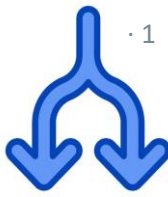- If you struggle with Gradle, let us know! In your message include:
  - OS (and version), what shell you use (if any), whether you use an IDE (and version)

- Try to become fluent in using Gradle before the exam
  - Examiners will use the produce explained in the guidelines

- Remember to write programs compatible with **JDK 8** and later
  - For instance, `var` is not compatible with JDK 8

- Code is not self-explanatory
  - For the exam, avoid answers that simply point to a piece of code. They should be accompanied by a textual explanation
  - This is not criticism, but a kind advice aiming to help you write better answers in the exam
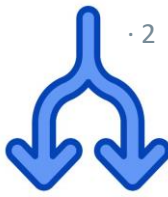
Practical Concurrent and Parallel Programming XI

Message Passing I

Raúl Pardo
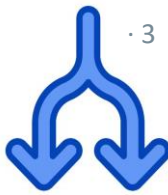
- Problems in shared memory concurrency (revisited)
- Actors
- Akka
- Example systems
  - Turnstile (counter)
  - Broadcaster
  - Bounded Buffer
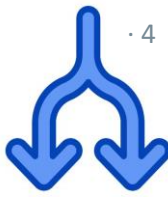
"Writing thread-safe code is, at its core, about
__managing access to shared mutable data__"

Goetz

*"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?

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

Goetz

- Race conditions
- Data races
- Visibility
- Reasoning is tricky
  - Specially lock-free computation 😁

*"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?

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

Goetz

- Happens-before reasoning
- For race conditions and data races:
  - Ensuring mutual exclusion
    - Locks (introduce the problem of deadlocks)
  - Immutability
  - Compare and Swap (CAS) algorithms
- For visibility:
  - Volatile and final variables, idioms for safe publication, etc

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

Goetz

- Happens-before reasoning
- For race conditions and data races:
  - Ensuring mutual exclusion
    - Locks (introduce the problem of deadlocks)
  - Immutability
  - Compare and Swap (CAS) algorithms
- For visibility:
  - Volatile and final variables, idioms for safe publication, etc

Why don´t we simply avoid sharing state?
This is the idea behind message passing!

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

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

Joe Armstrong
@joeerl

Following

Copying = good, sharing=bad

Hey @joeerl, do you think the inter-process communication should never be done by sharing memory? Otherwise, when it's okay?
Thanks a lot!
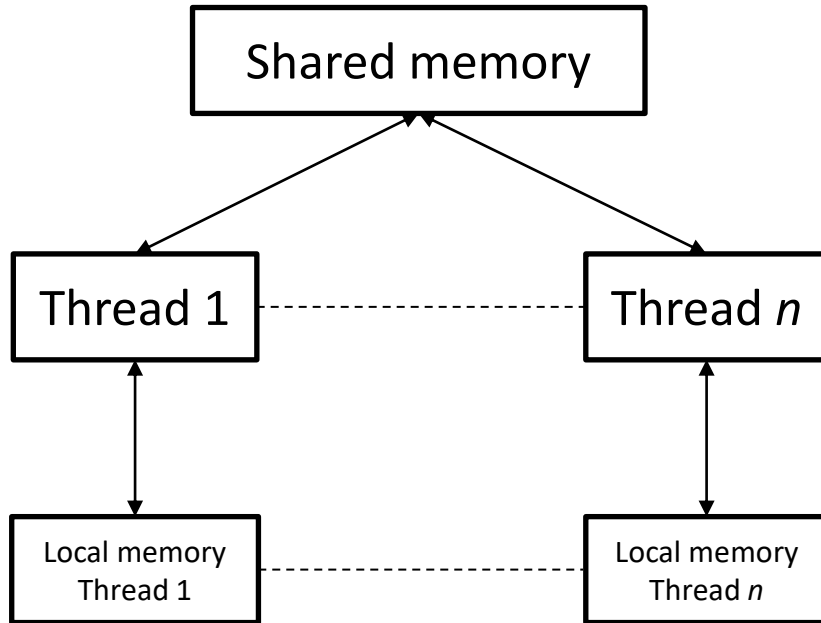
12:11 PM - 22 Nov 2018

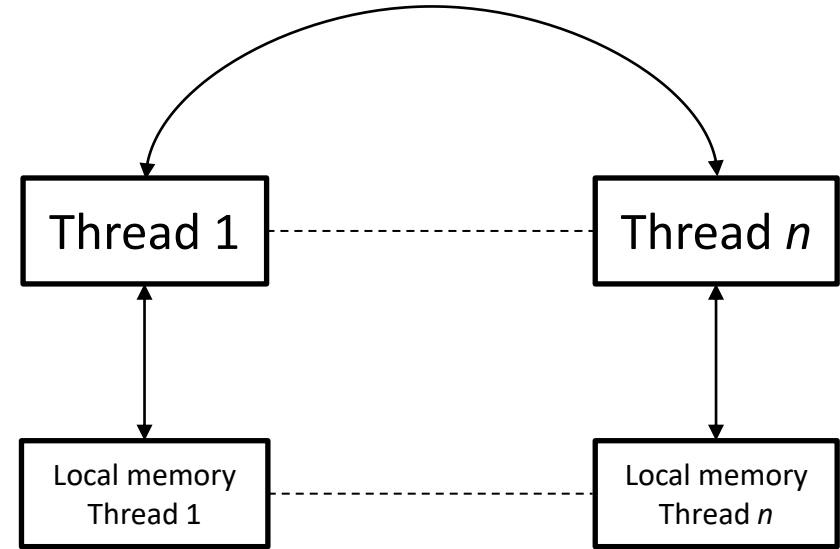11 Retweets 18 Likes

Shared memory vs Message Passing

- ## Shared Memory
  - Synchronisation by writing in shared memory

- ## Message Passing
  - Synchronisation by sending messages

PROBLEM:
Sharing && Mutability!

SOLUTIONS:
1) atomic access!
   locking or transactions
   NB: avoid deadlock!
2) avoid mutability!
3) avoid sharing...

Thread T1

OK
(unshared resource)

This year we would say
thread-safe

Atomic access:
Locking (pessimistic) or
Transactions (optimistic)

x=x+1 || x=x+1

DANGER
(shared
mutable
resource)

OK
(immutable)

OK
(atomic access)

This year we would say
lock-free

immutable
datastructures

OK
(unshared resource)

Thread T2

Slide from lecture 13 in PCPP 2019

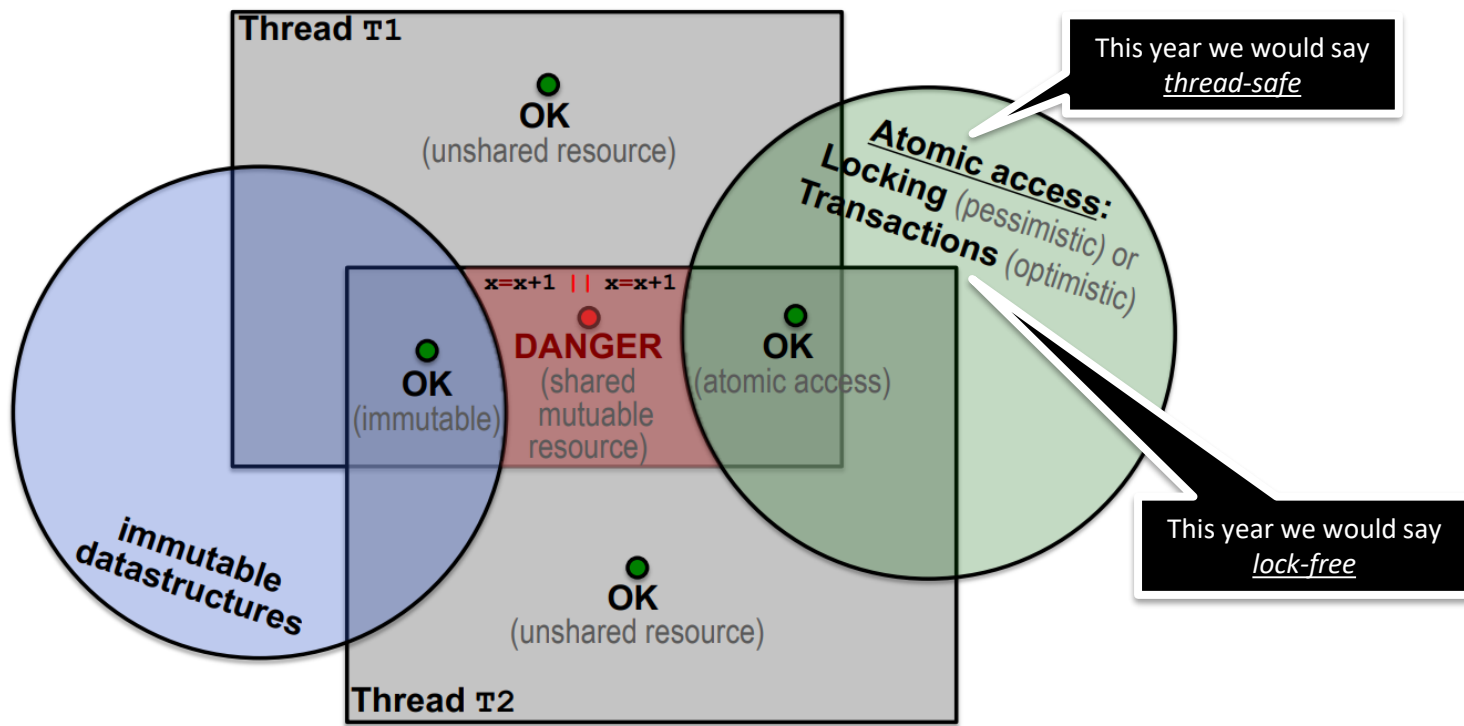© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

PROBLEM:
Sharing && Mutability!

SOLUTIONS:
1) atomic access!
   locking or transactions
   NB: avoid deadlock!
2) avoid mutability!
3) avoid sharing...

Thread T1

OK
(unshared resource)

message

This year we would say
thread-safe

Atomic access:
Locking (pessimistic) or
Transactions (optimistic)

immutable
datastructures

No sharing!

