access to shared mutable data****"*

Goetz

Problems in shared memory concurrency



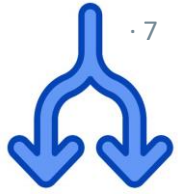
"Writing thread-safe code is, at its core, about ***managing access to shared mutable data****"*

Goetz

What problems have we seen in concurrent access to shared memory?

https://www.menti.com/w217eexk52

Problems in shared memory concurrency



"Writing thread-safe code is, at its core, about ***managing access to shared mutable data****"*

Goetz

What solutions have we seen to the problems in concurrent access to shared memory?

https://www.menti.com/w217eexk52
(same link as before)

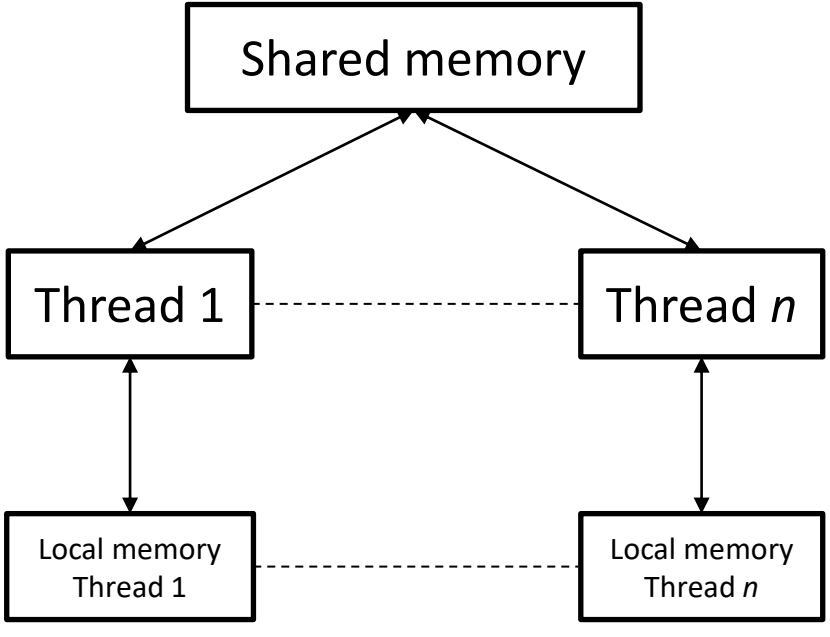
Message passing concurrency

- Threads do not share state
- If threads need to share data, then it is communicated by sending messages
- Threads work only on their own local memory

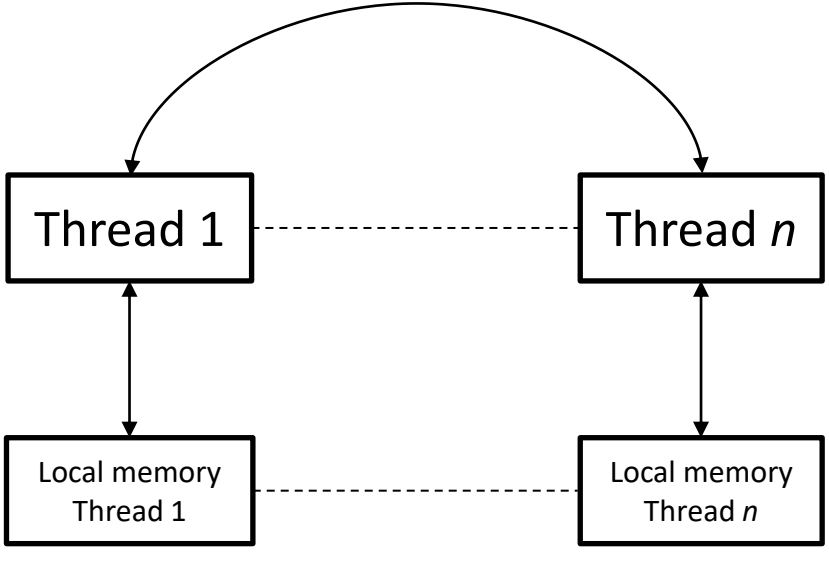
One of the designers of Erlang

Shared memory vs Message Passing

- Shared Memory
  - Synchronisation by writing in shared memory



- Message Passing
  - Synchronisation by sending messages



Message passing concurrency

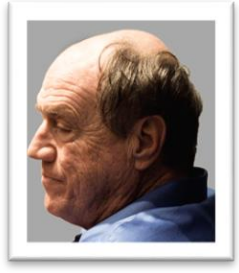
- How should we implement message passing concurrency?

- A possible solution is use standard communication systems
  - Sockets
  - Remote Procedure Calls (RPC)
  - Java Remote Method Invocation (RMI)
  - Message passing interfaces (MPI)

  combined with concurrency as we have seen so far

Message passing concurrency

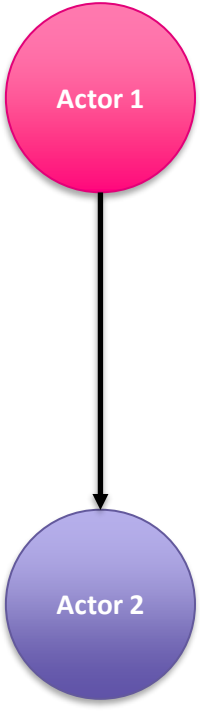
- How should we implement message passing concurrency?
- Another option is to ***use a concurrency model with message passing built-in***
  - That is, the ***actors model***!
- The actors model was first introduced by [Hewitt'73] and later formalized by [Agha'85] (part of the readings)
  - [Hewitt'73] - Carl Hewitt, Peter Bishop & Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence. 1973.
  - [Agha'85] – Gul A. Agha. ACTORS: A Model of Concurrent Computation in Distributed Systems. MIT Press. 1985.



Actors model

What is an Actor? (Bird's eye)

- 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.
- Actors can send messages to other actors



Actors in distributed systems

The actors model has natural mapping in distributed systems

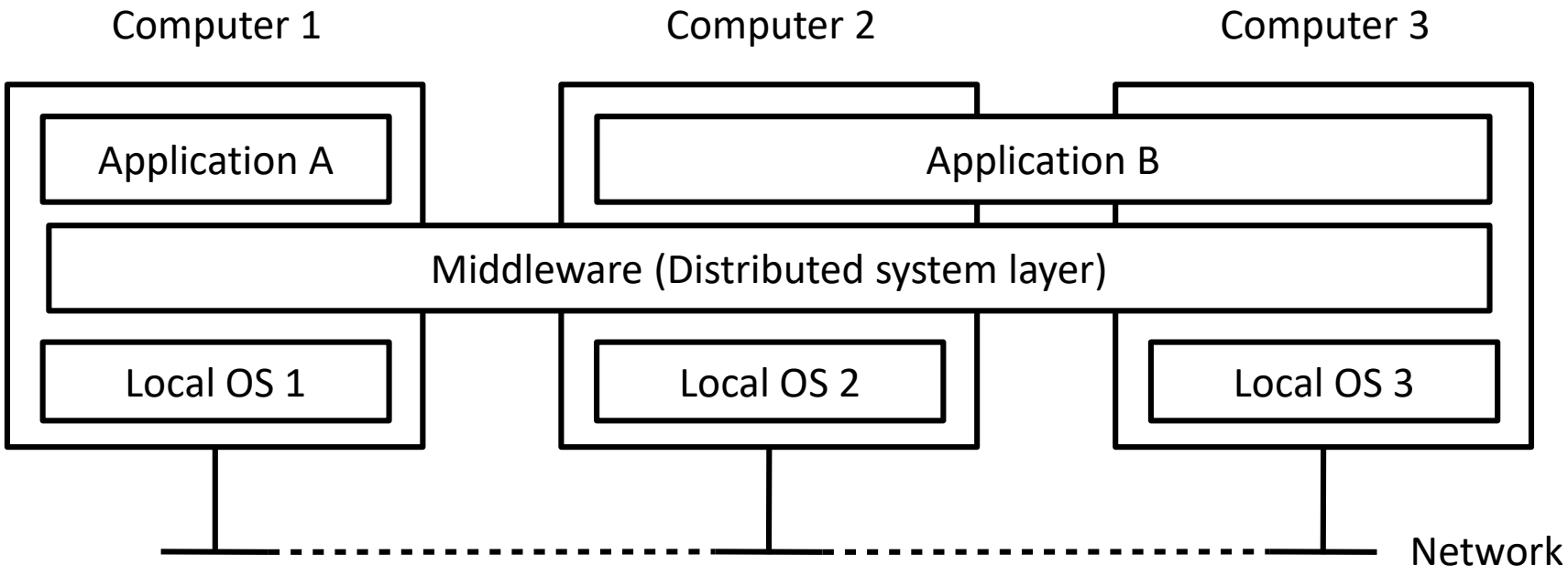
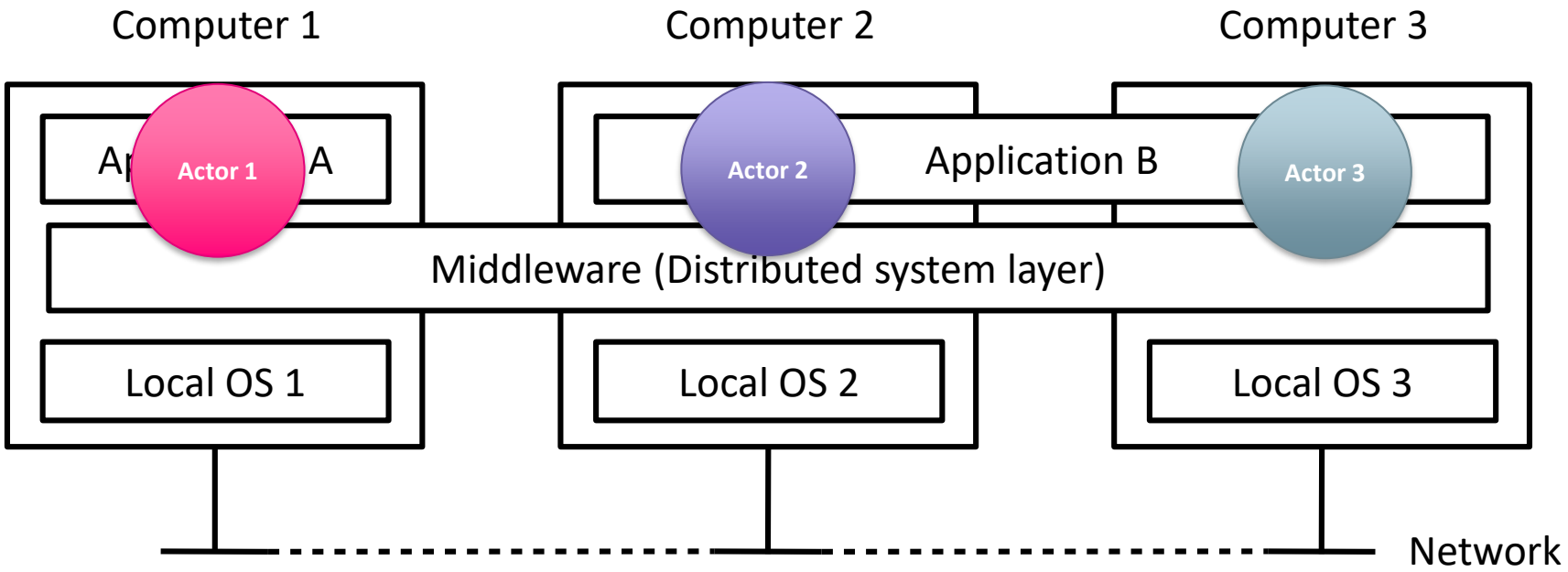


Figure taken from -> Distributed Systems: Principles and Paradigms. Andrew S. Tanenbaum and Maarten Van Steen. 2007.

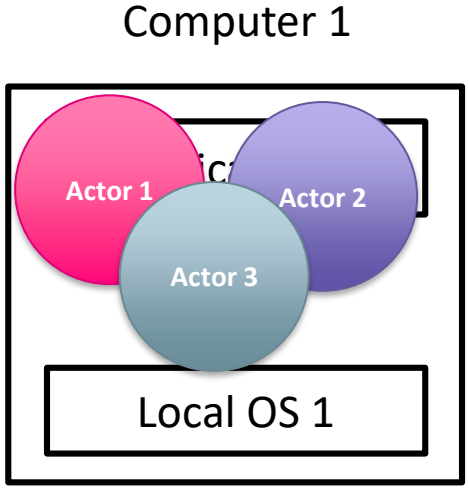
Actors in distributed systems

The actors model has natural mapping in distributed systems



Actors in a single computer

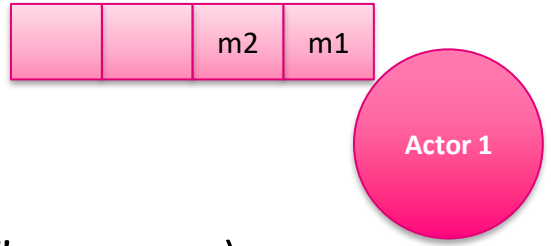
The actors model is applicable in a single computer as well



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

Actors

- An actor is an abstraction of a process (in the OS sense)
  - It can execute computation
  - It can create new actors (sub-processes)
- Actors *do not share memory*
  - They only have access to:
    - Their *local state* (local memory)
    - Their *mailbox* (multiset of fixed size with "received" messages)

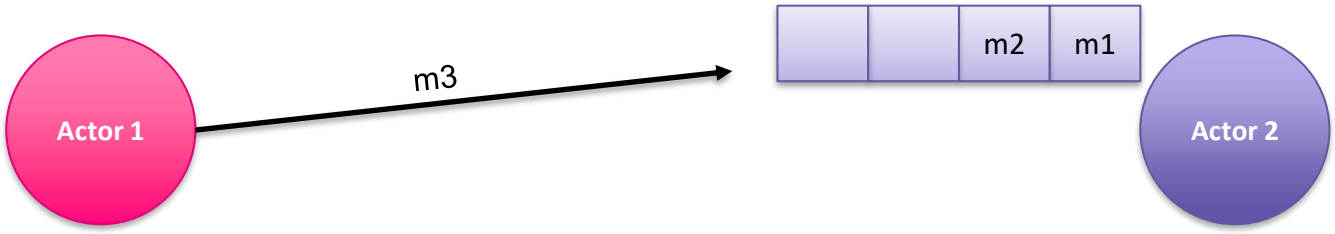


- Upon receiving a message an actor can
  - *send asynchronous messages* to other actors
  - *create new actors*
  - *change its behaviour* (local state and/or message handlers)

