

# ACTOR PROGRAMMING FOR THE CLOUD

# WHO AM I?

GUSTAVO PETRI

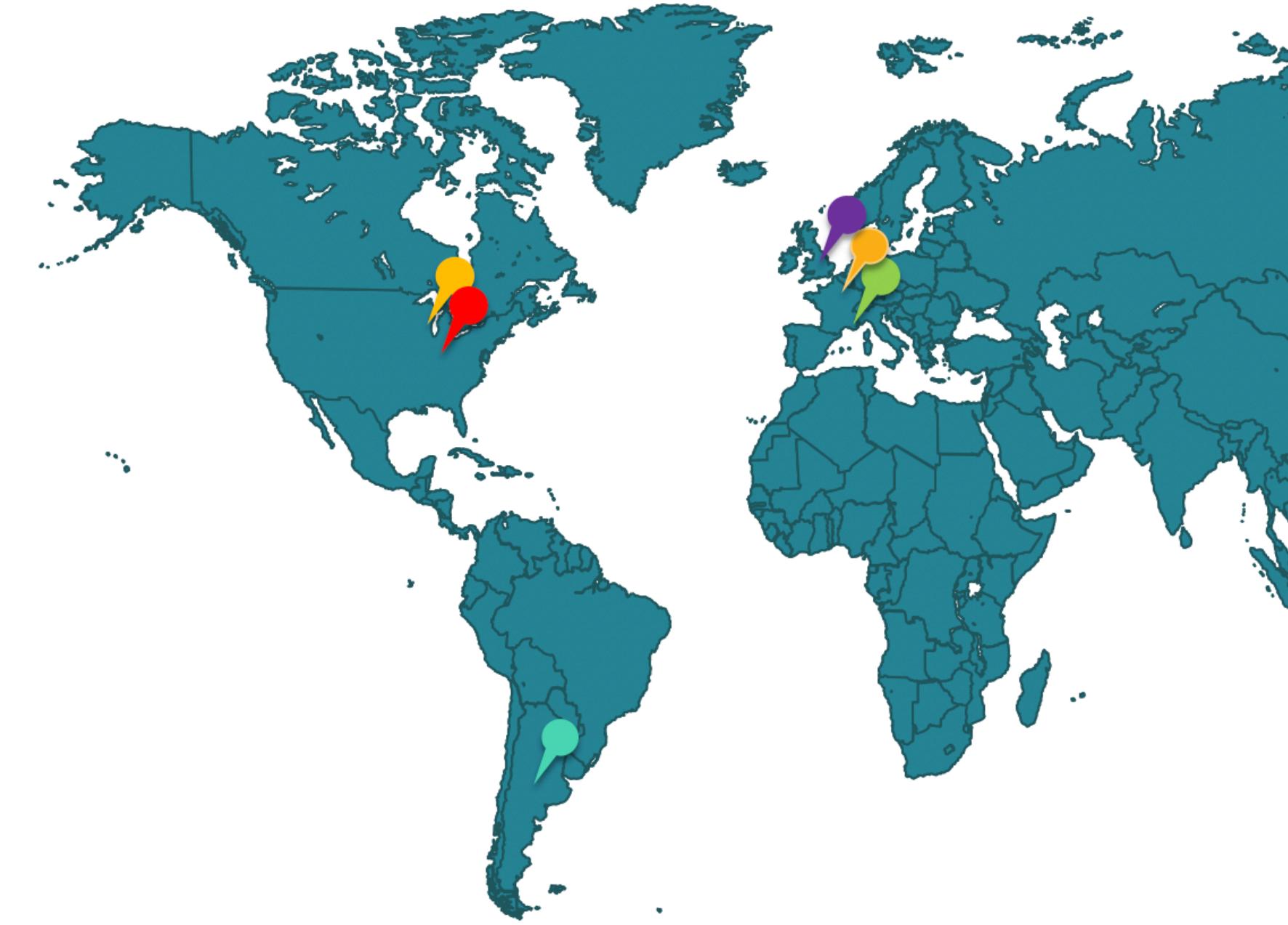
[2006 - 2010] PhD. Sophia Antipolis, *INRIA*, France

[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?

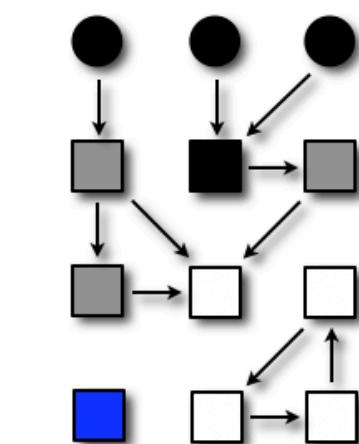
## Topics

- ▶ Relaxed Memory Models
- ▶ Concurrent Program Verification
- ▶ Distributed Systems Verification
- ▶ Models for Distributed Computing
- ▶ and now ... Security

## Tools

- ▶ Formal Semantics
- ▶ Formal Methods
- ▶ Programming Languages

Verifying a  
Concurrent GC  
(Rely/Guarantee)

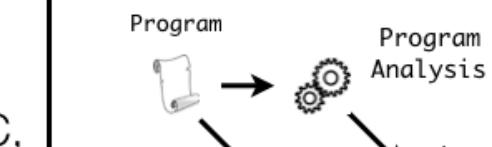


Theorem:  
In any execution of the CGC,

Learning Shape  
Specifications

$$\begin{aligned} \forall u, v, \nu : u \cup v \Rightarrow & \\ & \left( h1 : u \cup v \vee h1 : v \cup u \vee \right. \\ & \left. h2 : u \cup v \vee h2 : v \cup u \vee \right. \\ & \left. (h1 \rightarrow u \wedge h2 \rightarrow v) \vee \right. \\ & \left. (h2 \rightarrow u \wedge h1 \rightarrow v) \right) \end{aligned}$$

CEGAR Loop



Elastic Cloud  
Programming



Lower cost  
Dynamic workload  
Dynamic scaling: Elasticity

Invariants for  
Relaxed  
Consistency

Invariant	Protocol
1Ob	Single Object
2Ob	Partial Order 2 objects
NOb	Equivalence Relations

Total Order (10b)

# 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.

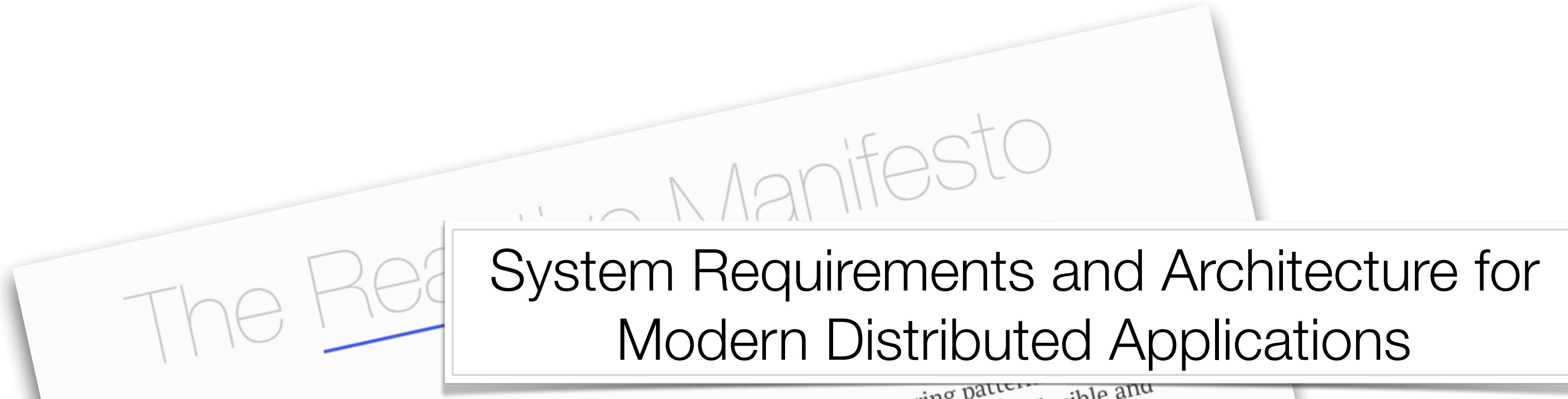
These changes are happening because application requirements have changed dramatically in recent years. Only a few years ago a large application had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile 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 it easier to develop and amenable to change. They are significantly more tolerant of failure as they meet it with elegance rather than disaster. Reactive Systems provide effective interactive feedback.



# THE REACTIVE MANIFESTO



Organisations working in disparate domains are independently discovering patterns of 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 it easier to develop and amenable to change. They are significantly more tolerant of failure as they occur they meet it with elegance rather than disaster. Reactive Systems provide effective interactive feedback.



# THE REACTIVE MANIFESTO

The Reactive Manifesto

## System Requirements and Architecture for Modern Distributed Applications

Published on September 16, 2014. (v2.0)

Organisations working software that look the better positioned to n These changes are ha recent years. Only a time, hours of offli everything from m processors. Users Petabytes. Today' We believe that necessary aspect Resilient, Elast

Systems built as Reactive Systems are easier to develop and amenable to change. They occur they meet it with elegance rather than effective interactive feedback.

```
graph TD; A[Responsive] <--> B[Resilient]; A <--> C[Message Driven]; B <--> D[Elastic]; C <--> D;
```



# 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



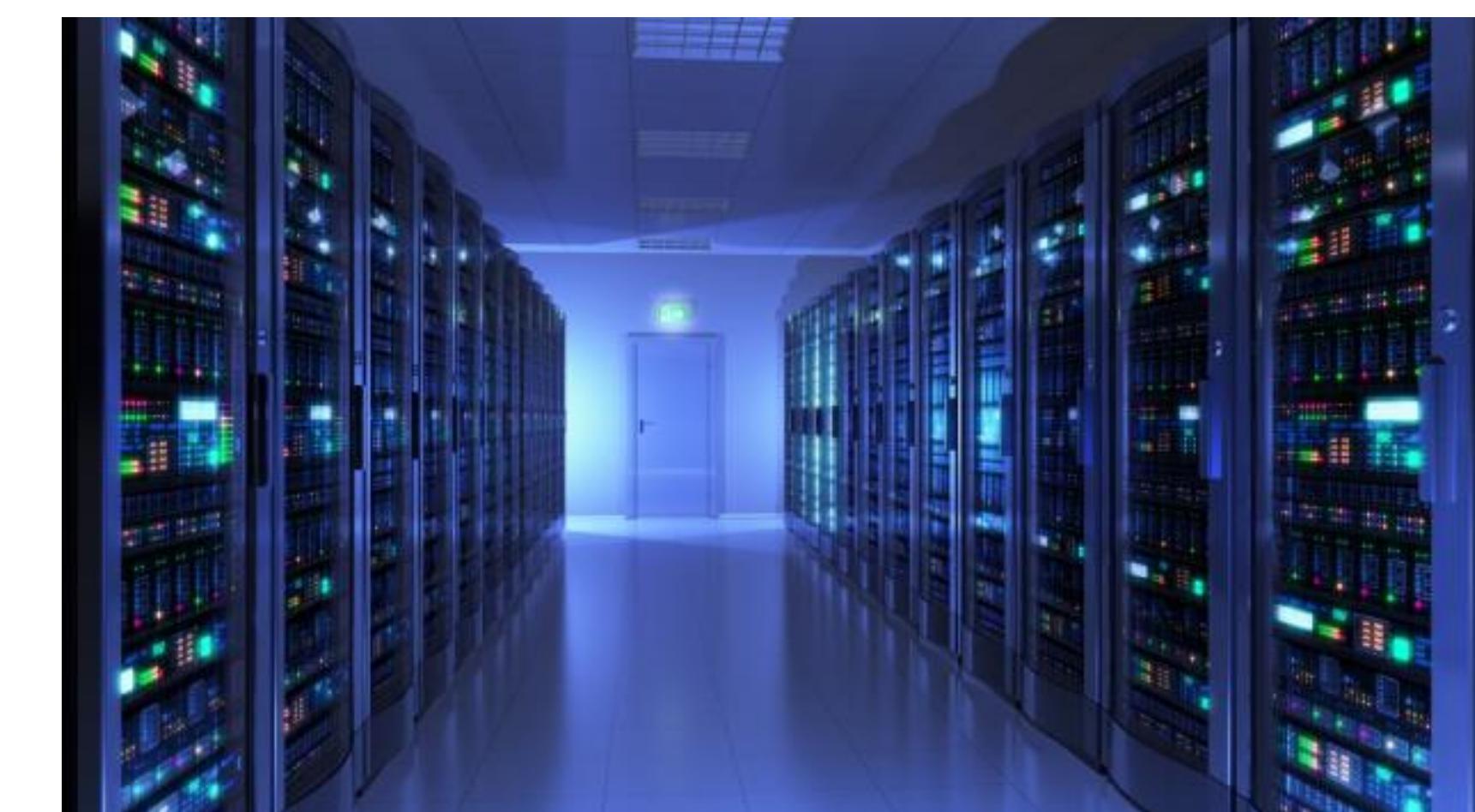
1980



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 event-driven control by signal, back-pressure, and communication, we can 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.

## How do we program for The Reactive Manifesto?

icity, and flow  
and applying  
means of  
ork with the

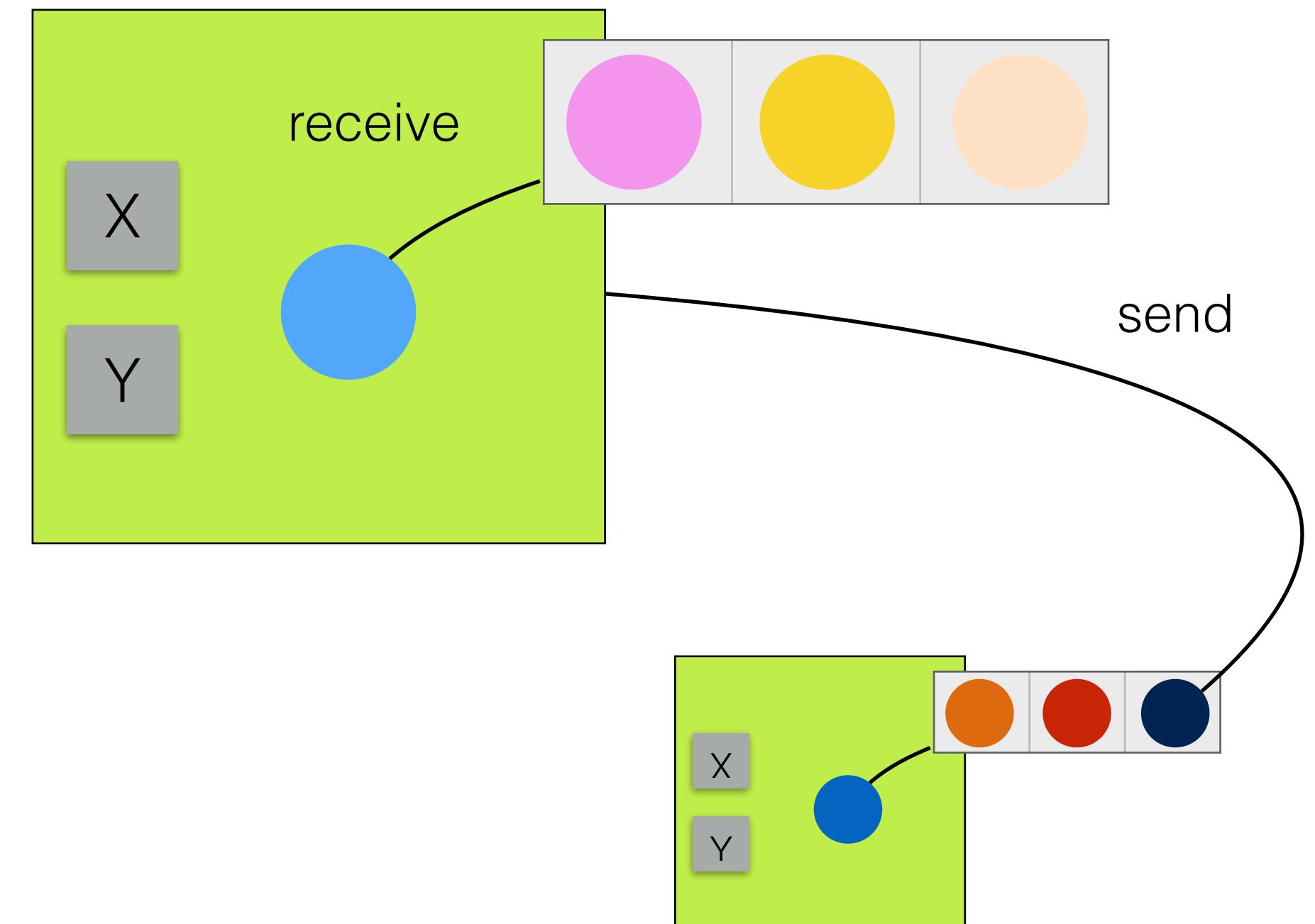
# ACTOR PROGRAMMING

# ACTOR EXECUTION MODEL

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

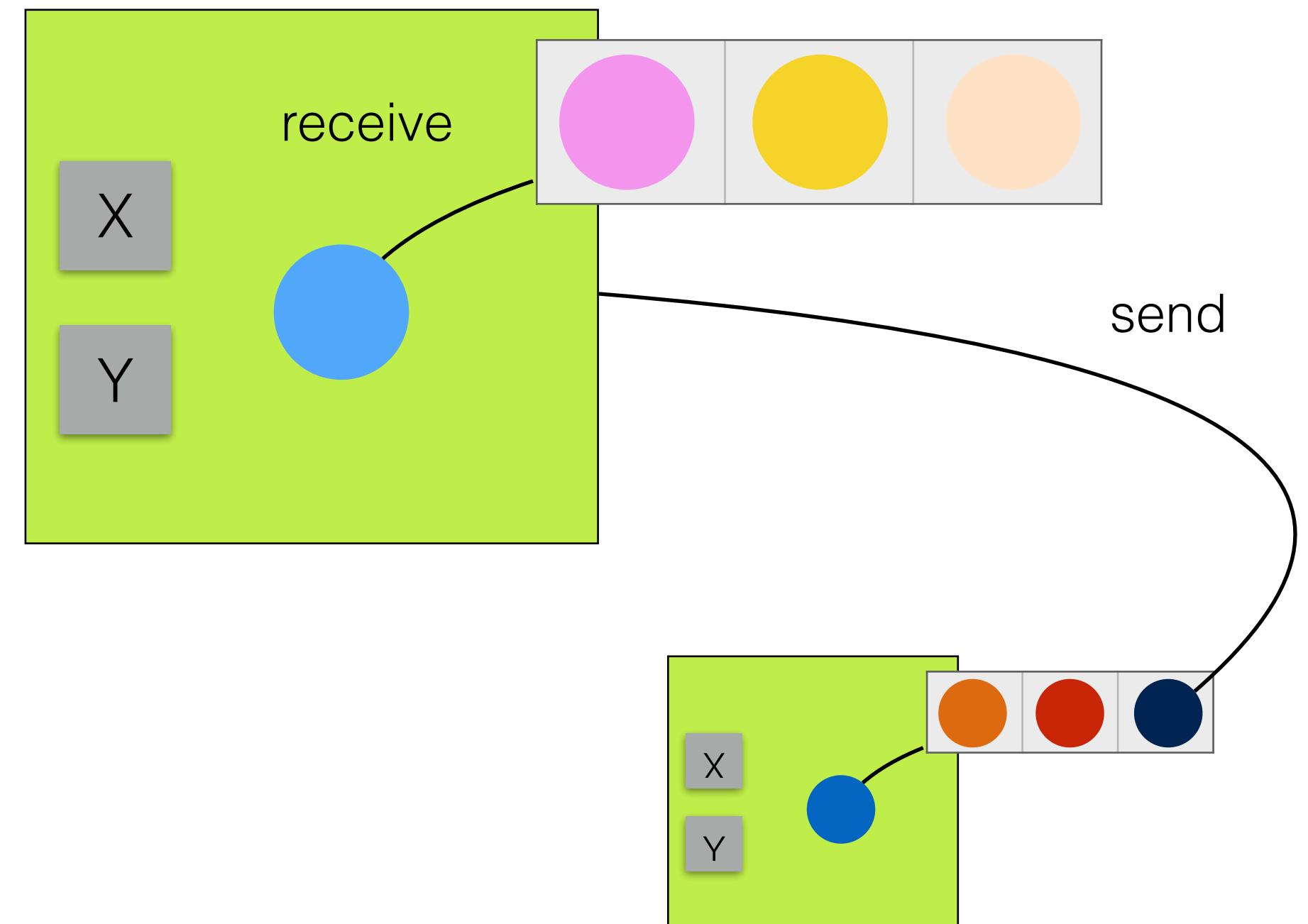
# WHAT IS AN ACTOR

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



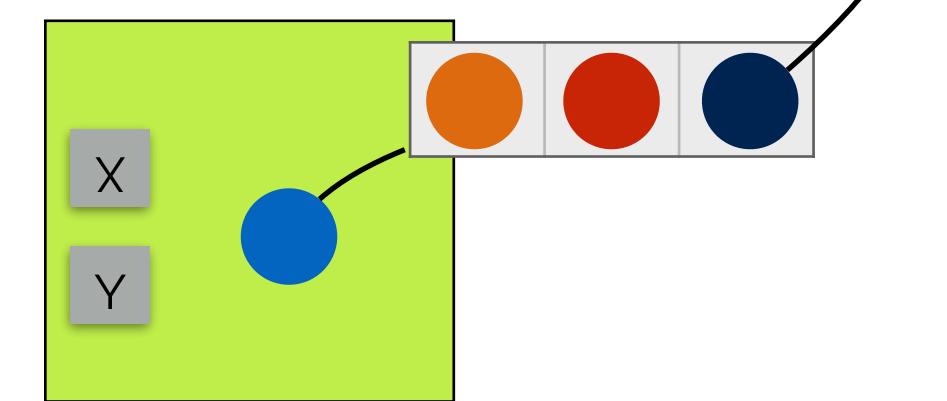
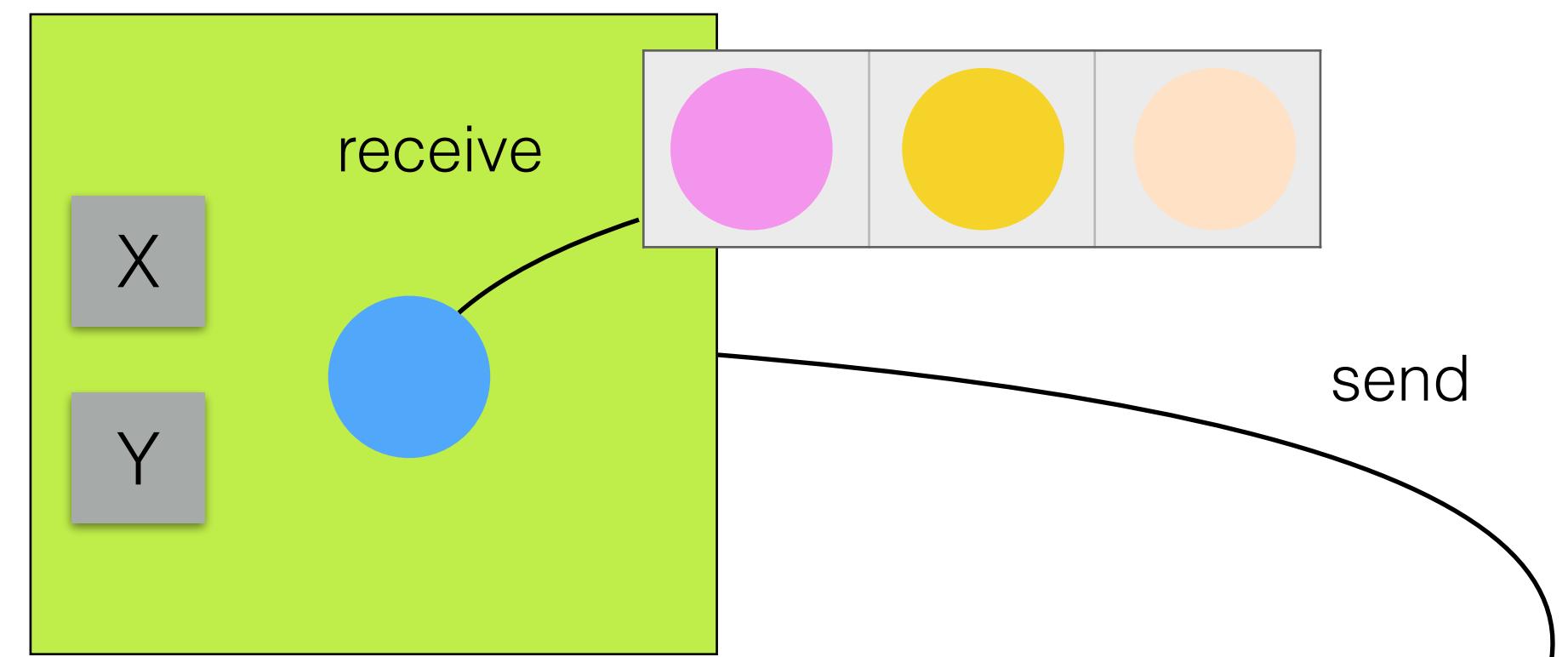
# WHAT IS AN ACTOR

- ▶ 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?

```
TS0.x = 0;
TS0.y = 0;
TS0.r1 = 0;
TS0.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
            TS0.x = 1;
            TS0.r1 = TS0.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
            TS0.y = 1;
            TS0.r2 = TS0.x;
        }
};
```

$$\begin{array}{ll} x = y = 0 & \\ x := 1; & \parallel \quad y := 1; \\ r_0 = y & \quad \quad \quad r_1 = x \\ r_0 = r_1 = 0 & \end{array}$$

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

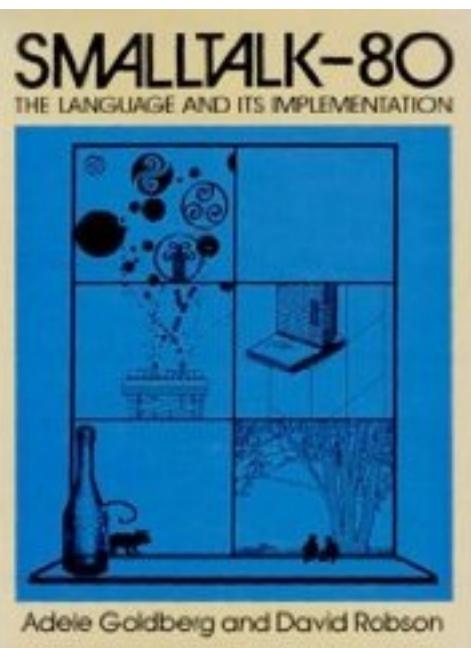
# PROGRAMMING WITH ACTORS

- ▶ 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

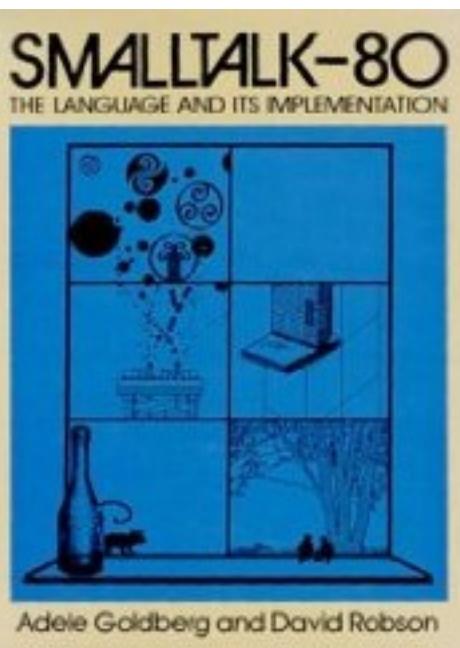
# PROGRAMMING WITH ACTORS

- ▶ 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

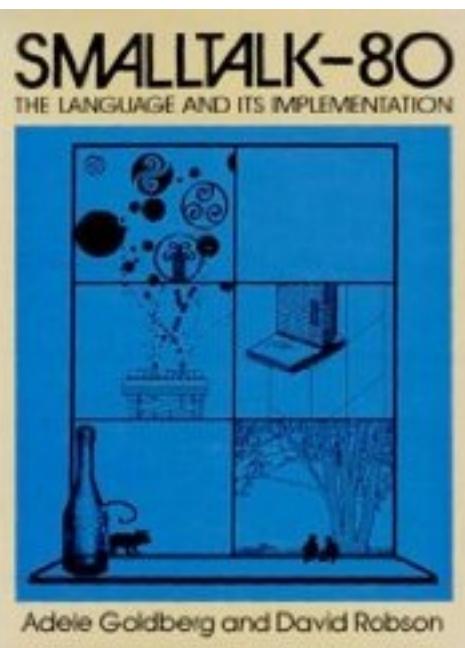


# A NOTE ON SMALLTALK



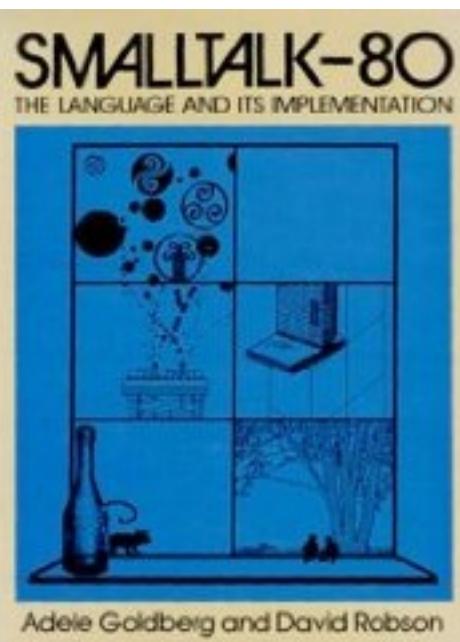
- ▶ Created in the '70 by Alan Kay and other
  - ▶ Introduces Objects  
“abstract representation of a concrete entity”
  - ▶ Everything is an object
  - ▶ Interaction with objects is through message/response
  - ▶ Methods bind messages to functionality
  - ▶ Method binding is dynamic

# A NOTE ON SMALLTALK



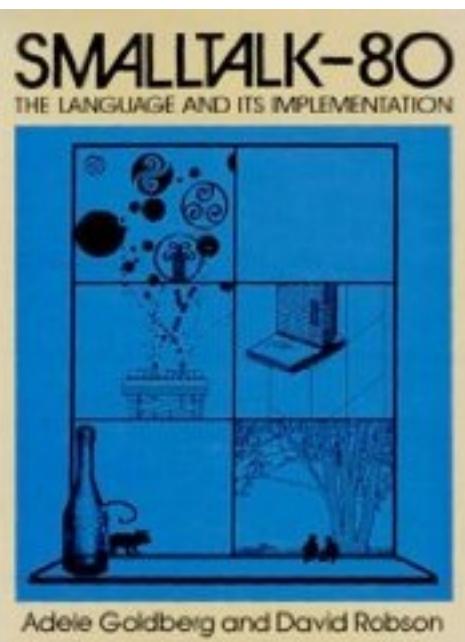
- ▶ Created in the '70 by Alan Kay and other
- ▶ Introduces Objects  
“abstract representation of a concrete entity”
- ▶ Everything is an object
- ▶ Interaction with objects is through message/response
- ▶ Methods bind messages to functionality
- ▶ Method binding is dynamic
- ▶ Classes define the representation 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 entity”
- ▶ Everything is an object
- ▶ Interaction with objects through message-response
- ▶ Classes define the representation of objects
- ▶ Class Protocol
- ▶ Code reuse
- ▶ Generalization
- ▶ Specializatoin
- ▶ Purist Object Orientation
  - ▶ Everything is an object
  - ▶ No conditionals:
    - ed as messages
    - False
  - ▶ was to construct for all foreseeable 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
- ▶ Key characteristics
  - ▶ Distributed
  - ▶ Fault-tolerant
  - ▶ Soft real-time
  - ▶ Highly available
  - ▶ Hot swapping
- ▶ 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
- ▶ Popularizes Actors:
  - ▶ No shared state
  - ▶ Lightweight processes
  - ▶ Asynch message-passing
  - ▶ Mailboxes messages

