Actor Fundamentals

[Prokopec, 2017; p. 247 ff.]

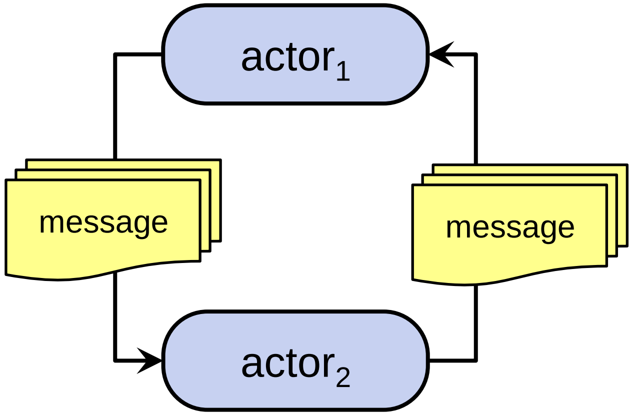
[Odersky et al., 2015; Why Actors?]

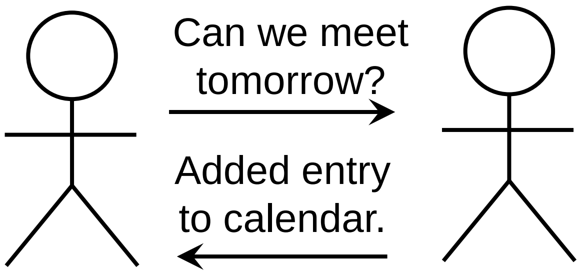
History

- Carl Hewitt et al, 1973:
  - The actor model was first described by Hewitt, Bishop, and Steiger (MIT) in the context of their research in artificial intelligence.
- Gul Agha, 1986:
  - Ph.D thesis: *Actors: A Model of Concurrent Computation in Distributed Systems*.
    - Showed how actor model can be exploited for large-scale parallelism
    - Described communication patterns for actor systems
    - Laid the foundation for actor-based languages
- Armstrong et al (Ericsson), 1986:
  - Erlang: Functional programming language whose concurrency model is based on actors.
  - Open Telecom Platform (OTP): 99.9999999 % availability (1995).
- 2006: Addition of actors to the Scala standard library.
- 2009: Implementation of first version of Akka.
- 2019: First stable version of Akka Typed Actors.

What is an Actor?

- According to Hewitt et al an actor
  - is an object with an identity,
  - that has an internal *state* and a *behavior*,
  - only interacts using *asynchronous message passing*.
- This mimics the behavior of humans.
  - They also communicate by transmitting messages (speaking, sending mails, etc.).
  - Transmitting message takes time.
  - Can perform activities while receiving messages.
  - Carry out activities one after the other.
  - Humans’ brains are totally “isolated”.





Actor Model

- The Actor Model is a model for *concurrent computation*.
  - Actors are the processing units.
  - It defines what is necessary for a computation to be distributed.
- When receiving a message an actor can do one of the following *fundamental operations*:
  - *Send* a finite number of *messages* to actors it knows.
  - *Create* a finite number of *new actors*.
  - *Define the behavior* to be applied to the next message (behavior can change over time).

A Simple Actor Example (1)

- Message types are usually modelled as case classes in the companion object of an actor.

```
object Calculator:
  case class Add(value: Int)
  case object Get
```

- Messages are processed in the `receive` method of an actor.

```
class Calculator extends Actor:
  import Calculator._
  private var value: Int = 0
  override def receive =
    case Add(s) => value += s
    case Get    => sender() ! value
```

  - `receive` returns a *partial function* that matches the incoming messages.
  - `sender()` returns a reference to the sender of the processed message.

A Simple Actor Example (2)

- The `Calculator` actor is used in another actor

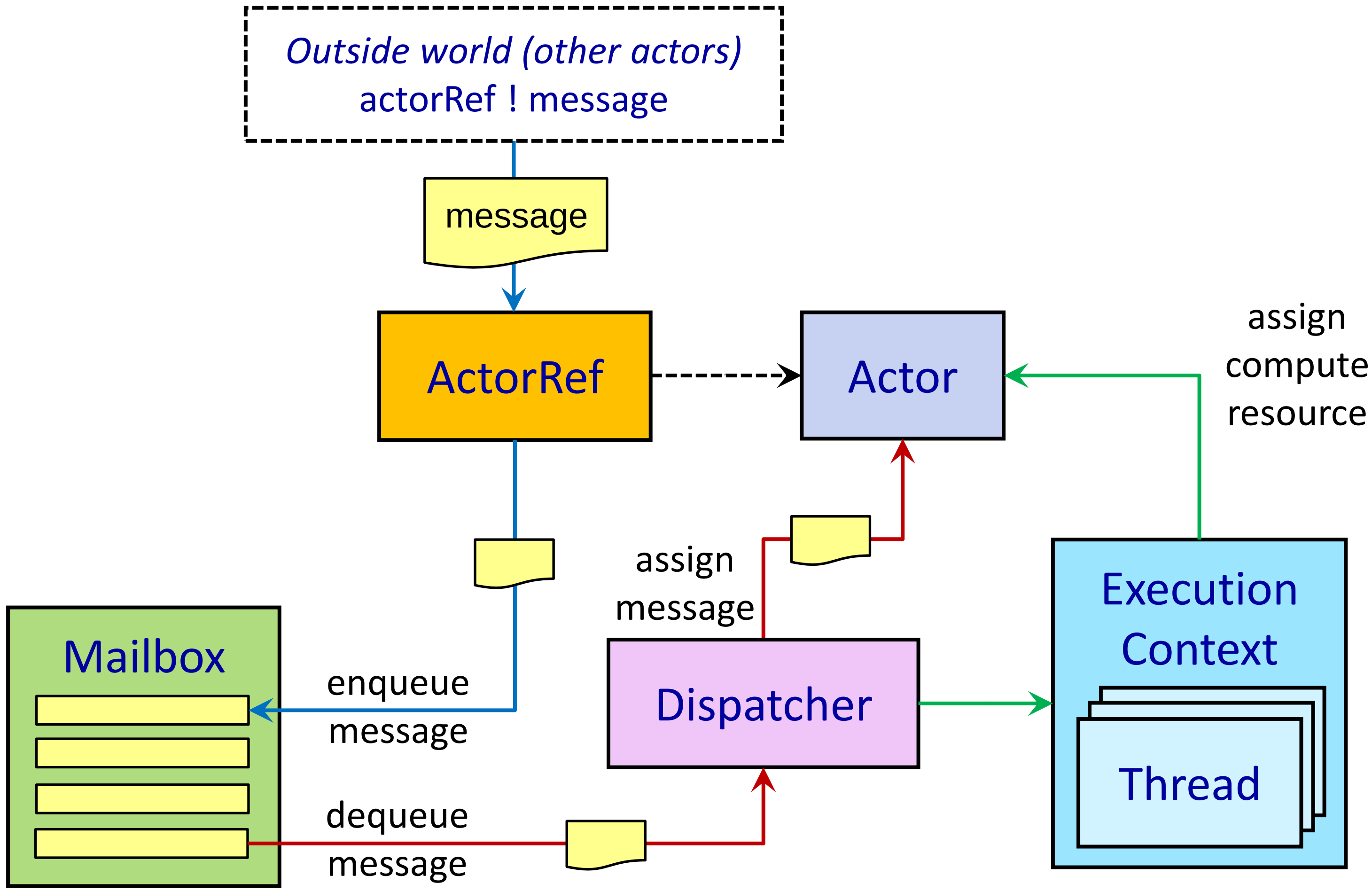
```
class MainActor extends Actor {
  import Calculator._
  val calculator = context.actorOf(Props[Calculator](), "calculator")
  calculator ! Get
  calculator ! Add(30)
  calculator ! Get
  override def receive = {
    case value : Int => println(s"current value = $value")
  }
}
```

  - `MainActor` creates an instance of the `Calculator` actor → instance becomes a child of the main actor.
  - `MainActor` sends messages to Calculator using the binary `!` *(tell)* operator.
    We say: *"calculator **tell** Add(30)".*
  - Messages are sent asynchronously → sender is not blocked until message is processed.

Components of an Actor System

- *Actor System:* Hierarchical group of actors.
- *Actor Class:* Template for an actor.
- *Actor Instance:* Entity that exists at runtime. Created from an actor class.
- *Actor Reference:* Object used to address an actor. Hides information about the location of an actor.
- *Message:* The unit of information transmitted between actors.
- *Mailbox:* Queue associated with an actor that is used to buffer messages.
- *Dispatcher:* Component that assigns compute resources to actors.

Components of an Actor

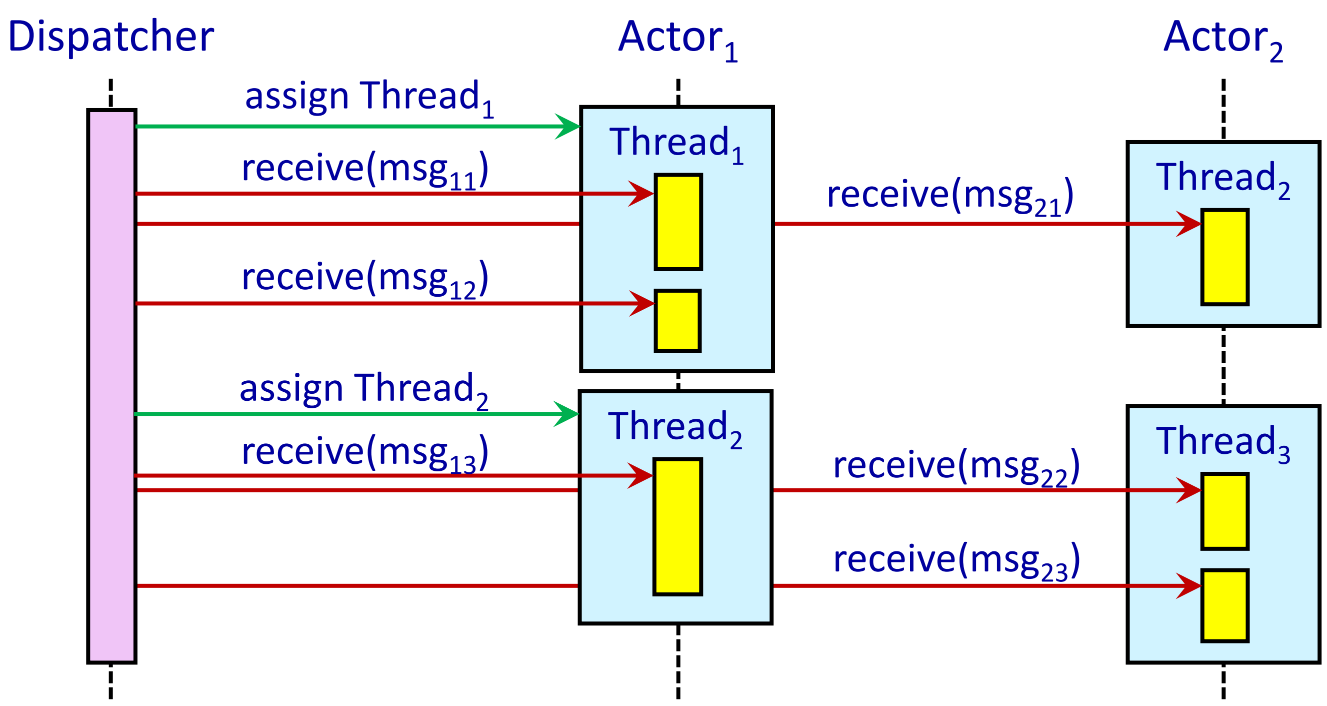


Actor Features

- Encapsulation
  - The actor's state and behavior is totally isolated for the outside world.
  - One cannot access methods and fields of the actor class.
- Message centric
  - Actors communicate solely via one-way message-passing.
    - Send work orders to actors.
    - Publish results to other actors.
  - Messages are immutable.
- No synchronization within actor necessary.
  - Messages are processed sequentially.

Message Processing

- Actors are *lightweight*.
  - Actors are not mapped to a single thread.
  - A server may host a few thousands of threads, but millions of actors.



