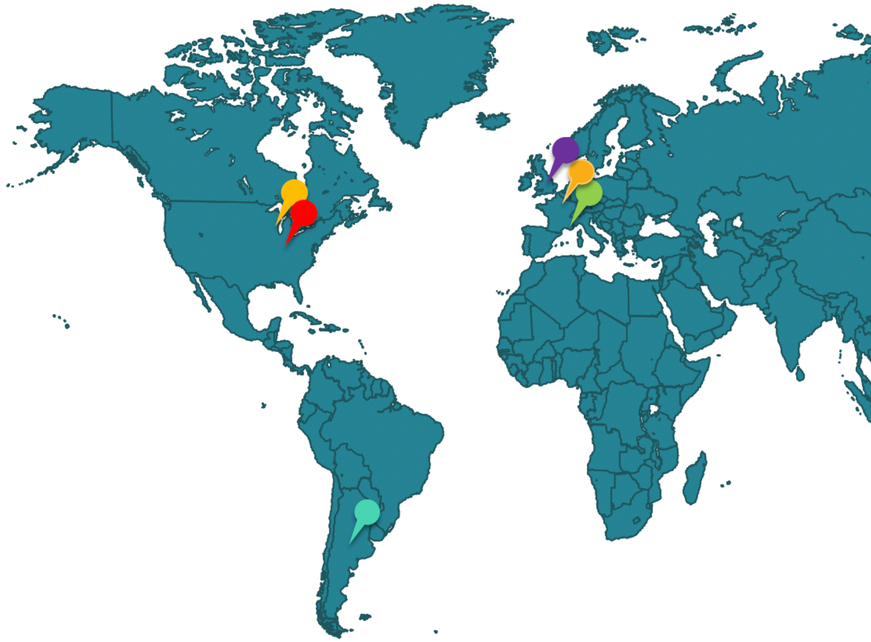
Actor Programming

For the Cloud

Who am I?



Gustavo Petri

[2006 - 2010] PhD. Sophia Antipolis, *INRIA*, France

[2011 - 2012] PostDoc *DePaul University*, Chicago

[2012 - 2015] Visiting A. Professor *Purdue University*, Indiana

[2015 - 2018] Assistant Professor *IRIF* - Universite Paris Diderot, Paris

[2018 - ….] Formal Methods Researcher - *ARM Research*, Cambridge

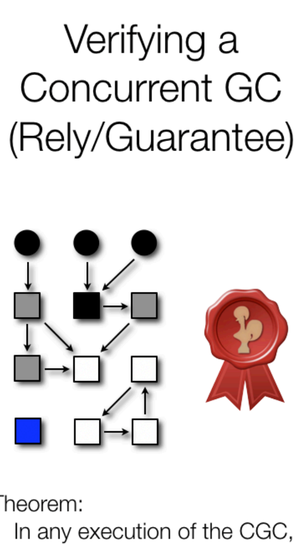
Who am I?

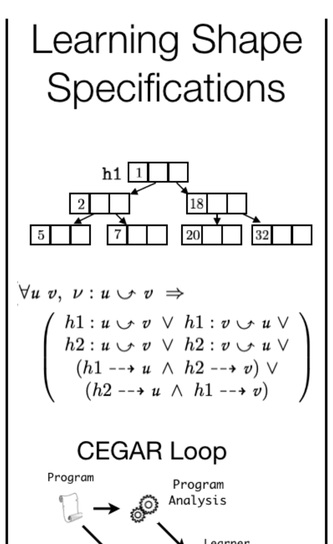
Topics

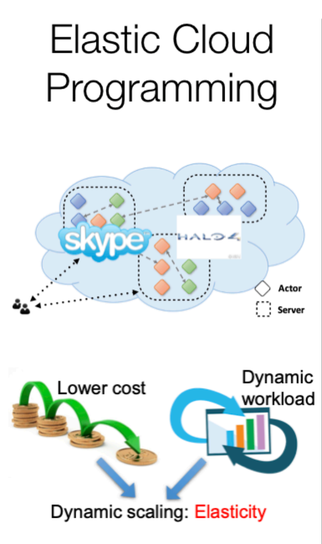
- Relaxed Memory Models
- Concurrent Program Verification
- Distributed Systems Verification
- Models for Distributed Computir
- and now … Security

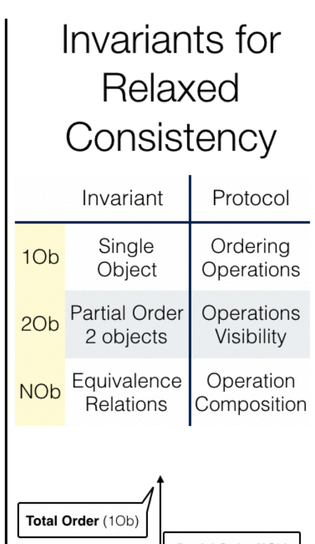
Tools

- Formal Semantics
- Formal Methods
- Programming Languages









WHENCE THIS COURSE?

- The way we write programs is changing
  - Move to mobile applications
  - Data availability
  - IoT, embedded, etc.
- Programming for the cloud
  - Distributed Systems
  - Elasticity
  - Application Architecture

The Reactive Manifesto (/Microservices)

The Reactive Manifesto

The Reactive Manifesto

Published on September 16 2014. (v2.0)

Organisations working in disparate domains are independently discovering patterns for building software that look the same. These systems are more robust, more resilient, more flexible and better positioned to meet modern demands.

These changes are happening because application requirements have changed dramatically in recent years. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and 100% uptime. Data is measured in Petabytes. Today's demands are simply not met by yesterday's software architectures.

We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems.

Systems built as Reactive Systems are more flexible, loosely-coupled and scalable. This makes ier to develop and amenable to change. They are significantly more tolerant of failure occur they meet it with elegance rather than disaster. Reactive Systems ffective interactive feedback.

The Reactive Manifesto

System Requirements and Architecture for Modern Distributed Applications

Published on September 16 2014. (v2.0)

Organisations working in disparate domains are independently discovering patte
software that look the same. These systems are more robust, more resilient, more flexible and
better positioned to meet modern demands.

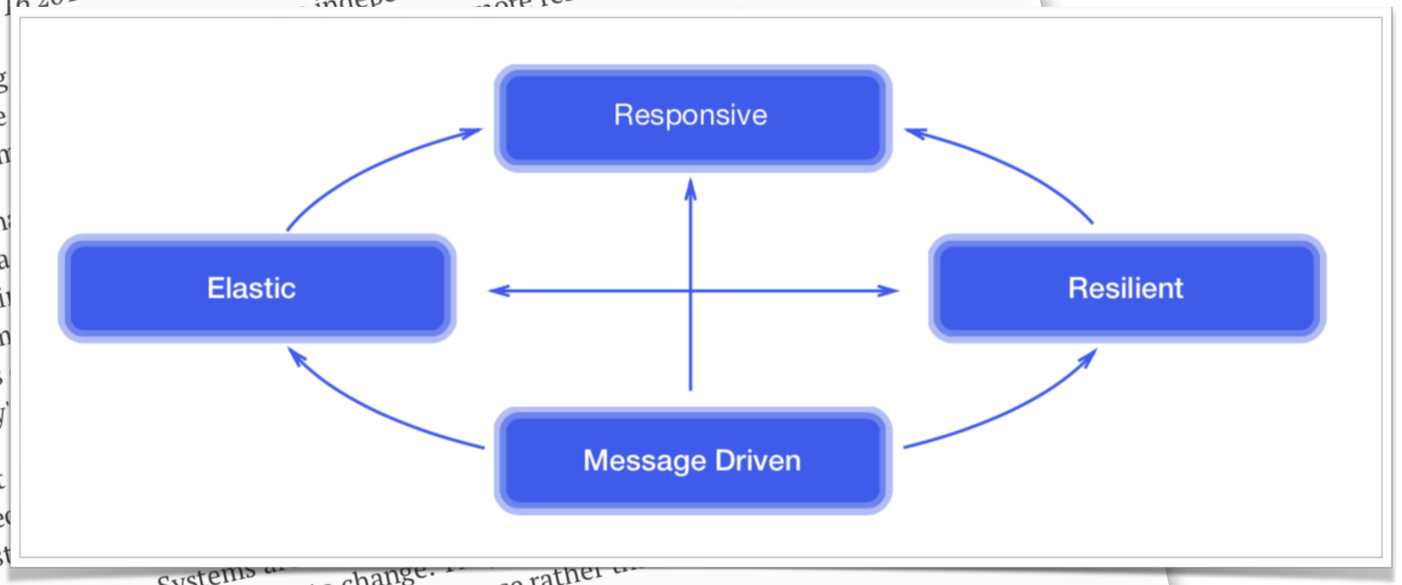
These changes are happening because application requirements have changed dramatically in recent years. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users expect millisecond response times and 100% uptime. Data is measured in Petabytes. Today's demands are simply not met by yesterday's software architectures.

We believe that a coherent approach to systems architecture is needed, and we believe that all necessary aspects are already recognised individually: we want systems that are Responsive, Resilient, Elastic and Message Driven. We call these Reactive Systems.

Systems built as Reactive Systems are more flexible, loosely-coupled and scalable. This makes
ier to develop and amenable to change. They are significantly more tolerant of failure
occur they meet it with elegance rather than disaster. Reactive Systems
ffective interactive feedback.

The Reactive Manifesto

System Requirements and Architecture for Modern Distributed Applications



Infrastructure Through Time

Typical infrastructure for a single application

Infrastructure Through Time

Typical infrastructure for a single application

1970

Infrastructure Through Time

Typical infrastructure for a single application

1970

1980

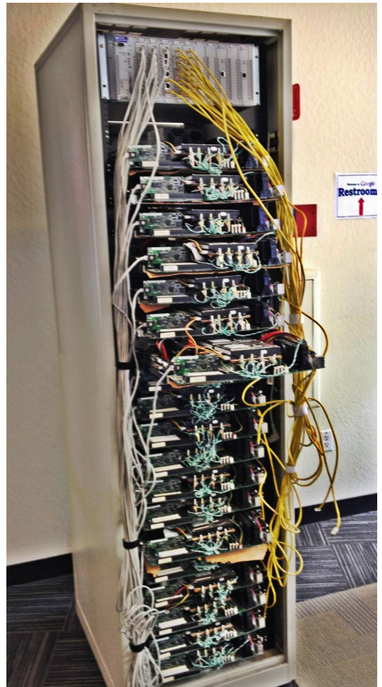
Infrastructure Through Time

Typical infrastructure for a single application

1970

1980

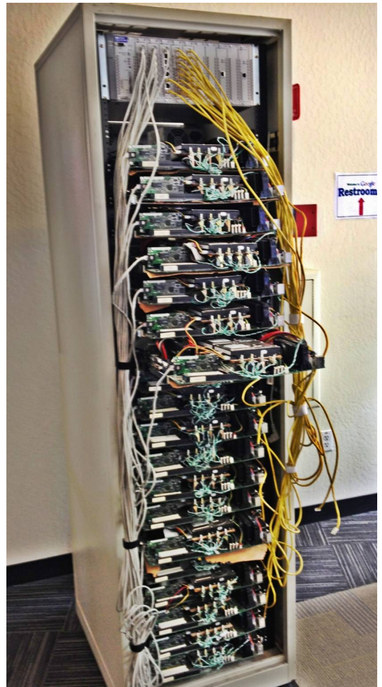
1990



Infrastructure Through Time

Typical infrastructure for a single application

1970 1980 1990 2010



The Reactive Manifesto: Responsive

Responsive: The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.

The Reactive Manifesto: Responsive

Responsive: The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.

The Reactive Manifesto: Responsive

Responsive: The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.

The Reactive Manifesto: Responsive

Responsive: The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.

The Reactive Manifesto: Responsive

Responsive: The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.

The Reactive Manifesto: Responsive

Responsive: The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.

The Reactive Manifesto: Resilient

Resilient: The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

The Reactive Manifesto: Resilient

Resilient: The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

The Reactive Manifesto: Resilient

Resilient: The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

The Reactive Manifesto: Resilient

Resilient: The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

The Reactive Manifesto: Elastic

Elastic: The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

The Reactive Manifesto: Elastic

Elastic: The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

The Reactive Manifesto: Elastic

Elastic: The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.

The Reactive Manifesto: Message Driven

Message Driven: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

The Reactive Manifesto: Message Driven

Message Driven: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

The Reactive Manifesto: Message Driven

Message Driven: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

The Reactive Manifesto: Message Driven

Message Driven: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

The Reactive Manifesto: Message Driven

Message Driven: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

The Reactive Manifesto: Message Driven

Message Driven: Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing e icity, and flow control by sh and applying back-pressu neans of communicat ork with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

How do we program for The Reactive Manifesto?

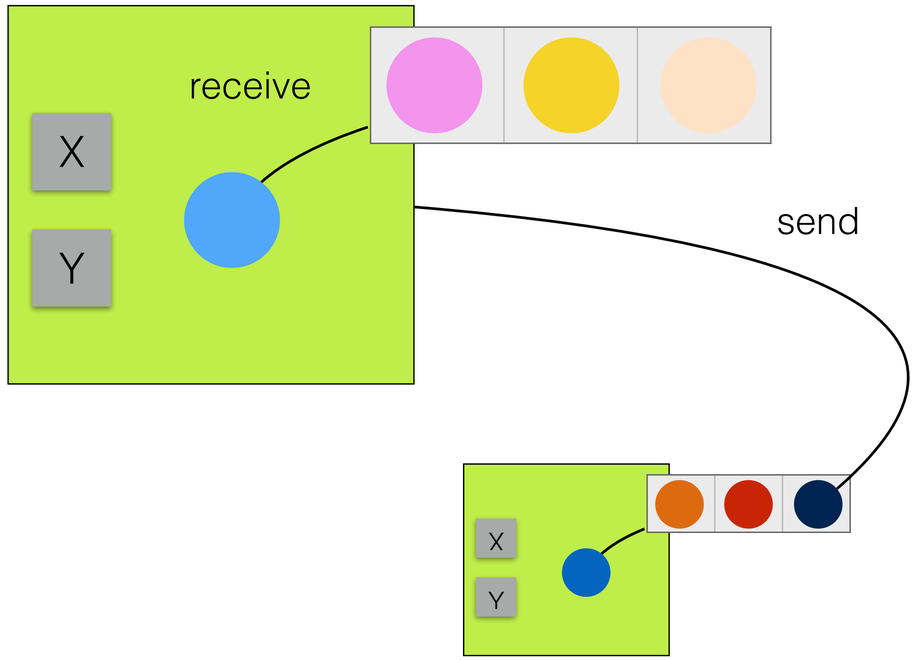
Actor Programming

Actor Execution Model

- Outline
  - What is an Actor
  - Actors vs. OO + Threads
  - Why Actors
  - Why Actors for Distributed Computing