Actors and Messages

- Every actor in the system has a *unique identifier*
  - Other names for this identifier are: *mail address* or *actor reference*
  - Formally, an actor's mail address may be associated with several distinct references (this is not important for this course)
- Actors can
  - Send (finitely many) messages
  - Receive (finitely many) message
    - Received messages are placed in the actor's mailbox (asynchronous communication, see next slide)
- Messages include
  - Content of the message (arbitrary payload)

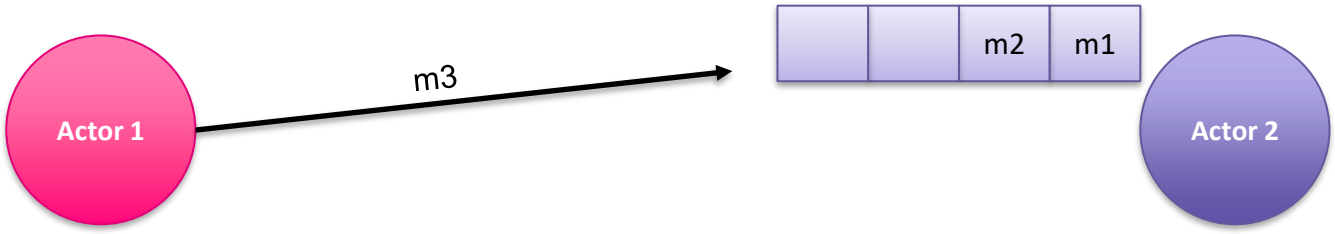
Asynchronous communication



- Asynchronous *send:*
  - The sender places the message in the mailbox of the receiver
  - It is *non-blocking*

- Asynchronous *receive:*
  - The receiver takes the message from the mailbox
  - The receiver *blocks* if the mailbox is empty

Asynchronous communication



- Asynchronous *send:*
  - The sender places the message in the mailbox of the receiver
  - It is *non-blocking*

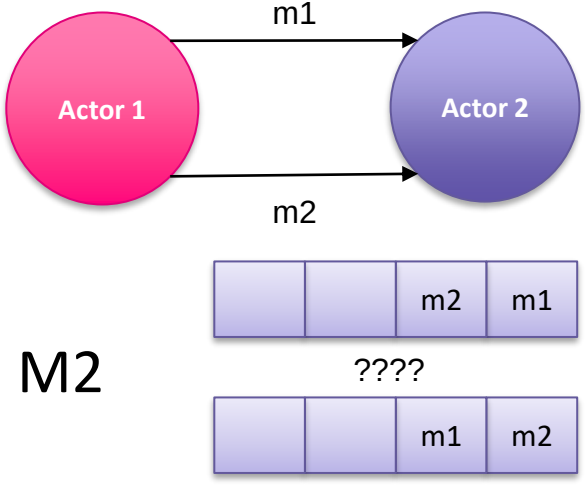
What is the difference with synchronous communication?

- Asynchronous *receive:*
  - The receiver takes the message from the mailbox
  - The receiver *blocks* if the mailbox is empty

No requirements on message arrival order

- No assumptions should be made about the order of arrival of messages

- For instance, consider this sequence of operations
  1. Actor1 sends message M1 to Actor2
  2. Actor1 sends message M2 to Actor2

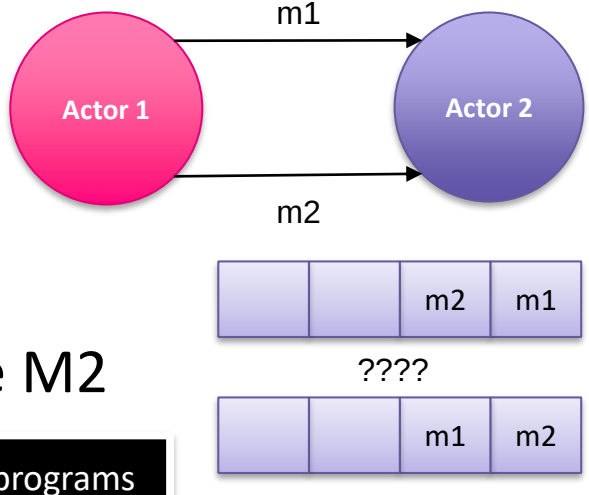


- It is *not* guaranteed that M1 arrives before M2

No requirements on message arrival order

- No assumptions should be made about the order of arrival of messages

- For instance, consider this sequence of operations
  1. Actor1 sends message M1 to Actor2
  2. Actor1 sends message M2 to Actor2



- It is *not* guaranteed that M1 arrives before M2

This is actually not true in Akka, but we will ignore that detail. Note that correct programs without this assumption will be correct if the assumption holds. But not viceversa.

Akka toolkit

Akka Toolkit

***Akka** is a **free and open-source toolkit** and runtime simplifying the construction of **concurrent** and distributed **applications** on the JVM. Akka supports multiple programming models for concurrency, but it emphasizes **actor-based concurrency**, with inspiration drawn from Erlang.*

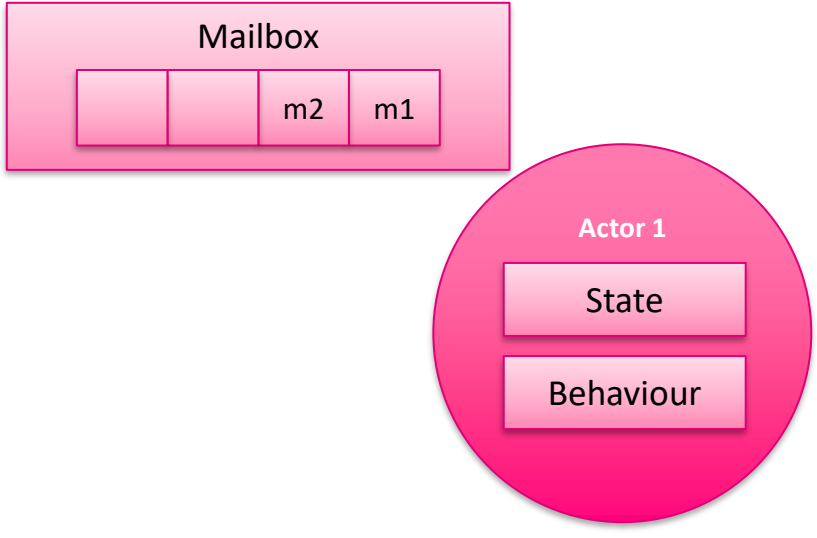
[Wikipedia]

Proven in production

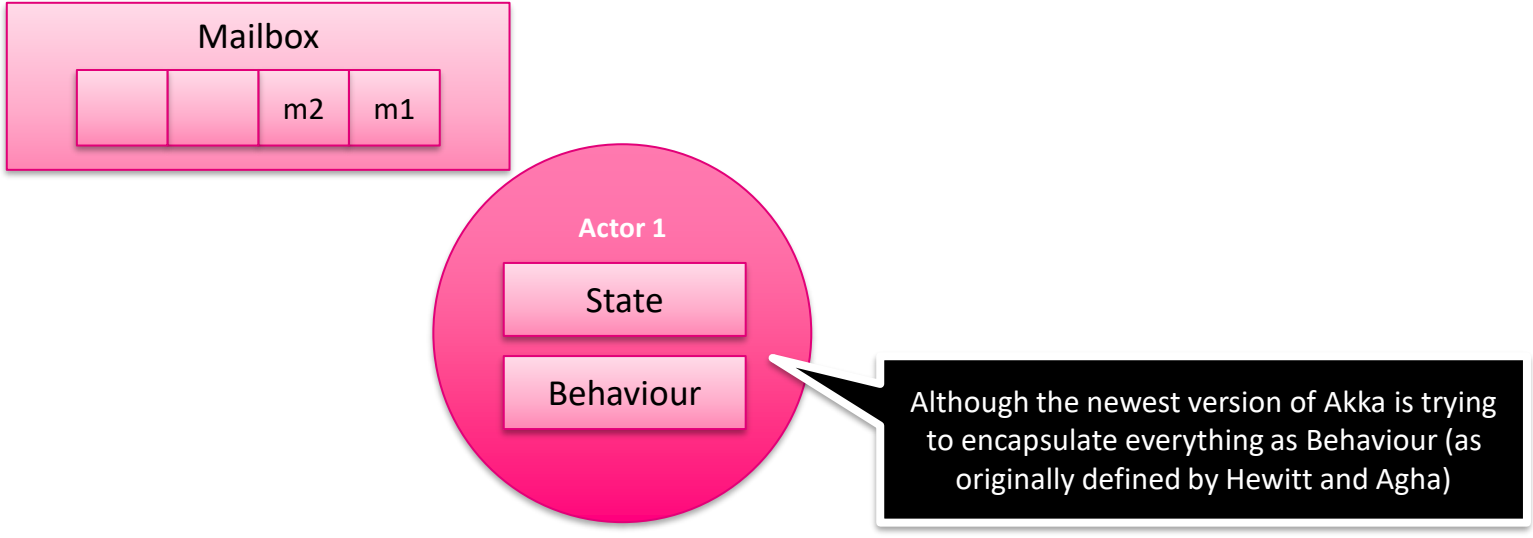
Organizations with extreme requirements rely on Akka and other Lightbend technologies. Read about their experiences in our case studies and learn more about how Lightbend can contribute to success with its commercial offerings.

iHeart MEDIA · CapitalOne · credit karma · intel · Hootsuite · NCL NORWEGIAN CRUISE LINE · UPSIDE · Walmart · PayPal · amazon.com · zalando · weightwatchers

Akka actors



Akka actors



Akka actors

- 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
Terminate actor	API function
Send message	API function
Receive message	Message handler builder (from API)

Defining Akka actors

- Actor class and local state

```
import akka.actor.typed.ActorRef;
import akka.actor.typed.Behavior;
import akka.actor.typed.javadsl.*;

public class Actor extends AbstractBehavior<T> {
    …

    /* --- State ---------------------------------------- */
    private int a1;
    private final List<String> a2;
    private ActorRef<T> a3;
    …

    …
}
```

Defining Akka actors

- Actor class and local state

The class to define actors is called AbstractBehavior (new Akka API), which is closer to the original definition of the Actors model

```
import akka.actor.typed.ActorRef;
import akka.actor.typed.Behavior;
import akka.actor.typed.javadsl.*;

public class Actor extends AbstractBehavior<T> {
    …

    /* --- State ---------------------------------------- */
    private int a1;
    private final List<String> a2;
    private ActorRef<T> a3;
    …

    …
}
```

Defining Akka actors

- Actor constructor and initial behaviour

```
public class Actor extends AbstractBehavior<T1> {
     …
    /* --- Constructor ---------------------------------- */
     private Actor(ActorContext<T> context,
                   int a1, ActorRef<T2> a3) {
          super(context)
          this.a1 = a1;
          this.a2 = new ArrayList<String>();
          this.a3 = a3;
          …
     }

     /* --- Actor initial behavior ----------------------- */
     public static Behavior<T1> create(int a1, ActorRef<T2> a3) {
          return Behaviors.setup(context -> new Actor(context,a1,a3));
     }
     …
}
```

Defining Akka actors

- Actor constructor and initial behaviour

Note that the constructor is defined as private

```
public class Actor extends AbstractBehavior<T1> {
     …
    /* --- Constructor ---------------------------------- */
     private Actor(ActorContext<T> context,
                   int a1, ActorRef<T2> a3) {
          super(context)
          this.a1 = a1;
          this.a2 = new ArrayList<String>();
          this.a3 = a3;
          …
     }

     /* --- Actor initial behavior ----------------------- */
     public static Behavior<T1> create(int a1, ActorRef<T2> a3) {
          return Behaviors.setup(context -> new Actor(context,a1,a3));
     }
     …
}
```

Defining Akka actors

- Actor constructor and initial behaviour

```
public class Actor extends AbstractBehavior<T1> {
     …
    /* --- Constructor ---------------------------------- */
     private Actor(ActorContext<T> context,
                   int a1, ActorRef<T2> a3) {
          super(context)
          this.a1 = a1;
          this.a2 = new ArrayList<String>();
          this.a3 = a3;
          …
     }

     /* --- Actor initial behavior ----------------------- */
     public static Behavior<T1> create(int a1, ActorRef<T2> a3) {
          return Behaviors.setup(context -> new Actor(context,a1,a3));
     }
     …
}
```

Note that the constructor is defined as private

Returns an behaviour with initial values for all the elements of the local state.

Defining Akka actors

- Actor constructor and initial behaviour

```
public class Actor extends AbstractBehavior<T1> {
     …
    /* --- Constructor ---------------------------------- */
     private Actor(ActorContext<T> context,
                   int a1, ActorRef<T2> a3) {
          super(context)
          this.a1 = a1;
          this.a2 = new ArrayList<String>();
          this.a3 = a3;
          …
     }

     /* --- Actor initial behavior ----------------------- */
     public static Behavior<T1> create(int a1, ActorRef<T2> a3) {
          return Behaviors.setup(context -> new Actor(context,a1,a3));
     }
     …
}
```

Note that the constructor is defined as private

We must pass the Actor context (with info about the complete Actor system) to the constructor of the parent class

Returns an behaviour with initial values for all the elements of the local state.