This year we would say
lock-free

OK
(unshared resource)

Thread T2

Slide from lecture 13 in PCPP 2019

IT UNIVERSITY OF COPENHAGEN

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

- 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

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022
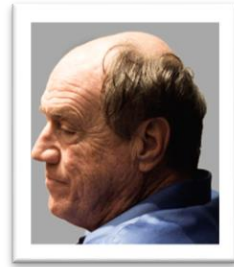
- An actor can be seen as a sequential unit of computation
  - Although, formally, the model allows for parallelism within the actor, one can safely assume that there are not concurrency issues within the actor.
  - You can think of an actor as a thread

- Actors can send messages to other actors
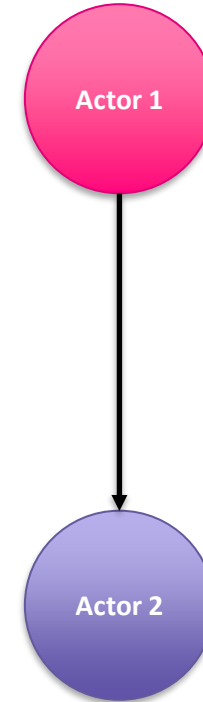
Actor 1

Actor 2

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

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

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

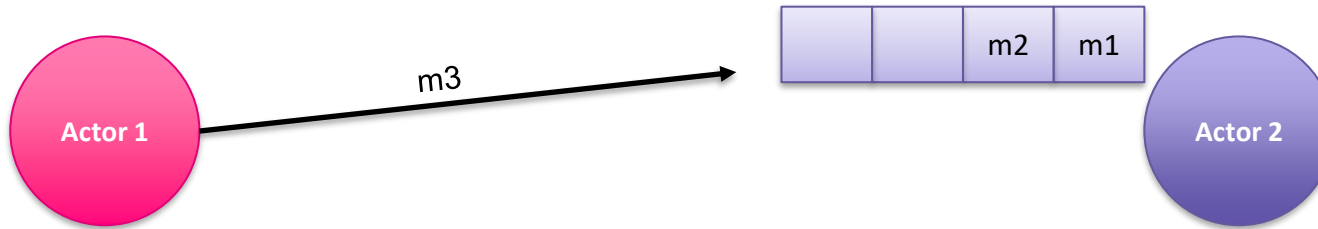|		m2	m1
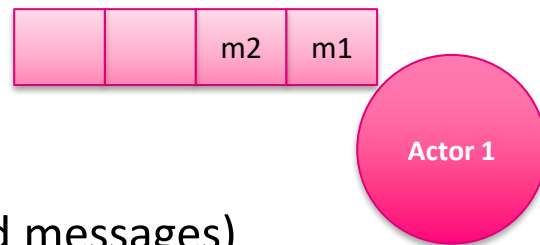
Actor 1

- Every actor in the system has a _unique identifier_
  - A.k.a. _mail address_ or _actor reference_

- 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 _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

- 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 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

m1

Actor 1

Actor 2

m2

		m2	m1

????

		m1	m2

- It is _not_ guaranteed that M1 arrives before M2
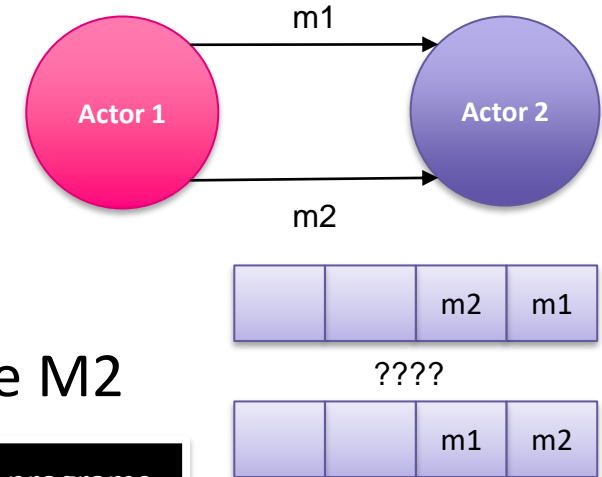
This is actually not true in Akka, but we will ignore that detail. Note that correct programs without this assumption will be correct if the assumption holds. But not viceversa.

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

Akka toolkit
(Actors implementation)

< 2.7   ≥ 2.7

*Akka is a ~~free and open-source~~ **source available 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** [...]*

[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.

Akka actors

Mailbox

		m2	m1

Actor 1

State
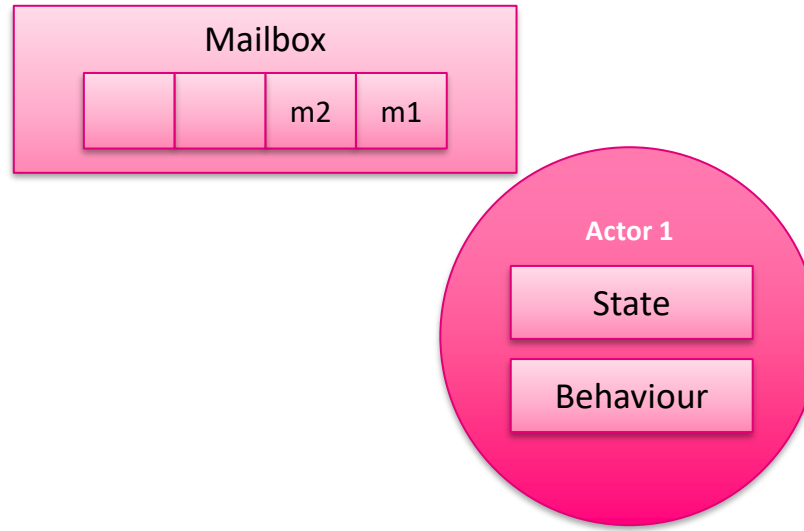
Behaviour

- Actors system to count the numbers of visitors in Tivoli

Type of messages that can be sent

- **Increment()**
- **PrintTotal()**

Turnstile₁

Counter

total

Models visitors crossing the turnstile

One way communication (solid directed arrows indicate communication)

Keeps track of the number of visitors

Dotted lines model an unbounded but finite sequence of elements

Turnstileₙ

```java
public class Counter extends AbstractBehavior<Counter.CounterCommand> {

    /* --- Messages ------------------------------------ */
    public interface CounterCommand {}
    public static final class Increment implements CounterCommand { }
    public static final class PrintTotal implements CounterCommand { }


    /* --- State --------------------------------------- */
    private int total;

    … // constructor missing (see next slides)

    /* --- Message handling ---------------------------- */
    @Override
    public Receive<CounterCommand> createReceive() {
        return newReceiveBuilder()
            .onMessage(Increment.class, this::onIncrement)
            .onMessage(PrintTotal.class, this::onPrintTotal)
            .build();
    }

    /* --- Handlers ------------------------------------ */
    public Behavior<CounterCommand> onIncrement(Increment msg) {
        this.getContext()
            .getLog()
            .info("A visitor arrived!");
        total++;
        return this;
    }

    public Behavior<CounterCommand> onPrintTotal(PrintTotal msg) {
        this.getContext()
            .getLog()
            .info("Total people in the park: {}", total);
        return this;
    }
}
```

- **Increment()**
- **PrintTotal()**

total

Counter

```java
public class Counter extends AbstractBehavior<Counter.CounterCommand> {

    /* --- Messages ------------------------------------ */
    public interface CounterCommand {}
    public static final class Increment implements CounterCommand { }
    public static final class PrintTotal implements CounterCommand { }


    /* --- State --------------------------------------- */
    private int total;

    … // constructor missing (see next slides)

    /* --- Message handling ---------------------------- */
    @Override
    public Receive<CounterCommand> createReceive() {
        return newReceiveBuilder()
            .onMessage(Increment.class, this::onIncrement)
            .onMessage(PrintTotal.class, this::onPrintTotal)
            .build();
    }

    /* --- Handlers ------------------------------------ */
    public Behavior<CounterCommand> onIncrement(Increment msg) {
        this.getContext()
            .getLog()
            .info("A visitor arrived!");
        total++;
        return this;
    }

    public Behavior<CounterCommand> onPrintTotal(PrintTotal msg) {
        this.getContext()
            .getLog()
            .info("Total people in the park: {}", total);
        return this;
    }
}
```
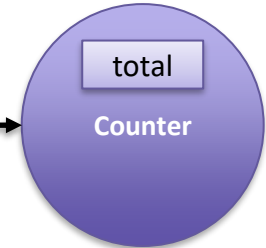
Actors vs Threads
- Like threads, Actors are defined in their own class

- **Increment()**
- **PrintTotal()**

total

Counter

- An actor class extends from an Akka AbstractBehavior
- Parameterized with the type of messages the actor handles (see next slide)

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

```java
public class Counter extends AbstractBehavior<Counter.CounterCommand> {

    /* --- Messages ------------------------------------- */
    public interface CounterCommand {}
    public static final class Increment implements CounterCommand { }
    public static final class PrintTotal implements CounterCommand { }


    /* --- State ---------------------------------------- */
    private int total;

    … // constructor missing (see next slides)

    /* --- Message handling ----------------------------- */
    @Override
    public Receive<CounterCommand> createReceive() {
        return newReceiveBuilder()
            .onMessage(Increment.class, this::onIncrement)
            .onMessage(PrintTotal.class, this::onPrintTotal)
            .build();
    }

    /* --- Handlers ------------------------------------- */
    public Behavior<CounterCommand> onIncrement(Increment msg) {
        this.getContext()
            .getLog()
            .info("A visitor arrived!");
        total++;
        return this;
    }

    public Behavior<CounterCommand> onPrintTotal(PrintTotal msg) {
        this.getContext()
            .getLog()
            .info("Total people in the park: {}", total);
        return this;
    }
}
```
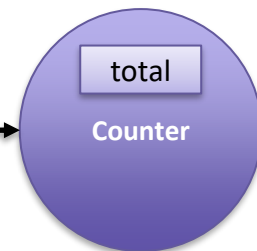
- **Increment()**
- **PrintTotal()**

total

Counter
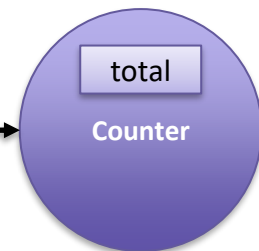
```java
public class Counter extends AbstractBehavior<Counter.CounterCommand> {

    /* --- Messages ------------------------------------- */
    public interface CounterCommand {}
    public static final class Increment implements CounterCommand { }
    public static final class PrintTotal implements CounterCommand { }


    /* --- State ---------------------------------------- */
    private int total;

    … // constructor missing (see next slides)

    /* --- Message handling ---------------------------- */
    @Override
    public Receive<CounterCommand> createReceive() {
        return newReceiveBuilder()
            .onMessage(Increment.class, this::onIncrement)
            .onMessage(PrintTotal.class, this::onPrintTotal)
            .build();
    }

    /* --- Handlers ------------------------------------ */
    public Behavior<CounterCommand> onIncrement(Increment msg) {
        this.getContext()
            .getLog()
            .info("A visitor arrived!");
        total++;
        return this;
    }

    public Behavior<CounterCommand> onPrintTotal(PrintTotal msg) {
        this.getContext()
            .getLog()
            .info("Total people in the park: {}", total);
        return this;
    }
}
```

It is a good practice to define the type of messages that the actors handles as inner classes

- **Increment()**
- **PrintTotal()**

total

Counter

If the actor handles more than one type of message, then define a top level interface that is implemented by each type of message

Message classes must be thread-safe. The recommended Akka practice is to define them as static and final; making them immutable.