# ERLANG'S HELLO WORLD



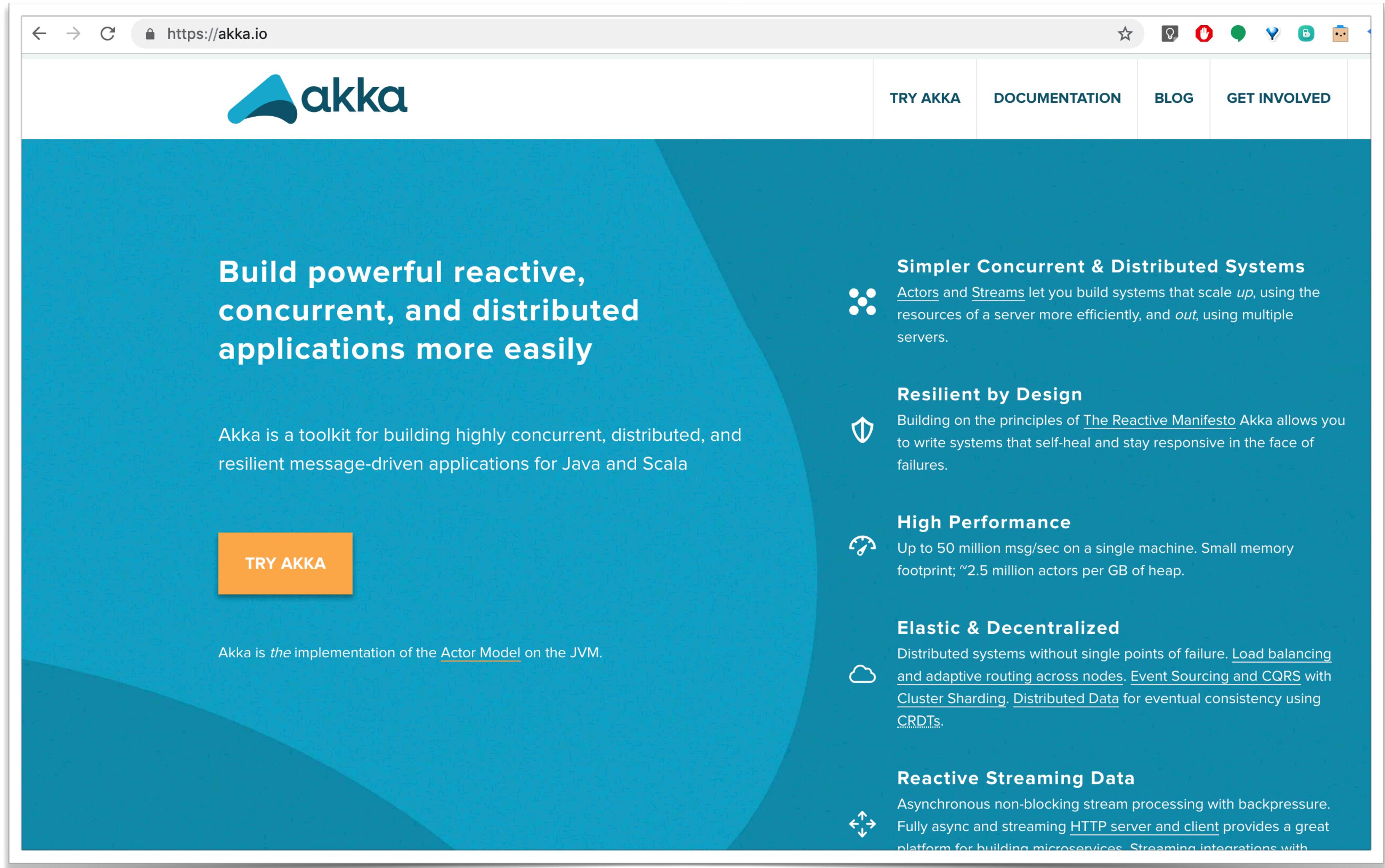
```
ping(0, PongNode) ->
    {pong, PongNode} ! finished,
    io:format("ping finished~n", []);
ping(N, PongNode) ->
    {pong, PongNode} ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong~n", [])
    end,
    ping(N - 1, PongNode).

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



The screenshot shows the official Akka website at <https://akka.io>. The page features a large teal header with the Akka logo and navigation links for TRY AKKA, DOCUMENTATION, BLOG, and GET INVOLVED. Below the header, a main section highlights the ability to "Build powerful reactive, concurrent, and distributed applications more easily". A central orange button labeled "TRY AKKA" is visible. To the right, five key features are listed with corresponding icons:

- 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 platform for building microservices. [Streaming integrations with](#)

# 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

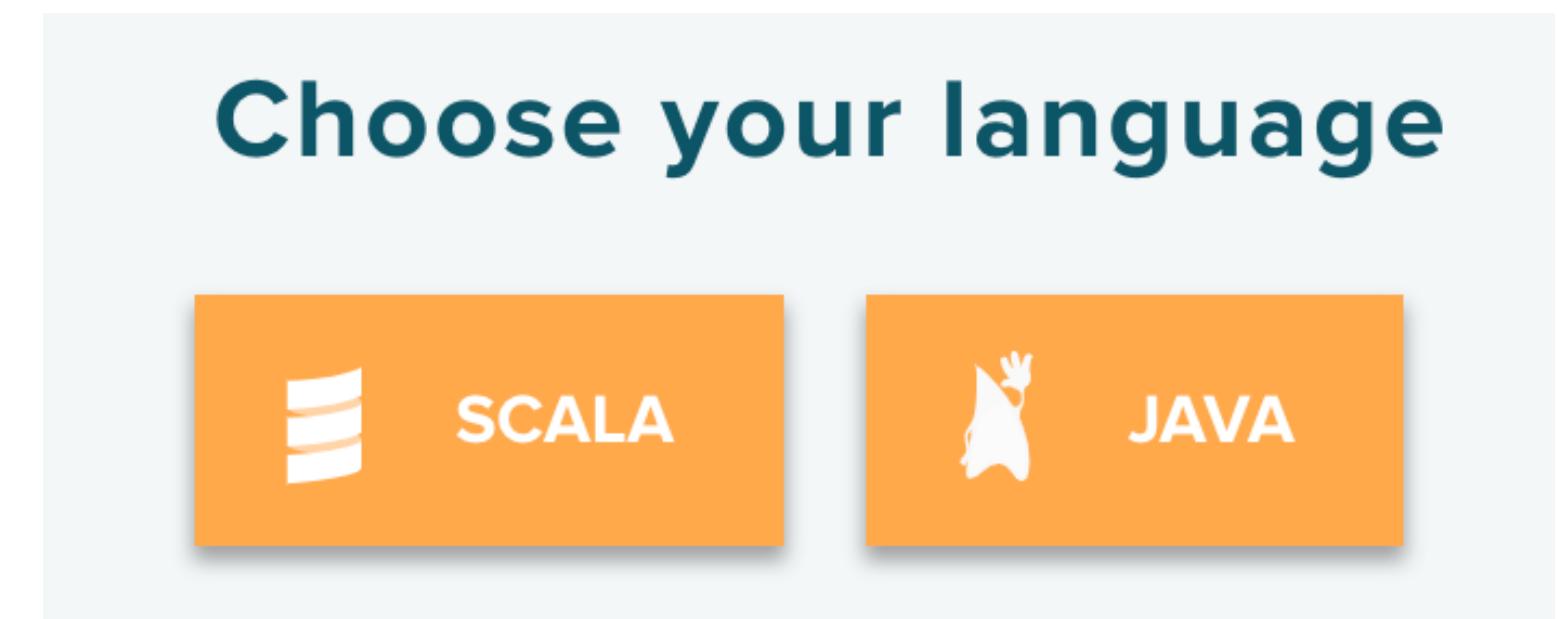


- ▶ We will be using Scala for the lectures, but ...

# AKKA ACTORS



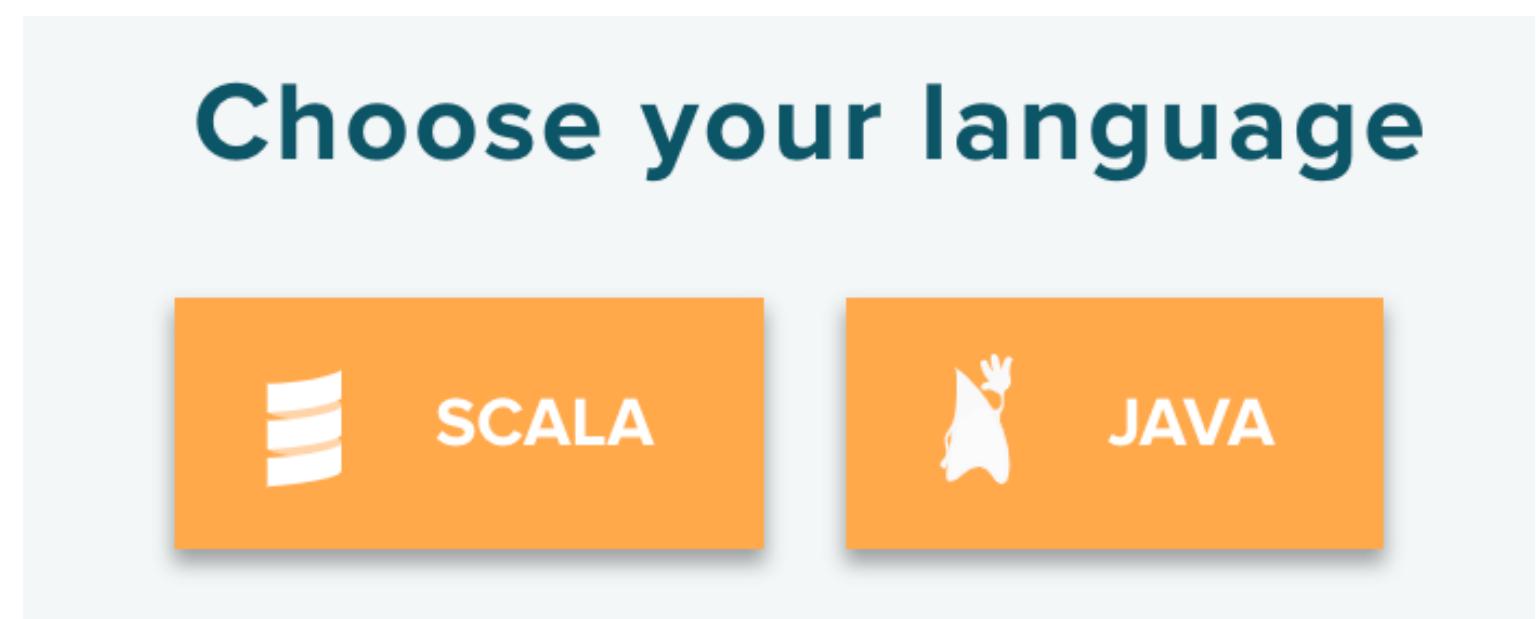
- We will be using Scala for the lectures, but ...



# AKKA ACTORS



- ▶ We will be using Scala for the lectures, but ...

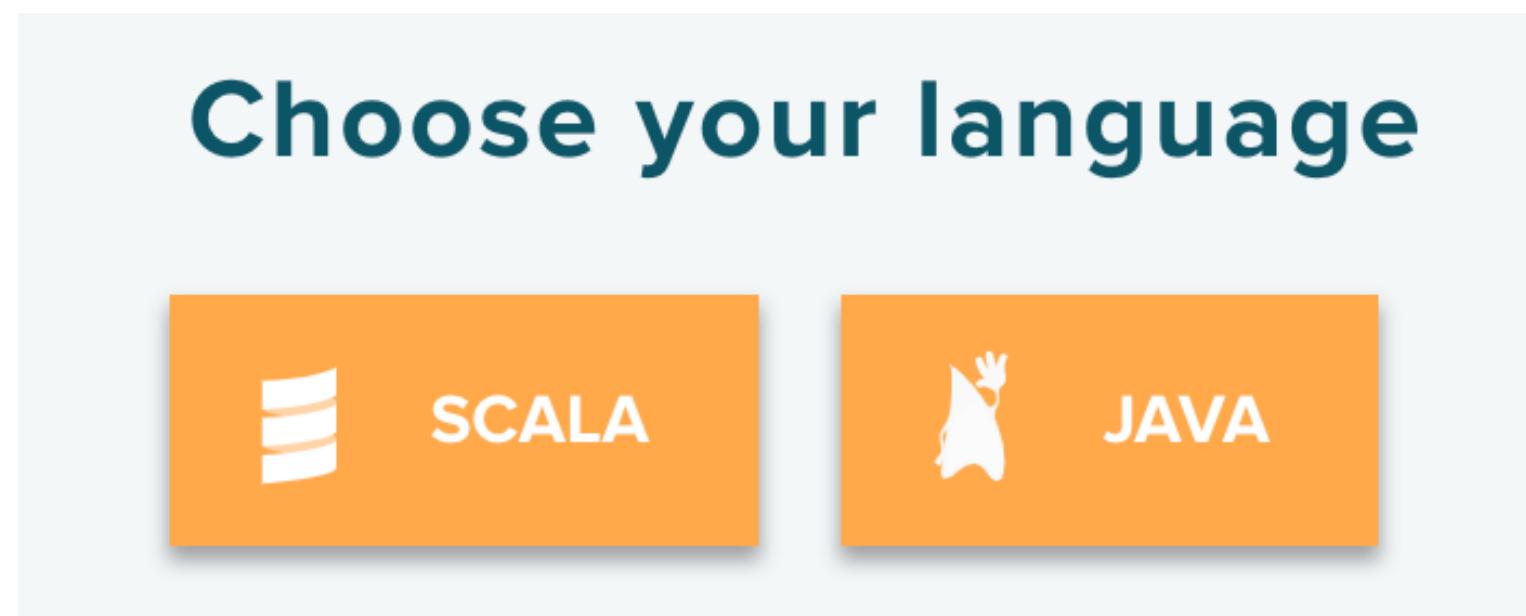


- ▶ The lectures are not Scala specific

# AKKA ACTORS



- ▶ We will be using Scala for the lectures, but ...

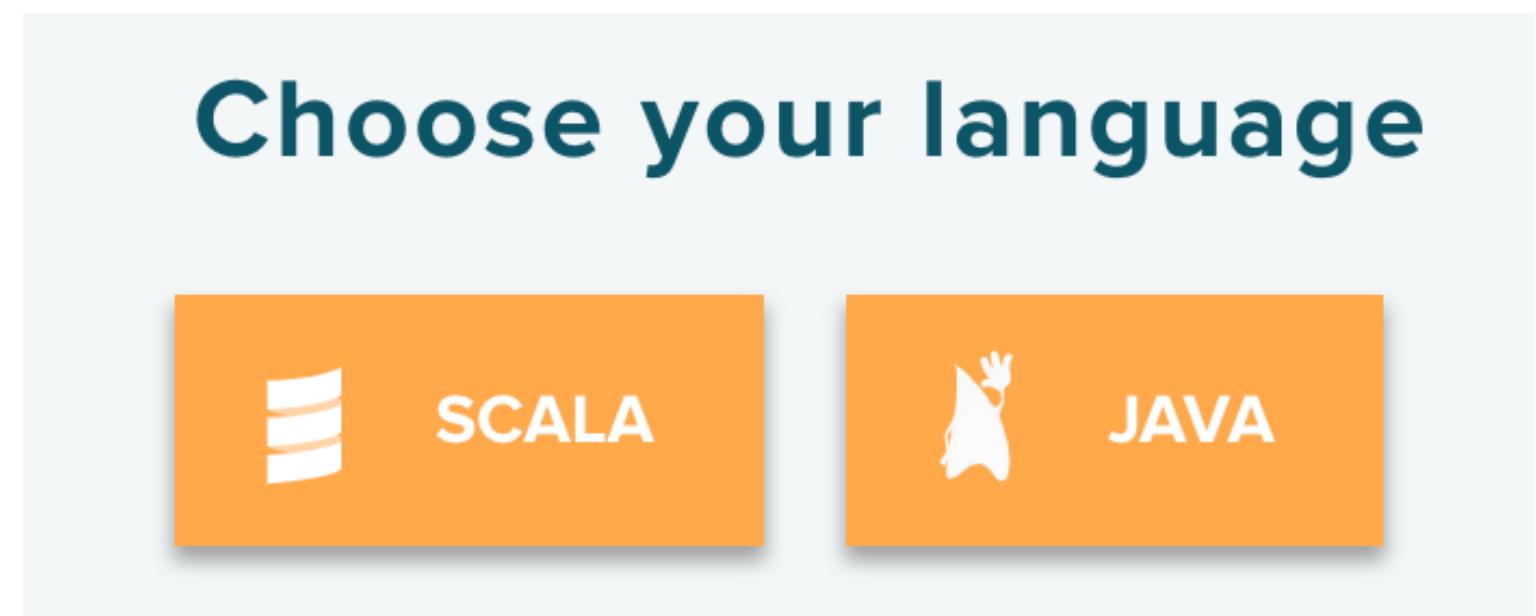


- ▶ The lectures are not Scala specific
- ▶ The documentation of Akka is great! Go try it out!

# AKKA ACTORS



- ▶ We will be using Scala for the lectures, but ...

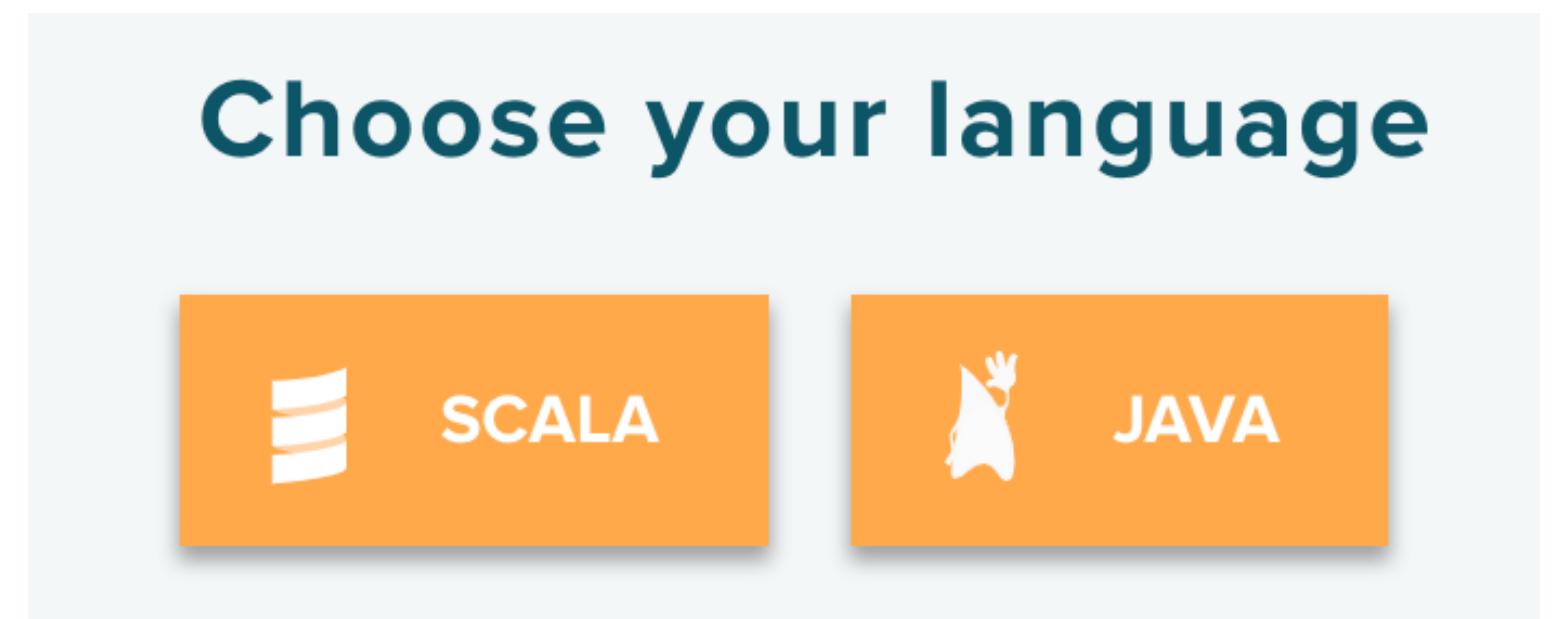


- ▶ The lectures are not Scala specific
- ▶ The documentation of Akka is great! Go try it out!
- ▶ There will be a small project at the end of the course

# AKKA ACTORS



- ▶ We will be using Scala for the lectures, but ...

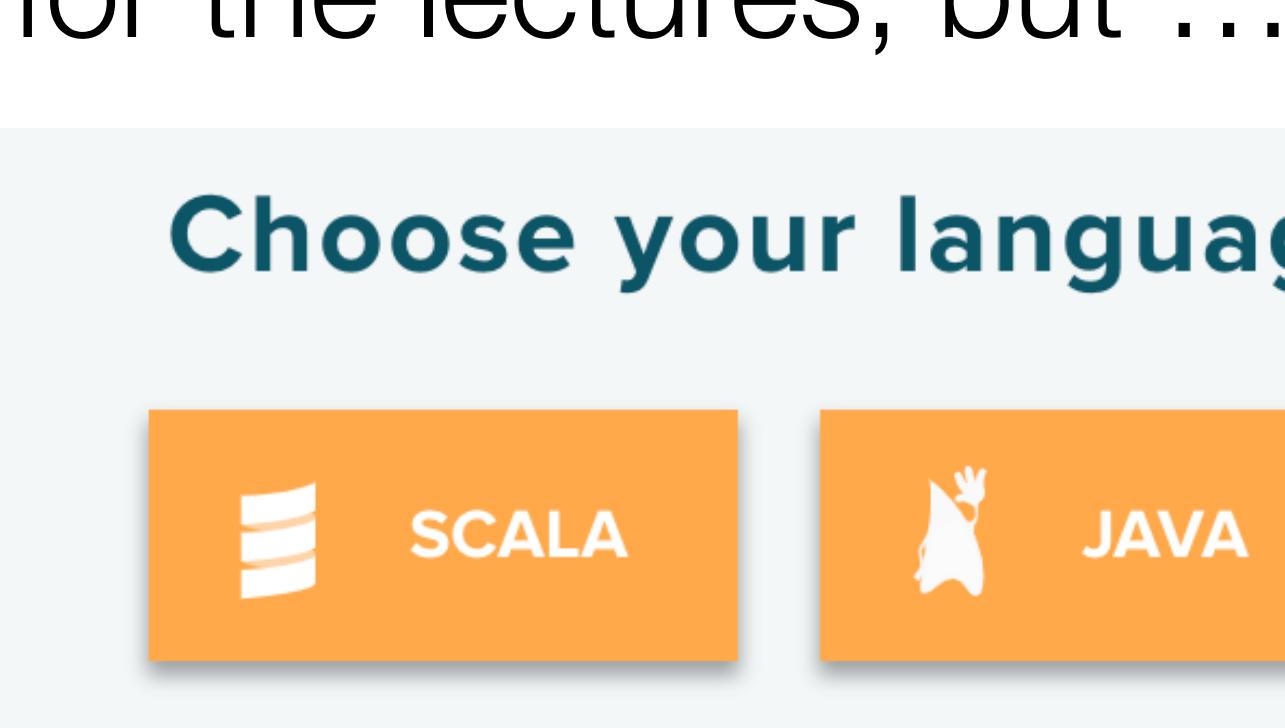


- ▶ The lectures are not Scala specific
- ▶ The documentation of Akka is great! Go try it out!
- ▶ There will be a small project at the end of the course
- ▶ Start reading the documentation of Akka soon!

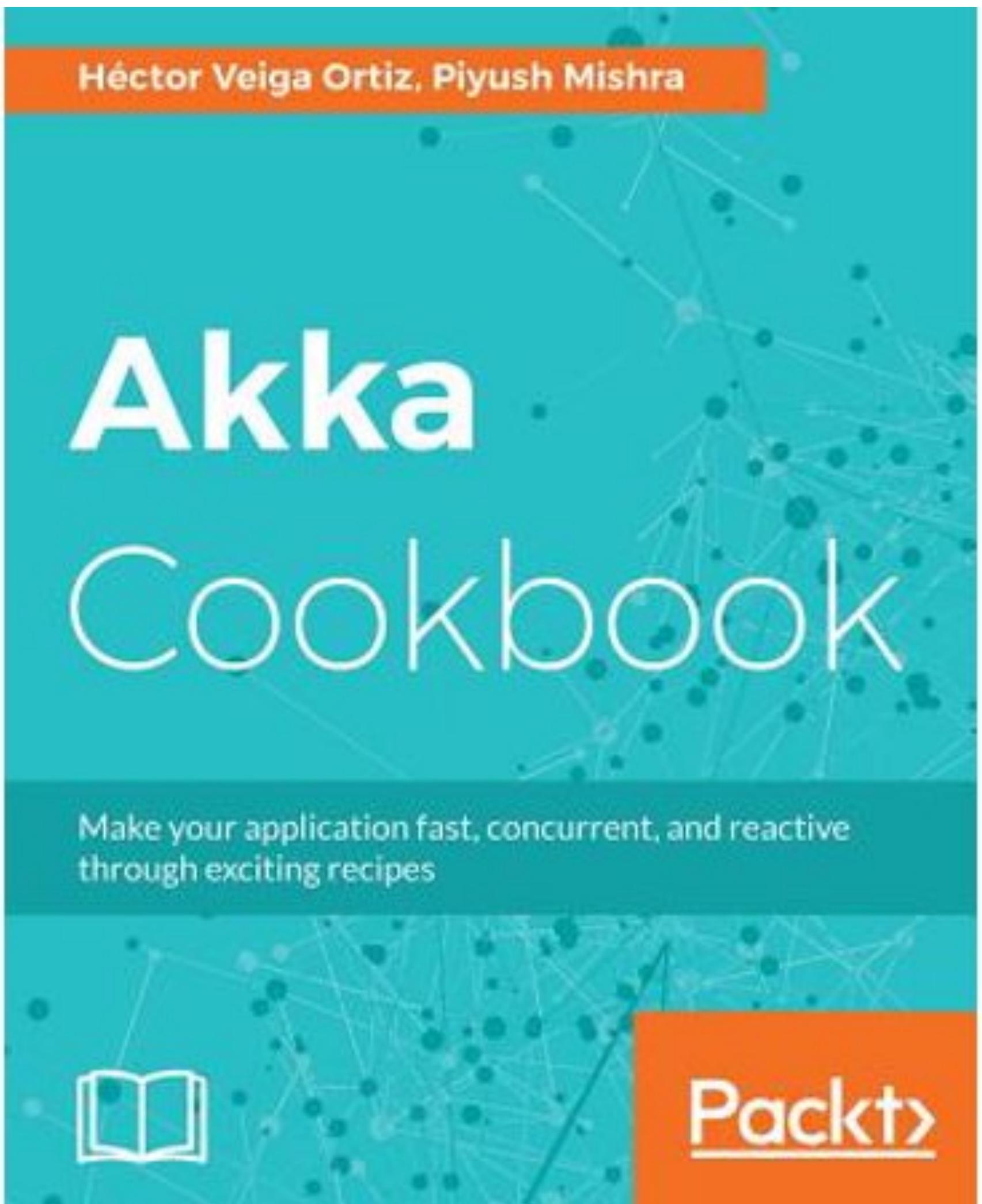
# AKKA ACTORS



- ▶ 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 so

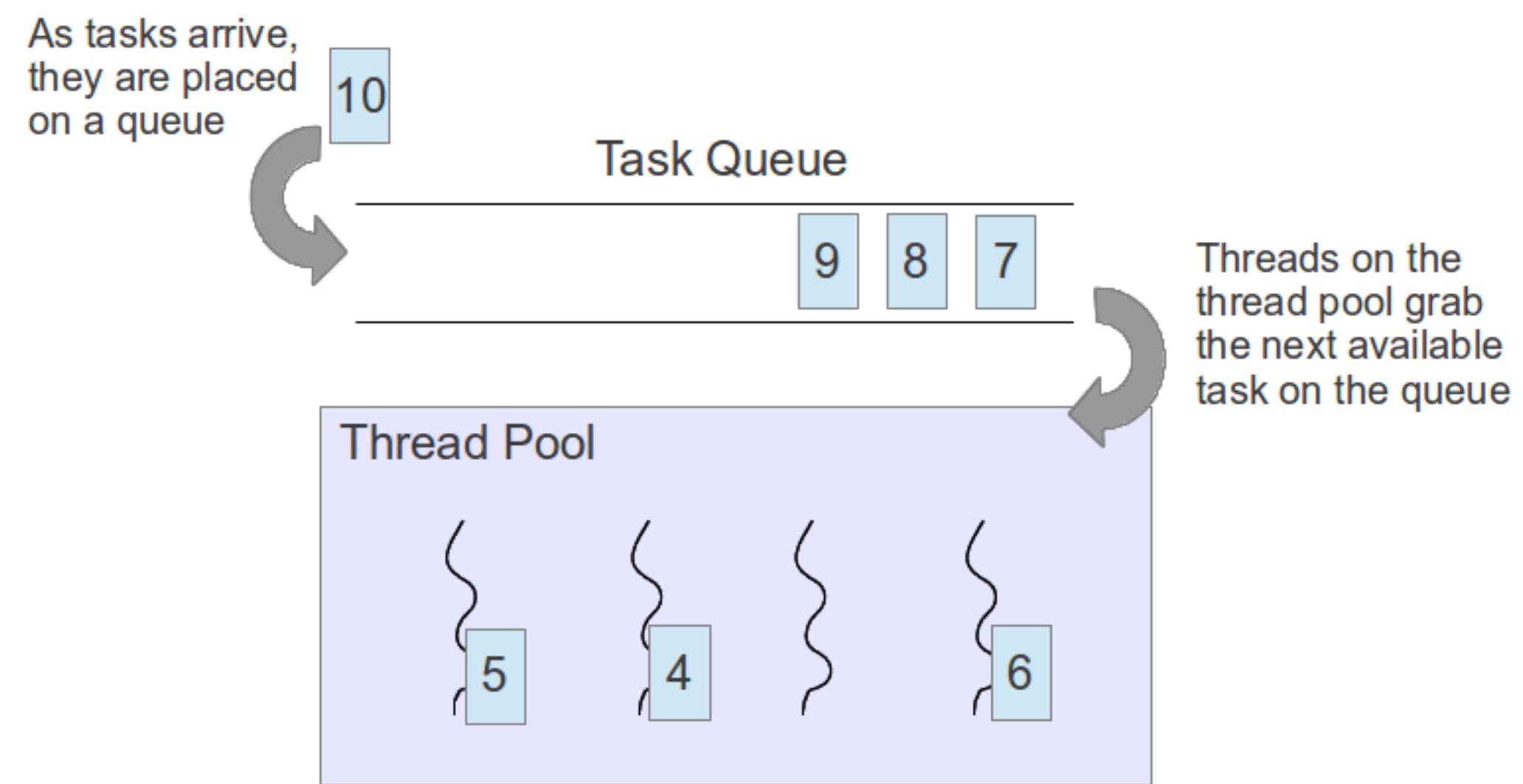


# ACTOR SYSTEM: CREATION

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

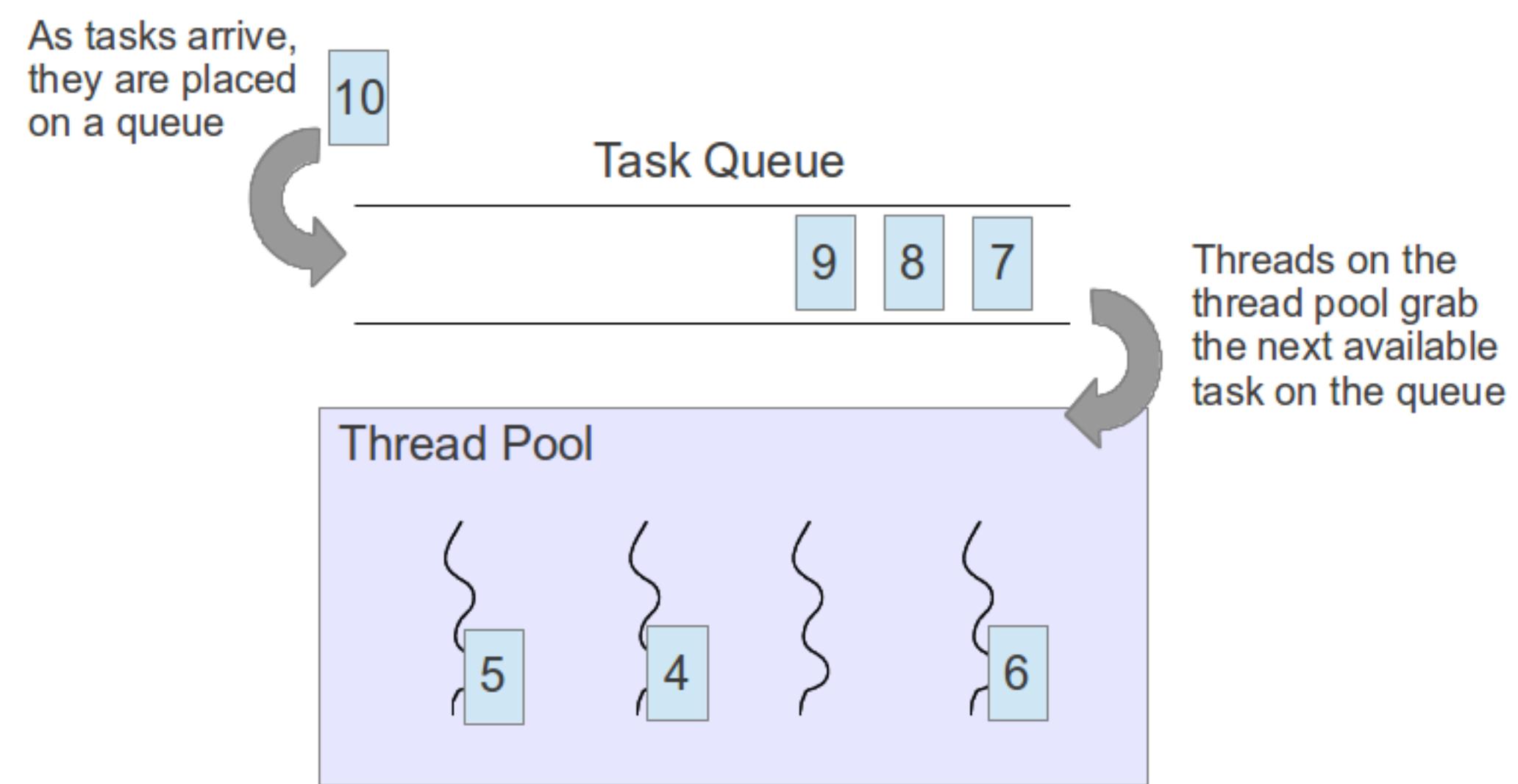
# ACTOR SYSTEM: CREATION

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



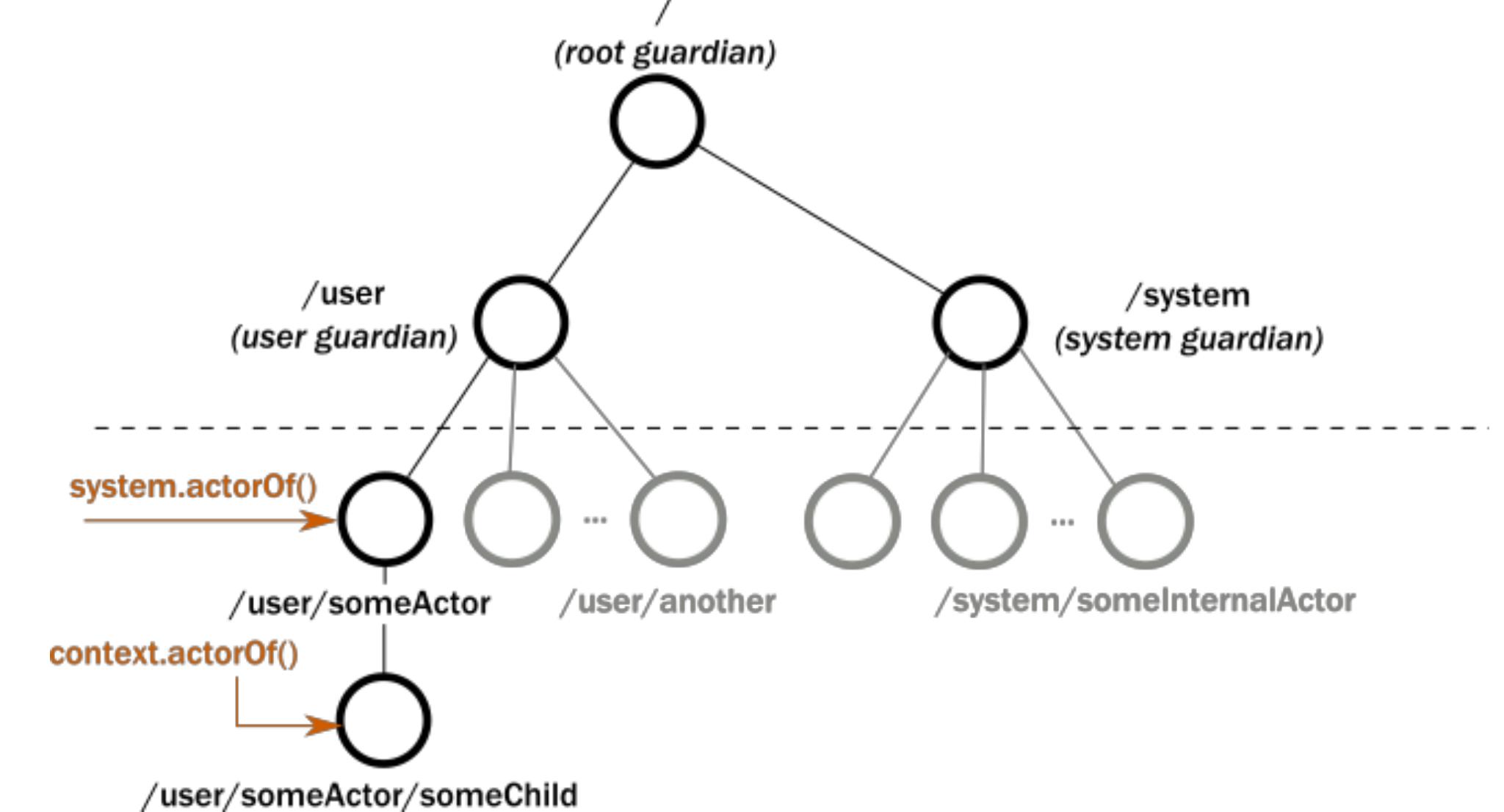
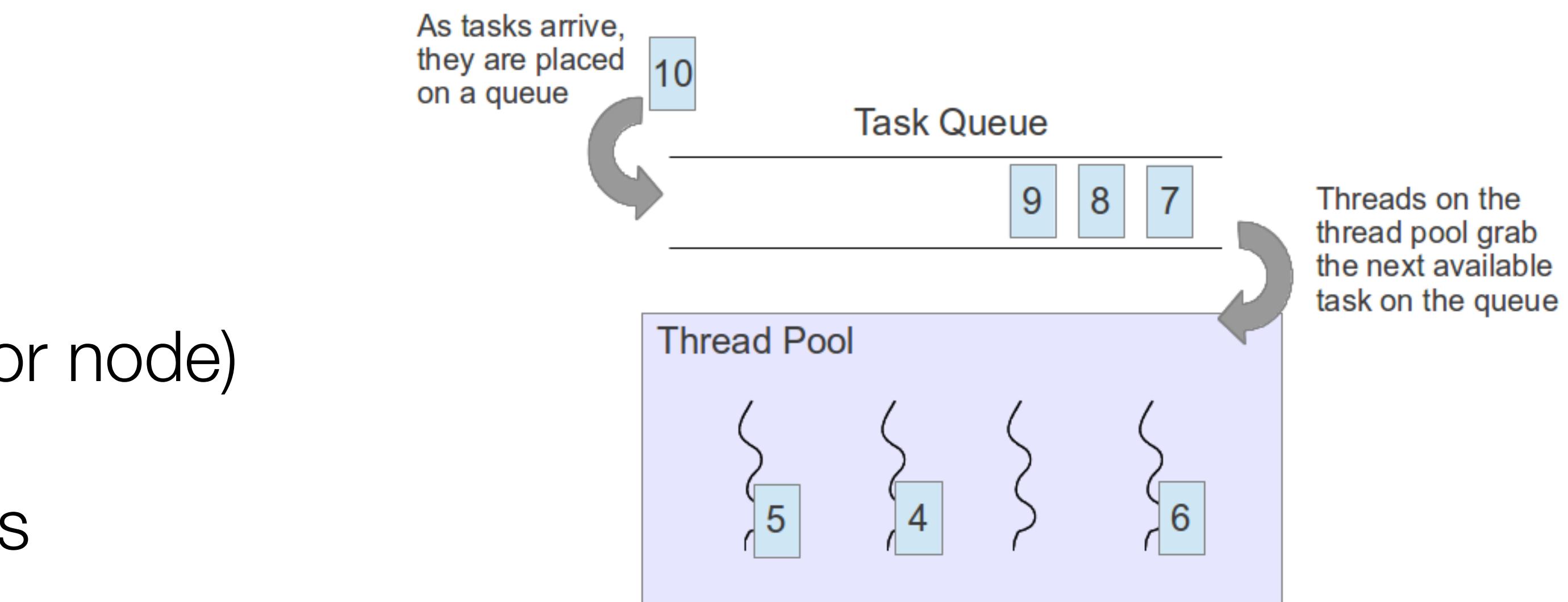
# ACTOR SYSTEM: CREATION

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



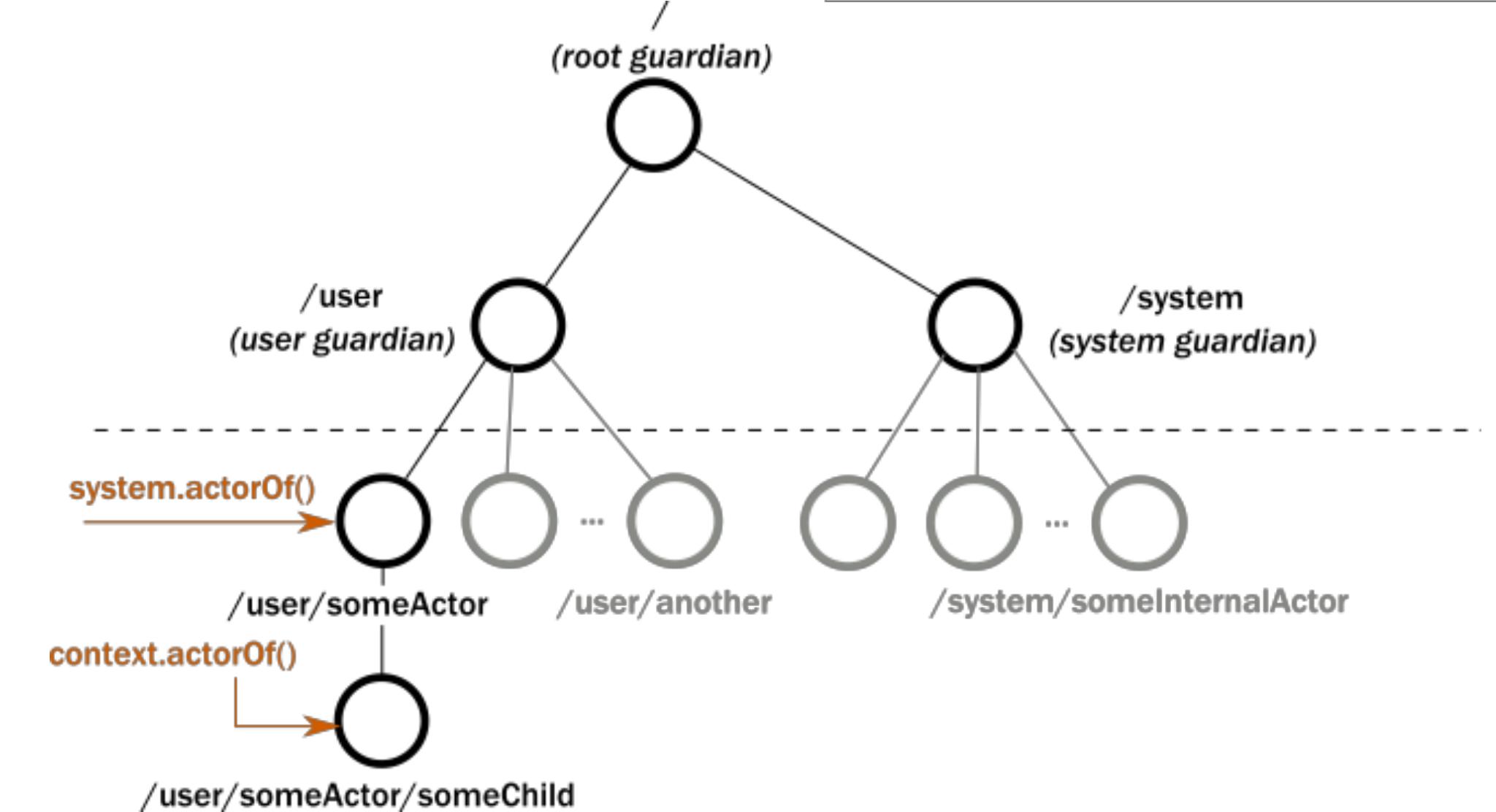
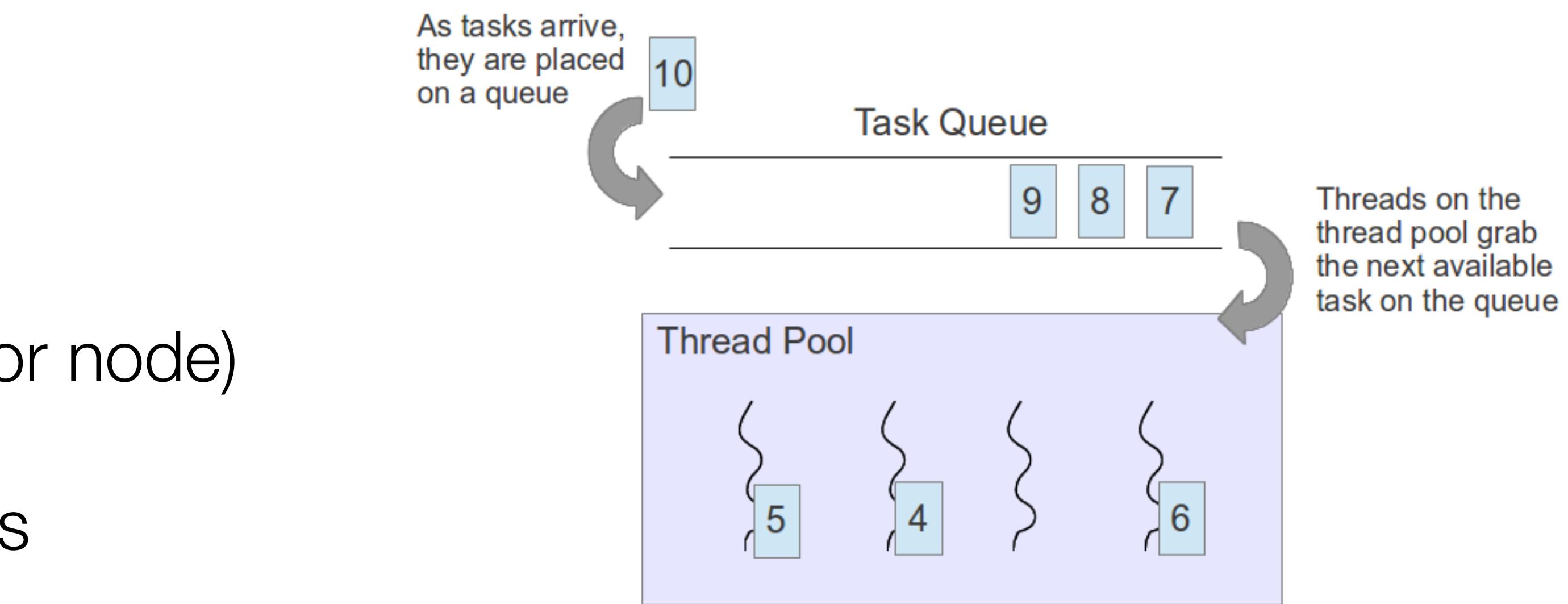
# ACTOR SYSTEM: CREATION

- ▶ 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)



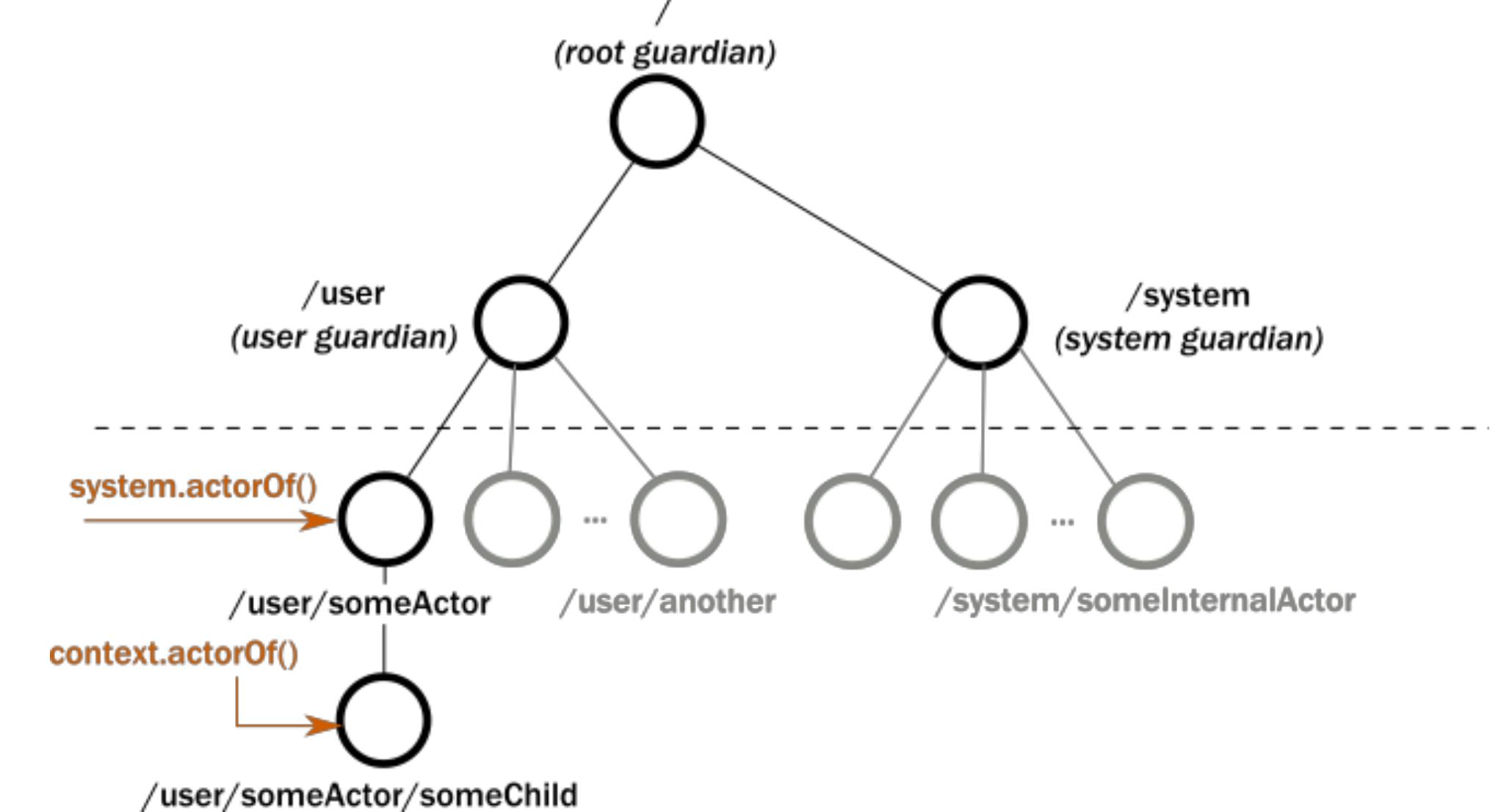
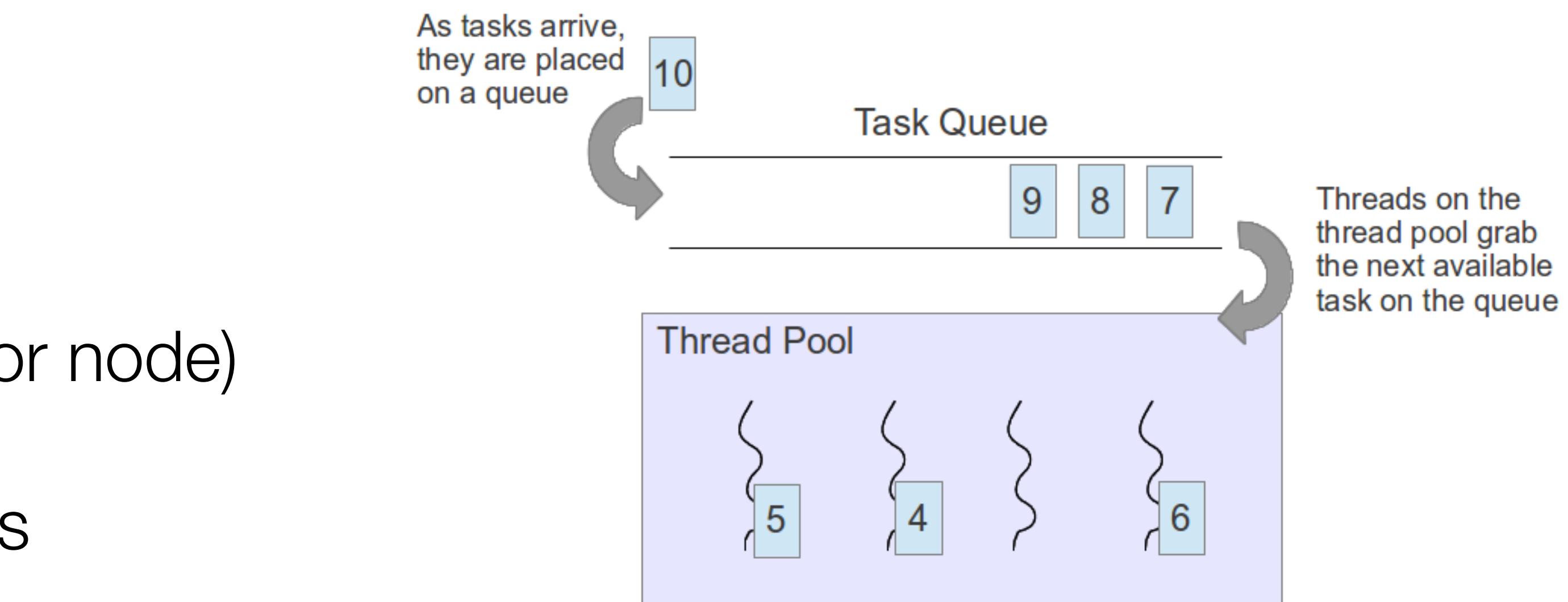
# ACTOR SYSTEM: CREATION

- ▶ 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

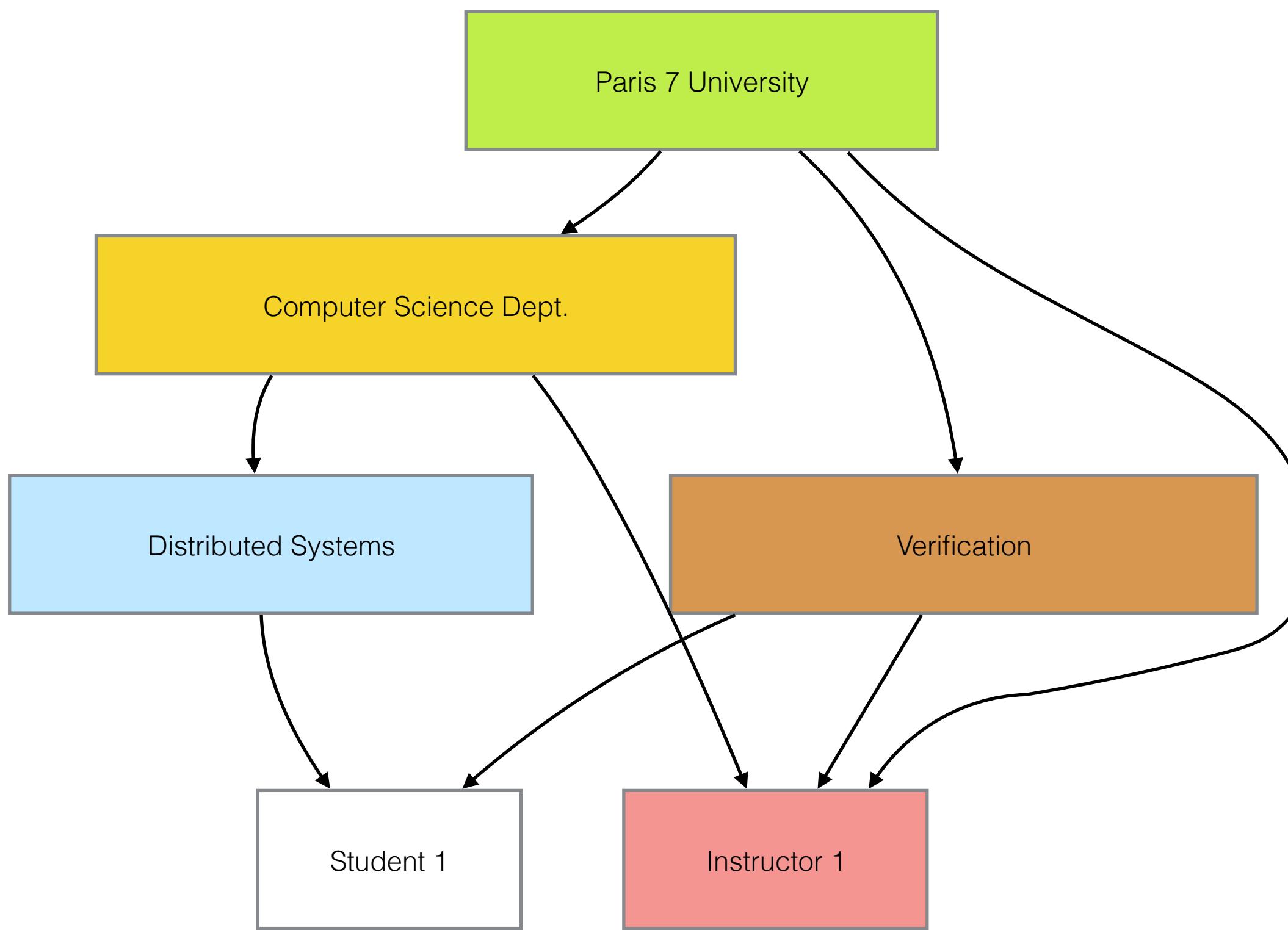


○ Exercise 1: Create an UM6P Actor System

# WHAT IS AN ACTOR?

► State

An actor system runtime

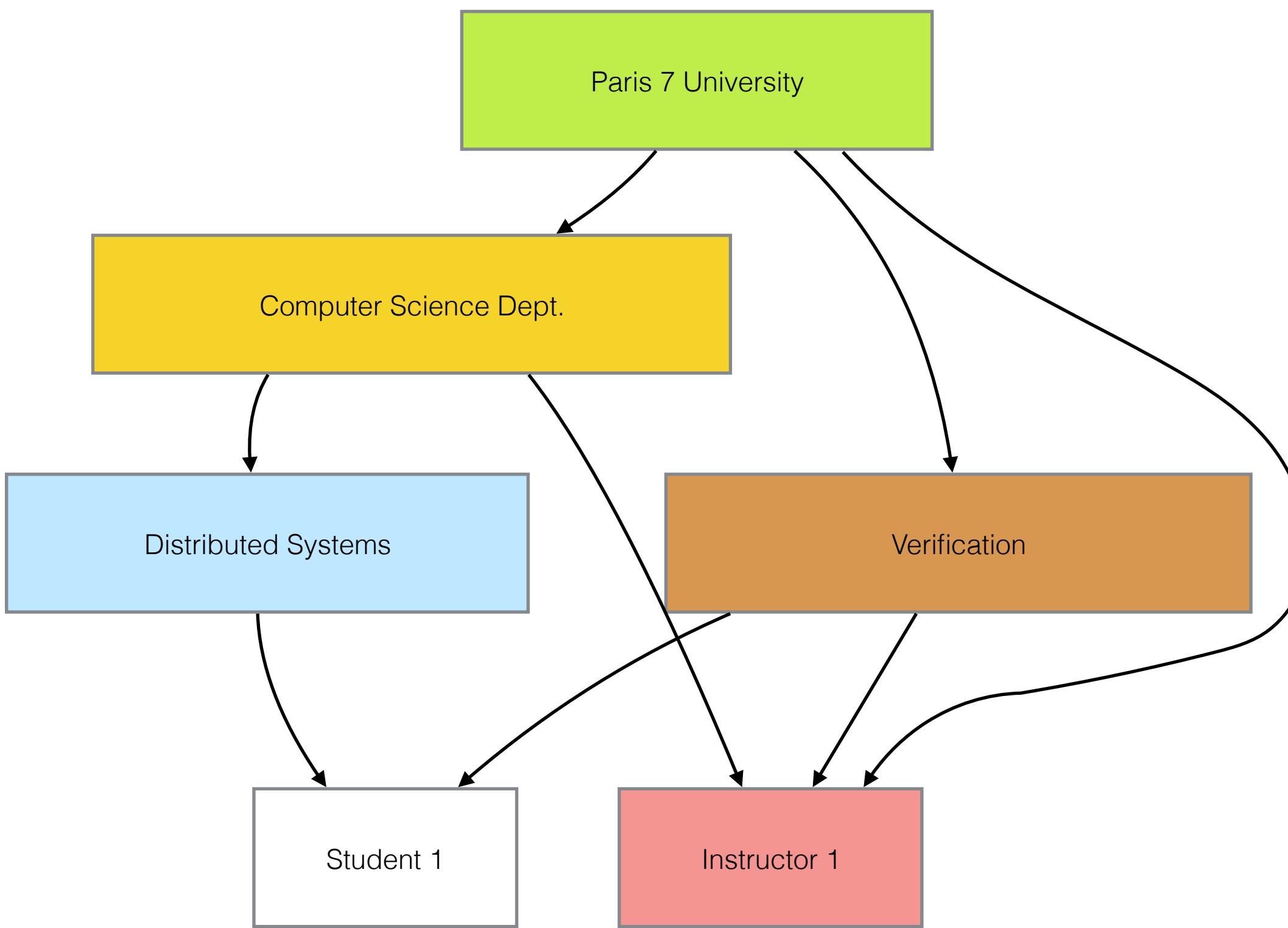


Two actors

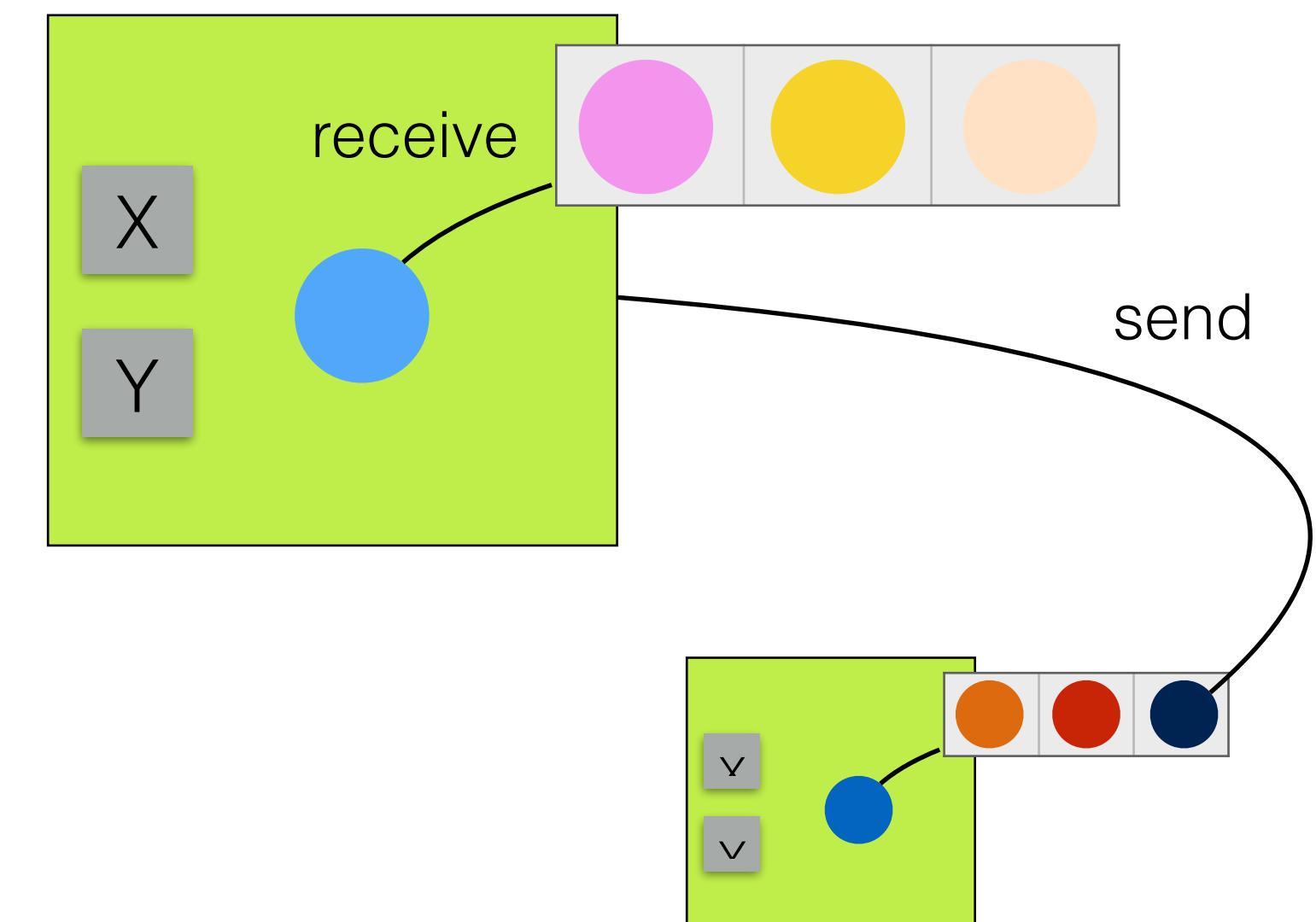
# WHAT IS AN ACTOR?

- ▶ State
- ▶ Messages

An actor system runtime



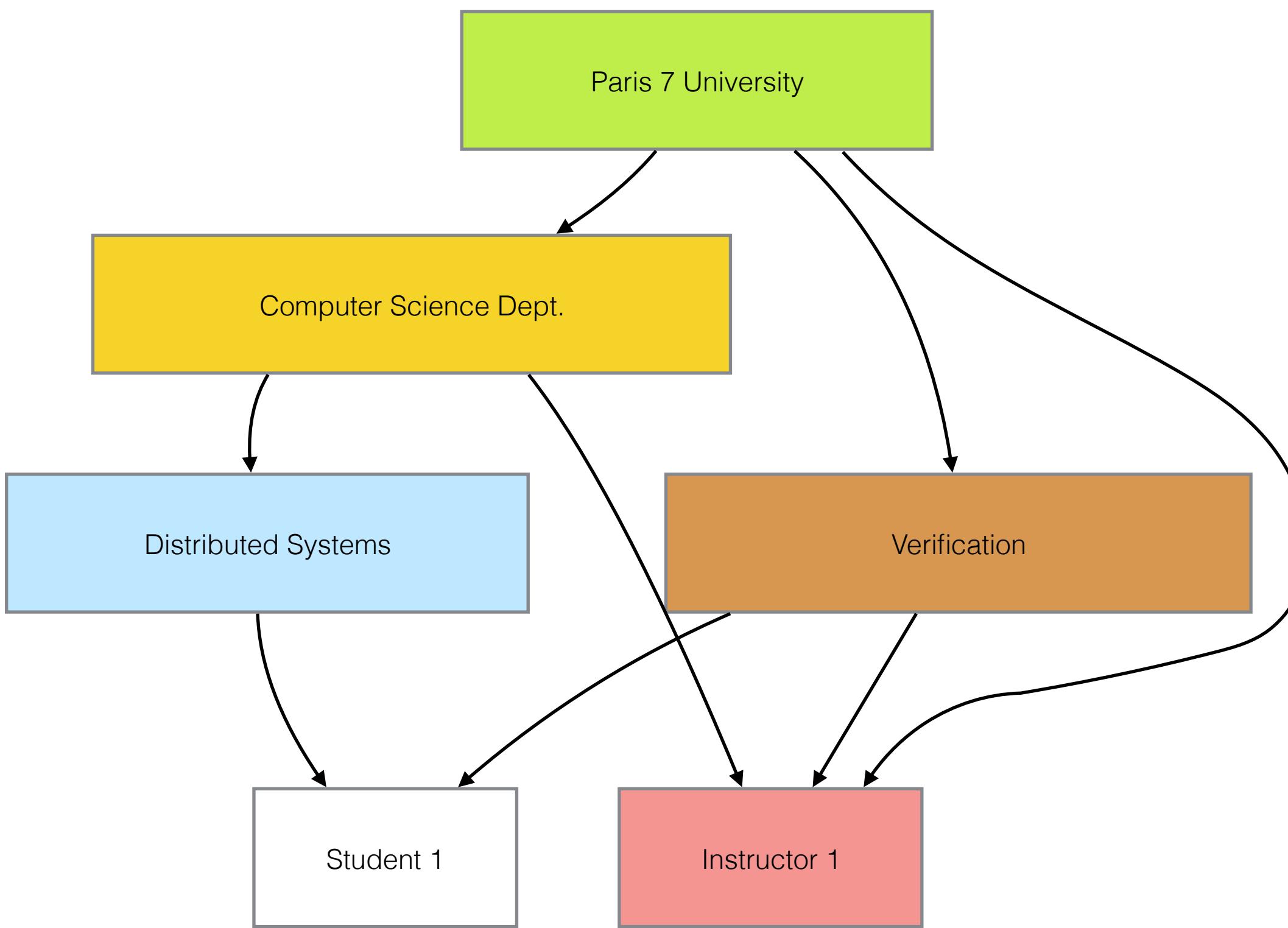
Two actors



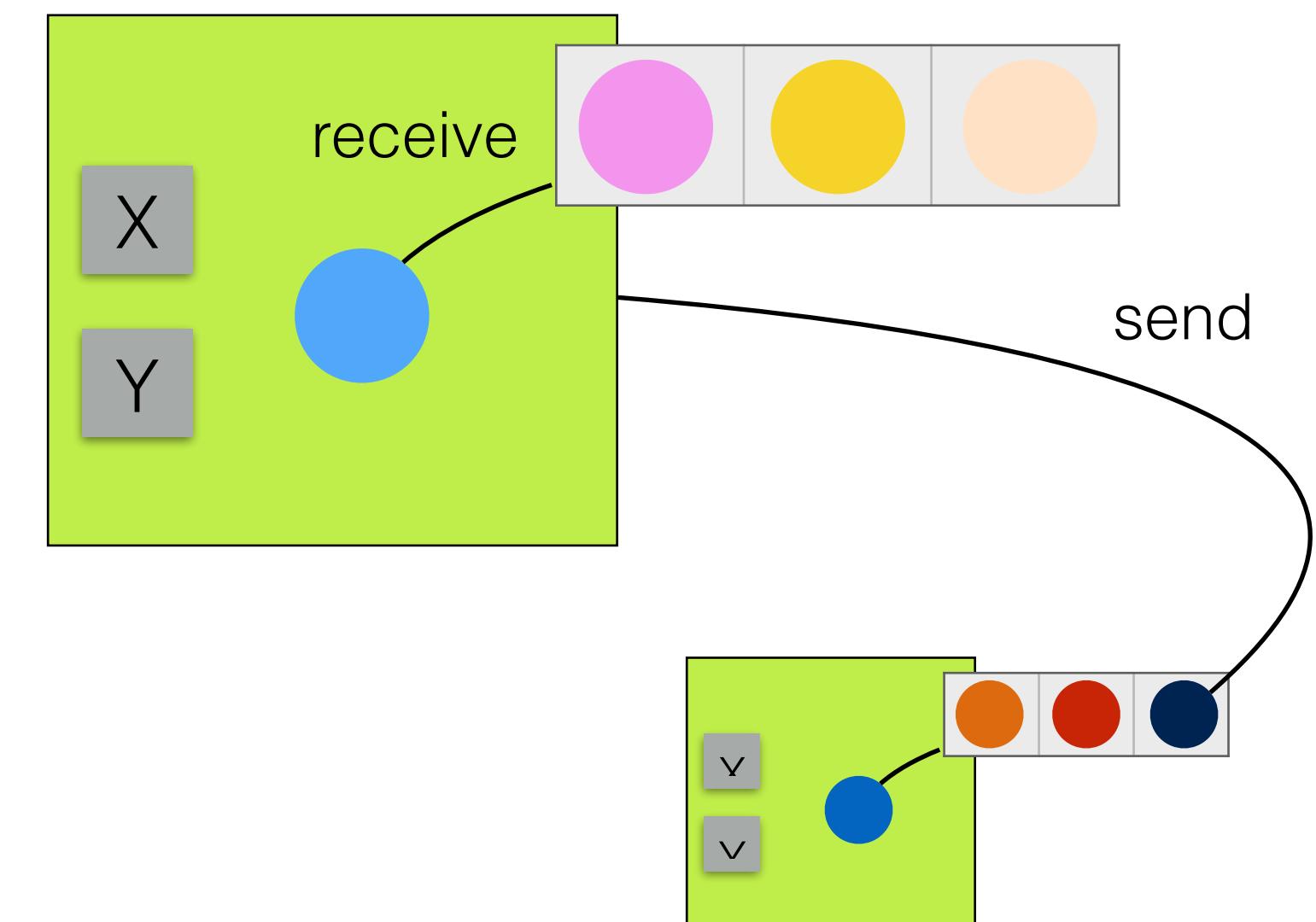
# WHAT IS AN ACTOR?

- ▶ State
- ▶ Messages
- ▶ Inbox

An actor system runtime



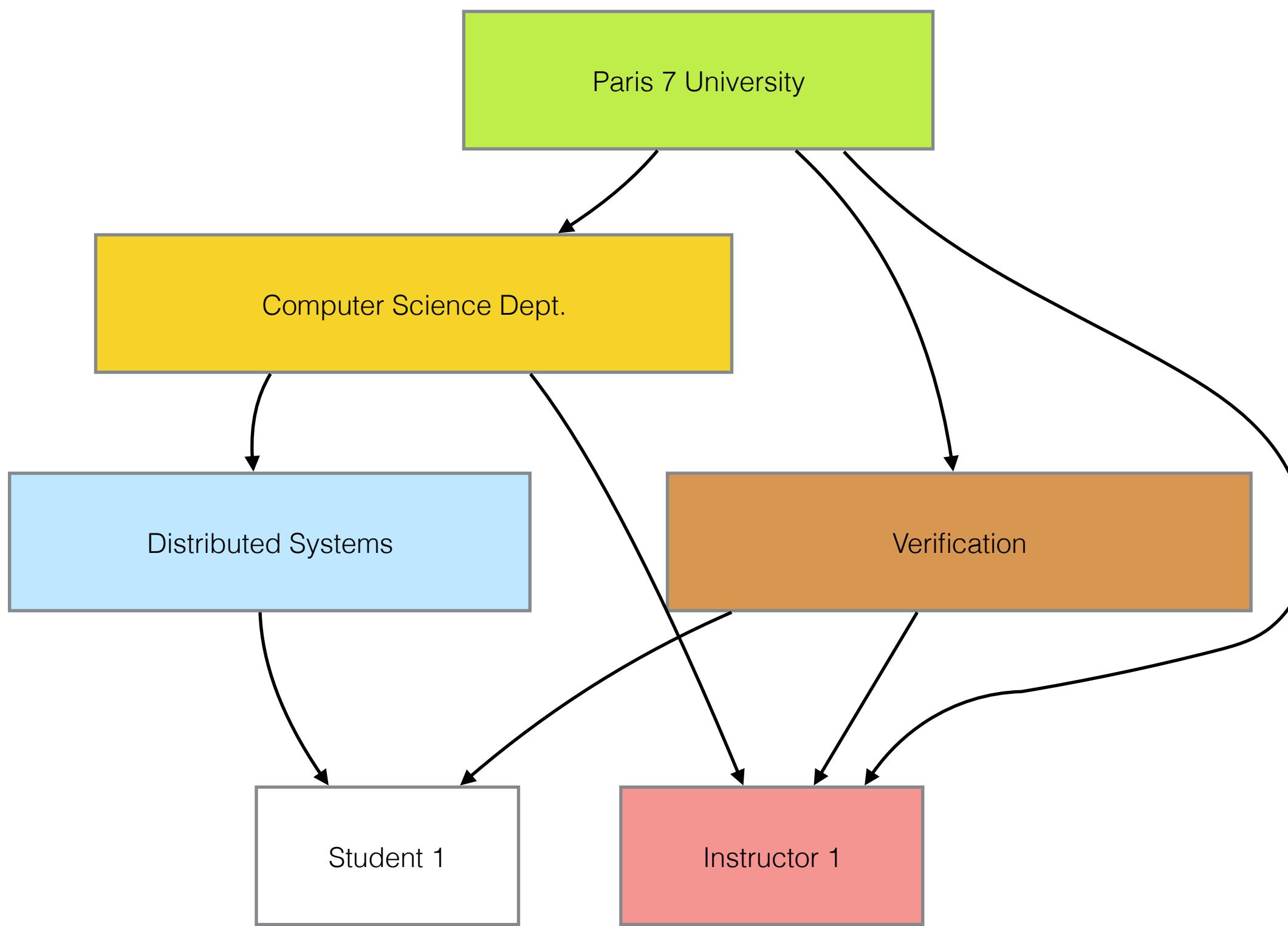
Two actors



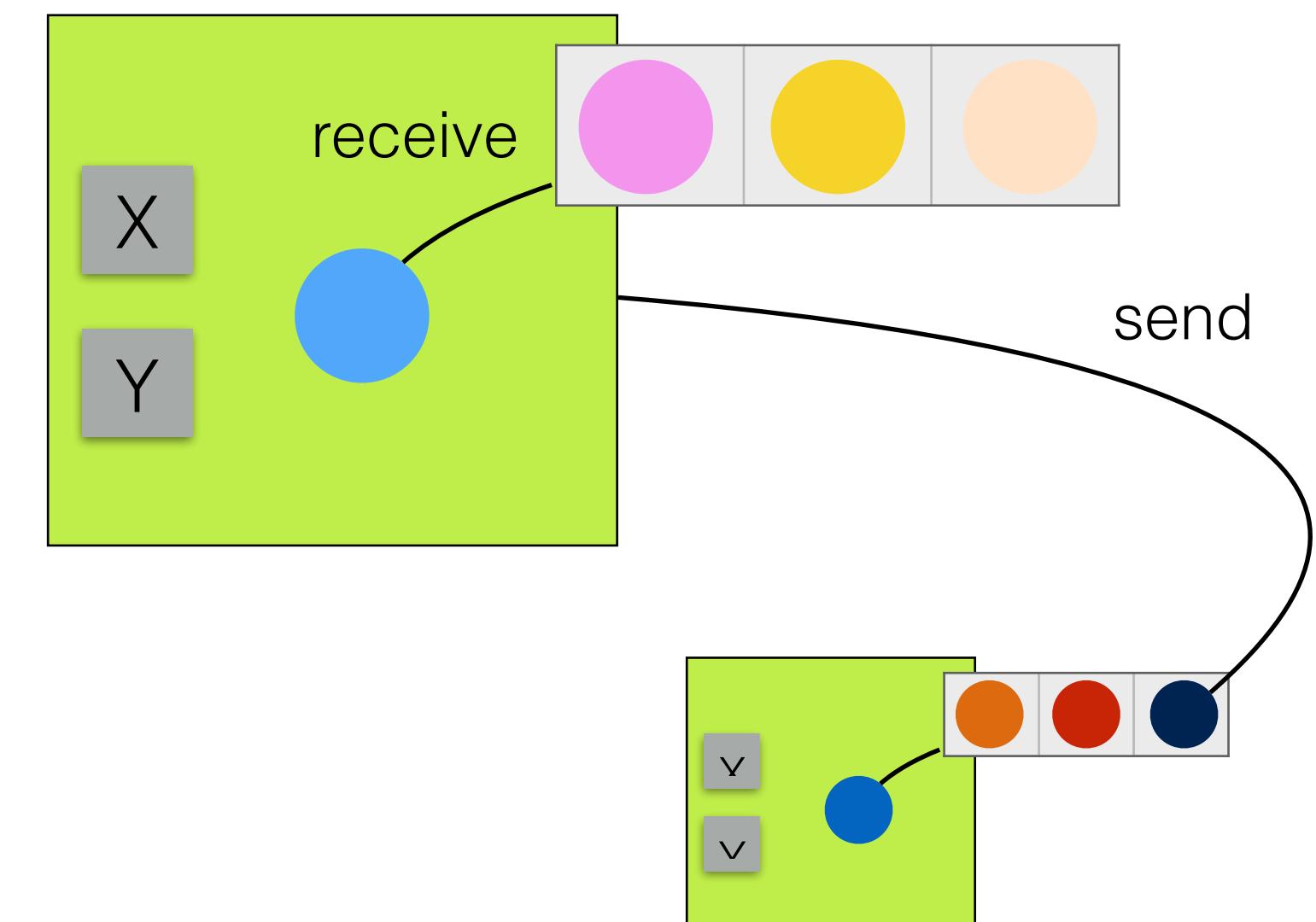
# WHAT IS AN ACTOR?

- ▶ State
- ▶ Messages
- ▶ Inbox
- ▶ Supervision

An actor system runtime



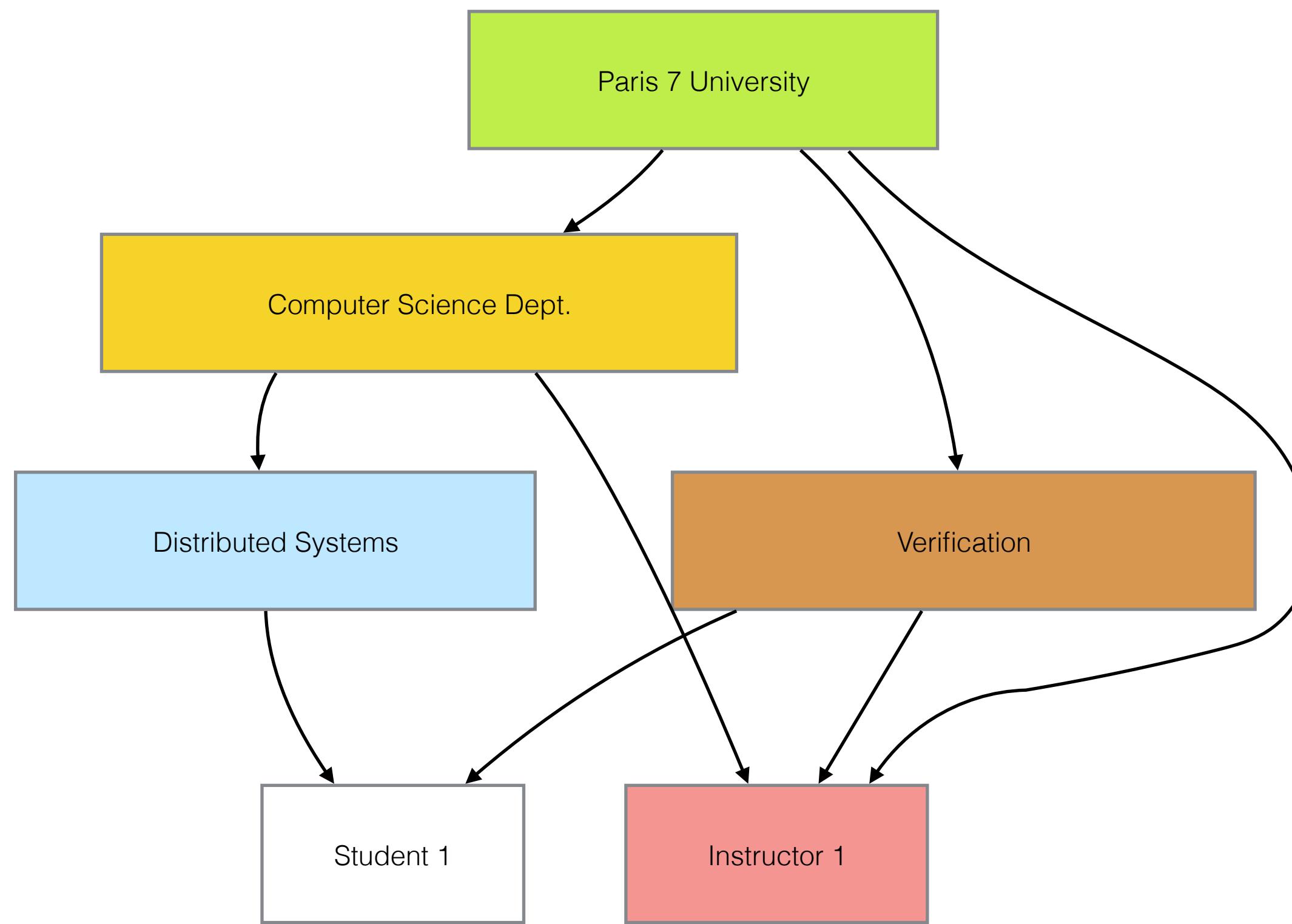
Two actors



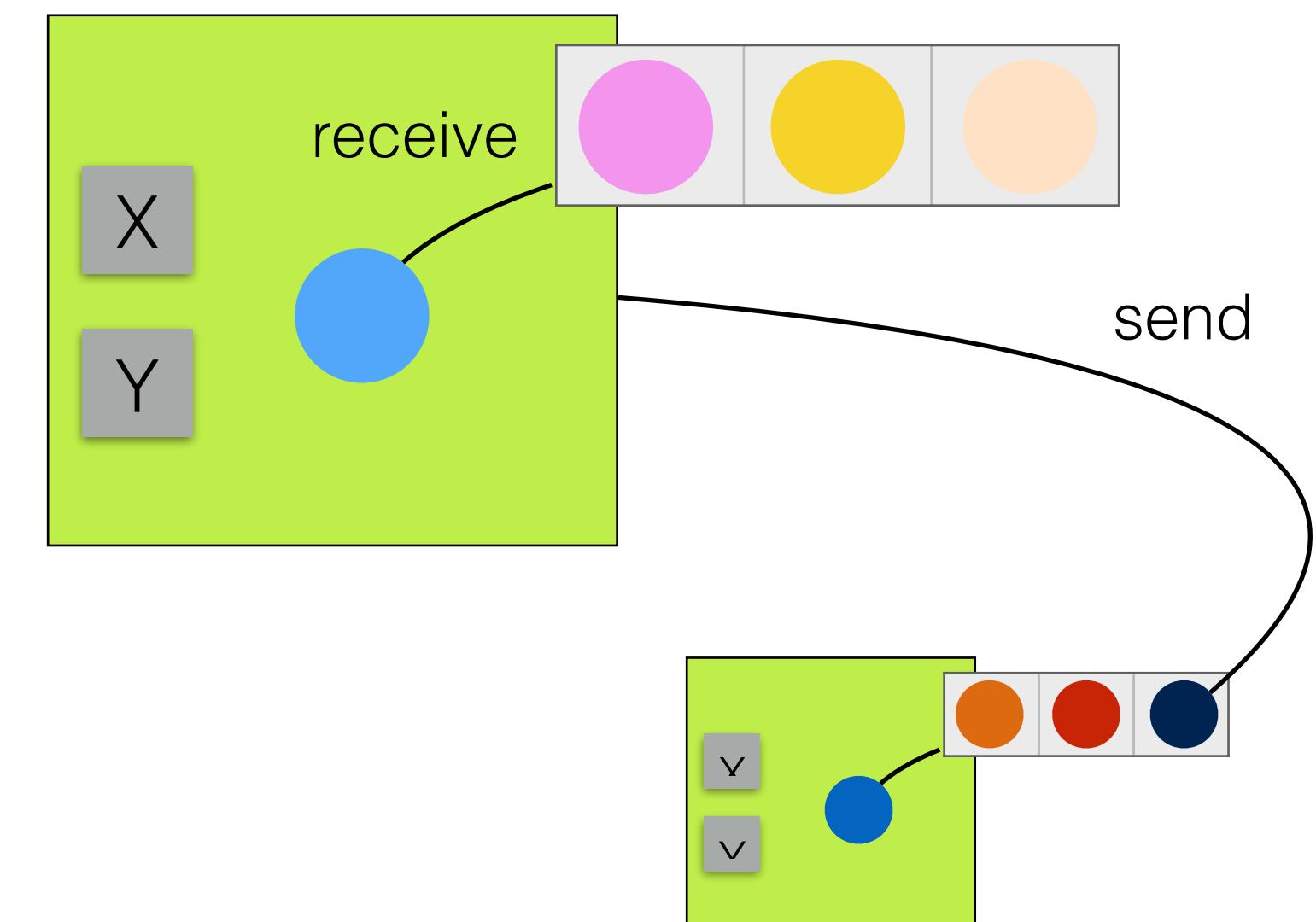
# WHAT IS AN ACTOR?

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

An actor system runtime

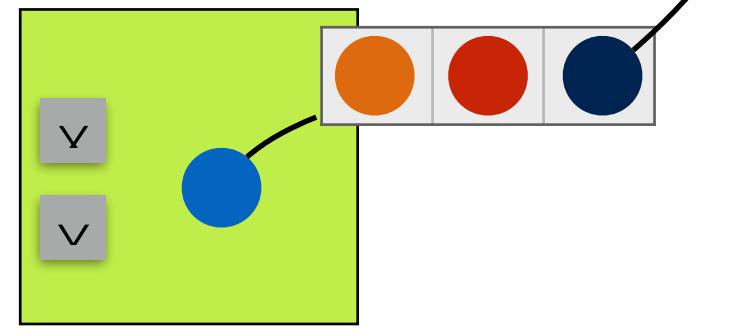
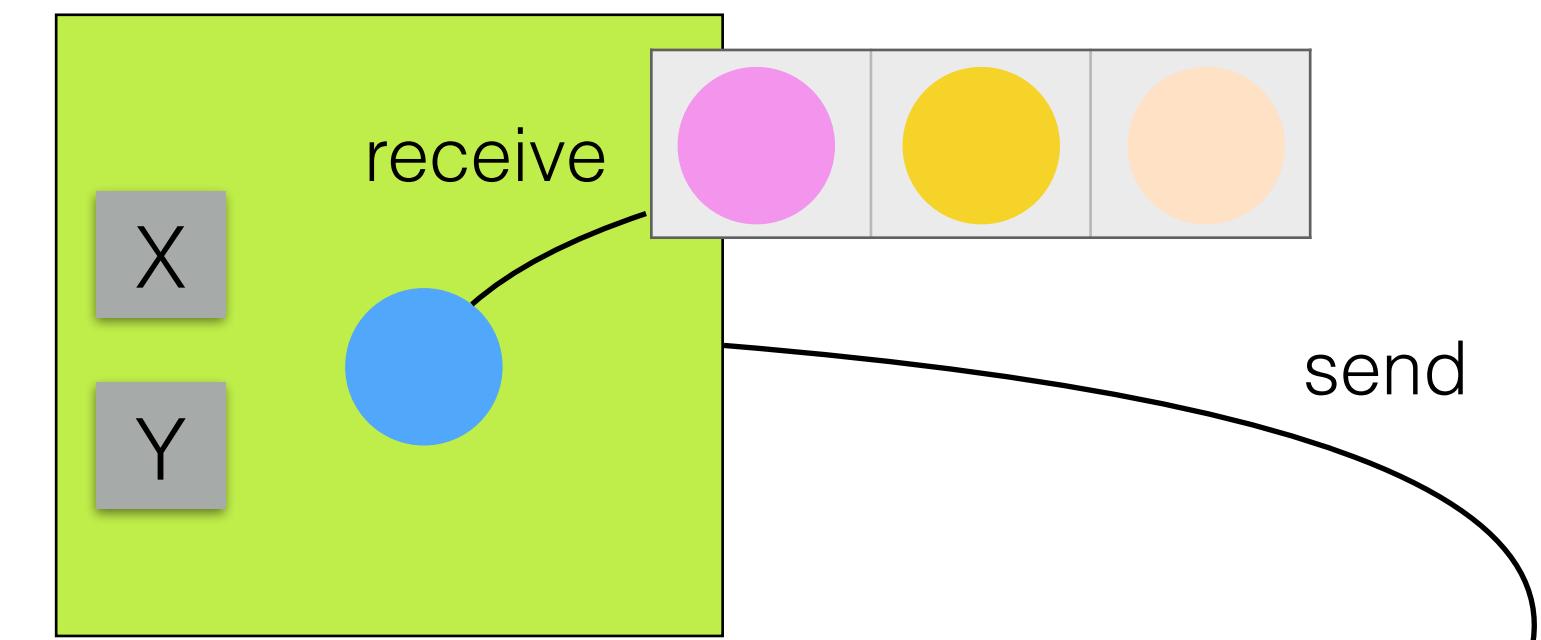


Two actors



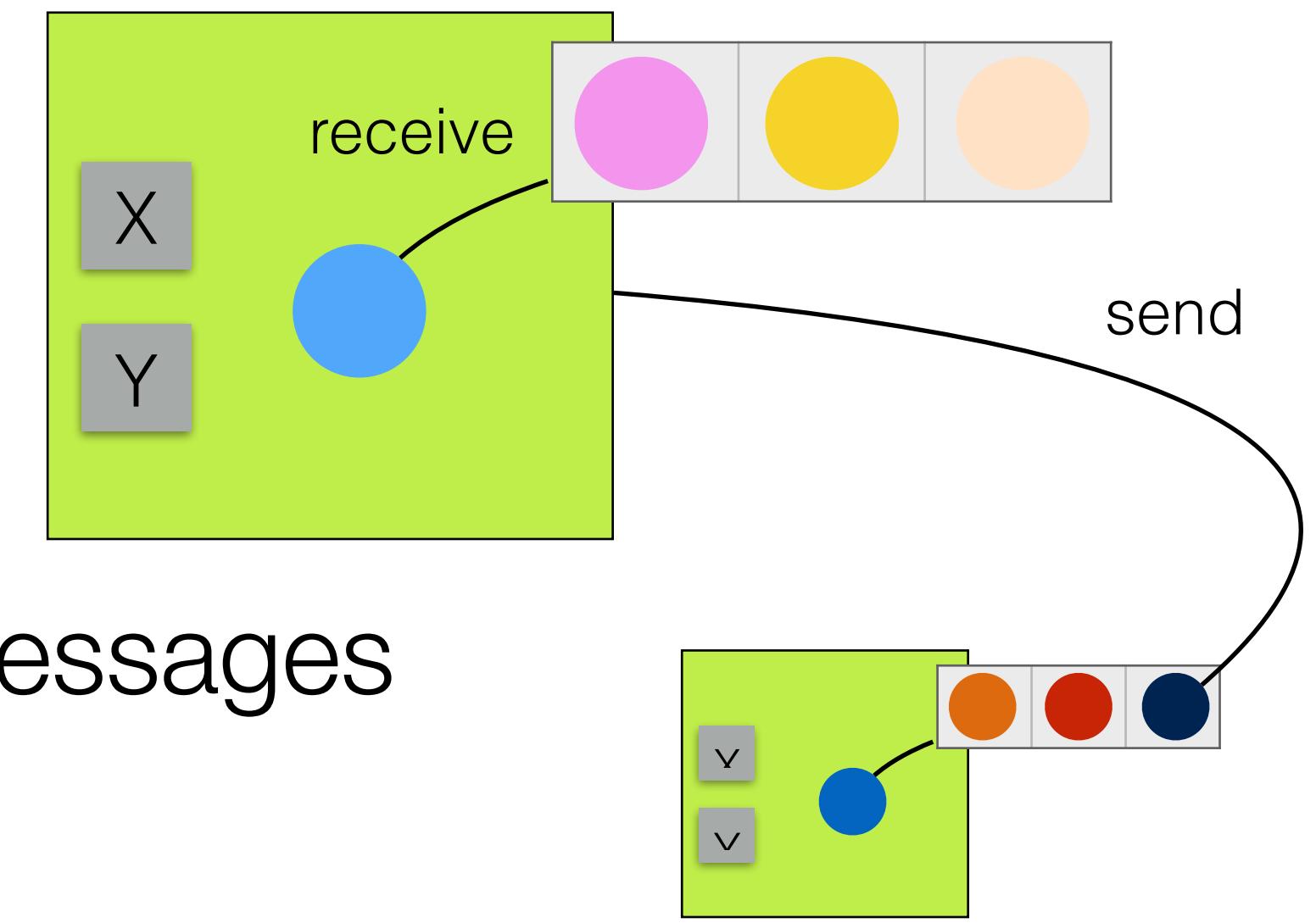
# STATE AND COMPUTATION

- ▶ 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



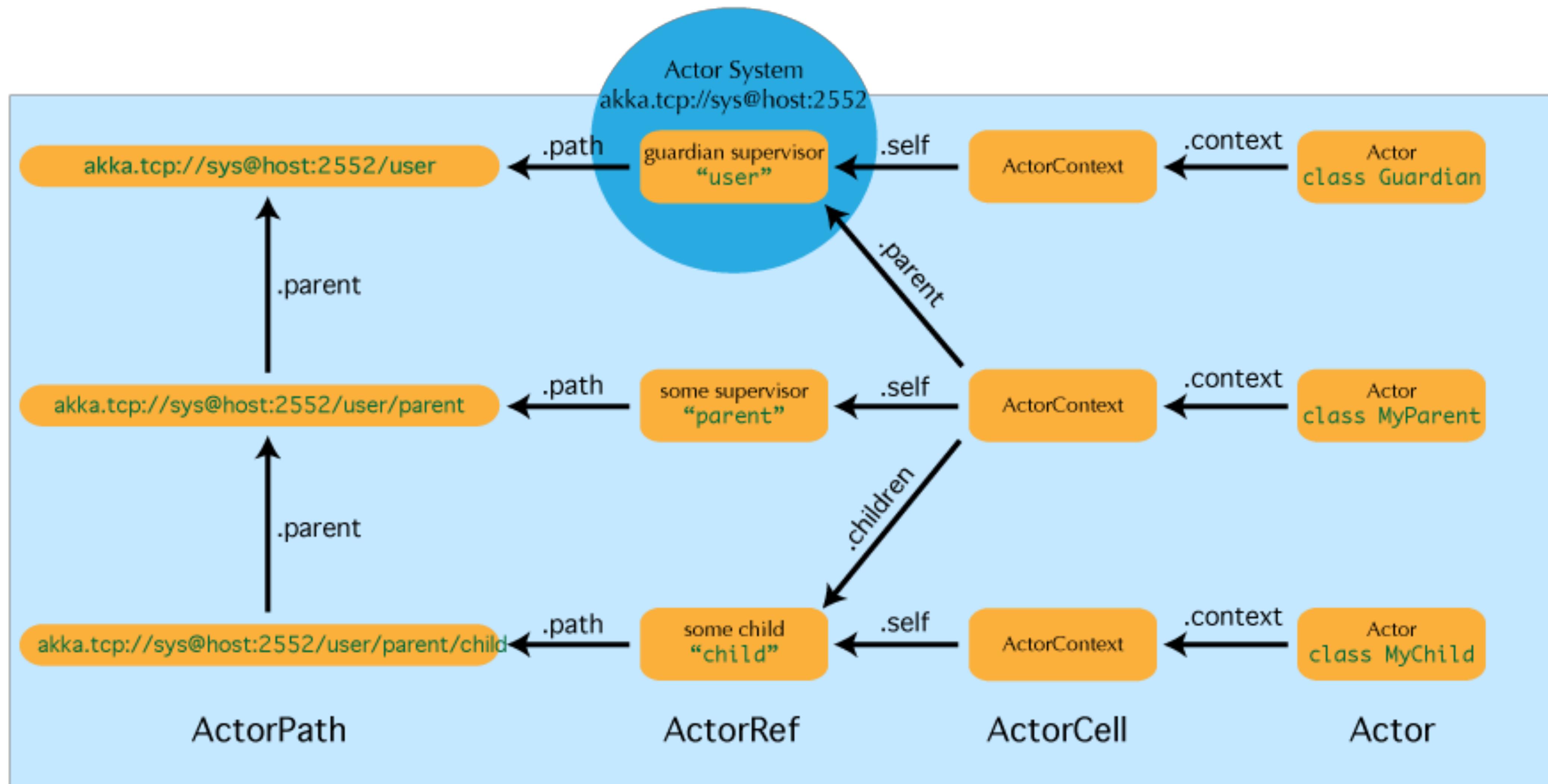
# STATE AND COMPUTATION

- ▶ 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



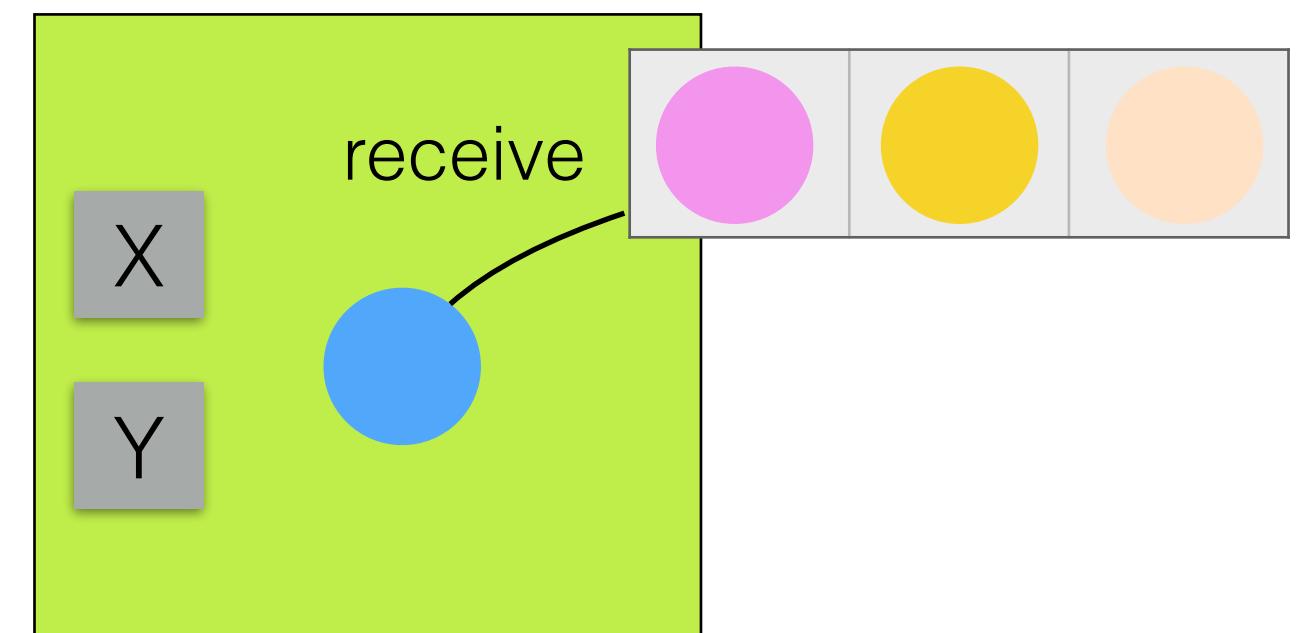
◉ **Exercise 2:** Create actors for a University Department, for Courses and for Students

# PATH OF AN ACTOR



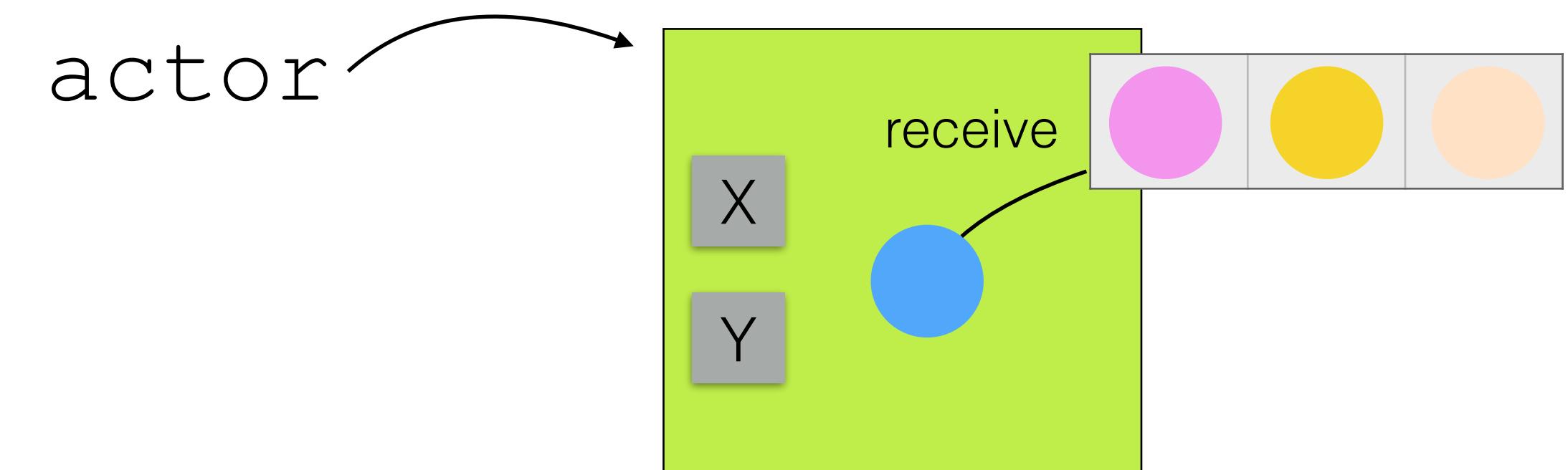
# SENDING A MESSAGE TO AN ACTOR

- ▶ How can we refer to an actor?



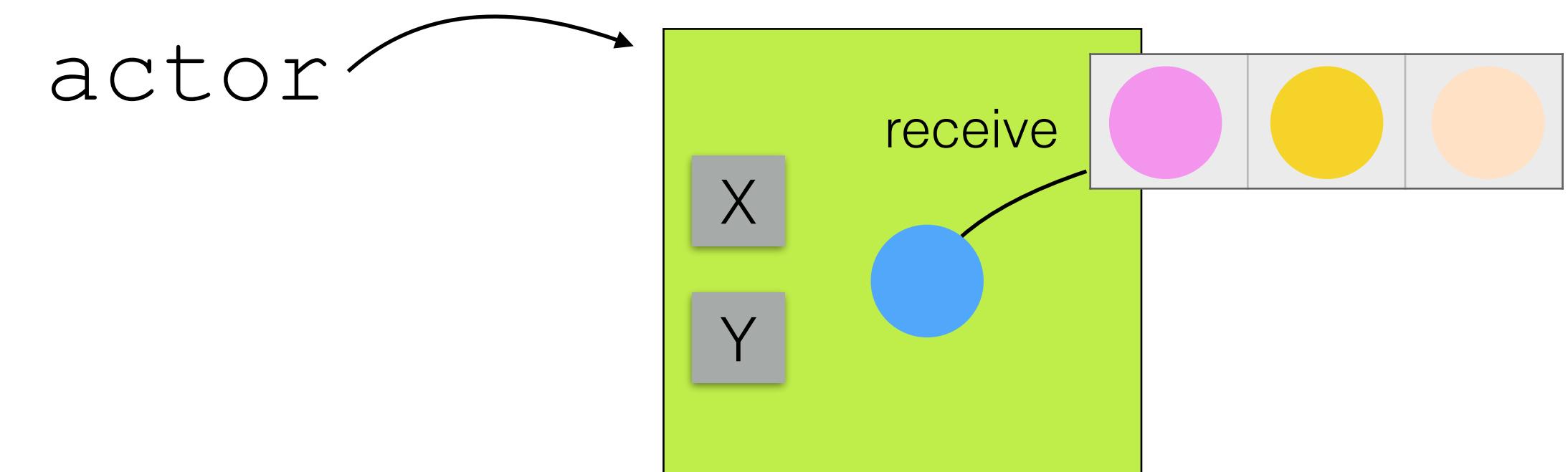
# SENDING A MESSAGE TO AN ACTOR

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



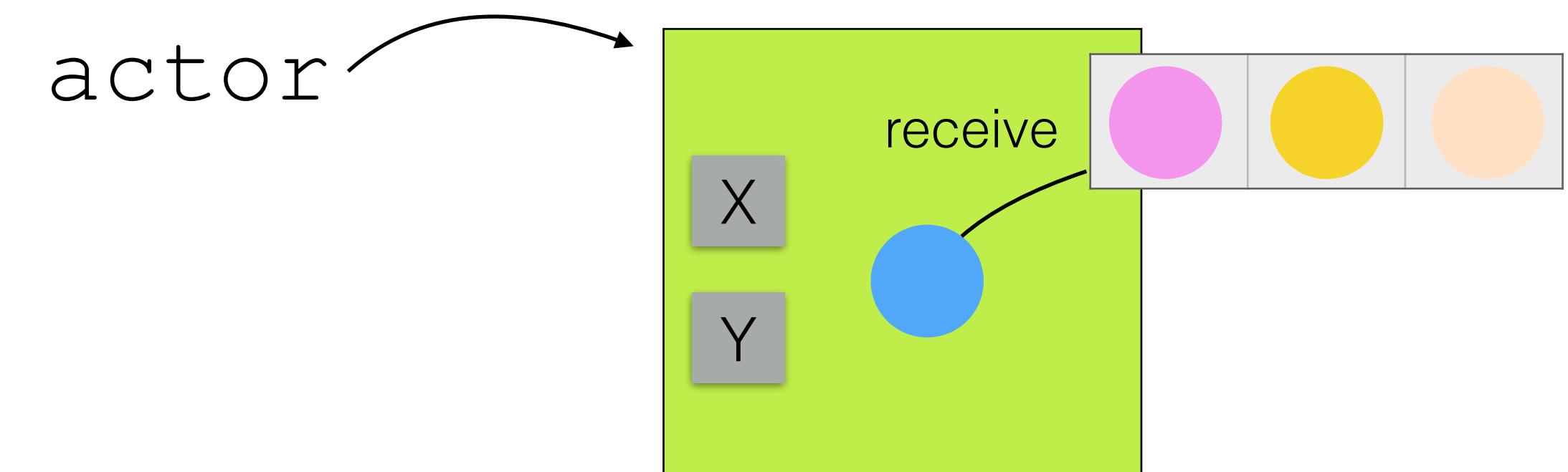
# SENDING A MESSAGE TO AN ACTOR

- ▶ 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`



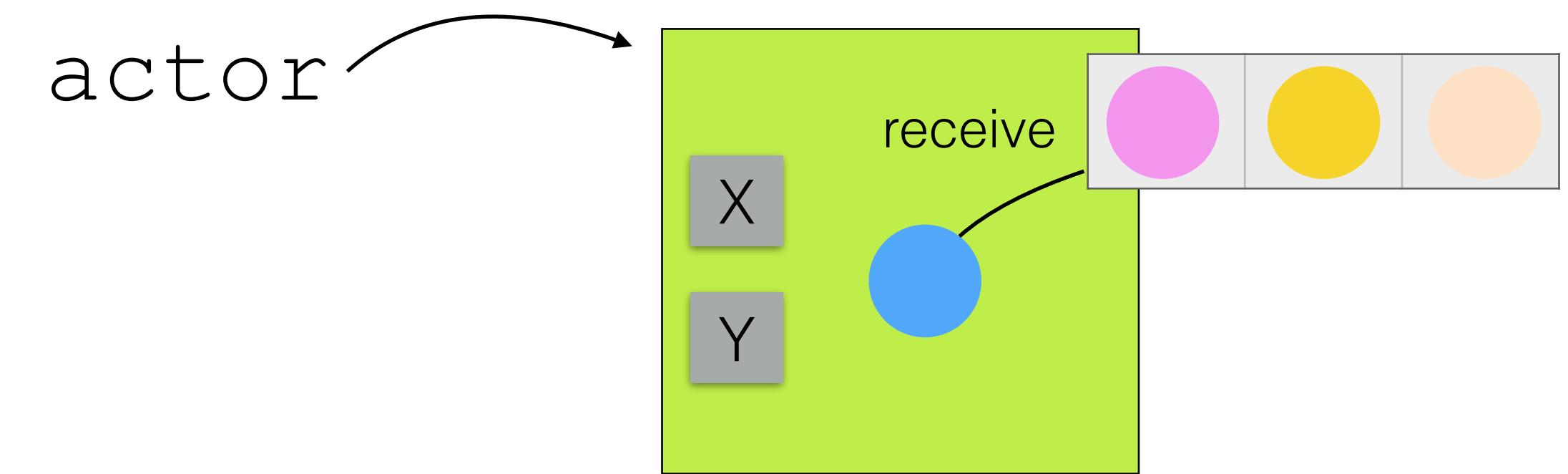
# SENDING A MESSAGE TO AN ACTOR

- ▶ 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.
- ▶ Infrastructure failures
- ▶ Bugs happen
- ▶ Graceful Degradation