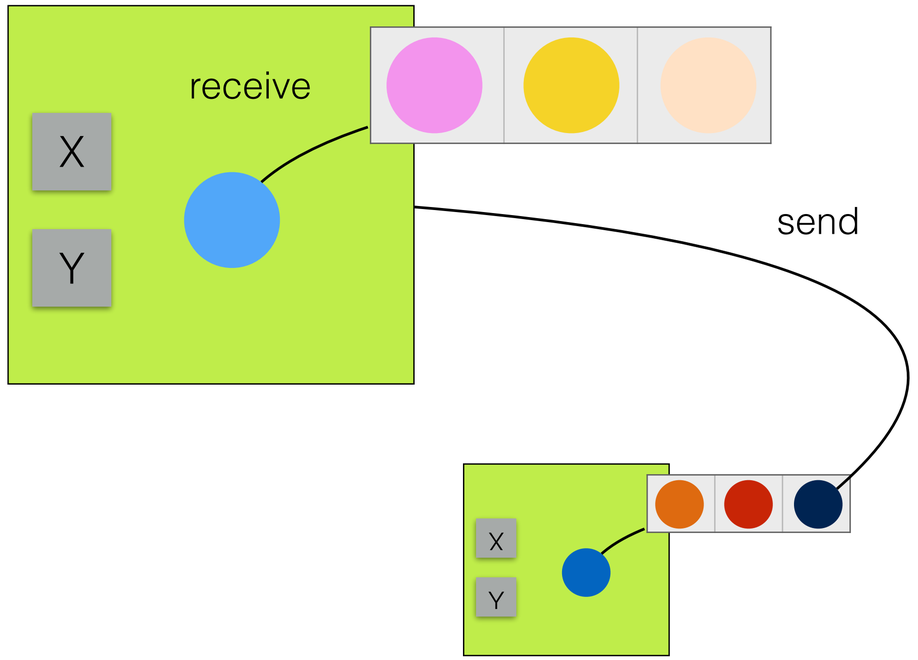
What is an Actor

- Focus on Distribution (and parallelism)
- Sometimes Stateful
- Single-threaded
- Asynchronous
- Message Passing
- Messages go into a Mailbox



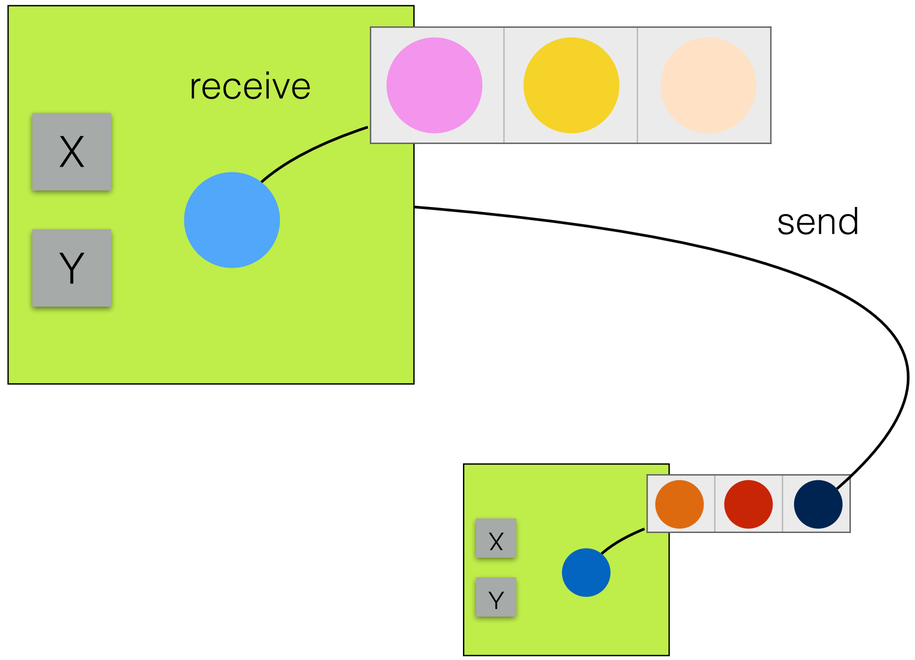
What is an Actor

- Actor Reference
  - Information Hiding
  - Location Transparency
  - Naming: ActorPath (public addresses)
- Local Actor State
  - No access from other actors
  - Interaction only through messages



What is an Actor

- Behavior
  - Represented through message/response
- Mailbox: FIFO? Priorities? Transactions?
- Fault Tolerance: Supervision / Monitoring
- Scalability
- Elasticity
- …



Actors vs. OO

	OO Programming	Actor Programming

ACTORS VS. OO

	OO Programming	Actor Programming
State	Public fields accessible to other objects	Encapsulated inside each actor

ACTORS VS. OO

	OO Programming	Actor Programming
State	Public fields accessible to other objects	Encapsulated inside each actor
Communication	Method invocation	Message passing

ACTORS VS. OO

	OO Programming	Actor Programming
State	Public fields accessible to other objects	Encapsulated inside each actor
Communication	Method invocation	Message passing
Scalability	Multi-threading Shared-Memory	Actors are independent processes

Thought Experiment in OO

- Consider developing an application that maintains state at runtime
- The application receives requests from clients on the internet

Thought Experiment in OO

- Consider developing an application that maintains state at runtime
- The application receives requests from clients on the internet
- But the workloads are unknown

Thought Experiment in OO

- Consider developing an application that maintains state at runtime
- The application receives requests from clients on the internet
- But the workloads are unknown
- Can we buy more machines to accommodate the workload?

Thought Experiment in OO

- Consider developing an application that maintains state at runtime
- The application receives requests from clients on the internet
- But the workloads are unknown
- Can we buy more machines to accommodate the workload?
- What happens if one machine fails?

Thought Experiment in OO

- Consider developing an application that maintains state at runtime
- The application receives requests from clients on the internet
- But the workloads are unknown
- Can we buy more machines to accommodate the workload?
- What happens if one machine fails?
- How can we architect this application?

OO + Threads

- What is the underlying programming language Memory Model?

```
TSO.x = 0;
TSO.y = 0;
TSO.r1 = 0;
TSO.r2 = 0;

Thread wrXrdY =
    new Thread() {
        public void run() {
            // wait 1 millisecond
            try { Thread.sleep(1); }
            catch (Exception e) {System.exit(1);}
            // write x read y
            TSO.x = 1;
            TSO.r1 = TSO.y;
        }
};

Thread wrYrdX =
    new Thread() {
        public void run() {
            // wait 1 millisecond
            try { Thread.sleep(1); }
            catch (Exception e) {System.exit(1);}
            // write y read x
            TSO.y = 1;
            TSO.r2 = TSO.x;
```

$$x = y = 0$$

$$\begin{array}{lcl} x := 1; & \| & y := 1; \\ r_0 = y & \| & r_1 = x \end{array}$$

$$r_0 = r_1 = 0$$

Demo

Actors as Concurrency Control

- Actors enforce a crude [and rather specific] form of Concurrency Control
  - Each actor's state is manipulated by strictly one process [conceptually]
  - No races are possible at the individual actor level
- Well designed actors can fail independently
  - "Let it crash" model (later)

Programming with Actors

- How can we structure the application into Actors?

Programming with Actors

- How can we structure the application into Actors?
  - Each actor performs a small and specific function

Programming with Actors

- How can we structure the application into Actors?
  - Each actor performs a small and specific function
  - Actors are created and terminated upon request

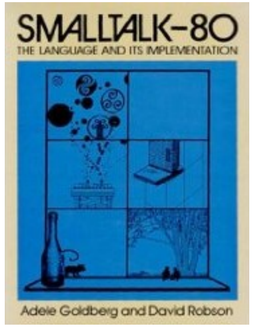
Programming with Actors

- How can we structure the application into Actors?
  - Each actor performs a small and specific function
  - Actors are created and terminated upon request
  - Actors are supervised, and faults are treated where necessary

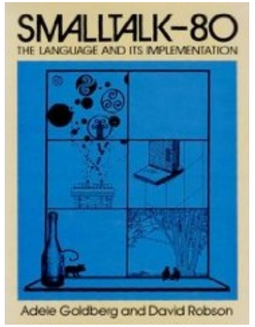
Programming with Actors

- How can we structure the application into Actors?
  - Each actor performs a small and specific function
  - Actors are created and terminated upon request
  - Actors are supervised, and faults are treated where necessary
  - Some long-lived actors serve as services entry points

A NOTE ON SMALLTALK

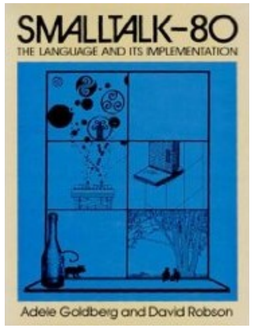


A NOTE ON SMALLTALK



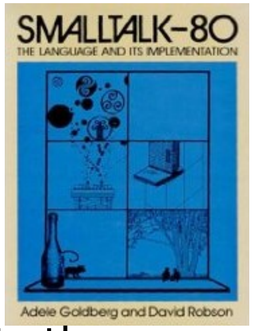
- Created in the ‘70 by Alan Kay and other
  - Introduces Objects “abstract representation of a concrete entity”
  - Everything is an object
  - Interaction with objects is through message/response
  - Methods bind messages to functionality
  - Method binding is dynamic

A NOTE ON SMALLTALK



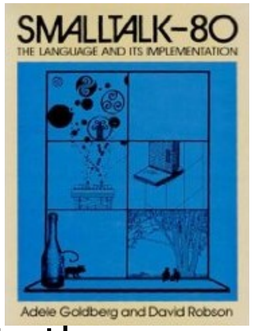
- Created in the ‘70 by Alan Kay and other
  - Introduces Objects “abstract representation of a concrete entity”
  - Everything is an object
  - Interaction with objects is through message/response
  - Methods bind messages to functionality
  - Method binding is dynamic
- Classes define the representation of objects
- Class Protocol (messages that the object accepts — API)
- Inheritance
  - Code reuse
  - Generalization
  - Specializatoin

A NOTE ON SMALLTALK



- Created in the ‘70 by Alan Kay and other
  - Introduces Objects “abstract representation of a concrete entity”
  - Everything is an object
  - Interaction with objects is through message/response
  - Methods bind messages to functionality
  - Method binding is dynamic
- Classes define the representation of objects
- Class Protocol (messages that the object accepts — API)
- Inheritance
  - Code reuse
  - Generalization
  - Specializatoin
- Purist Object Orientation
  - Everything is an object
  - No conditionals: expressed as messages ifTrue, ifFalse
- The intent was to construct libraries for all expectable objects out there
- Inspired the OO languages we know and love :-)

A NOTE ON SMALLTALK



- Created in the ‘70 by Alan Kay and other
  - Introduces Objects “abstract representation of a concrete entit
  - Everything is ar
  - Interaction with through message/response
  - Methods bind messages to functionality
  - Method binding is dynamic
- Classes define the representation of objects
- Class Protocol (messages that the
- Code reuse
- Generalization
- Specializatoin
- Purist Object Orientation
  - Everything is an object
  - No conditionals: ed as messages False
  - was to construct r all expectable objects out there
- Inspired the OO languages we know and love :-)

Actors borrow a lot from this view of OO

A SNAPSHOT OF ERLANG

A SNAPSHOT OF ERLANG

- Actors due to Hewitt,
  Bishop & Steiger ’73

A snapshot of Erlang

- Actors due to Hewitt, Bishop & Steiger ’73
- Erlang: Joe Armstrong (Ericson) ’86
  - Functional Language
  - Pattern Matching
  - Immutable State
  - Everything is a Process

A SNAPSHOT OF ERLANG

- Actors due to Hewitt, Bishop & Steiger ’73
- Erlang: Joe Armstrong (Ericson) ’86
  - Functional Language
  - Pattern Matching
  - Immutable State
  - Everything is a Process
- Key characteristics
  - Distributed
  - Fault-tolerant
  - Soft real-time
  - Highly available
  - Hot swapping

A SNAPSHOT OF ERLANG

- Actors due to Hewitt, Bishop & Steiger ’73
- Erlang: Joe Armstrong (Ericson) ’86
  - Functional Language
  - Pattern Matching
  - Immutable State
  - Everything is a Process
- Key characteristics
  - Distributed
  - Fault-tolerant
  - Soft real-time
  - Highly available
  - Hot swapping
- Popularizes Actors:
  - No shared state
  - Lightweight processes
  - Asynch message-passing
  - Mailboxes messages

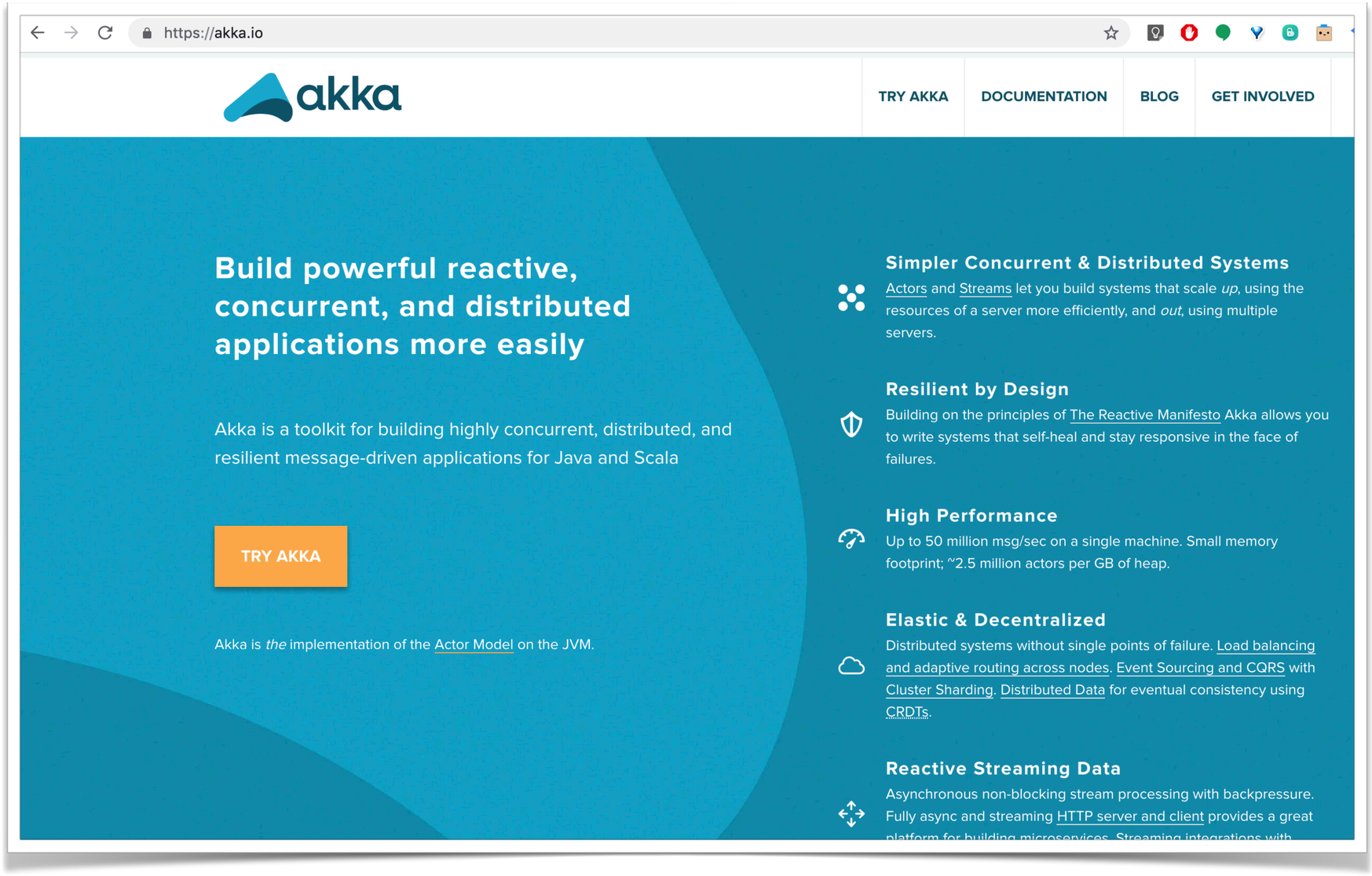
Erlang's Hello World


```erlang
ping(0, PongNode) ->
    {pong, PongNode} ! finished,
    io:format("ping finished~n", []);
ping(N, PongNode) ->
    {pong, PongNode} ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, PongNode).

pong() ->
     receive
         finished ->
             io:format("Pong finished~n", []);
         {ping, Ping_Node} ->
             io:format("Pong received ping~n", []),
             Ping_Node ! pong,
             pong()
     end.
```