Messages in Akka

- The message that the actor can handle are defined as inner classes of the Actor class

```
public class Actor extends AbstractBehavior<ActorMessage> {
    …
    /* --- Messages ------------------------------------ */
    public interface ActorMessage { }

    public static final Message1 implements ActorMessage {
        public final String content;
        public final ActorRef<T> sender;

        public Message1(String content, ActorRef<T> sender) {…}
    }

    public static final PrintState implements ActorMessage { }
    …
}
```

Messages in Akka

- The message that the actor can ha…ner classes of the Actor class

The AbstractBehavior is parameterized with the type of messages the actor handles

```
public class Actor extends AbstractBehavior<ActorMessage> {
    …
    /* --- Messages ------------------------------------ */
    public interface ActorMessage { }

    public static final Message1 implements ActorMessage {
        public final String content;
        public final ActorRef<T> sender;

        public Message1(String content, ActorRef<T> sender) {…}
    }

    public static final PrintState implements ActorMessage { }
    …
}
```

Messages in Akka

If the actor handles more than one type of message, it is common to define a top level interface and implement it for each type of message

The AbstractBehavior is parameterized with the type of messages the actor handles

: the actor can ha…er
…or class

```
public cla…or extends AbstractBehavior<ActorMessage> {
    …
    /* --- Message… ------------------------------------ */
    public interface ActorMessage { }

    public static final Message1 implements ActorMessage {
        public final String content;
        public final ActorRef<T> sender;

        public Message1(String content, ActorRef<T> sender) {…}
    }

    public static final PrintState implements ActorMessage { }
    …
}
```

Messages in Akka

If the actor handles more than one type of message, it is common to define a top level interface and implement it for each type of message

The AbstractBehavior is parameterized with the type of messages the actor handles

: the actor can ha… …er
…or class

```
public cla…or extends AbstractBehavior<ActorMessage> {
    …
    /* --- Message… ------------------------------------ */
    …blic interface ActorMessage { }

    …lic static final Message1 implements ActorMessage {
        public final String content;
        public final ActorRef<T> sender;

        public Message1(String content, ActorRef<T> sender) {…}


    …blic static final PrintState implements ActorMessage { }
    …
}
```

Message classes are recommended to be immutable, so it is common practice to define them as static final, and fields as final. ***In the course, it is a requirement that message classes are immutable***

Messages in Akka

- Message handling

```
public class Actor extends AbstractBehavior<ActorMessage> {
    …
    /* --- Message handling ----------------------------- */
    @Override
    public Receive<ActorMessage> createReceive() {
     return newReceiveBuilder()
          .onMessage(Message1.class, this::onMessage1)
          .onMessage(PrintState.class, this::onPrintState)
          .build();
    }
    /* --- Handlers ------------------------------------- */
    public Behavior<ActorMessage> onMessage1(Message1 msg) {
          // code to handle msg
          return this;
    }
    public Behavior<ActorMessage> onPrintState(PrintState msg) {…}
    …
}
```

Messages in Akka

- Message handling

```
public class Actor extends AbstractBehavior<ActorMes
    …
    /* --- Message handling ------------------------
    @Override
    public Receive<ActorMessage> createReceive() {
     return newReceiveBuilder()
          .onMessage(Message1.class, this::onMessage1)
          .onMessage(PrintState.class, this::onPrintState)
          .build();
    }
    /* --- Handlers ------------------------------------ */
    public Behavior<ActorMessage> onMessage1(Message1 msg) {
          // code to handle msg
          return this;
    }
    public Behavior<ActorMessage> onPrintState(PrintState msg) {…}
    …
}
```

We create a receive builder that defines the behavior to be executed depending on the message received

Akka actor class – Summary

- In summary an Akka actor class should have these elements
  1. Messages
  2. State
  3. Constructor
  4. Initial behaviour
  5. Message handler
  6. Handlers

- You may notice that all files in the code-lecture folder have the structure on the right to make it easier to write actor classes

```
public class Actor extends AbstractBehavior<ActorMessage> {

    /* --- Messages ------------------------------------ */
    …


    /* --- State --------------------------------------- */
    …


    /* --- Constructor --------------------------------- */
    private Actor(…) {…}


    /* --- Actor initial behavior ---------------------- */
    public static Behaviour<ActorMessage> create(…) {…}


    /* --- Message handling ---------------------------- */
    @Override
    public Receive<PrimerCommand> createReceive() {…}


    /* --- Handlers ------------------------------------ */
    …
}
```

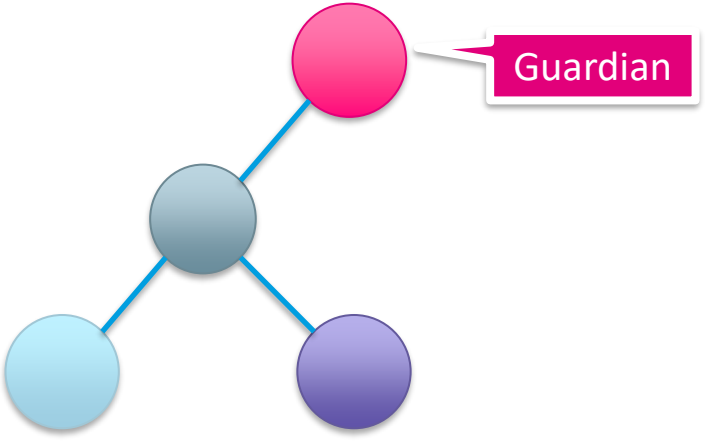
Akka actors

- 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
Terminate actor	API function
Send message	API function
Receive message ✓	Message handler builder (from API)

Akka actors – Guardian

- Akka actor systems have an implicit hierarchical structure



- The first actor to be created in the system is a top-level actor known as *guardian*, this actor is created as follows

```
final ActorSystem<ActorMessage> guardian = ActorSystem.create(Actor.create(…), "actor_system");
```

This should be placed in a Main file

The constructor takes the initial behaviour of the actor and its name

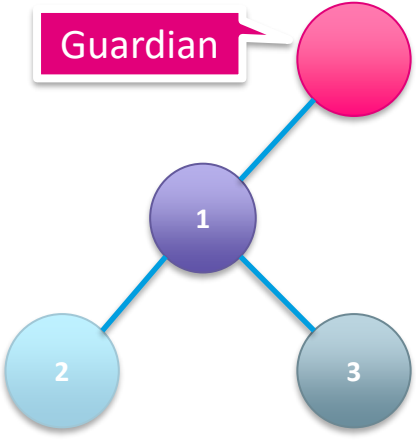
Akka actors

- Standard way of creating new actors (not the top-level one)
  - Using spawn from the current context

The constructor takes the initial behaviour of the actor and its name

```
ActorRef<Actor> actor = this.getContext().spawn(Actor.create(…), "1");
```

- This call can be made anywhere inside the actor code
  - During creation and initialization of the actor
  - In a message handling method
    - This is the most common use case



- All actors spawned by an actor become their children
  - On the right, all actors are children of the guardian, and 2 and 3 are children of 1.

Akka actors

- An actor can terminate its execution in a message handler by returning a stop behaviour, e.g.,

```
/* --- Handlers ------------------------------------ */
public Behavior<ActorMessage> onStopExecution(StopExecution msg) {
     // code to handle msg
     return Behaviors.stop();
}
```

- It is common to define a specific message to terminate the execution of an actor

- Parents can directly terminate the execution of their children

Akka actors

- The function tell(…) is used to send asynchronous messages

```
public class GuardianActor extends AbstractBehavior<T> {
     …
     actor1.tell(new Message(…));
     …
}
```

Akka actors

- The function tell(…) is used to send asynchronous messages

```
public class GuardianActor extends AbstractBehavior<T> {
      …
      actor1.tell(new Message(…));
      …
}
```

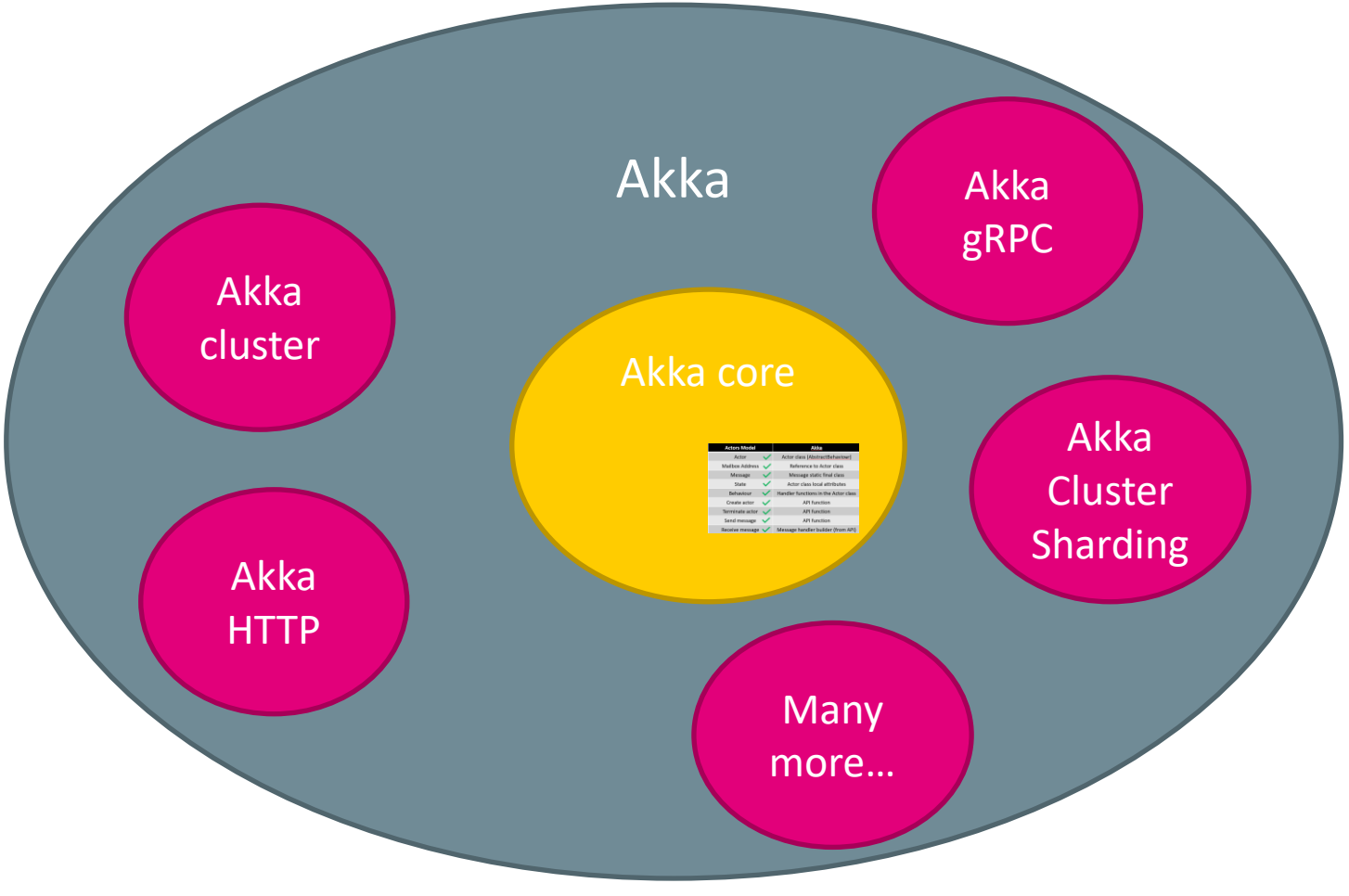
Note that this is the receiver actor, the sender is implicit

Akka actors

- 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
Terminate actor ✓	API function
Send message ✓	API function
Receive message ✓	Message handler builder (from API)

We only use a tiny bit of Akka



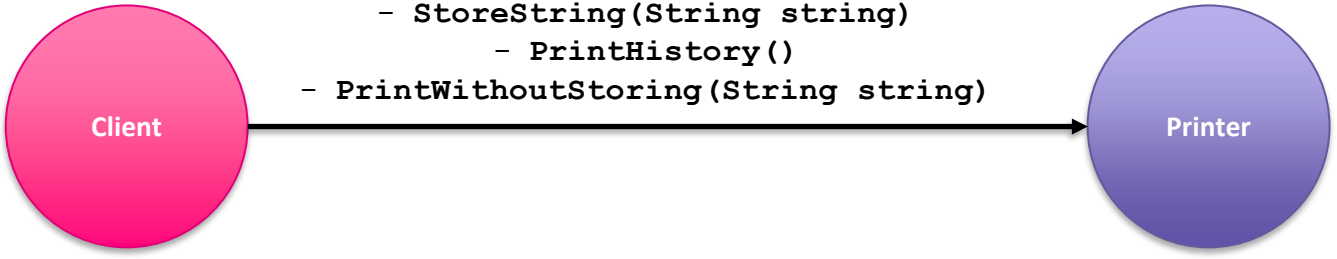
Example Actor Systems

Example actor systems with static topology

1. Printer
2. Broadcast (observer)
3. Primer

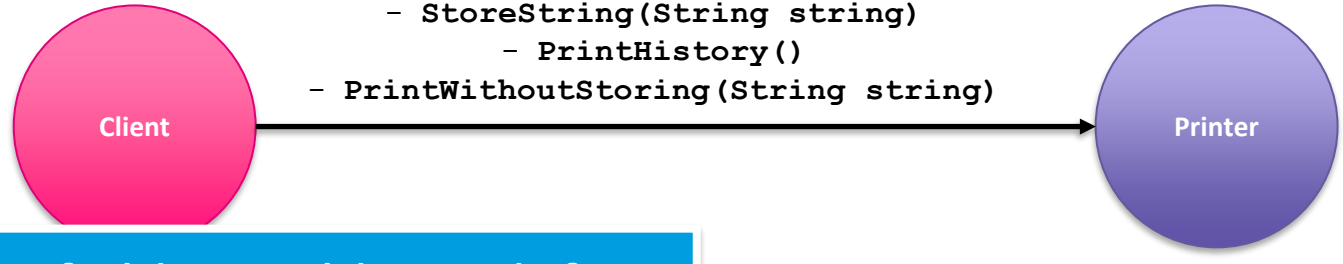
Printer

- A printer actor can receive 3 type of messages
  - StoreString(String string) – It stores the sent string in its local state
  - PrintHistory() – Prints all stored messages
  - PrintWithoutStoring(String string) – Prints the string without storing it



Printer

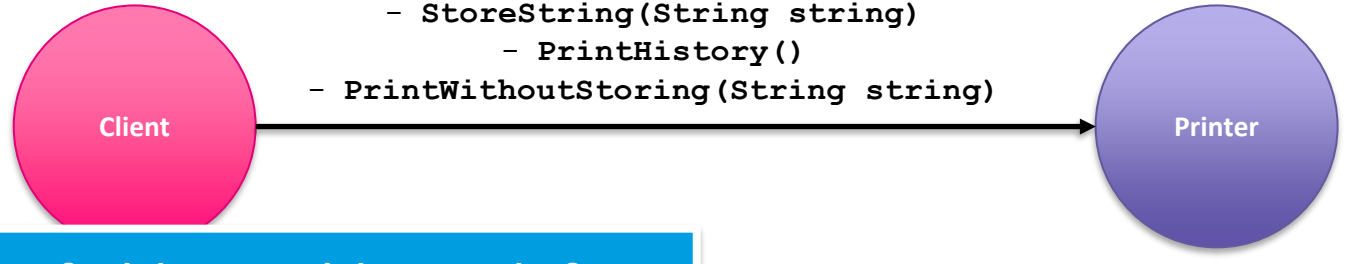
- A printer actor can receive 3 type of messages
  - StoreString(String string) – It stores the sent string in its local state
  - PrintHistory() – Prints all stored messages
  - PrintWithoutStoring(String string) – Prints the string without storing it



What fields would you define for the local state of Printer?