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

```java
public class Counter extends AbstractBehavior<Counter.CounterCommand> {

    /* --- Messages ------------------------------------ */
    public interface CounterCommand {}
    public static final class Increment implements CounterCommand { }
    public static final class PrintTotal implements CounterCommand { }


    /* --- State --------------------------------------- */
    private int total;

    … // constructor missing (see next slides)

    /* --- Message handling ---------------------------- */
    @Override
    public Receive<CounterCommand> createReceive() {
        return newReceiveBuilder()
            .onMessage(Increment.class, this::onIncrement)
            .onMessage(PrintTotal.class, this::onPrintTotal)
            .build();
    }

    /* --- Handlers ------------------------------------ */
    public Behavior<CounterCommand> onIncrement(Increment msg) {
        this.getContext()
            .getLog()
            .info("A visitor arrived!");
        total++;
        return this;
    }

    public Behavior<CounterCommand> onPrintTotal(PrintTotal msg) {
        this.getContext()
            .getLog()
            .info("Total people in the park: {}", total);
        return this;
    }
}
```

Actors vs Threads
- Like threads, Actors' local state is defined as private fields

- **Increment()**
- **PrintTotal()**

total

Counter

```java
public class Counter extends AbstractBehavior<Counter.CounterCommand> {

    /* --- Messages -------------------------------------- */
    public interface CounterCommand {}
    public static final class Increment implements CounterCommand { }
    public static final class PrintTotal implements CounterCommand { }


    /* --- State ----------------------------------------- */
    private int total;

    … // constructor missing (see next slides)

    /* --- Message handling ------------------------------ */
    @Override
    public Receive<CounterCommand> createReceive() {
        return newReceiveBuilder()
            .onMessage(Increment.class, this::onIncrement)
            .onMessage(PrintTotal.class, this::onPrintTotal)
            .build();
    }

    /* --- Handlers -------------------------------------- */
    public Behavior<CounterCommand> onIncrement(Increment msg) {
        this.getContext()
            .getLog()
            .info("A visitor arrived!");
        total++;
        return this;
    }

    public Behavior<CounterCommand> onPrintTotal(PrintTotal msg) {
        this.getContext()
            .getLog()
            .info("Total people in the park: {}", total);
        return this;
    }
}
```
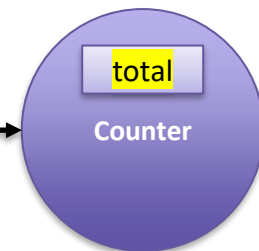
Actors vs Threads
- Like threads, Actors' local state is defined as private fields

– **Increment()**
– **PrintTotal()**

total

Counter

Are there visibility issues in the actor state?

```java
public class Counter extends AbstractBehavior<Counter.CounterCommand> {

    /* --- Messages ------------------------------------ */
    public interface CounterCommand {}
    public static final class Increment implements CounterCommand { }
    public static final class PrintTotal implements CounterCommand { }


    /* --- State --------------------------------------- */
    private int total;

    … // constructor missing (see next slides)

    /* --- Message handling ---------------------------- */
    @Override
    public Receive<CounterCommand> createReceive() {
        return newReceiveBuilder()
            .onMessage(Increment.class, this::onIncrement)
            .onMessage(PrintTotal.class, this::onPrintTotal)
            .build();
    }

    /* --- Handlers ------------------------------------ */
    public Behavior<CounterCommand> onIncrement(Increment msg) {
        this.getContext()
            .getLog()
            .info("A visitor arrived!");
        total++;
        return this;
    }

    public Behavior<CounterCommand> onPrintTotal(PrintTotal msg) {
        this.getContext()
            .getLog()
            .info("Total people in the park: {}", total);
        return this;
    }
}
```
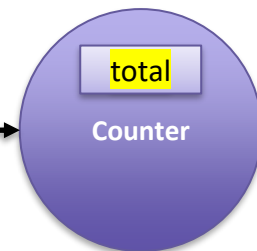
- **Increment()**
- **PrintTotal()**

total

Counter

Actors vs Threads
- Actors do not simply require implementing a `run()` method.
- Actors are "reactive", they act upon receiving a message
- This is implemented via message handlers

```java
public class Counter extends AbstractBehavior<Counter.CounterCommand> {

    /* --- Messages ------------------------------------ */
    public interface CounterCommand {}
    public static final class Increment implements CounterCommand { }
    public static final class PrintTotal implements CounterCommand { }


    /* --- State --------------------------------------- */
    private int total;

    … // constructor missing (see next slides)

    /* --- Message handling ---------------------------- */
    @Override
    public Receive<CounterCommand> createReceive() {
        return newReceiveBuilder()
                .onMessage(Increment.class, this::onIncrement)
                .onMessage(PrintTotal.class, this::onPrintTotal)
                .build();
    }

    /* --- Handlers ------------------------------------ */
    public Behavior<CounterCommand> onIncrement(Increment msg) {
        this.getContext()
            .getLog()
            .info("A visitor arrived!");
        total++;
        return this;
    }

    public Behavior<CounterCommand> onPrintTotal(PrintTotal msg) {
        this.getContext()
            .getLog()
            .info("Total people in the park: {}", total);
        return this;
    }
}
```
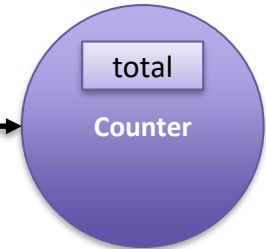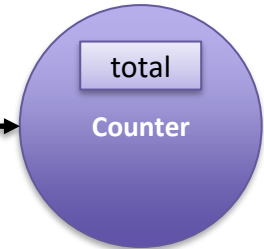
`Counter.java` in `turnstile` package

– **`Increment()`**
– **`PrintTotal()`**

total

Counter

Actors vs Threads
- Actors do not simply require implementing a `run()` method.
- Actors are "reactive", they act upon receiving a message
- This is implemented via message handlers

```java
public class Counter extends AbstractBehavior<Counter.CounterCommand> {

    /* --- Messages ------------------------------------ */
    public interface CounterCommand {}
    public static final class Increment implements CounterCommand { }
    public static final class PrintTotal implements CounterCommand { }


    /* --- State ------------------------------------ */
    private int total;

    … // constructor missing (see next slides)

    /* --- Message handling ------------------------- */
    @Override
    public Receive<CounterCommand> createReceive() {
        return newReceiveBuilder()
            .onMessage(Increment.class, this::onIncrement)
            .onMessage(PrintTotal.class, this::onPrintTotal)
            .build();
    }

    /* --- Handlers --------------------------------- */
    public Behavior<CounterCommand> onIncrement(Increment msg) {
        this.getContext()
            .getLog()
            .info("A visitor arrived!");
        total++;
        return this;
    }

    public Behavior<CounterCommand> onPrintTotal(PrintTotal msg) {
        this.getContext()
            .getLog()
            .info("Total people in the park: {}", total);
        return this;
    }
}
```
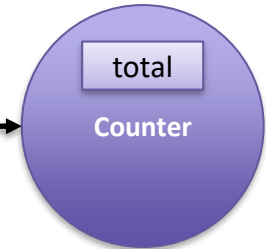
– `Increment()`
– `PrintTotal()`

total

Counter

Message handlers return the *behavior* of the actor after processing the message

In this lecture, we only consider actors that do not change behavior, i.e., they simply return this (the current bheavior)

```java
public class Turnstile extends AbstractBehavior<Turnstile.TurnstileCommand> {

    /* --- State -------------------------------------- */
    private final ActorRef<Counter.CounterCommand> countActor;

    /* --- Constructor -------------------------------- */
    private Turnstile(ActorContext<TurnstileCommand> context,
                      ActorRef<Counter.CounterCommand> countActor) {
        super(context);
        this.countActor = countActor;
    }

    /* --- Actor initial behavior --------------------- */
    public static Behavior<TurnstileCommand> create(ActorRef<Counter.CounterCommand> countActor) {
        return Behaviors.setup(context -> new Turnstile(context, countActor));
    }


    /* --- Message handling --------------------------- */
    @Override
    public Receive<TurnstileCommand> createReceive() {
        return newReceiveBuilder()
            .onMessage(Start.class, this::onStart)
            .build();
    }

    /* --- Handlers ----------------------------------- */
    private Behavior<TurnstileCommand> onStart(Start msg) {
        // send 20 increments to the counter
        IntStream.range(0,20)
            .forEach( i -> {
                    countActor.tell(new Counter.Increment());
                });
        countActor.tell(new Counter.PrintTotal());

        // continue with the same behavior
        return this;
    }
}
```
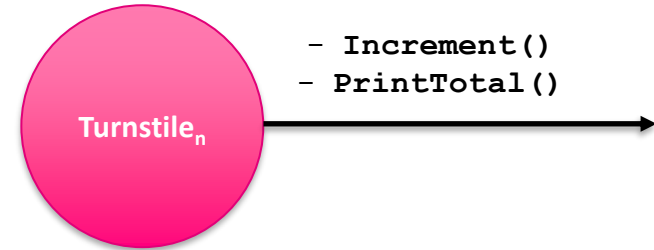
Turnstile$_n$

– **Increment()**
– **PrintTotal()**

```java
public class Turnstile extends AbstractBehavior<Turnstile.TurnstileCommand> {

    /* --- State --------------------------------------- */
    private final ActorRef<Counter.CounterCommand> countActor;

    /* --- Constructor --------------------------------- */
    private Turnstile(ActorContext<TurnstileCommand> context,
                      ActorRef<Counter.CounterCommand> countActor) {
        super(context);
        this.countActor = countActor;
    }

    /* --- Actor initial behavior ---------------------- */
    public static Behavior<TurnstileCommand> create(ActorRef<Counter.CounterCommand> countActor) {
        return Behaviors.setup(context -> new Turnstile(context, countActor));
    }


    /* --- Message handling ---------------------------- */
    @Override
    public Receive<TurnstileCommand> createReceive() {
        return newReceiveBuilder()
            .onMessage(Start.class, this::onStart)
            .build();
    }

    /* --- Handlers ------------------------------------ */
    private Behavior<TurnstileCommand> onStart(Start msg) {
        // send 20 increments to the counter
        IntStream.range(0,20)
            .forEach( i -> {
                    countActor.tell(new Counter.Increment());
                });
        countActor.tell(new Counter.PrintTotal());

        // continue with the same behavior
        return this;
    }
}
```
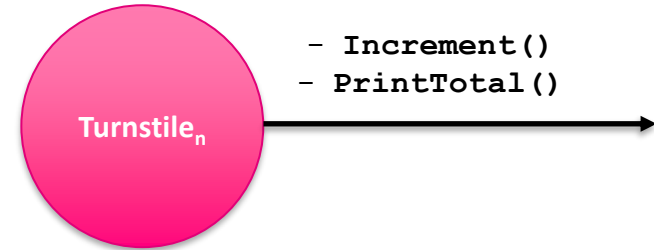
Turnstile_n

- **Increment()**
- **PrintTotal()**

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

Turnstile with Actors - Implementation

```java
public class Turnstile extends AbstractBehavior<Turnstile.TurnstileCommand> {

    /* --- State --------------------------------------- */
    private final ActorRef<Counter.CounterCommand> countActor;

    /* --- Constructor --------------------------------- */
    private Turnstile(ActorContext<TurnstileCommand> context,
                      ActorRef<Counter.CounterCommand> countActor) {
        super(context);
        this.countActor = countActor;
    }

    /* --- Actor initial behavior ---------------------- */
    public static Behavior<TurnstileCommand> create(ActorRef<Counter.CounterCommand> countActor) {
        return Behaviors.setup(context -> new Turnstile(context, countActor));
    }

    /* --- Message handling ---------------------------- */
    @Override
    public Receive<TurnstileCommand> createReceive() {
        return newReceiveBuilder()
            .onMessage(Start.class, this::onStart)
            .build();
    }

    /* --- Handlers ------------------------------------ */
    private Behavior<TurnstileCommand> onStart(Start msg) {
        // send 20 increments to the counter
        IntStream.range(0,20)
            .forEach( i -> {
                    countActor.tell(new Counter.Increment());
                });
        countActor.tell(new Counter.PrintTotal());

        // continue with the same behavior
        return this;
    }
}
```
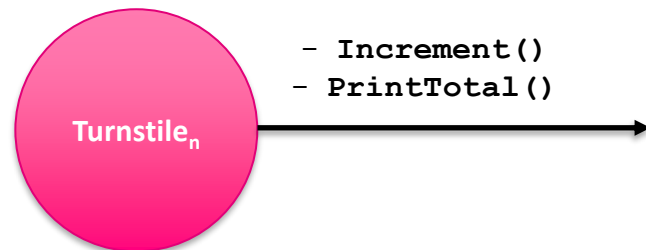
Turnstile_n