Demo

AKKA ACTORS


https://akka.io
akka
TRY AKKA
DOCUMENTATION
BLOG
GET INVOLVED
Build powerful reactive, concurrent, and distributed applications more easily
Akka is a toolkit for building highly concurrent, distributed, and resilient message-driven applications for Java and Scala
TRY AKKA
Akka is *the* implementation of the Actor Model on the JVM.
Simpler Concurrent & Distributed Systems
Actors and Streams let you build systems that scale *up*, using the resources of a server more efficiently, and *out*, using multiple servers.
Resilient by Design
Building on the principles of The Reactive Manifesto Akka allows you to write systems that self-heal and stay responsive in the face of failures.
High Performance
Up to 50 million msg/sec on a single machine. Small memory footprint; ~2.5 million actors per GB of heap.
Elastic & Decentralized
Distributed systems without single points of failure. Load balancing and adaptive routing across nodes. Event Sourcing and CQRS with Cluster Sharding. Distributed Data for eventual consistency using CRDTs.
Reactive Streaming Data
Asynchronous non-blocking stream processing with backpressure. Fully async and streaming HTTP server and client provides a great

Akka Actors (Outline)

- Monday
  - Basics of Akka Programming
  - Fault Tolerance
  - Some Scala goodies

Akka Actors (Outline)

- Monday
  - Basics of Akka Programming
  - Fault Tolerance
  - Some Scala goodies
- Tuesday
  - Distribution
  - Scalability / Elasticity
  - Some Actor Patterns

Akka Actors (Outline)

- Monday
  - Basics of Akka Programming
  - Fault Tolerance
  - Some Scala goodies
- Tuesday
  - Distribution
  - Scalability / Elasticity
  - Some Actor Patterns
- Wednesday
  - Cloud Computing
  - Virtualization
  - Akka on a cluster

Akka Actors (Outline)

- Monday
  - Basics of Akka Programming
  - Fault Tolerance
  - Some Scala goodies
- Tuesday
  - Distribution
  - Scalability / Elasticity
  - Some Actor Patterns
- Wednesday
  - Cloud Computing
  - Virtualization
  - Akka on a cluster
- Thursday
  - Synchronization and State
  - Sharing Data
  - Cluster Orchestration

Akka Actors (Outline)

- Monday
  - Basics of Akka Programming
  - Fault Tolerance
  - Some Scala goodies
- Tuesday
  - Distribution
  - Scalability / Elasticity
  - Some Actor Patterns
- Wednesday
  - Cloud Computing
  - Virtualization
  - Akka on a cluster
- Thursday
  - Synchronization and State
  - Sharing Data
  - Cluster Orchestration
- Friday
  - Orleans / AEON
  - Transactions
  - Lambda?

Akka Actors (Outline)

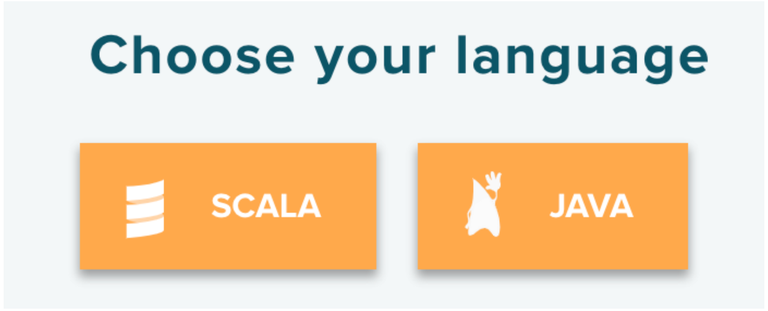
- Monday
  - Basics of Akka Programming
  - Fault Tolerance
  - Some Scala goodies
- Tuesday
  - Distribution
  - Scalability / Elasticity
  - Some Actor Patterns
- Wednesday
  - Cloud Computing
  - Virtualization
  - Akka on a cluster
- Thursday
  - Synchronization and State
  - Sharing Data
  - Cluster Orchestration
- Friday
  - Orleans / AEON
  - Transactions
  - Lambda?

Akka Actors

- We will be using Scala for the lectures, but …

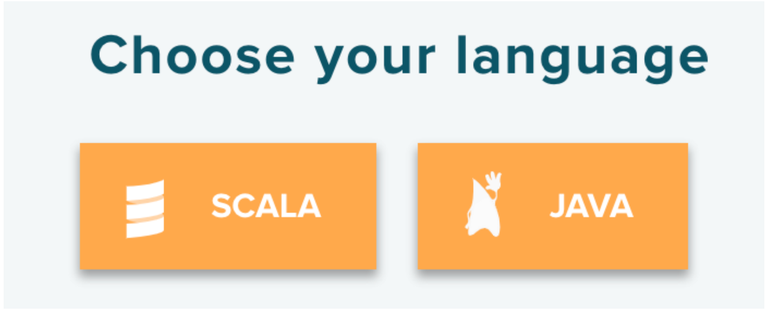
Akka Actors

- We will be using Scala for the lectures, but …



Akka Actors

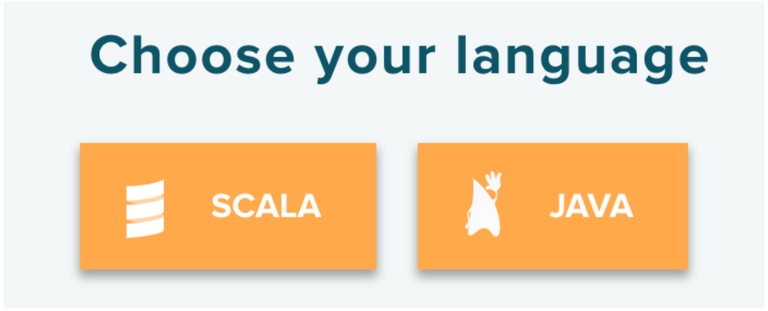
- We will be using Scala for the lectures, but …



- The lectures are not Scala specific

Akka Actors

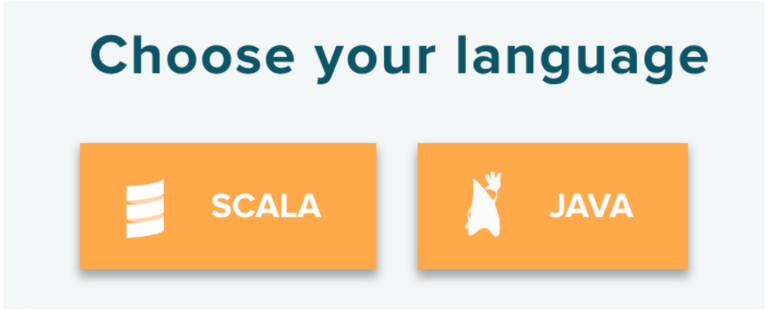
- We will be using Scala for the lectures, but …



- The lectures are not Scala specific
- The documentation of Akka is great! Go try it out!

Akka Actors

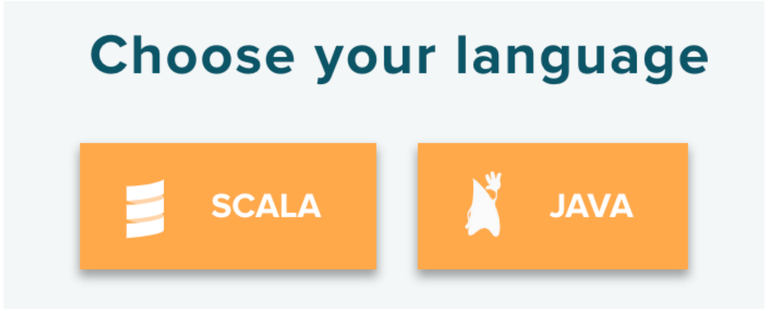
- We will be using Scala for the lectures, but …



- The lectures are not Scala specific
- The documentation of Akka is great! Go try it out!
- There will be a small project at the end of the course

Akka Actors

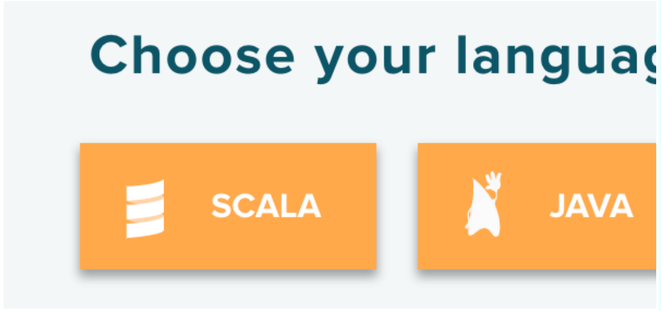
- We will be using Scala for the lectures, but …

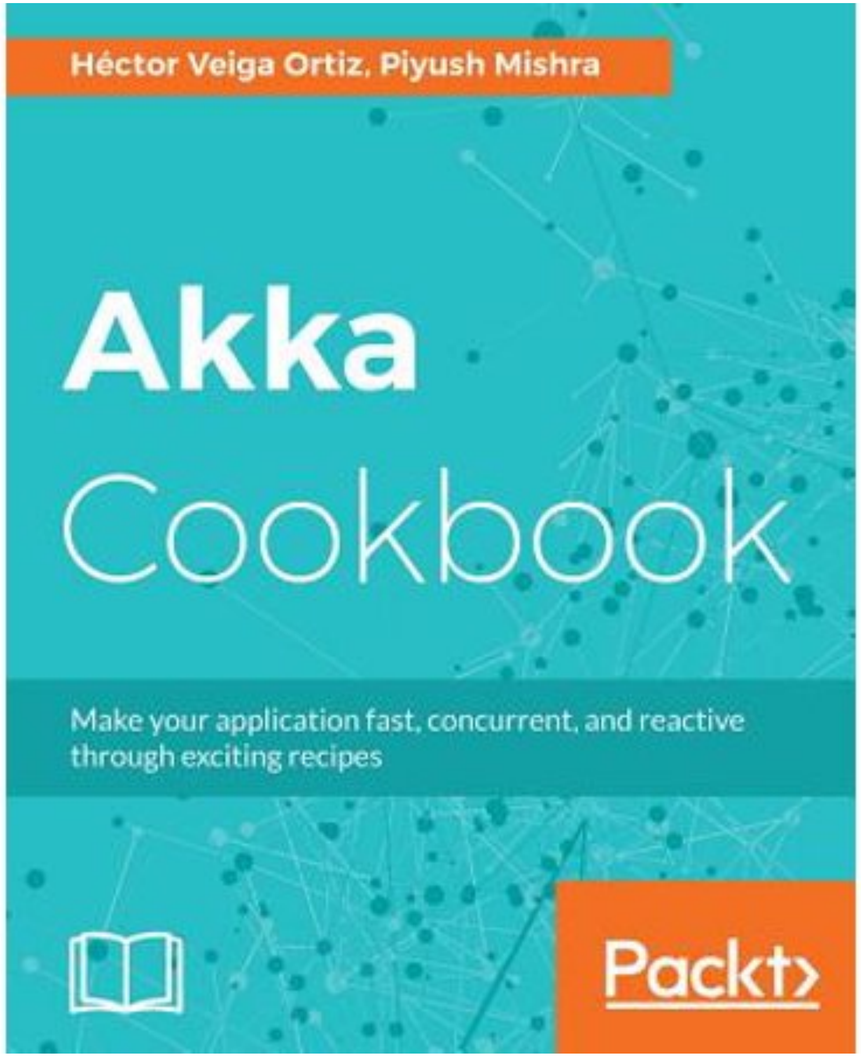


- The lectures are not Scala specific
- The documentation of Akka is great! Go try it out!
- There will be a small project at the end of the course
  - Start reading the documentation of Akka soon!

Akka Actors

- We will be using Scala for the lectures, but …





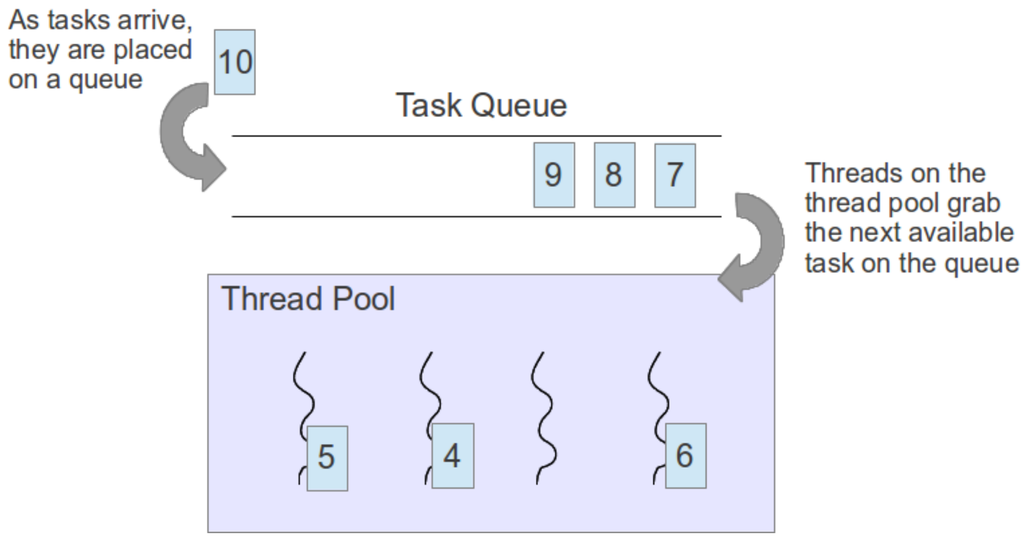
- The lectures are not Scala specific
- The documentation of Akka is great! Go try it
- There will be a small project at the end of the
  - Start reading the documentation of Akka sc

Actor System: Creation

- It manages a thread pool
  - Most likely one per application (or node)