Printer

- A printer actor can receive 3 type of messages
  - StoreString(String string) – It stores the sent string in its local state
  - PrintHistory() – Prints all stored messages
  - PrintWithoutStoring(String string) – Prints the string without storing it

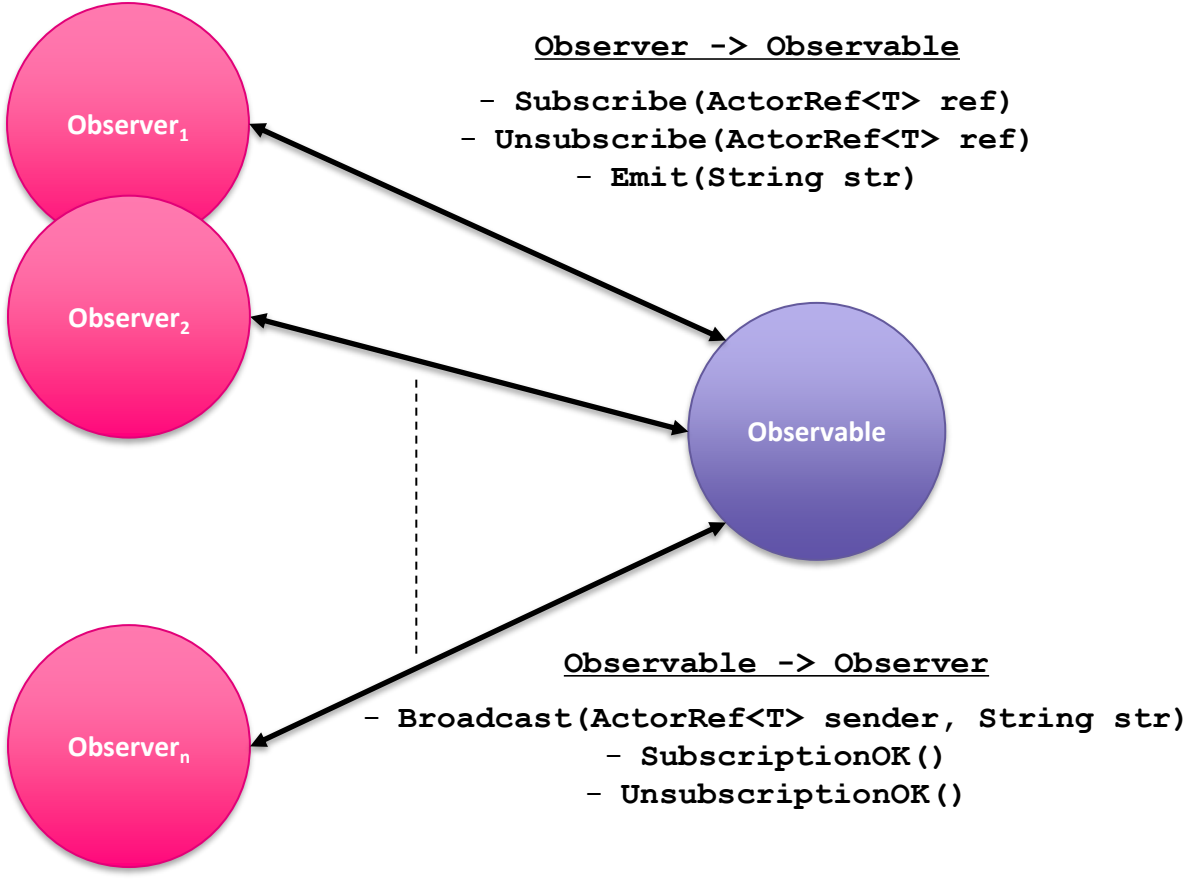


What fields would you define for the local state of Printer?

Let's look at the code (printersystem package)

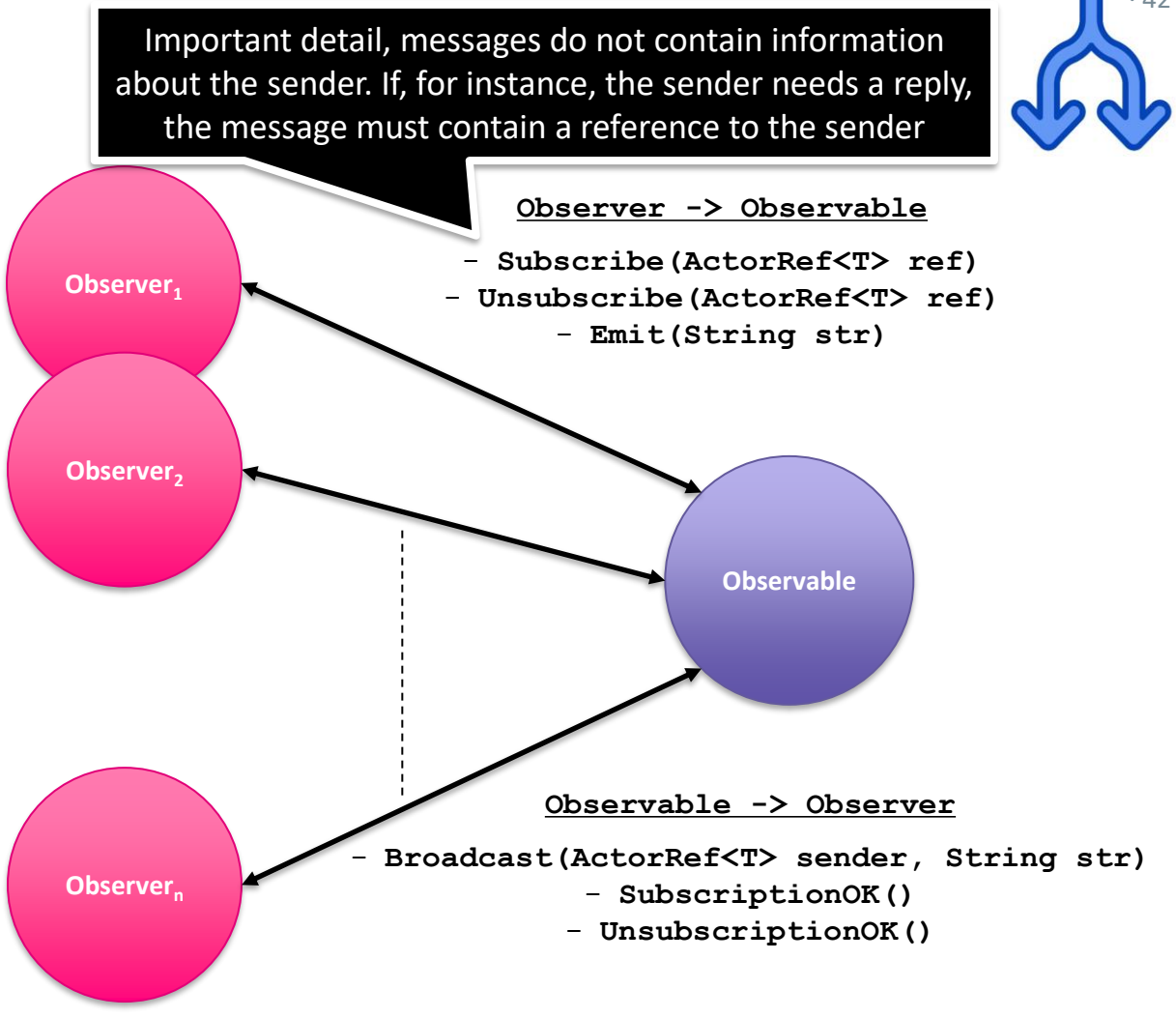
Broadcaster

- A set of observers actor may subscribe to an observable actor
  - The observable must confirm the subscription
- Observer may emit messages that the observable broadcasts to all subscribers (except for the sender)
- Observers may unsubscribe
  - The observable must confirm the un-subcription.



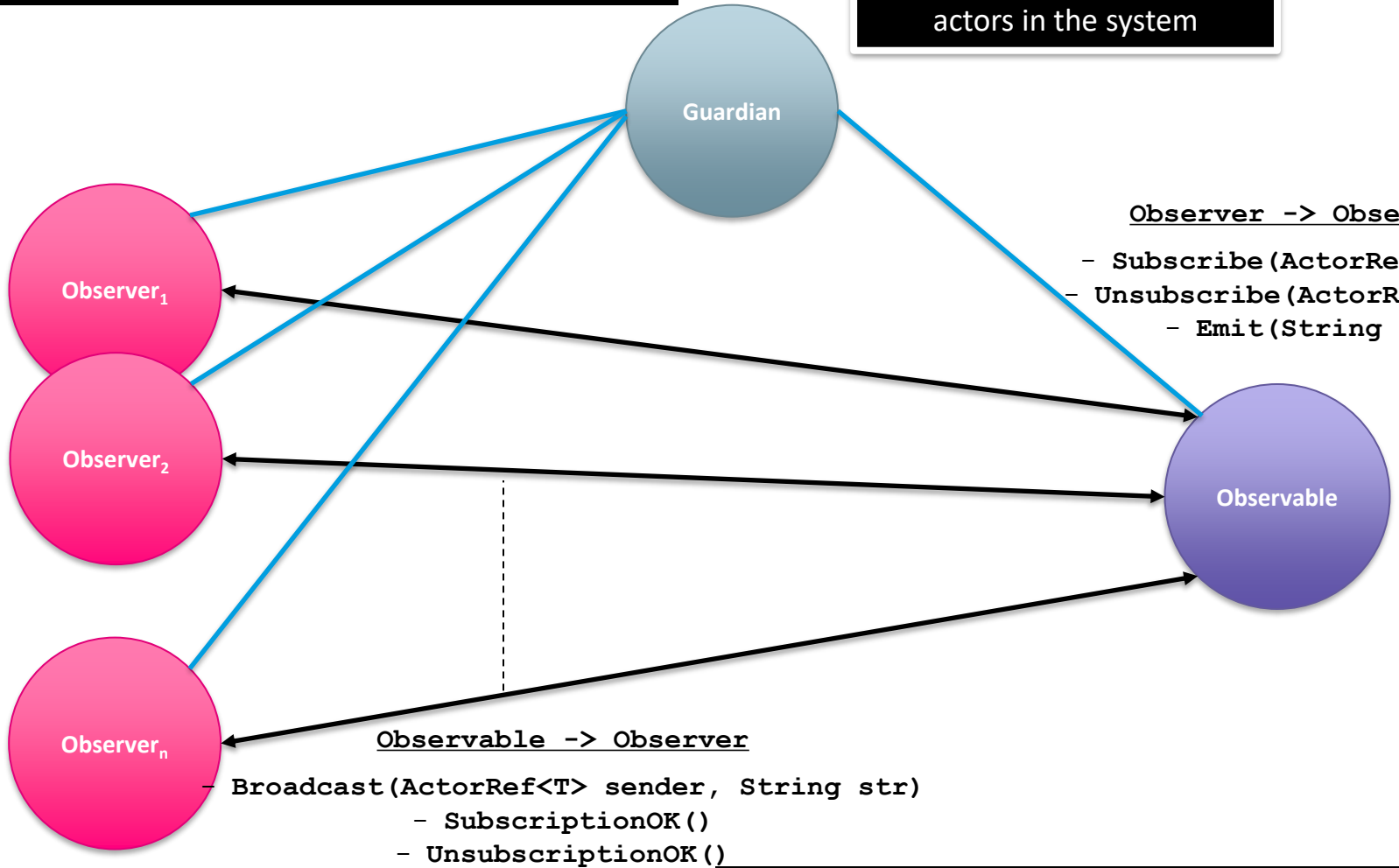
Broadcaster

- A set of observers actor may subscribe to an observable actor
  - The observable must confirm the subscription
- Observer may emit messages that the observable broadcasts to all subscribers (except for the sender)
- Observers may unsubscribe
  - The observable must confirm the un-subcription.