- **Increment()**
- **PrintTotal()**

Actors vs Threads
- Like threads, the initial state of the actor is defined via a constructor
- In Akka, the constructor must be defined as `private`; as it is never directly used for actor creation (see next slide)

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

```java
public class Turnstile extends AbstractBehavior<Turnstile.TurnstileCommand> {

    /* --- State -------------------------------------- */
    private final ActorRef<Counter.CounterCommand> countActor;

    /* --- Constructor -------------------------------- */
    private Turnstile(ActorContext<TurnstileCommand> context,
                      ActorRef<Counter.CounterCommand> countActor) {
        super(context);
        this.countActor = countActor;
    }

    /* --- Actor initial behavior --------------------- */
    public static Behavior<TurnstileCommand> create(ActorRef<Counter.CounterCommand> countActor) {
        return Behaviors.setup(context -> new Turnstile(context, countActor));
    }

    /* --- Message handling --------------------------- */
    @Override
    public Receive<TurnstileCommand> createReceive() {
        return newReceiveBuilder()
            .onMessage(Start.class, this::onStart)
            .build();
    }

    /* --- Handlers ----------------------------------- */
    private Behavior<TurnstileCommand> onStart(Start msg) {
        // send 20 increments to the counter
        IntStream.range(0,20)
            .forEach( i -> {
                    countActor.tell(new Counter.Increment());
                });
        countActor.tell(new Counter.PrintTotal());

        // continue with the same behavior
        return this;
    }
}
```

Turnstile$_n$

‒ **Increment()**
‒ **PrintTotal()**

- Actors are created via an *initial behavior*
- The initial behavior is defined as a create method
- The create method uses the private constructor
- Behaviors.setup initializes the actor in an *actor context* (the context contains information about the actor system)

```java
public class Turnstile extends AbstractBehavior<Turnstile.TurnstileCommand> {

    /* --- State --------------------------------------- */
    private final ActorRef<Counter.CounterCommand> countActor;

    /* --- Constructor --------------------------------- */
    private Turnstile(ActorContext<TurnstileCommand> context,
                      ActorRef<Counter.CounterCommand> countActor) {
        super(context);
        this.countActor = countActor;
    }

    /* --- Actor initial behavior ---------------------- */
    public static Behavior<TurnstileCommand> create(ActorRef<Counter.CounterCommand> countActor) {
        return Behaviors.setup(context -> new Turnstile(context, countActor));
    }


    /* --- Message handling ---------------------------- */
    @Override
    public Receive<TurnstileCommand> createReceive() {
        return newReceiveBuilder()
            .onMessage(Start.class, this::onStart)
            .build();
    }

    /* --- Handlers ------------------------------------ */
    private Behavior<TurnstileCommand> onStart(Start msg) {
        // send 20 increments to the counter
        IntStream.range(0,20)
            .forEach( i -> {
                    countActor.tell(new Counter.Increment());
                });
        countActor.tell(new Counter.PrintTotal());

        // continue with the same behavior
        return this;
    }
}
```
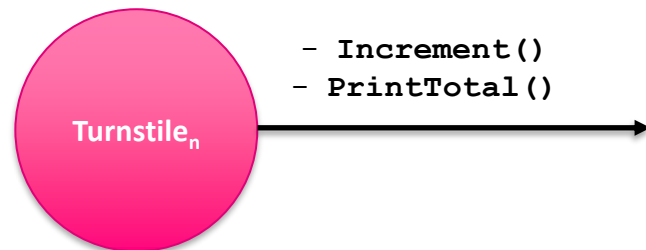
Turnstile$_n$

- **Increment()**
- **PrintTotal()**

- To send asynchronous messages, we call the `tell(…)` method on a reference to the actor
- The tell method takes as parameter an object of the type of messages that the actor can process

Turnstile with Actors - Implementation

```java
public class Turnstile extends AbstractBehavior<Turnstile.TurnstileCommand> {

    /* --- State -------------------------------------- */
    private final ActorRef<Counter.CounterCommand> countActor;

    /* --- Constructor -------------------------------- */
    private Turnstile(ActorContext<TurnstileCommand> context,
                      ActorRef<Counter.CounterCommand> countActor) {
        super(context);
        this.countActor = countActor;
    }

    /* --- Actor initial behavior --------------------- */
    public static Behavior<TurnstileCommand> create(ActorRef<Counter.CounterCommand> countActor) {
        return Behaviors.setup(context -> new Turnstile(context, countActor));
    }


    /* --- Message handling --------------------------- */
    @Override
    public Receive<TurnstileCommand> createReceive() {
        return newReceiveBuilder()
            .onMessage(Start.class, this::onStart)
            .build();
    }

    /* --- Handlers ----------------------------------- */
    private Behavior<TurnstileCommand> onStart(Start msg) {
        // send 20 increments to the counter
        IntStream.range(0,20)
            .forEach( i -> {
                    countActor.tell(new Counter.Increment());
                });
        countActor.tell(new Counter.PrintTotal());

        // continue with the same behavior
        return this;
    }
}
```

Turnstile.java in turnstile package

Turnstile$_n$

- **Increment()**
- **PrintTotal()**

- To send asynchronous messages, we call the `tell(…)` method on a reference to the actor
- The tell method takes as parameter an object of the type of messages that the actor can process

- 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

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

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


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


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


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


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


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

- There is a one-to-one correspondence of the basic actor operations and the Akka API

Actors Model	Akka
Actor	Actor class (AbstractBehaviour)
Mailbox Address	Reference to Actor class
Message	Message static final class
State	Actor class local attributes
Behaviour	Handler functions in the Actor class
Create actor	API function
Send message	API function
Receive message	Message handler builder (from API)

- There is a one-to-one correspondence of the basic actor operations and the Akka API

Actors Model	Akka
Actor	Actor class (AbstractBehaviour)
Mailbox Address	Reference to Actor class
Message	Message static final class
State	Actor class local attributes
Behaviour	Handler functions in the Actor class
Create actor	API function
Send message	API function
Receive message	Message handler builder (from API)

This is what you would expect

```
public class Main {

    public static void main(String[] args) {
        Actor a1 = new Actor();
        Actor a2 = new Actor();
        a1.start();
        a2.start();

    }
}
```
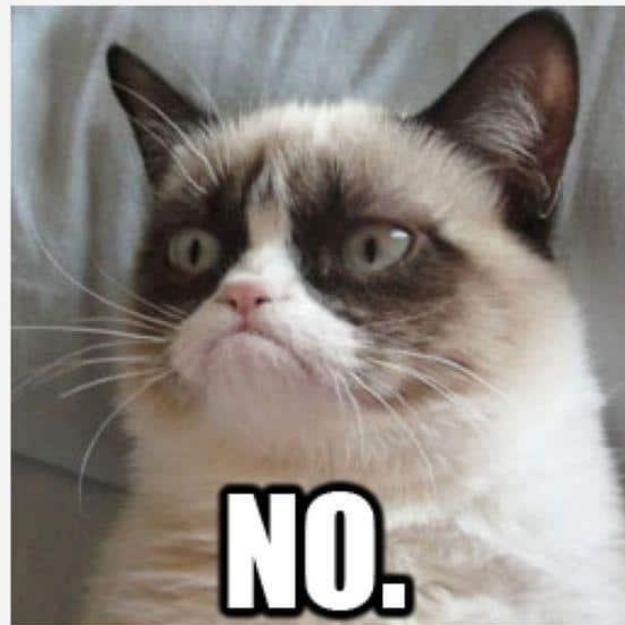
This is what you would expect

```
public cl      in {

    public            main(String[] args) {
            Ac        ew Actor();
            Act        new Actor();
            a
                rt

    }

}
```

NO.

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

total

Counter

```
public class MainNG {

    public static void main(String[] args) {
        // start the counter actor
        ActorSystem<Counter.CounterCommand> counter = ActorSystem.create(Counter.create(), "counter_actor");

        …
    }
}
```

This line creates an initial counter actor

How do we start Actors?

total

Counter

```
public class MainNG {

    public static void main(String[] args) {
        // start the counter actor
        ActorSystem<Counter.CounterCommand> counter = ActorSystem.create(Counter.create(), "counter_actor");



        // simulate 5 people entering the park
        IntStream.range(0,5)
            .forEach(i -> {
                counter.tell(new Counter.Increment());
            });
        counter.tell(new Counter.PrintTotal());


        …
    }
}
```
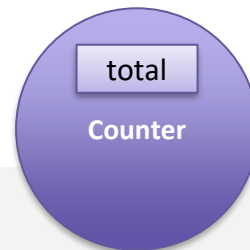
This line creates an initial counter actor

We can send messages to the counter actor

MainNG.java in turnstile package

IT UNIVERSITY OF COPENHAGEN

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

total

Counter

```java
public class MainNG {

    public static void main(String[] args) {
        // start the counter actor
        ActorSystem<Counter.CounterCommand> counter = ActorSystem.create(Counter.create(), "counter_actor");


        // simulate 5 people entering the park
        IntStream.range(0,5)
            .forEach(i -> {
                counter.tell(new Counter.Increment());
            });
        counter.tell(new Counter.PrintTotal());


        …
    }
}
```
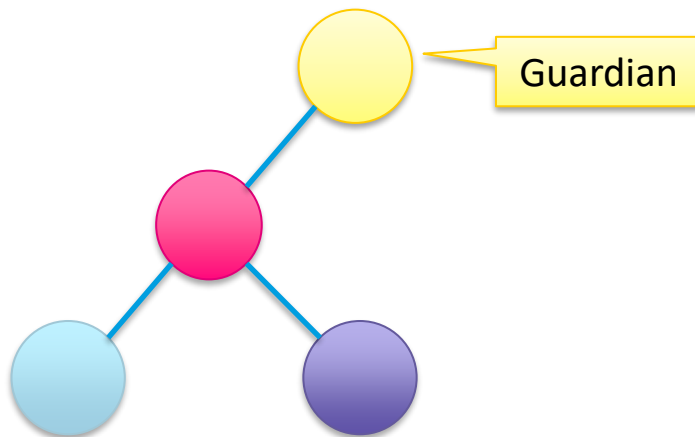
This line creates an initial counter actor

We can send messages to the counter actor

Unfortunately, ActorSystem.create can only be used to create one actor. What about the others?