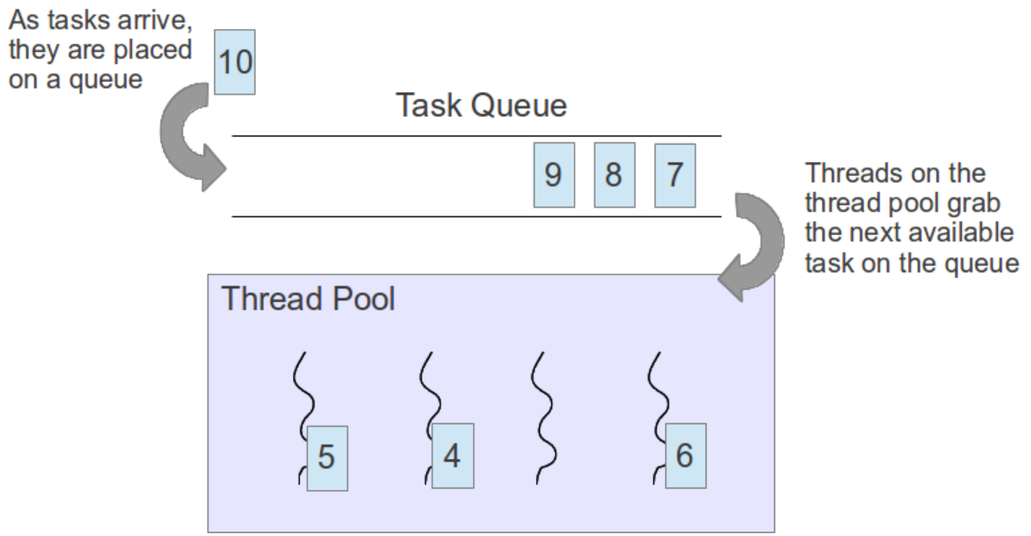
Actor System: Creation

- It manages a thread pool
  - Most likely one per application (or node)



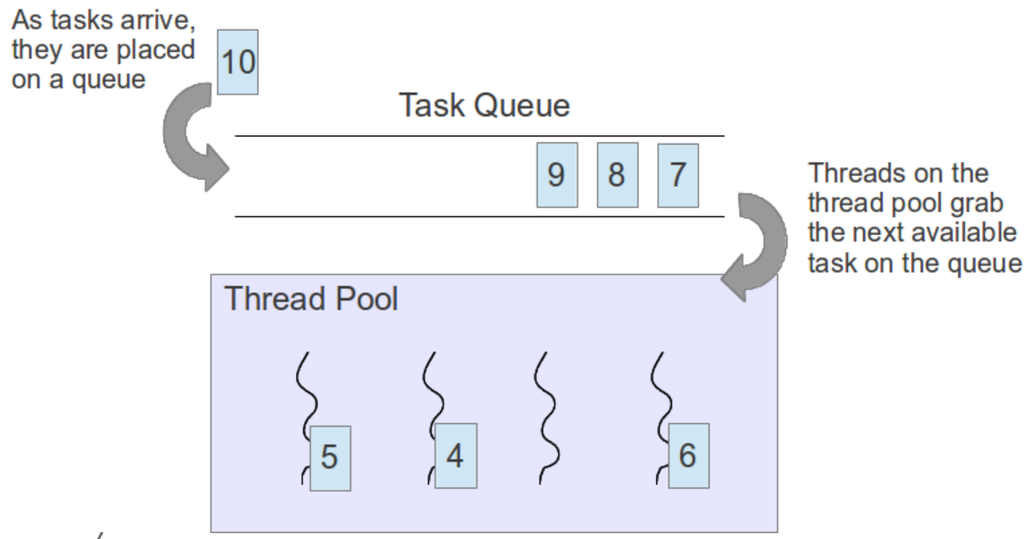
Actor System: Creation

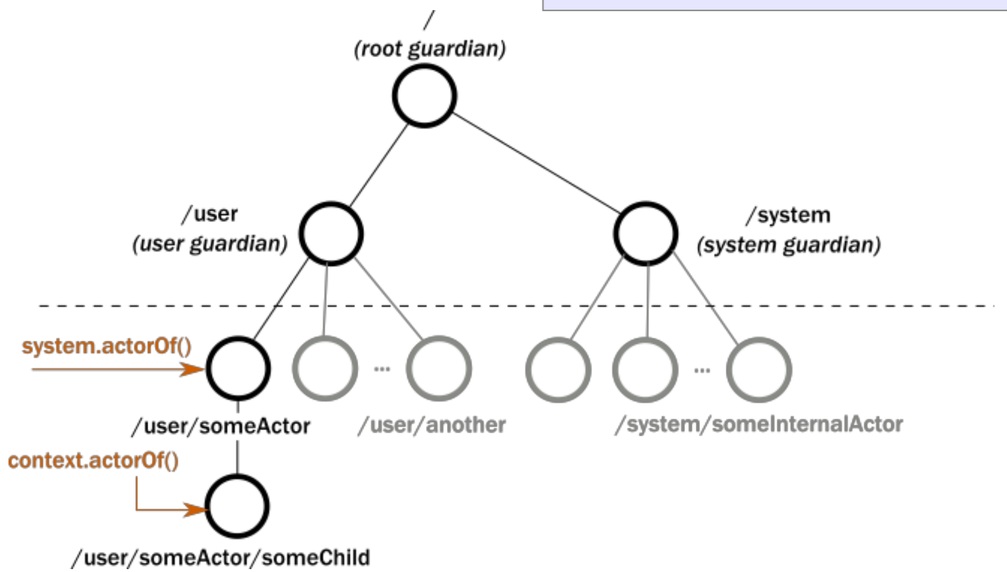
- It manages a thread pool
  - Most likely one per application (or node)
- Takes care of the creation of Actors
  - Cares for the Actor lifecycle



Actor System: Creation

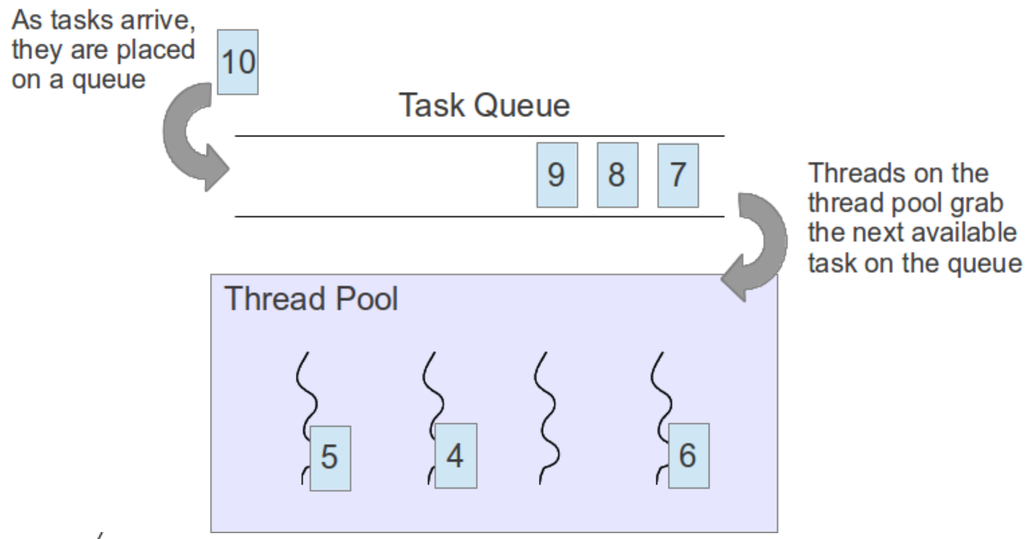
- It manages a thread pool
  - Most likely one per application (or node)
- Takes care of the creation of Actors
  - Cares for the Actor lifecycle
- Creates 3 actors on startup
  - `/user`, `/system`, `/` (the root)

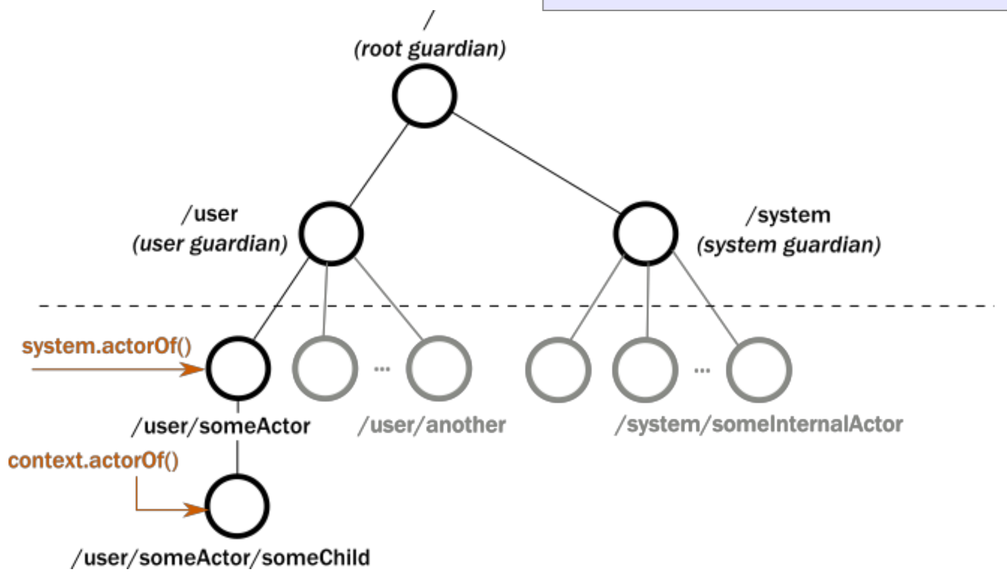




Actor System: Creation

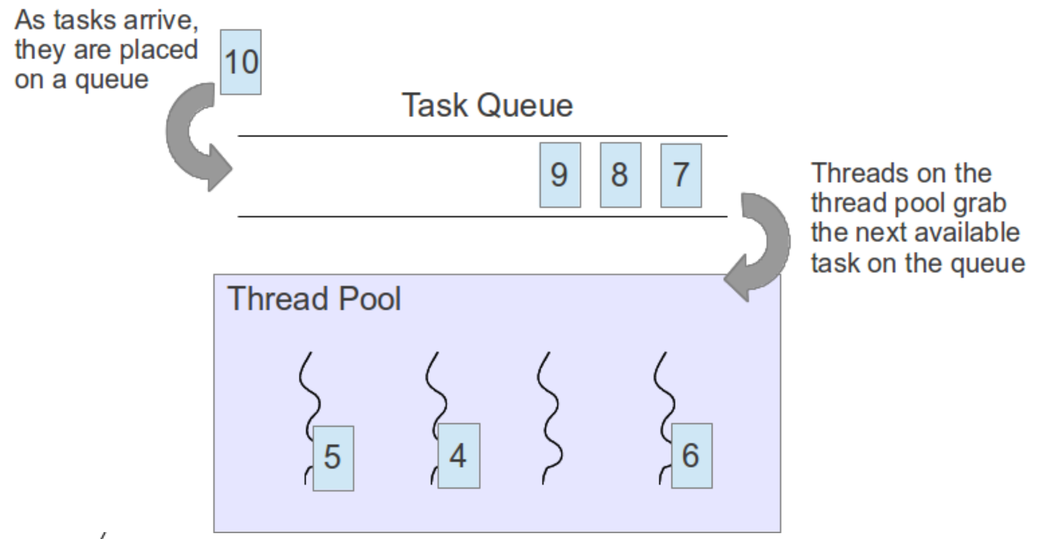
- It manages a thread pool
  - Most likely one per application (or node)
- Takes care of the creation of Actors
  - Cares for the Actor lifecycle
- Creates 3 actors on startup
  - `/user`, `/system`, `/` (the root)
- `AkkaActors.scala`

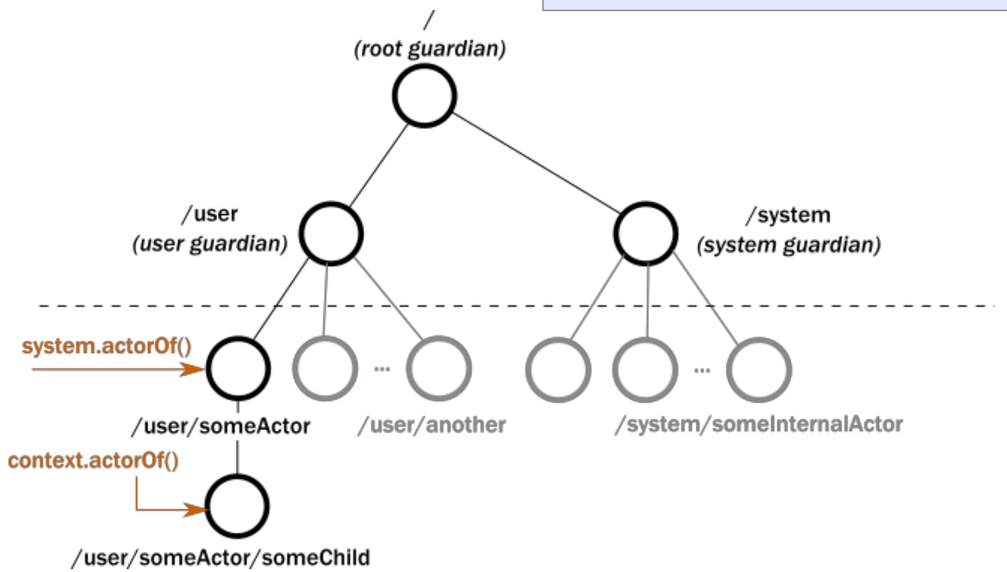




ACTOR SYSTEM: CREATION

- It manages a thread pool
  - Most likely one per application (or node)
- Takes care of the creation of Actors
  - Cares for the Actor lifecycle
- Creates 3 actors on startup
  - `/user`, `/system`, `/` (the root)
- `AkkaActors.scala`