Classic Actors

Implementing Actors – Classic

- The *behavior* of an actor is defined in an *actor class*.
- This class also encapsulates the actor's *state*.
- In the `receive` method it is described how messages are processed.

```
trait Actor:
  abstract def receive: PartialFunction[Any, Unit]
```

  - returns a *partial function* that processes the actor's messages
  - changes the state
  - communicates with other actors
- Example:

```
class Calculator extends Actor:
  import Calculator._
  private var value: Int = 0          // representation of state
  override def receive =              // processing of messages
    case Add(n) => value += n         // state change
    case Get    => sender() ! value   // send message to sender of message
```

Actor API – Classic

```
trait Actor:
  type Receive = PartialFunction[Any, Unit]

  abstract def receive:  Actor.Receive
  implicit val context:  ActorContext
  implicit val self:     ActorRef
  def          sender(): ActorRef
```

- `receive`: must be overridden with a function that
  - returns a *partial function* that accepts *any* type of messages and *returns nothing*,
  - accepts any number of arguments.
- `context`: exposes contextual information for the actor and the current message, e. g. the actor's children, passed implicitly to various methods
- `self`: reference to this actor
- `sender`: reference to the sender of the current message

ActorContext – Classic

- The *actor context* decouples the actor's logic from the actor's infrastructure functionality.
- It provides methods for the following actions:
  - Actor creation; stopping an actor
  - Changing the actor's behavior
  - Navigating in the actor hierarchy

```
trait ActorContext:
  def props: Props
  def actorOf(props: Props, name: String): ActorRef
  def stop(actor: ActorRef): Unit
  def become(behavior: Receive, discardOld: Boolean): Unit
  def parent: ActorRef
  def children: Iterable[ActorRef]
  def system: ActorSystem
  def actorSelection(path: ActorPath): ActorSelection
```

Creating Actors - Classic

- The *actor configuration* contains information about
  - the actor class and its constructor arguments
  - deployment information
  - various other system components: mailbox, dispatcher, router
- The configuration parameters are bundled in a `Props` object.

```
object Calculator:
  def props1() = Props[Calculator]()    // → new Calculator()
  def props2(initValue: Int) = Props(classOf[Calculator], initValue)
                                        // → new Calculator(initValue)
  def props3() = Props(classOf[Calculator], initValue).withDeploy(...)
```

- `Props` allows the creation of actor instances in arbitrary environments.

```
val calc = context.actorOf(Calculator.props2(10), "calculator")
```

Actor Systems – Classic

- An *actor system* is a container for a hierarchical group of actors.
- Supported functions:
  - `actorOf`: creation of user actor a the top of the actor hierarchy
  - `actorSelection`: obtaining a reference to running actors
  - `stop` : stopping of individual actors
  - `terminate`: termination of actor system (propagated to all actors)
- Example:

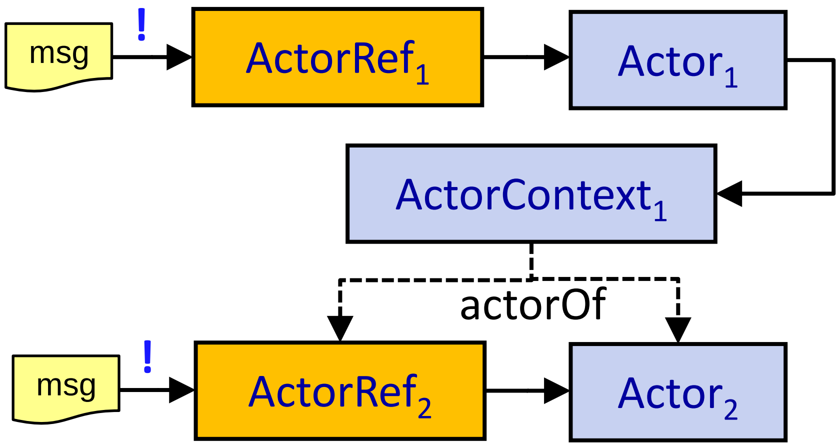
```
val system = ActorSystem("MyActorSystem")
system.actorOf(Props[MainActor](), "mainActor")
…
val f : Future[Terminated] = system.terminate();
```

Actor References (`ActorRef`) – Classic

- Actors communicate by exchanging messages.

```
receivingActorRef ! message
```

- Actor instances are never addressed directly but via *actor references.*
- Actor references hide information about the location of an actor.
- It is not guaranteed that an actor reference is backed by an actor.



- `ActorRef` API:

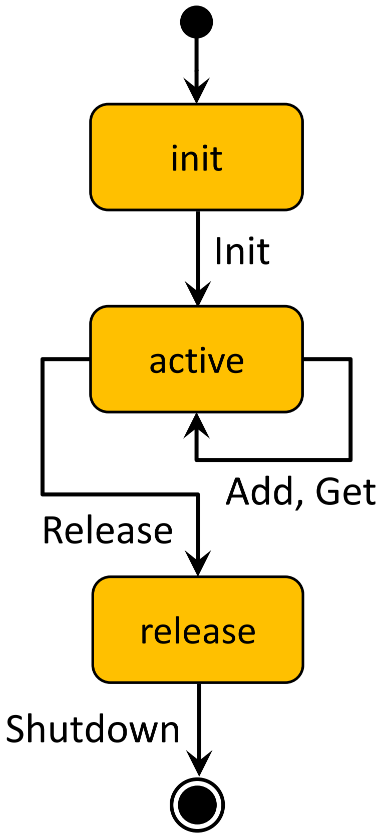
```
abstract class ActorRef:
  def !(message: Any)(using sender: ActorRef): Unit
```

- A message my be any type of (serializable) object.
- By default, the `sender` is a reference to the actor that posted the message (`self`).

Actor Behavior (1) – Classic

- The way an actor handles messages is called the *behavior* of an actor.
- The initial behavior is defined by the `receive` method.
- `context.become` changes the actor's behavior.
  - It expects a partial function that defines the new behavior
- Example:

```
class Calculator extends Actor:
  private var value: Int = 0
  def init : Actor.Receive =
    case Init(n) => { value = n; context.become(active) }
  def active : Actor.Receive = {
    case Add(s)  => value += s
    case Get     => sender() ! value
    case Release => context.become(release)
  def release : Actor.Receive = …
  override def receive = init
```



Actor Behavior (2) – Classic

- It is good practice to scope the state to the current behavior.
  - State is stored locally.
  - State is only changed in `context.become`.
  - `become` evaluates its argument only when the next message comes in.
- Example:

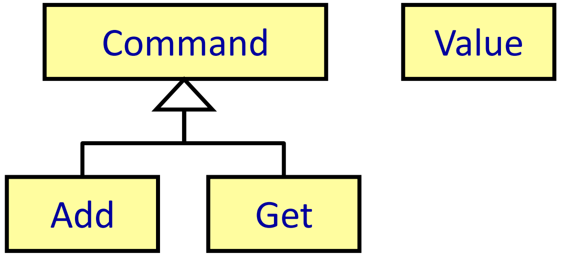
```
class Calculator extends Actor:
  def init : Actor.Receive =
    case Init(n) => context.become(active(n))

  def active(value: Int) : Actor.Receive =
    case Add(s) => context.become(active(value + n))
    case Get    => sender() ! value
    case Release => context.become(release)

  def release : Actor.Receive = …

  def receive = init
```

Typed Actors

Definition of the Message Types

- Messages define the protocol used by the actors to communicate with each other.
- A typed actor accepts a message of a specific type only (template parameter).
  - Message types must have a common root.
  - Usually, messages are defined in the companion object or a designated wrapper object.

```
object Calculator:
  sealed trait Command
  final case class Add(n: Int) extends Command
  final case class Get(replyTo: ActorRef[Value]) extends Command
  final case class Value(n: Int)
```



- Sealed base trait
  - Can be extended, but only in the same source file.
  - Improved pattern matching: compiler warns if matching is not exhaustive.
- Sending of responses: Reference to sender must be contained in message (`replyTo`).

Actor Behavior

- The *behavior* of an actor defines how it reacts to incoming messages.
  - `receive/onMessage`: processes messages of type parameter T sent by other actors.
  - `receiveSignal/onSignal`: processes signals (life cycle events) sent by the system.

```
abstract class Behavior[T]
```

```
abstract class ExtensibleBehavior[T] extends Behavior[T]:
  def receive      (ctx: TypedActorContext[T], msg: T):      Behavior[T]
  def receiveSignal(ctx: TypedActorContext[T], msg: Signal): Behavior[T]
```

```
abstract class AbstractBehavior[T](protected val context: ActorContext[T])
    extends ExtensibleBehavior[T]:
  override final def receive      (ctx: TypedActorContext[T], msg: T):      Behavior[T]
  override final def receiveSignal(ctx: TypedActorContext[T], msg: Signal): Behavior[T]

  abstract def onMessage(msg: T): Behavior[T]
  def onSignal: PartialFunction[Signal, Behavior[T]] = PartialFunction.empty
```

Actor Behavior: OO-based Implementation

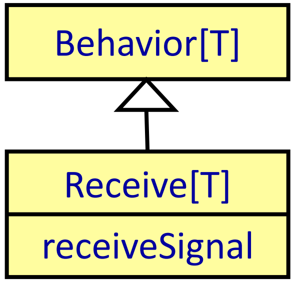
- The actor's behavior can be implemented in a class

```
class Calculator(context: ActorContext[Calculator.Command]) extends
    AbstractBehavior[Calculator.Command](context):

  private var value: Int = 0

  override def onMessage(msg: Calculator.Command): Behavior[Calculator.Command] =
    msg match
      case Add(n)       => value += n
      case Get(replyTo) => replyTo ! Value(value)

    this
```

- apply creates a behavior instance and injects the actor context.

```
object Calculator:
  def apply(): Behavior[Calculator.Command] =
    Behaviors.setup(context => new Calculator(context))
```

Factory Methods for Behaviors



- Object `Behaviors` contains methods for creating behaviors.
  - Behavior implementation is passed as a function.
  - Basis for functional implementation of actors.

```
object Behaviors:
  def setup[T]         (factory:   ActorContext[T] => Behavior[T]):      Behavior[T]
  def receive[T]       (onMessage: (ActorContext[T], T) => Behavior[T]): Receive[T]
  def receiveMessage[T](onMessage: T => Behavior[T]):                    Receive[T]
  def receiveSignal[T] (handler:   PartialFunction[(ActorContext[T], Signal), Behavior[T]]):
                                                                         Behavior[T]

  def same[T]:    Behavior[T]
  def stopped[T]: Behavior[T]
  def ignore[T]:  Behavior[T]
  def empty[T]:   Behavior[T]
```

- `setup`: Initializes actor's behavior.
- `receive[T]`: Defines how incoming messages are processed.
- `receiveSignal[T]`: Handles processing of system signals.