MainNG.java in turnstile package

Akka actors – Guardian

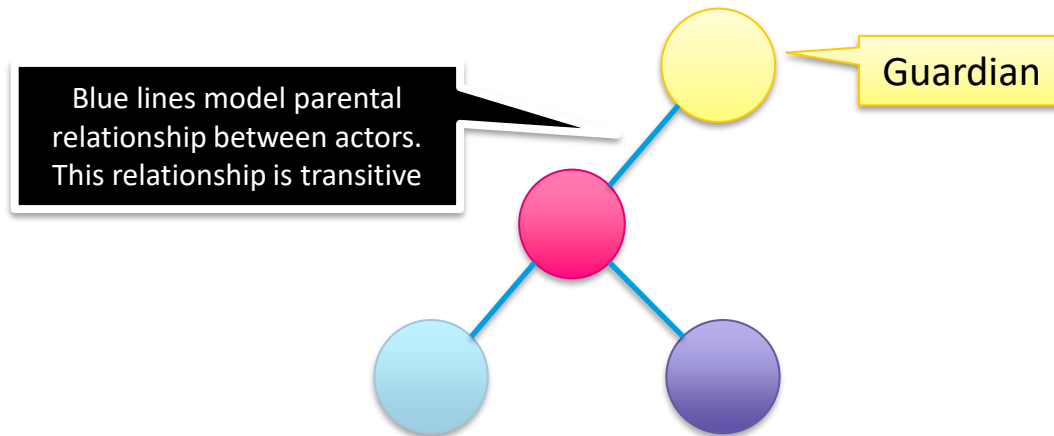- Akka actor systems have an implicit hierarchical structure

Guardian

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

```
ActorSystem<Counter.CounterCommand> counter = ActorSystem.create(Counter.create(), "counter_actor")
```

In our example, we use counter as the
guardian, but this is not idiomatic

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

- Akka actor systems have an implicit hierarchical structure

Guardian

Blue lines model parental relationship between actors. This relationship is transitive

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

```
ActorSystem<Counter.CounterCommand> counter = ActorSystem.create(Counter.create(), "counter_actor")
```

In our example, we use counter as the guardian, but this is not idiomatic

- Typically the Guardian creates the initial actors in the system

```java
public class Guardian extends AbstractBehavior<Guardian.KickOff> {
    public static final class KickOff { }

    private Guardian(ActorContext<KickOff> context) {
        super(context);
    }

    public static Behavior<Guardian.KickOff> create() {
        return Behaviors.setup(Guardian::new);
    }

    @Override
    public Receive<KickOff> createReceive() {
        return newReceiveBuilder()
            .onMessage(KickOff.class, this::onKickOff)
            .build();
    }

    private Behavior<KickOff> onKickOff(KickOff msg) {
        // spawn the counter actor
        ActorRef<Counter.CounterCommand> counter =
            getContext().spawn(Counter.create(), "counter_actor");

        // spawn two turnstile actors
        ActorRef<Turnstile.TurnstileCommand> t1 =
            getContext().spawn(Turnstile.create(counter), "t1");
        t1.tell(new Turnstile.Start());

        ActorRef<Turnstile.TurnstileCommand> t2 =
            getContext().spawn(Turnstile.create(counter), "t2");
        t2.tell(new Turnstile.Start());

        // The behaviour stays the same
        return this;
    }
}
```
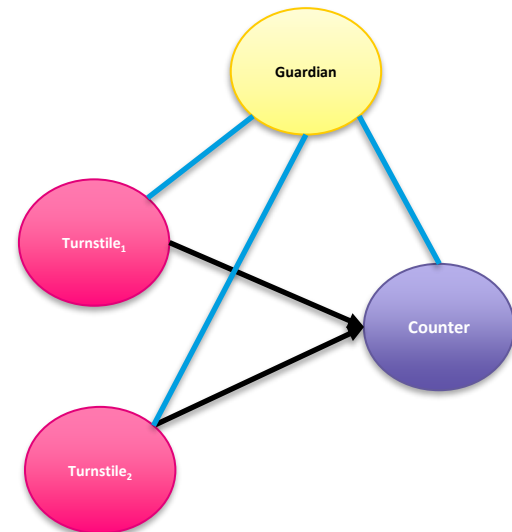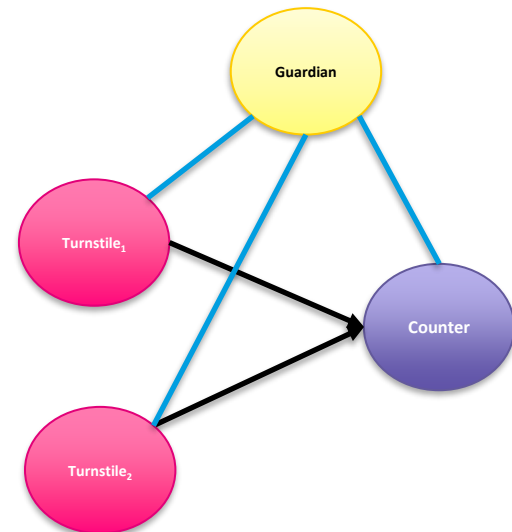
- The Guardian is an actor like any other
- It typically receives a kick-off messages that indicates to start the system

- Typically the Guardian creates the initial actors in the system

```java
public class Guardian extends AbstractBehavior<Guardian.KickOff> {
    public static final class KickOff { }

    private Guardian(ActorContext<KickOff> context) {
        super(context);
    }

    public static Behavior<Guardian.KickOff> create() {
        return Behaviors.setup(Guardian::new);
    }

    @Override
    public Receive<KickOff> createReceive() {
        return newReceiveBuilder()
            .onMessage(KickOff.class, this::onKickOff)
            .build();
    }

    private Behavior<KickOff> onKickOff(KickOff msg) {
        // spawn the counter actor
        ActorRef<Counter.CounterCommand> counter =
            getContext().spawn(Counter.create(), "counter_actor");

        // spawn two turnstile actors
        ActorRef<Turnstile.TurnstileCommand> t1 =
            getContext().spawn(Turnstile.create(counter), "t1");
        t1.tell(new Turnstile.Start());

        ActorRef<Turnstile.TurnstileCommand> t2 =
            getContext().spawn(Turnstile.create(counter), "t2");
        t2.tell(new Turnstile.Start());

        // The behaviour stays the same
        return this;
    }
}
```


- Children actors are created with `spawn()`
- The code on the right creates the counter actor and two turnstile actors

Akka Main + Guardian

```java
public class Main {

    public static void main(String[] args) {
        // actor system
        final ActorSystem<Guardian.KickOff> guardian =
            ActorSystem.create(Guardian.create(), "counter_akka");

        // trigger message
        guardian.tell(new Guardian.KickOff());

        // wait until user presses enter
        try {
            System.out.println(">>> Press ENTER to exit <<<");
            System.in.read();
        }
        catch (IOException e) {
            System.out.println("Error " + e.getMessage());
            e.printStackTrace();
        } finally {
            guardian.terminate();
        }
    }
}
```

Main.java in turnstile package

- This is a template that you can use to start any actor system in Akka.
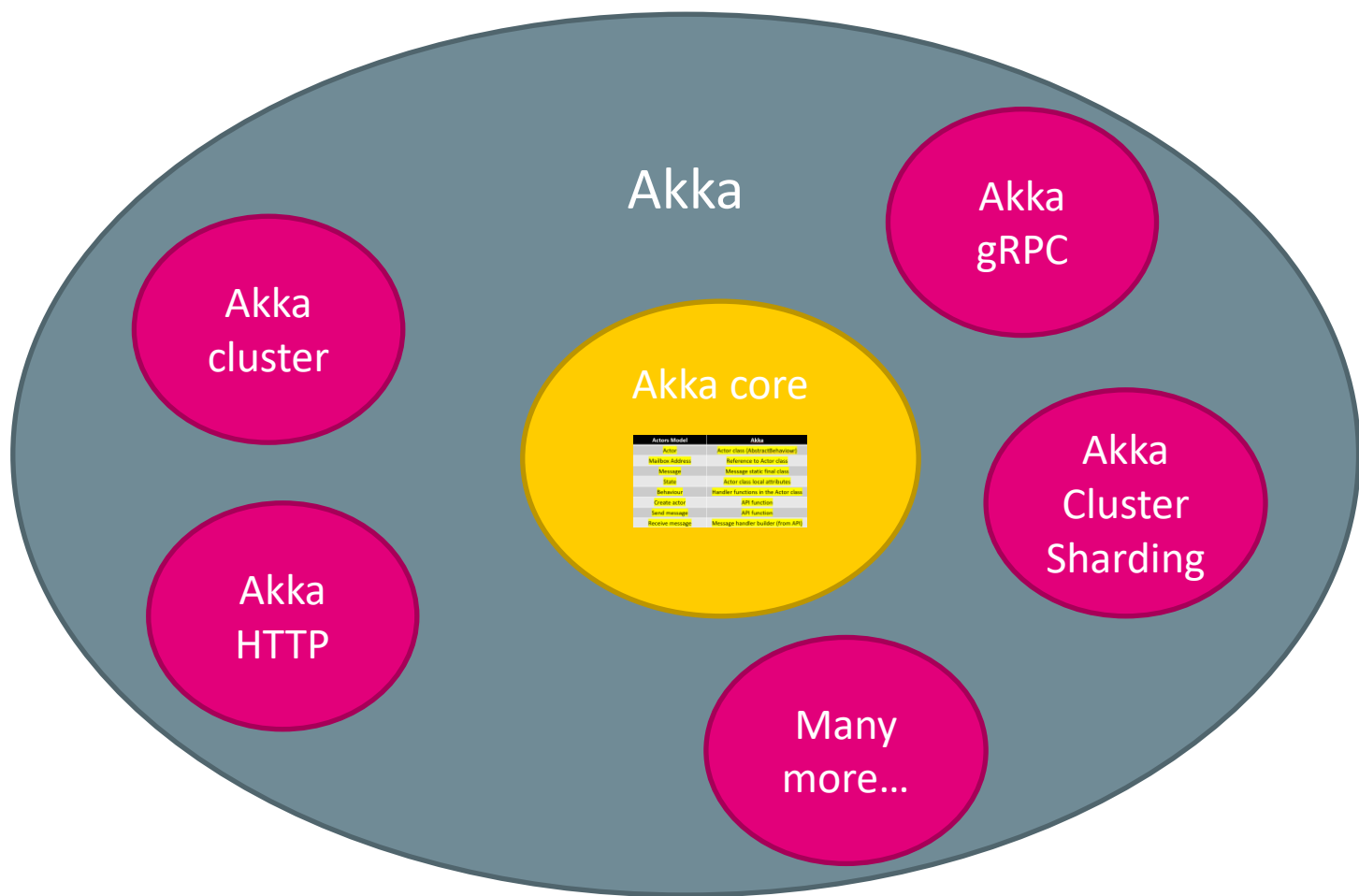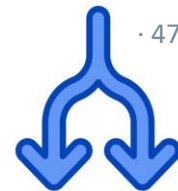- Simply replace the content of the `onKickOff()` method on the right to spawn the desired actors

```java
public class Guardian extends AbstractBehavior<Guardian.KickOff> {
    public static final class KickOff { }

    private Guardian(ActorContext<KickOff> context) {
        super(context);
    }

    public static Behavior<Guardian.KickOff> create() {
        return Behaviors.setup(Guardian::new);
    }

    @Override
    public Receive<KickOff> createReceive() {
        return newReceiveBuilder()
            .onMessage(KickOff.class, this::onKickOff)
            .build();
    }

    private Behavior<KickOff> onKickOff(KickOff msg) {
        // spawn the counter actor
        ActorRef<Counter.CounterCommand> counter =
            getContext().spawn(Counter.create(), "counter_actor");

        // spawn two turnstile actors
        ActorRef<Turnstile.TurnstileCommand> t1 =
            getContext().spawn(Turnstile.create(counter), "t1");
        t1.tell(new Turnstile.Start());

        ActorRef<Turnstile.TurnstileCommand> t2 =
            getContext().spawn(Turnstile.create(counter), "t2");
        t2.tell(new Turnstile.Start());

        // The behaviour stays the same
        return this;
    }
}
```