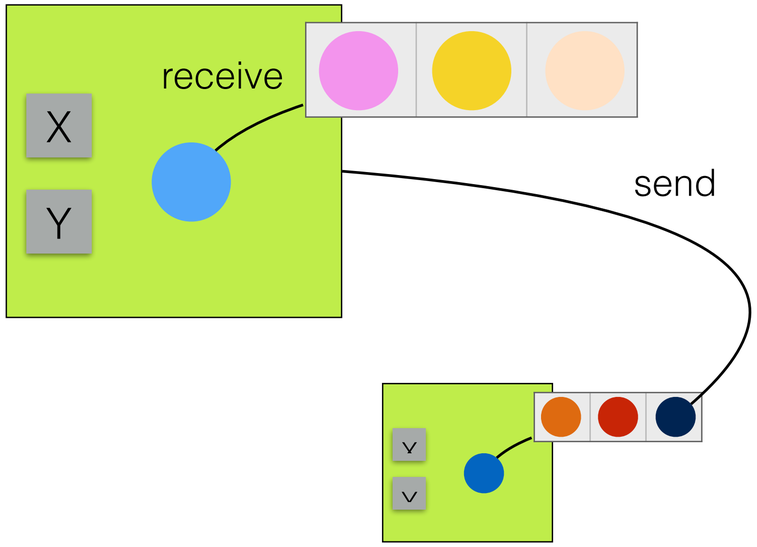


◉ Exercise 1: Create an UM6P Actor System

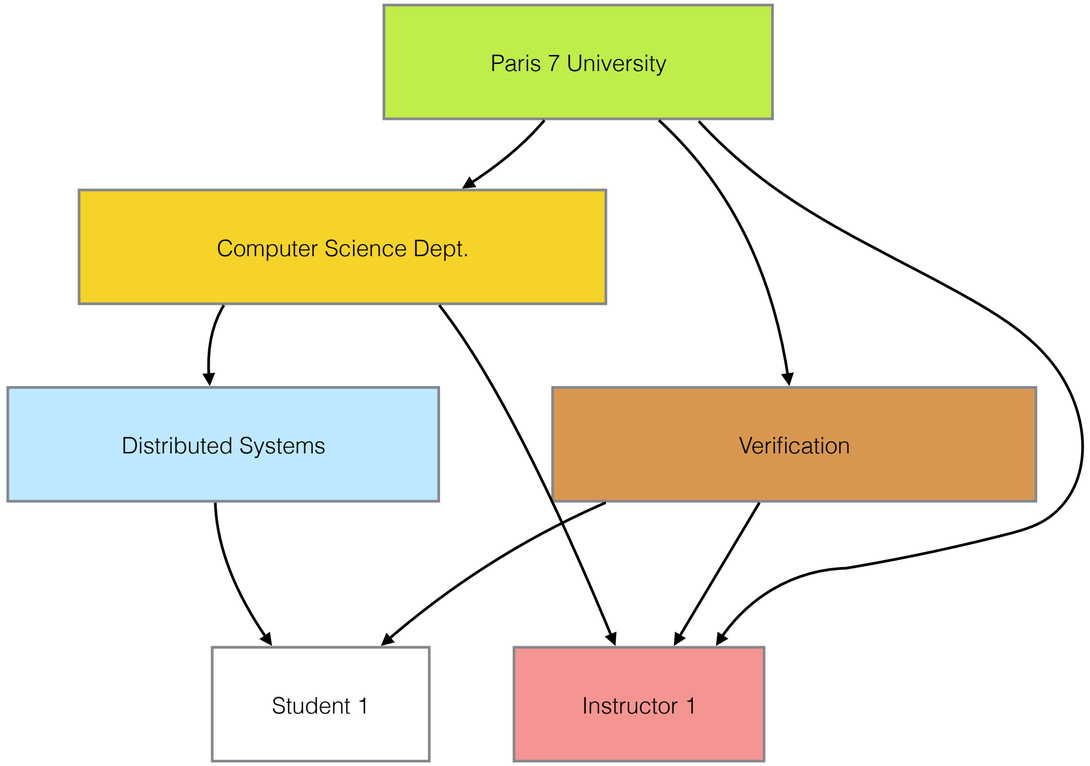
What is an Actor?

- State

Two actors



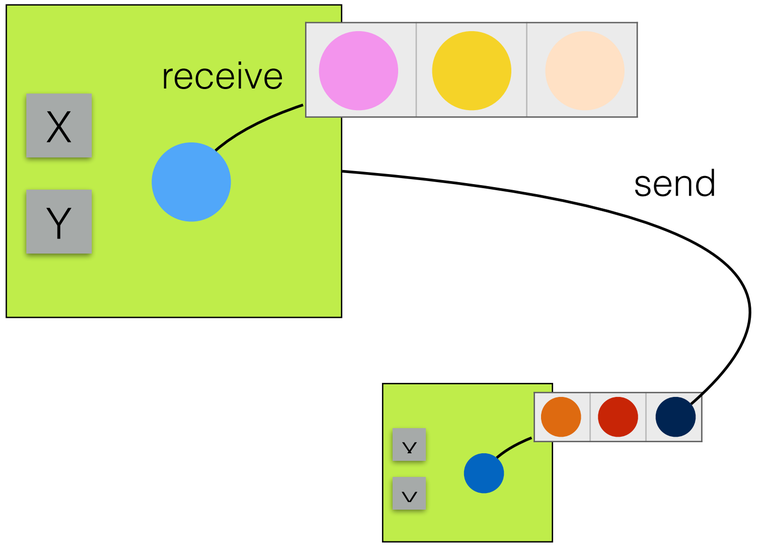
An actor system runtime



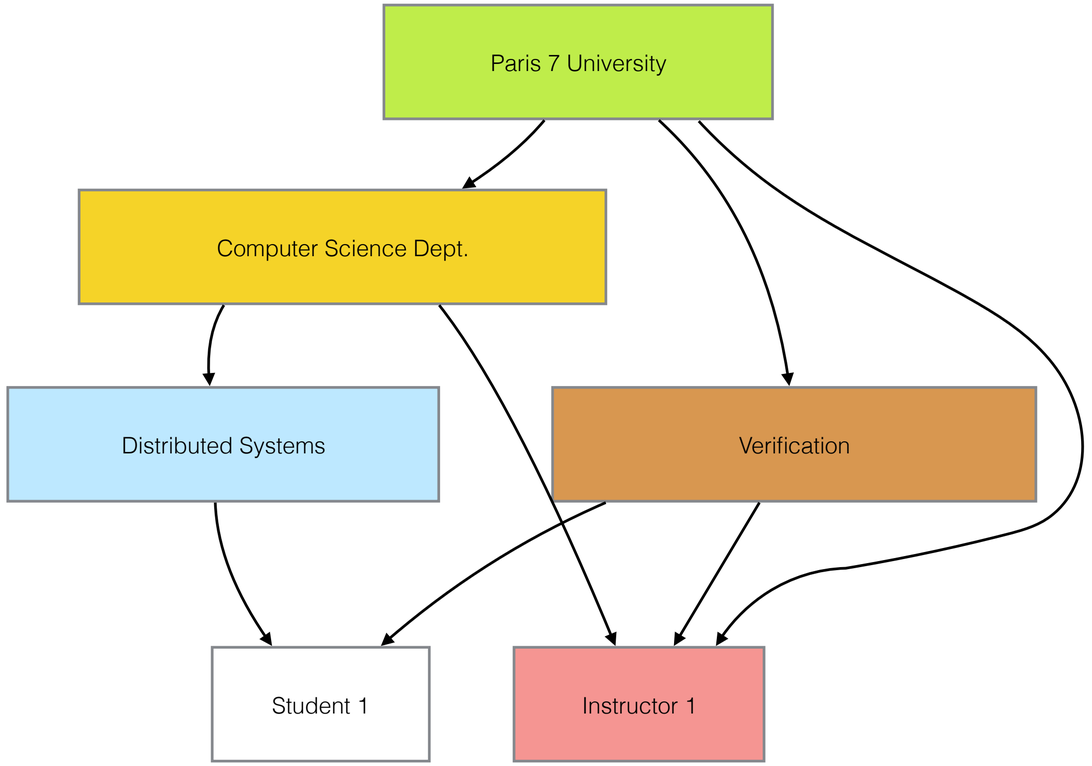
What is an Actor?

- State
- Messages

Two actors



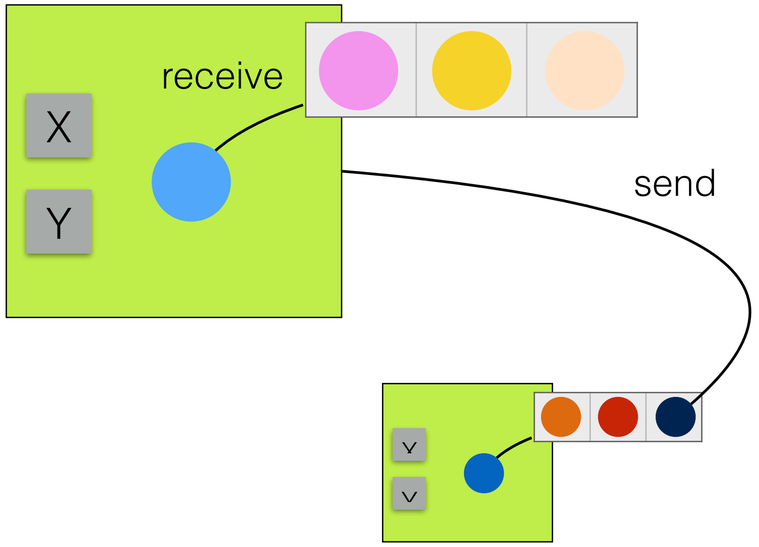
An actor system runtime



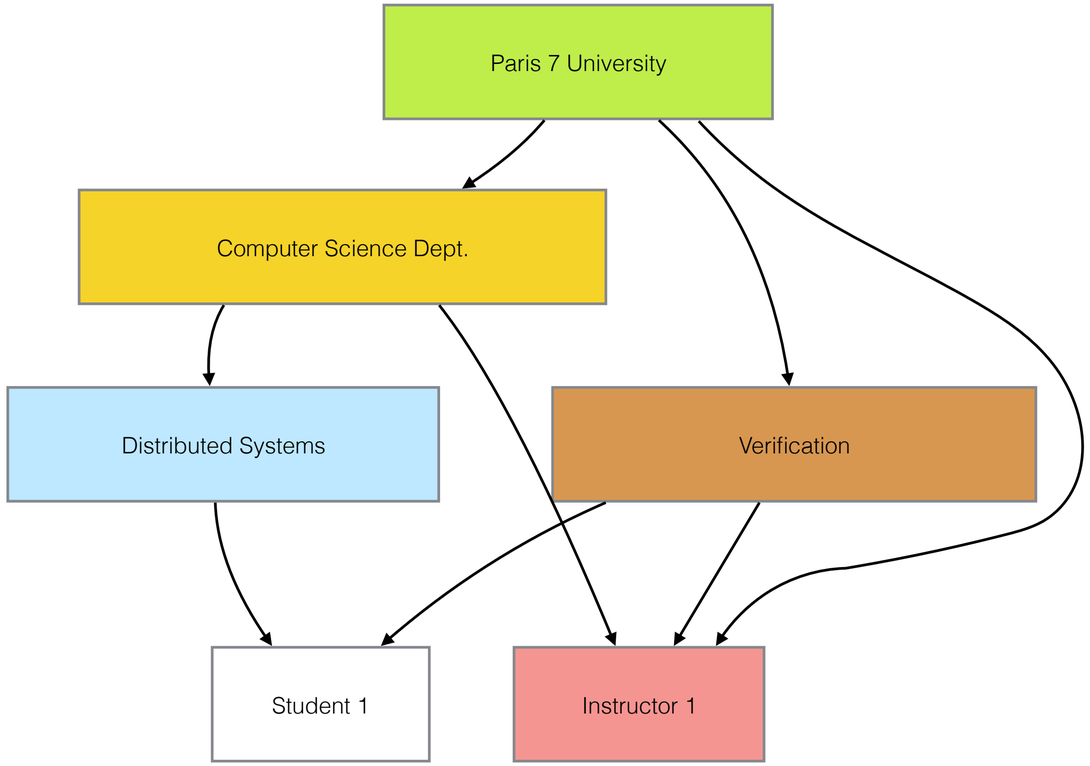
What is an Actor?

- State
- Messages
- Inbox

Two actors



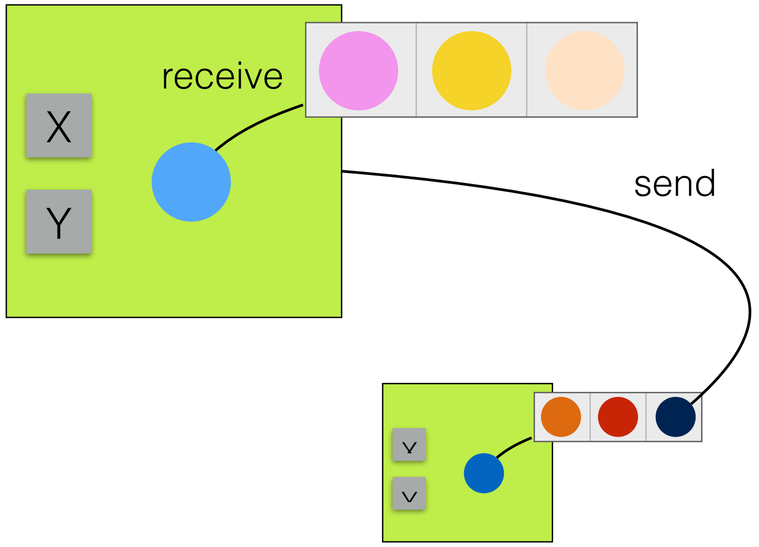
An actor system runtime



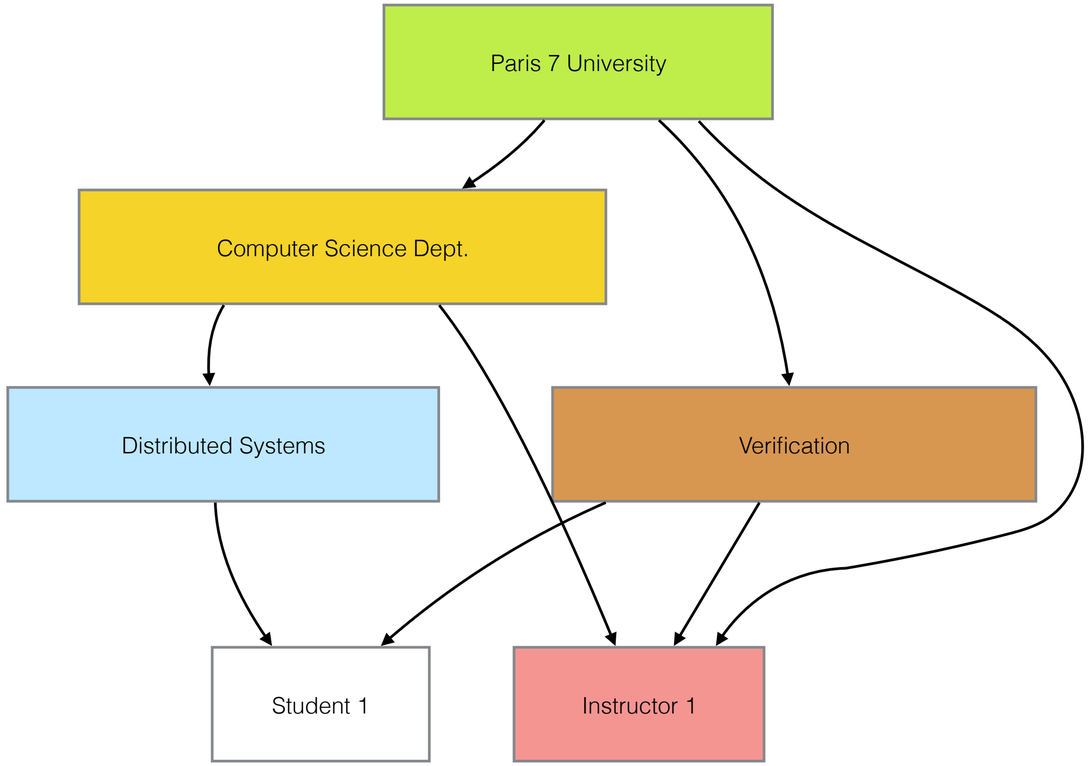
What is an Actor?