Changing the Behavior of an Actor

- Each message can change the behavior of an actor
  - The new behavior must be returned in `onMessage`/`onSignal`.

```
abstract class AbstractBehavior[T] … :
  def onMessage(msg: T): Behavior[T]
  def onSignal: PartialFunction[Signal, Behavior[T]]
```

- Example

```
override def onMessage(msg: Command): Behavior[Calculator.Command] =
  msg match
    case Add(n)       => value += n;              Behaviors.same // or this
    case Get(replyTo) => replyTo ! Value(value); Behaviors.same // or this
    case Shutdown     =>                          Behaviors.stopped
```

Functional Implementation of an Actor – Example

```
object CalculatorMain:
  def apply(): Behavior[Value] =
    Behaviors.setup[Value] { context =>
      val calc: ActorRef[Calculator.Command] = context.spawn(Calculator(), "calculator")
      calc ! Add(100)
      calc ! Get(context.self)
      Behaviors.receiveMessage[Value] {
        case Value(n) =>
          println(s"Current value: $n")
          context.system.terminate()
          Behaviors.stopped
      }
    }
```

- `{ context => … Behaviors.receiveMessage { … } }`

  Function that maps an `ActorContext[Value]` to a `Behavior[Value]`
- `{ case Value(n) => … Behaviors.stopped }`

  Literal for partial function mapping a message of type `Value` to a `Behavior[Value]`

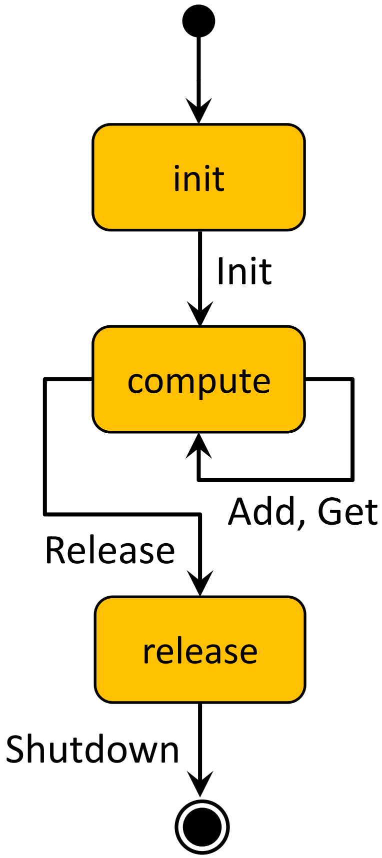
Functional Implementation – State Management

- State Management
  - Behavior classes can hold state in private fields.
  - Functional implementations must pass state in parameters.
- Example:

```
object Calculator:
  def apply(): Behavior[Command] = calculator(0)
  private def calculator(value: Int): Behavior[Command] =
    Behaviors.receiveMessage {
      case Add(n) =>
        calculator(value + n)
      case Get(replyTo) =>
        replyTo ! Value(value)
        Behaviors.same
    }
```

Actors as Finite State Machines

```
object Calculator:
  def apply(): Behavior[Command] = init()
  private def init(): Behavior[Command] =
    Behaviors.receiveMessage {
      case Init(value) => compute(value)
      case _           => throw IllegalStateException()
    }
  private def compute(value: Int): Behavior[Command] =
    Behaviors.receiveMessage {
      case Add(n)       => compute(value + n)
      case Get(replyTo) => replyTo ! Value(value); Behaviors.same
      case Release      => release
      case _            => throw IllegalStateException()
    }
  private def release: Behavior[Command] = …
```



ActorContext

- The *actor context* decouples the actor's logic from the actor's infrastructure functionality.
- It provides methods for the following actions:
  - Creating and stopping actors
  - Navigating in the actor hierarchy
  - Registering for events

```
trait ActorContext[T]:
  def spawn[U](behavior: Behavior[U], name: String): ActorRef[U]
  def stop[U](actor: ActorRef[U]): Unit
  def self: ActorRef[T]
  def child(name: String): Option[ActorRef[Nothing]]
  def children: Iterable[ActorRef[Nothing]]
  def system: ActorSystem[Nothing]
  def watch[U](actor: ActorRef[U]): Unit
  def scheduleOnce[U](delay: FiniteDuration, target: ActorRef[U], msg: U): Cancellable
```

- Many methods in `ActorContext` are not thread-safe.

Actor Systems

- An *actor system* is a container for a hierarchical group of actors.
- The constructor creates the top-level actor, also known as the *user guardian actor*.
- Supported functions:
  - `terminate`: termination of actor system (propagated to all actors).
  - `whenTerminated`: returns a futures that is completed when actor system finishes termination.

```
val system = ActorSystem(UserGuardianBehavior(), "my-actor-system")
system.terminate()
Await.ready(system.whenTerminated, Duration.Inf)
```

- Provides access to other facilities:
  - `settings`: configuration information
  - `deadLetters`: actor where messages to non-existing actors are routed to
  - `scheduler`: for running asynchronous tasks
  - `eventstream`: event bus all actors can subscribe to

Creating Actors

- The ability to create new actors is a fundamental capability of an actor.
- `context.spawn` creates new actor instances
  - Input: Actor behavior
  - Output: Reference to newly created actor
- New actor becomes a child of its creator.

```
object Calculator
  def apply(): Behavior[Command] = …
```

```
Behaviors.setup[Value] { context =>
  val calc: ActorRef[Calculator.Command] = context.spawn(Calculator(), "calculator")
}
```

- In the 3rd parameter additional actor properties can be specified:

```
context.spawn(childBehavior, "bounded-mailbox-child", MailboxSelector.bounded(100))
context.spawn(childBehavior, "DispatcherFromConfig",
              DispatcherSelector.fromConfig("custom-dispatcher"))
```

Stopping Actors

- `context.stop`: Terminate actor immediately after current message is processed.

```
context.stop(actorRef)
```

  - Child actors will be stopped before.

- `Behaviors.stopped`: Actor stops itself.

```
Behaviors.receive[Command] { (context, msg) => msg match
  case GracefulShutdown =>
    Behaviors.stopped
}
```

  - Do not use `context.stop(context.self)`.

- `system.terminate()`: Terminates the actor system.

```
val f: Future[Terminated] = actorSystem.terminate()
```

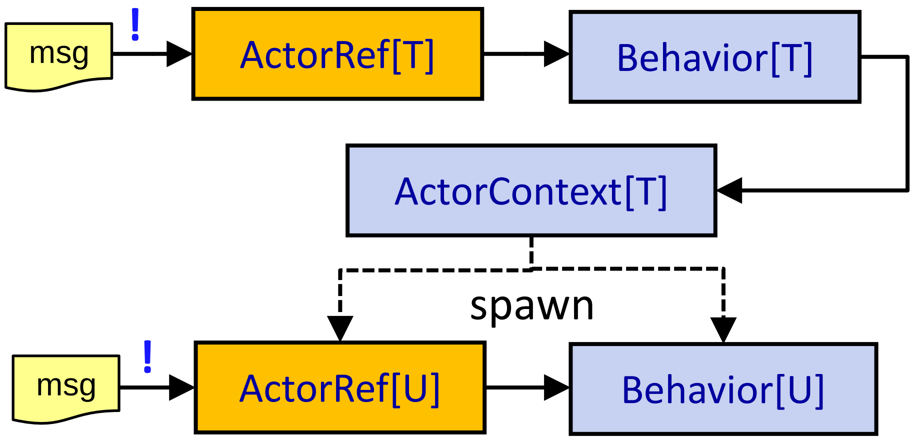
  - Stops the guardian actor, which in turn will recursively stop all child actors.

Actor References (ActorRef)

- Actors communicate by exchanging messages.

```
receivingActorRef ! message
```

- Actor instances are never addressed directly but via *actor references.*
- Actor references hide information about the location of an actor.
- It is not guaranteed that an actor reference is backed by an actor.



- `ActorRef` API:

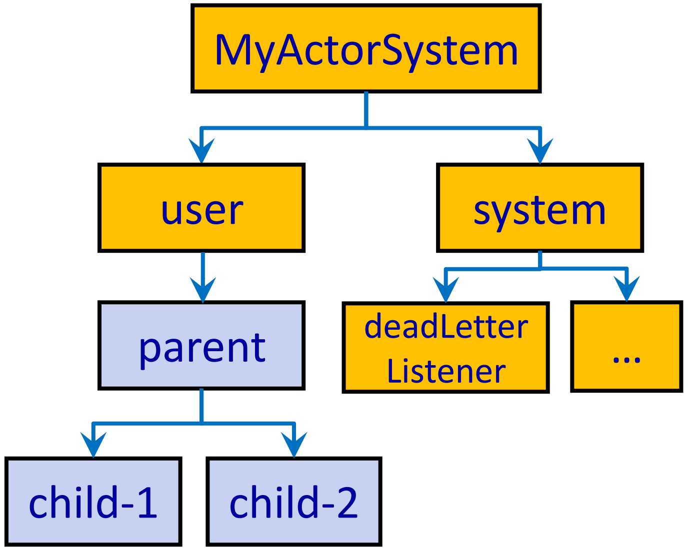
```
trait class ActorRef[T]:
  def tell(msg: T): Unit
```

- An `ActorRef` can only send messages of the type the corresponding actor accepts.
- `ActorRef` can implicitly be converted to `ActorRefOps` that provides the bang (`!`) operator (alias for `tell` method).

Actor Hierarchy

- Actors are organized in a tree structure
  - Actors closer to the root perform more general tasks.
  - They can delegate work items to child actors.
  - This structure is the basis of error handling → parent actors are the supervisors of its children.

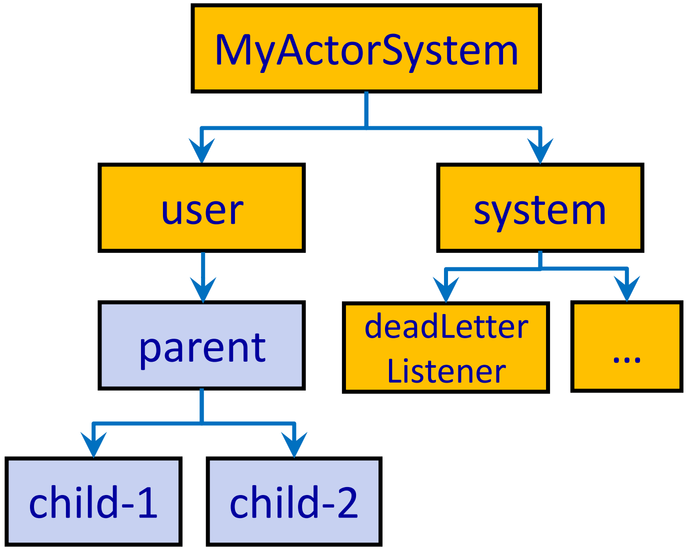
```
// Root defines the behavior of the user guardian actor.
val system = ActorSystem(Root(), "MyActorSystem")

object Root:
  def apply() = Behaviors.setup[Unit] {
    context => context.spawn(Parent(), "parent")
  }

object Parent:
  def apply() = Behaviors.receive[Parent.Command] {
    (context, CreateChild(name)) =>
      context.spawn(Child(), name);
      Behaviors.same
  }
```



- `user` is the *guardian actor* for all user-created top-level actors.
- `system` is the *guardian actor* for all system-created top-level actors.

Navigating in the Actor Hierarchy

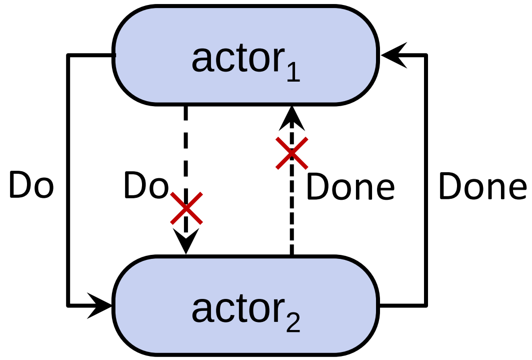
```
object Parent:
  def apply() = Behaviors.receive[Parent.Command] {
    case (context, CreateChild(name)) => …
    case (context, PrintHierarchy) =>
      println(s"this actor: ${context.self.path.name}")
      for (child <- context.children)
        println(s"  child: ${child.path.name}")
      Behaviors.same
  }
```



- Each actor can access all its children (`context.children`).
- The parent actor is not accessible.
- The actor path (`actorRef.path`) is a string containing actor names from the user guardian do the designated actor.
  - Example: `akka://MyActorSystem/user/parent/child-1`