  - ▶ Slow might be acceptable
  - ▶ Retries might be acceptable
  - ▶ **Unreachable is not!**
- ▶ 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: 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



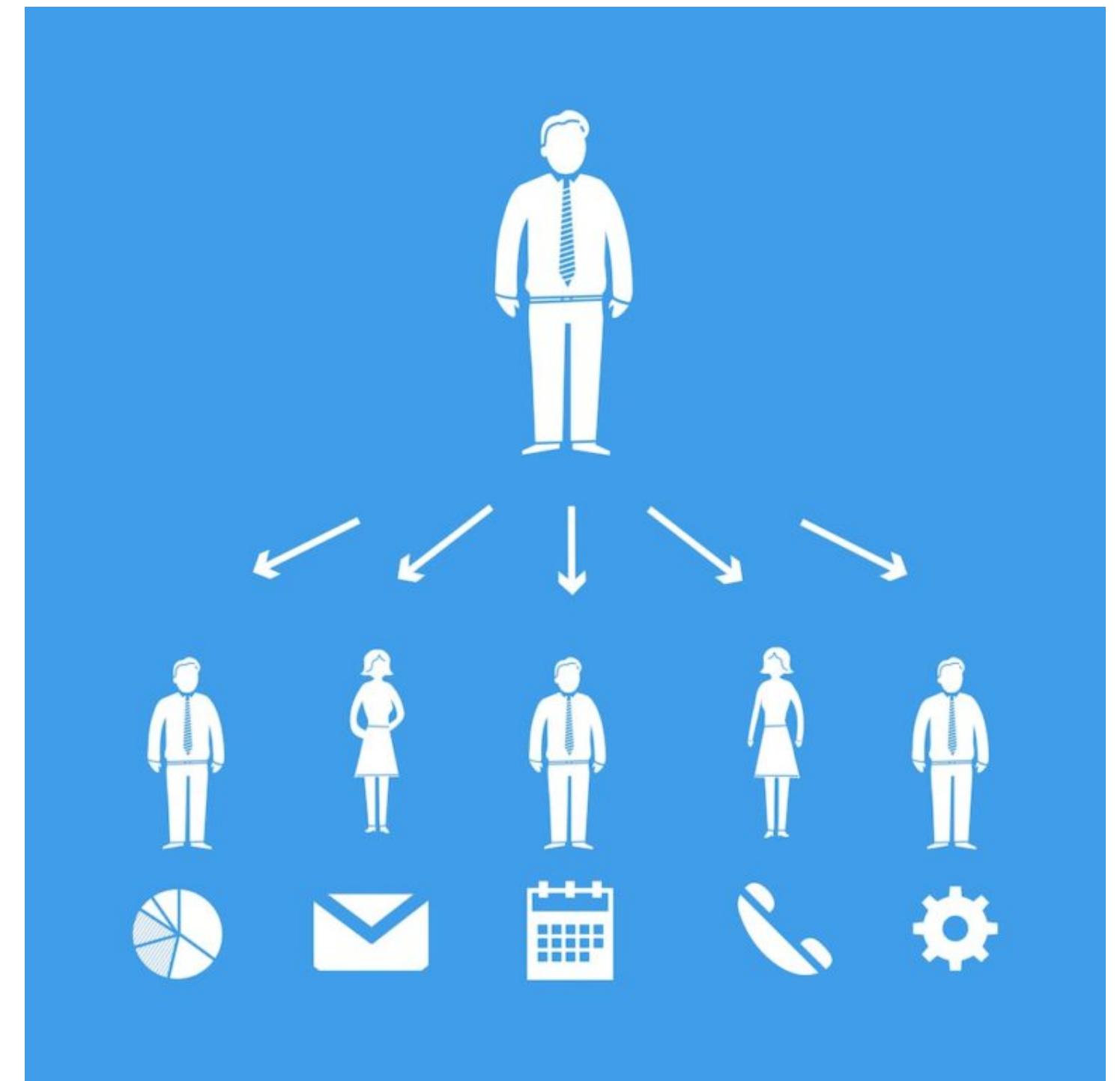
# FAULT TOLERANCE: MECHANISMS

- ▶ 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)



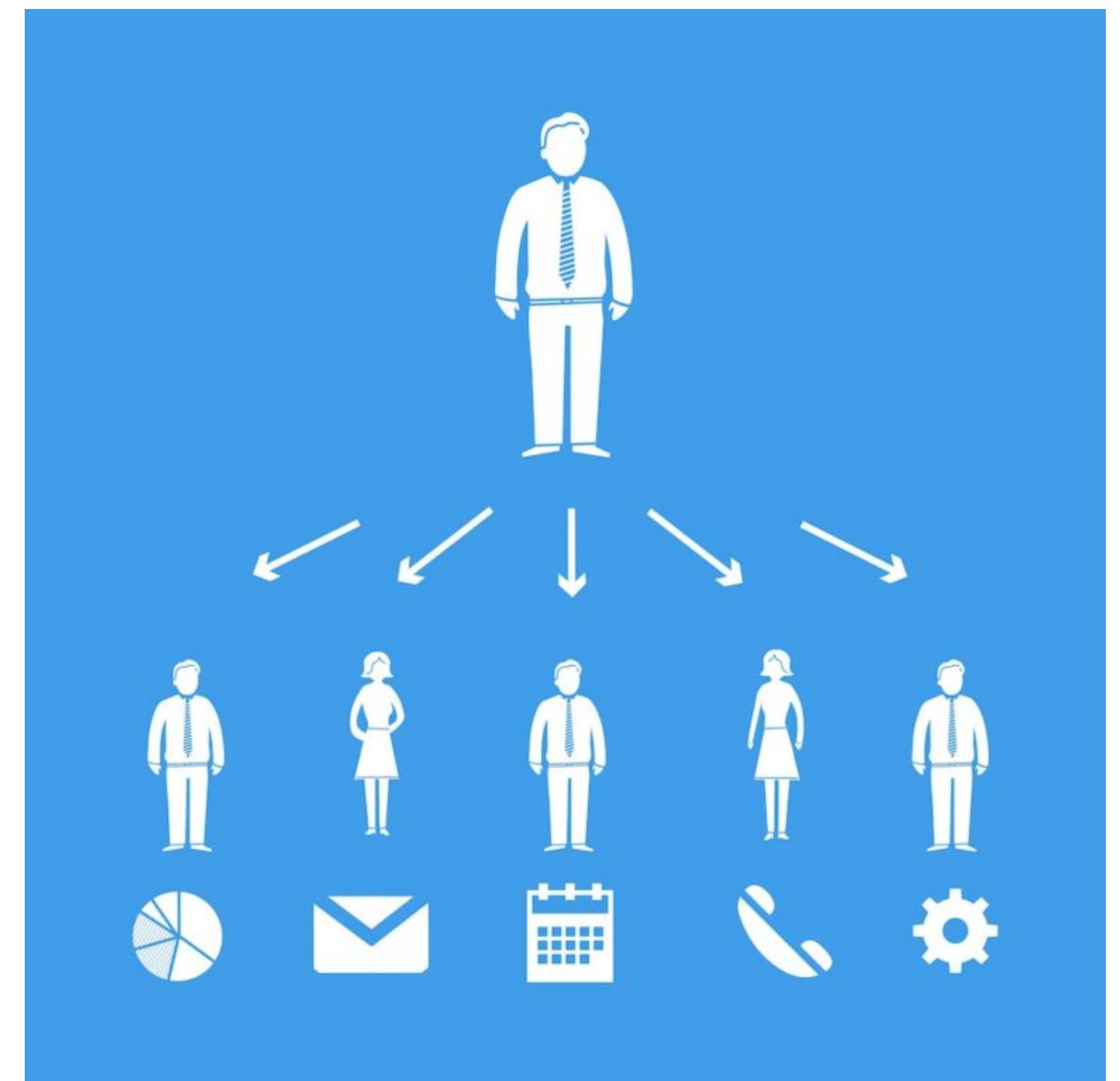
# FAULT TOLERANCE: MECHANISMS

- ▶ Delegation
- ▶ Hand-over responsibility



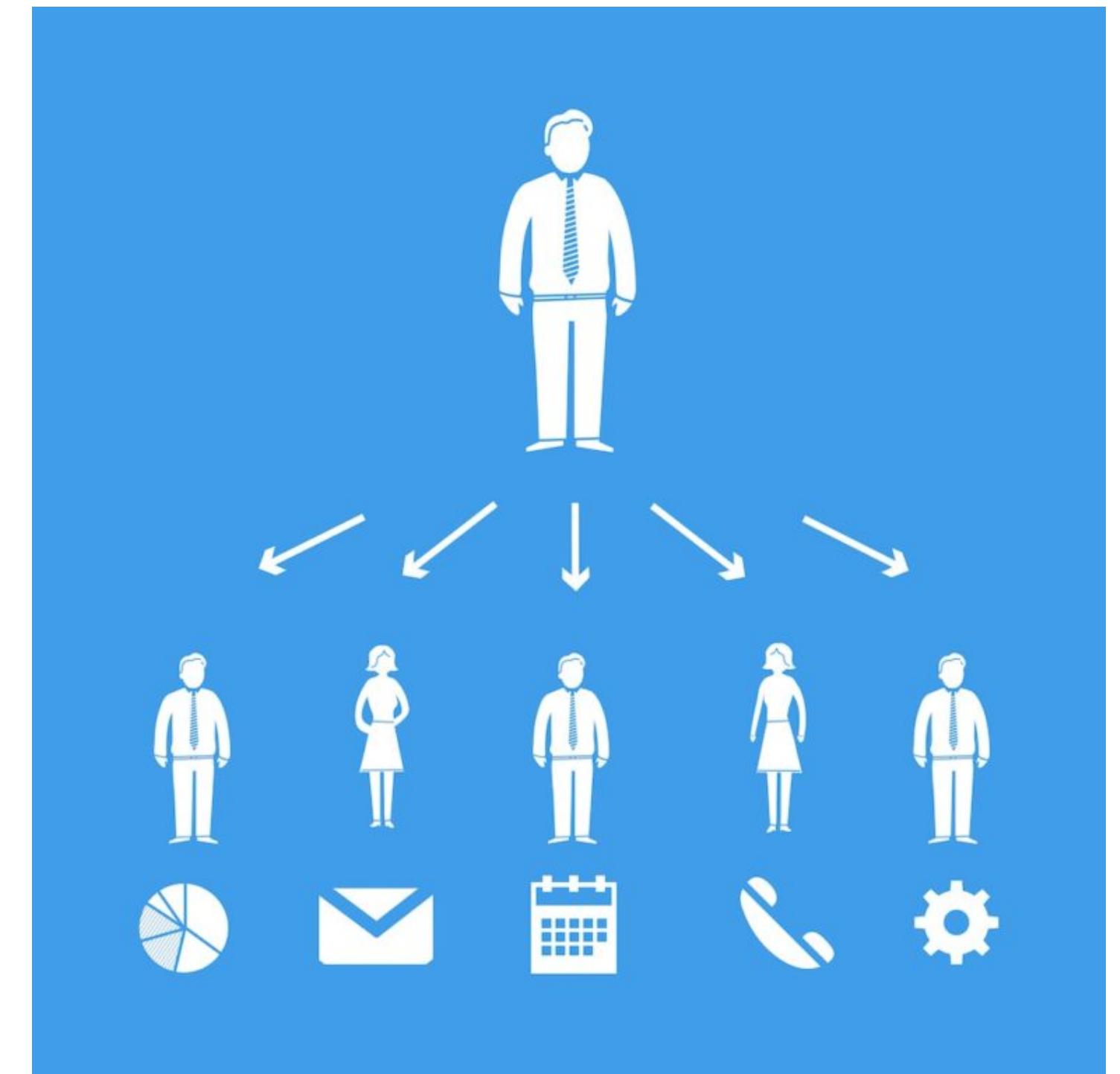
# FAULT TOLERANCE: MECHANISMS

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



# FAULT TOLERANCE: MECHANISMS

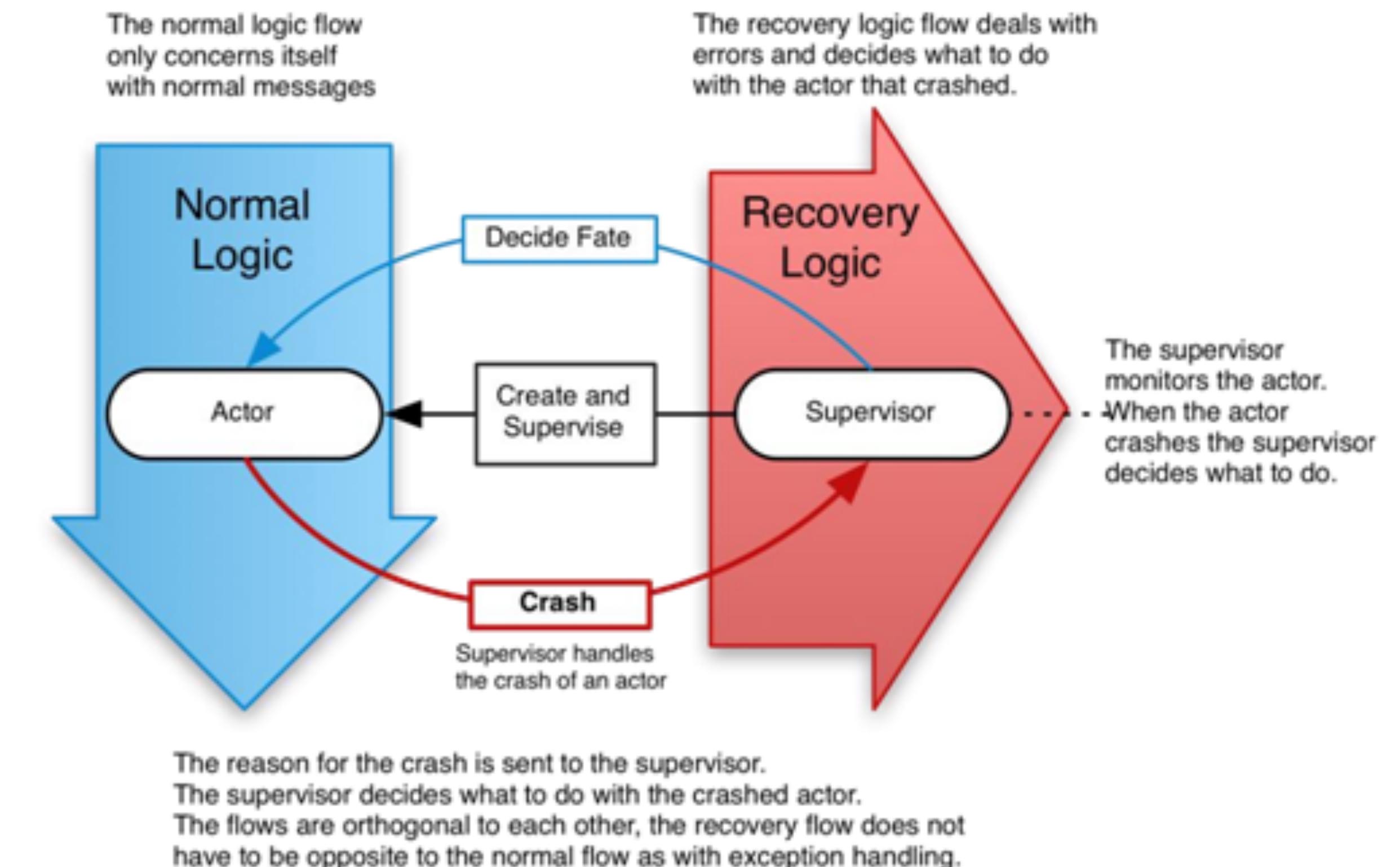
- ▶ Delegation
  - ▶ Hand-over responsibility
- ▶ Components
  - ▶ Self-contained
  - ▶ Isolated
  - ▶ Encapsulated
- ▶ Micro-services / Actors



# FAULT TOLERANCE: SUPERVISION

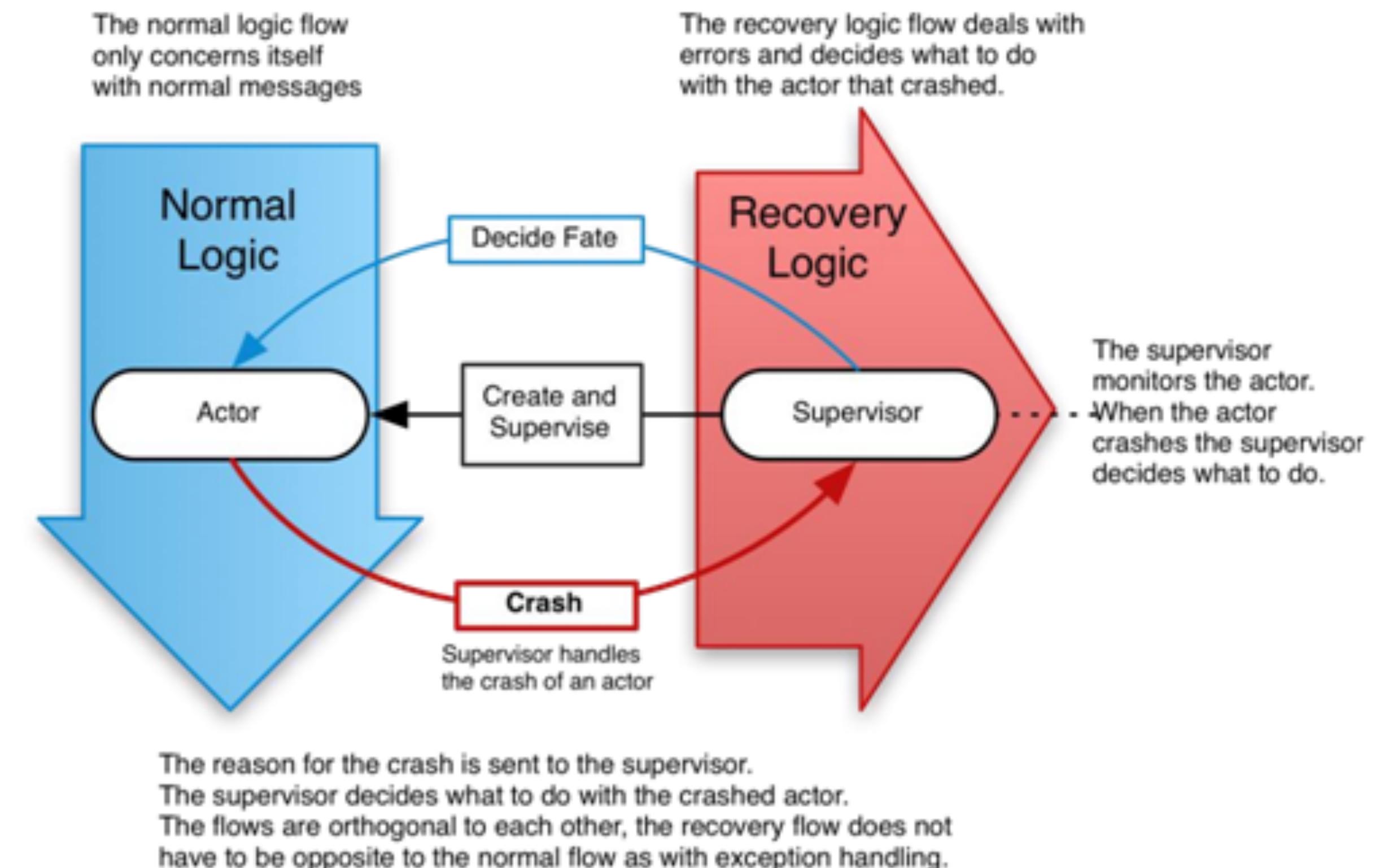
# FAULT TOLERANCE: SUPERVISION

- ▶ “Let it crash”



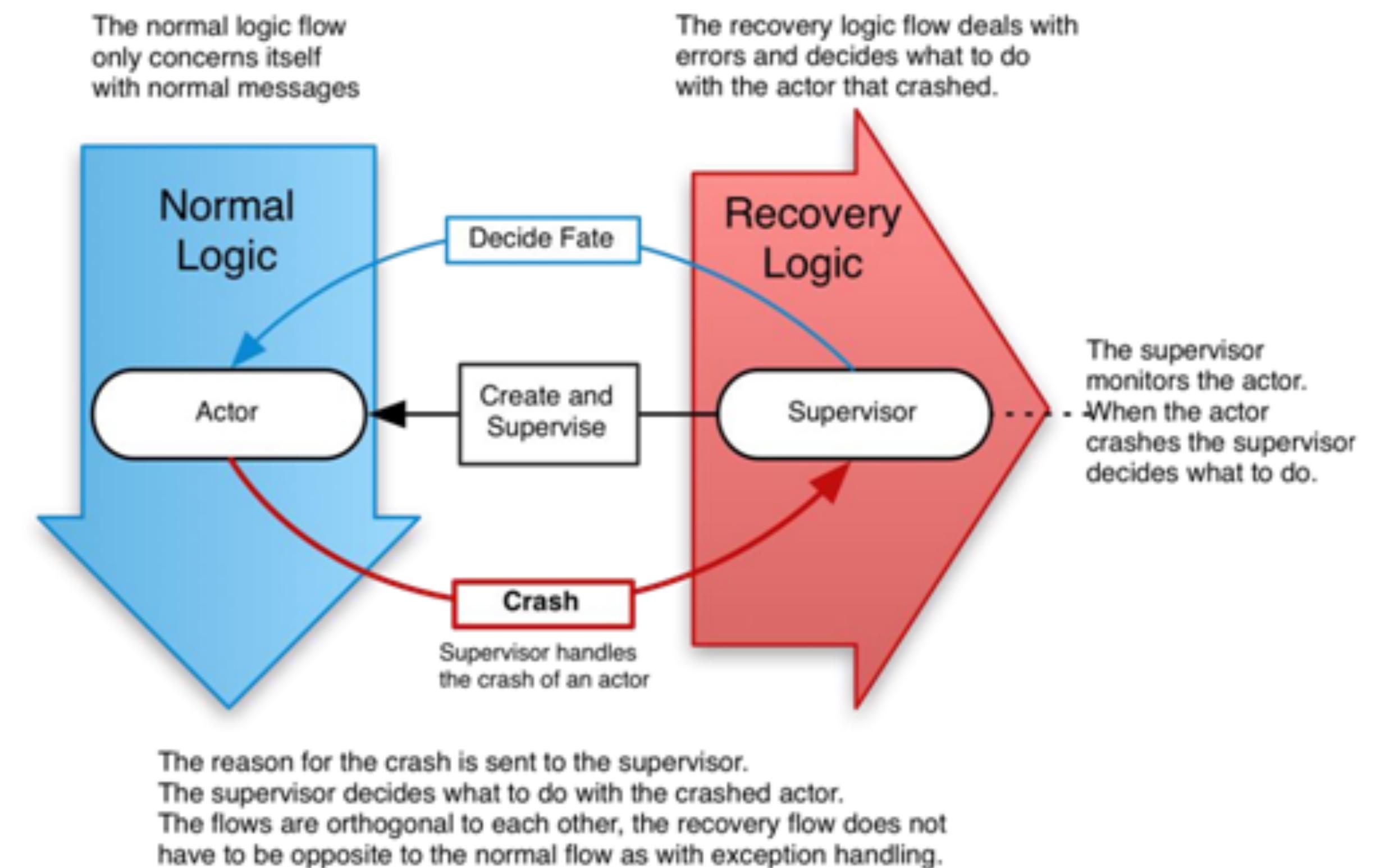
# FAULT TOLERANCE: SUPERVISION

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



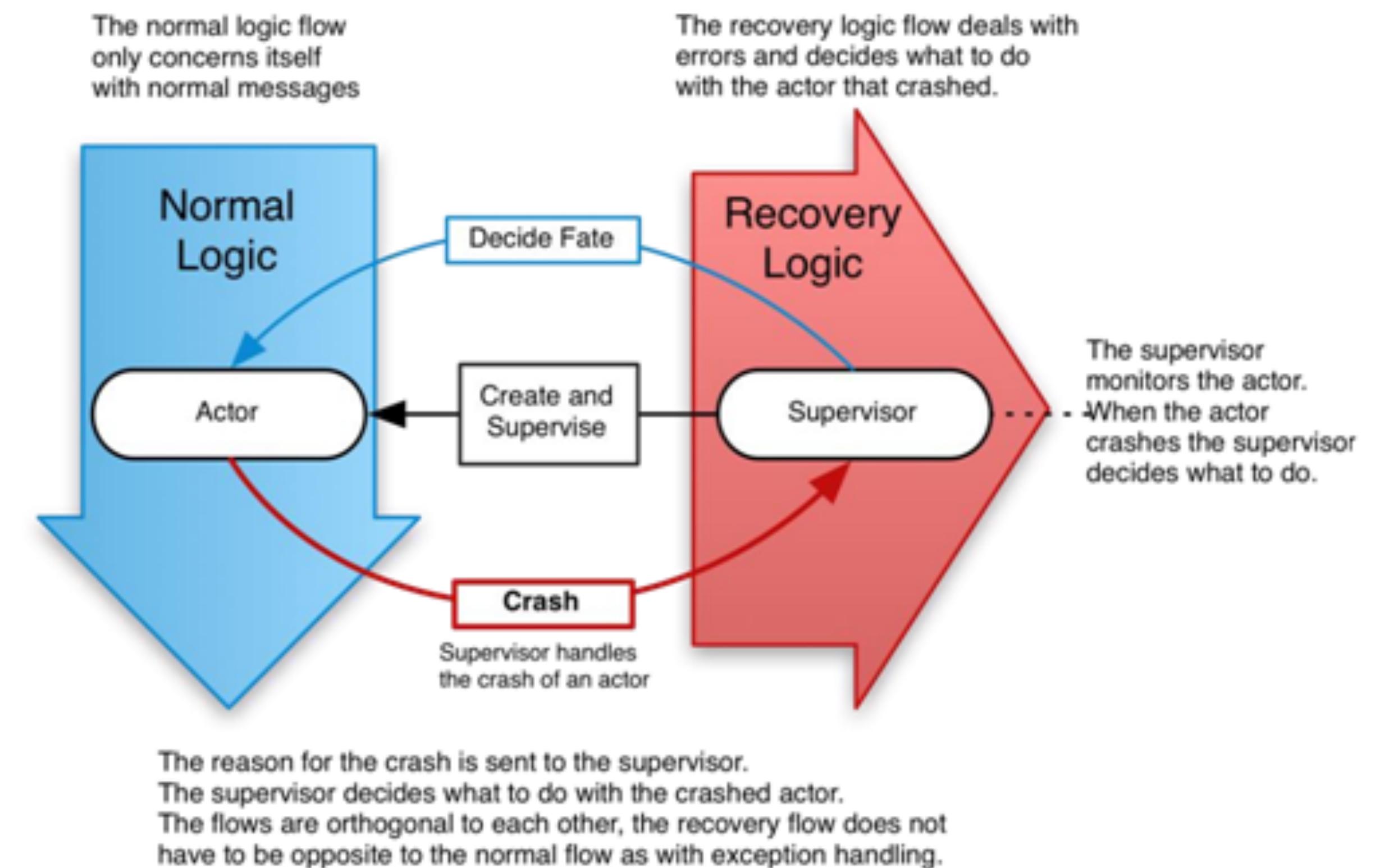
# FAULT TOLERANCE: SUPERVISION

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



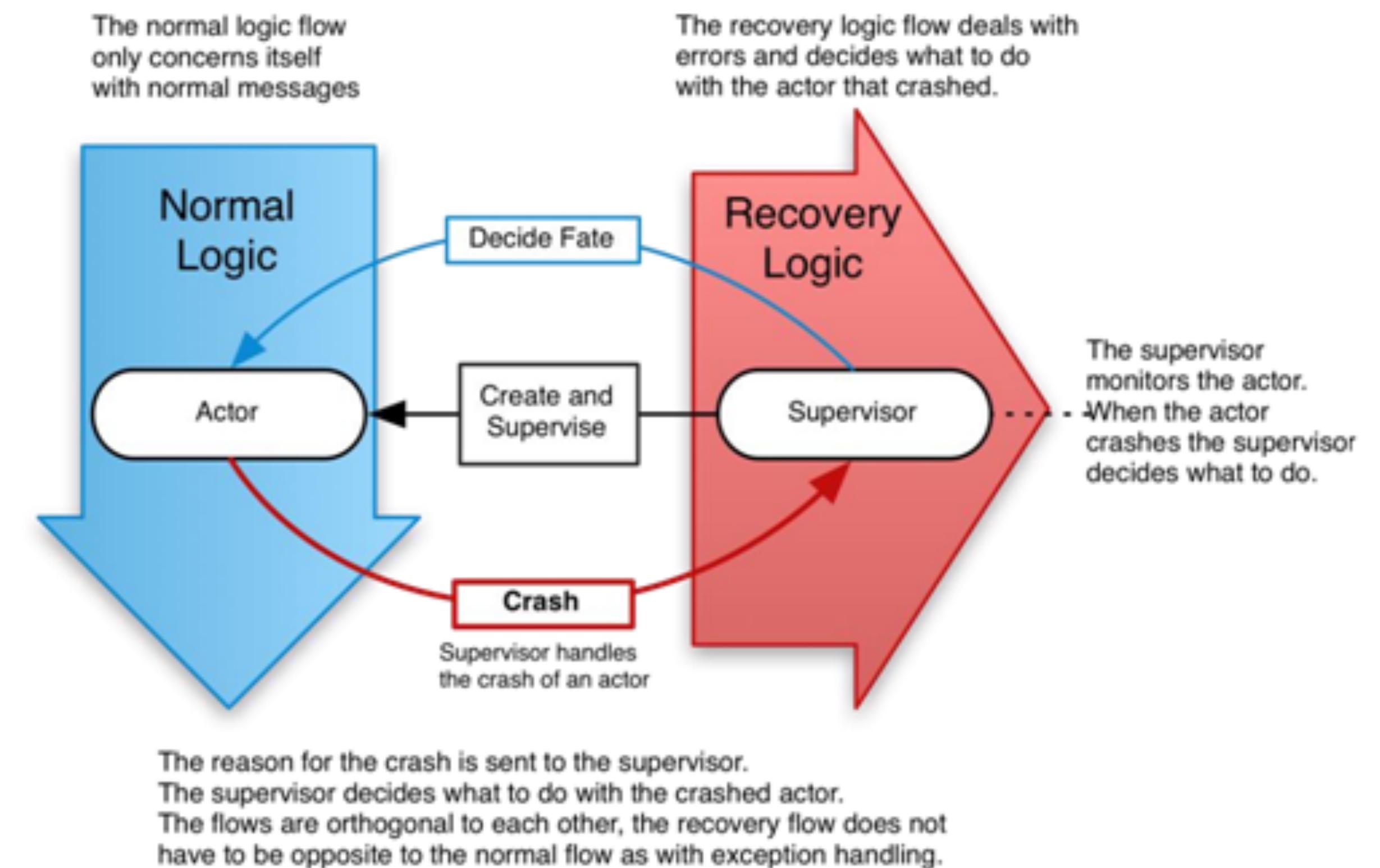
# FAULT TOLERANCE: SUPERVISION

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



# FAULT TOLERANCE: SUPERVISION

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



# PERSISTENCE (SMALL EXAMPLE)

# PERSISTENCE (SMALL EXAMPLE)

- ▶ Allow actors to be stored for later usage

# PERSISTENCE (SMALL EXAMPLE)

- ▶ Allow actors to be stored for later usage
- ▶ Snapshotting and error recovery

# PERSISTENCE (SMALL EXAMPLE)

- ▶ Allow actors to be stored for later usage
- ▶ Snapshotting and error recovery
- ▶ Liberate resources (cpu/memory vs. storage)

# PERSISTENCE (SMALL EXAMPLE)

- ▶ Allow actors to be stored for later usage
- ▶ Snapshotting and error recovery
- ▶ Liberate resources (cpu/memory vs. storage)
- ▶ Restarting an application

# PERSISTENCE (SMALL EXAMPLE)

- ▶ Allow actors to be stored for later usage
- ▶ Snapshotting and error recovery
- ▶ Liberate resources (cpu/memory vs. storage)
- ▶ Restarting an application
- ▶ We are not going to cover Persistence here
  - ▶ Akka has good support for it
  - ▶ Great project!

# FUTURES AND AGENTS

- ▶ Scala/Java concurrency, not necessarily Actor programming

# AKKA ACTORS (OUTLINE)

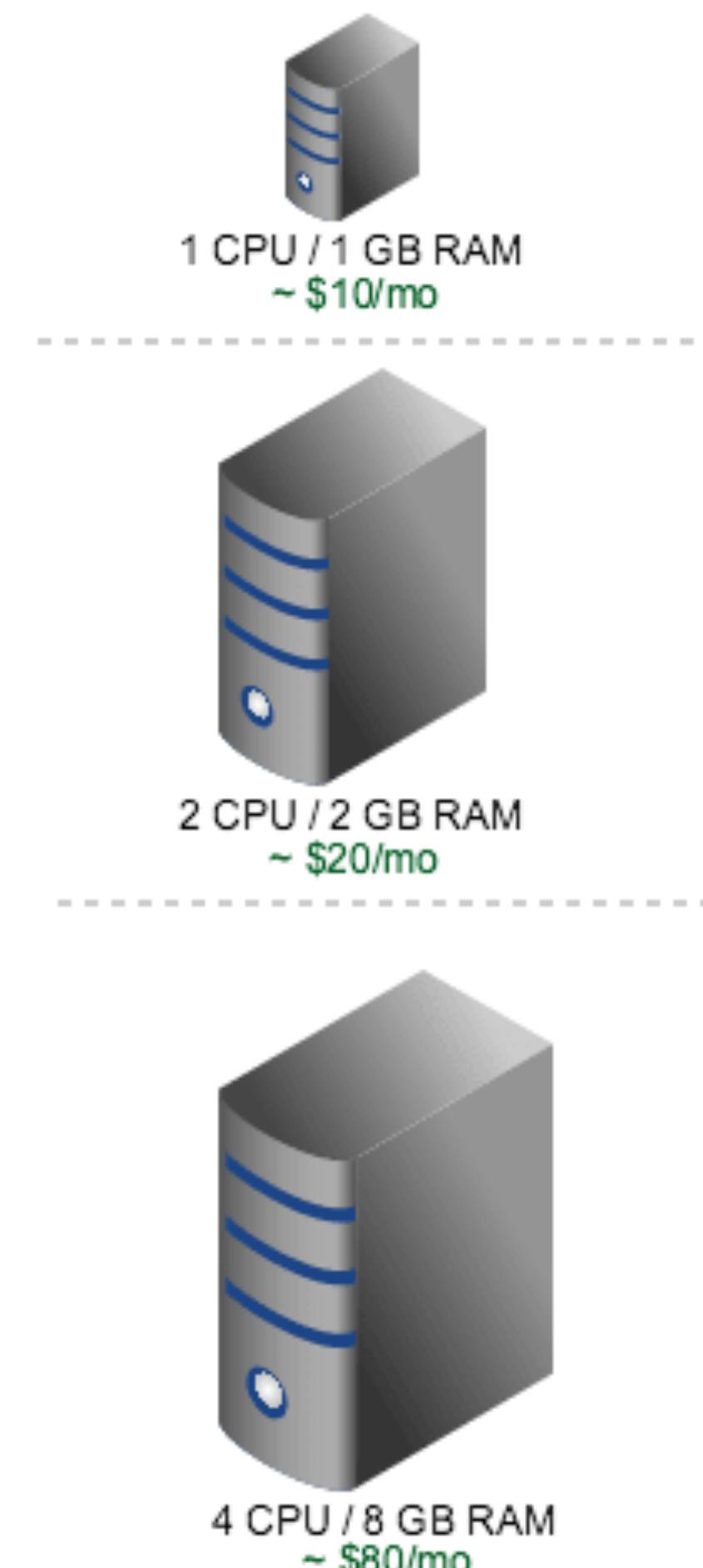
- ▶ Monday
  - ▶ Basics of Akka Programming
  - ▶ Fault Tolerance
  - ▶ Some Scala goodies
- ▶ Tuesday
  - ▶ Distribution
  - ▶ Scalability / Elasticity
  - ▶ Some Actor Patterns
- ▶ Wednesday
  - ▶ Cloud Computing
  - ▶ Virtualization
  - ▶ Akka on a cluster
- ▶ Thursday
  - ▶ Synchronization and State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

# DISTRIBUTED ACTORS

# SCALABILITY

- ▶ Vertical Scalability
  - ▶ Buy a new larger server
  - ▶ Improve the usage of resources and algorithms
  - ▶ Resources always bounded

**Vertical Scaling**



# SCALABILITY

- ▶ Vertical Scalability

- ▶ Buy a new larger server
- ▶ Improve the usage of resources and algorithms
- ▶ Resources always bounded

- ▶ Horizontal Scalability

- ▶ Add more resources
- ▶ Potentially unbounded resources (Cloud)

Vertical Scaling



1 CPU / 1 GB RAM  
~ \$10/mo

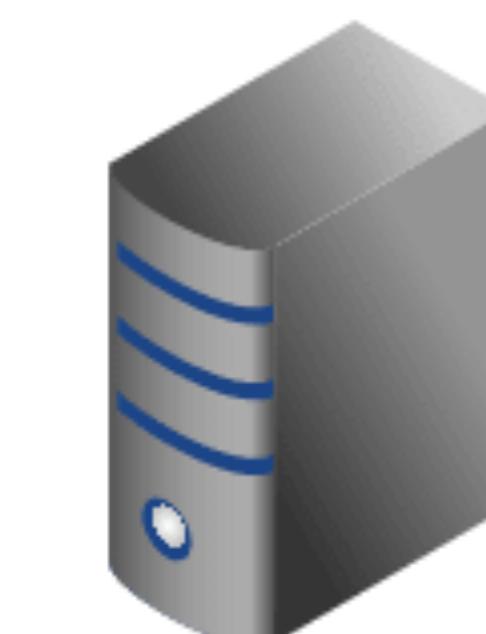
Horizontal Scaling



1 CPU / 1 GB RAM  
~ \$10/mo



2 x (1 CPU / 1 GB RAM)  
~ \$20/mo



4 CPU / 8 GB RAM  
~ \$80/mo

# HORIZONTAL SCALING

# HORIZONTAL SCALING

- ▶ Requires designing Services to be *Distributed*

# HORIZONTAL SCALING

- ▶ Requires designing Services to be *Distributed*
- ▶ Clustering to coordinate the services

# HORIZONTAL SCALING

- ▶ Requires designing Services to be *Distributed*
- ▶ Clustering to coordinate the services
- ▶ Sharding to distributed work among resources

# HORIZONTAL SCALING

- ▶ Requires designing Services to be *Distributed*
- ▶ Clustering to coordinate the services
- ▶ Sharding to distributed work among resources
- ▶ The application has to be architected from the ground-up to be scalable

# HORIZONTAL SCALING

- ▶ Requires designing Services to be *Distributed*
- ▶ Clustering to coordinate the services
- ▶ Sharding to distributed work among resources
- ▶ The application has to be architected from the ground-up to be scalable
- ▶ Akka has great support for Clustering/Sharding

# REMOTING (ACTUAL DISTRIBUTION)

- ▶ Create an actor in another “machine”
- ▶ Find an actor in a different machine
- ▶ Programmatic actor deployment
- ▶ Scaling with remote actors
- ▶ Chat example

# CLUSTERING

- ▶ Publish/Subscribe model
- ▶ Sharding
- ▶ Sharding between clusters!!!

# ACTOR PROGRAMMING PATTERNS

# AKKA PROGRAMMING PATTERNS

- ▶ Master / Slave
- ▶ Ordered Termination
- ▶ Shutdown patterns
- ▶ Throttling messages (back-pressure)
- ▶ Load balancing

# AKKA PROGRAMMING PATTERNS

- ▶ Aggregator
- ▶ CountDown latch
- ▶ Finite-State-Machine (FSM)
- ▶ Enveloping an Actor

# ACTORS FOR DISTRIBUTED PROGRAMMING

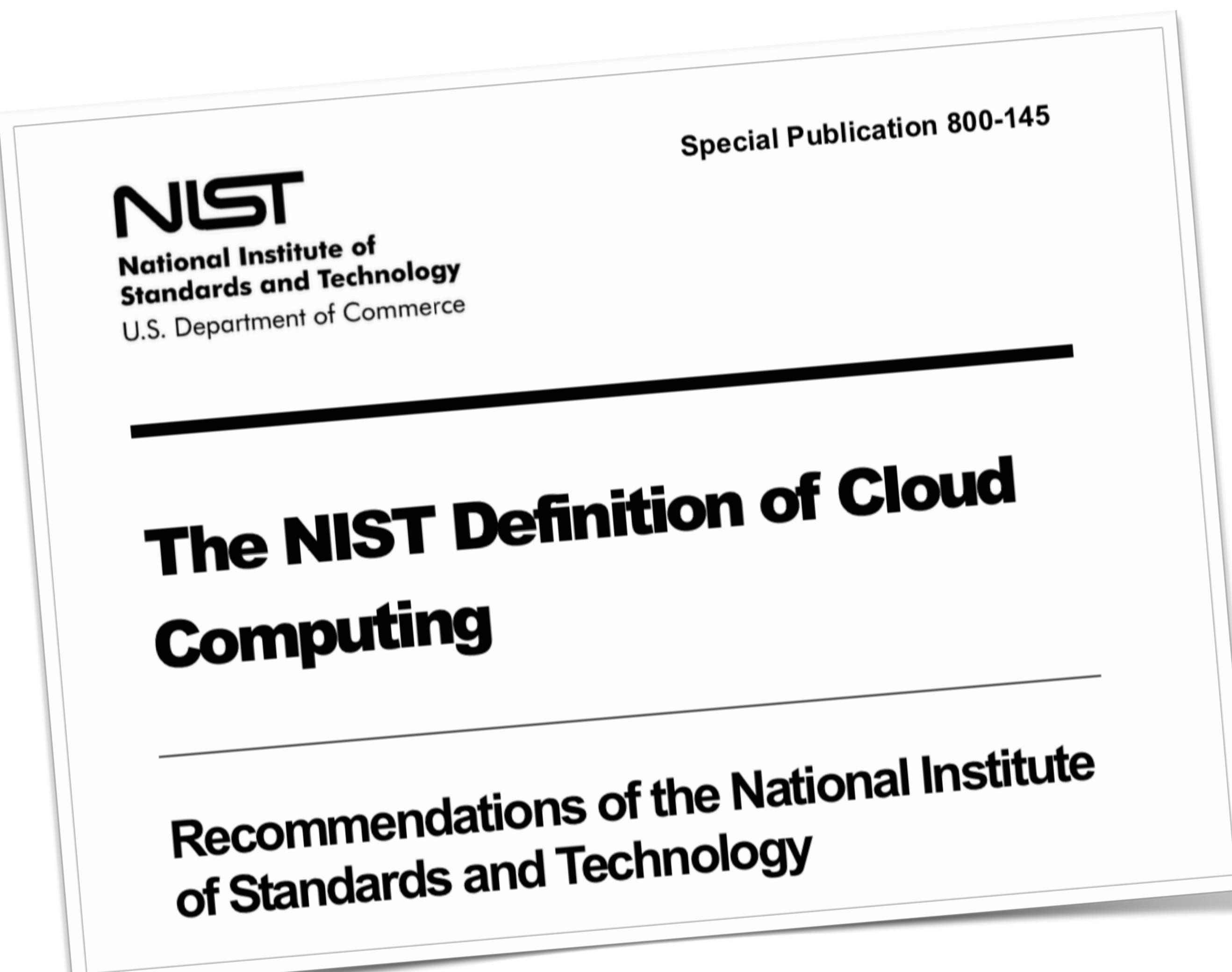