Guardian.java in turnstile package

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

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
Send message	API function
Receive message	Message handler builder (from API)

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022
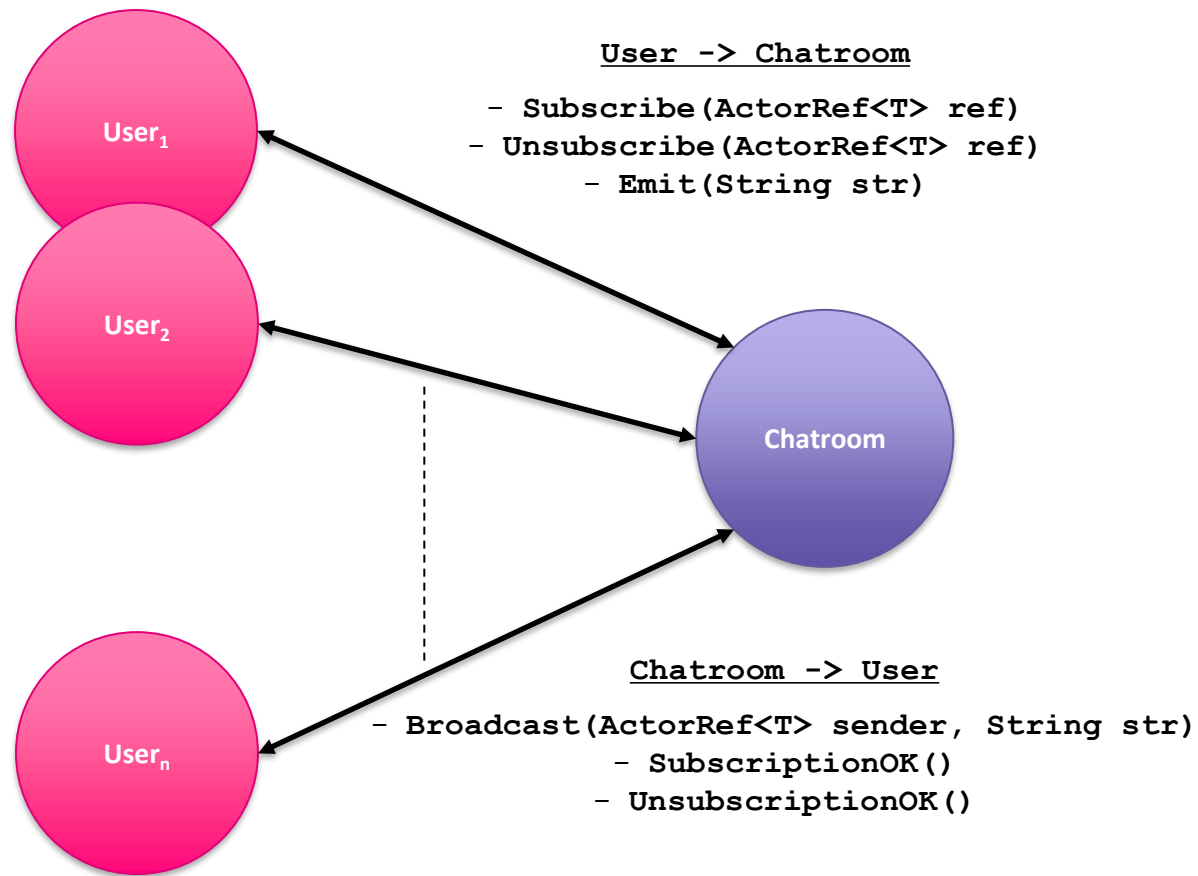
A broadcast chatroom

- A set of user actors may subscribe to a chatroom actor
  - The chatroom must confirm the subscription

- Users may emit messages that the chatroom broadcasts to all subscribers (except for the sender)

- Users may unsubscribe
  - The chatroom must confirm the un-subcription.

User -> Chatroom

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

Chatroom -> User

– **Broadcast(ActorRef<T> sender, String str)**
– **SubscriptionOK()**
– **UnsubscriptionOK()**
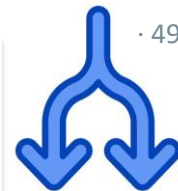
Broadcaster

- A set of user actors may subscribe to a chatroom actor
  - The chatroom must confirm the subscription

- Users may emit messages that the chatroom broadcasts to all subscribers (except for the sender)

- Users may unsubscribe
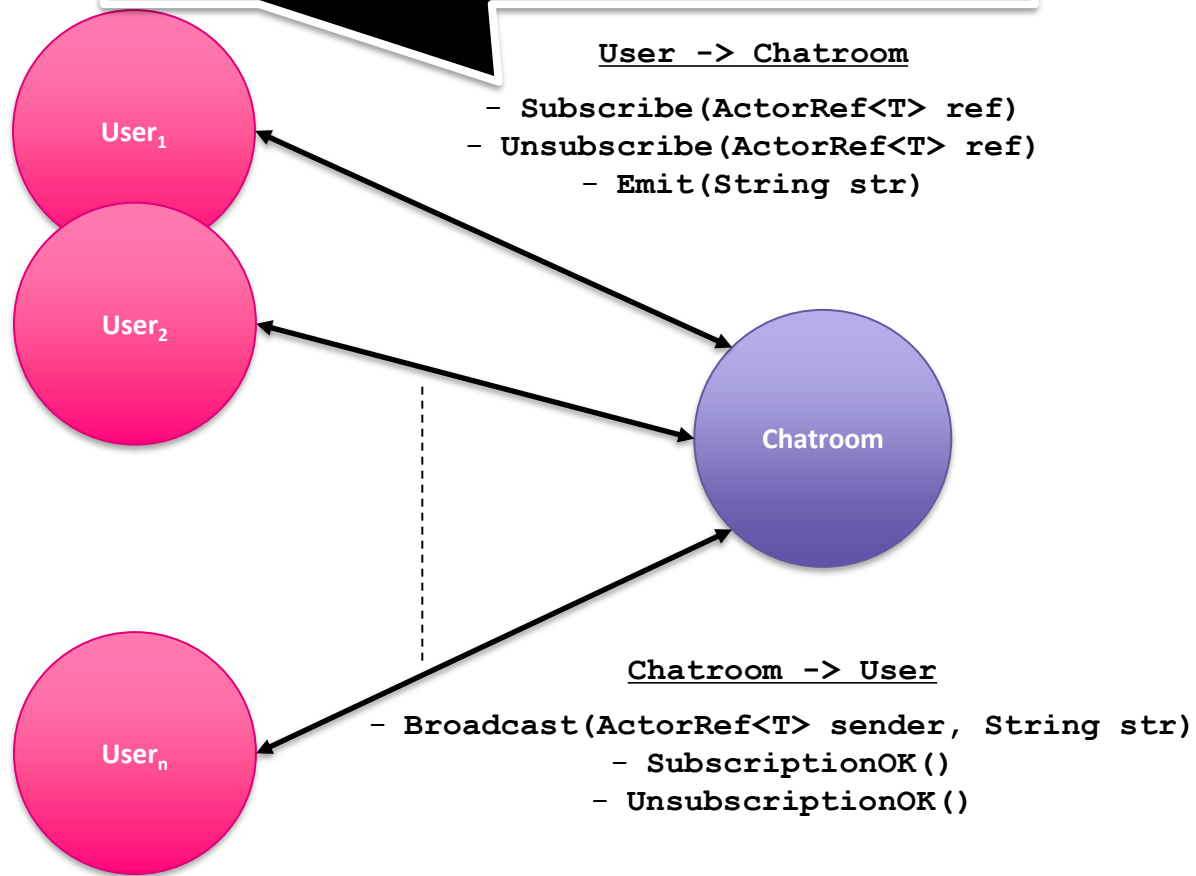 - The chatroom must confirm the un-subcription.

Important detail, messages do not contain information about the sender. If, for instance, the sender needs a reply, the message must contain a reference to the sender
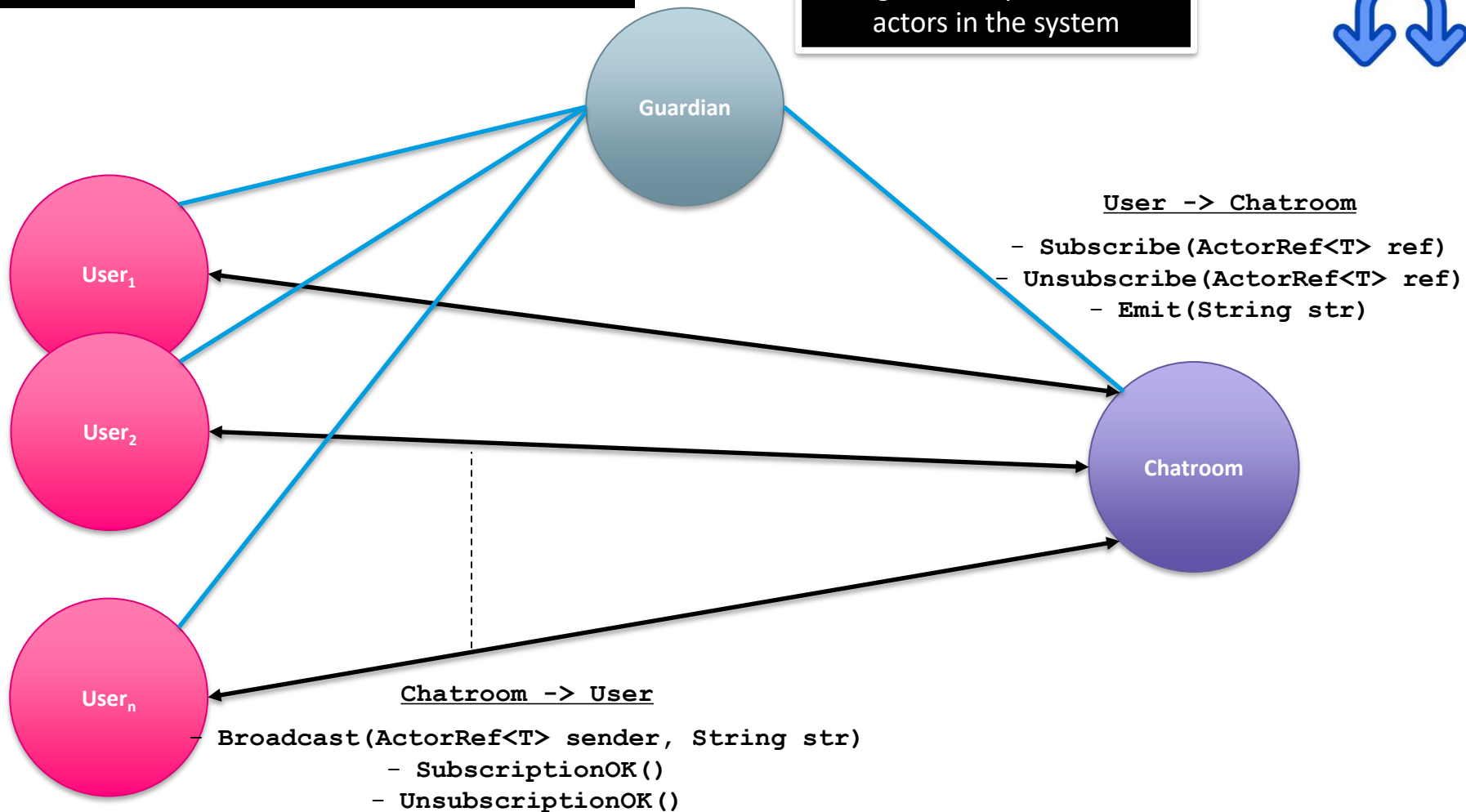
User -> Chatroom