Interaction Patterns

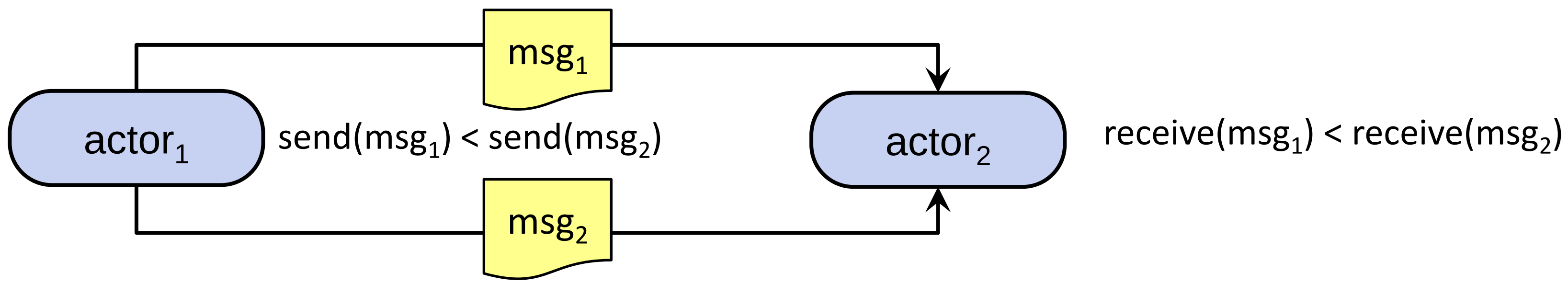
Message Delivery Semantics



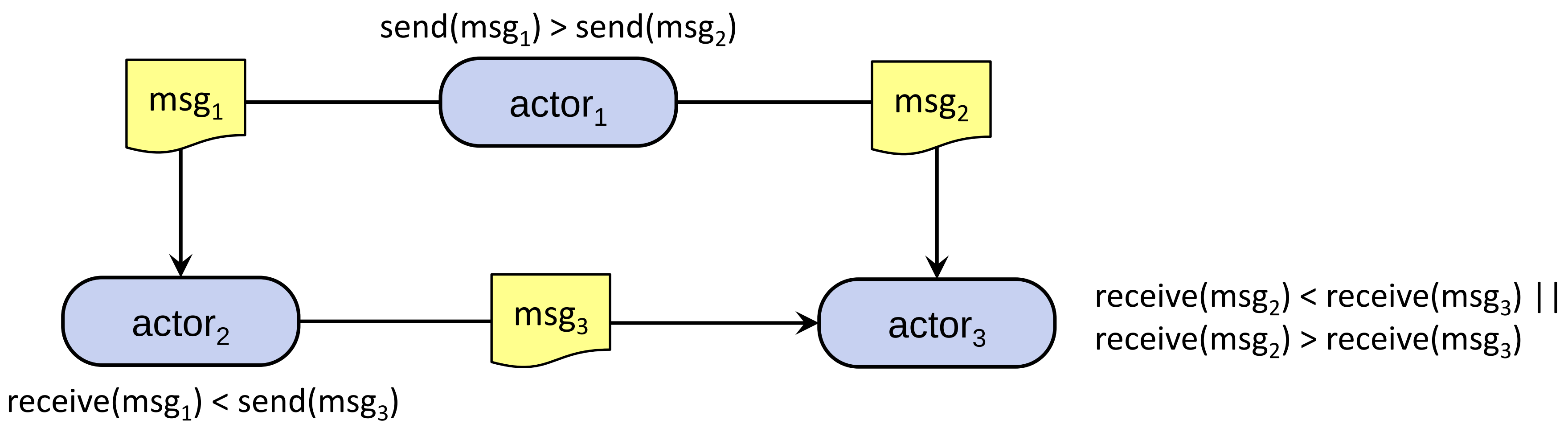
- Message delivery is *inherently unreliable*.
- Higher delivery guaranties can only be achieved, if
  - the receiver sends *confirmation* messages,
  - *messages are resent* when no confirmation arrives,
  - the sender/receiver keeps track of confirmations/processed messages.
- Classification of delivery guaranties:
  - **at-most-once**: sending once, [0,1] deliveries.
  - **at-least-once**: resending until confirmation arrives, $[1, \infty)$ deliveries.
    - Sender must keep track of messages which it got no confirmation for.
  - **exactly-once**: at-least-once + process only first message.
    - Receiver must keep track of messages it already processed.
- Message processing can only be guaranteed, if the receiver confirms messages in its business logic.

Message Ordering

- Multiple messages sent from one actor to another keep their ordering.



- Message ordering is not guarantied for a group of three or more actors.



Fire and Forget

- Receiver

```
object Receiver:
  case class Command(message: String)
  def apply(): Behavior[Command] =
    Behaviors.receiveMessage {
      case Command(message) =>
        …
        Behaviors.same
    }
```

- Sender

```
val receiver: ActorRef[Receiver.Command] = context.spawn(Receiver(), "receiver")
receiver ! Receiver.Command("message 1")
```

- Problems
  - Lost messages cannot be detected.
  - Sender can produce messages at a higher rate than the receiver is able to process.

Request/Response

Messages

```scala
case class Request(message: String, replyTo: ActorRef[Response])
case class Response(result: String)
```

Receiver

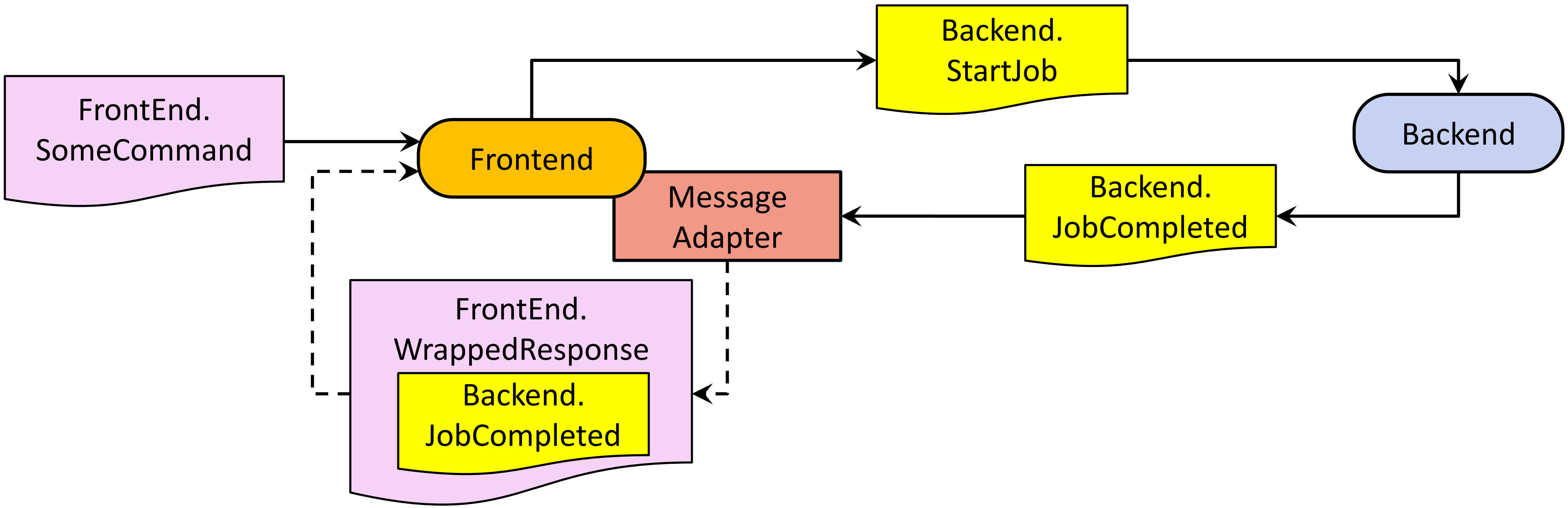
```scala
def apply(): Behaviors.Receive[Request] =
  Behaviors.receiveMessage[Request] {
    case Request(message, replyTo) =>
      // process message
      replyTo ! Response("response message")
      Behaviors.same
  }
```

Sender

```scala
def apply() = Behaviors.setup[Response] {
  context =>
    val receiver = context.spawn(Receiver(), …)
    receiver ! Request("message", context.self)

    Behaviors.receiveMessage[Response] {
      case Response(result) =>
        // process response
        Behaviors.same
    }
}
```

- Problems
  - It is hard to detect that a message request was not delivered or processed.
  - Actors seldom have a response message from the receiving actor as a part of their protocol.

Adapted Response (1)

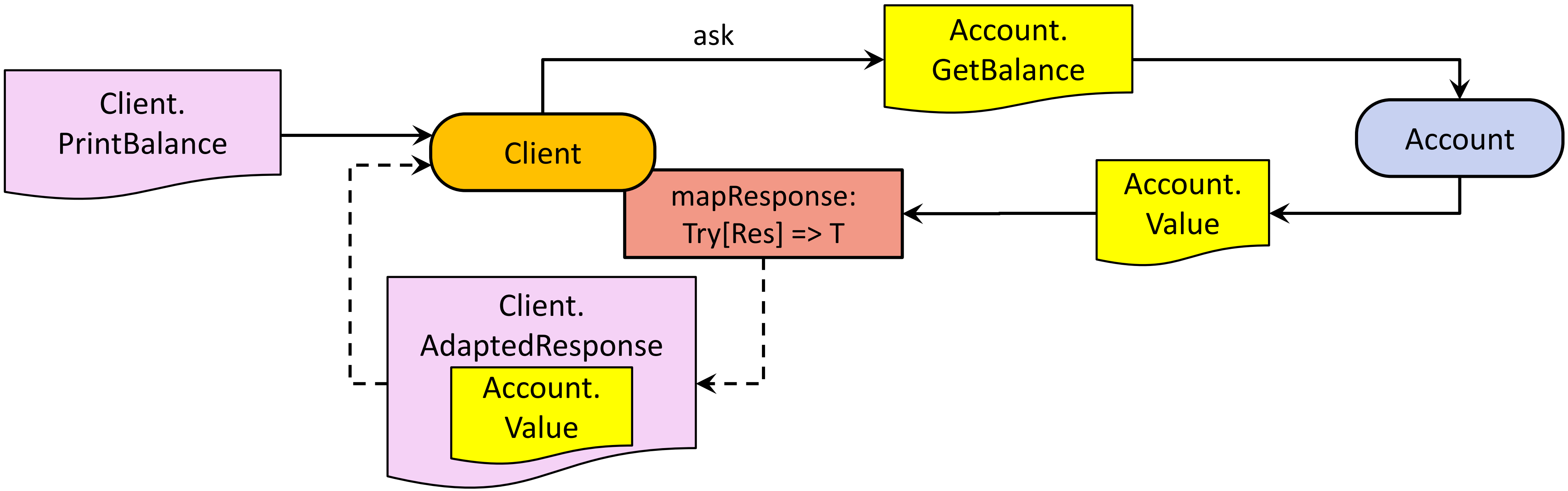



```scala
object Backend:
  sealed trait Request
  case class StartJob(task: Task, replyTo: ActorRef[Response]) extends Request
  sealed trait Response
  case class JobCompleted(result: Result) extends Response
```

Adapted Response (2)

```scala
object Frontend:

  sealed trait Command
  case class SomeCommand() extends Command
  private case class WrappedBackendResponse(response: Backend.Response) extends Command

  def apply(backend: ActorRef[Backend.Request]): Behavior[Command] =
    Behaviors.setup[Command] { context =>
      val backendResponseMapper: ActorRef[Backend.Response] =
        context.messageAdapter(rsp => WrappedBackendResponse(rsp))

      Behaviors.receiveMessage[Command] {
        case SomeCommand =>
          backend ! Backend.StartJob(Task(), backendResponseMapper)
          Behavior.same

        case WrappedBackendResponse(Backend.JobCompleted, result) =>
          // process response
          Behavior.same
      }
    }
  }
```

The Ask Pattern (1)




```
object Account:
  sealed trait Command
  case class GetBalance(replyTo: ActorRef[Value]) extends Command
  case class Value(amount: Double)
```

The Ask Pattern (2)

```
object Client:
  sealed trait Command
  case object PrintBalance extends Command
  private case class AdaptedResponse(value: Account.Value) extends Command
  private case class FailedResponse(msg: String)           extends Command

  def apply(account: ActorRef[Account.Command]): Behavior[Command] =
    Behaviors.receive[Command] {
      case (context, PrintBalance) =>
        given Timeout = 1.second
        context.ask(account, Account.GetBalance) {
          case Success(value) => AdaptedResponse(value)
          case Failure(_)     => FailedResponse("Got no reply from account")
        }
        Behaviors.same

      case (_, AdaptedResponse(Account.Value(balance))) => …

      case (_, FailedResponse(msg)) => …
    }
  }
```