Broadcaster + Guardian

The guardian spawns all the actors in the system



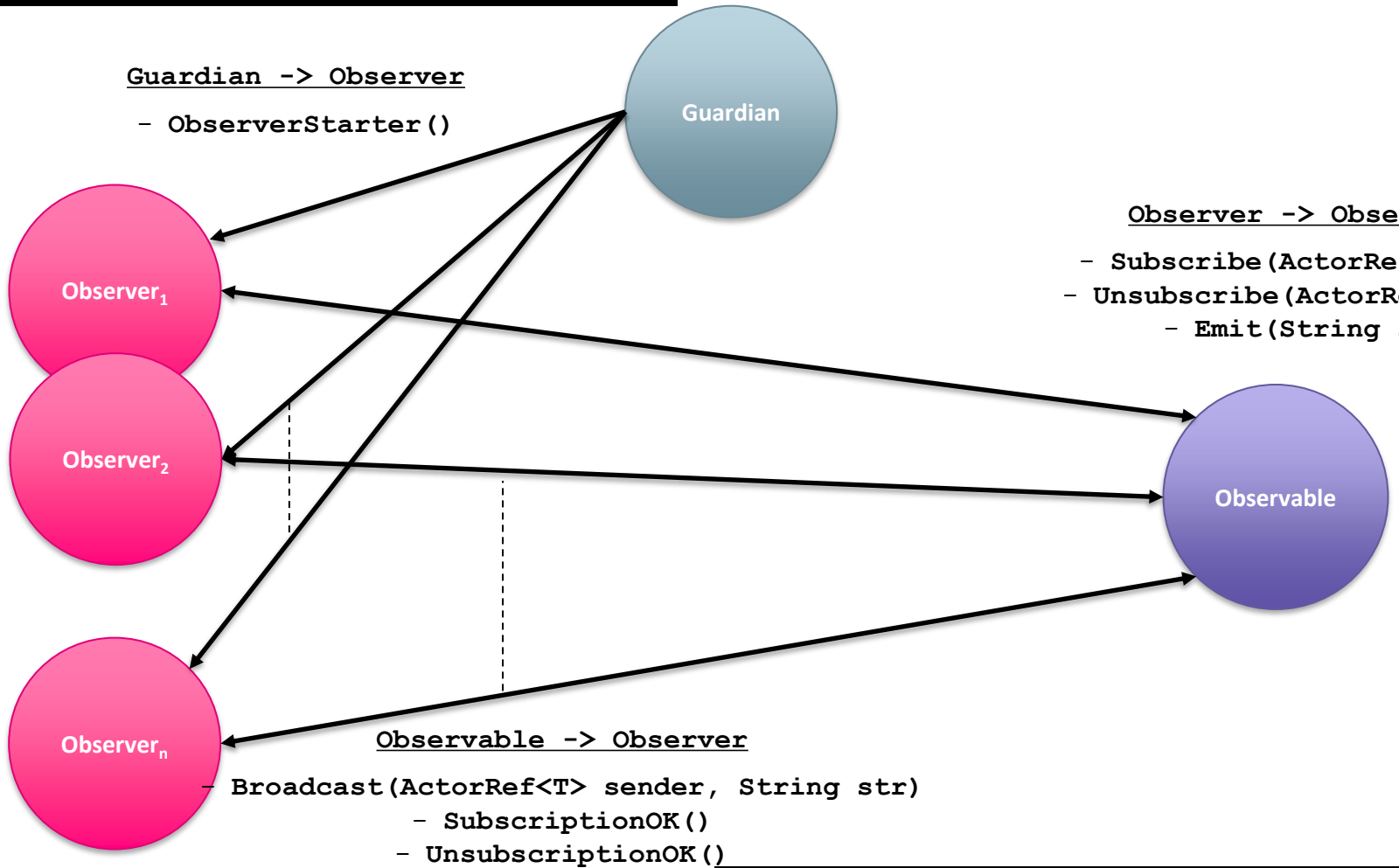
Observer -> Observable

- Subscribe(ActorRef<T> ref)
- Unsubscribe(ActorRef<T> ref)
- Emit(String str)

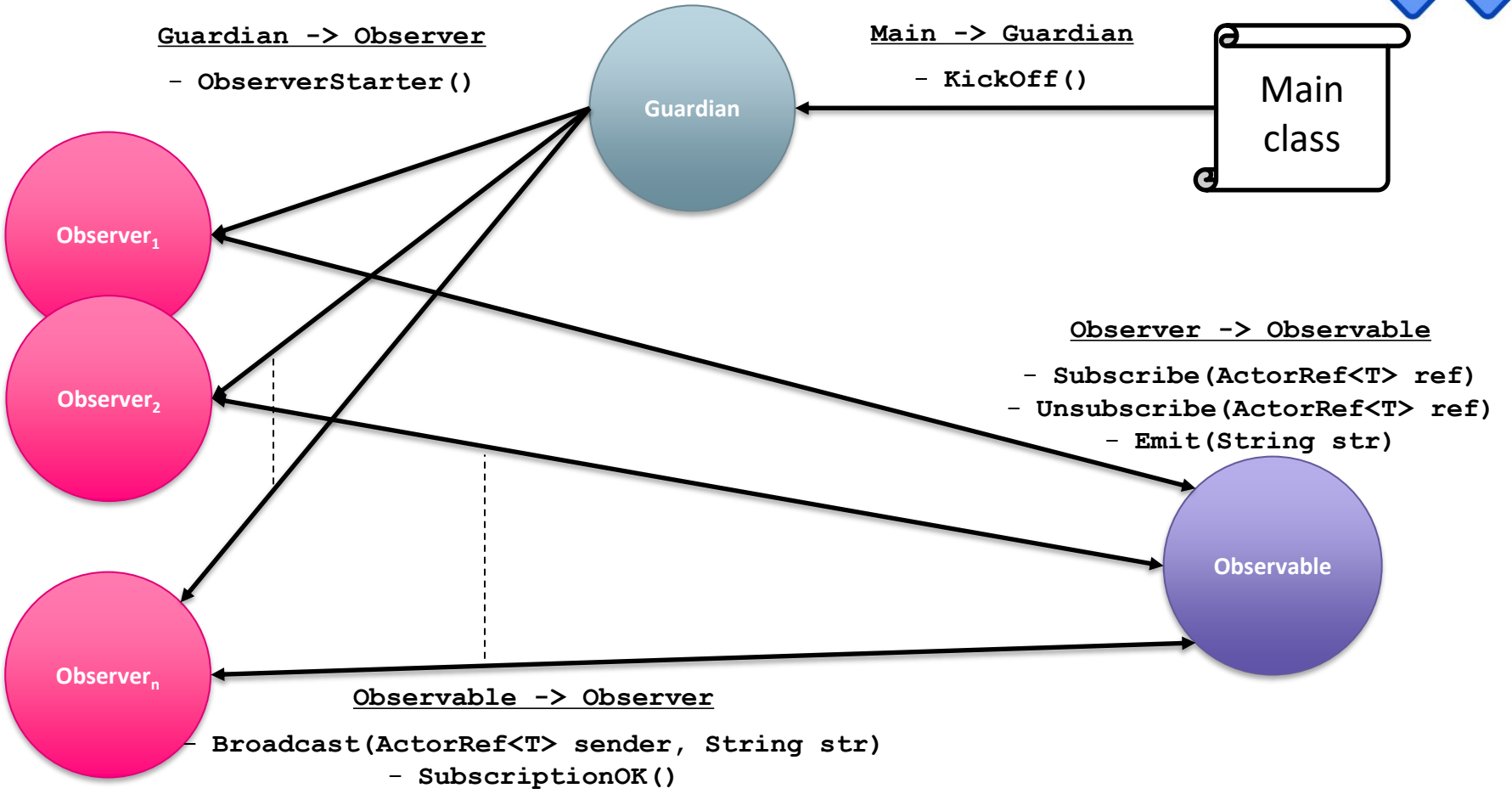
Observable -> Observer

- Broadcast(ActorRef<T> sender, String str)
- SubscriptionOK()
- UnsubscriptionOK()