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

User₁

User₂

Chatroom

Userₙ

Chatroom -> User

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

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

Broadcaster + Guardian

The guardian spawns all the actors in the system

Guardian

User₁

User₂

Userₙ

Chatroom

User -> Chatroom

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

Chatroom -> User

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

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

Guardian -> User

– **UserStarter()**

Guardian

User -> Chatroom

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

User$_1$

User$_2$

Chatroom

User$_n$

Chatroom -> User

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

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

Broadcaster + Guardian + Main

Guardian -> User

- `UserStarter()`

Main -> Guardian

- `KickOff()`

Guardian

Main class

User$_1$

User$_2$

User$_n$

User -> Chatroom

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

Chatroom

Chatroom -> User

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

Guardian -> User

‒ **UserStarter()**

Main -> Guardian

‒ **KickOff()**

Main class

Guardian

A main class creates an actors system with the guardian. Then it sends a kickoff message for the guardian to start the system

User₁

User₂

Userₙ

User -> Chatroom

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

Chatroom

Chatroom -> User

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

Guardian -> User

- **UserStarter()**

Main -> Guardian

- **KickOff()**

Main class

Guardian

A main class creates an actors system with the guardian. Then it sends a kickoff message for the guardian to start the system

User₁

User₂

This System + Guardian + Main style is the standard way one should use to implement Akka actor systems.

User -> Chatroom

- ~~Su~~bscribe(ActorRef<T> ref)
- ~~U~~nsubscribe(ActorRef<T> ref)
- **Emit(String str)**

Chatroom

User_n

Chatroom -> User

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

Brodcaster – execution example

Let's look at the code
(broadcaster package)

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

Assuming FIFO mailboxes
(Akka's default)

- Consider this execution

1. User1 sends Subscription to Chatroom
2. User2 sends Subscription to Chatroom
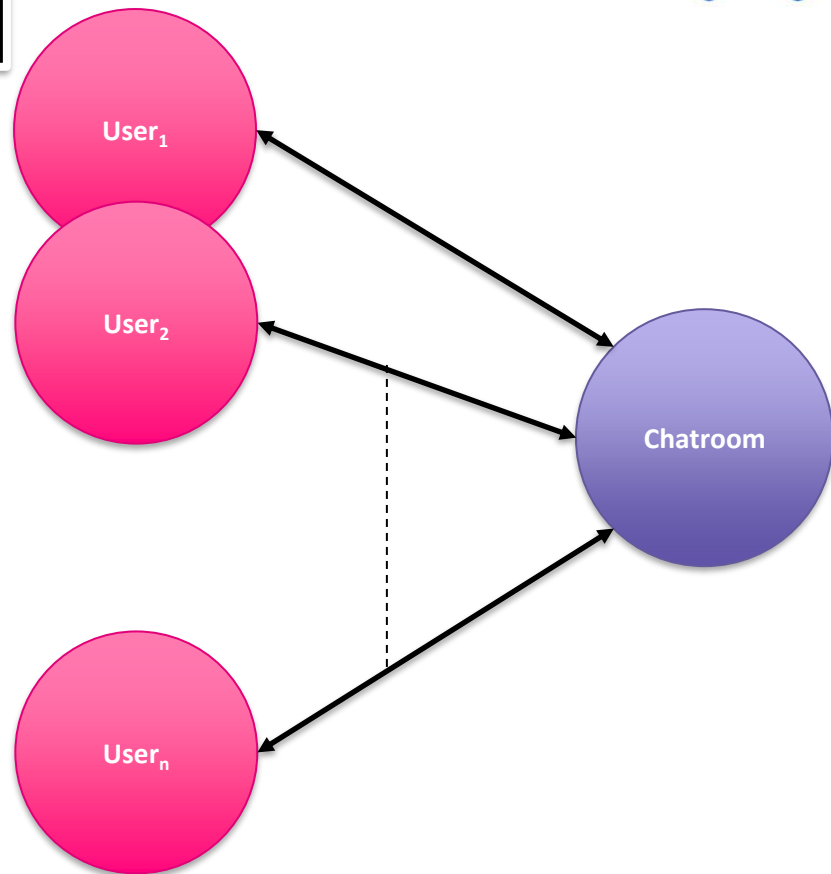3. …

What actor will receive first SubscriptionOK?
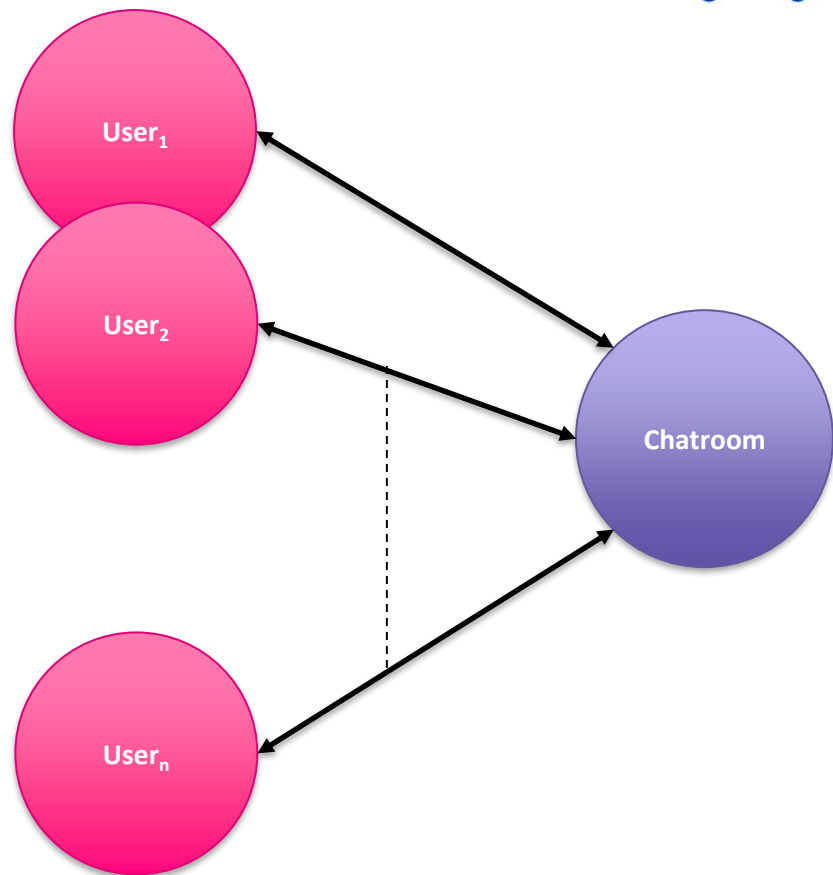
User$_1$

User$_2$

User$_n$

Chatroom

Broadcaster interesting executions

Assuming FIFO mailboxes
(Akka's default)

- Consider this execution

1. User1 sends Subscription to Chatroom
2. Chatroom replies SubscriptionOK to User1
3. User1 emits message to Chatroom
4. User2 sends Subscription to observable
5. …

Can User2 receive the message sent by User1 in step 3?

User$_1$

User$_2$

User$_n$

Chatroom

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022

Broadcaster interesting executions

Assuming FIFO mailboxes
(Akka's default)

- Consider this execution

1. User1 send Subscription to Chatroom
2. Chatroom replies SubscriptionOK to User1
3. User1 emits message to Chatroom
4. User2 sends Subscription to Chatroom
5. Chatroom replies SubscriptionOK to User2
6. …

**Can User2 receive the message sent by User1 in step 3?**

$User_1$

$User_2$

$User_n$

Chatroom

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022
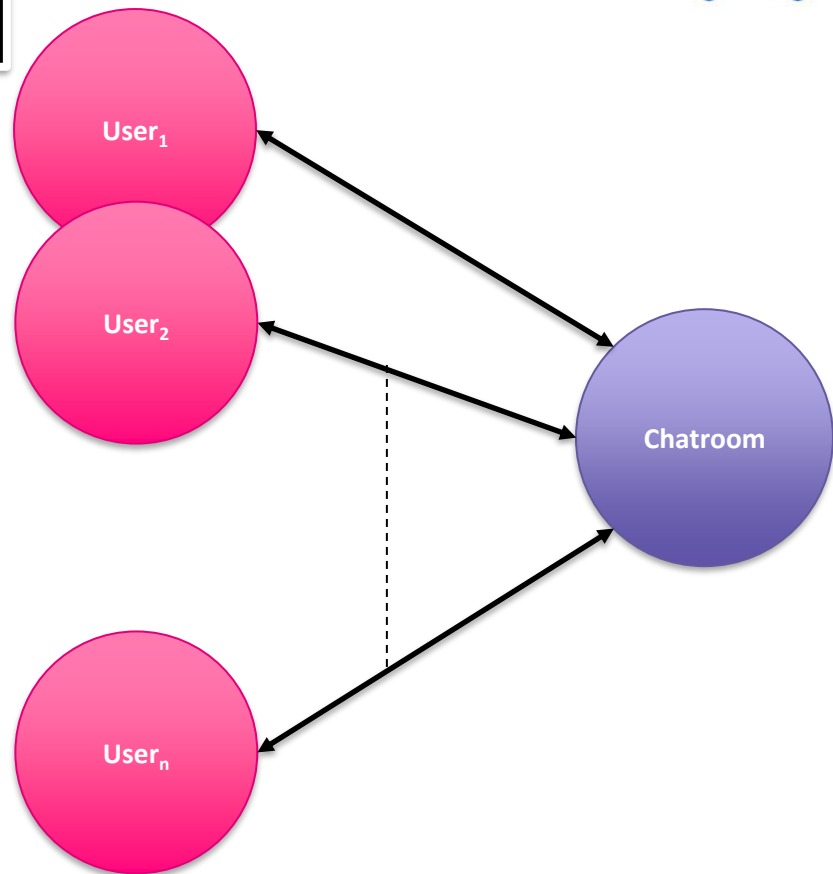
- Note that in the previous questions the behaviour of the systems depends on the reception of messages