- State
- Messages
- Inbox
- Supervision

Two actors



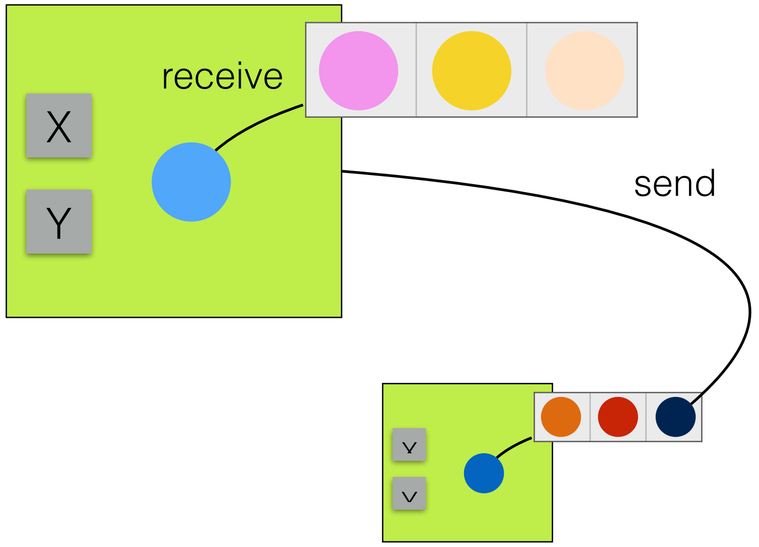
An actor system runtime



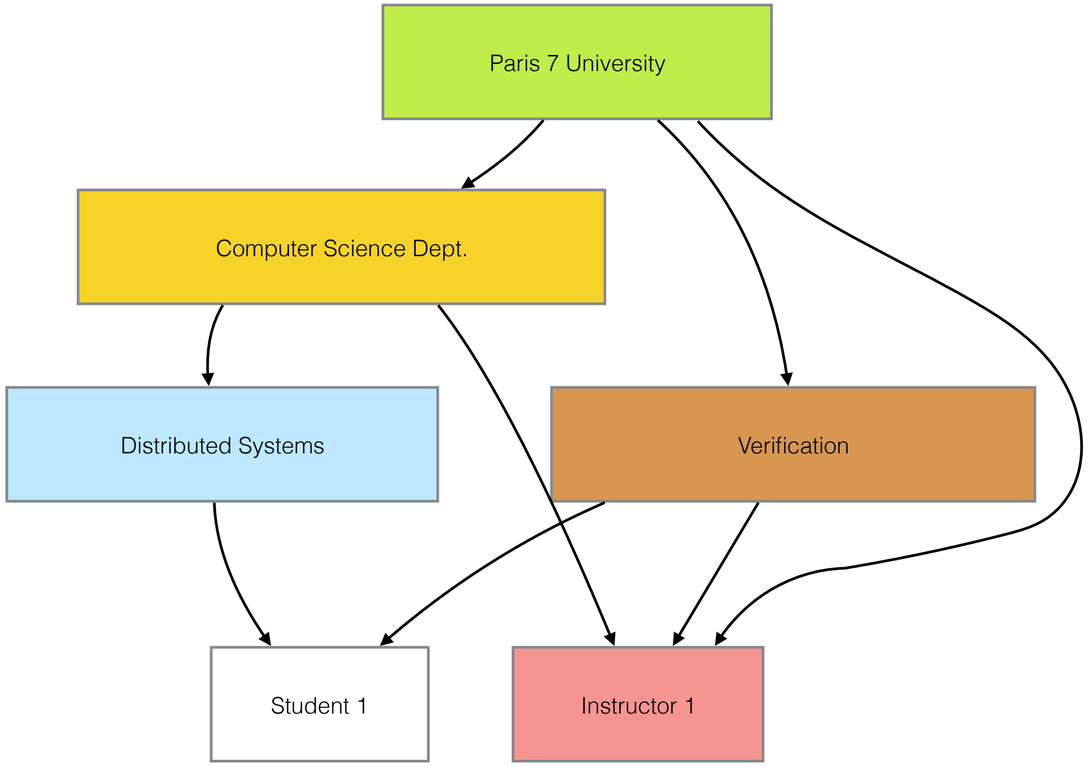
What is an Actor?

- State
- Messages
- Inbox
- Supervision
- Routing (Path)

Two actors

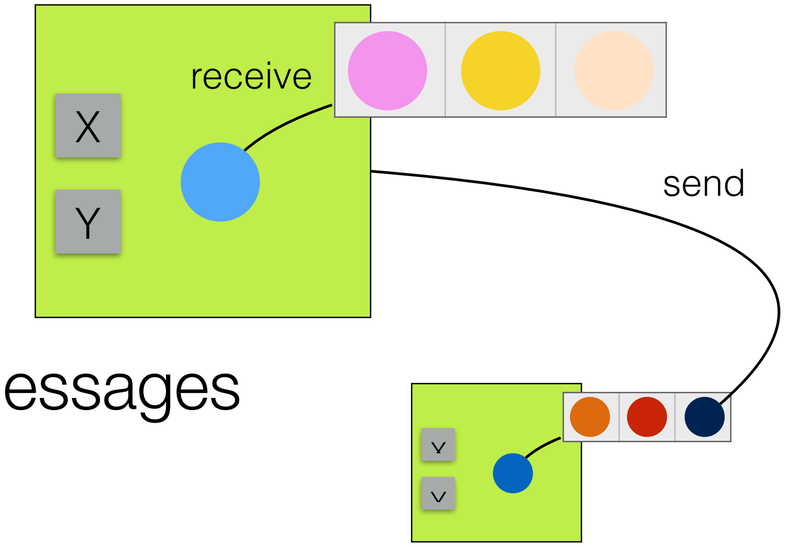


An actor system runtime



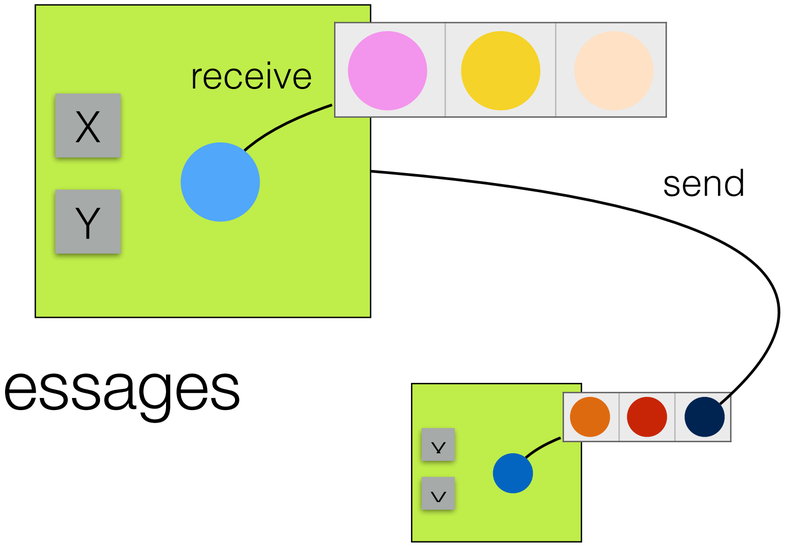
State and Computation

- An class of Actors extends the `Actor` trait
- It can have state (like a normal class)
- It implements the `receive` method to respond to messages
- To send a message we use `!`
- `BehaviorAndState.scala`
- The path



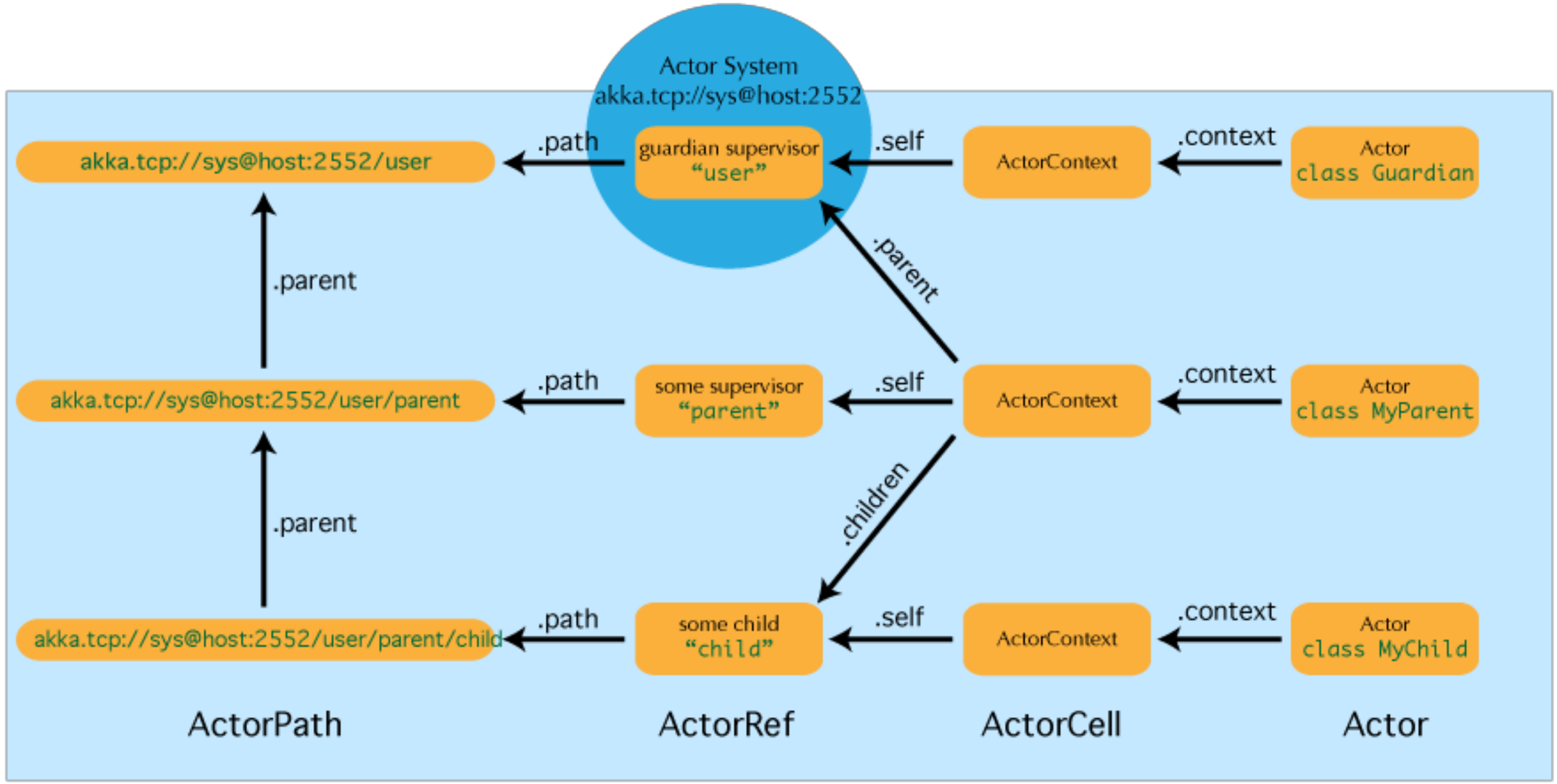
State and Computation

- An class of Actors extends the `Actor` trait
- It can have state (like a normal class)
- It implements the `receive` method to respond to messages
- To send a message we use `!`
- `BehaviorAndState.scala`
- The path



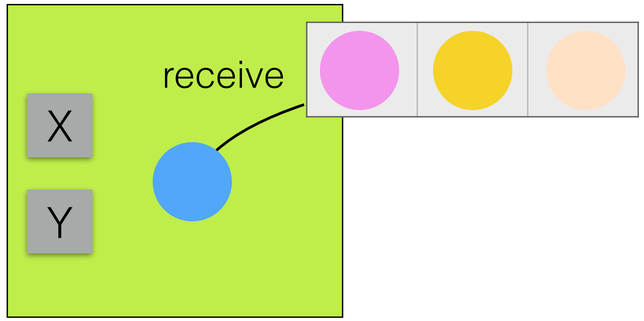
◉ Exercise 2: Create actors for a University Department, for Courses and for Students

PATH OF AN ACTOR

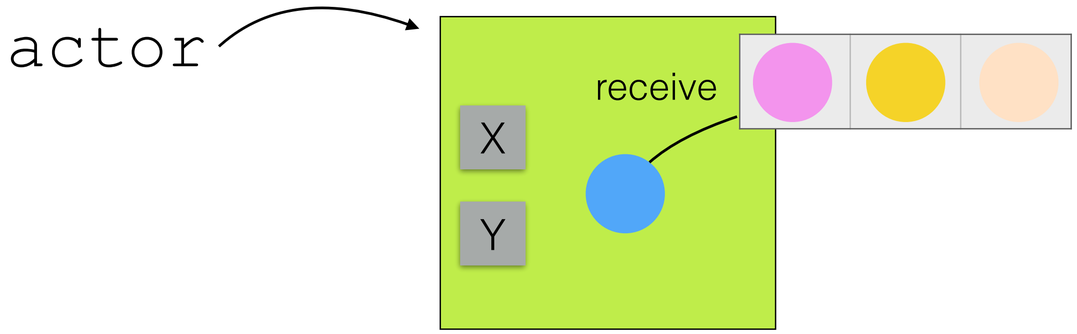


Sending a Message to an Actor

- How can we refer to an actor?