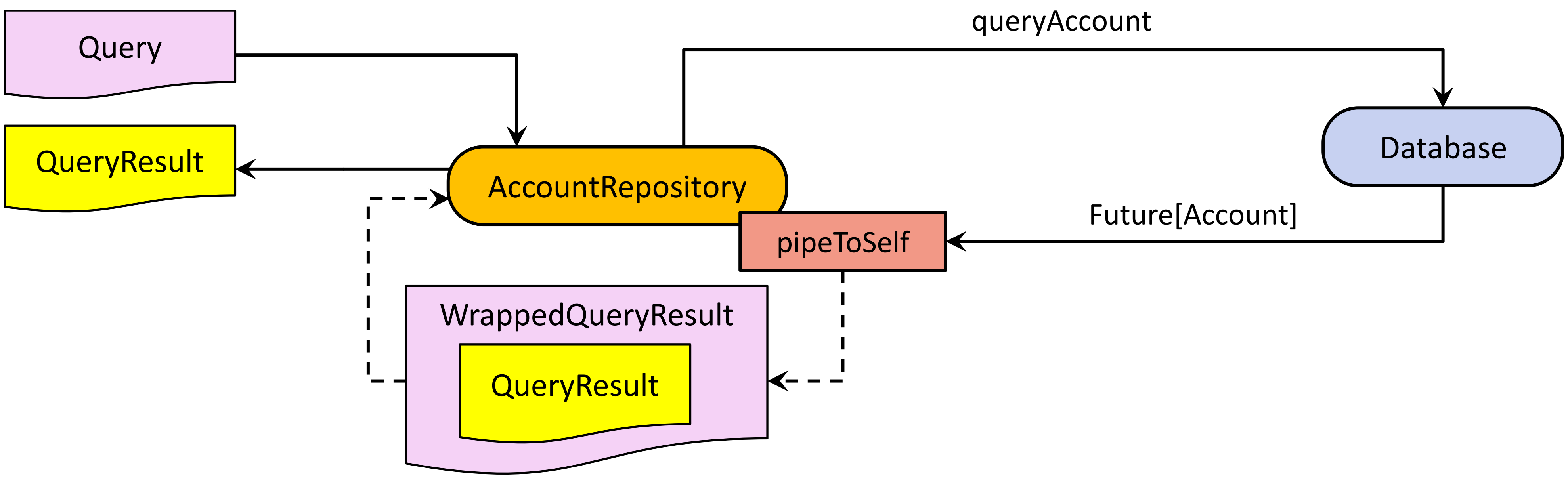
reference to temporary actor

The Ask Pattern (3)

- There is a version of ask that returns a `Future[Response]`.
  - Can be used to interact with an actor from the outside of the actor system

```
def askOutsideActor(account: ActorRef[Account.Command])
                   (using system: ActorSystem[_]): Unit =
  import akka.actor.typed.scaladsl.AskPattern._
  given Timeout = 1.seconds
  given ExecutionContext = system.executionContext
  val result: Future[Account.Value] = account.ask(Account.GetBalance)
  result.onComplete {
    case Success(Account.Value(balance)) => println(s"Balance : $balance")
    case Failure(_)                      => println(s"Got no reply from account")
  }
```

The Pipe Pattern (1)




```scala
final case class Account(balance: Double)
trait Database:
  def queryAccount(id: Int): Future[Account]
```

The Pipe Pattern (2)

```
object AccountRepository:
  sealed trait Command
  case class Query(accountId: Int, replyTo: ActorRef[QueryResult]) extends Command

  sealed trait QueryResult
  case class QuerySuccess(account: Account) extends QueryResult
  case class QueryFailure(ex: Throwable)    extends QueryResult
  private case class WrappedQueryResult(result: QueryResult,
                                        replyTo: ActorRef[QueryResult]) extends Command

  def apply(db: Database): Behavior[Command] =
    Behaviors.receive { (context, command) =>
      case Query(id, replyTo) =>
        val futureResult: Future[Account] = db.queryAccount(id)
        context.pipeToSelf(futureResult) {
          case Success(account) => WrappedQueryResult(QuerySuccess(account), replyTo)
          case Failure(ex)      => WrappedQueryResult(QueryFailure(ex), replyTo)
        }
        Behaviors.same

      case WrappedQueryResult(result, replyTo) =>
        replyTo ! result
        Behaviors.same
    }
  }
```

Scheduling Messages

- `system.scheduler` provides methods for sending messages to actors after some delay.
- Use `TimerScheduler` to send scheduled messages within an actor.
  - Timers are cancelled when an actor is restarted or stopped.

```
def apply() = Behaviors.withTimers(timer => active(timer))
private def active(timer: TimerScheduler[Command]): Behavior[Command] =
  Behaviors.receiveMessage {
    case StartTimer(after) =>
      timer.startSingleTimer(Timeout, after)
      Behaviors.same
    case StopTimer =>
      timer.cancel()
      Behaviors.same
    case Timeout =>
      println("Timer timed out");
      Behaviors.same
  }
```

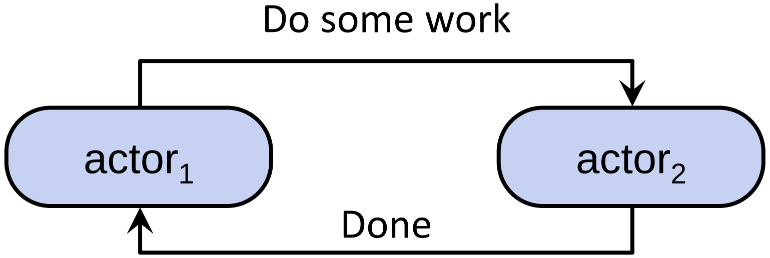
Designing Actor Systems

[Roestenburg et al., 2017]

[Odersky et al., 2015; Message Processing Semantics]

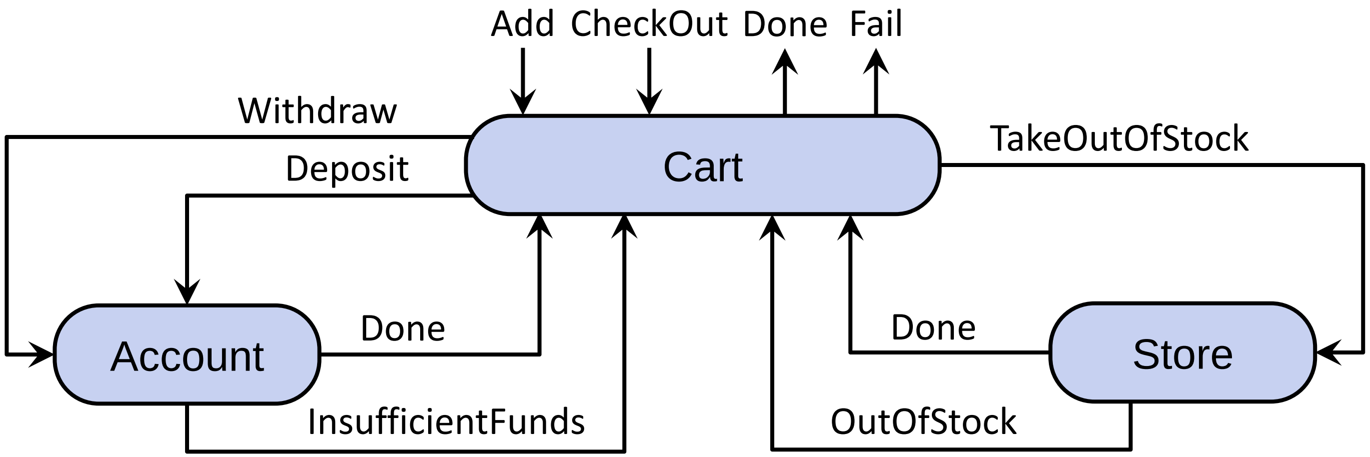
Actor Programming Model

- Actors are completely independent processing units.
  - Outside view: Actors run fully concurrently → Scalability
  - Inside view: Actors are single-threaded.
    - Messages are received sequentially.
    - Message processing need not be synchronized.
- Actor state and behavior can only be influenced by sending messages
  - Messages are sent via references (`ActorRef`).
  - Only one-way communication is used.



Example: Order System

- Requirements
  - `Account`: One can *deposit* and *withdraw* money to an account.
  - `Store`: Inventory can be managed by *delivering* products and taking *them out of stock*. Store contains only one product type.
  - `Cart`: Supports *adding* of products. When items are *checked-out*, the account is debited and the inventory is decreased.
- Design: Actors and Messages



Example: Order System – Account and Store

- Implementation of the actor class Account:

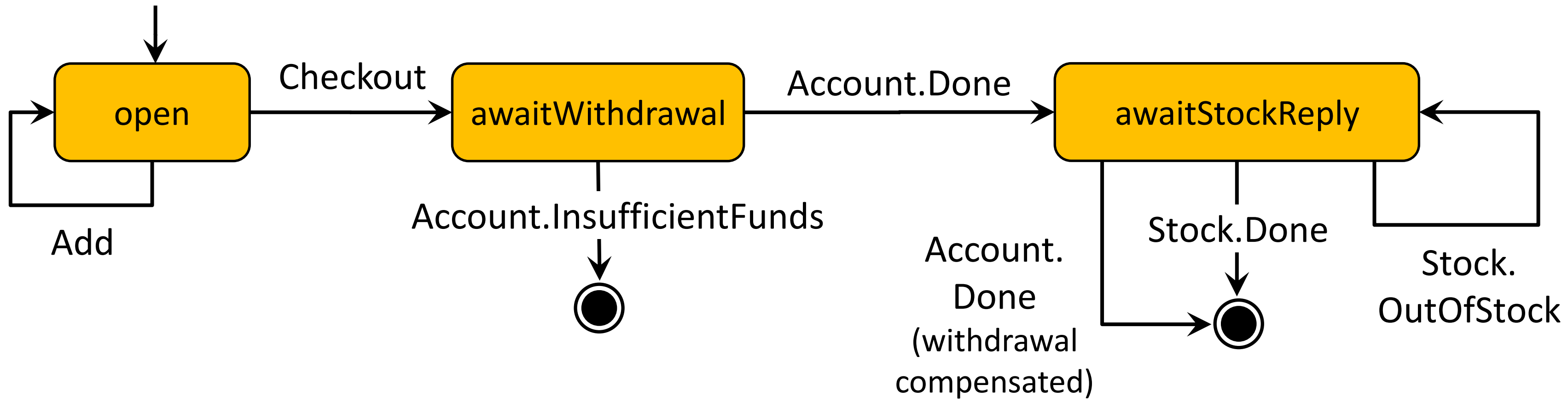
```
object Account:
  def apply(balance: Double) =
    Behaviors.receiveMessage[Account.Command] {
      case Deposit(amount, replyTo) =>
        replyTo ! Done
        apply(balance + amount)
      case Withdraw(amount, replyTo)
             if amount <= balance =>
         replyTo ! Done
         apply(balance - amount)
      case Withdraw(amount, replyTo) =>
        replyTo ! InsufficientFunds
        apply(balance)
    }
```

```
object Account:
  sealed trait Command
  case class Withdraw(
    amount: Double,
    replyTo: ActorRef[Reply]) extends Command
  case class Deposit(
    amount: Double,
    replyTo: ActorRef[Reply]) extends Command

  sealed trait Reply
  case object Done              extends Reply
  case object InsufficientFunds extends Reply
```

- Stock can be implemented in a similar way.

Example: Order System – State Diagram

- Modelling of `Cart`'s states and state transitions:
  - First cart accepts adding of new products (open).
  - When items are checked-out, the account is debited, and we wait for a successful withdrawal (`awaitWithdraw`).
  - Finally, we decrease the inventory and wait for a receipt from stock (`awaitStockReply`).
  - If product is out of stock, we have to compensate the withdrawal.