- 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

A bounded buffer

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022
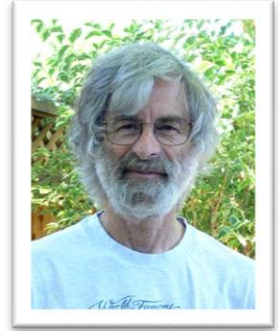
- Perhaps more intuitive example

Producers

Consumers

Shared data structure of fixed size

Producers

Consumers

Shared data structure of fixed size

Producer -> Buffer

- **Put(ActorRef<Producer> ref, int elem)**

Consumer -> Buffer

- **Get(ActorRef<Consumer> ref)**

Bounded Buffer

Buffer -> Consumer

- **Get(int elem)**
- **BufferEmpty**

Buffer -> Producer

- **ElementAdded**
- **BufferFull**

Consumer$_i$

Producer$_i$

Bounded Buffer with Actors

Producer -> Buffer

- **Put(ActorRef<Producer> ref, int elem)**

Consumer -> Buffer

- **Get(ActorRef<Consumer> ref)**

Bounded Buffer

Buffer -> Consumer

- **Get(int elem)**
- **BufferEmpty**

Buffer -> Producer

- **ElementAdded**
- **BufferFull**

Consumer_i

Producer_i

- 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 when the actors can make progress

Let's look at the code
(boundedbuffer package)

Bounded Buffer with Actors

Producer -> Buffer

– `Put(ActorRef<Producer> ref, int elem)`
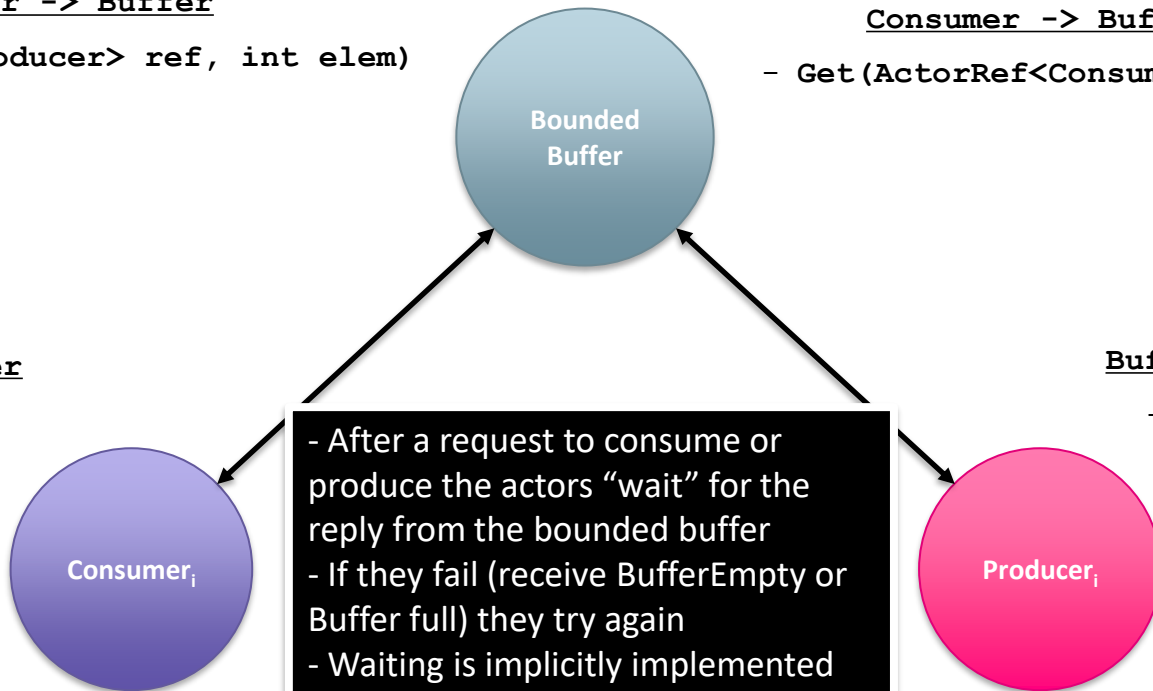
Consumer -> Buffer

– `Get(ActorRef<Consumer> ref)`

Bounded Buffer

Buffer -> Consumer

– `Get(int elem)`
 – `BufferEmpty`

Buffer -> Producer

– `ElementAdded`
 – `BufferFull`

Consumer$_i$

Producer$_i$

- 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 when the actors can make progress

Let's look at the code (boundedbuffer package)

Is this a good solution to the problem?

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2022
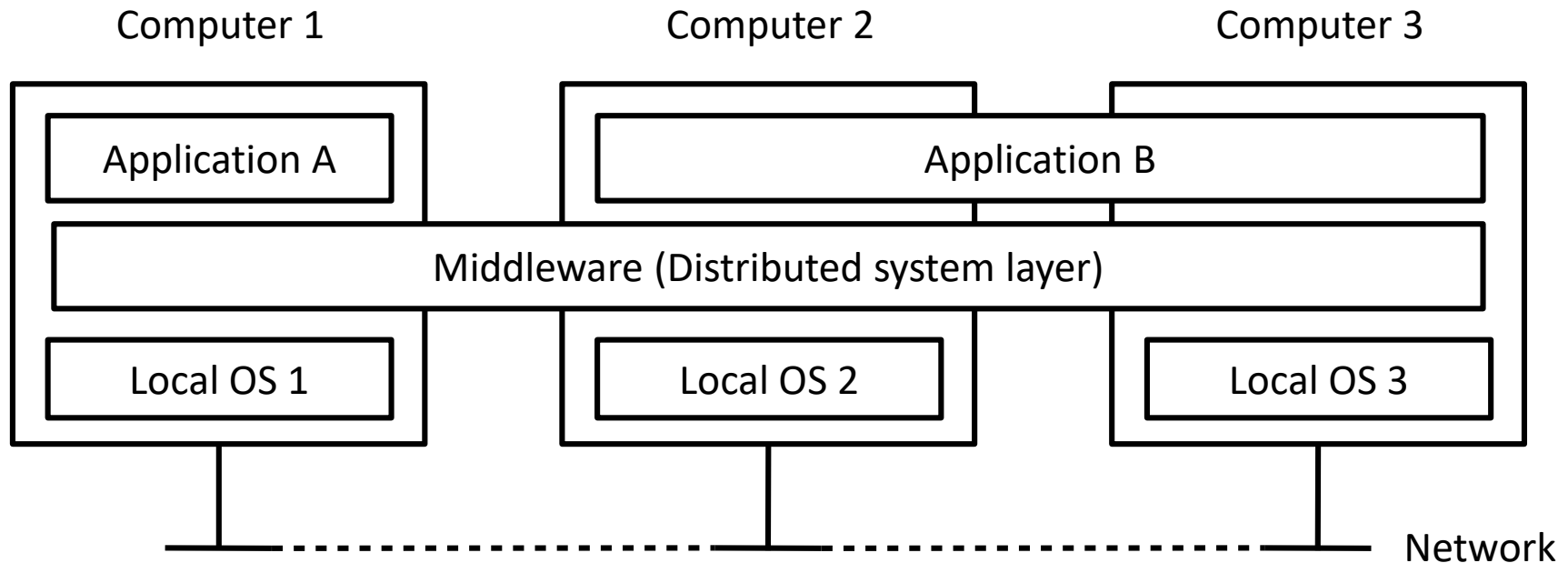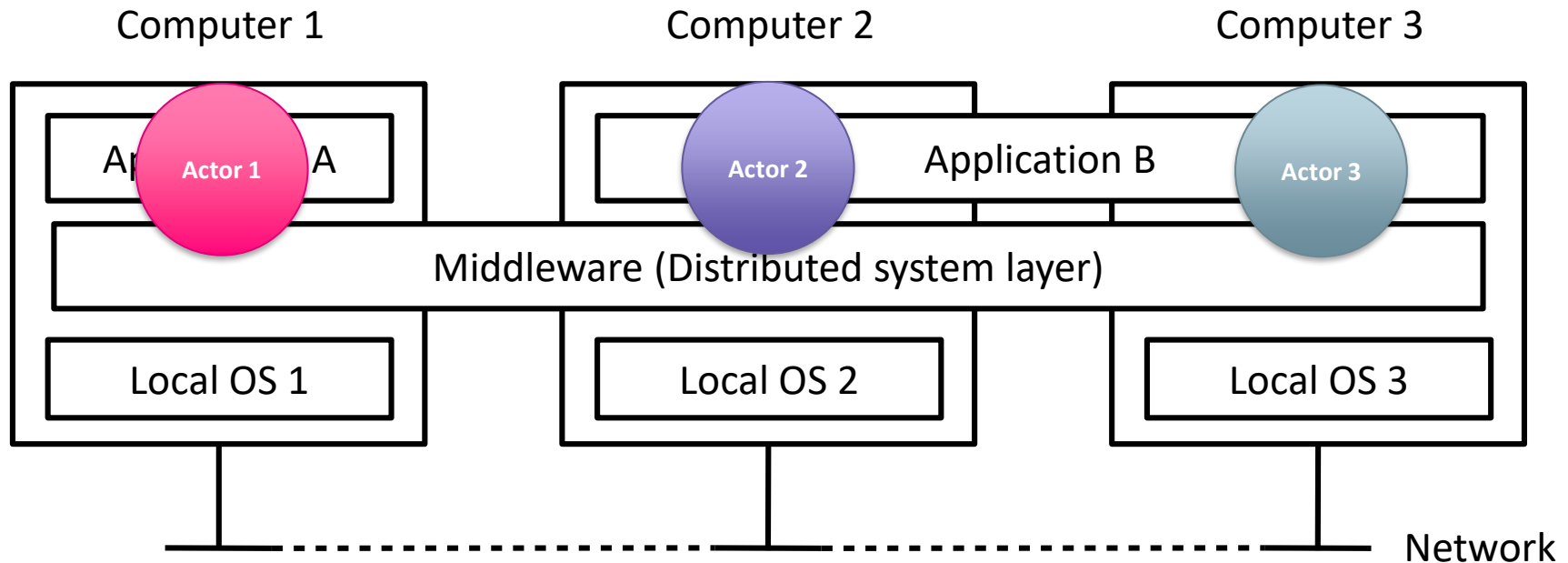
The actors model has natural mapping in distributed systems

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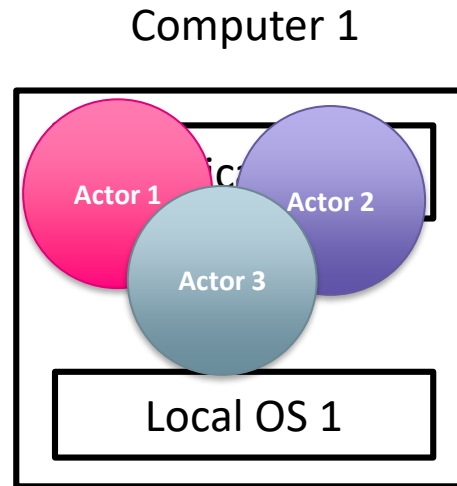
The actors model has natural mapping in distributed systems

The actors model is applicable in a single computer as well

Computer 1

Actor 1

Actor 2

Actor 3

Local OS 1

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

- Problems in shared memory concurrency (revisited)
- Actors
- Akka
- Example systems
  - Turnstile (counter)
  - Broadcaster
  - Bounded Buffer