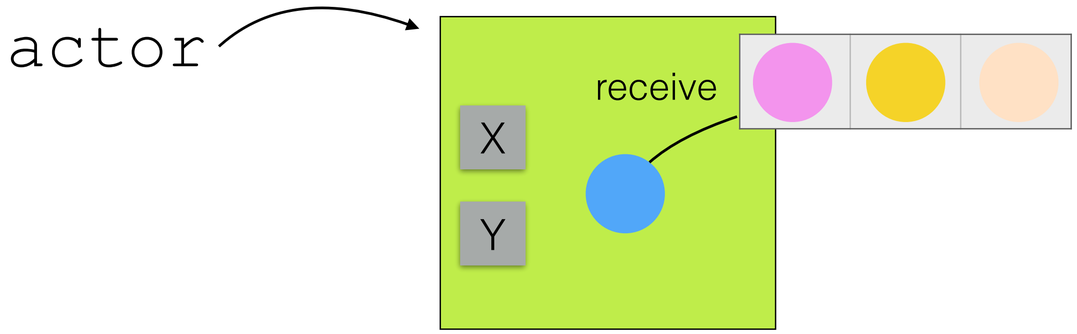


Sending a Message to an Actor



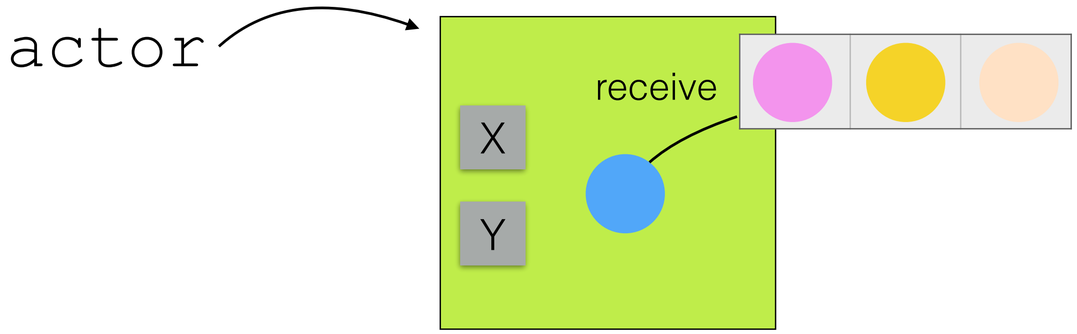
- How can we refer to an actor?
  - The creation of an actor returns an `ActorRef`, a reference to the actor

Sending a Message to an Actor



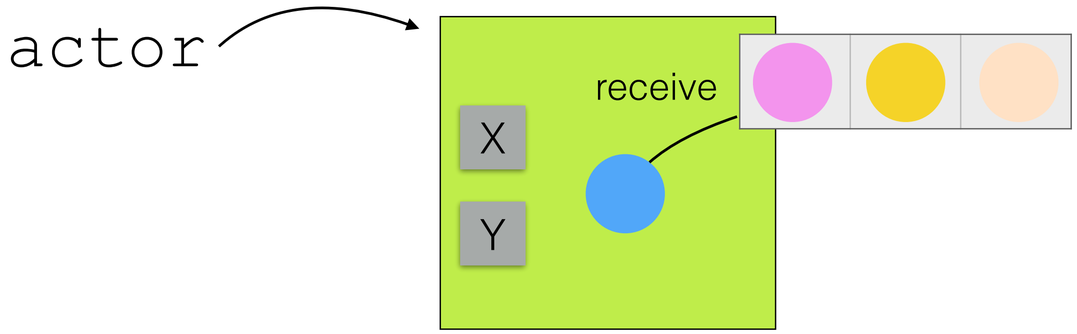
- How can we refer to an actor?
  - The creation of an actor returns an `ActorRef`, a reference to the actor
- We can send a message to an actor through this reference: `actor!1`

Sending a Message to an Actor



- How can we refer to an actor?
  - The creation of an actor returns an `ActorRef`, a reference to the actor
- We can send a message to an actor through this reference: `actor!1`
- The message is dispatched to the mailbox, and executed later

Sending a Message to an Actor



- How can we refer to an actor?
  - The creation of an actor returns an `ActorRef`, a reference to the actor
- We can send a message to an actor through this reference: `actor!1`
- The message is dispatched to the mailbox, and executed later

◉ Exercise 3: A small application with two departments of UMP6, a few courses and a few students. Have the students register to courses

Getting an Answer

- How do we get an answer from an actor?

Getting an Answer

- How do we get an answer from an actor?
  - Each request defines a special ActorRef `sender`
  - The receiver sends a message to the sender

Getting an Answer

- How do we get an answer from an actor?
  - Each request defines a special ActorRef `sender`
  - The receiver sends a message to the sender
- `akka.pattern.ask` to get answers

Getting an Answer

- How do we get an answer from an actor?
  - Each request defines a special ActorRef `sender`
  - The receiver sends a message to the sender
- `akka.pattern.ask` to get answers
- The answer is not immediately present (since messages are asynchronous)
  - Therefore we must use futures

Getting an Answer

- How do we get an answer from an actor?
  - Each request defines a special ActorRef `sender`
  - The receiver sends a message to the sender
- `akka.pattern.ask` to get answers
- The answer is not immediately present (since messages are asynchronous)
  - Therefore we must use futures
- `FiboActorApp.scala`

Getting an Answer

- How do we get an answer from an actor?
  - Each request defines a special ActorRef `sender`
  - The receiver sends a message to the sender
- `akka.pattern.ask` to get answers
- The answer is not immediately present (since messages are asynchronous)
  - Therefore we must use futures
- `FiboActorApp.scala`

◉ Exercise 4: Define a maximum capacity for courses and confirm or reject student registrations

Communication between of Actors

- Actors can talk back and forth with each other
- `Communication.scala`

Communication between of Actors

- Actors can talk back and forth with each other
- `Communication.scala`

> Exercise 5: Assuming the students are registered in the department, have the registration in a course confirm with the department if the student is registered at the university

Fault Tolerance

Fault Tolerance

- Bad things happen ~~sometimes~~ all the time

Fault Tolerance

- Bad things happen ~~sometimes~~ all the time
  - Catastrophic failures: floods, fires, etc.

Fault Tolerance

- Bad things happen ~~sometimes~~ all the time
  - Catastrophic failures: floods, fires, etc.
  - Infrastructures fail: electricity, network, etc.

Fault Tolerance

- Bad things happen ~~sometimes~~ all the time
  - Catastrophic failures: floods, fires, etc.
  - Infrastructures fail: electricity, network, etc.
  - Bugs happen: ALL THE TIME!!!

Fault Tolerance

- Bad things happen ~~sometimes~~ all the time
  - Catastrophic failures: floods, fires, etc.
  - Infrastructures fail: electricity, network, etc.
  - Bugs happen: ALL THE TIME!!!
- Graceful Degradation

FAULT TOLERANCE

- Bad things happen ~~sometimes~~ all the time
  - Catastrophic failures: floods, fires, etc.
  - Infrastructures fail: electricity, network, etc.
  - Bugs happen: ALL THE TIME!!!
- Graceful Degradation
  - Decreased Throughput: meh …, some clients will have to retry

FAULT TOLERANCE

- Bad things happen ~~sometimes~~ all the time
  - Catastrophic failures: floods, fires, etc.
  - Infrastructures fail: electricity, network, etc.
  - Bugs happen: ALL THE TIME!!!
- Graceful Degradation
  - Decreased Throughput: meh …, some clients will have to retry
  - Increased Latency: meh …, some clients will be unhappy waiting

FAULT TOLERANCE

- Bad things happen ~~sometimes~~ all the time
  - Catastrophic failures: floods, fires, etc.
  - Infrastructures fail: electricity, network, etc.
  - Bugs happen: ALL THE TIME!!!
- Graceful Degradation
  - Decreased Throughput: meh …, some clients will have to retry
  - Increased Latency: meh …, some clients will be unhappy waiting
  - *Unavailable:* oh s*** …, clients are going away!

FAULT TOLERANCE

- Bad things happen ~~sometimes~~ all the time
  - Catastrophic failures: floods, fires, etc.
  - Infrastruct
  - Bugs happ
- Graceful Deg
  - Decreased Throughput: meh …, some clients will have to retry
  - Increased Latency: meh …, some clients will be unhappy waiting
  - *Unavailable:* oh s*** …, clients are going away!

Slow might be acceptable
Retries might be acceptable
Unreachable is not!

Fault Tolerance: Architecture

Fault Tolerance: Architecture

- Compartmentalization
  - Separation of Concerns and Functionality
  - We can harden / restart critical sub-systems

Fault Tolerance: Architecture

- Compartmentalization
  - Separation of Concerns and Functionality
  - We can harden / restart critical sub-systems
- Avoid cascading Failures

Fault Tolerance: Architecture

- Compartmentalization
  - Separation of Concerns and Functionality
  - We can harden / restart critical sub-systems
- Avoid cascading Failures
- Isolate failures from important parts of the system

Fault Tolerance: Architecture

- Compartmentalization
  - Separation of Concerns and Functionality
  - We can harden / restart critical sub-systems
- Avoid cascading Failures
- Isolate failures from important parts of the system
- Back-up / Replicate / Scale Bottlenecks

Fault Tolerance: Mechanisms

Fault Tolerance: Mechanisms

- State Machine Replication
  - Multiple nodes perform the same action
  - Easy for state-less applications
  - Requires Synchronization for stateful applications

Fault Tolerance: Mechanisms

- State Machine Replication
  - Multiple nodes perform the same action
  - Easy for state-less applications
  - Requires Synchronization for stateful applications
- State Replication
  - Requires Synchronization
  - Or weakened consistency (this will be discussed Thu. or Fri.)

Fault Tolerance: Mechanisms

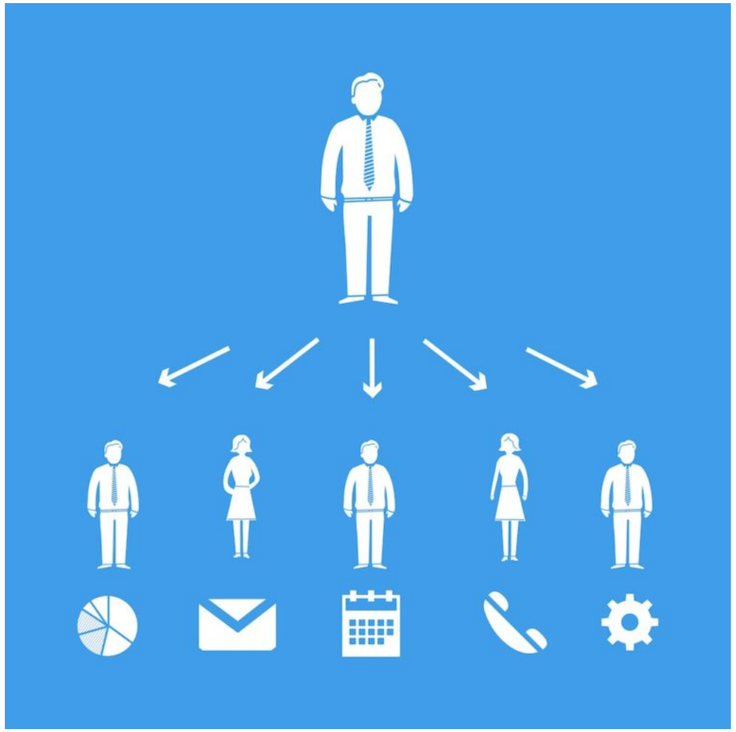
- State Machine Replication
  - Multiple nodes perform the same action
  - Easy for state-less applications
  - Requires Synchronization for stateful applications
- State Replication
  - Requires Synchronization
  - Or weakened consistency (this will be discussed Thu. or Fri.)
- Component Isolation
  - Message Passing

Fault Tolerance: Mechanisms

- State Machine Replication
  - Multiple nodes perform the same action
  - Easy for state-less applications
  - Requires Synchronization for stateful applications
- State Replication
  - Requires Synchronization
  - Or weakened consistency (this will be discussed Thu. or Fri.)
- Component Isolation
  - Message Passing
- Persistent state (Snapshots)

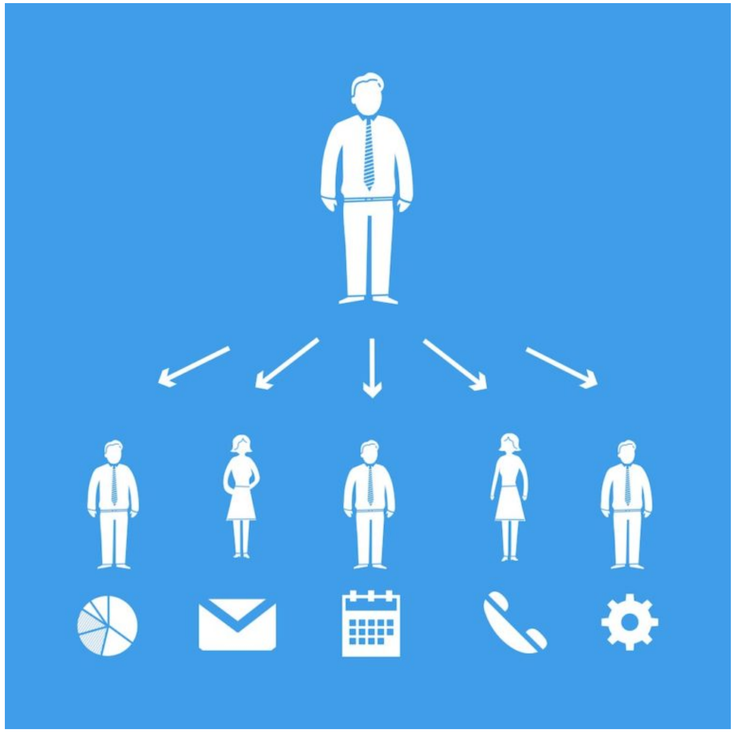
Fault Tolerance: Mechanisms

- Delegation
  - Hand-over responsibility



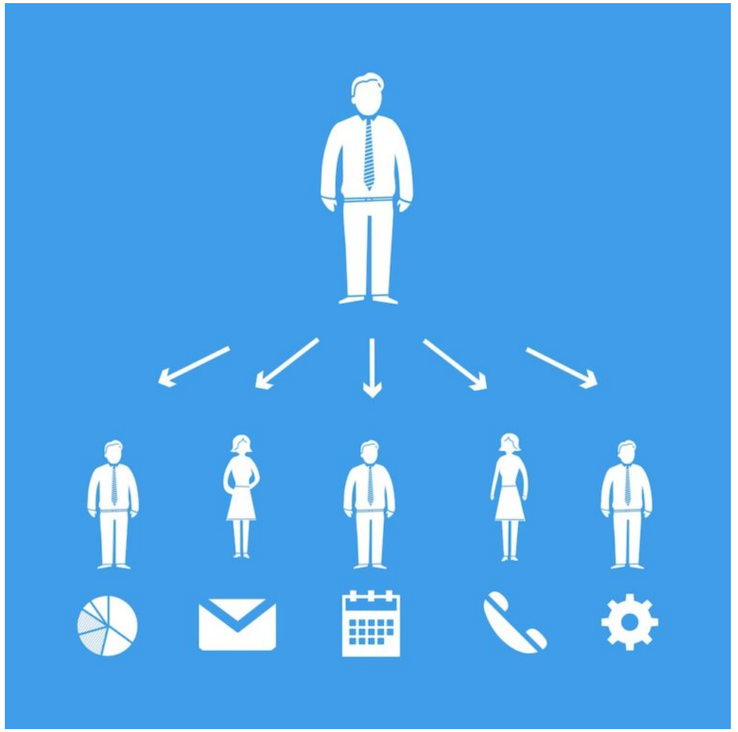
Fault Tolerance: Mechanisms

- Delegation
  - Hand-over responsibility
- Components
  - Self-contained
  - Isolated
  - Encapsulated



Fault Tolerance: Mechanisms

- Delegation
  - Hand-over responsibility
- Components
  - Self-contained
  - Isolated
  - Encapsulated
- Micro-services / Actors