Example: Order System – Cart (1)

```scala
object Cart:
  sealed trait Command
  final case class  Add(quantity: Int) extends Command
  final case object Checkout           extends Command

  private final case class WrappedAccountReply(account: Account.Reply) extends Command
  private final case class WrappedStockReply  (stock:   Stock.Reply)   extends Command

  sealed trait Reply
  final case object Done   extends Reply
  final case object Failed extends Reply
```

```scala
def apply(account: ActorRef[Account.Command], stock: ActorRef[Stock.Command], unitPrice: Double,
          client: ActorRef[Reply]) = Behaviors.setup[Cart.Command] { context =>

  def open           (quantity: Int): Behavior[Cart.Command] = ???

  def awaitWithdrawal(quantity: Int): Behavior[Cart.Command] = ???

  def awaitStockReply(quantity: Int): Behavior[Cart.Command] = ???

  open(quantity = 0)
}
```

Example: Order System – Cart (2)

```
val accountReplyMapper: ActorRef[Account.Reply] = context.messageAdapter(
                                                       reply => WrappedAccountReply(reply))

def open(quantity: Int) =
  Behaviors.receiveMessage[Cart.Command] {
    case Add(q) =>
      open(quantity + q)
    case Checkout =>
      account ! Account.Withdraw(quantity * unitPrice, accountReplyMapper)
      awaitWithdrawal(quantity)
  }
```

Example: Order System – Cart (3)

```
val stockReplyMapper: ActorRef[Stock.Reply] = context.messageAdapter(
                                                reply => WrappedStockReply(reply))

def awaitWithdrawl(quantity: Int) =

  Behaviors.receiveMessage[Cart.Command] {

    case WrappedAccountReply(Account.Done) =>
      stock ! Stock.TakeOutOfStock(quantity, stockReplyMapper)
      awaitStockReply(quantity)

    case WrappedAccountReply(Account.InsufficientFunds) =>
      client ! Failed
      Behaviors.stopped
  }
```

Example: Order System – Cart (4)

```
val stockReplyMapper: ActorRef[Stock.Reply] = context.messageAdapter(
                                                 reply => WrappedStockReply(reply))

def awaitStockReply(quantity: Int) =
  Behaviors.receiveMessage[Cart.Command] {
    case WrappedStockReply(Stock.Done) =>
      client ! Done
      Behaviors.stopped
    case WrappedStockReply(Stock.OutOfStock) => // compensate withdrawal
      account ! Account.Deposit(quantity * unitPrice, accountReplyMapper)
      Behaviors.same
    case WrappedAccountReply(Account.Done) =>
      client ! Failed
      Behaviors.stopped
  }
```

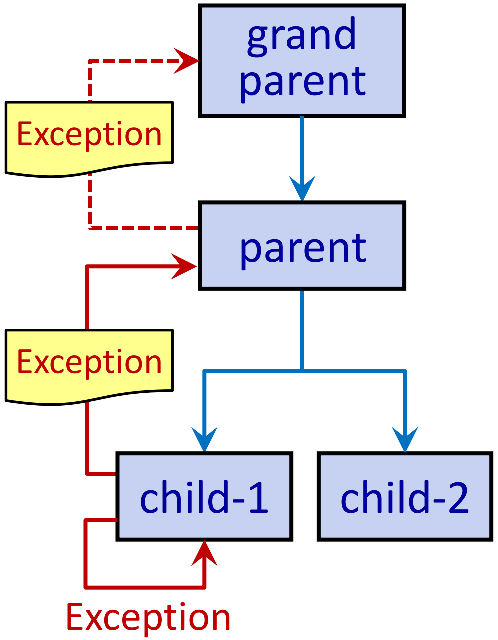
Actor Supervision

[Odersky et al., 2015; Failure Handling with Actors]

[Lopez-Sancho Abraham, 2023; Fault Tolerance]

Failure Handling in Asynchronous Systems

- In typical OO languages the caller of a method (= sender of the message) must handle exceptions.
- This is not applicable to actor systems.
  - Sending and processing a message is totally decoupled.
  - The sender may not exist anymore when the failure occurs.
- Actors are *resilient.*
  - Actor state and behavior is isolated → no other actor is affected by a failure.
  - Another actor – the **supervisor** – must handle failures.
- The parent actor acts as the supervisor of its children.
  - Unhandled exceptions are bundled to messages and sent to the supervisor.
  - The supervisor decides how to handle errors and how to deal with failed children.



Supervision Strategies

- Error types
  - *Validation errors*: Data sent to the actor is not valid
    → should be handled in the actor's business logic.
  - *Failures:* Something unexpected, beyond the control of the actor's business logic
    → should be handled outside the actor.
- The parent actor's *supervision strategy* specifies how to handle *failures*:
  - `resume`: The actor will continue with the next message.
  - `restart`: Immediately restarts actor with or without any limit on number of restart retries.
  - `restartWithBackoff`: Restart with delay.
  - `stop`: Permanently stop the actor (and all its children).
- Defaults
  - User actors are stopped by default.
  - When an actor is restarted its children get stopped.

Supervision Strategies – Implementation

- An existing behavior is *wrapped with a supervising behavior:*

```
Behaviors.supervise(Calculator())
         .onFailure[ArithmeticException](SupervisorStrategy.restart)
```

- To handle different exceptions with different strategies calls to `supervise` can be nested:

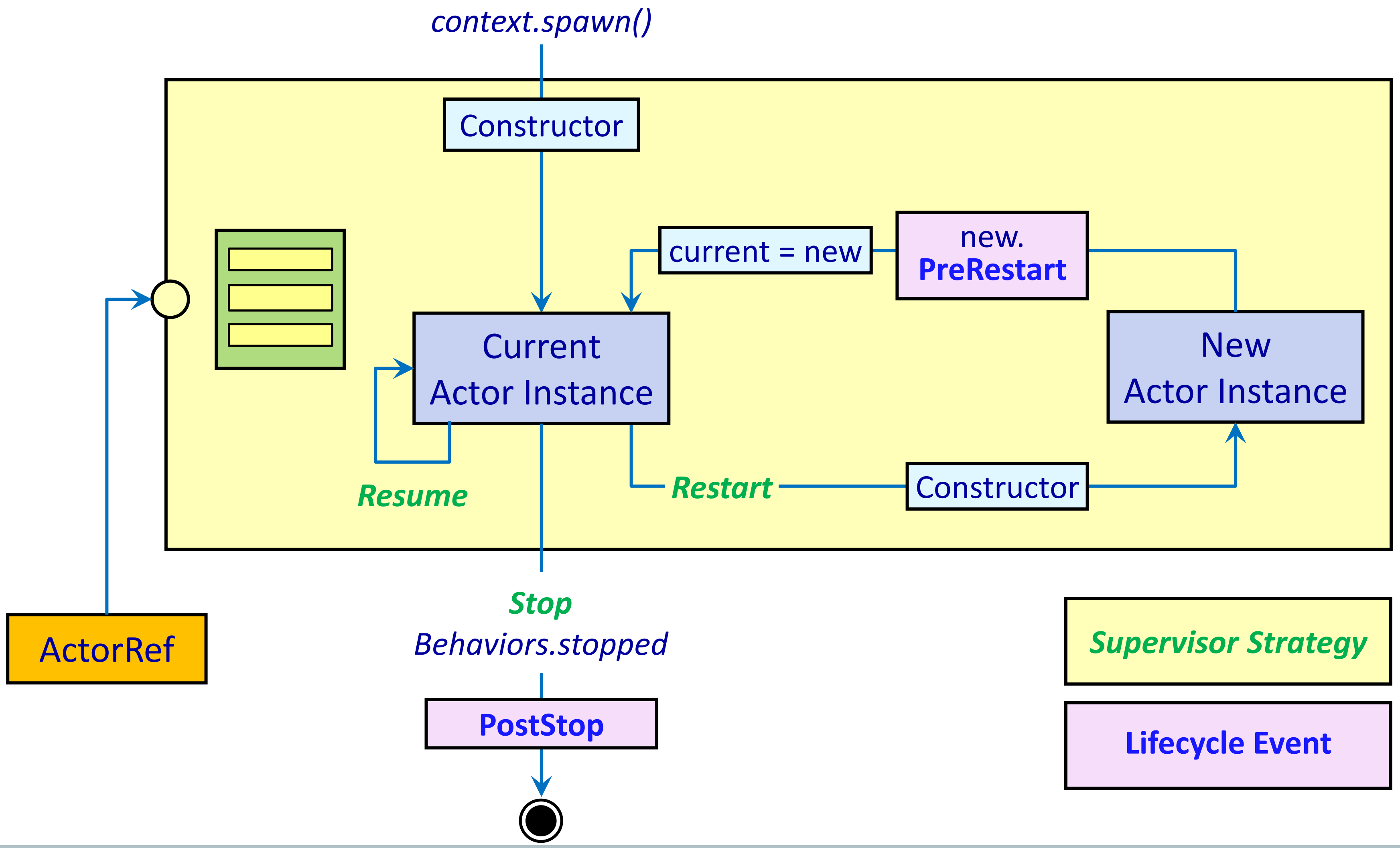
```
Behaviors.supervise(
  Behaviors.supervise(Calculator())
           .onFailure[ArithmeticException](SupervisorStrategy.restart))
           .onFailure[IllegalArgumentException](SupervisorStrategy.stop)
```

- By using parameters, more complicated strategies can be defined:

```
Behaviors.supervise(Calculator()).onFailure[ArithmeticException](
  SupervisorStrategy.restart
    .withLimit(maxNrOfRetries = 10, withinTimeRange = 10.seconds)
    .withStopChildren(false))
```

```
Behaviors.supervise(Calculator()).onFailure[ArithmeticException](
  SupervisorStrategy.restartWithBackoff(minBackoff = 1.second, maxBackoff = 10.seconds,
                                        randomFactor = 0.2))
```

Actor Lifecycle (1)



Actor Lifecycle (2)

- Semantics of *Restart*:
  - The *actor state is reset* by exchanging the actor object.
  - The *actor reference stays stable* during a restart.
- Message processing during Restart:
  - No messages will be processed between the failure and `PreRestart`.
  - The message that caused the error will not be delivered again.
- Lifecycle events

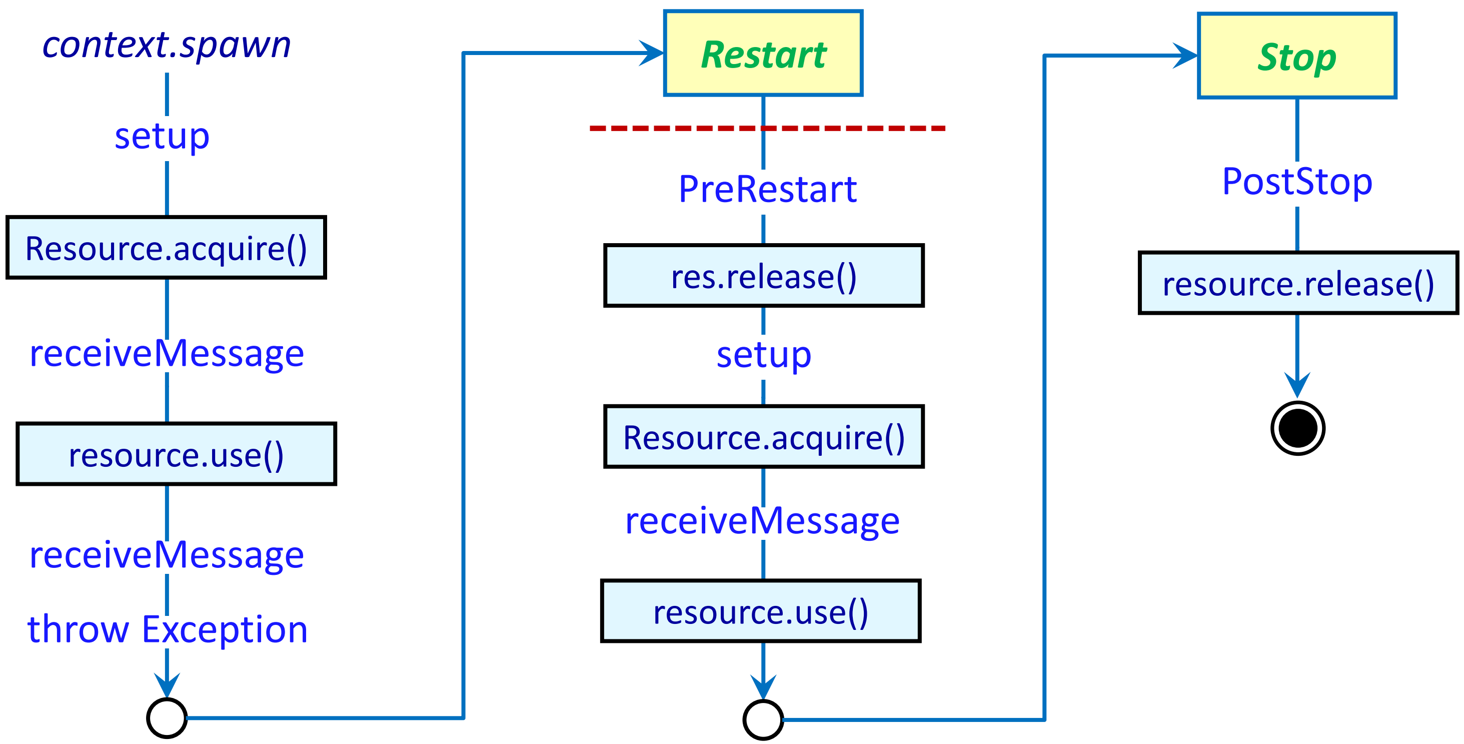
```
Behaviors.receiveSignal {
  case (context, PreRestart) => …
  case (context, PostStop)   => …
}
```

  - `PreRestart`: Sent immediately after new actor instance has been started.
  - `PostStop`: Sent before actor is ultimately terminated.
  - Can be used for resource management.