- ▶ Because each actor is an individual process in full control of its state, it doesn't matter "where" it resides: location transparency
- ▶ Same VM? Same node? Same Cluster?
- ▶ Easy to scale by spawning new actors in different locations
- ▶ Moving actors around (migration)
- ▶ The FT model makes it easy to distribute the application

# 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?

# ACTORS ON THE CLOUD

# CLOUD COMPUTING



# CLOUD COMPUTING



**The  
Computing**

*“Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction...”*

**Recommendations of the National Institute  
of Standards and Technology**

# CLOUD COMPUTING

## Essential Characteristics:

*On-demand self-service.* A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.

*Broad network access.* Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).

*Resource pooling.* The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.

# CLOUD COMPUTING

*Rapid elasticity.* Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

*Measured service.* Cloud systems automatically control and optimize resource use by leveraging a metering capability<sup>1</sup> at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

# CLOUD COMPUTING

- ▶ Software as a Service (SaaS) [mailing]
- ▶ Infrastructure as a Service (IaaS) [compute, network, storage]
- ▶ Platform as a Service (PaaS) [application engine, service fabric]
- ▶ Metal as a Service [crypto miners]
- ▶ Function as a Service (FaaS) [aws-lamnda]
- ▶ ....

# WHY SHOULD I USE CC

- ▶ Workload Provisioning
  - ▶ How many machine should I buy?
  - ▶ How much storage?
  - ▶ What kind of machine?
  - ▶ Too few resources = bad service
  - ▶ Too many resources = waste
  - ▶ But the workloads change over time

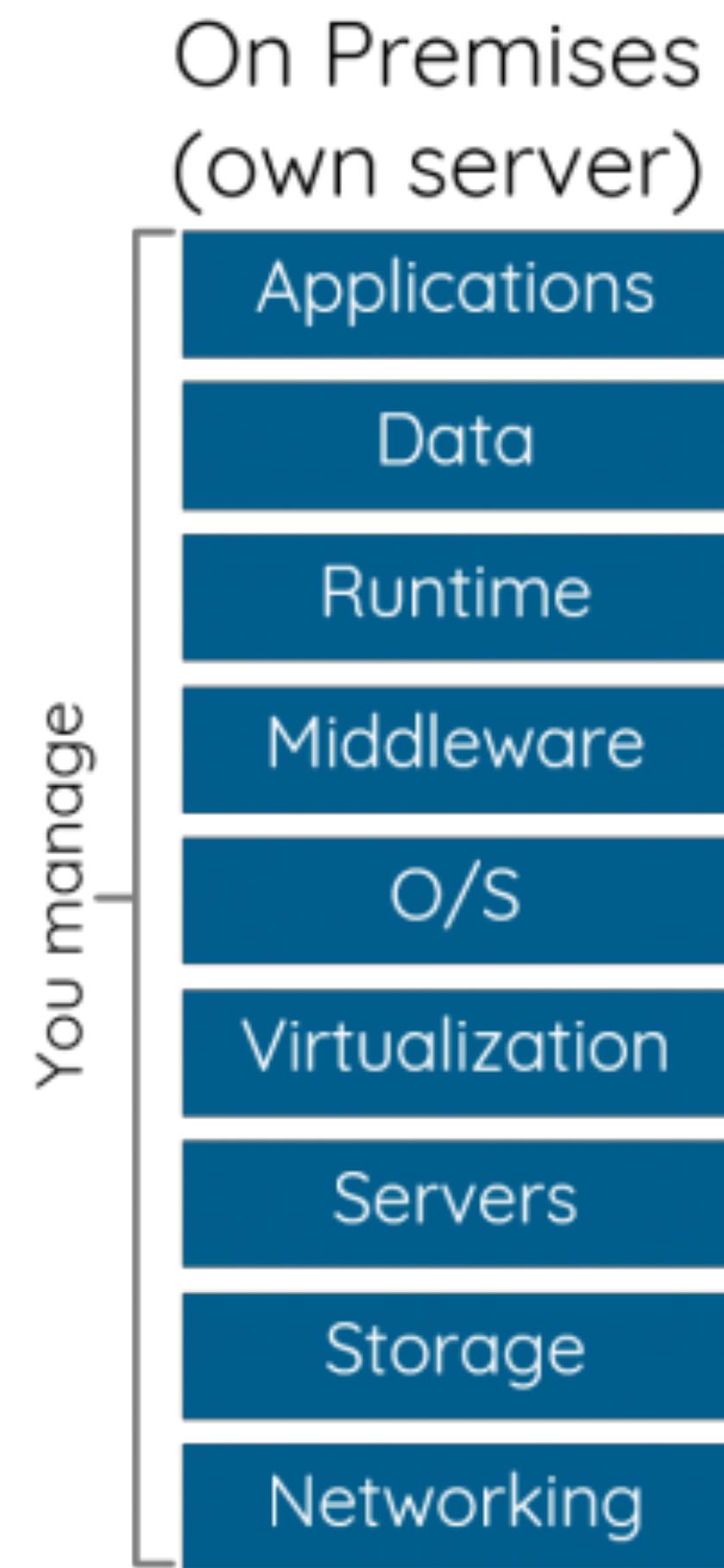
# WHY SHOULD I USE CC

- ▶ Workload Provisioning
  - ▶ How many machine should I buy?
  - ▶ How much storage?
  - ▶ What kind of machine?
  - ▶ Too few resources = bad service
  - ▶ Too many resources = waste
  - ▶ But the workloads change over time

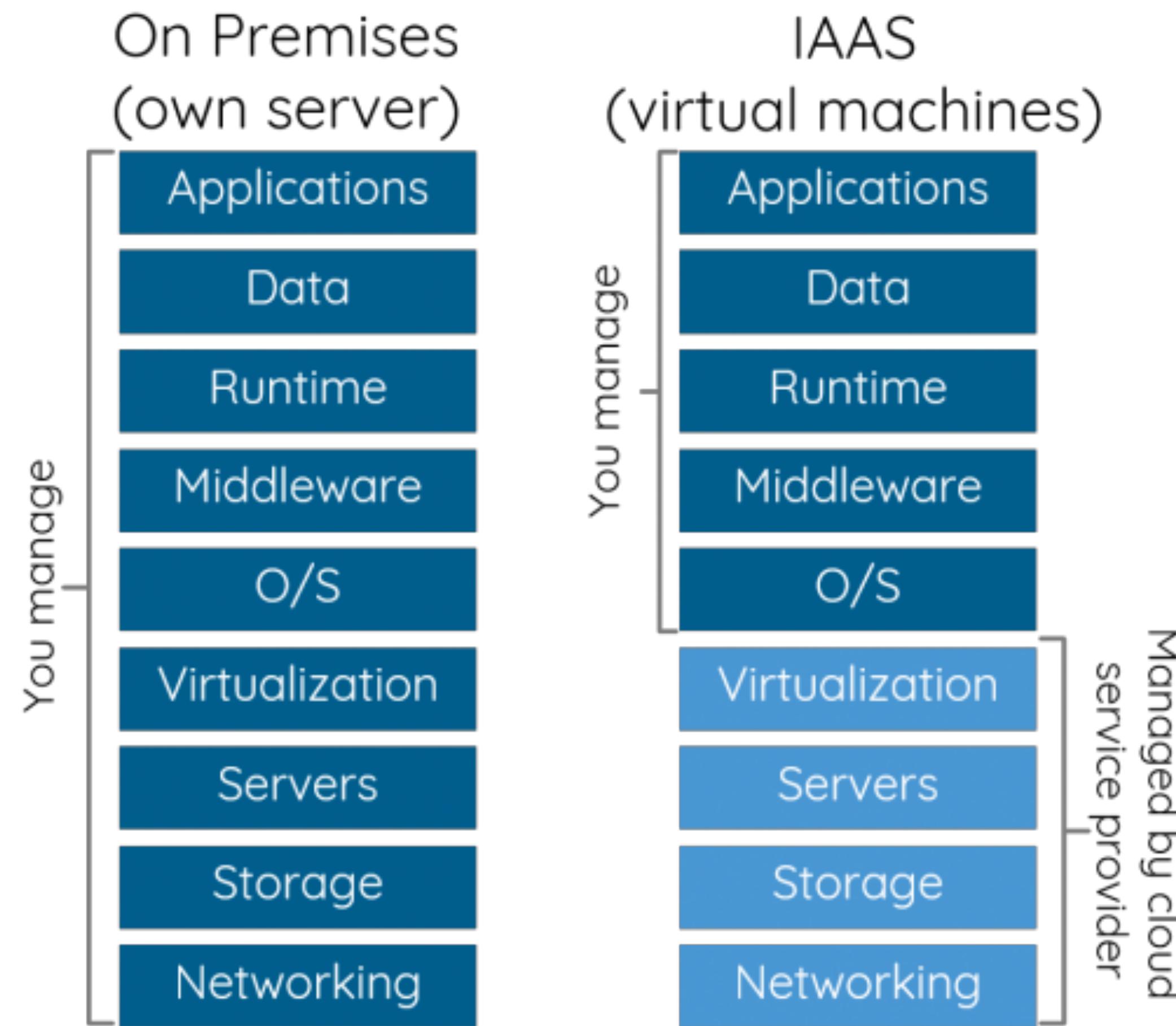


Elasticity

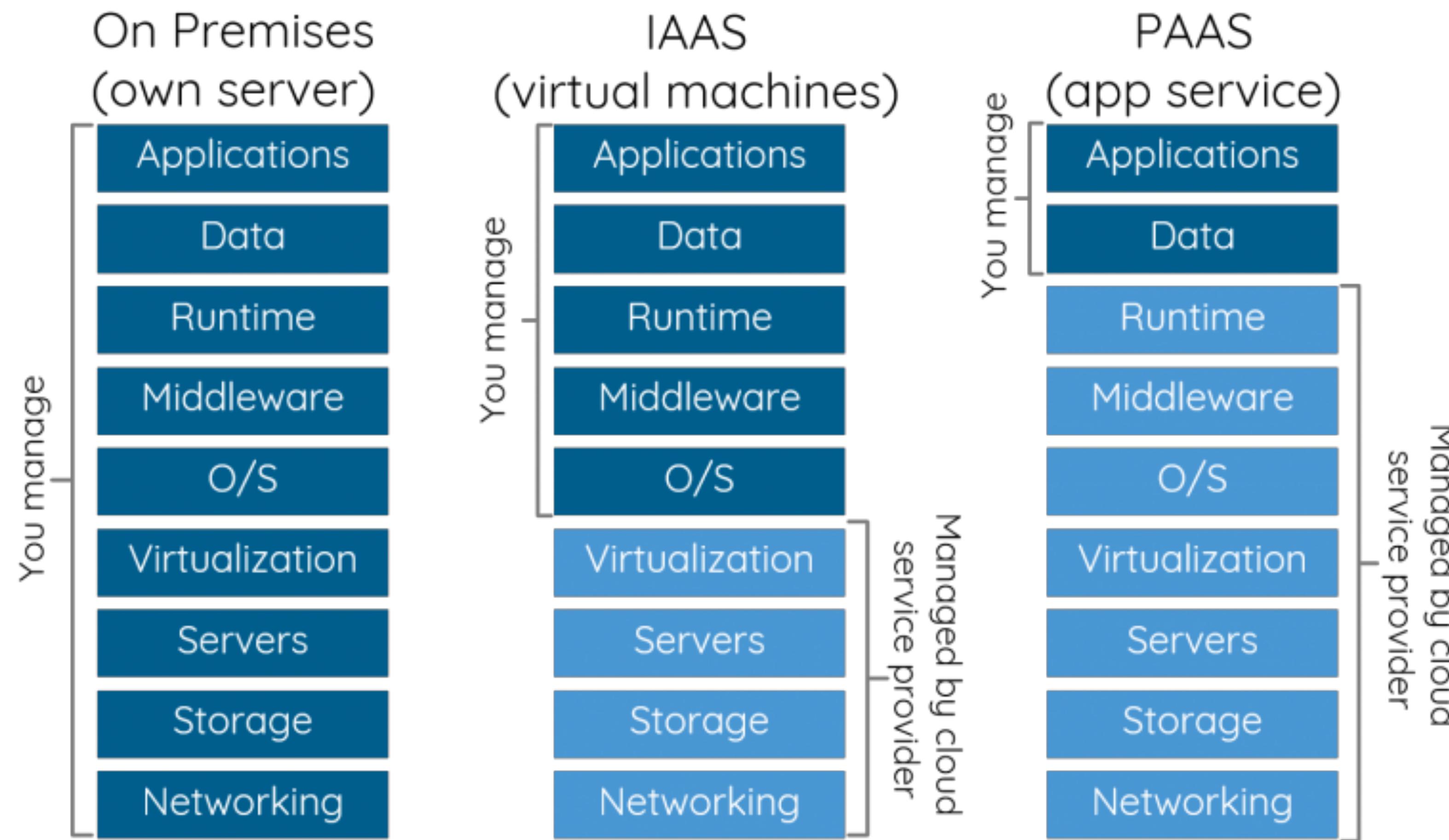
# ADVANTAGES OF CC



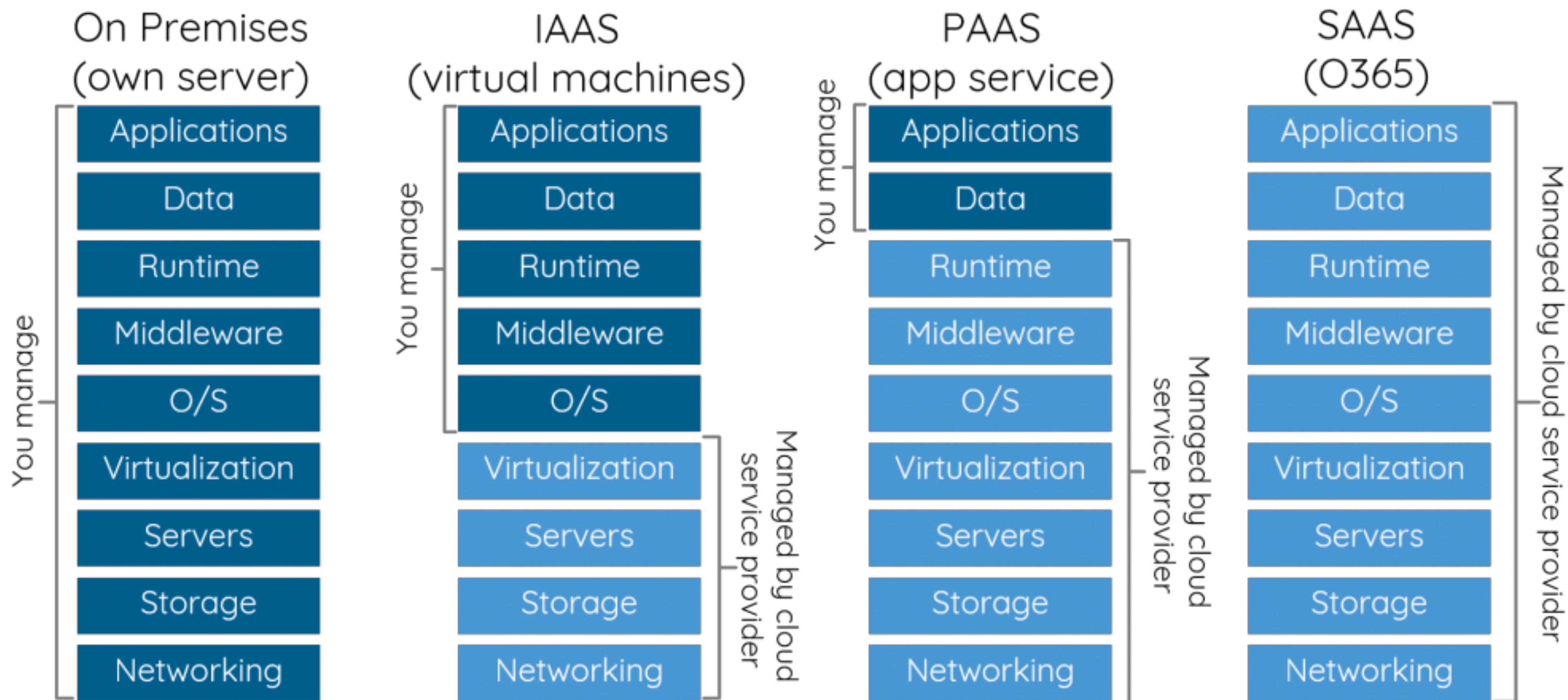
# ADVANTAGES OF CC



# ADVANTAGES OF CC



# ADVANTAGES OF CC



# BENEFITS OF CC

- ▶ Customer
  - ▶ Economy
  - ▶ Scalability / Elasticity
  - ▶ Management
  - ▶ Availability

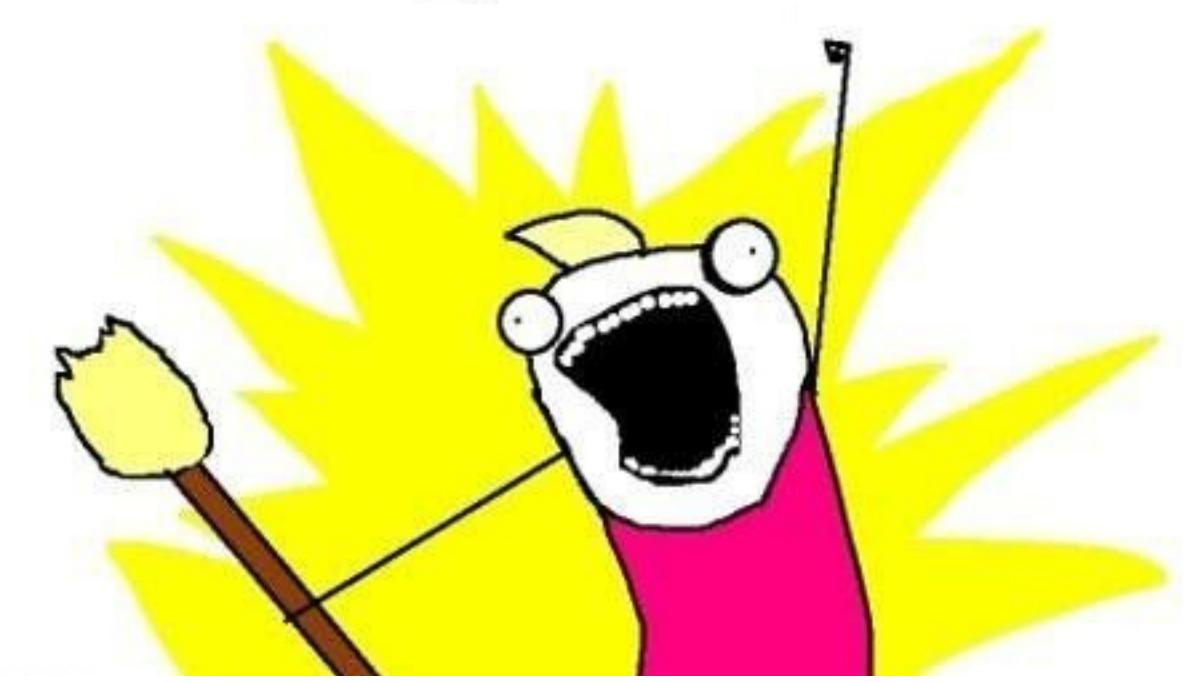
# BENEFITS OF CC

- ▶ Provider
  - ▶ Monetization of existing infrastructure
  - ▶ Multi-tenancy → near optimal use of resources
  - ▶ New business model(s)
  - ▶ IoT

# VIRTUALIZATION

- ▶ CC requires virtualization
- ▶ Abstracts from the underlying infrastructure
- ▶ Provide the same API as the expected infrastructure would provide

**VIRTUALIZE ALL THE THINGS**



# BENEFITS OF VIRTUALIZATION

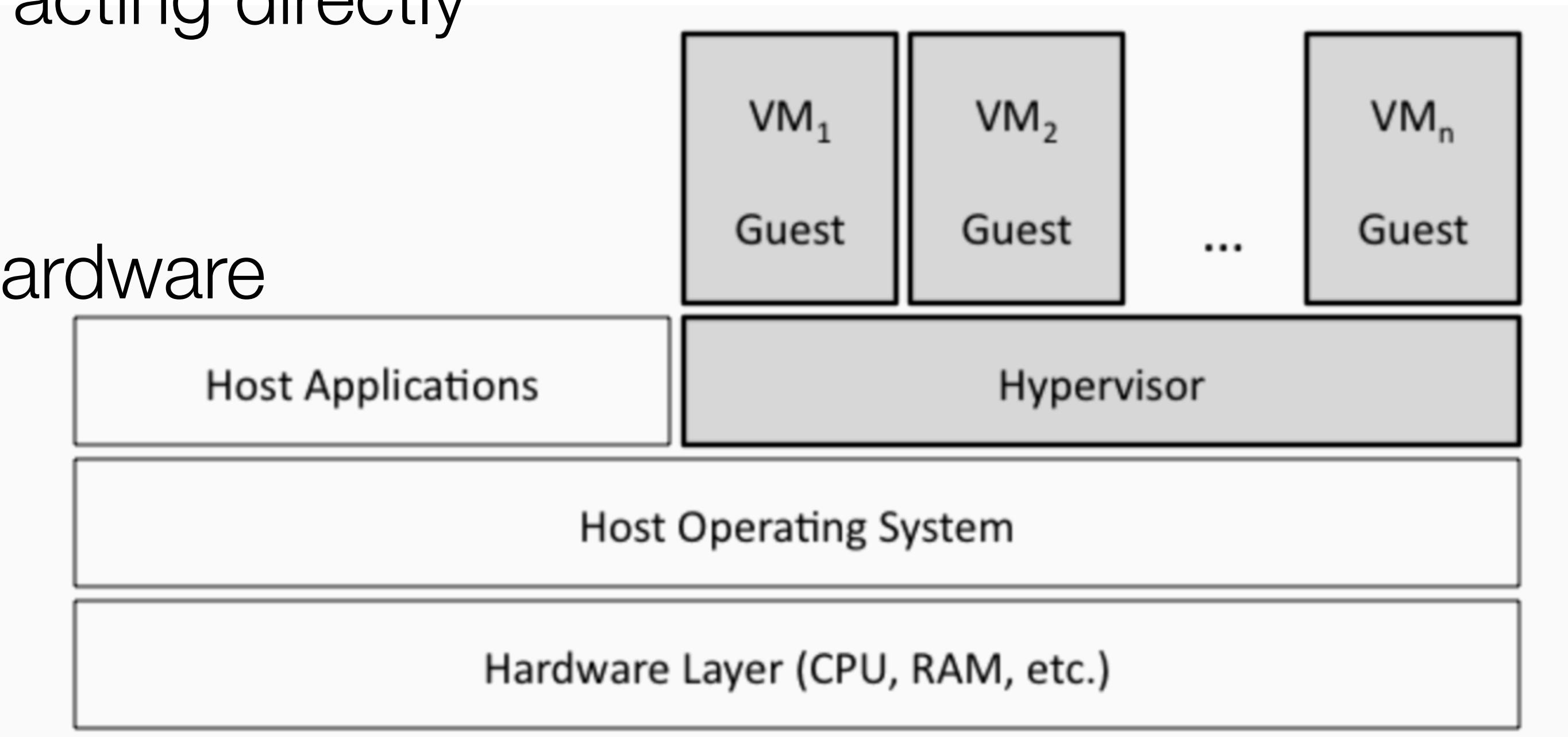
- ▶ Isolates application and its resources
- ▶ It decouples the application from the infrastructure
- ▶ Applications are self-contained
- ▶ Can run anywhere
- ▶ No physical hardware assumptions = simpler maintenance
- ▶ Helps optimizing resource usage

**VIRTUALIZE ALL THE THINGS**



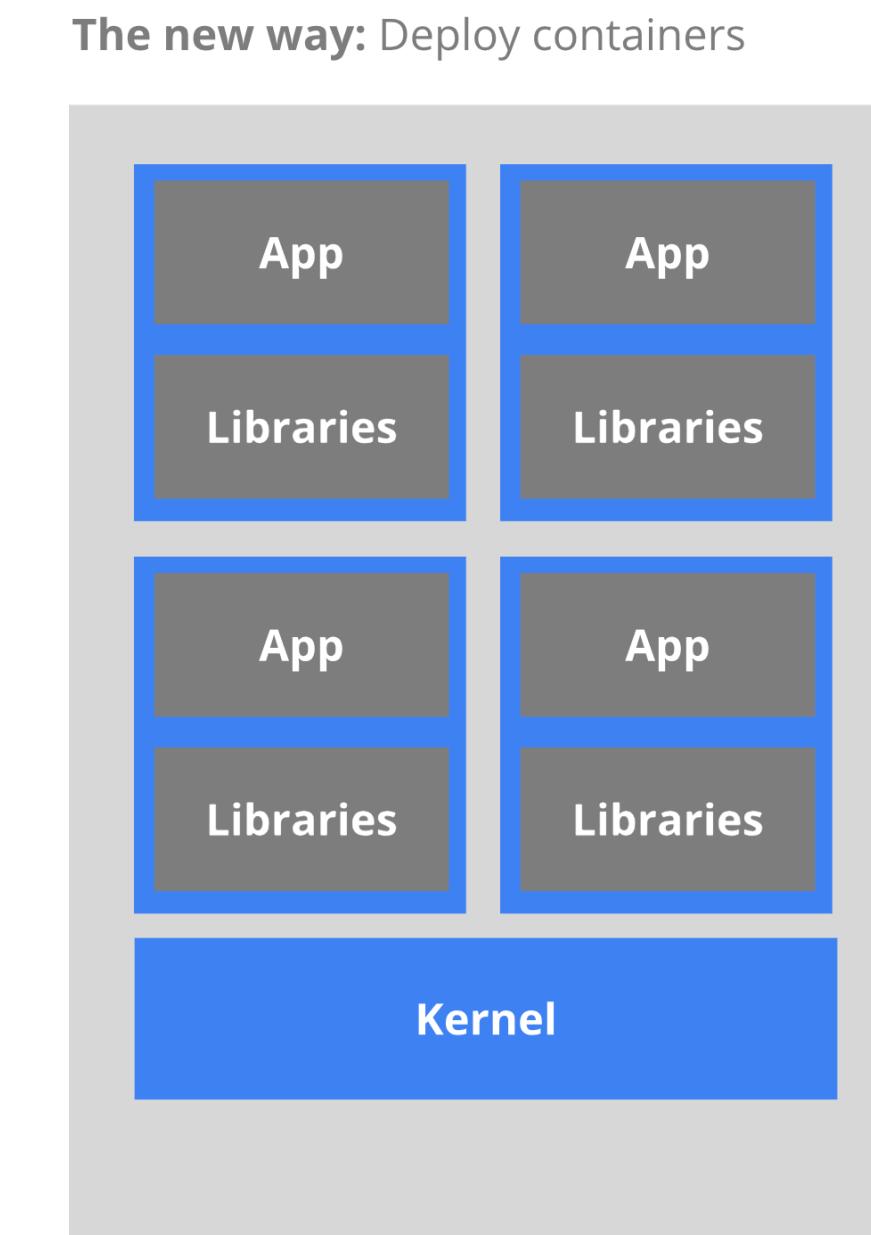
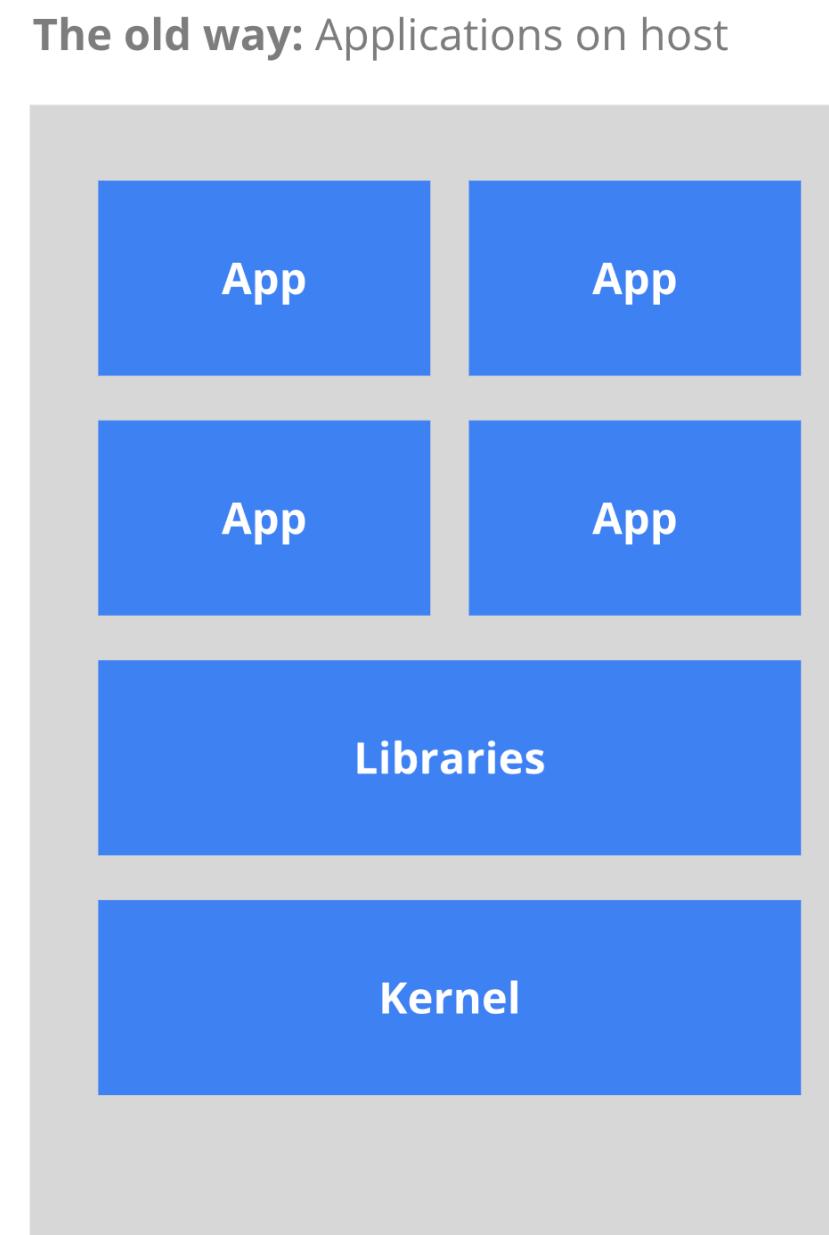
# VIRTUAL MACHINES

- ▶ VMs run on top of a hypervisor
  - ▶ Either a program in a host OS
  - ▶ Or a bare-metal program interacting directly with the devices
- ▶ Provides an abstraction of the hardware
  - ▶ The Operating System
  - ▶ VirtualBox quick demo



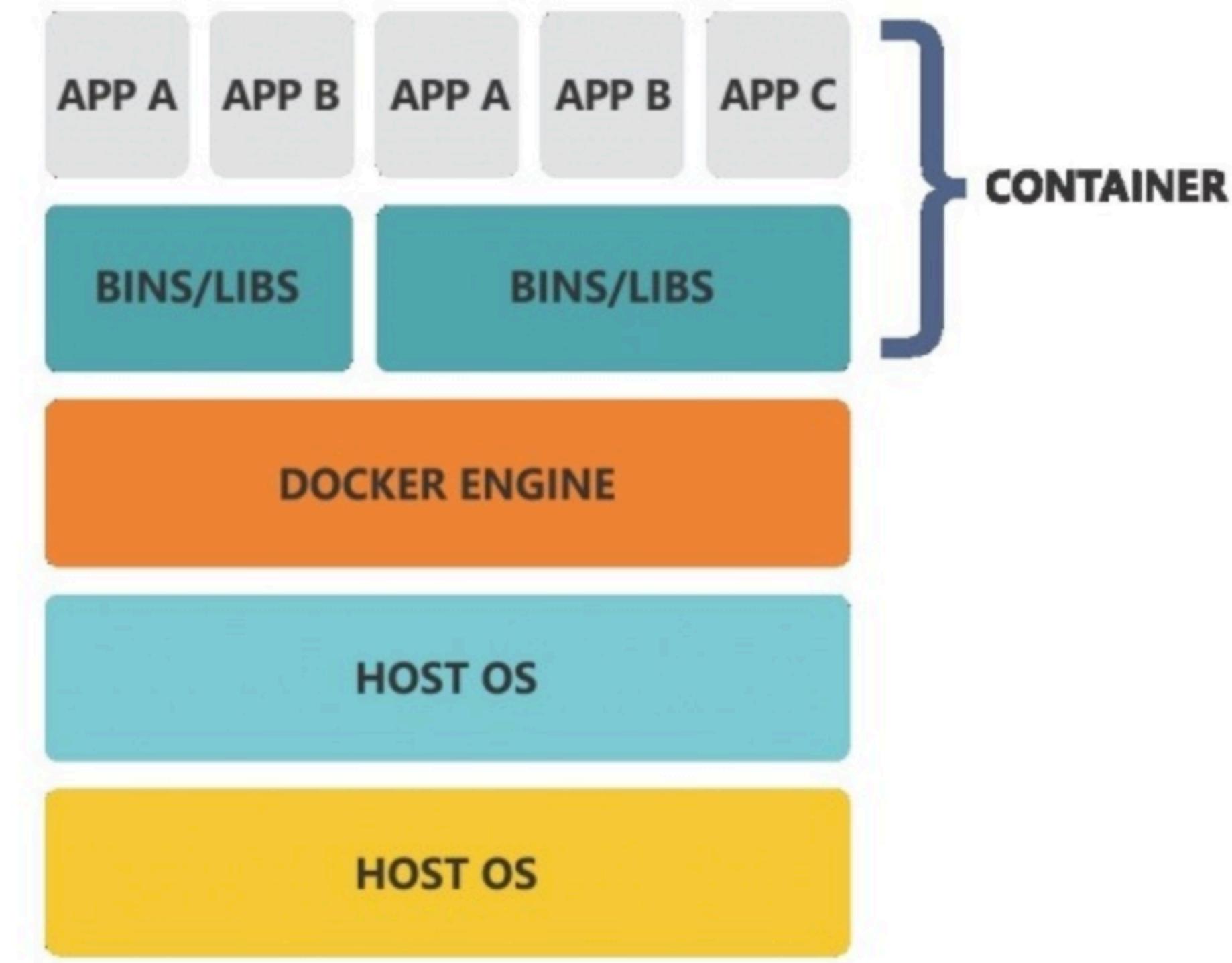
# CONTAINERS

- ▶ Provides Operating System virtualization
  - ▶ A user-space for each container
- ▶ Containers share a single kernel on the host OS



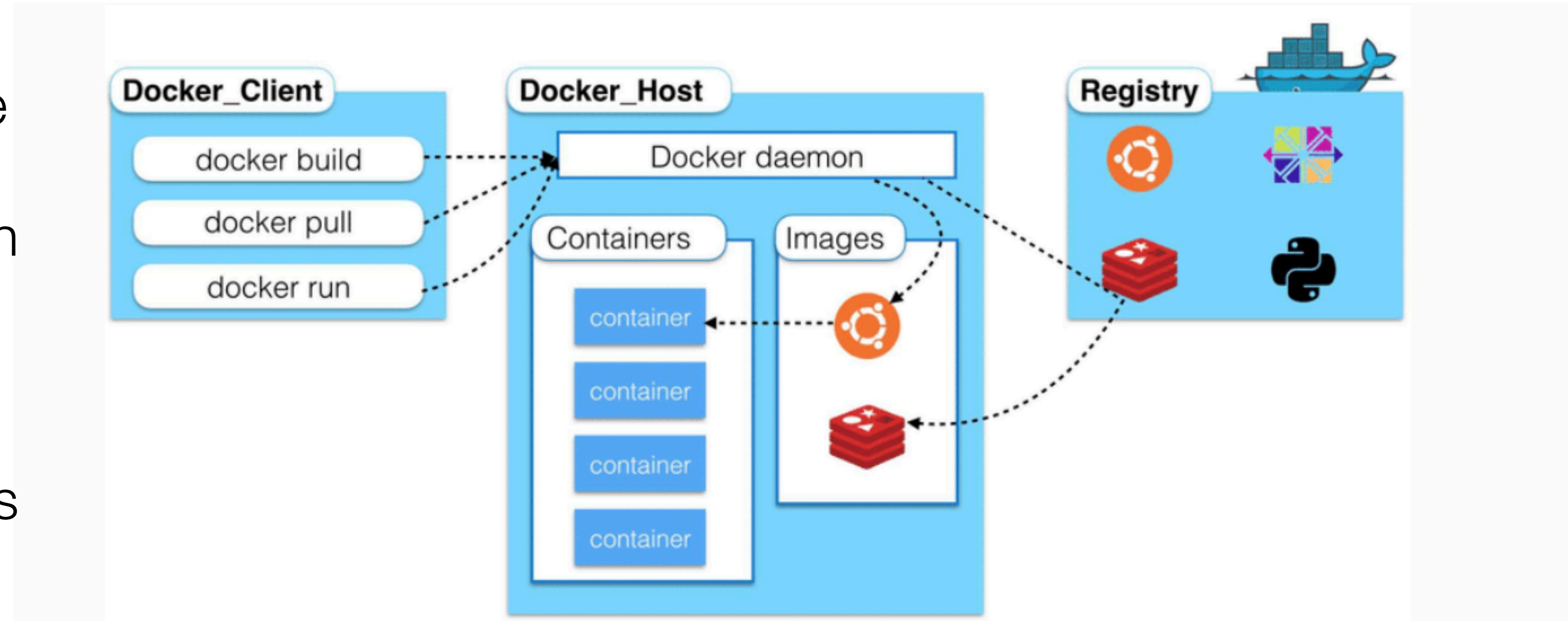
*Heavyweight, non-portable  
Relies on OS package manager*

*Small and fast, portable  
Uses OS-level virtualization*



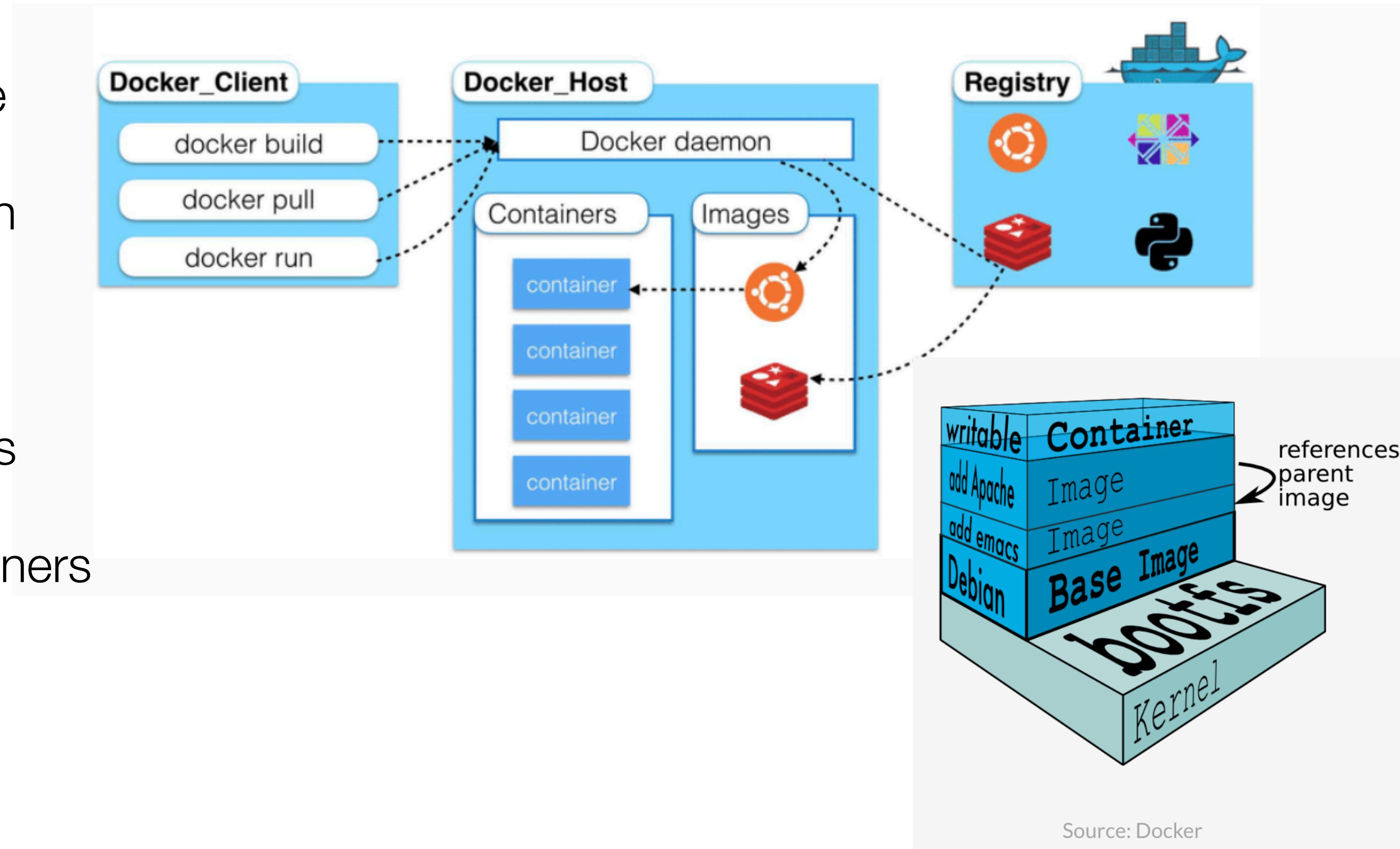
# CONTAINERS (DOCKER)

- ▶ Docker Engine
- ▶ Docker Demon
- ▶ Docker Client
- ▶ Docker Images
- ▶ Docker Containers
- ▶ Quick Demo



# CONTAINERS (DOCKER)

- ▶ Docker Engine
- ▶ Docker Demon
- ▶ Docker Client
- ▶ Docker Images
- ▶ Docker Containers
- ▶ Quick Demo



# UTILITIES

- ▶ Packaging and deploying an Akka application in a docker container
- ▶ Configuration of Akka applications
  - ▶ /src/main/resources

# RUNNING AKKA ON EC2

# RUNNING AKKA ON EC2

Akka Actor Application



# RUNNING AKKA ON EC2

Akka Actor Application

Akka Runtime System  
(Scala/JVM)



# RUNNING AKKA ON EC2

Akka Actor Application

Akka Runtime System  
(Scala/JVM)

Docker Container



# RUNNING AKKA ON EC2

Akka Actor Application



Akka Runtime System  
(Scala/JVM)



Docker Container



Linux Virtual Machine



# RUNNING AKKA ON EC2

Akka Actor Application



Akka Runtime System  
(Scala/JVM)



Docker Container



Linux Virtual Machine



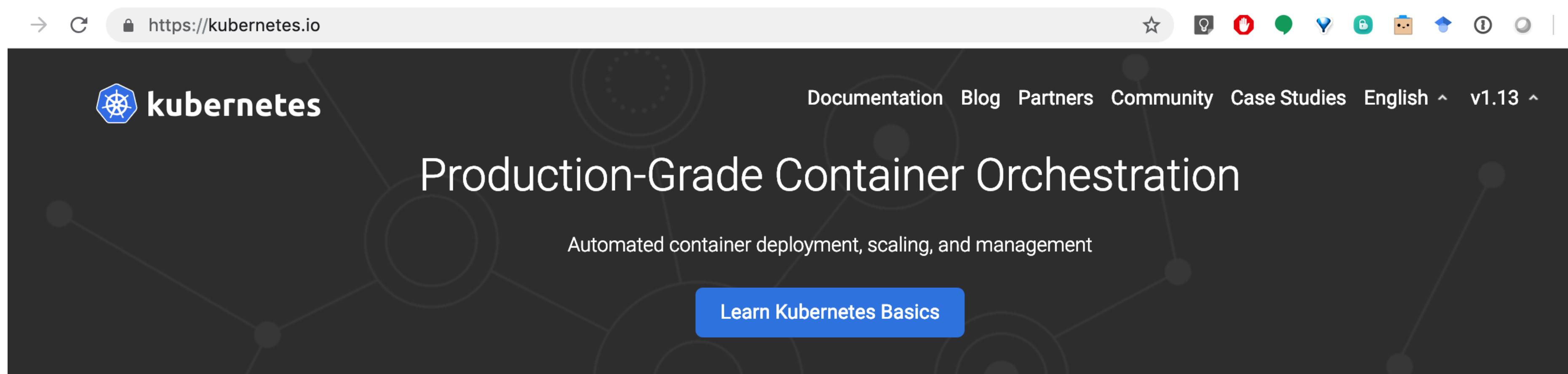
EC2 Cloud Infrastructure



# CLOUD OFFERINGS

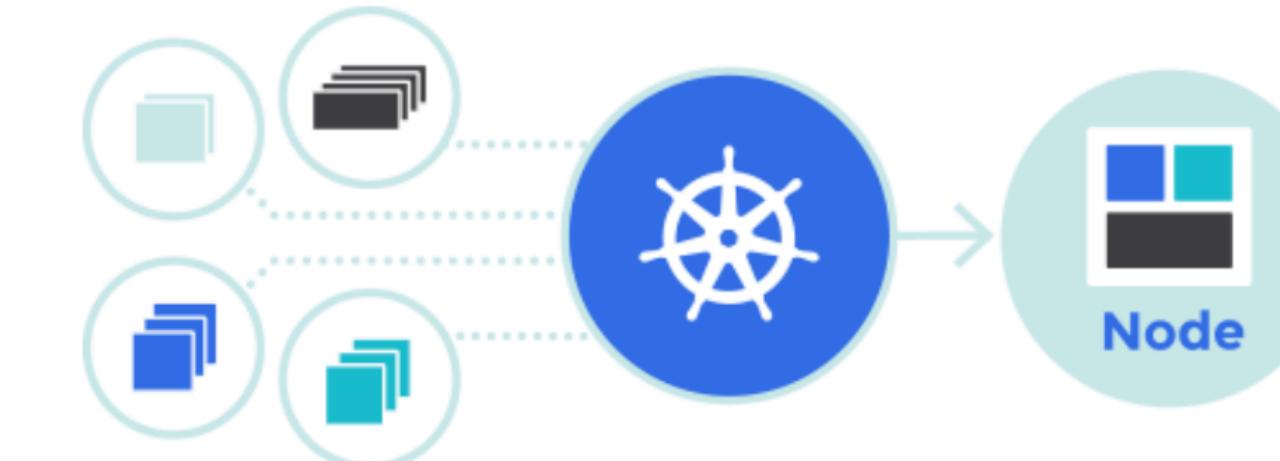
- ▶ Get a VM in AWS running
  - ▶ Demo
- ▶ Deploy a simple server in AWS