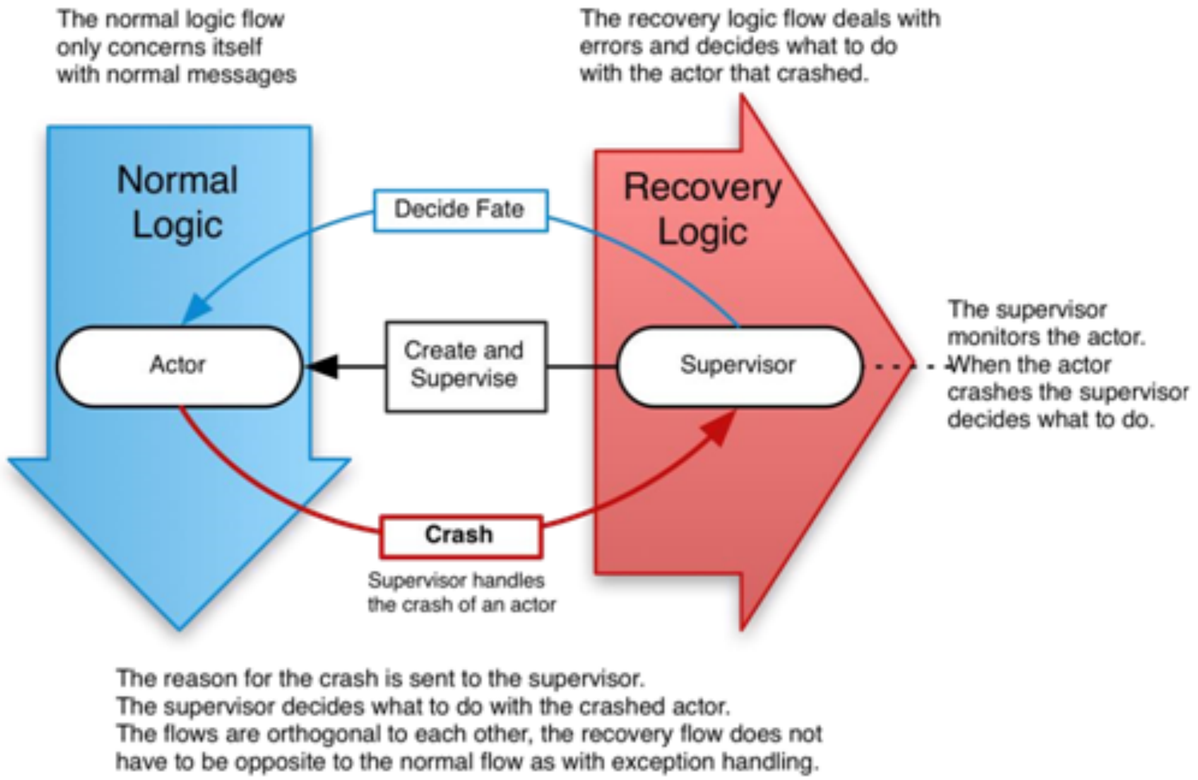


Fault Tolerance: Supervision

https://www.developer.com/mgmt/akka-in-action-let-it-crash.html

FAULT TOLERANCE: SUPERVISION

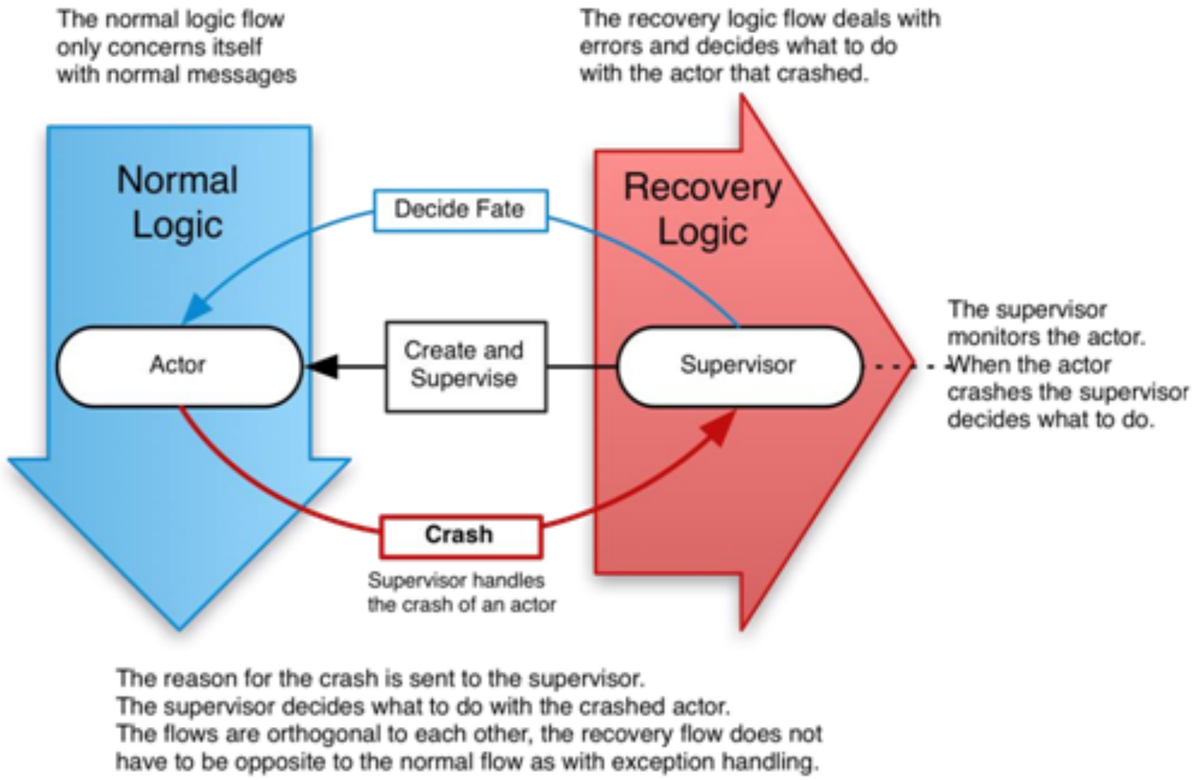
- *“Let it crash”*



https://www.developer.com/mgmt/akka-in-action-let-it-crash.html

FAULT TOLERANCE: SUPERVISION

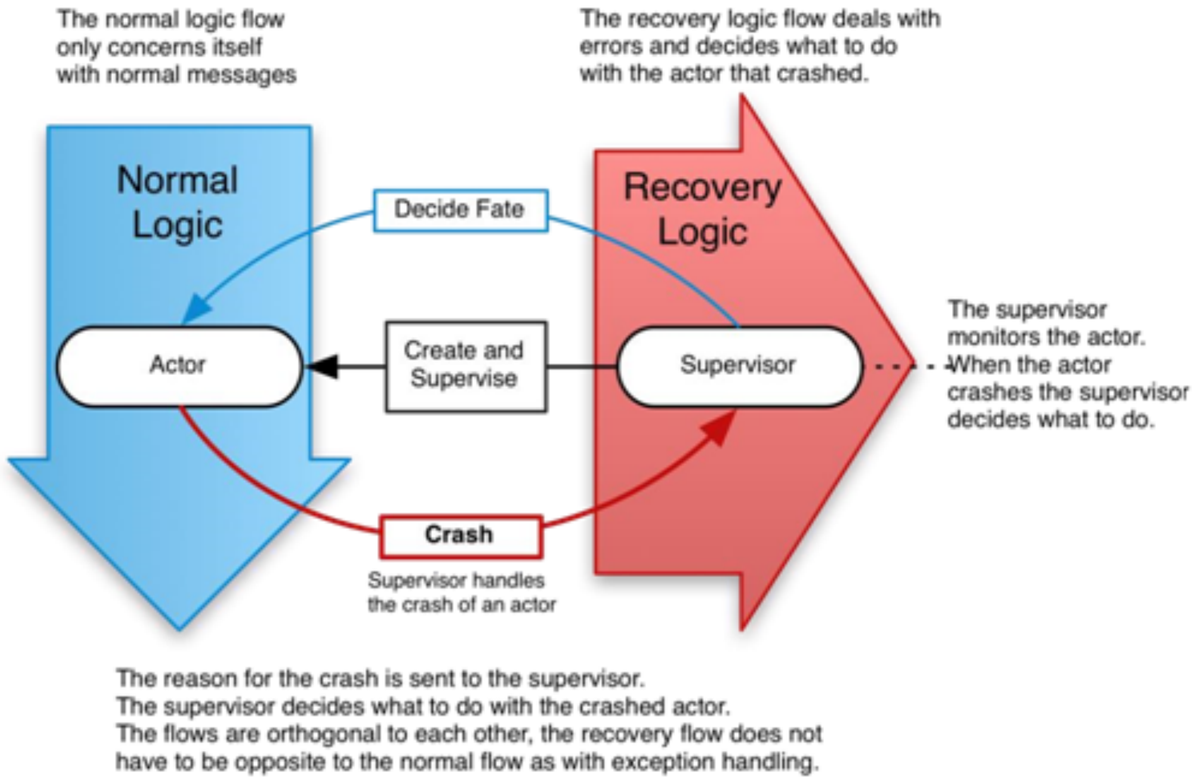
- *“Let it crash”*
- Parent / Child Supervision:
  - send messages
  - collect answers



https://www.developer.com/mgmt/akka-in-action-let-it-crash.html

FAULT TOLERANCE: SUPERVISION

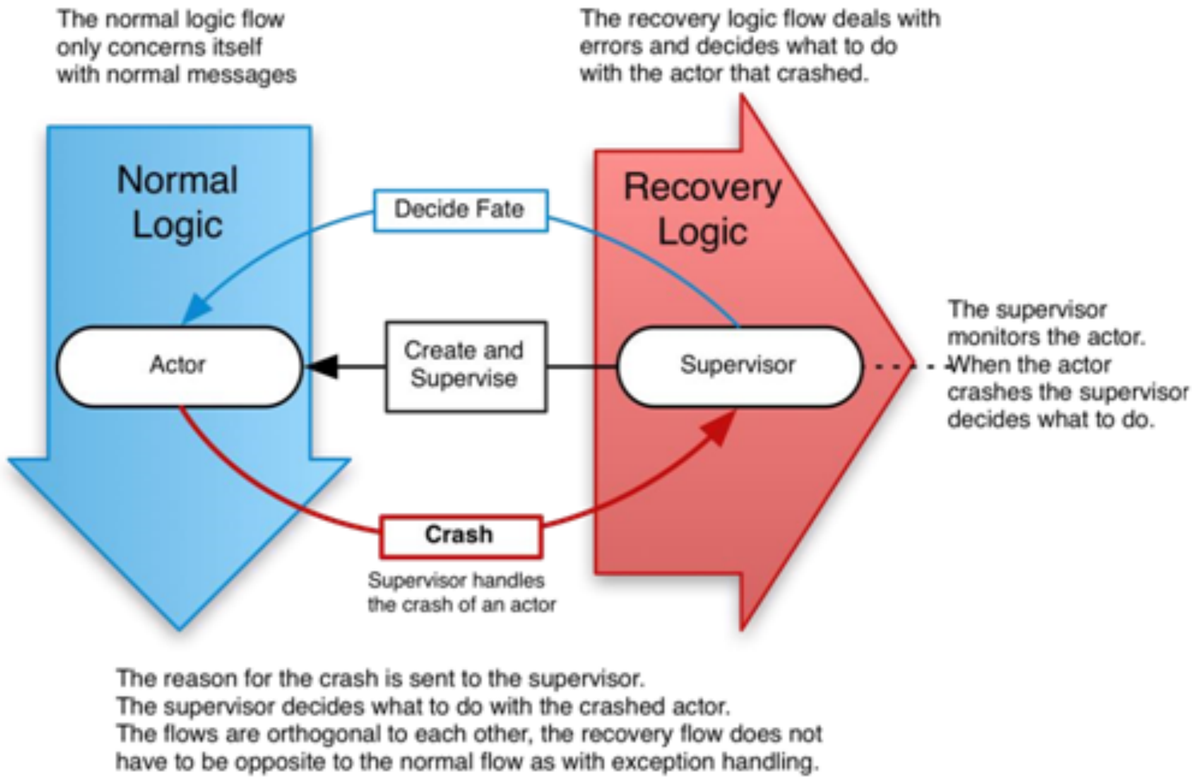
- *“Let it crash”*
- Parent / Child Supervision:
  - send messages
  - collect answers
- One-for-one Strategy



https://www.developer.com/mgmt/akka-in-action-let-it-crash.html

FAULT TOLERANCE: SUPERVISION

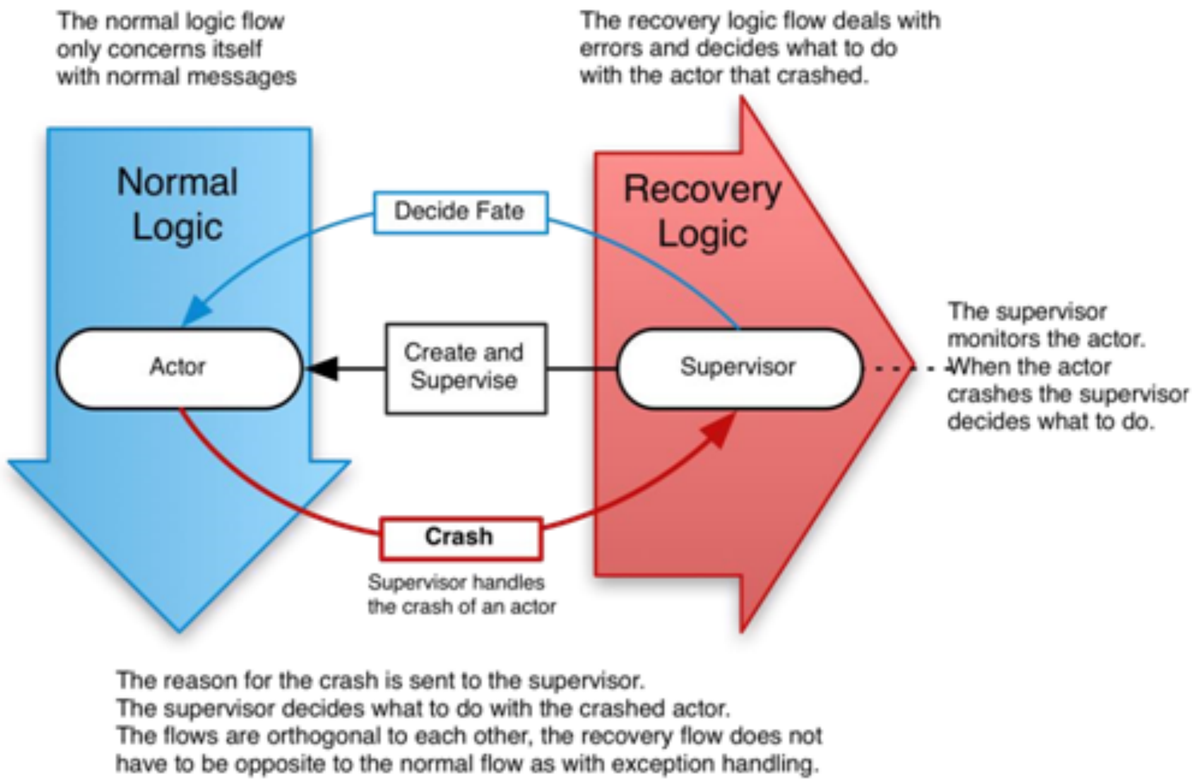
- *"Let it crash"*
- Parent / Child Supervision:
  - send messages
  - collect answers
- One-for-one Strategy
- All-for-one Strategy



https://www.developer.com/mgmt/akka-in-action-let-it-crash.html

FAULT TOLERANCE: SUPERVISION

- *“Let it crash”*
- Parent / Child Supervision:
  - send messages
  - collect answers
- One-for-one Strategy
- All-for-one Strategy
- Death-Watch



https://www.developer.com/mgmt/akka-in-action-let-it-crash.html