Resource Management with Lifecycle Methods (1)

```
object ResourceManager:
  def apply() =
    Behaviors.supervise[String](process())
             .onFailure[Exception](SupervisorStrategy.restart.withLimit(maxNrOfRetries = 1))

  private def process() =
    Behaviors.setup[String] { ctx =>
      val resource = Resource.acquire()
      Behaviors
        .receiveMessage[String] {
          case msg =>
            if (msg.isEmpty) throw new IllegalArgumentException("…")
            resource.use()
            Behaviors.same
        }
        .receiveSignal {
          case (_, signal) if signal == PreRestart || signal == PostStop =>
            resource.release()
            Behaviors.same
        }
    }
```

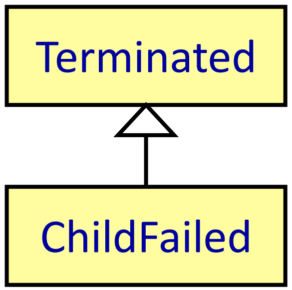
Resource Management with Lifecycle Methods (2)



Lifecycle Monitoring – Death Watch

- An actor reference proves that an actor is living or lived at an earlier point of time.
- Death Watch can be used to get informed
  - when an actor is stopped, or that an actor is already stopped.
  - when a child actor failed.
- `Terminated` event:
  - Sent by the actor, if it was alive when registering for death watch.
  - Otherwise, sent by the dispatcher.

```
def apply(): Behavior[Unit] =
  Behaviors.setup { context =>
    context.watch(someActorRef)
    Behaviors.receiveSignal {
      case (context, ChildFailed(actorRef, cause)) => ..
      case (context, Terminated(actorRef)) => …
    }
  }
```



Distributed Programming with Actors

[Prokopec, 2017; p. 277 ff.]

[Roestenburg et al., 2017; p. 118 – 146, 322 – 353]

[Odersky et al., 2015; Actors are Distributed]

Actors and Distributed Systems

- Impact of network communication to distributed programs:
  - Data is shared by value
    - Data must be serializable
    - Reference semantics must not be used
  - Lower bandwidth, higher latency
  - Network packets may be lost
- Many assumptions from a synchronous programming model does not hold for distributed programs.
- The actor model already provides solution for these problems:
  - Messages are immutable.
  - Actor communication is asynchronous, and unreliable.
  - Actors are isolated → Location transparency.
- Actors are designed for distributed systems.

Remote Actors - Configuration

- To make an actor system reachable from actor systems located on different hosts, `application.conf` has to be extended:

```
akka {
  actor {
    provider = remote
    allow-java-serialization = on
  }
  remote {
    artery {
      transport = tcp # See Selecting a transport below
      canonical.hostname = <host>
      canonical.port = <port>
    }
  }
}
```

  - *provider=remote:* actor references are remote aware
  - *host/port:* machine/port the actor system will listen on
  - Different serialization mechanisms are supported: Java, Jackson, Protobuf
- No changes to the actor implementation are necessary.

Looking up Remote Actors

- Remote actors are addressable by the following *path* pattern:

```
akka.<protocol>://<actor system>@<hostname>:<port>/<actor path>
```

- One can obtain an `ActorSelection` to the remote actor and send messages to it:

```
val remoteCalc = context.actorSelection(
                   s"akka.tcp://CalculatorSystem@$host:$port/user/calculator"
remoteCalc ! Add(100)
```

- To acquire an `ActorRef` for a selection one needs to send the build-in `Identify` message to the actor:

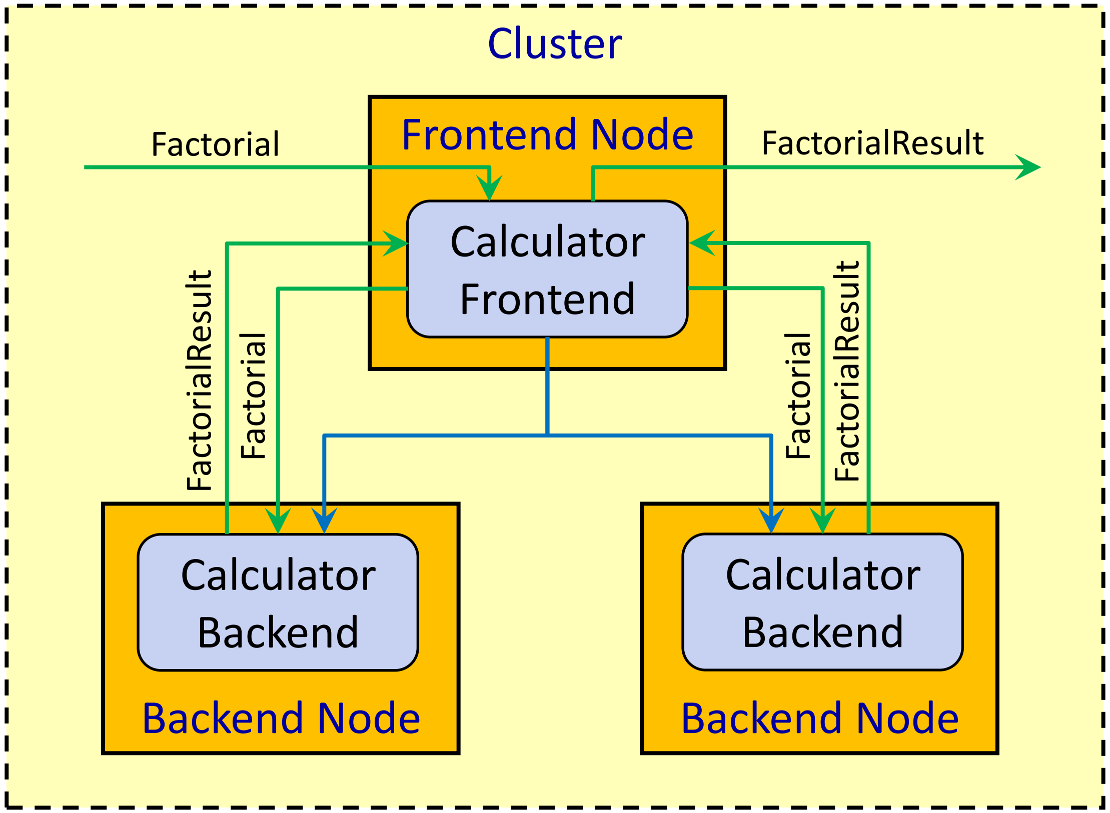
```
class MyActor(remoteCalcPath: String) extends Actor:
  context.actorSelection(remoteCalcPath) ! Identify(0)
  def receive = {
    case ActorIdentity(0, Some(remoteCalcRef)) => remoteCalcRef ! Add(100)
    case ActorIdentity(0, None) => println("Remote actor unreachable")
  }
```

Cluster

- A cluster is a set of nodes (actor systems) that collaborate on a common task.
  - All nodes must know from each other.
- Creation of a cluster:
  - A single node declares itself as a cluster.
  - Other nodes can join the cluster.
    - All existing members are informed.
    - When all agree, the new node becomes part of the cluster.
- Information in the cluster is spread using a gossip protocol.
  - There is no central coordinator.
  - Nodes send information to some of its neighbors.

Cluster Example

- Use case: CPU-intensive calculations are distributed to a set of compute nodes.



Building up a Cluster – Configuration

- Frontend node:

```
akka {
  actor.provider = cluster
  remote.artery.canonical.hostname = <host>
  remote.artery.canonical.port = <port>
  cluster.roles = ["frontend"]
}
```

- Backend node:

```
akka {
  actor.provider = cluster
  remote.artery.canonical.hostname = <host>
  remote.artery.canonical.port = 0
  cluster.roles = ["backend"]
}
```

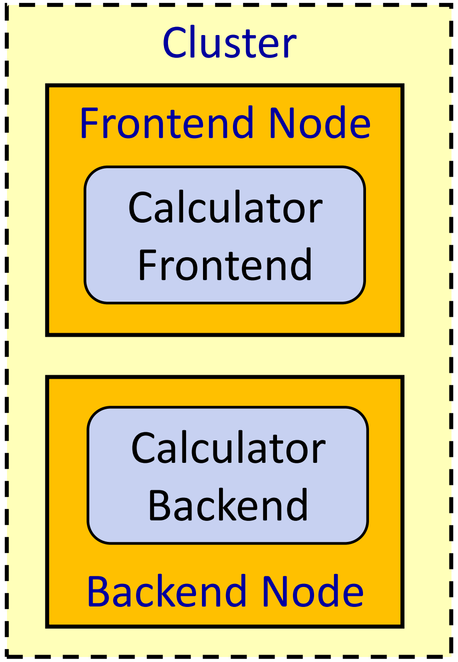
- `actor.provider=cluster` will trigger the loading of the `Cluster` extension
- `port` = 0 → free random port is chosen, if omitted → default port 25520 is assigned.
- *Roles* can be assigned to nodes that perform the same function.
  - The role of a node is part of the membership information.

Building up a Cluster – Joining the Cluster

- An actor system can be added to the cluster using `cluster.join`.
  - The first node adds itself (`cluster.selfAddress`) to the cluster.
  - Further nodes can be registered be sending a join request to any node of the cluster.

```
class CalculatorFrontend extends Actor:
  val cluster = Cluster(context.system)
  cluster.join(cluster.selfAddress)
  …
```

```
class CalculatorBackend extends Actor:
  val cluster = Cluster(context.system)
  cluster.join(Address("akka", context.system.name,
                       frontEndHost, frontEndPort)
  …
```



Building up a Cluster – Seed Nodes

- A new node may not know where cluster nodes are located.
- For this reason *seed nodes* can be configured:

```
akka {
  cluster {
    seed-nodes = ["akka.tcp://ClusterSystem@<host-1>:<port-1>",
                  "akka.tcp://ClusterSystem@<host-2>:<port-2>"]
  }
}
```

- When a new node is started up, one of the seed nodes is contacted automatically.
  - The seed node propagates the membership information.
  - No manual join (`cluster.join`) is necessary.