- ▶ Orchestration: Kubernetes
- ▶ Elasticity

# ORCHESTRATION (KUBERNETES)



[Kubernetes \(k8s\)](#) is an open-source system for automating deployment, scaling, and management of containerized applications.

It groups containers that make up an application into logical units for easy management and discovery. Kubernetes builds upon [15 years of experience of running production workloads at Google](#), combined with best-of-breed ideas and practices from the community.

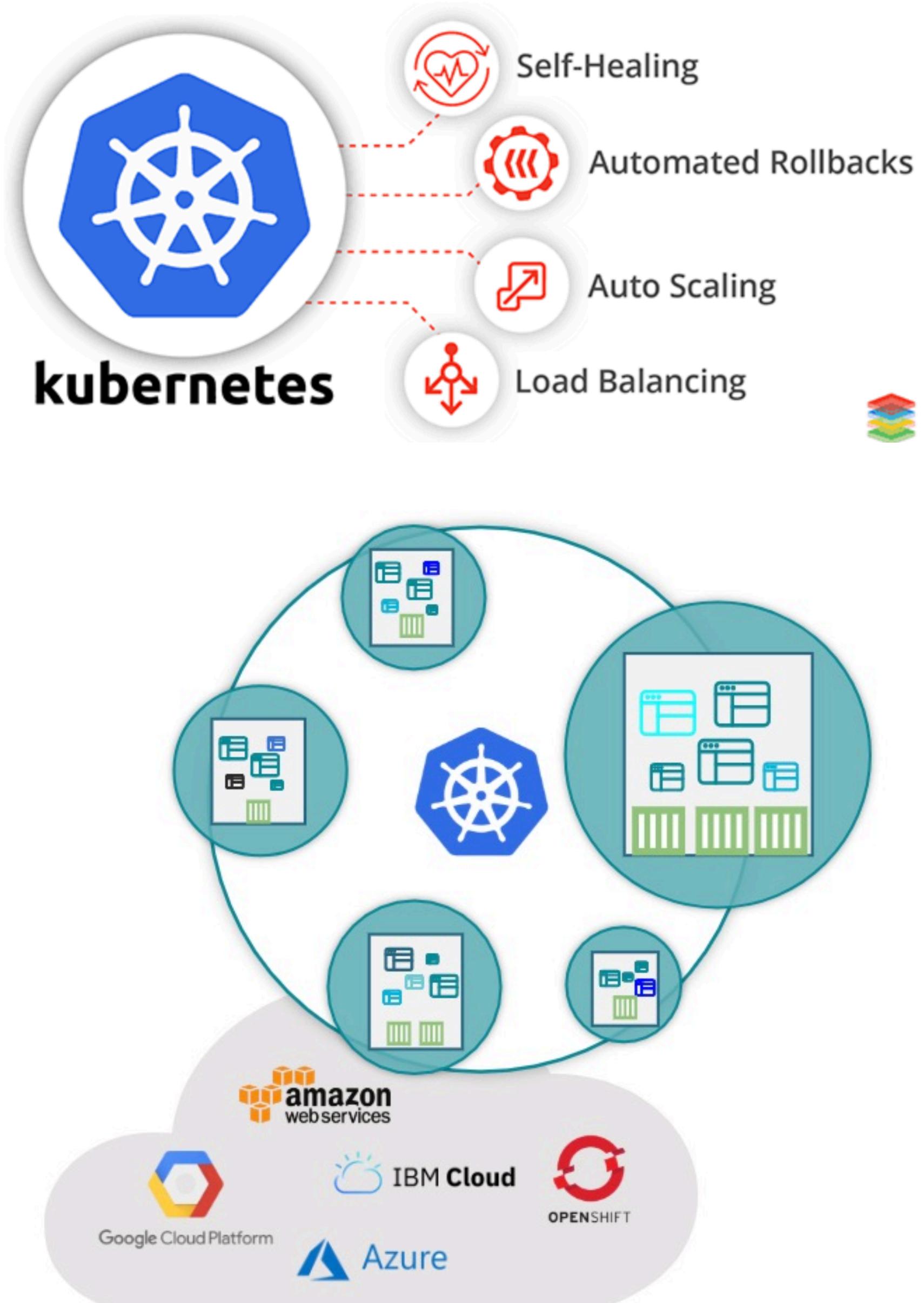


## Planet Scale

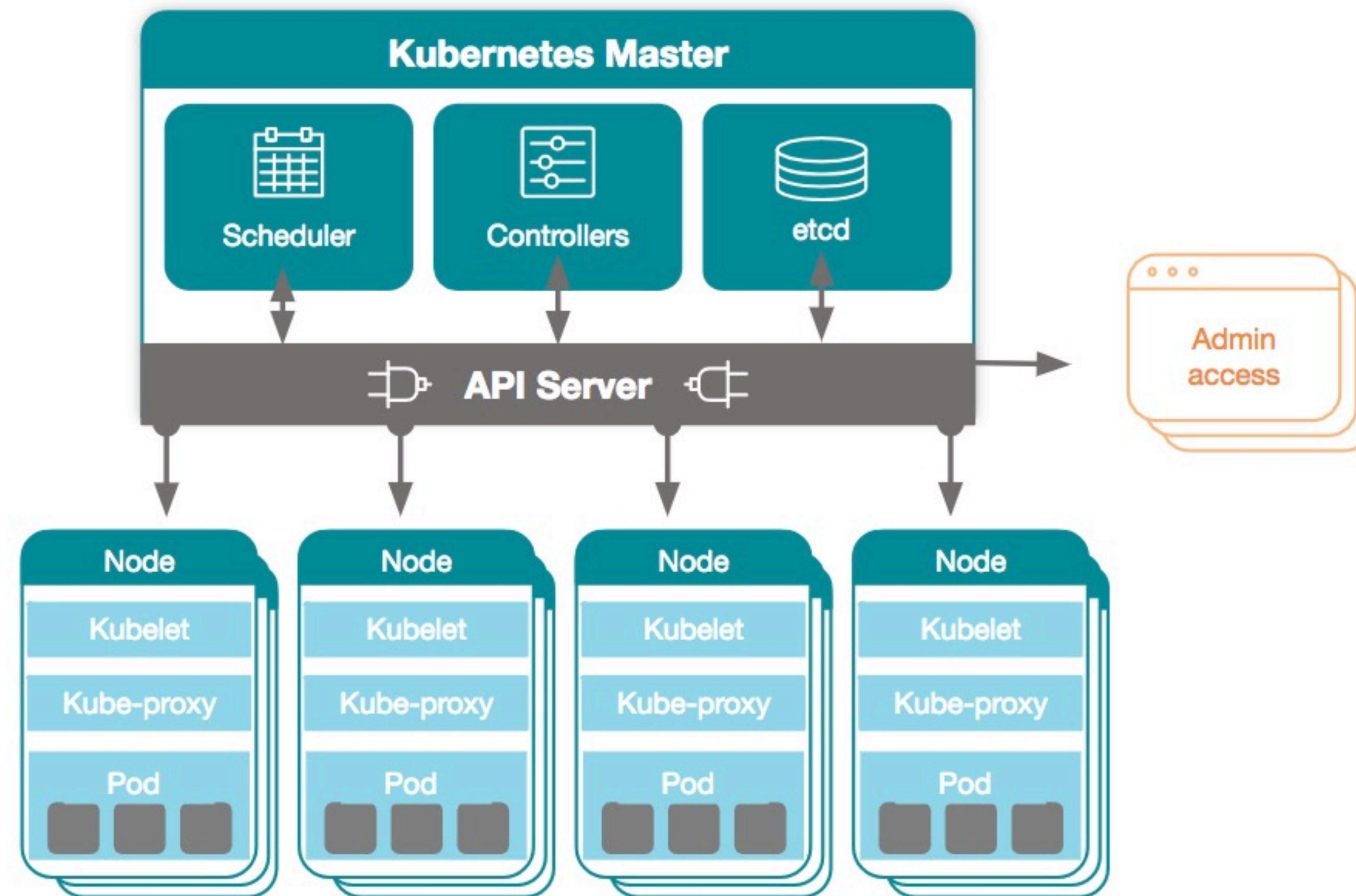
Designed on the same principles that allows Google to run billions of containers a week, Kubernetes can scale

# KUBERNETES

- ▶ Orchestration for containerized applications
- ▶ Virtualizes application deployment and operation (DevOps)
- ▶ Makes applications portable
- ▶ Automates the deployment and management
- ▶ Control Master (KubeMaster)
- ▶ Worker Node Controllers (Kubelet)



# KUBERNETES ARCHITECTURE



# KUBERNETES

## ► **KubeMaster**

- ▶ GUI + kubectl to control/configure the applications in the cluster
- ▶ Controls Nodes, Services, APIs, Networking, etc.

## ► **Kubelet**

- ▶ Manages the Docker containers running the workload
- ▶ Implements the commands from the KubeMaster

# KUBERNETES (CORE ENTITIES)

- ▶ **Cluster**: A set of hosts where applications can run
  - ▶ Could be virtual
- ▶ **Master**: Control Plane of Kubernetes
- ▶ **Node**: A host. This is where workloads run
- ▶ **Namespace**: Logical cluster or environment
- ▶ **Labels**: Keys used to name and filter objects/entities
- ▶ **Selector**: Predicates used to filter objects (using labels)

# KUBERNETES (CORE ENTITIES)

- ▶ **Pod**: A small set of containers that are managed together
  - ▶ Tightly coupled docker containers for instance
- ▶ **ReplicaSet**: Management of *sets of replicas* of pods
  - ▶ For instance, one could require that there must be at least 3 copies of a pod at all times
- ▶ **Deployment**: Instantiation of a number of Pods
- ▶ **Service**: Declares network aspects of the pods
- ▶ **Ingress**: Declares the mechanisms to access the application

# KUBERNETES (CORE ENTITIES)

- ▶ **Pod**: A small set of containers that are managed together
  - ▶ Tightly coupled docker containers for instance
- ▶ **ReplicaSet**: Management of *sets of replicas* of pods
  - ▶ For instance, one could require that there must be at least 3 copies of a pod at all times
- ▶ **Deployment**: Instantiation of a number of Pods
- ▶ **Service**: Declares network aspects of the pods
- ▶ **Ingress**: Declares the mechanisms to access the application
- ▶ **Volumes, Secrets, ...**

# MINIKUBE

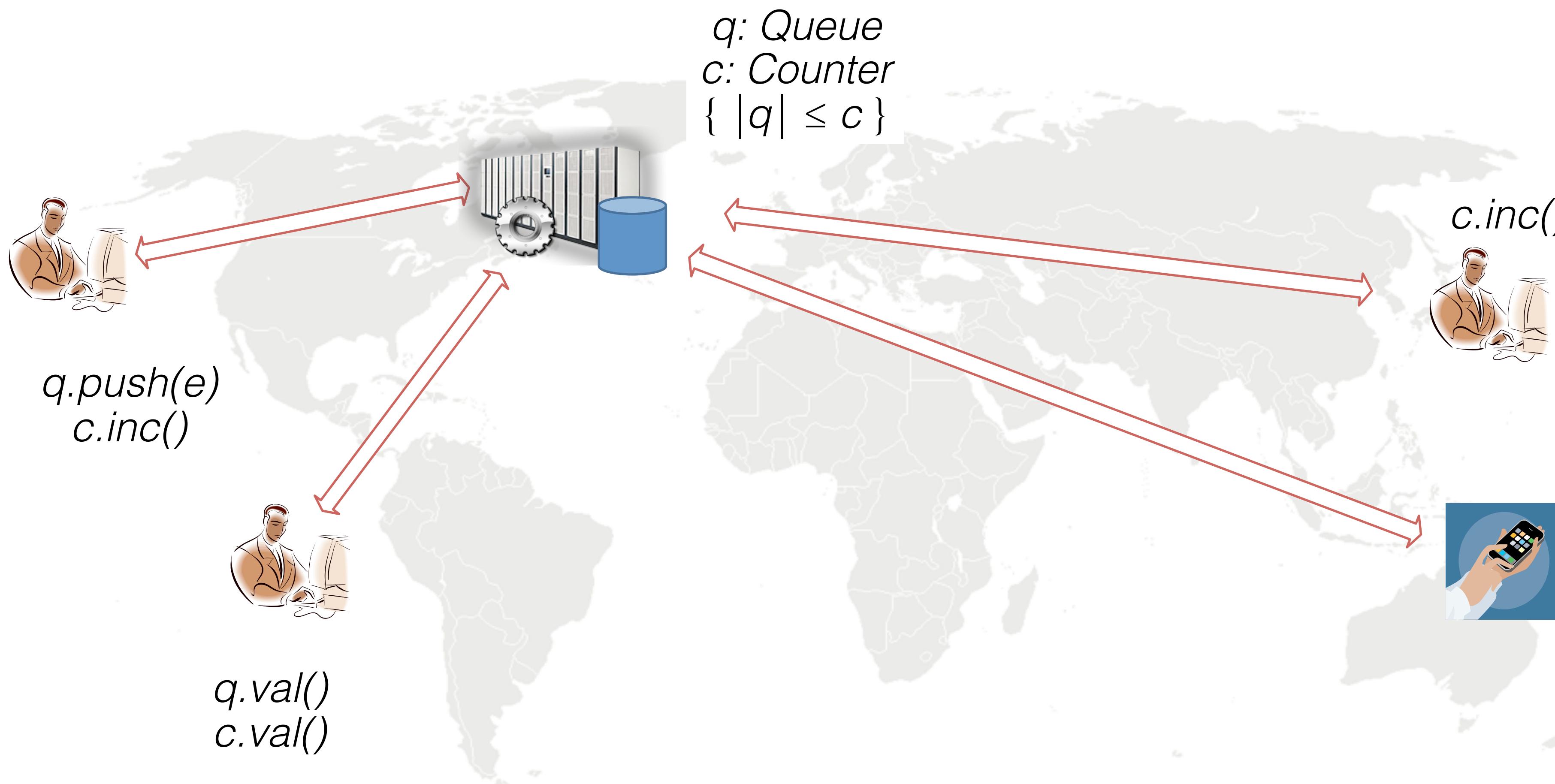
- ▶ Small VM that simulates a Kubernetes cluster
- ▶ Great for Testing kubernetes policies
- ▶ Lets see what it does: Demo

# AKKA ACTORS (OUTLINE)

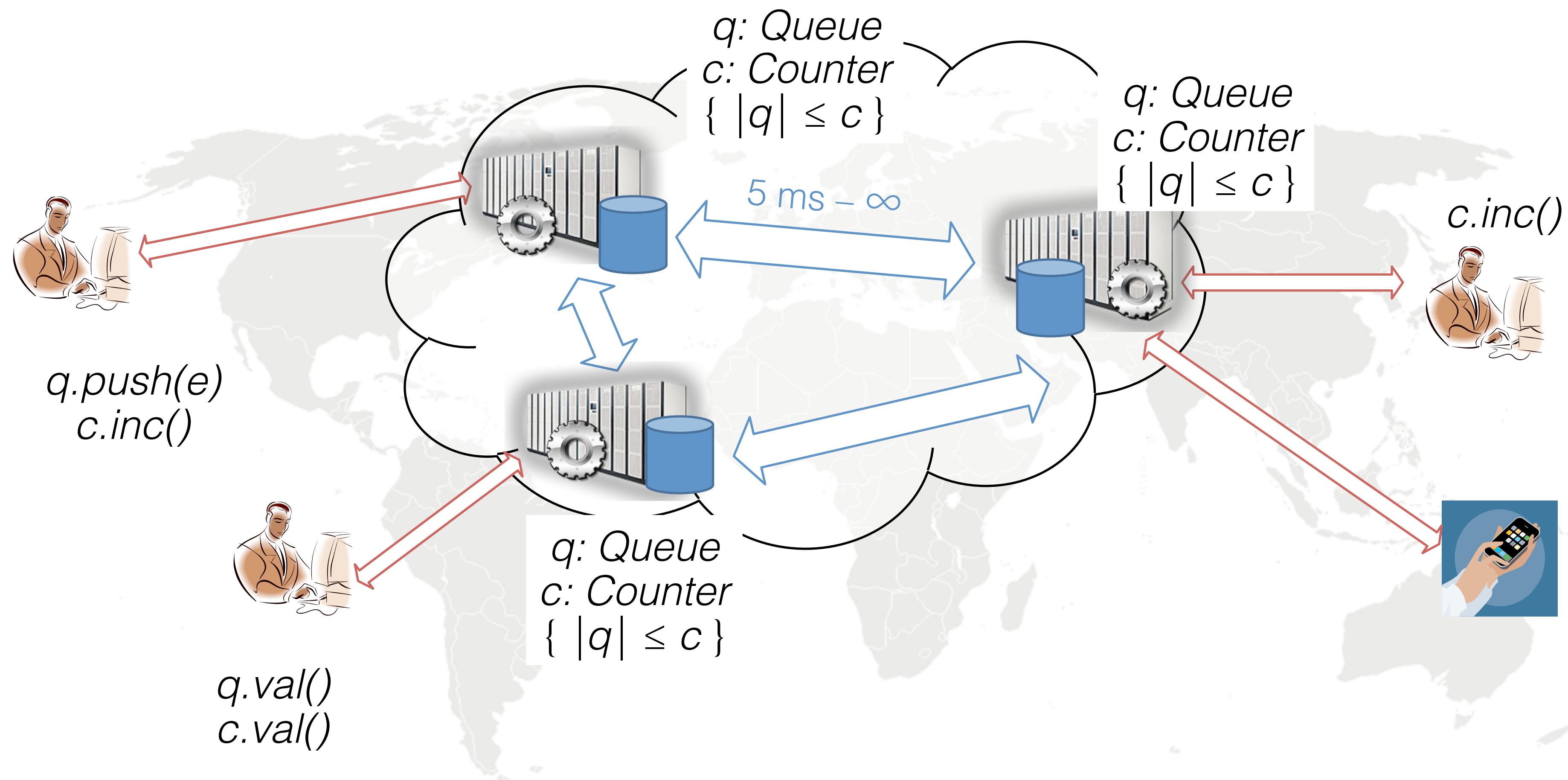
- ▶ Monday
  - ▶ Basics of Akka Programming
  - ▶ Fault Tolerance
  - ▶ Some Scala goodies
- ▶ Tuesday
  - ▶ Distribution
  - ▶ Scalability / Elasticity
  - ▶ Some Actor Patterns
- ▶ Wednesday
  - ▶ Cloud Computing
  - ▶ Virtualization
  - ▶ Akka on a cluster
- ▶ Thursday
  - ▶ Synchronization and State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

# CRDTs

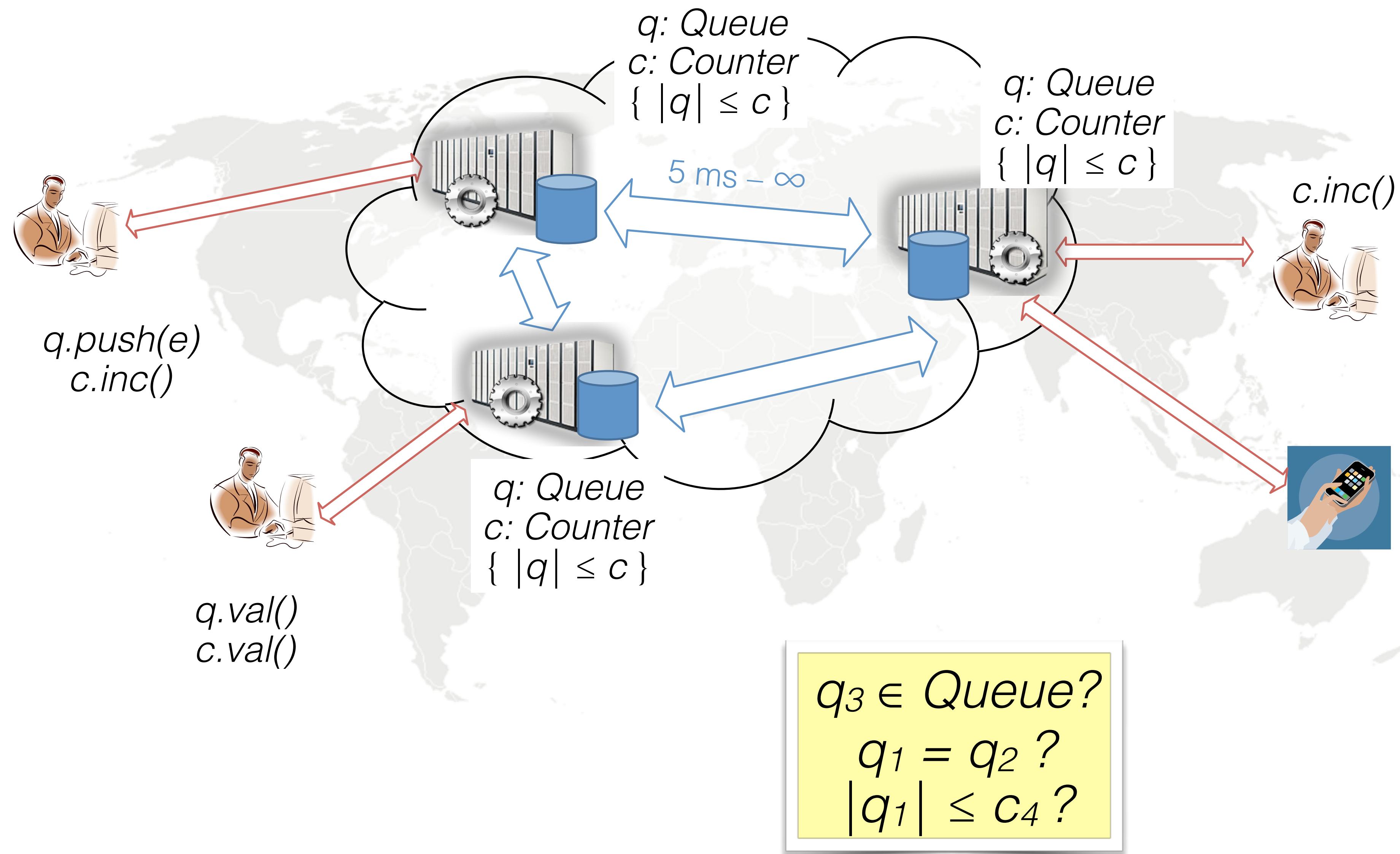
# SHARED DATA TYPE



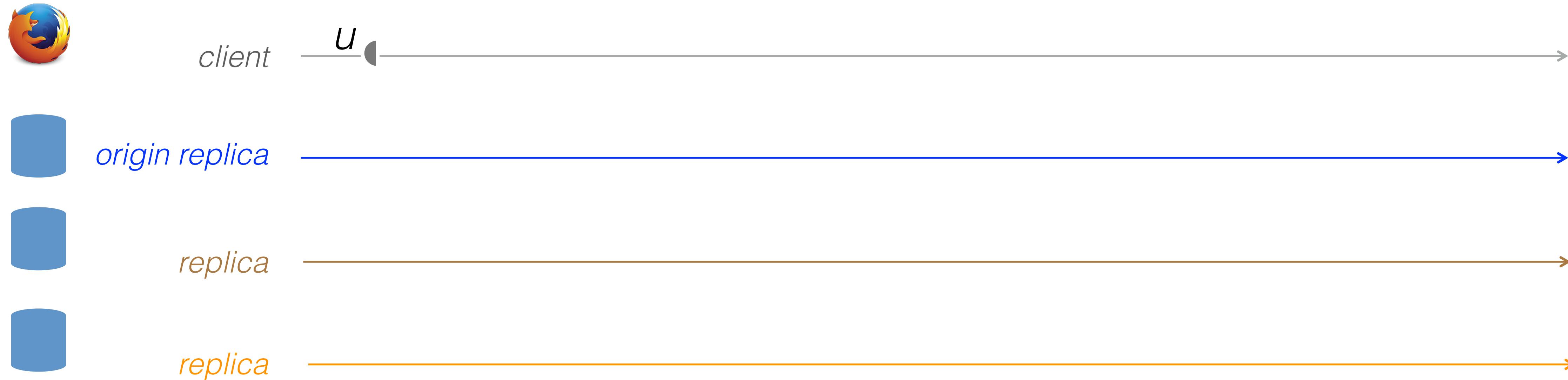
# GEO-REPLICATED DATA TYPE



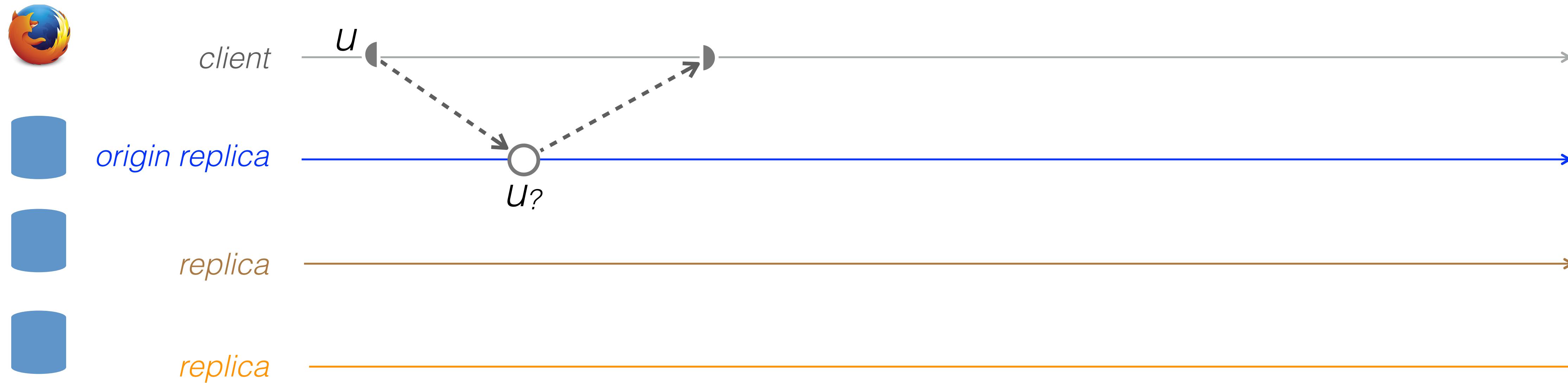
# GEO-REPLICATED DATA TYPE



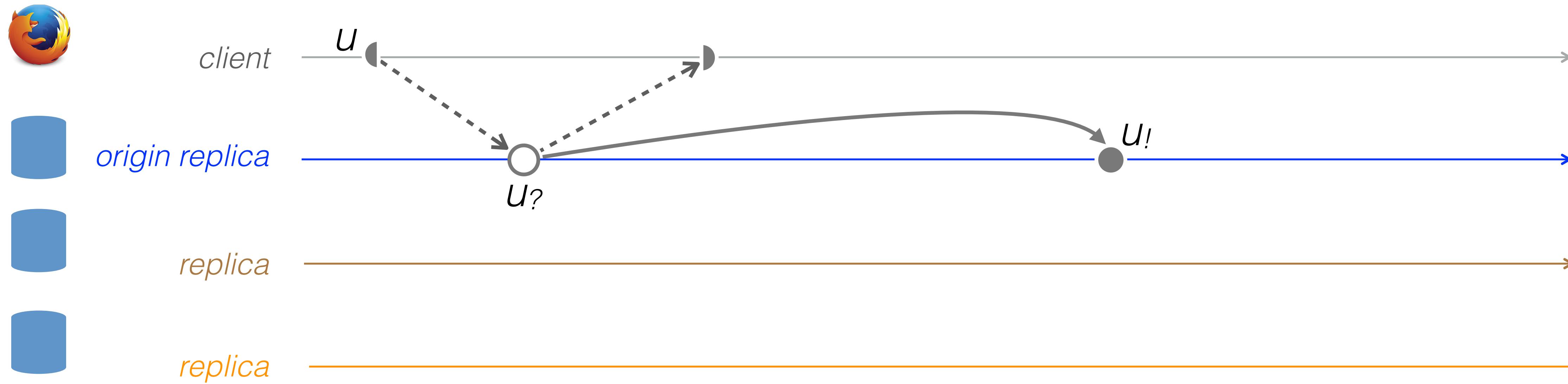
# PROGRAM MODEL (OPERATION)



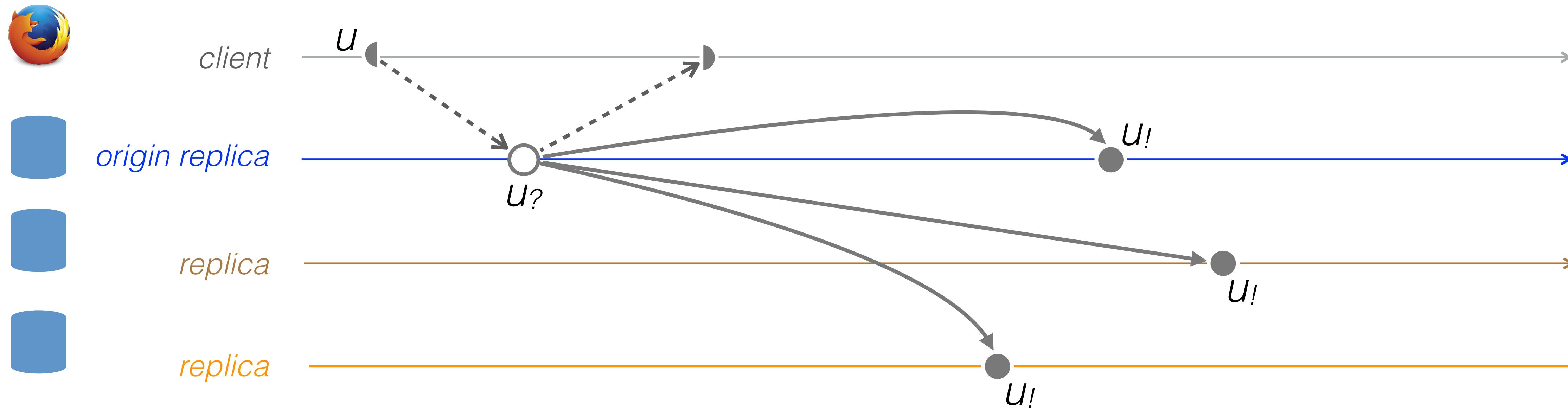
# PROGRAM MODEL (OPERATION)



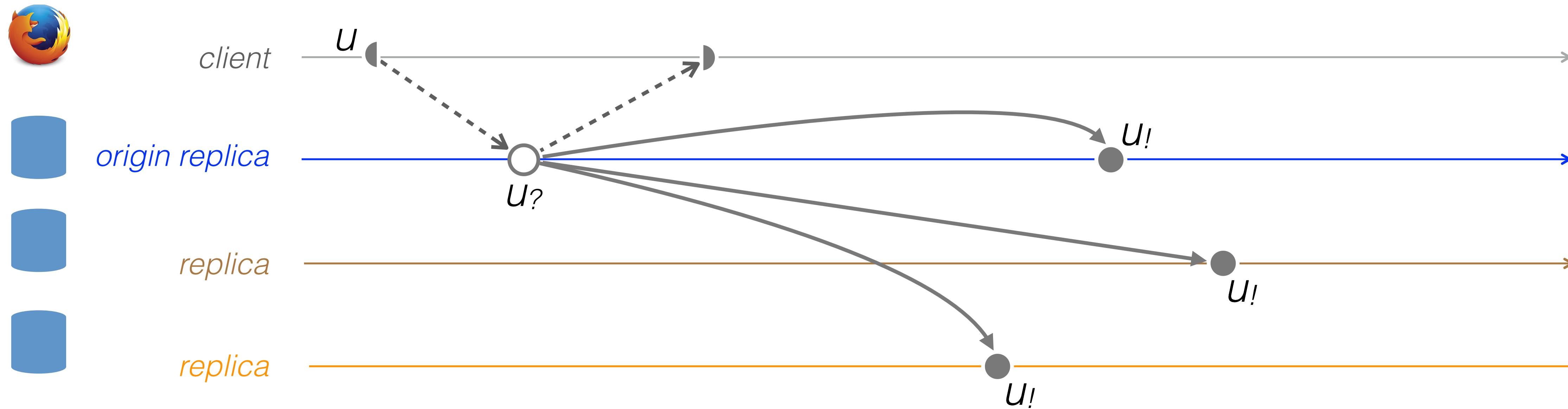
# PROGRAM MODEL (OPERATION)



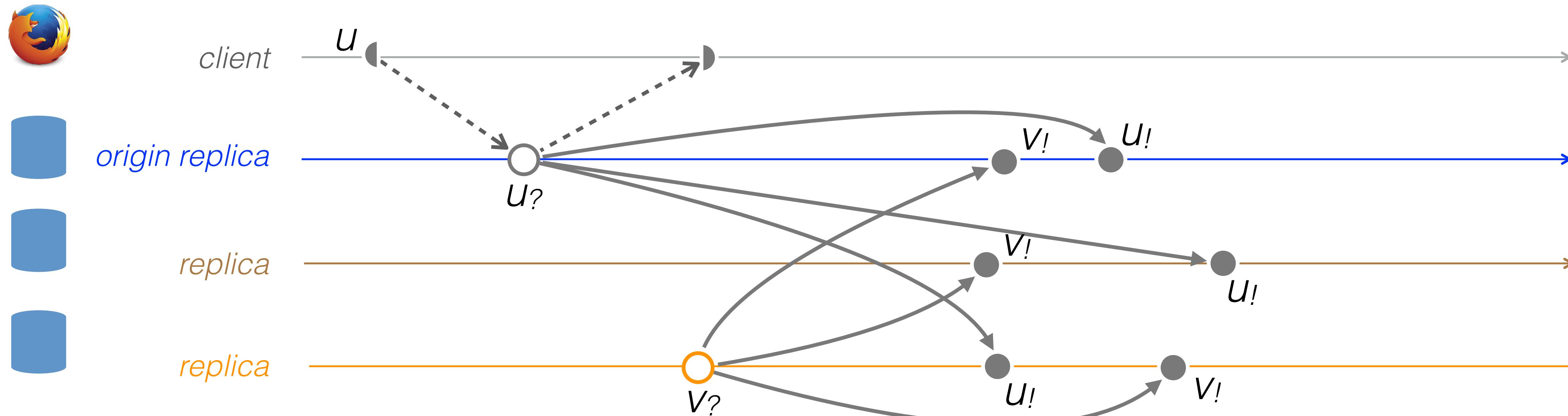
# PROGRAM MODEL (OPERATION)



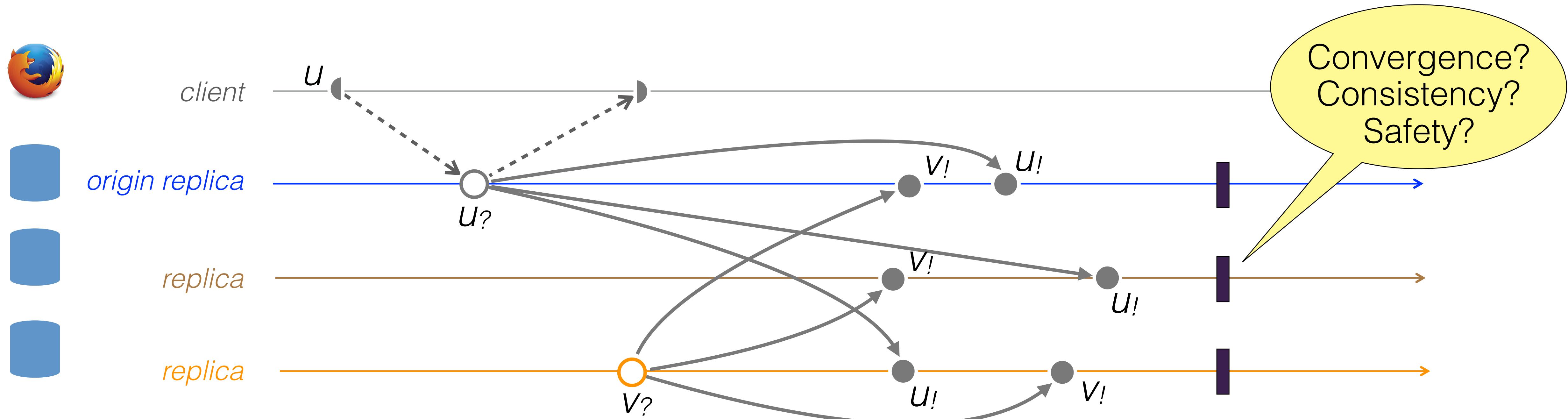
# PROGRAM MODEL (SYSTEM)



# PROGRAM MODEL (SYSTEM)

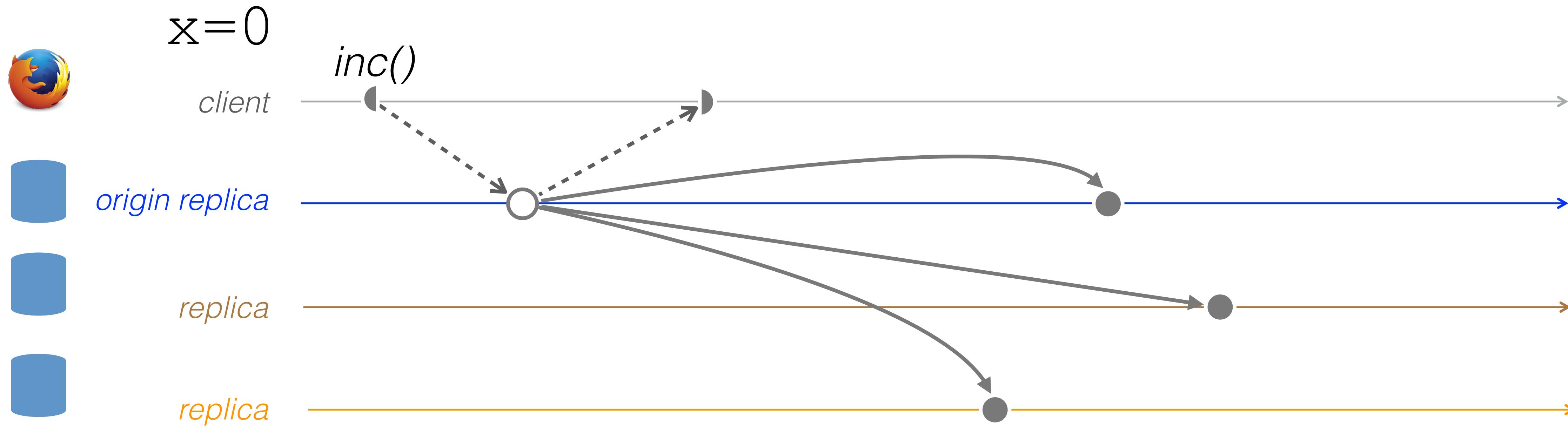


# PROGRAM MODEL (SYSTEM)

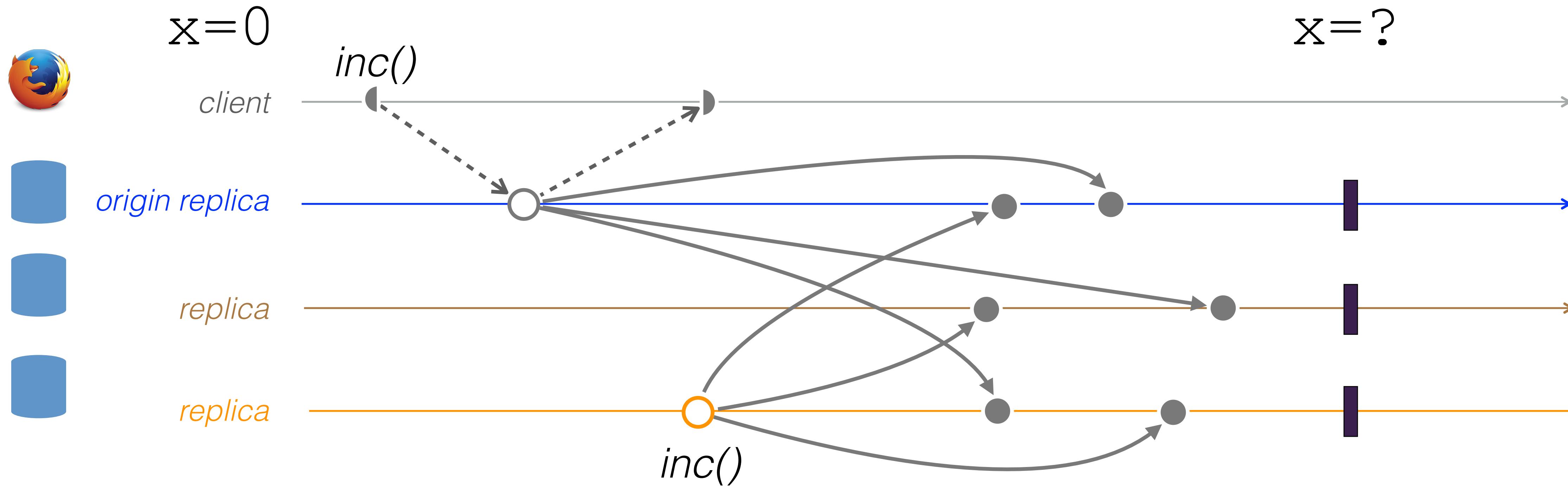


# CRDT EXAMPLES

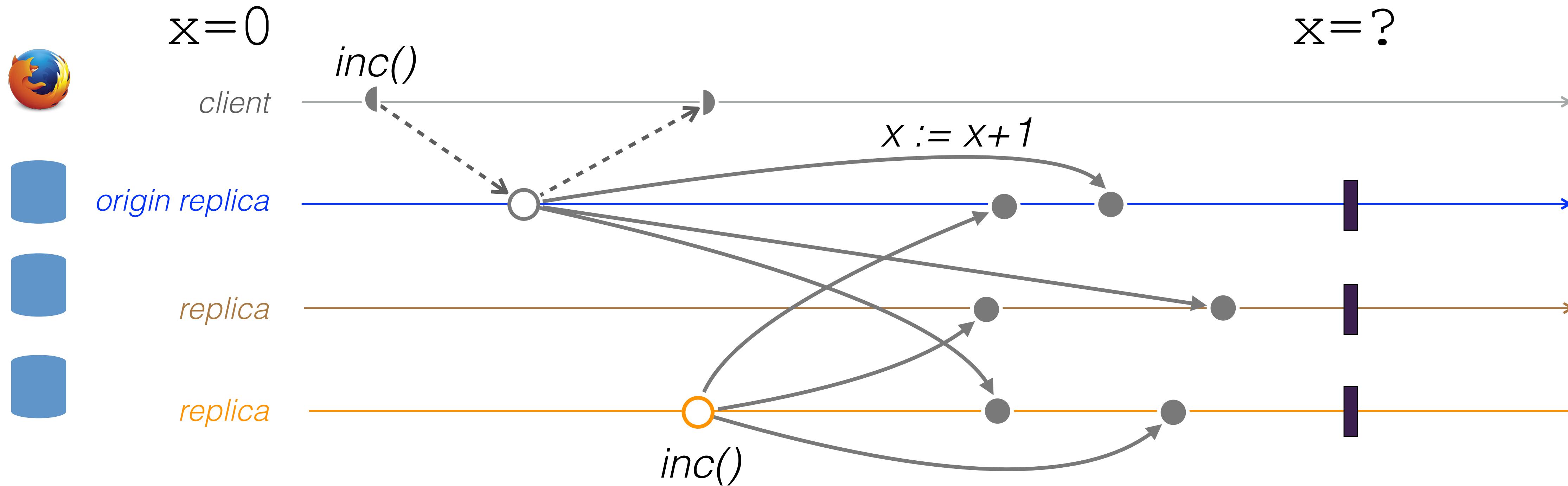
# EXAMPLE: GROW-ONLY COUNTER



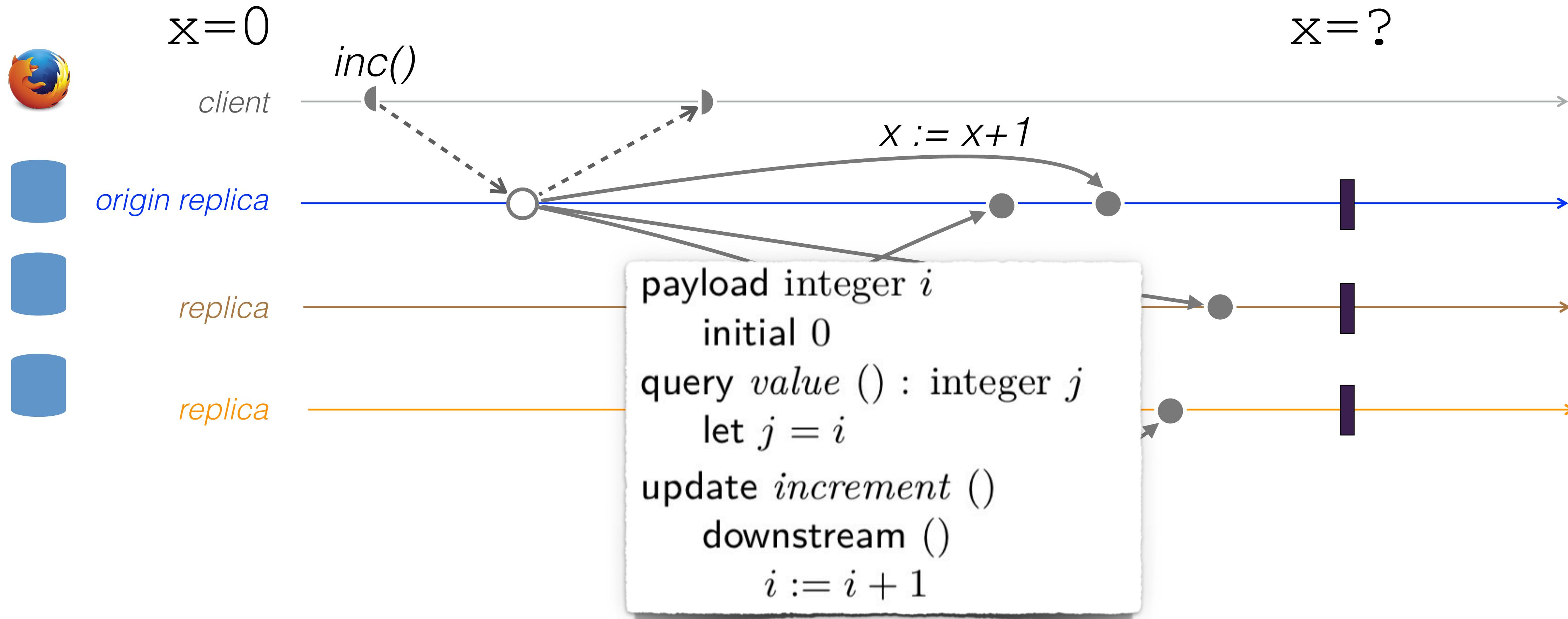
# EXAMPLE: GROW-ONLY COUNTER



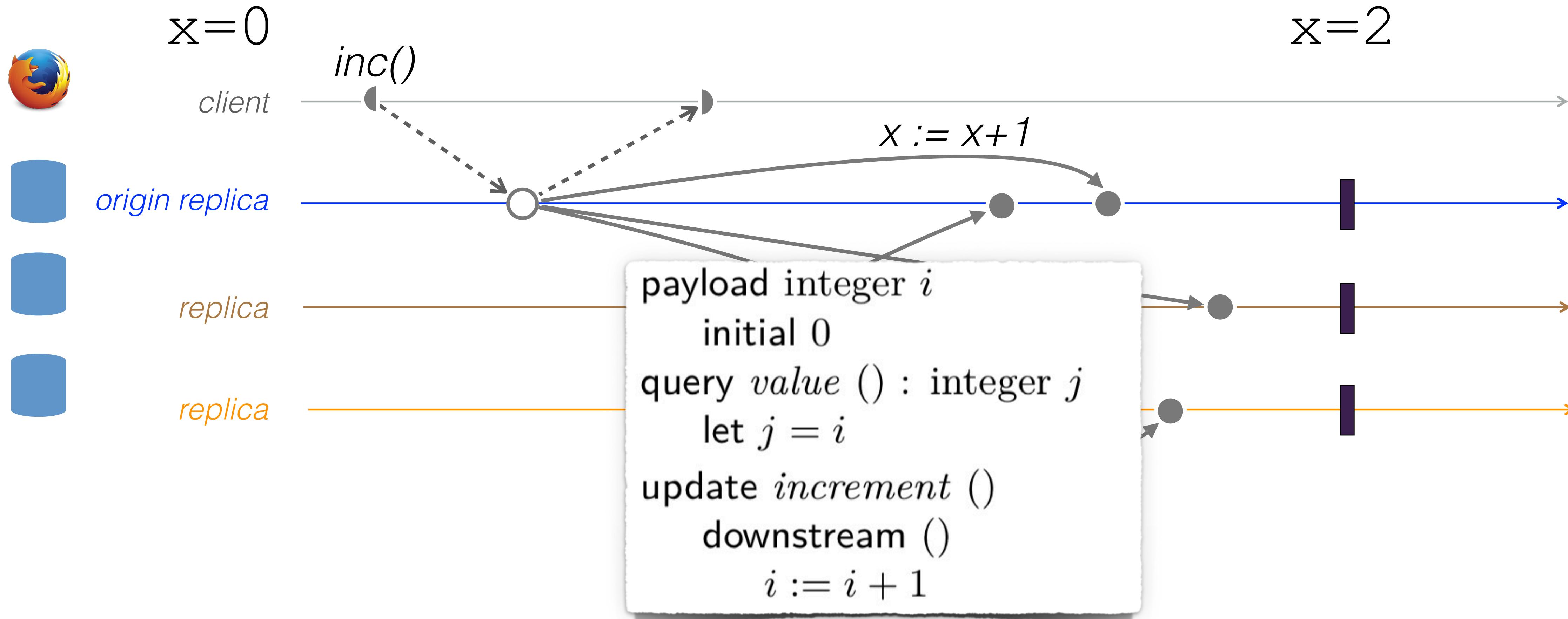
# EXAMPLE: GROW-ONLY COUNTER



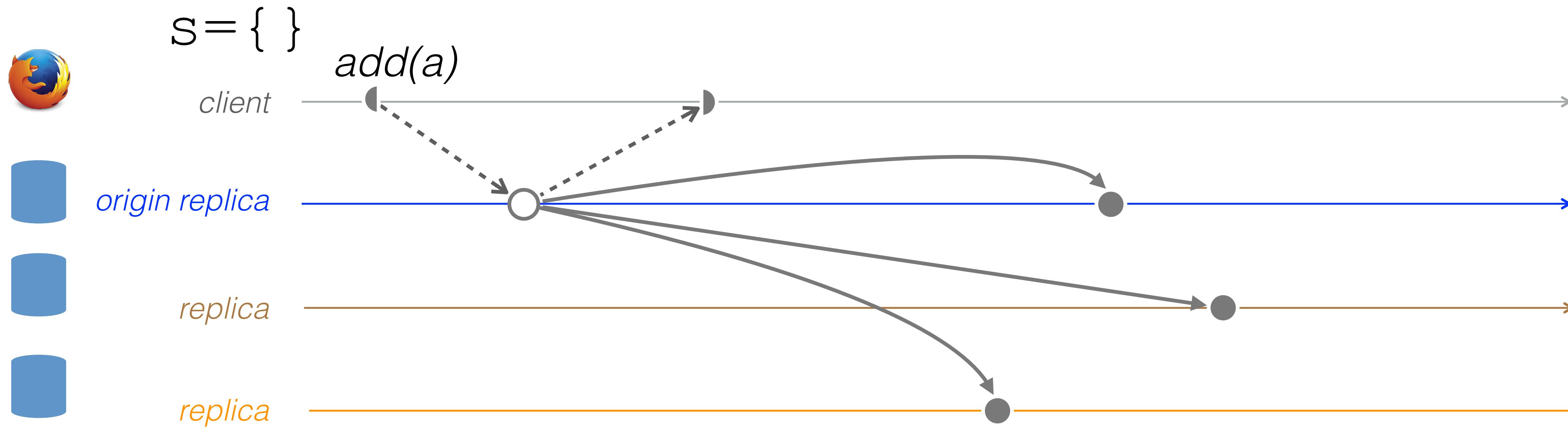
# EXAMPLE: GROW-ONLY COUNTER



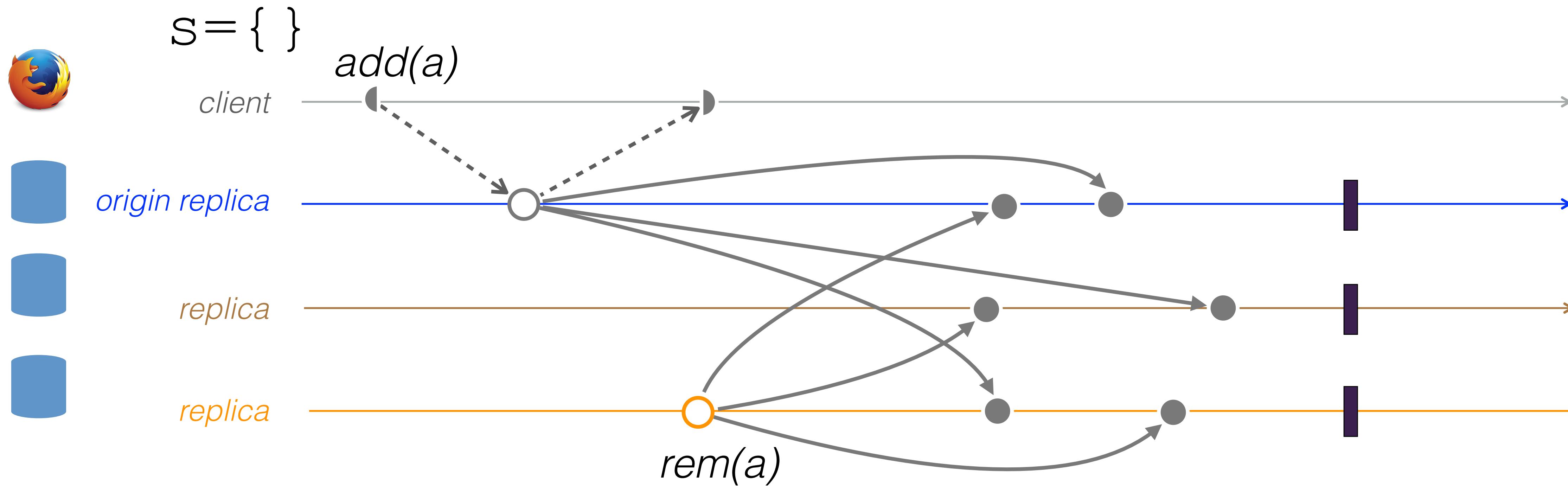
# EXAMPLE: GROW-ONLY COUNTER



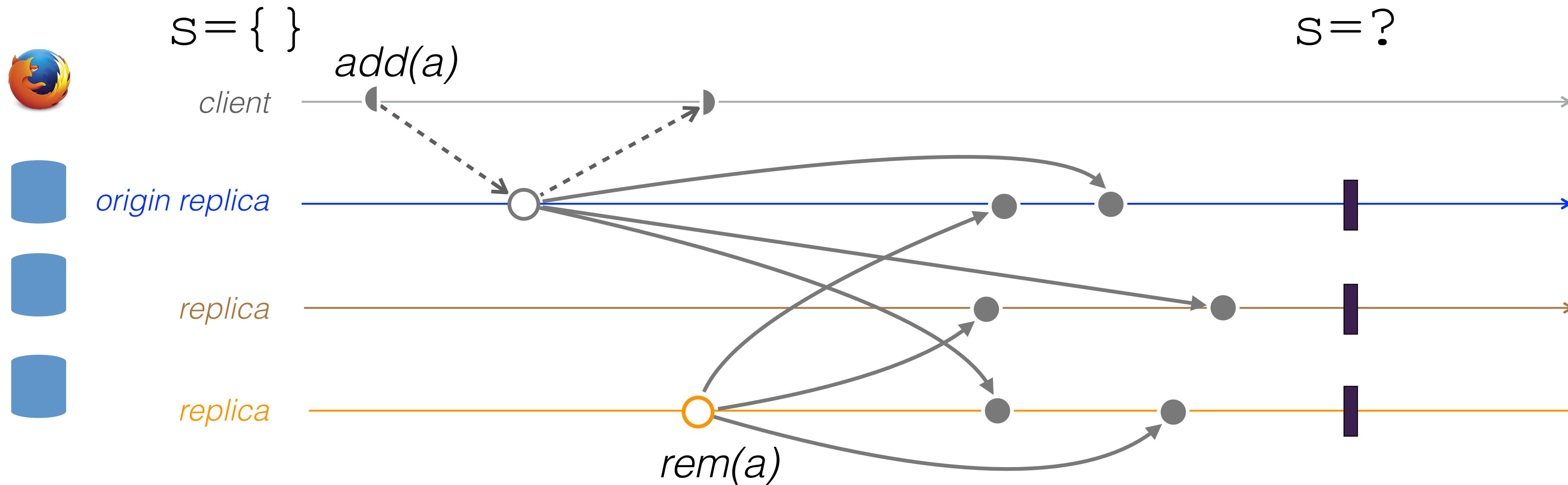
# EXAMPLE: OBSERVED-REMOVE SET



# EXAMPLE: OBSERVED-REMOVE SET



# EXAMPLE: OBSERVED-REMOVE SET



# EXAMPLE: OBSERVED-REMOVE SET



$S = \{ \}$

*client* ————— *add(a)*

payload set  $S$   
initial  $\emptyset$

*origin replica*

query *lookup* (element  $e$ ) : boolean  $b$

let  $b = (\exists u : (e, u) \in S)$

*replica*

update *add* (element  $e$ )

atSource ( $e$ )

let  $\alpha = \text{unique}()$

downstream ( $e, \alpha$ )

$S := S \cup \{(e, \alpha)\}$

*replica*

update *remove* (element  $e$ )

atSource ( $e$ )

pre *lookup*( $e$ )

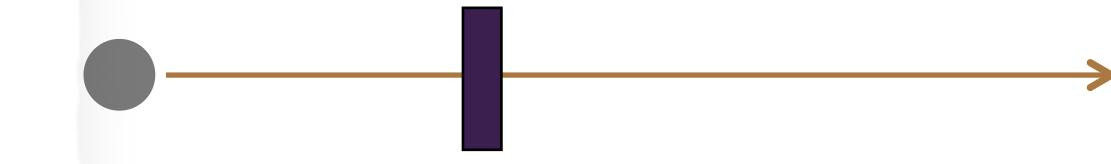
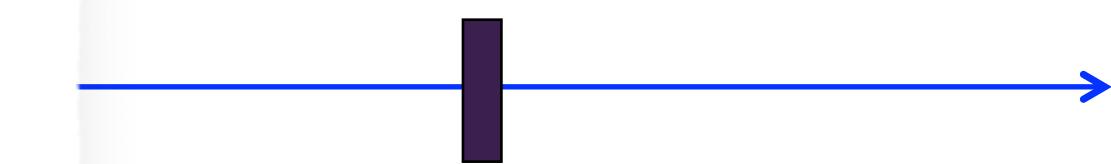