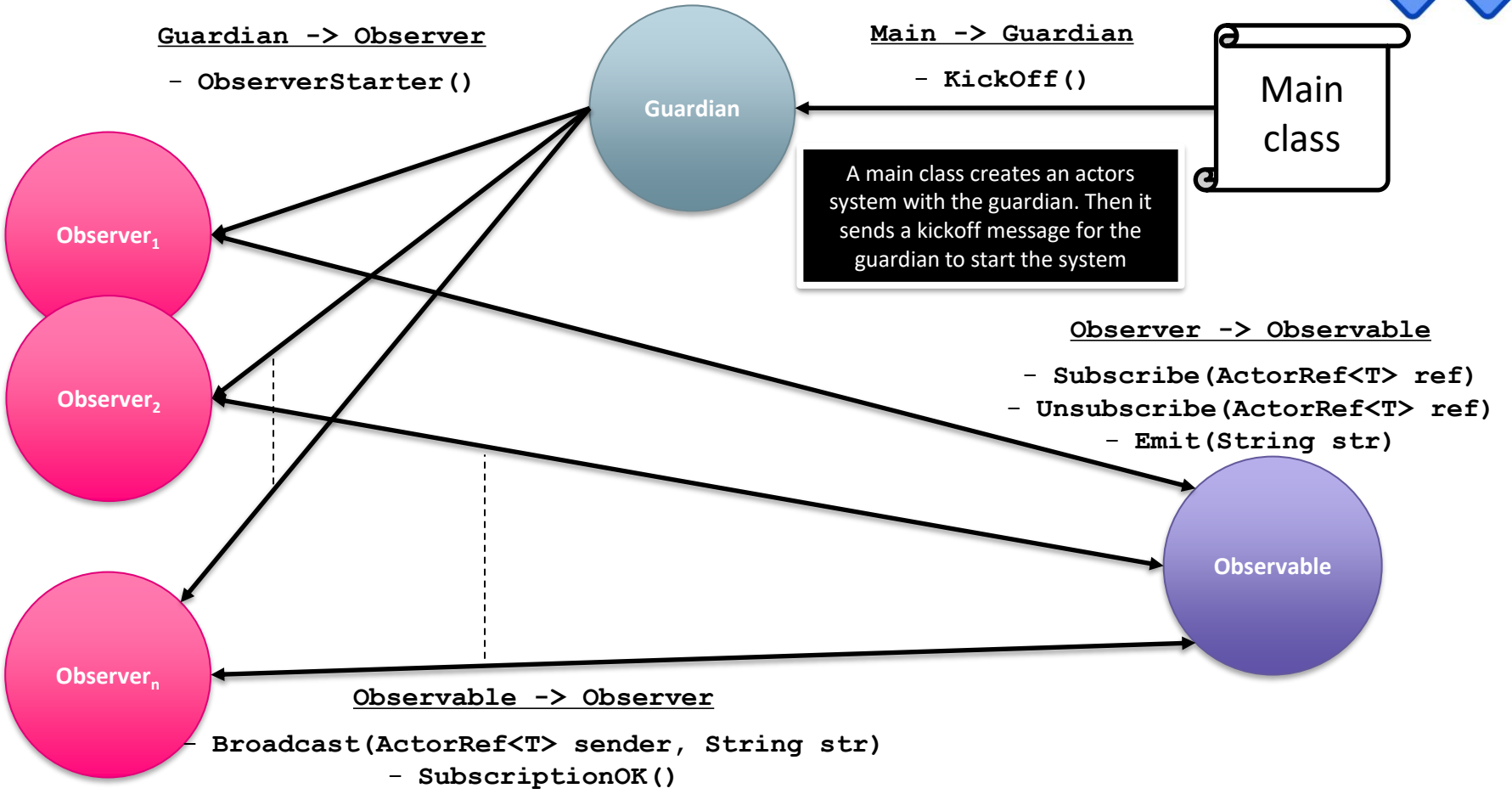
Broadcaster + Guardian



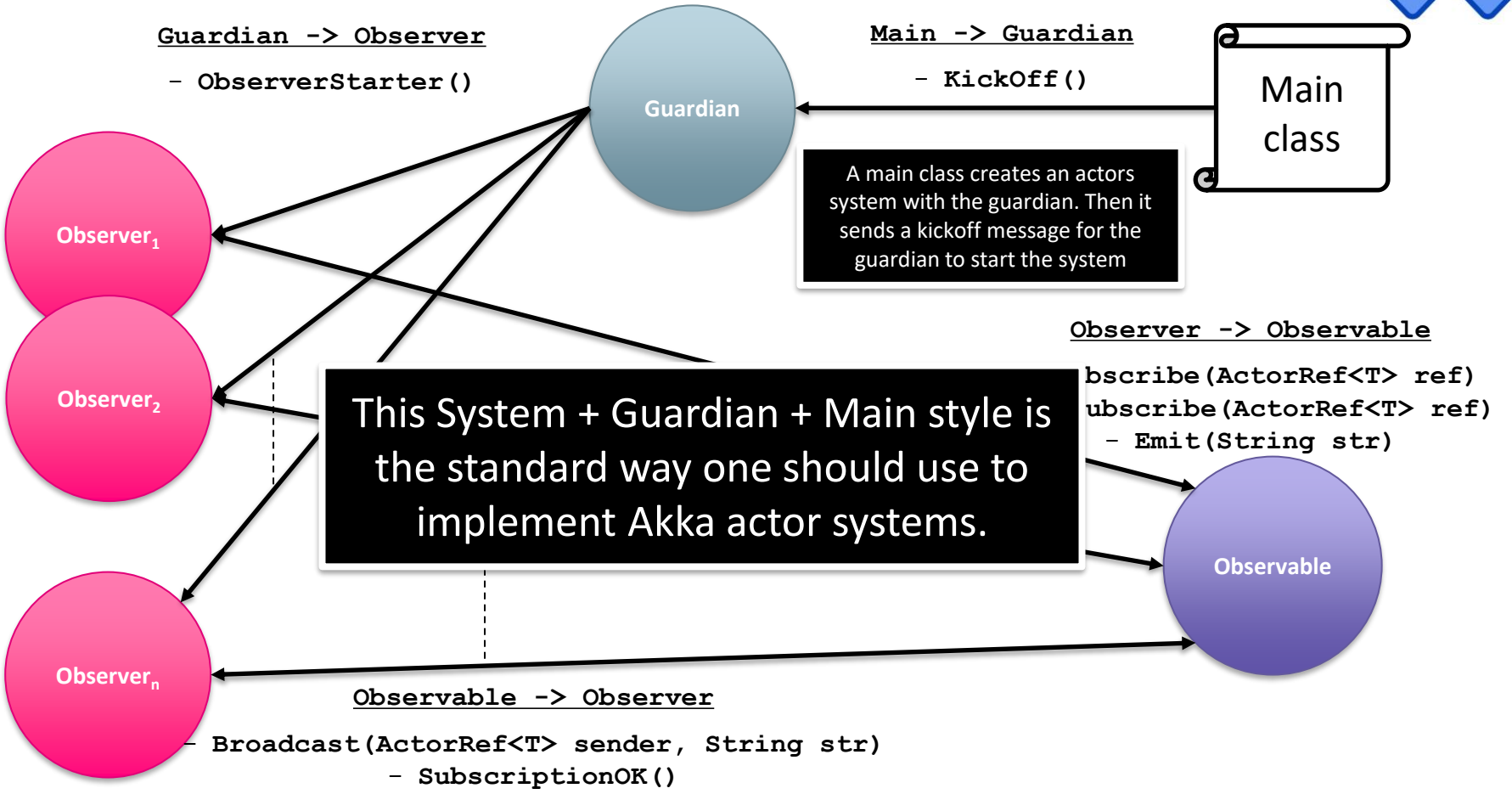
Broadcaster + Guardian + Main



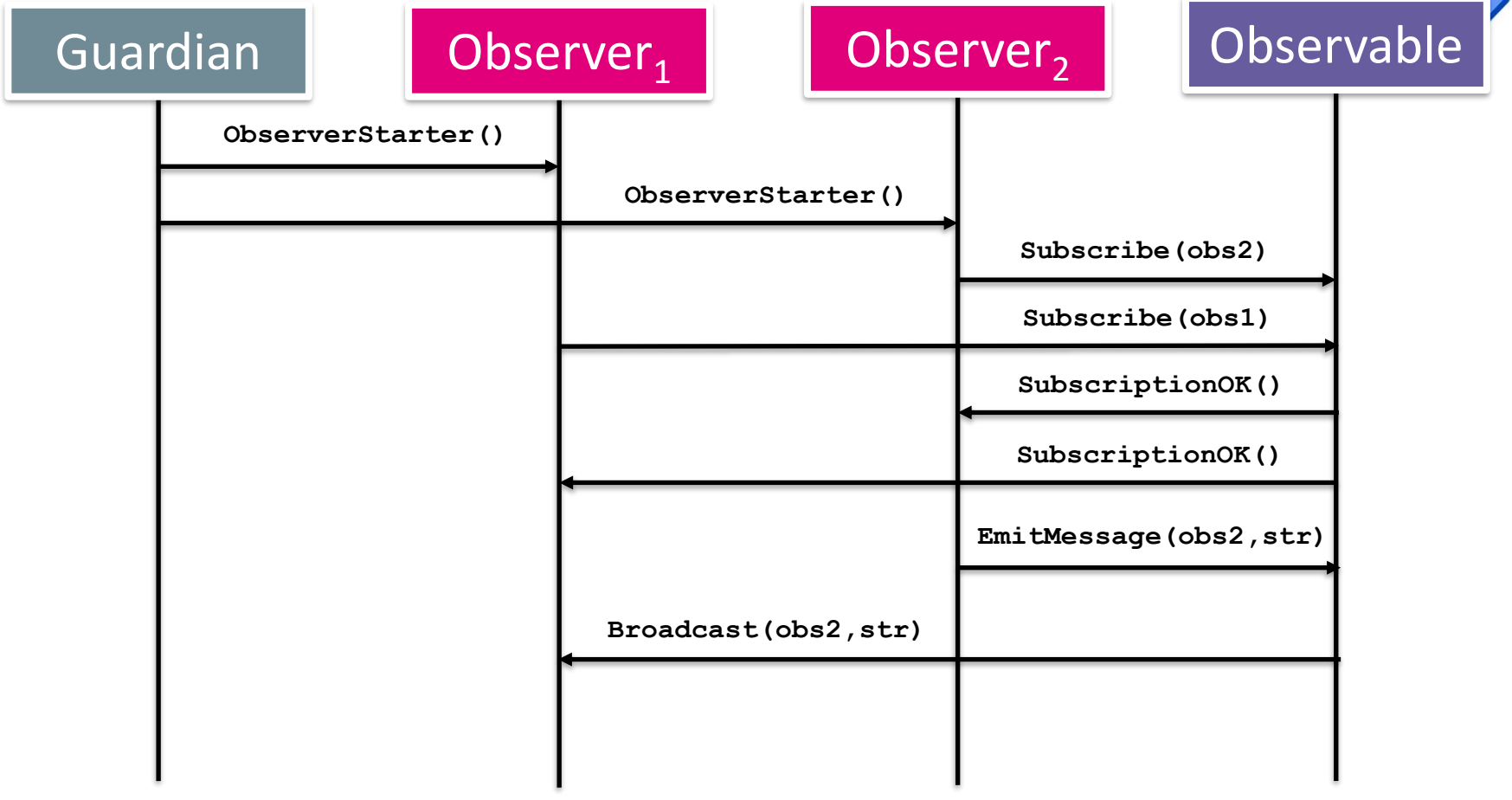
Broadcaster + Guardian + Main



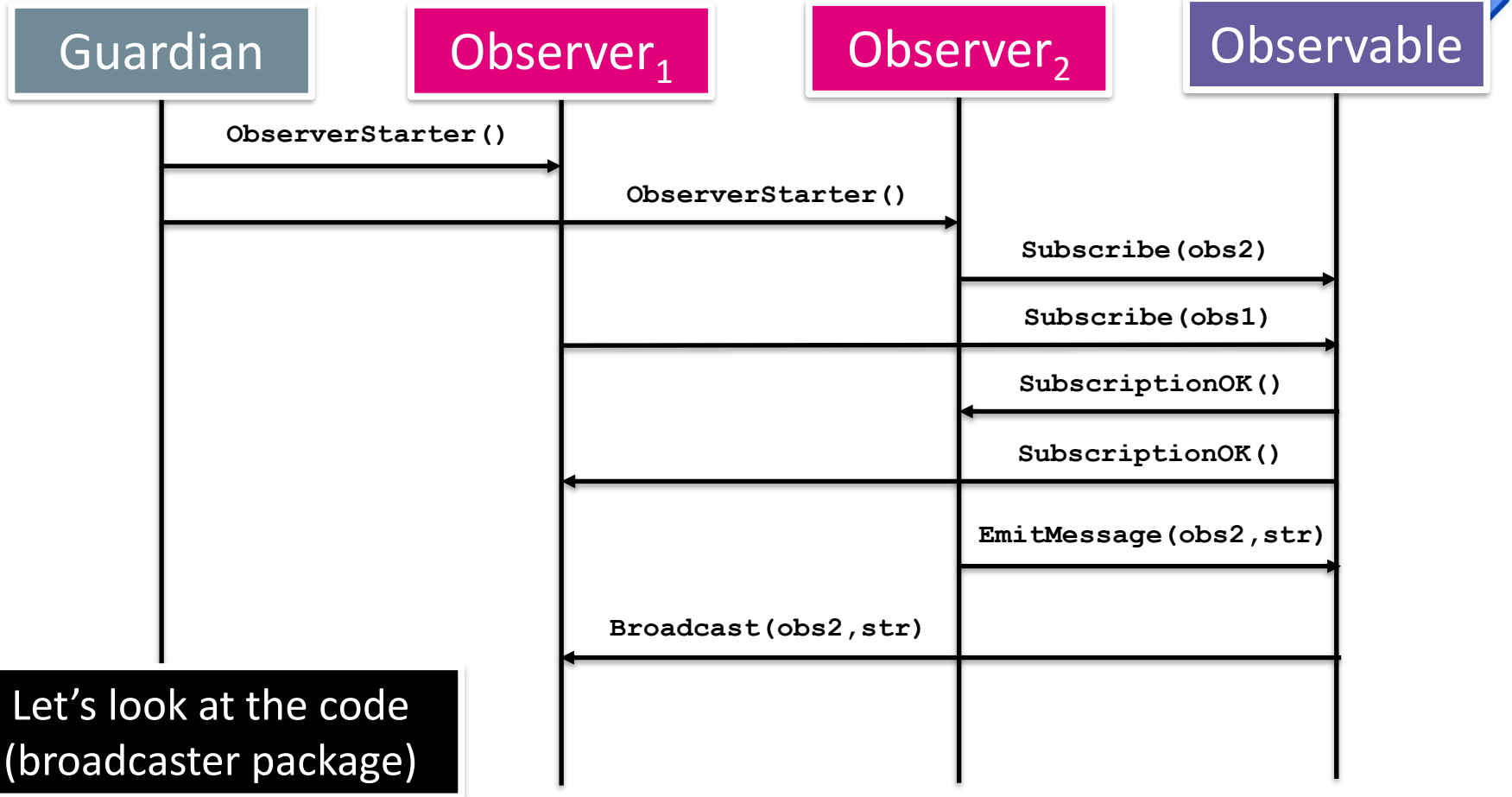
Broadcaster + Guardian + Main



Primer – execution example



Primer – execution example



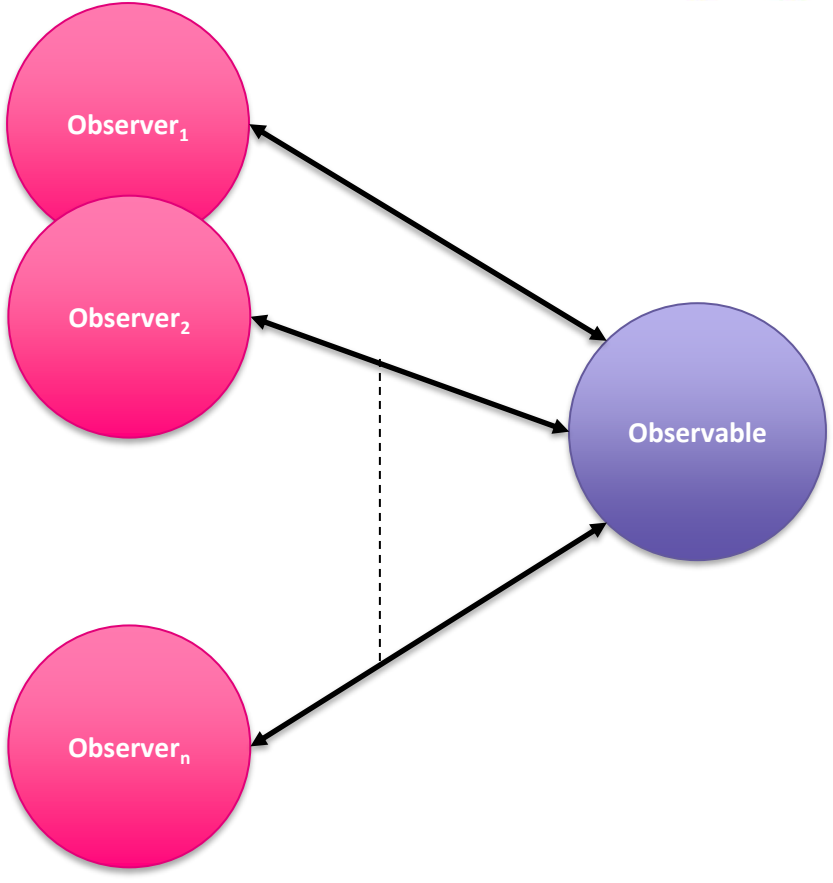
Let's look at the code (broadcaster package)

Broadcaster interesting executions

Assuming FIFO mailboxes
(Akka's default)

- Consider this execution

1. Observer1 sends Subscription to observable
2. Observer2 sends Subscription to observable
3. …



What actor will receive first SubscriptionOK?

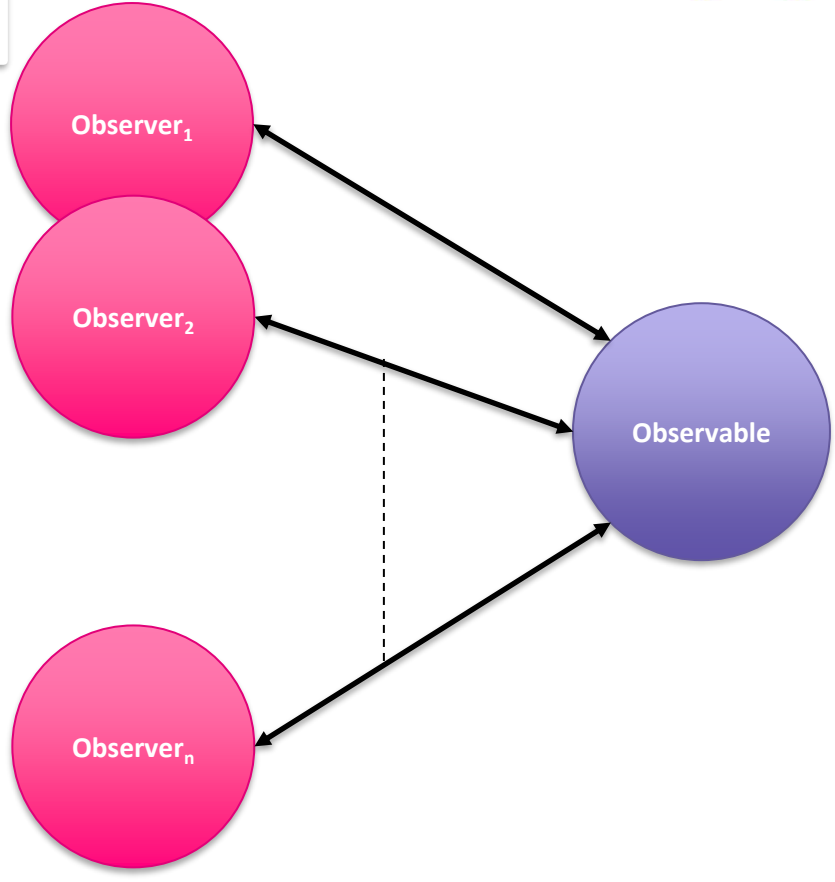
Broadcaster interesting executions

Assuming FIFO mailboxes (Akka's default)

- Consider this execution

1. Observer1 sends Subscription to observable
2. Observable replies SubscriptionOK to observer1
3. Observer1 emits message to observable
4. Observer2 sends Subscription to observable
5. …

Can observer2 receive the message sent by observer1 in step 3?