Creating a Cluster – Cluster Events

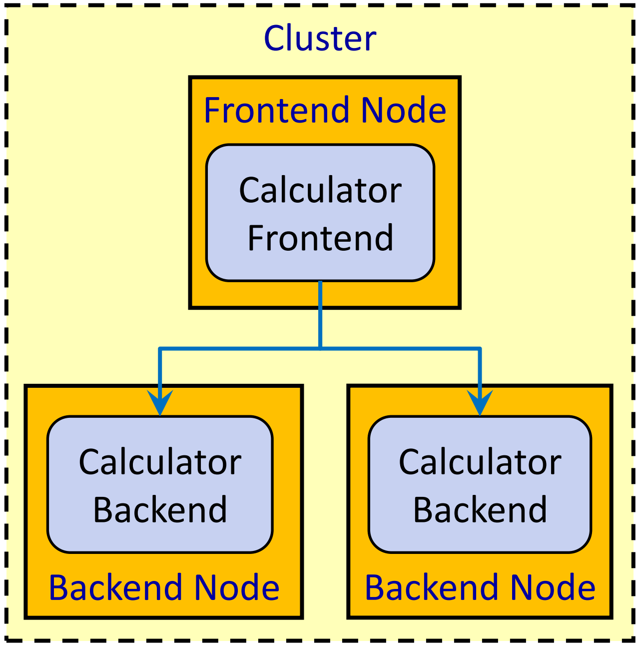
- We subscribe to cluster-related events using `cluster.subscribe`.
  - We get informed when new nodes join (`MemberUp`), leave the cluster (`MemberRemove`), or become unreachable (`UnreachableMember`).
  - The cluster gives us information about its current members (`CurrentClusterState`).

```
class CalculatorFrontend extends Actor:
  // join the cluster
  cluster.subscribe(self, classOf[ClusterEvent.MemberUp],
                          classOf[ClusterEvent.MemberRemoved])
  override def postStop(): Unit = cluster.unsubscribe(self)
  def receive =
    case state: ClusterEvent.CurrentClusterState =>
      state.members filter { _.status = MemberStatus.Up } foreach doSomeThing
    case ClusterEvent.MemberUp(member) =>
      if (member.address != cluster.selfAddress) doSomeThing(member)
    case ClusterEvent.MemberRemoved(member) => …
```

Cluster – Building up the Actor Hierarchy

- When new backend nodes join the cluster, the frontend node creates remote worker actors in the backend node.
- The resulting actor hierarchy spans across multiple nodes.

```
class CalculatorFrontend extends Actor:
  private def addBackendActor(node: Member): Unit =
    context.actorOf(
      Props[CalculatorBackend]()
        .withDeploy(Deploy(
          scope = RemoteScope(node.address))))

  def receive =
    case state: CurrentClusterState =>
      state.members filter (_.hasRole("backend"))
                    foreach addBackendActor
    case MemberUp(member) =>
      if (member.hasRole("backend"))
        addBackendActor(member)
```



Cluster – Simple Routing of Messages

- We use a simple round robin approach to distribute compute jobs to the backend nodes.

```
class CalculatorFrontend extends Actor
  var next = 0
  def receive =
    case Factorial(number) if context.children.isEmpty =>
      sender() ! Failed("Service unavailable, try again later.")
    case job @ Factorial(_) =>
      val children = context.children.toSeq
      children(next) forward job
      next = (next + 1) % children.size
```

```
class BackendNode extends Actor:
  def factorial(n: Int): BigInt = ...
  def receive = {
    case Factorial(n) =>
      Future { factorial(n)) } map { result => FactorialResult(n, result) }
                                pipeTo sender()
```

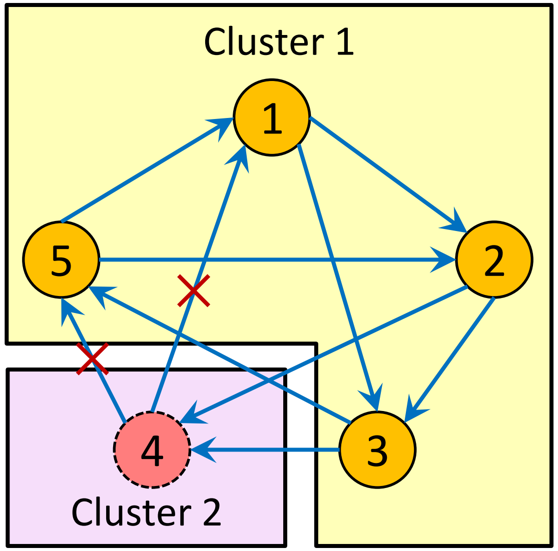
Cluster – Graceful Shutdown

- When the frontend node is stopped, the backend nodes stop themselves.
  - The backend nodes register the frontend for death watch.
  - It is guaranteed that `Terminated` will be delivered.
  - No messages will arrive after `Terminated`.

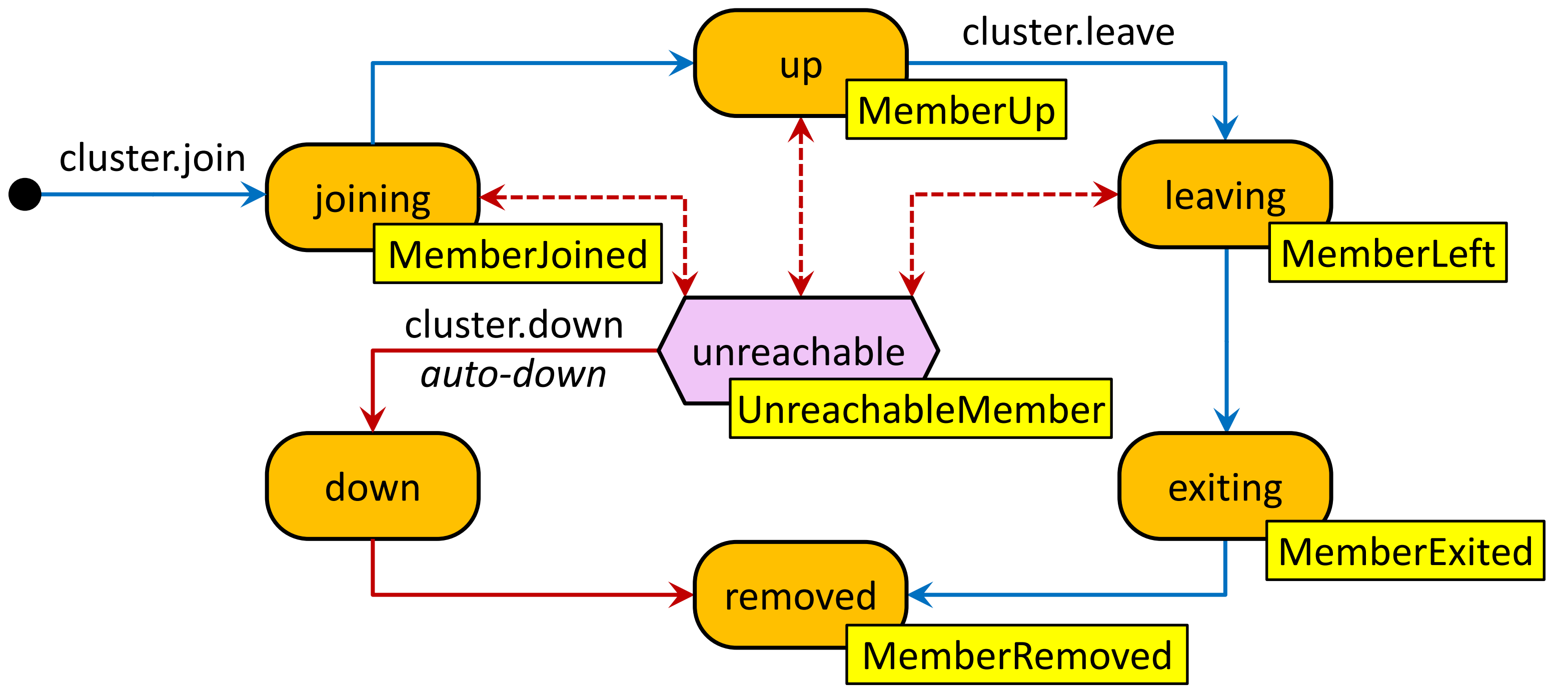
```
class CalculatorBackend extends Actor
  def receive =
    case MemberUp(member) =>
      if (member.hasRole("frontend")) {
        val path = RootActorPath(member.address) / "user" / "frontend"
        context.actorSelection(path) ! Identify("frontend")
      }
    case ActorIdentity("frontend", Some(frontend)) => context.watch(frontend)
    case ActorIdentity("frontend", None)           => context.stop(self)
    case Terminated(_)                              => context.stop(self)
```

Cluster Failure Detection

- There must be a universal agreement on the members of a cluster.
  - Unreachable nodes must be detected.
  - If a node is unreachable for one member it is considered unreachable for all.
- Information propagation
  - Message concerning cluster state is sent to some random nodes (gossip protocol).
- Failure detection
  - Every node sends heartbeat messages to its neighbors.
  - If no heartbeats arrive for some time, the node is consider unreachable and will be excluded from the cluster.



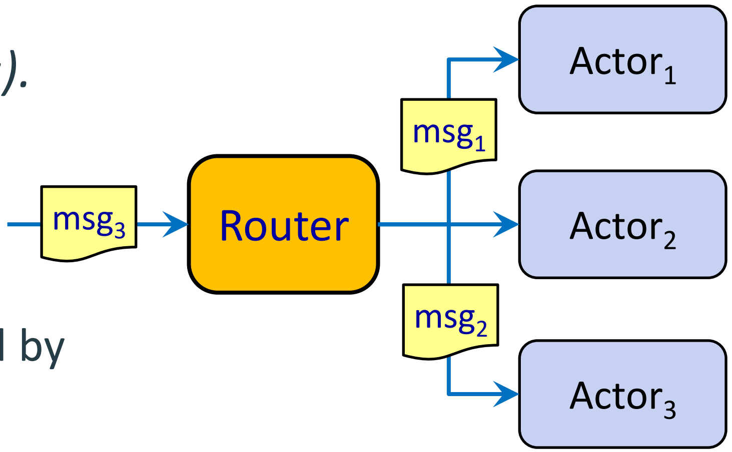
Cluster Lifecycle



- *joining/leaving/exiting/down*: Wait until nodes have seen that the node is joining/leaving/shut down *(gossip convergence)* and then transition to *up/exiting/removed*.
- A node can return to its original state after it was unreachable.
- *unreachable* → *down*: Can be done manually or automatically (configuration setting).

Routers

- *Routers* distribute messages to destination actors *(routees).*
- Router actors come in two different flavors:
  - *Pool:* The router creates and supervises its child actors.
  - *Group:* Routees are created externally. Routees are specified by their paths (`rootees.paths`).



- Supported routing strategies:
  - *RoundRobin*
  - *Random*
  - *SmallestMailbox:* Send messages to the actor with the fewest messages.
  - *ScatterGatherFirstCompleted:* Send messages to all routees, use the first response.
  - *ConsistentHashing:* Message is associated with a key that is hashed → messages with same key are routed to the same actor.
  - *Balancing Pool:* Routees share one mailbox.

Routers – Implementation

- Router Configuration

Pool:

```
akka.actor.deployment {
  /manager/myrouter {
    router = round-robin-pool
    nr-of-instances = 3
  }
}
```

Group:

```
akka.actor.deployment {
  /manager/myrouter {
    router = round-robin-group
    routees.paths = ["/user/worker1",
                     "/user/worker2"]
  }
}
```

- Creation of the router actor

```
class Manager extends Actor
  val router = context.actorOf(FromConfig.props(Props[Worker]()), "myrouter")
  def receive =
    case msg => router ! msg
```

Cluster Aware Routes

- Routers that need no access to the routees' mailboxes can also be used in cluster scenarios (*RoundRobin*, *ConsistentHashing*, …).
- These routers can be made aware of member nodes in the cluster.
  - They can deploy new routees to cluster nodes or look them up in cluster nodes.
  - When nodes are removed from the cluster, the are automatically unregistered from the router.
- Configuration:

```
akka.actor.deployment {
  /frontend/backendRouter {
    router = round-robin-pool
    cluster {
      enabled = on
      max-total-nr-of-instances = 10   // max no. of instances in total
      max-nr-of-instances-per-node = 1 // max no. of instances per node
      allow-local-routees = on         // allow routees on the router’s node
      use-role = "backend"             // only consider nodes with this role
    }
  }
}
```

Cluster Metrics

- The cluster metrics extension
  - collects system metrics (CPU usage, memory load) of cluster nodes and
  - publishes this data to the system event bus.
- The *AdaptiveLoadBalancing* router performs load balancing of messages to cluster nodes based on cluster metrics data.
  - Routees are selected randomly with probabilities derived from usage of system resources.

```
akka.actor.deployment {
  /frontend/backendRouter {
    router = cluster-metrics-adaptive-group
    # metrics-selector = heap  // heap capacity
    # metrics-selector = load  // system load over the past minute (linux top)
    # metrics-selector = cpu   // CPU utilization in percent
    metrics-selector = mix     // combines heap, cpu, and load
    routees.paths = ["/user/backend"]
    cluster { … }
  }
}
```