let  $R = \{(e, u) | \exists u : (e, u) \in S\}$

downstream ( $R$ )

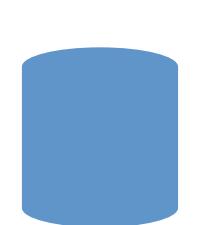
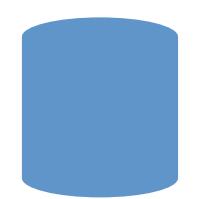
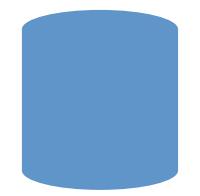
pre  $\forall (e, u) \in R : \text{add}(e, u)$  has been delivered

$S := S \setminus R$

$S = ?$



# EXAMPLE: OBSERVED-REMOVE SET



$S = \{ \}$

*client*

*add(a)*

payload set  $S$

initial  $\emptyset$

query *lookup* (element  $e$ ) : boolean  $b$

let  $b = (\exists u : (e, u) \in S)$

update *add* (element  $e$ )

atSource ( $e$ )

let  $\alpha = unique()$

downstream ( $e, \alpha$ )

$S := S \cup \{(e, \alpha)\}$

update *remove* (element  $e$ )

atSource ( $e$ )

pre *lookup*( $e$ )

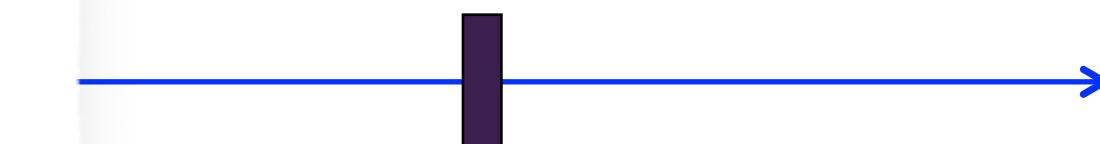
let  $R = \{(e, u) | \exists u : (e, u) \in S\}$

downstream ( $R$ )

pre  $\forall (e, u) \in R : add(e, u)$  has been delivered

$S := S \setminus R$

$S = \{ a \}$



# ANOMALIES OF CONCURRENT UPDATES

- ▶ Bank:
  - ▶  $\sigma_{\text{init}} = 100\text{€}$
  - ▶ Alice:  $\text{withdraw}(20) = \{ \sigma := 120 \}$
  - ▶ Bob:  $\text{debit } (60) = \{ \sigma := 40 \}$
  - ▶  $\sigma = ???$

# ANOMALIES OF CONCURRENT UPDATES

- ▶ Bank:
  - ▶  $\sigma_{\text{init}} = 100\text{\euro}$
  - ▶ Alice:  $\text{withdraw}(20) = \{ \sigma := 120 \}$
  - ▶ Bob:  $\text{debit}(60) = \{ \sigma := 40 \}$
  - ▶  $\sigma = ???$

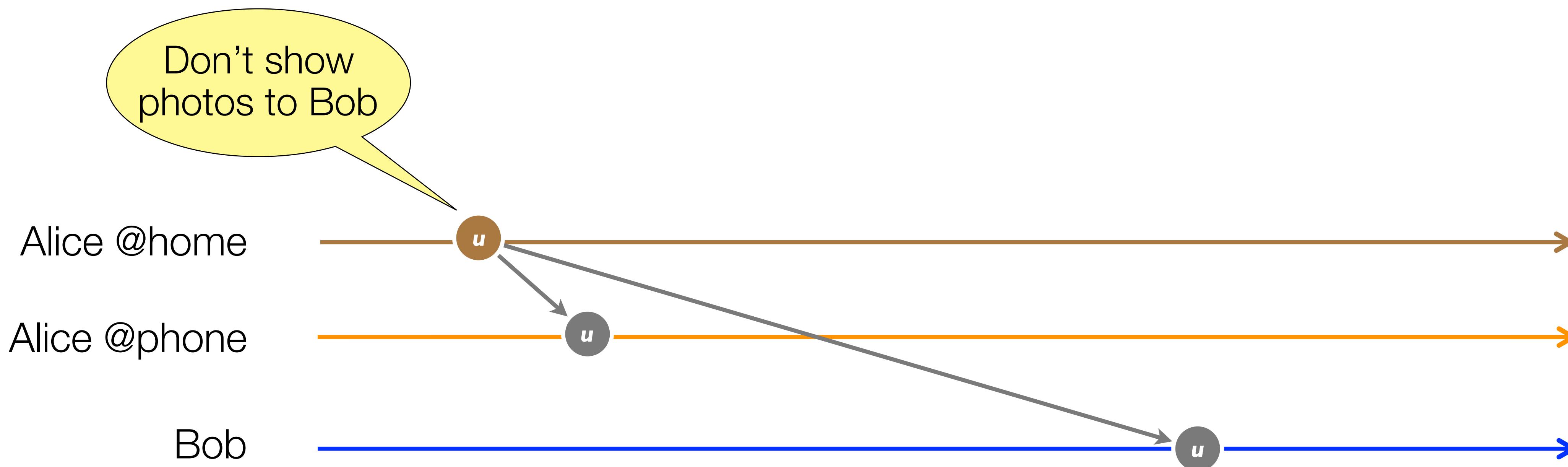
- ▶ File system:
  - ▶  $\sigma_{\text{init}} = "/$
  - ▶ Alice:  $\text{mkdir}("/\text{foo}"); \text{mkdir}("/\text{foo}/\text{bar}")$
  - ▶ Bob: receives  $\text{mkdir}("/\text{foo}/\text{bar}")$
  - ▶  $\sigma = ???$

# EVENTUAL CONSISTENCY



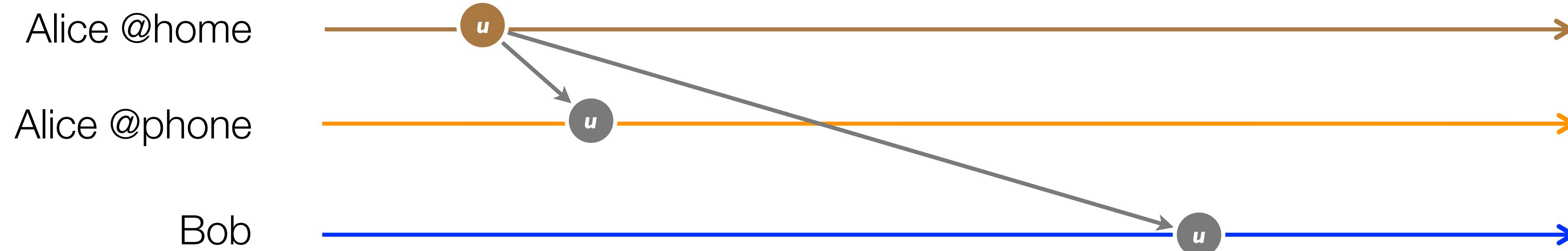
- ▶ access (Bob, photo)  $\implies$  ACL (Bob, photo)
- ▶ v observed effects of u  $\implies$  v should be delivered after u
- ▶ Available: doesn't slow down sender

# EVENTUAL CONSISTENCY



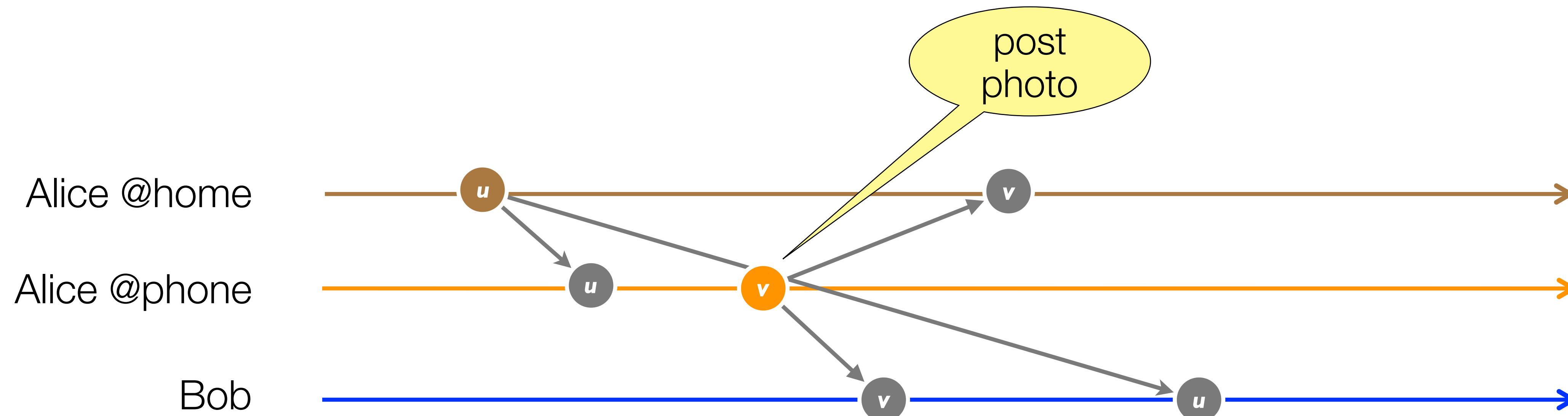
- ▶ access (Bob, photo)  $\implies$  ACL (Bob, photo)
- ▶ v observed effects of u  $\implies$  v should be delivered after u
- ▶ Available: doesn't slow down sender

# EVENTUAL CONSISTENCY



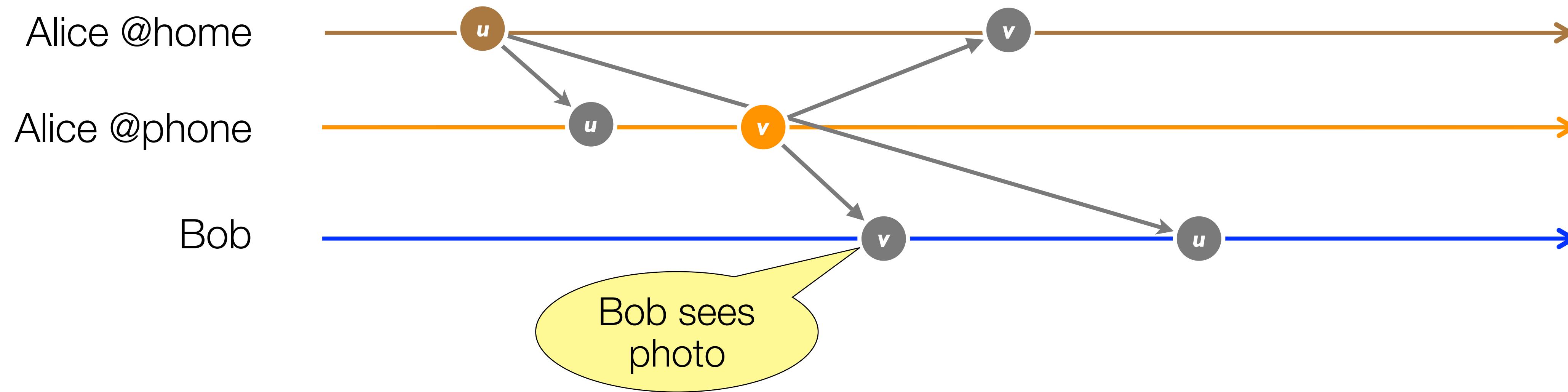
- ▶ access (Bob, photo)  $\Rightarrow$  ACL (Bob, photo)
- ▶  $v$  observed effects of  $u \Rightarrow v$  should be delivered after  $u$
- ▶ Available: doesn't slow down sender

# EVENTUAL CONSISTENCY



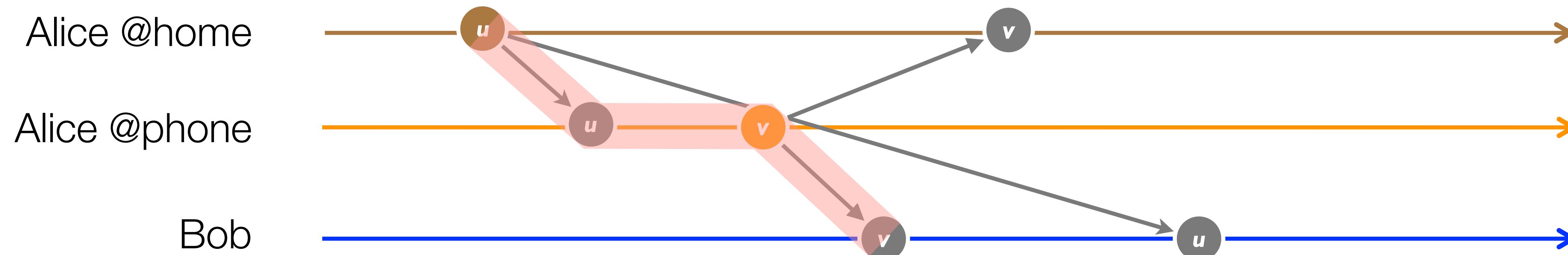
- ▶ access (Bob, photo)  $\Rightarrow$  ACL (Bob, photo)
- ▶ v observed effects of u  $\Rightarrow$  v should be delivered after u
- ▶ Available: doesn't slow down sender

# EVENTUAL CONSISTENCY



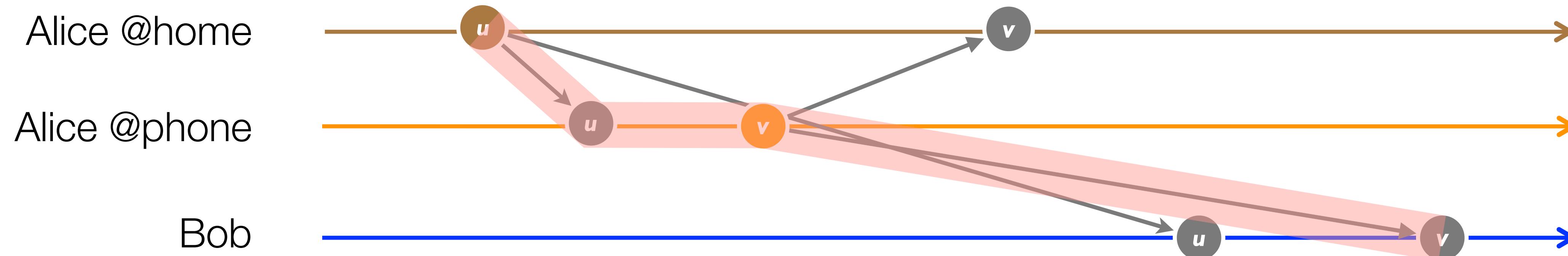
- ▶ access (Bob, photo)  $\Rightarrow$  ACL (Bob, photo)
- ▶  $v$  observed effects of  $u \Rightarrow v$  should be delivered after  $u$
- ▶ Available: doesn't slow down sender

# EVENTUAL CONSISTENCY



- ▶ access (Bob, photo)  $\implies$  ACL (Bob, photo)
- ▶ v observed effects of u  $\implies$  v should be delivered after u
- ▶ Available: doesn't slow down sender

# CAUSAL CONSISTENCY



- ▶ access (Bob, photo)  $\implies$  ACL (Bob, photo)
- ▶  $v$  observed effects of  $u$   $\implies$   $v$  should be delivered after  $u$
- ▶ Available: doesn't slow down sender

# COMMUTATIVE REPLICATED DATA TYPES

# COMMUTATIVE REPLICATED DATA TYPES



*A comprehensive study of  
Convergent and Commutative Replicated Data Types* [\*]

Marc Shapiro, INRIA & LIP6, Paris, France  
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal  
Carlos Baquero, Universidade do Minho, Portugal  
Marek Zawirski, INRIA & UPMC, Paris, France

Thème COM — Systèmes communicants  
Projet Regal

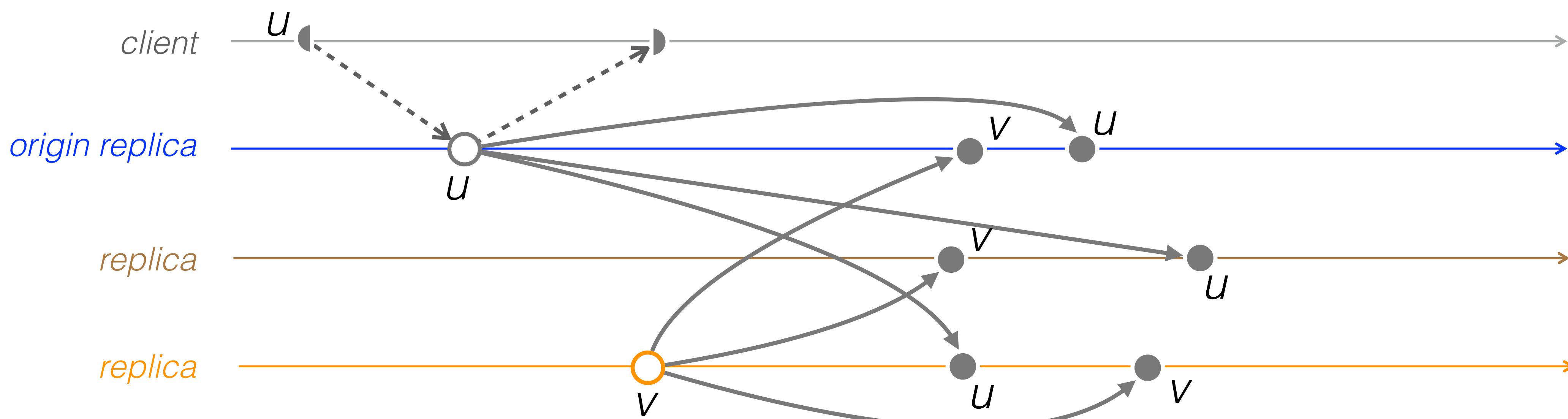
<http://bit.ly/1PBC4zc>

# OPERATION-BASED CRDTs

- ▶ Operation-based CRDTs
- ▶ Each operation is delivered to each replica

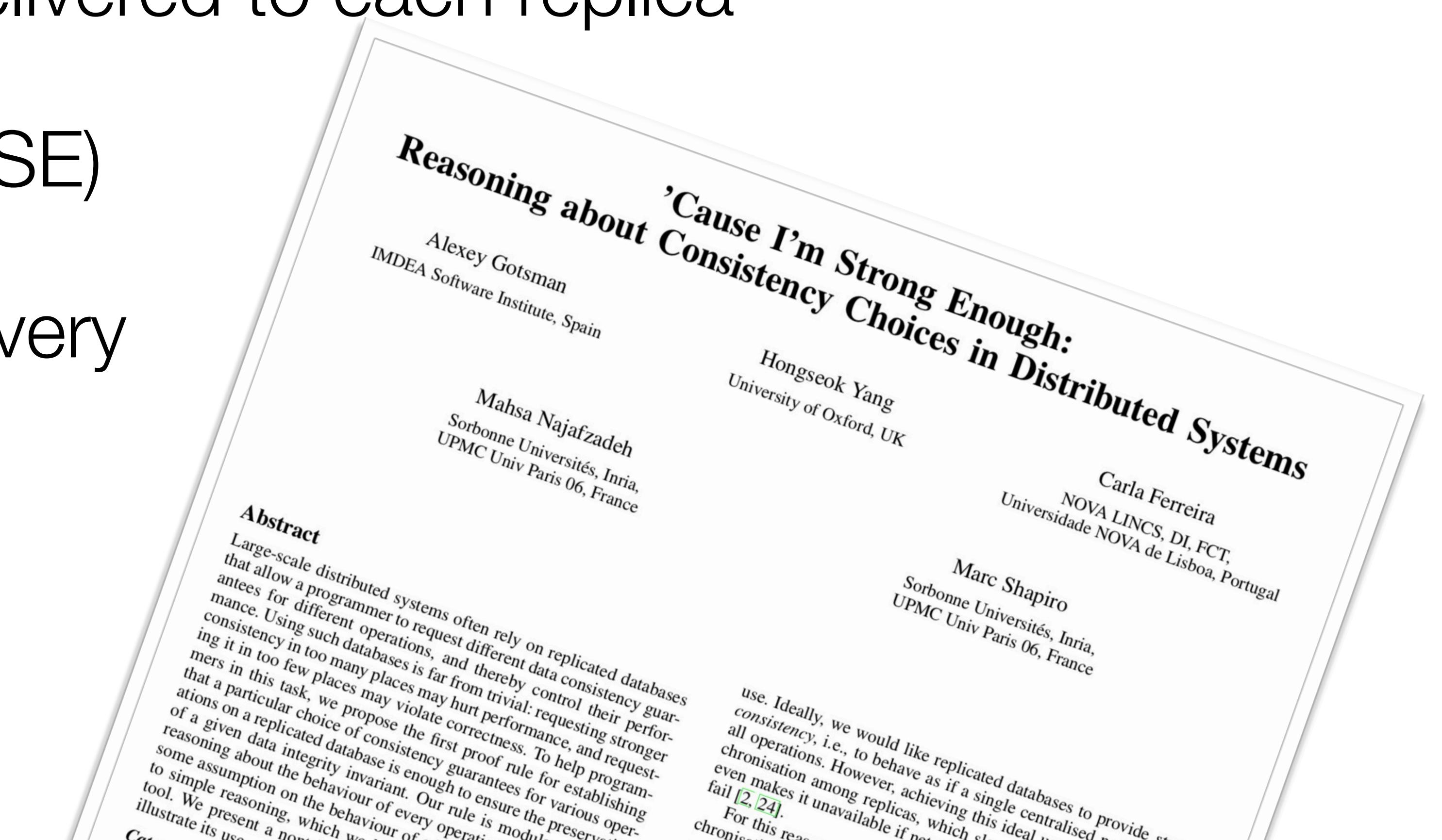
# OPERATION-BASED CRDTs

- ▶ Operation-based CRDTs
- ▶ Each operation is delivered to each replica



# OPERATION-BASED CRDTs

- ▶ Operation-based CRDTs
- ▶ Each operation is delivered to each replica
- ▶ Invariant Checking (CISE)
- ▶ Requires causal delivery



# LAST WRITER WINS REGISTER

---

## Specification 9 Op-based LWW-Register

---

payload  $X$   $x$ , timestamp  $t$   
initial  $\perp, 0$   
query  $value()$  :  $X$   $w$   
    let  $w = x$   
update  $assign(X x')$   
    atSource ()  $t'$   
        let  $t' = now()$   
        downstream  $(x', t')$   
            **if**  $t < t'$  **then**  $x, t := x', t'$

---

# SETS

---

## Specification 13 U-Set: Op-based 2P-Set with unique

---

- 1: payload set  $S$
- 2: initial  $\emptyset$
- 3: query *lookup* (element  $e$ ) : boolean  $b$
- 4: let  $b = (e \in S)$
- 5: update *add* (element  $e$ )  
atSource ( $e$ )  
pre  $e$  is unique  
downstream ( $e$ )  
 $S := S \cup \{e\}$
- 10: update *remove* (element  $e$ )  
atSource ( $e$ )  
pre *lookup*( $e$ )  
downstream ( $e$ )  
pre *add*( $e$ ) has been delivered  
 $S := S \setminus \{e\}$

---

# SETS

---

## Specification 13 U-Set: Op-based 2P-Set with unique

---