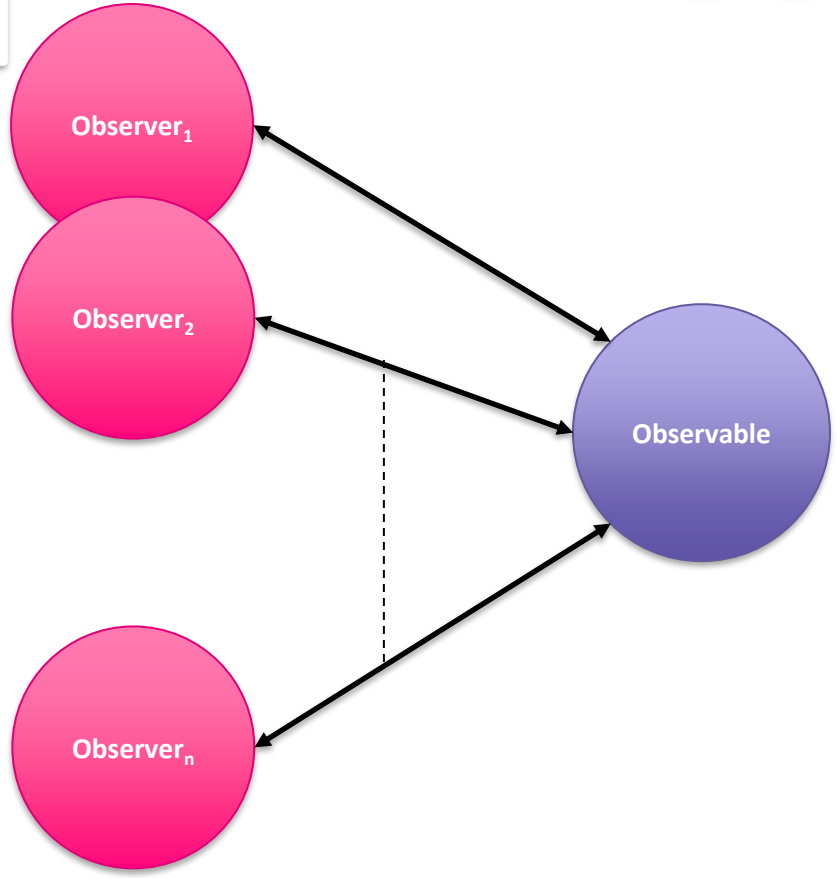


Broadcaster interesting executions

Assuming FIFO mailboxes (Akka's default)

- Consider this execution

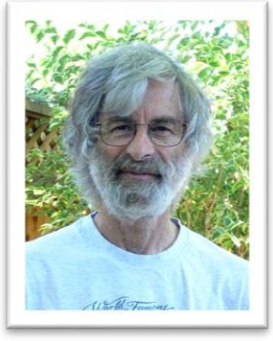
1. Observer1 send Subscription to observable
2. Observable replies SubscriptionOK to observer1
3. Observer1 emits message to observable
4. Observer2 sends Subscription to observable
5. Observable replies SubscriptionOK to observer2
6. …



Can observer2 receive the message sent by observer1 in step 3?

Happened-before in distributed systems

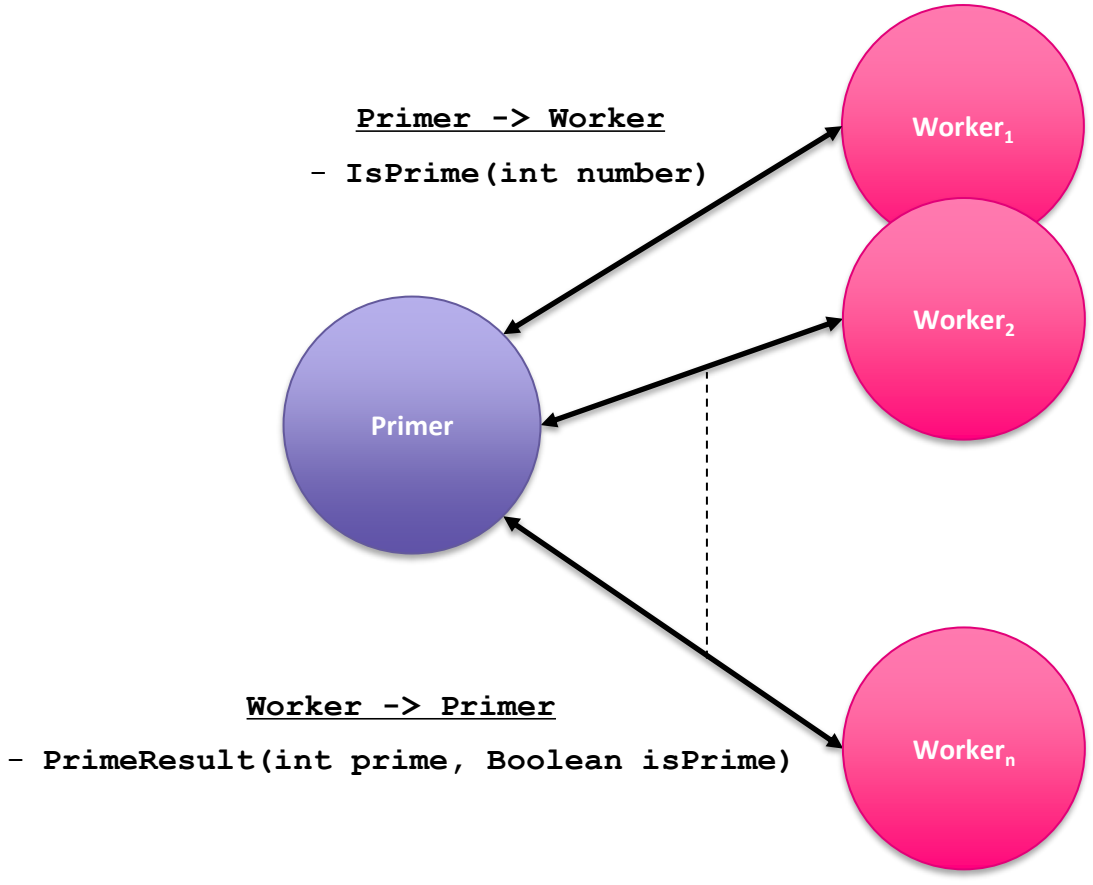
- Note that in the previous questions the behaviour of the systems depends on the reception of messages



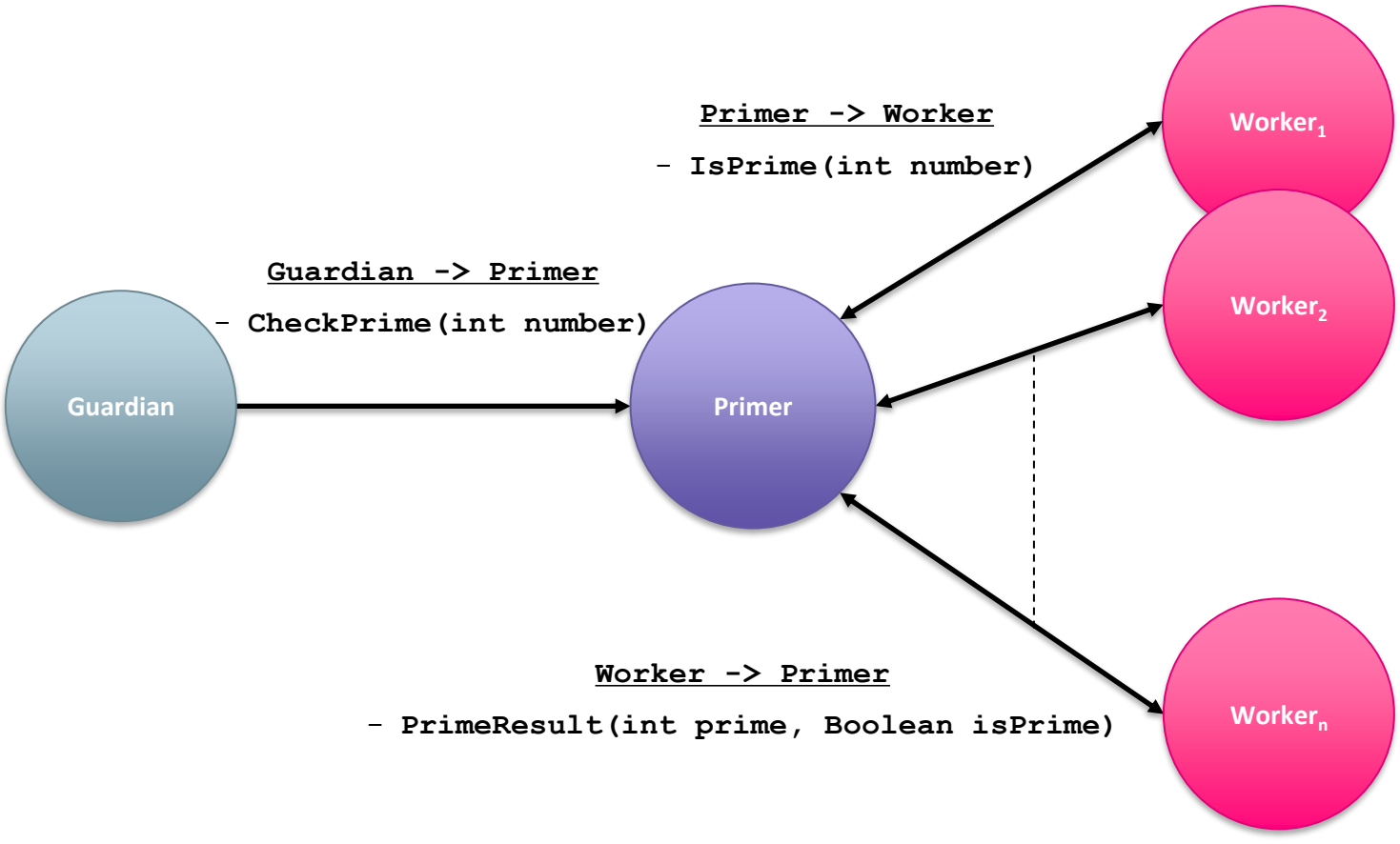
- Thus, the happened-before relation defined by Lamport is useful in reasoning about actor systems
  - An action *a* happens-before an action *b* if they belong to the same actor and *a* was executed before *b*
  - A send(m) action happens-before its corresponding receive(m)

- Note the similarity with the happens-before relation of the Java memory model
  - We reason about message exchange instead of locking (but inherent coordination problems remain)
  - Visibility issues disappear as actors only access local memory