```

1: payload set  $S$ 
2: initial  $\emptyset$ 
3: query lookup (element  $e$ ) : boolean  $b$ 
4: let  $b = (e \in S)$ 
5: update add (element  $e$ )
   atSource ( $e$ )
   pre  $e$  is unique
   downstream ( $e$ )
    $S := S \cup \{e\}$ 
10: update remove (element  $e$ )
    atSource ( $e$ )
    pre lookup( $e$ )
    downstream ( $e$ )
    pre add( $e$ ) has been delivered
     $S := S \setminus \{e\}$ 

```

---



---

## Specification 15 Op-based Observed-Remove Set (OR-Set)

---

```

1: payload set  $S$  ▷ set of pairs { $(e, u)$ }
2: initial  $\emptyset$ 
3: query lookup (element  $e$ ) : boolean  $b$ 
4: let  $b = (\exists u : (e, u) \in S)$ 
5: update add (element  $e$ )
6: atSource ( $e$ )
7: let  $\alpha = \text{unique}()$  ▷
8: downstream ( $e, \alpha$ )
9:  $S := S \cup \{(e, \alpha)\}$ 
10: update remove (element  $e$ )
11: atSource ( $e$ )
12: pre lookup( $e$ )
13: let  $R = \{(e, u) | \exists u : (e, u) \in S\}$ 
14: downstream ( $R$ )
15: pre  $\forall (e, u) \in R : \text{add}(e, u)$  has been delivered ▷ U-Set p
16:  $S := S \setminus R$  ▷ Downstream:

```

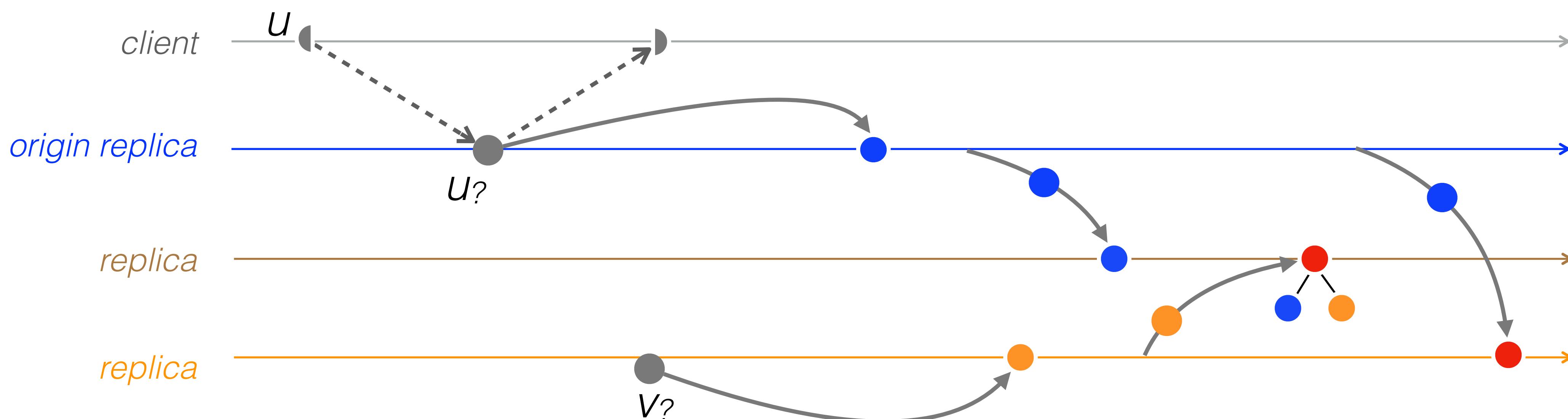
---

# STATE-BASED CRDTs

- ▶ State-based CRDTs
- ▶ Propagation of states (instead of operations)

# STATE-BASED CRDTs

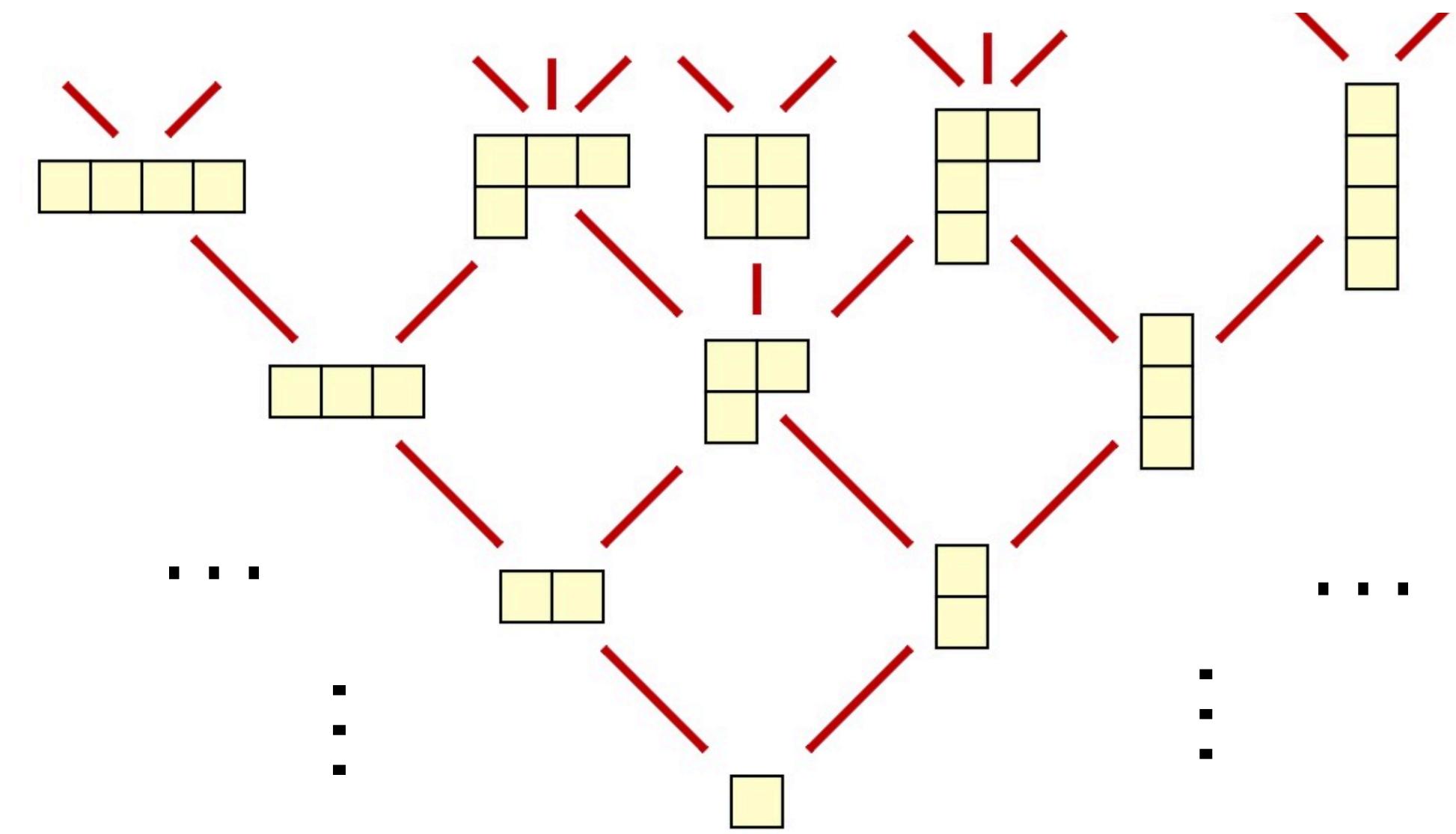
- ▶ State-based CRDTs
  - ▶ Propagation of states (instead of operations)



# STATE-BASED CRDTs

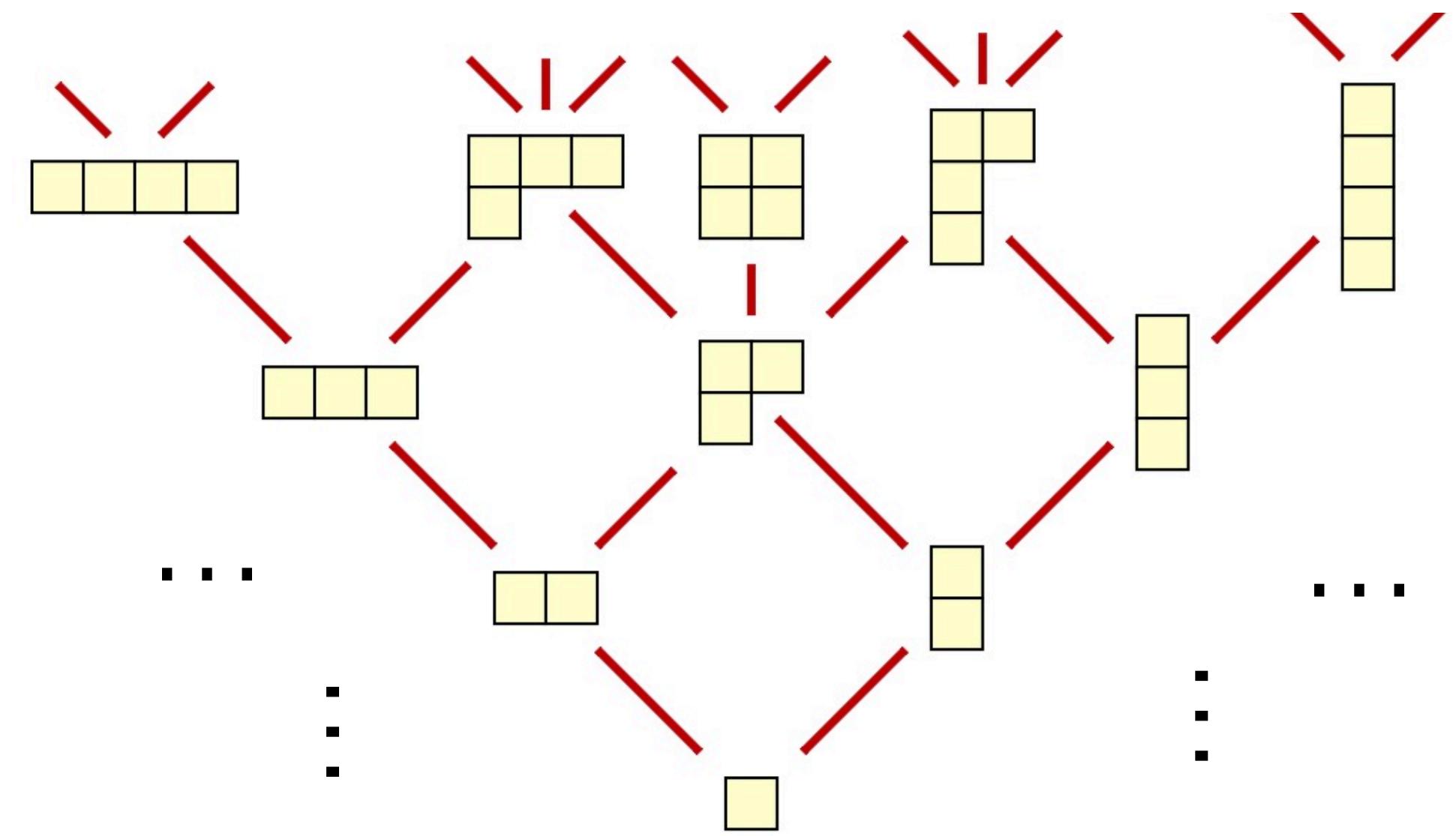
- ▶ State-based CRDTs
  - ▶ Propagation of states (instead of operations)
  - ▶ States are **merged** on receive
    - ▶ Convergence: states resulting from *concurrent* operations result deterministically on a single state
  - ▶ No delivery assumptions

# STATE-BASED CRDTs



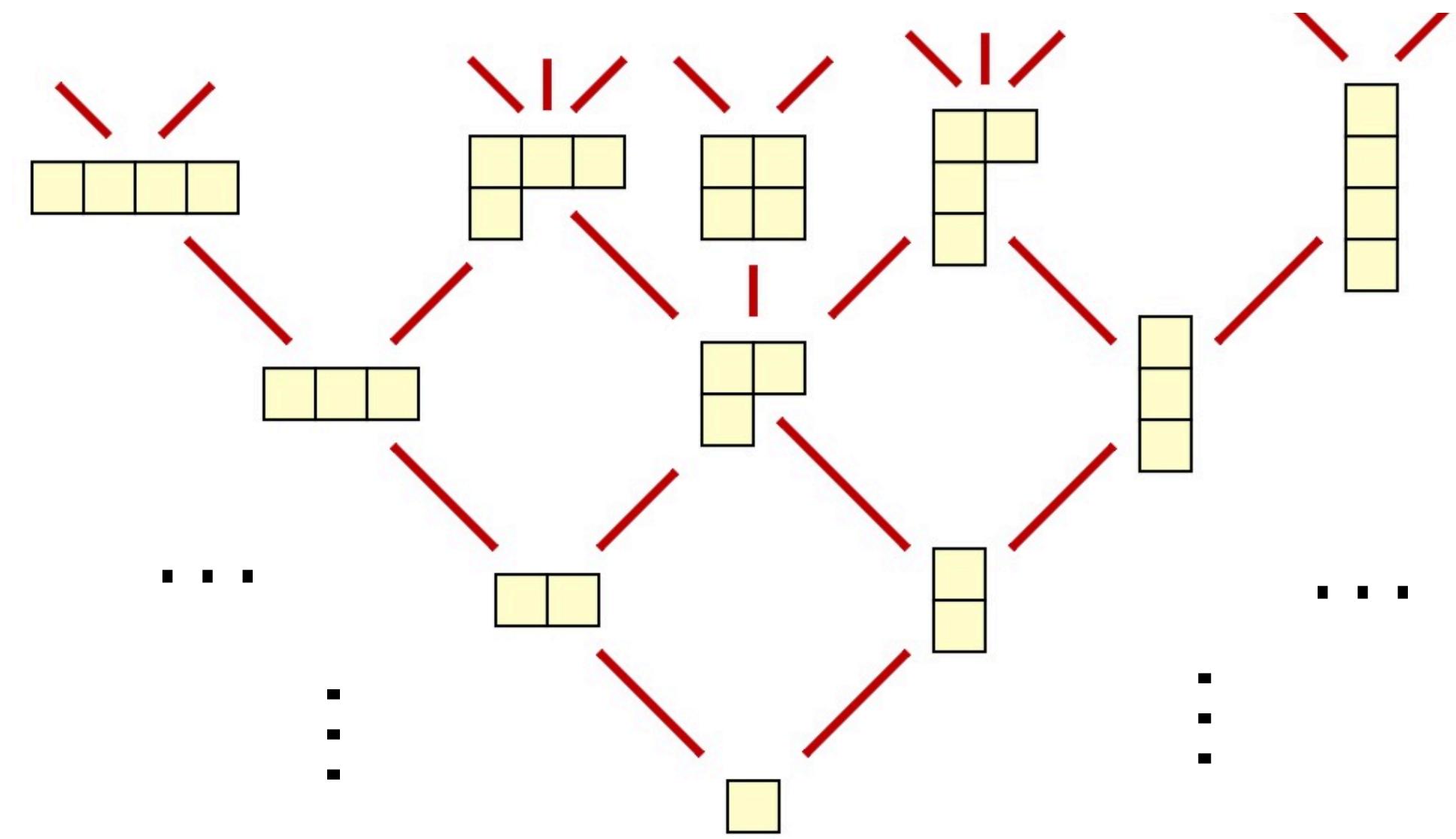
# STATE-BASED CRDTs

- ▶ State is a (join semi-)Lattice



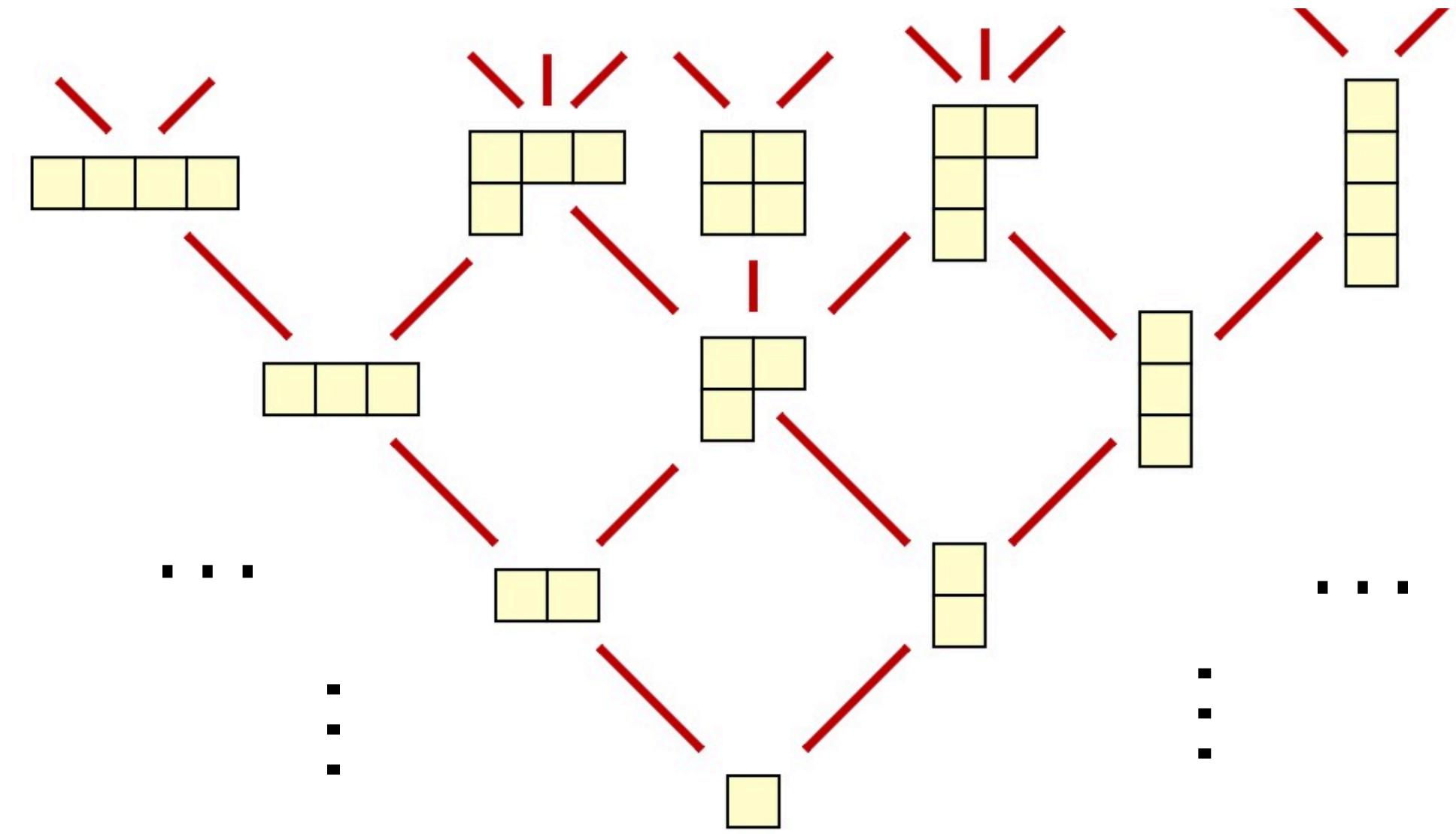
# STATE-BASED CRDTs

- ▶ State is a (join semi-)Lattice
- ▶ Effectors send the state at the origin
- ▶ Lazy update propagation



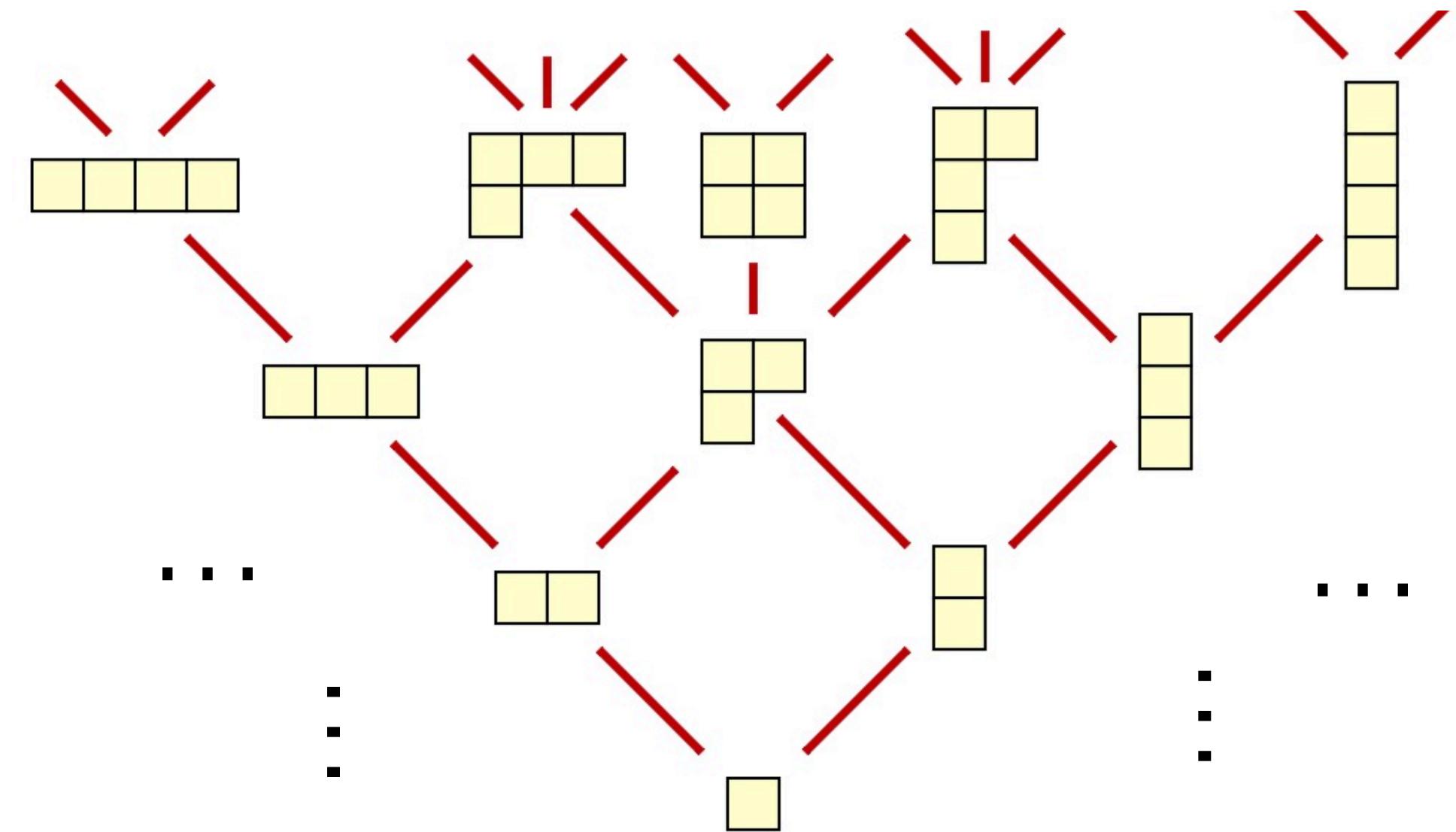
# STATE-BASED CRDTs

- ▶ State is a (join semi-)Lattice
- ▶ Effectors send the state at the origin
- ▶ Lazy update propagation
- ▶ **merge** function joins the state of two replicas
- ▶ Join of the lattice



# STATE-BASED CRDTs

- ▶ State is a (join semi-)Lattice
- ▶ Effectors send the state at the origin
- ▶ Lazy update propagation
- ▶ **merge** function joins the state of two replicas
- ▶ Join of the lattice
- ▶ Each operation is an inflation in the lattice



# BOUNDED COUNTER

- ▶ N Replicas
- ▶ R matrix of positive counts
- ▶ U vector of negative counts

10	2	2	1	5
3	6	1	0	2
5	0	4	2	1
1	0	2	1	0

# BOUNDED COUNTER

- ▶ N Replicas
- ▶ R matrix of positive counts
- ▶ U vector of negative counts

10	2	2	1	5
3	6	1	0	2
5	0	4	2	1
1	0	2	1	0

$$\text{Total: } \sum_i R[i][i] - \sum_i U[i]$$

13

# BOUNDED COUNTER

- ▶ N Replicas
- ▶ R matrix of positive counts
- ▶ U vector of negative counts
- ▶ Invariant:  $0 \leq \text{Total}$

10	2	2	1	5
3	6	1	0	2
5	0	4	2	1
1	0	2	1	0

$$\text{Total: } \sum_i R[i][i] - \sum_i U[i]$$

13

# BOUNDED COUNTER

- ▶ N Replicas
- ▶ R matrix of positive counts
- ▶ U vector of negative counts
- ▶ Invariant:  $0 \leq \text{Total}$
- ▶ **increment @i**

i	j			
i	10	2	2	1
j	3	6	1	0
i	5	0	4	2
j	1	0	2	1
				5
				2
				1
				0

$$\text{Total: } \sum_i R[i][i] - \sum_i U[i]$$

13

# BOUNDED COUNTER

- ▶ N Replicas
- ▶ R matrix of positive counts
- ▶ U vector of negative counts
- ▶ Invariant:  $0 \leq \text{Total}$
- ▶ **increment** @*i*
- ▶ **decrement** @*i*

j	i	j	
i	10	2	2
j	3	6	1
i	5	0	4
j	1	0	2
i			1
j			0

$$\text{Total: } \sum_i R[i][i] - \sum_i U[i]$$

# BOUNDED COUNTER

- ▶ N Replicas
- ▶ R matrix of positive counts
- ▶ U vector of negative counts
- ▶ Invariant:  $0 \leq \text{Total}$
- ▶ **increment** @*i*
- ▶ **decrement** @*i*

j	i	j	
i	10	2	2
j	3	6	1
j	5	0	4
j	1	0	2
j			1
j			0

$$\text{Total: } \sum_i R[i][i] - \sum_i U[i]$$

$$\text{Rights } @i: R[i][i] + \sum_j R[j][i] - \sum_{j \neq i} R[i][j] - U[i]$$

# BOUNDED COUNTER

- ▶ N Replicas
- ▶ R matrix of positive counts
- ▶ U vector of negative counts
- ▶ Invariant:  $0 \leq \text{Total}$
- ▶ **increment** @*i*
- ▶ **decrement** @*i*

j	i	j	
i	10	2	2
j	3	6	1
j	5	0	4
j	1	0	2
j			1
j			0

$$\text{Total: } \sum_i R[i][i] - \sum_i U[i]$$

$$\text{Rights } @i: R[i][i] + \sum_j R[j][i] - \sum_{j \neq i} R[i][j] - U[i]$$

@i 2

13

# BOUNDED COUNTER

- ▶ N Replicas
- ▶ R matrix of positive counts
- ▶ U vector of negative counts

▶ Invariant:  $0 \leq \text{Total}$

▶ **increment** @i

▶ **decrement** @i

▶ **transfer** @i → j

i	j			
i	10	2	2	1
j	3	6	1	0
i	5	0	4	2
j	1	0	2	1
	5	2	1	0

$$\text{Total: } \sum_i R[i][i] - \sum_i U[i]$$

$$\text{Rights } @i: R[i][i] + \sum_j R[j][i] - \sum_{j \neq i} R[i][j] - U[i]$$

@i 2

# BOUNDED COUNTER

- ▶ N Replicas
- ▶ R matrix of positive counts
- ▶ U vector of negative counts

▶ Invariant:  $0 \leq \text{Total}$

▶ **increment** @i

▶ **decrement** @i

▶ **transfer** @i → j

i	j		
i	10	2	2
j	3	6	1
i	5	0	4
j	1	0	2
			1
			0

$$\begin{aligned} \text{merge}((M_0, V_0), (M_1, V_1)) = \\ (\max(M_0, M_1), \max(V_0, V_1)) \end{aligned}$$

# STATE-BASED COUNTERS

---

**Specification 6** State-based increment-only counter (vector version)