Primer

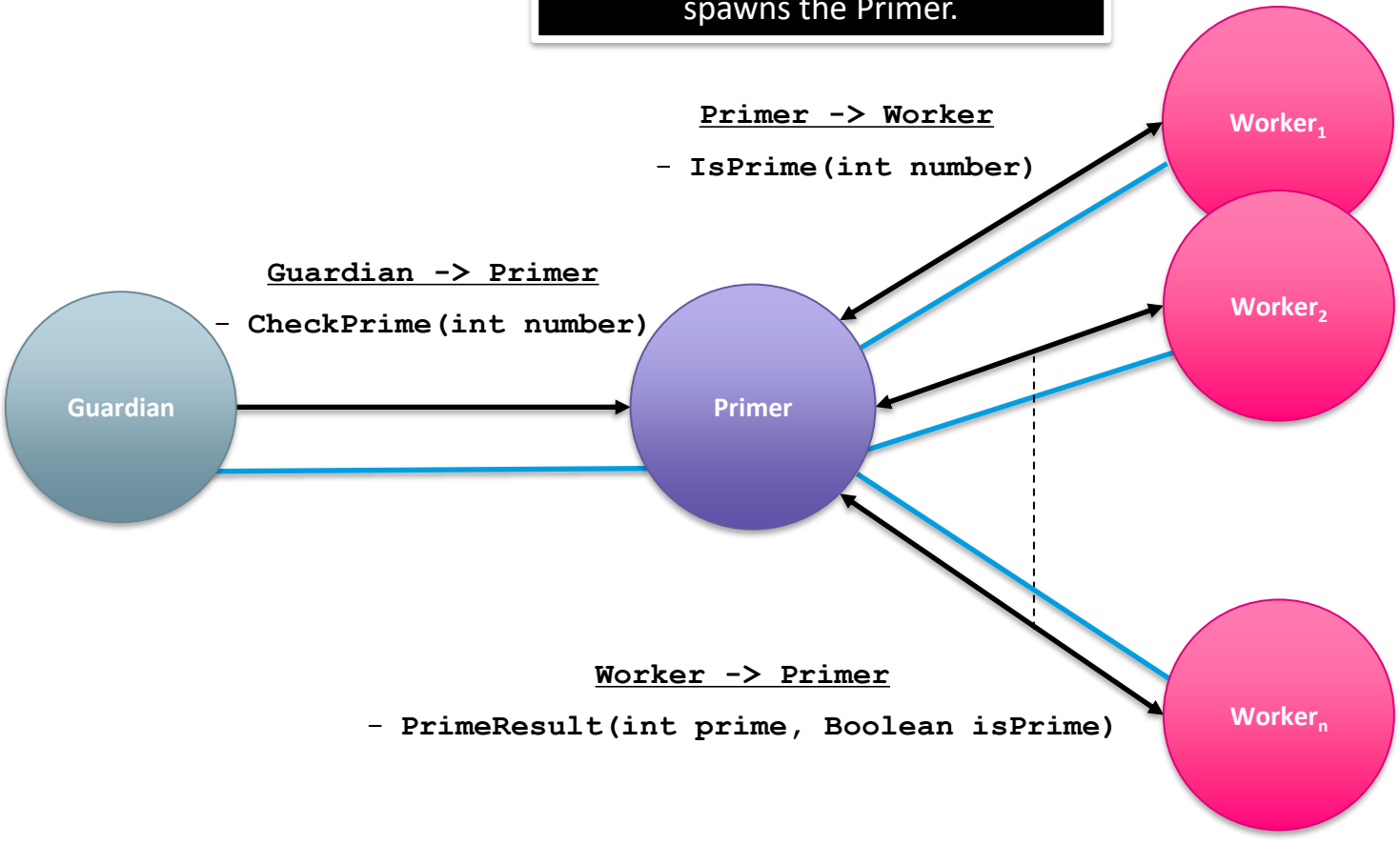


Primer



Primer

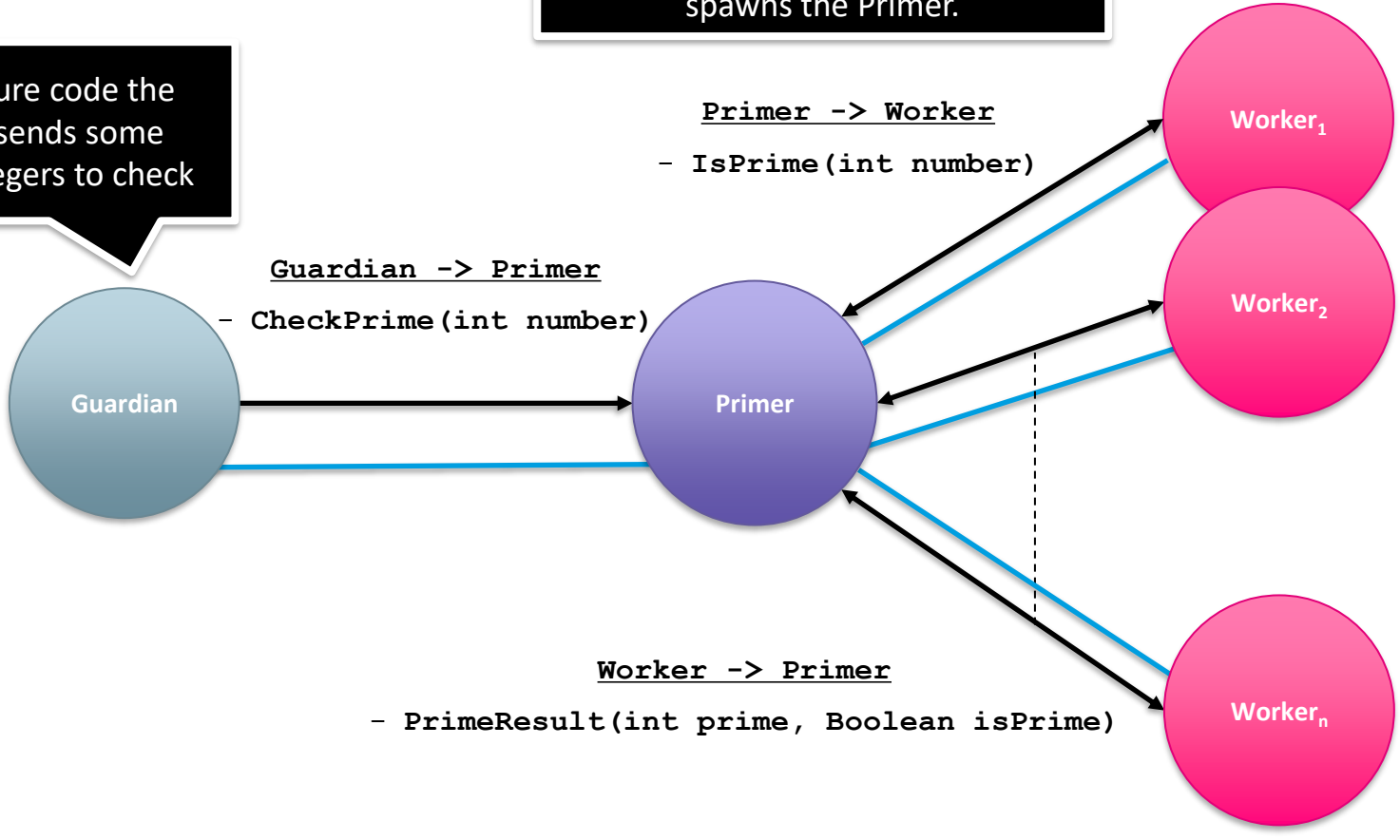
The Primer is in charge of spawning the workers. As usual, the guardian spawns the Primer.



Primer

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

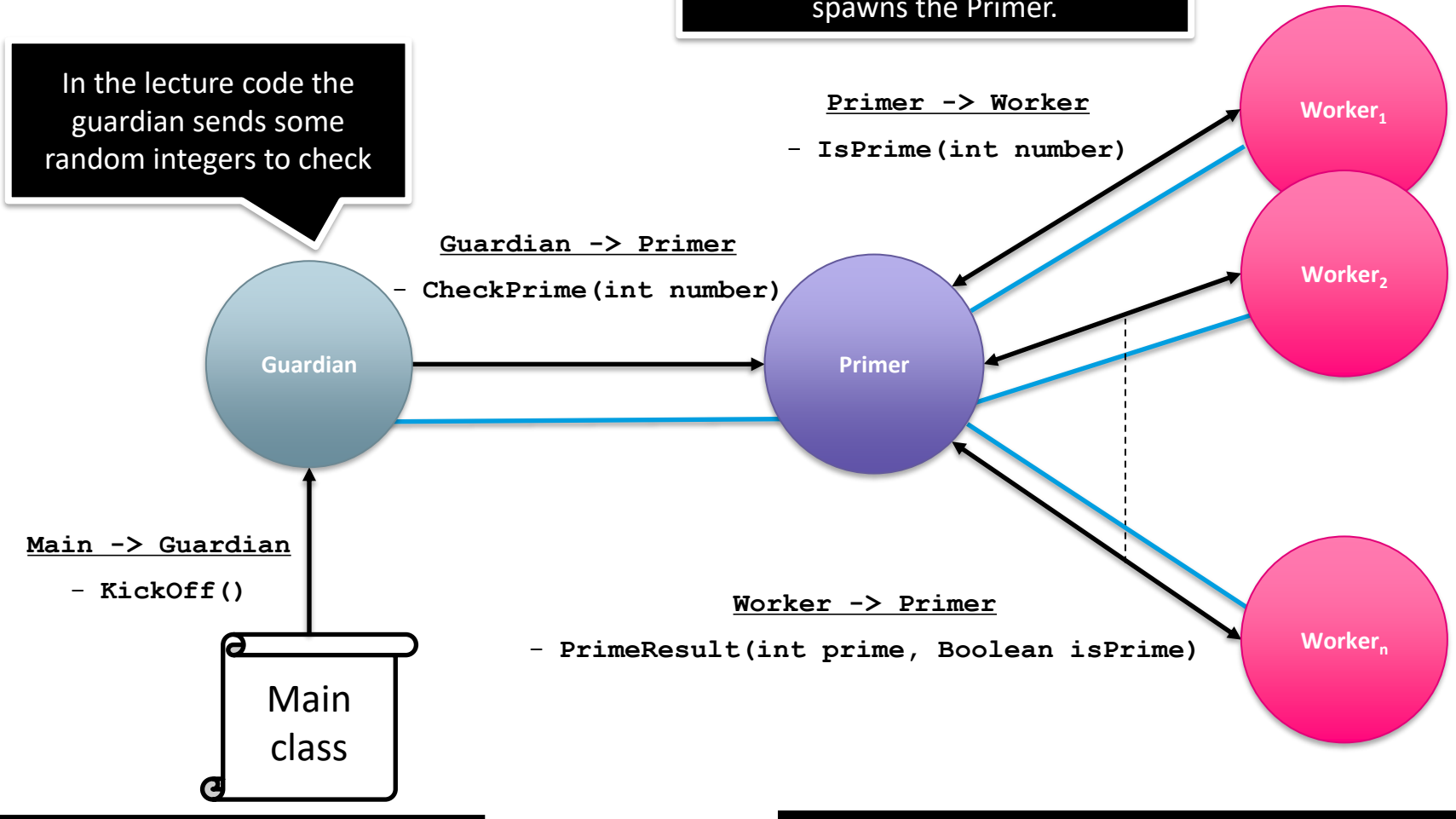
In the lecture code the guardian sends some random integers to check



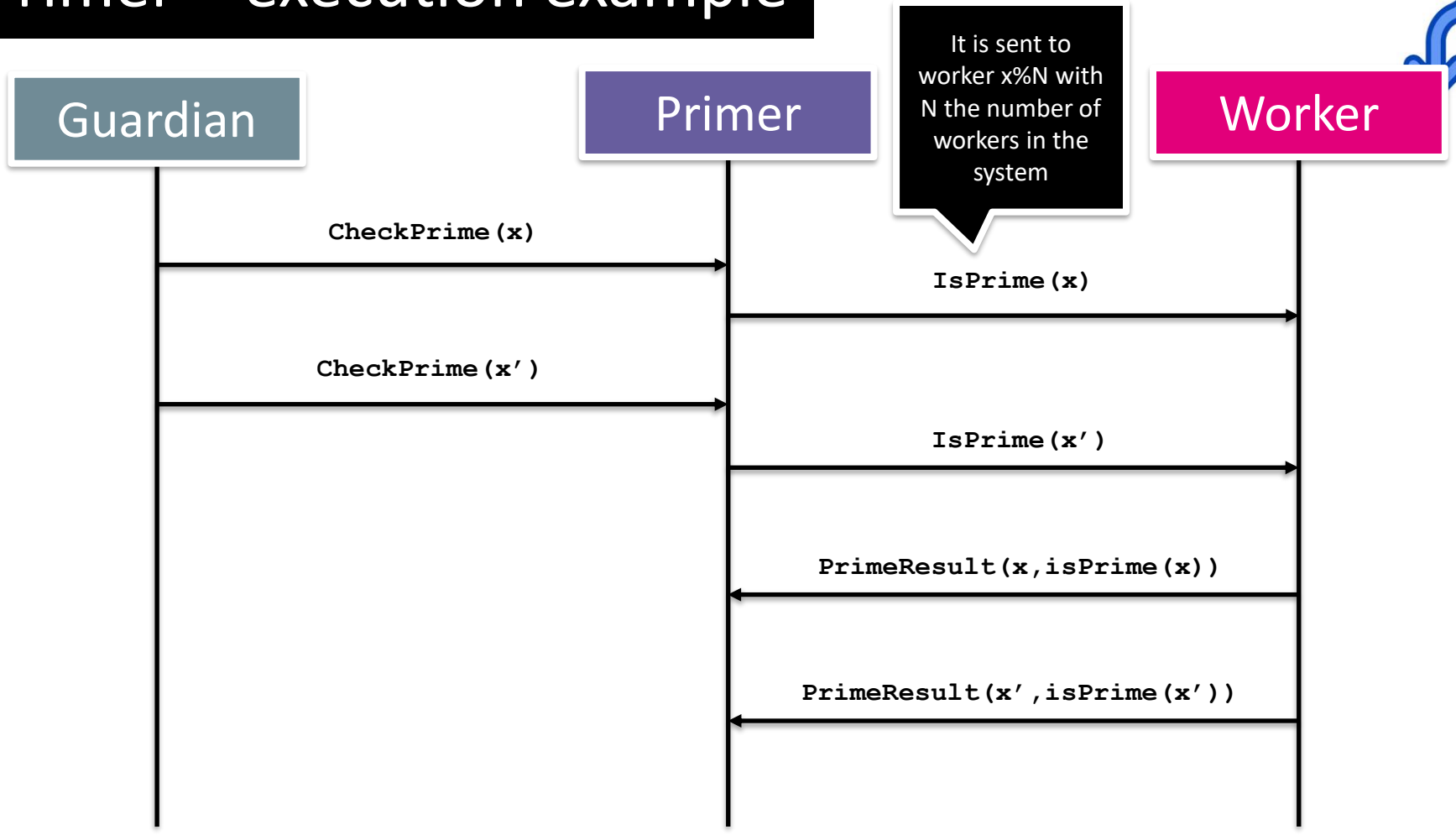
Primer

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

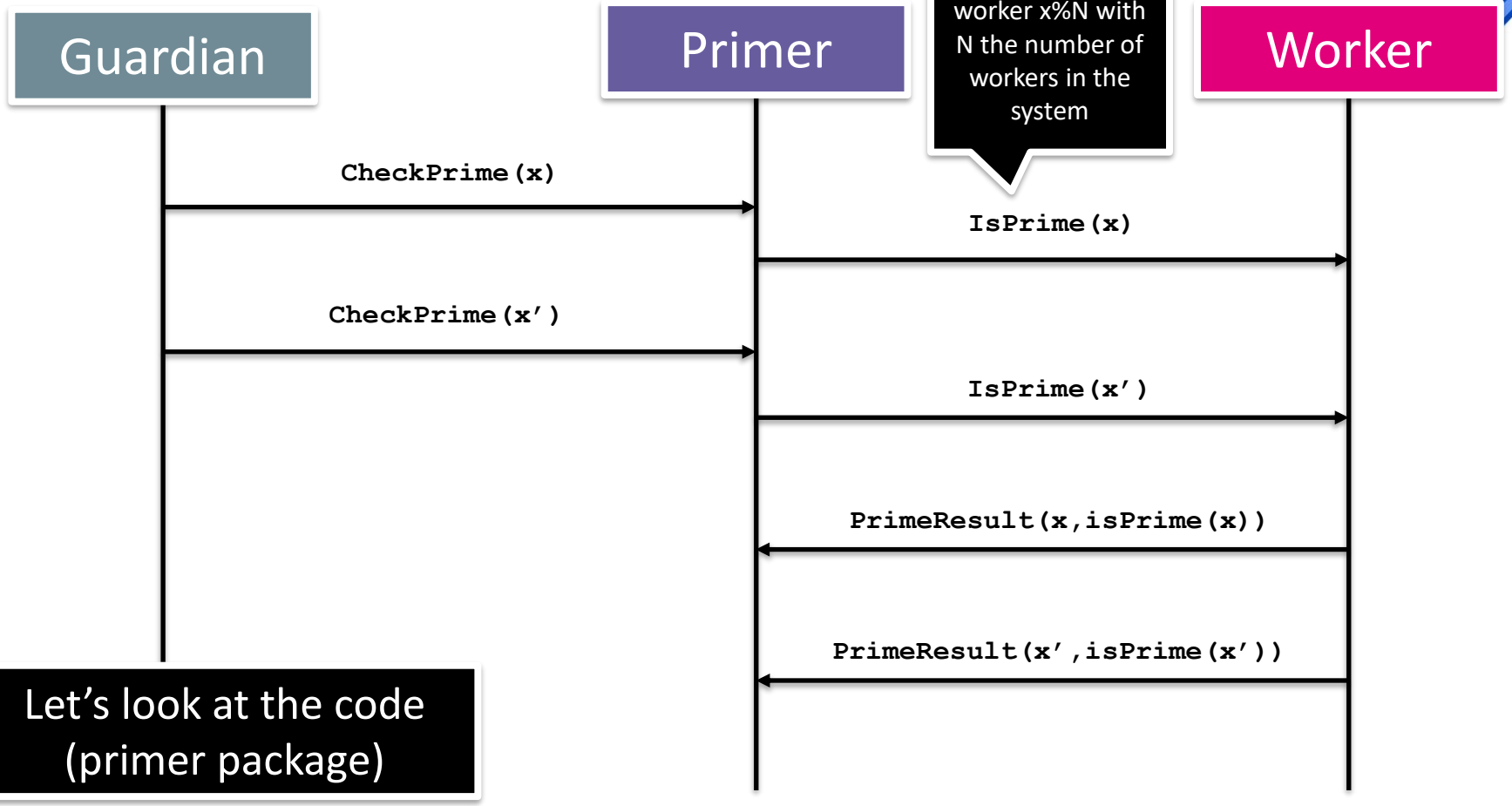
In the lecture code the guardian sends some random integers to check



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?

Agenda

- Problems in shared memory concurrency (revisited)
- Actors
- Akka
- Example systems
  - Printer
  - Broadcaster (observer/observable)
  - Primer