```
1: payload integer[n] P
2:   initial [0, 0, ..., 0]
3: update increment ()
4:   let  $g = myID()$ 
5:    $P[g] := P[g] + 1$ 
6: query value () : integer  $v$ 
7:   let  $v = \sum_i P[i]$ 
8: compare (X, Y) : boolean  $b$ 
9:   let  $b = (\forall i \in [0, n - 1] : X.P[i] \leq Y.P[i])$ 
10: merge (X, Y) : payload Z
11:   let  $\forall i \in [0, n - 1] : Z.P[i] = \max(X.P[i], Y.P[i])$ 
```

---

---

**Specification 7** State-based PN-Counter

```
1: payload integer[n] P, integer[n] N
2:   initial [0, 0, ..., 0], [0, 0, ..., 0]
3: update increment ()
4:   let  $g = myID()$ 
5:    $P[g] := P[g] + 1$ 
6: update decrement ()
7:   let  $g = myID()$ 
8:    $N[g] := N[g] + 1$ 
9: query value () : integer  $v$ 
10:  let  $v = \sum_i P[i] - \sum_i N[i]$ 
11: compare (X, Y) : boolean  $b$ 
12:  let  $b = (\forall i \in [0, n - 1] : X.P[i] \leq Y.P[i] \wedge \forall i \in [0, n - 1] : X.N[i] \leq Y.N[i])$ 
13: merge (X, Y) : payload Z
14:  let  $\forall i \in [0, n - 1] : Z.P[i] = \max(X.P[i], Y.P[i])$ 
15:  let  $\forall i \in [0, n - 1] : Z.N[i] = \max(X.N[i], Y.N[i])$ 
```

---

▷ One entry per replica  
▷  $g$ : source replica

# STATE-BASED LWW

---

## Specification 8 State-based Last-Writer-Wins Register (LWW-Register)

---

```
1: payload  $X$   $x$ , timestamp  $t$  ▷  $X$ : some type
2: initial  $\perp, 0$ 
3: update assign ( $X$   $w$ )
4:      $x, t := w, \text{now}()$  ▷ Timestamp, consistent with causality
5: query value () :  $X$   $w$ 
6:     let  $w = x$ 
7: compare ( $R, R'$ ) : boolean  $b$ 
8:     let  $b = (R.t \leq R'.t)$ 
9: merge ( $R, R'$ ) : payload  $R''$ 
10:    if  $R.t \leq R'.t$  then  $R''.x, R''.t = R.x, R.t$ 
11:    else  $R''.x, R''.t = R'.x, R'.t$ 
```

---

# MULTI-VALUE REGISTER

---

**Specification 10** State-based Multi-Value Register (MV-Register)

---

payload set  $S$  ▷ set of  $(x, V)$  pairs;  $x \in X$ ;  $V$  its version vector  
initial  $\{(\perp, [0, \dots, 0])\}$   
query  $incVV()$  : integer[n]  $V'$   
let  $g = myID()$   
let  $\mathcal{V} = \{V \mid \exists x : (x, V) \in S\}$   
let  $V' = [\max_{V \in \mathcal{V}}(V[j])]_{j \neq g}$   
let  $V'[g] = \max_{V \in \mathcal{V}}(V[g]) + 1$   
update assign (set  $R$ ) ▷ set of elements of type  $X$   
let  $V = incVV()$   
 $S := R \times \{V\}$   
query  $value()$  : set  $S'$   
let  $S' = S$   
compare  $(A, B)$  : boolean  $b$   
let  $b = (\forall(x, V) \in A, (x', V') \in B : V \leq V')$   
merge  $(A, B)$  : payload  $C$   
let  $A' = \{(x, V) \in A \mid \forall(y, W) \in B : V \parallel W \vee V \geq W\}$   
let  $B' = \{(y, W) \in B \mid \forall(x, V) \in A : W \parallel V \vee W \geq V\}$   
let  $C = A' \cup B'$

---

# STATE BASED SETS

---

## Specification 11 State-based grow-only Set (G-Set)

---

- 1: **payload** set  $A$
  - 2:     **initial**  $\emptyset$
  - 3: **update** *add* (element  $e$ )
  - 4:      $A := A \cup \{e\}$
  - 5: **query** *lookup* (element  $e$ ) : boolean  $b$
  - 6:     let  $b = (e \in A)$
  - 7: **compare** ( $S, T$ ) : boolean  $b$
  - 8:     let  $b = (S.A \subseteq T.A)$
  - 9: **merge** ( $S, T$ ) : payload  $U$
  - 10:    let  $U.A = S.A \cup T.A$
-

# STATE BASED SETS

---

## Specification 11 State-based grow-only Set (G-Set)

```
1: payload set  $A$ 
2: initial  $\emptyset$ 
3: update add (element  $e$ )
4:  $A := A \cup \{e\}$ 
5: query  $lookup$  (element  $e$ ) : boolean  $b$ 
6: let  $b = (e \in A)$ 
7: compare ( $S, T$ ) : boolean  $b$ 
8: let  $b = (S.A \subseteq T.A)$ 
9: merge ( $S, T$ ) : payload  $U$ 
10: let  $U.A = S.A \cup T.A$ 
```

---

---

## Specification 12 State-based 2P-Set

```
1: payload set  $A$ , set  $R$ 
2: initial  $\emptyset, \emptyset$ 
3: query  $lookup$  (element  $e$ ) : boolean  $b$ 
4: let  $b = (e \in A \wedge e \notin R)$ 
5: update add (element  $e$ )
6:  $A := A \cup \{e\}$ 
7: update remove (element  $e$ )
8: pre  $lookup(e)$ 
9:  $R := R \cup \{e\}$ 
10: compare ( $S, T$ ) : boolean  $b$ 
11: let  $b = (S.A \subseteq T.A \vee S.R \subseteq T.R)$ 
12: merge ( $S, T$ ) : payload  $U$ 
13: let  $U.A = S.A \cup T.A$ 
14: let  $U.R = S.R \cup T.R$ 
```

---

# AKKA CRDTs

# AKKA CRDTs

- ▶ Akka implements some CRDTs: akka-distributed-data
- ▶ **Counters:** GCounter, PNCounter
- ▶ **Sets:** GSet, ORSet
- ▶ **Maps:** ORMap, ORMultiMap, LWWMap, PNCounterMap
- ▶ **Registers:** LWWRegister, Flag

# AKKA CRDTs

- ▶ Akka implements some CRDTs: akka-distributed-data
  - ▶ **Counters:** GCounter, PNCounter
  - ▶ **Sets:** GSet, ORSet
  - ▶ **Maps:** ORMap, ORMultiMap, LWWMap, PNCounterMap
  - ▶ **Registers:** LWWRegister, Flag
- ▶ You are encouraged to learn about them and use them!
- ▶ Adding new types could be a great project!

# AKKA CRDTs

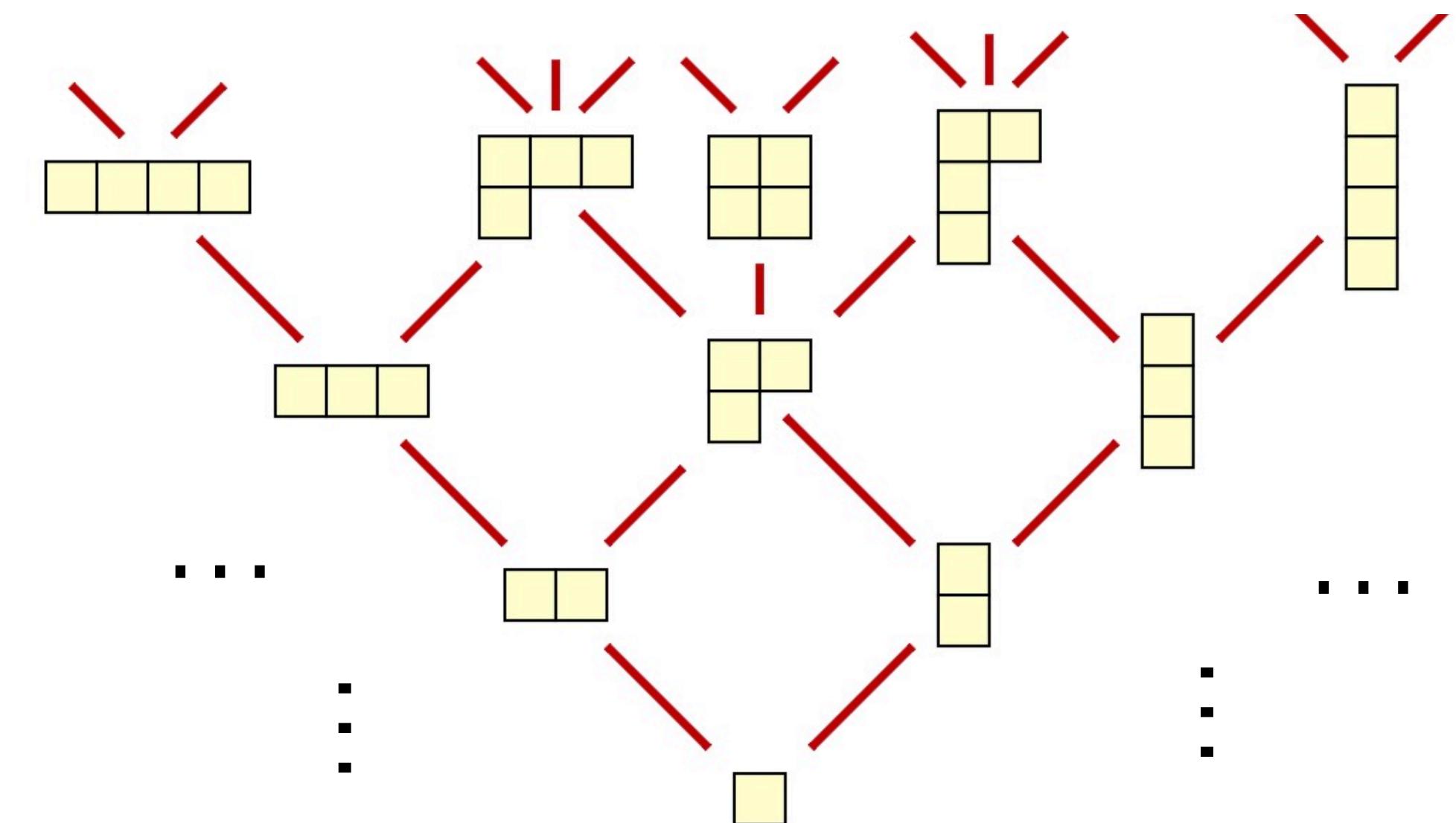
- ▶ Akka implements some CRDTs: akka-distributed-data
  - ▶ **Counters:** GCounter, PNCounter
  - ▶ **Sets:** GSet, ORSet
  - ▶ **Maps:** ORMap, ORMultiMap, LWWMap, PNCounterMap
  - ▶ **Registers:** LWWRegister, Flag
- ▶ You are encouraged to learn about them and use them!
  - ▶ Adding new types could be a great project!
  - ▶ SubscriptionManager.scala

# CHECKING INVARIANTS

## STATE-BASED CRDTs

# INVARIANTS FOR SB-CRDTs

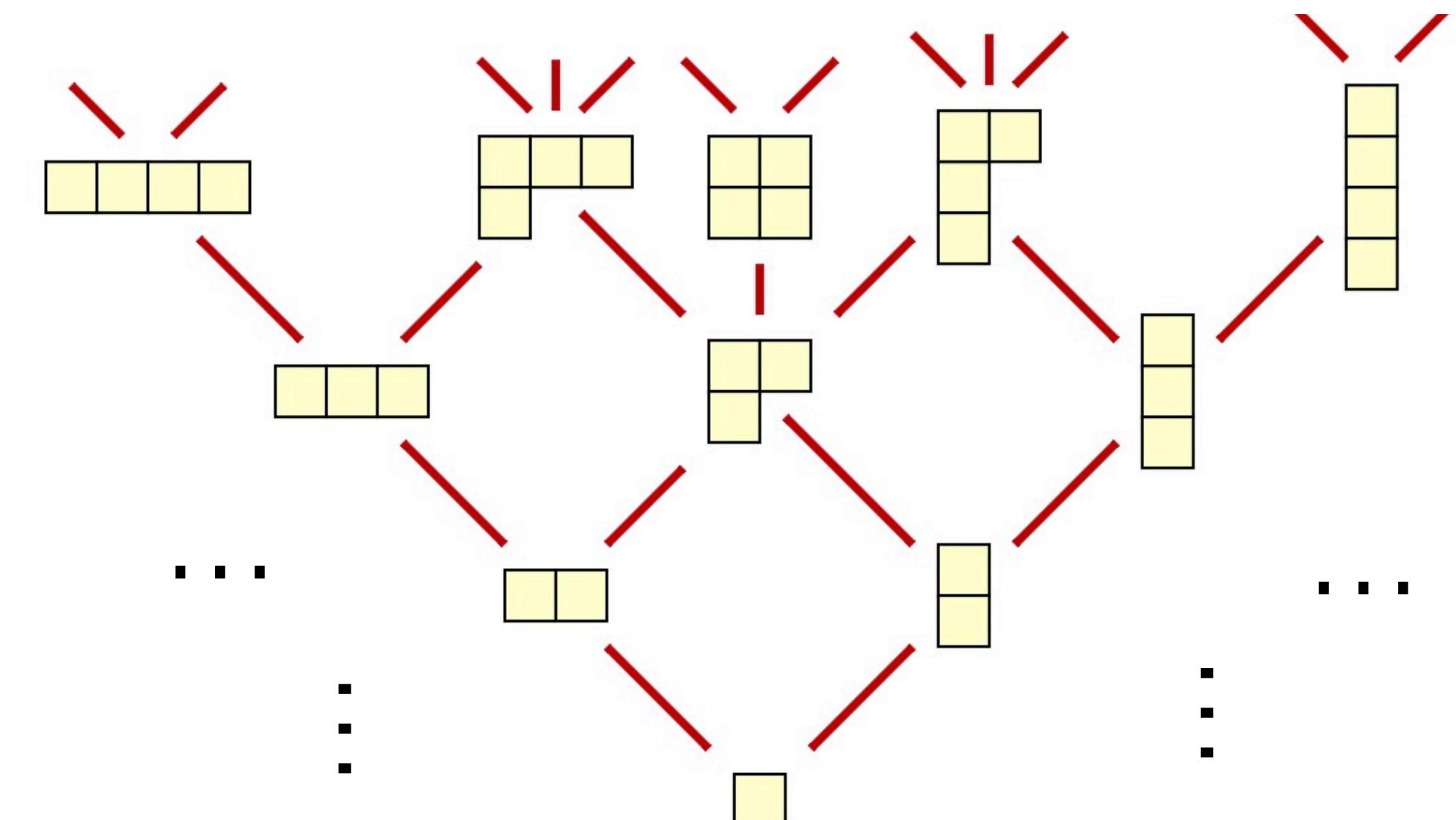
- ▶ CRDT (lattice) constraints



# INVARIANTS FOR SB-CRDTs

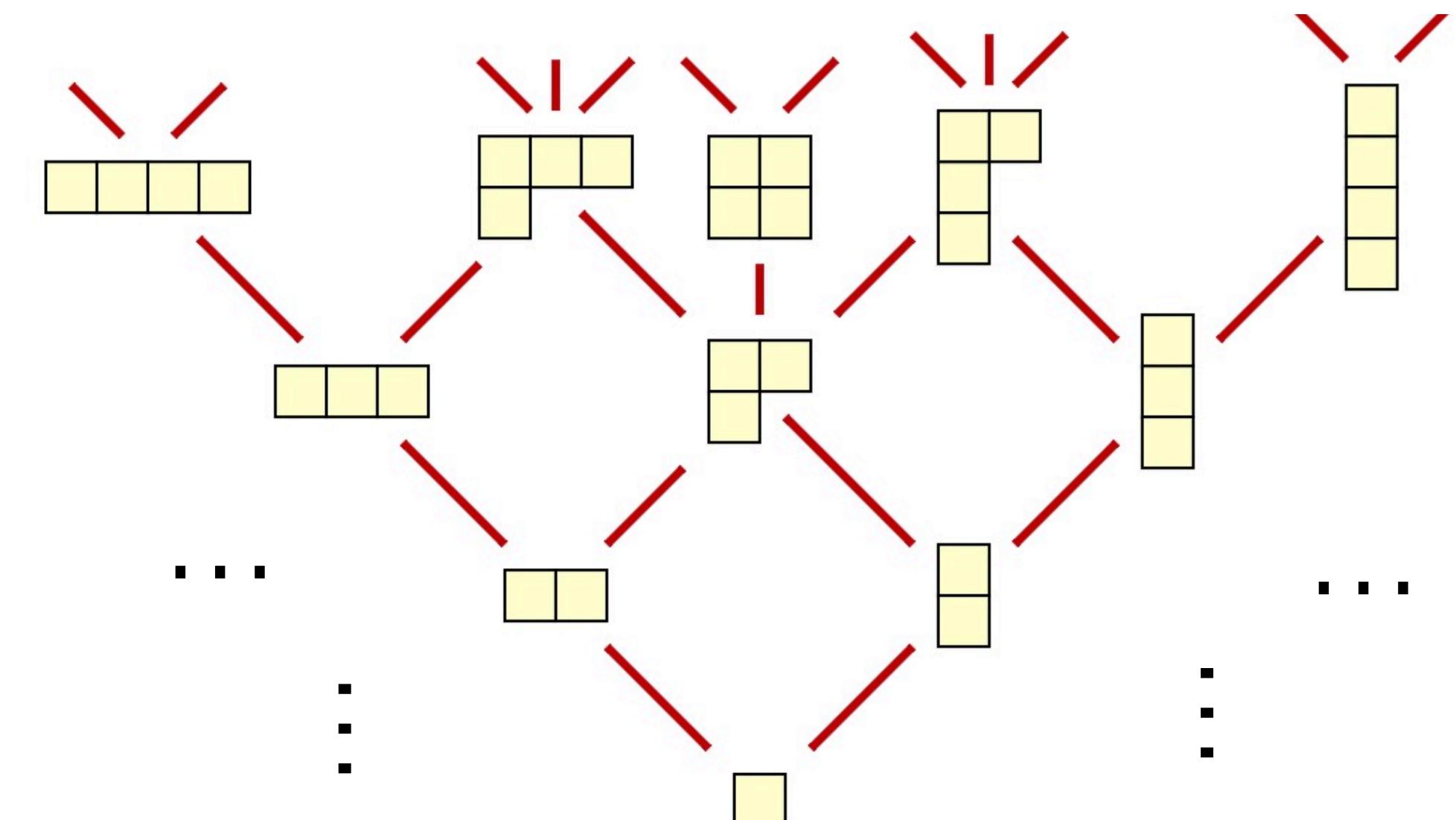
- ▶ CRDT (lattice) constraints
- ▶ Operations are inflations

$$\forall \text{op}, \sigma, \sigma', \sigma \models \text{Pre}_{\text{op}} \wedge (\sigma, \sigma') \in \llbracket \text{op} \rrbracket \Rightarrow \sigma \sqsubseteq \sigma'$$



# INVARIANTS FOR SB-CRDTs

- ▶ CRDT (lattice) constraints



- ▶ Operations are inflations

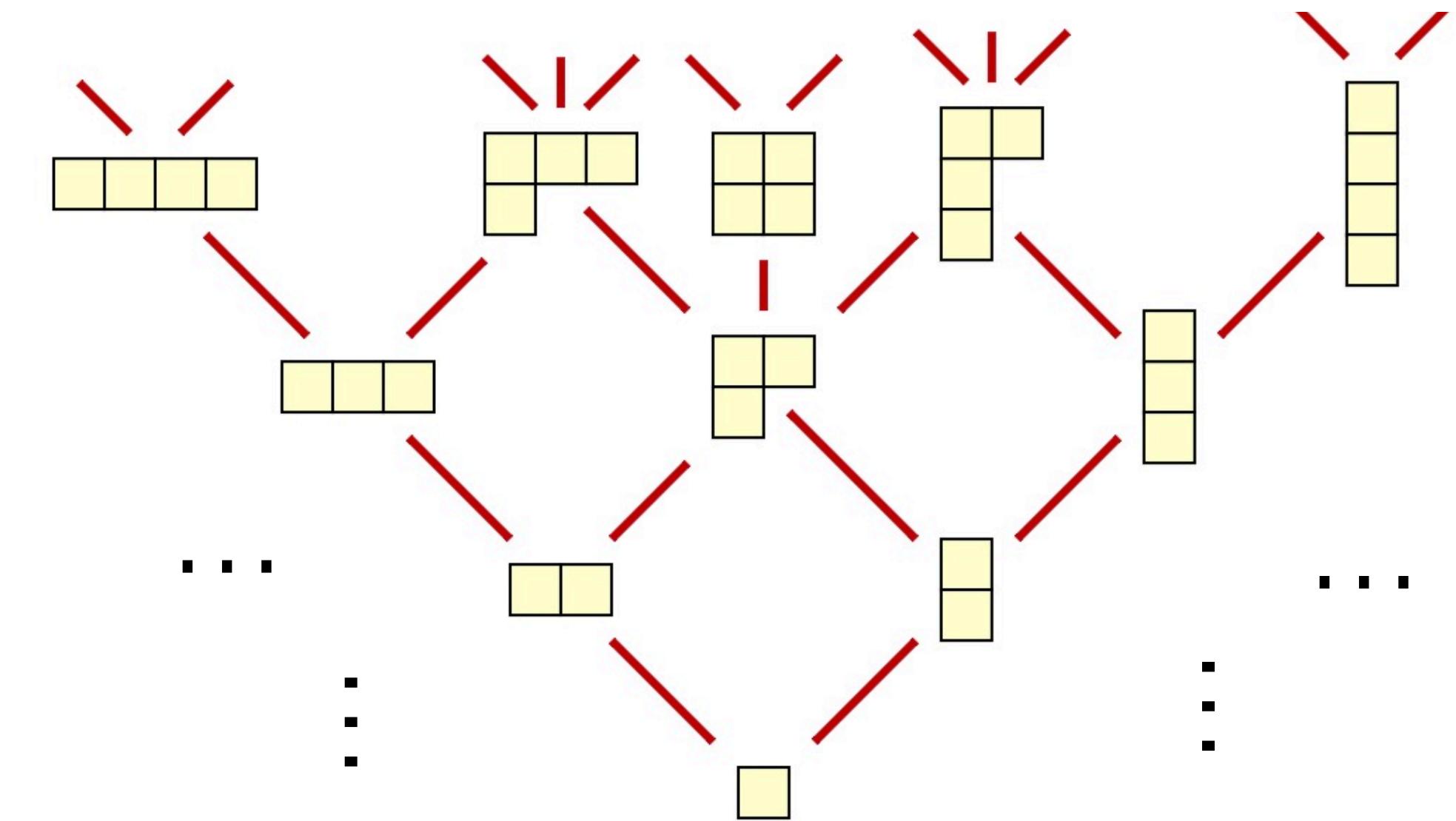
$$\forall \text{op}, \sigma, \sigma', \sigma \models \text{Pre}_{\text{op}} \wedge (\sigma, \sigma') \in \llbracket \text{op} \rrbracket \Rightarrow \sigma \sqsubseteq \sigma'$$

- ▶ **merge** is join (LUB)

$$\forall \sigma, \sigma', \text{merge}(\sigma, \sigma') = \sigma'' \Rightarrow \sigma'' = \text{LUB}_{\sqsubseteq}(\sigma, \sigma')$$

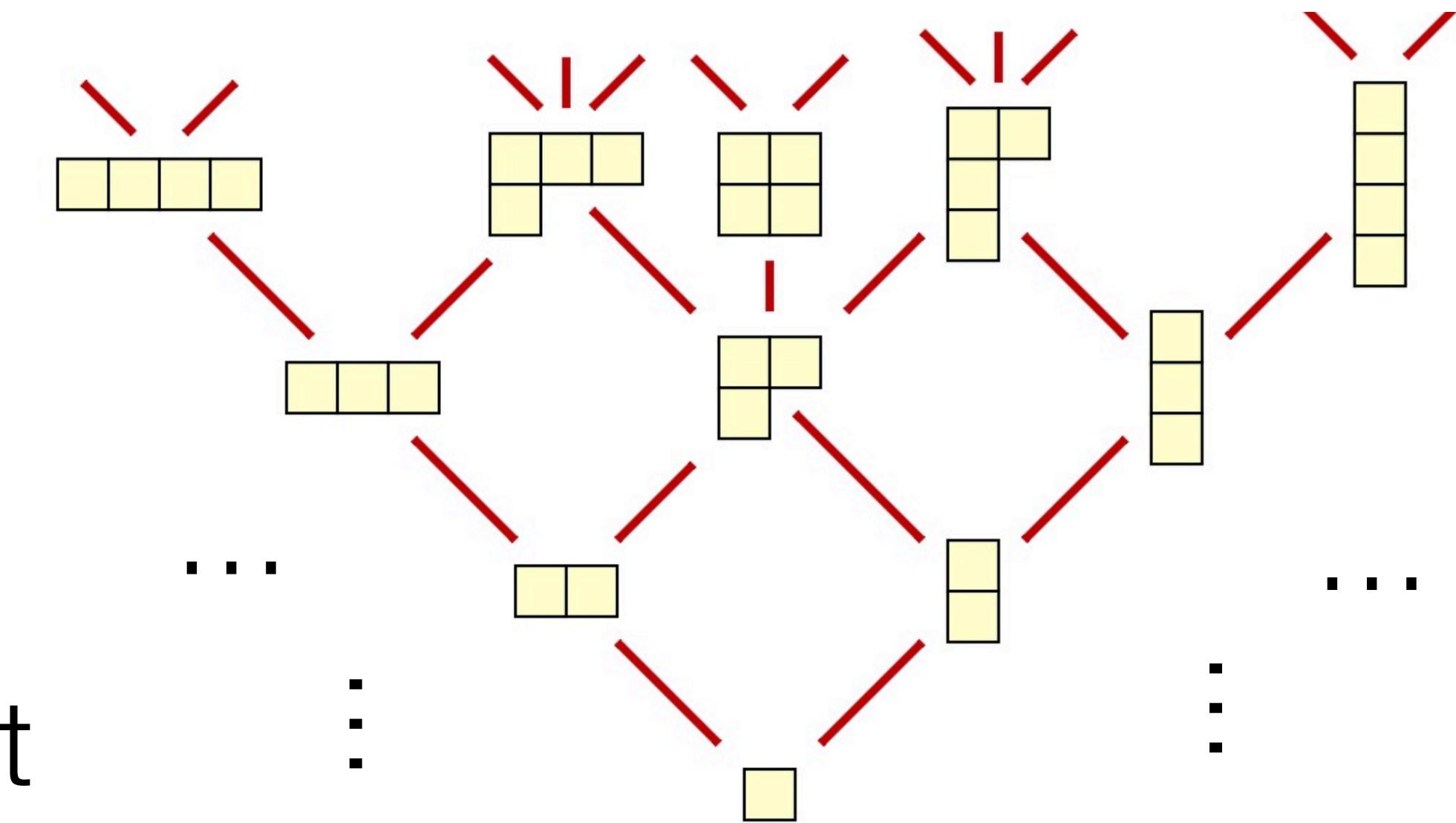
# INVARIANTS FOR SB-CRDTs

- Invariant constraints



# INVARIANTS FOR SB-CRDTs

- ▶ Invariant constraints
  - ▶ Operations preserve the invariant
- $$\forall \text{op}, \sigma, \sigma', \sigma \models \text{Pre}_{\text{op}} \wedge (\sigma, \sigma') \in [\![\text{op}]\!] \Rightarrow \sigma' \models \text{Inv}$$



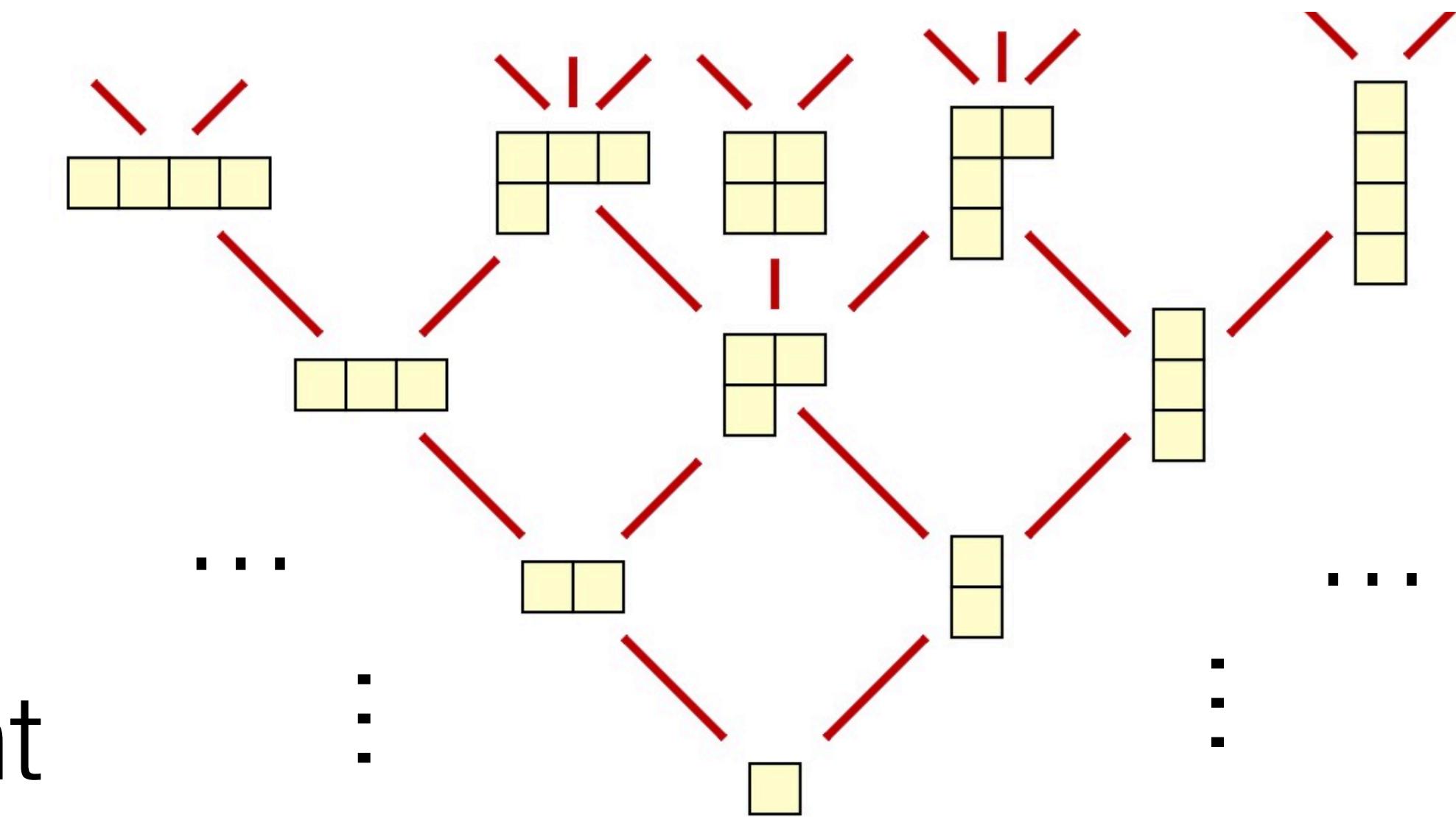
# INVARIANTS FOR SB-CRDTs

- ▶ Invariant constraints
- ▶ Operations preserve the invariant

$$\forall \text{op}, \sigma, \sigma', \sigma \models \text{Pre}_{\text{op}} \wedge (\sigma, \sigma') \in [\text{op}] \Rightarrow \sigma' \models \text{Inv}$$

- ▶ **merge** preserves the invariant

$$\forall \sigma, \sigma', \sigma'', \text{merge}(\sigma, \sigma'') = \sigma' \Rightarrow \sigma' \models \text{Inv}$$



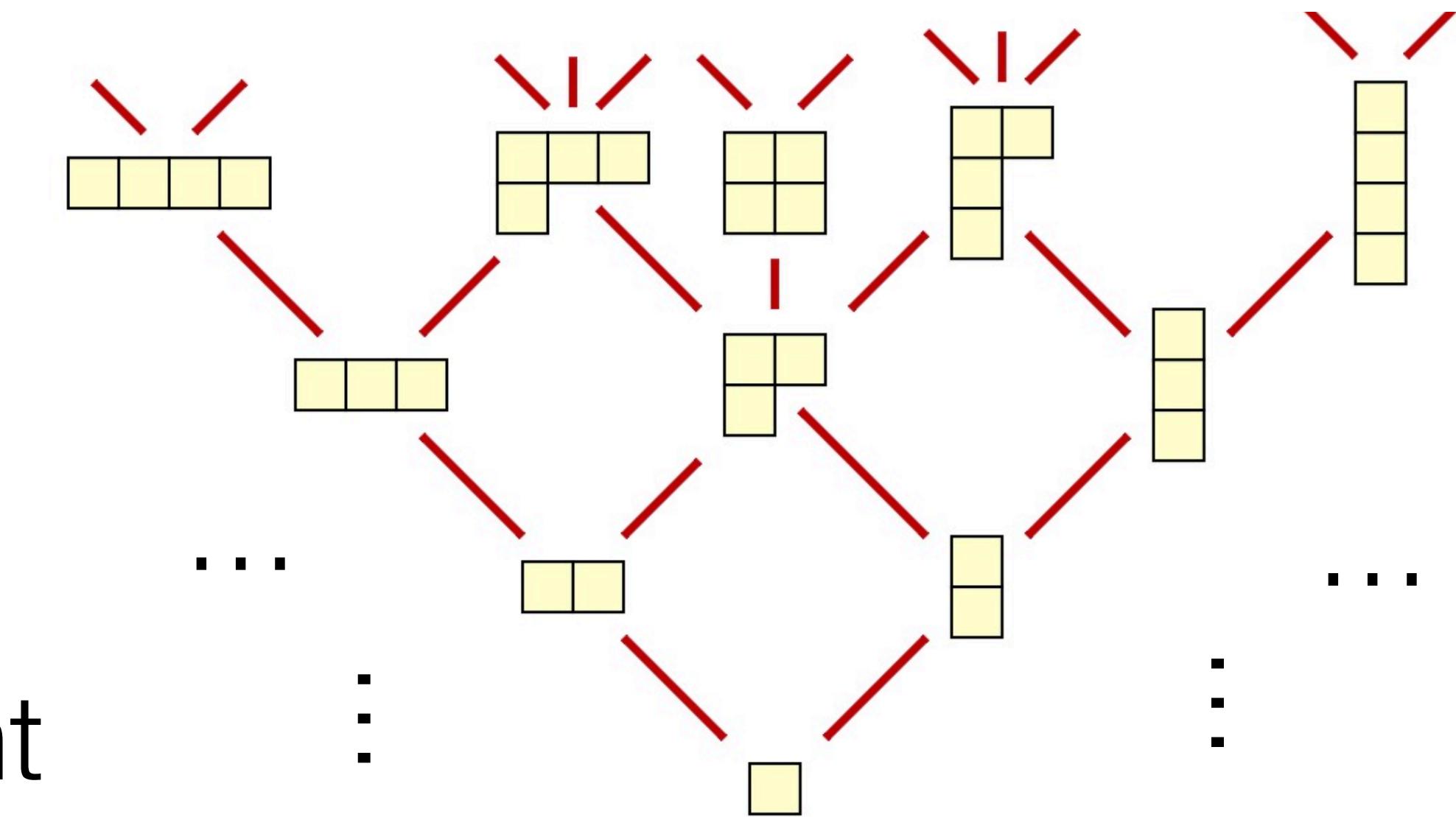
# INVARIANTS FOR SB-CRDTs

- ▶ Invariant constraints
- ▶ Operations preserve the invariant

$$\forall \text{op}, \sigma, \sigma', \sigma \models \text{Pre}_{\text{op}} \wedge (\sigma, \sigma') \in [\text{op}] \Rightarrow \sigma' \models \text{Inv}$$

- ▶ **merge** preserves the invariant

$$\forall \sigma, \sigma', \sigma'', (\sigma, \sigma'') \models \text{Pre}_{\text{merge}} \wedge \text{merge}(\sigma, \sigma'') = \sigma' \Rightarrow \sigma' \models \text{Inv}$$



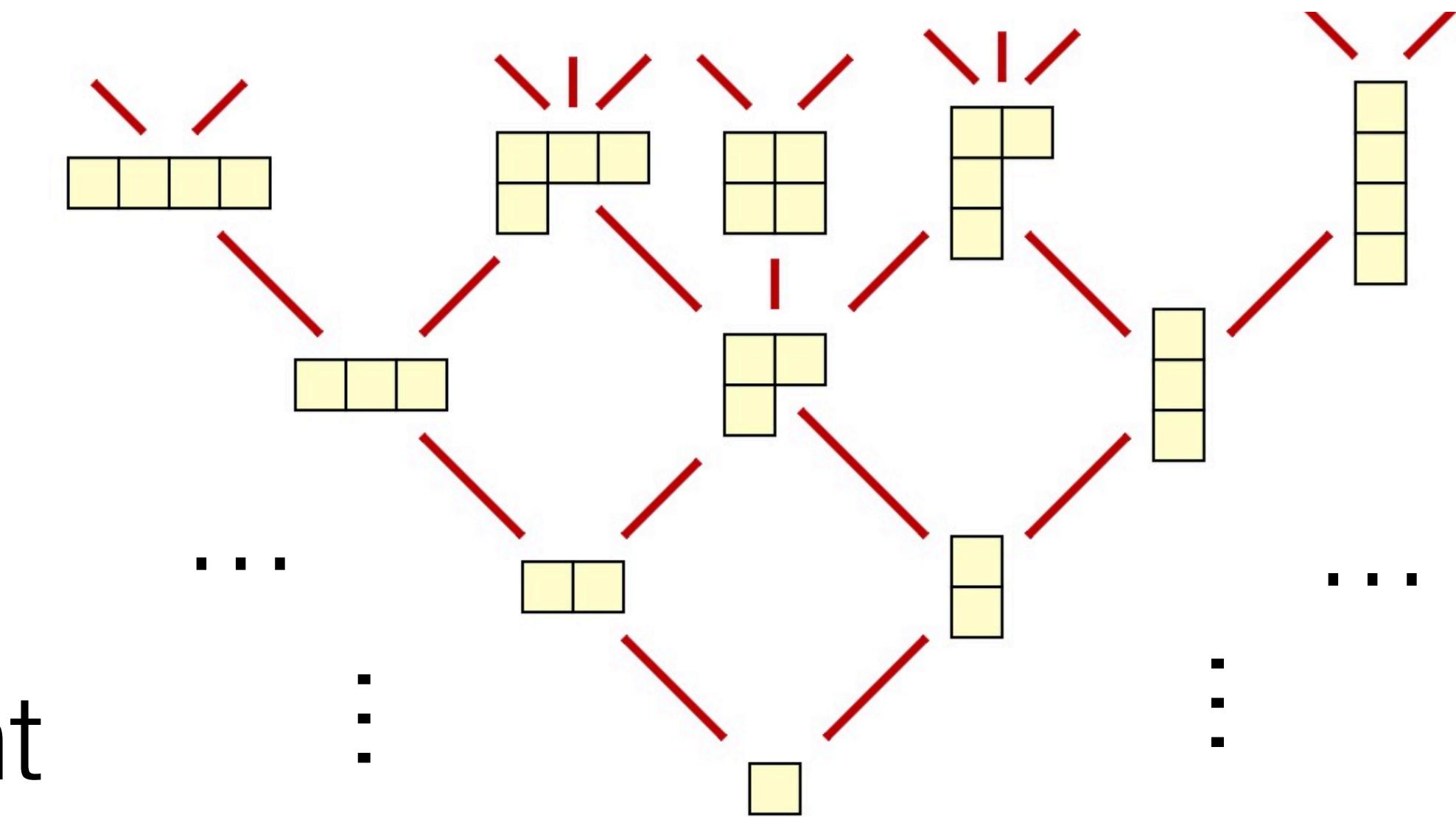
# INVARIANTS FOR SB-CRDTs

- ▶ Invariant constraints
- ▶ Operations preserve the invariant

$$\forall \text{op}, \sigma, \sigma', \sigma \models \text{Pre}_{\text{op}} \wedge (\sigma, \sigma') \in [\text{op}] \Rightarrow \sigma' \models \text{Inv}$$

- ▶ **merge** preserves the invariant

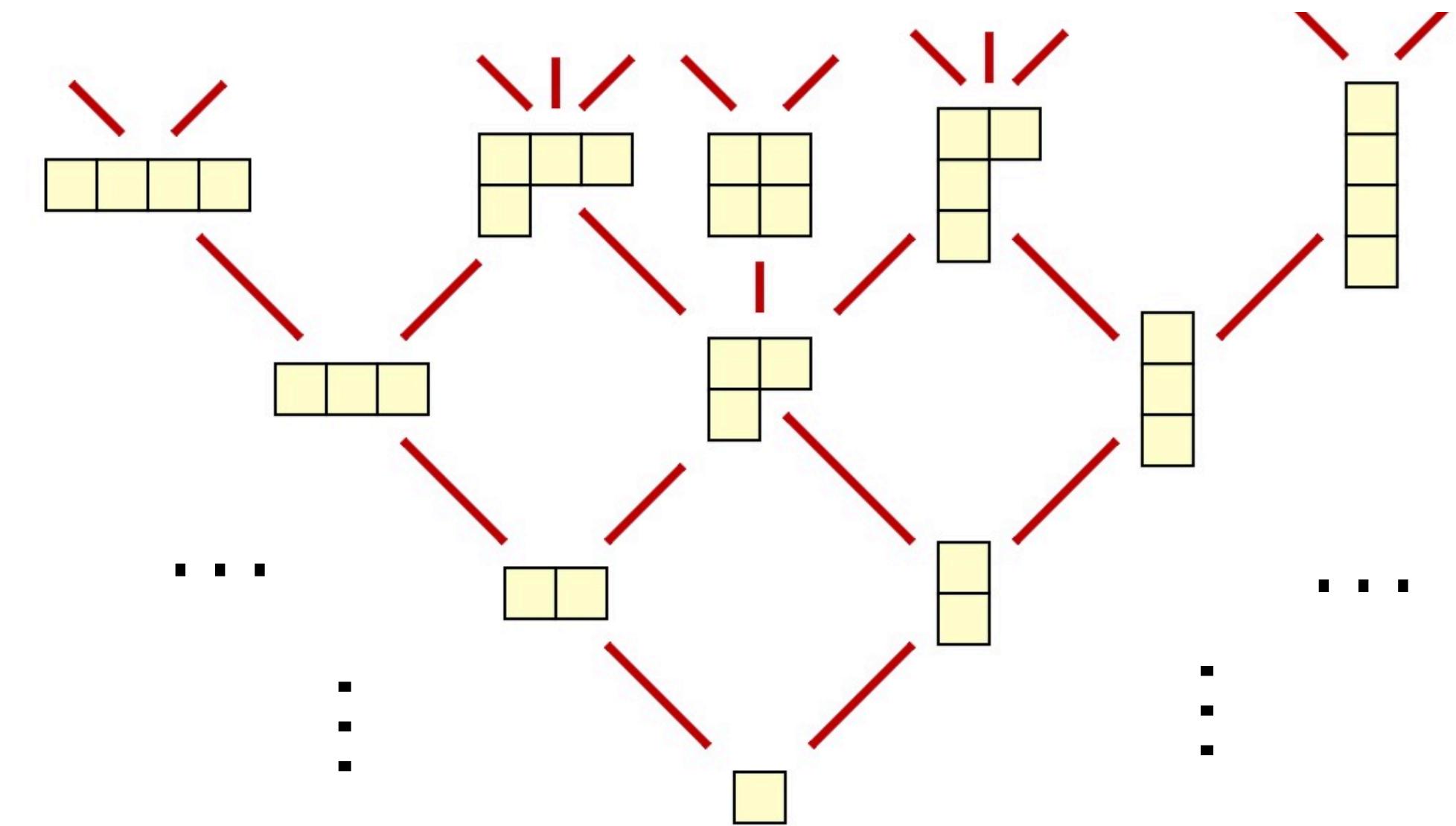
$$\forall \sigma, \sigma', \sigma'', (\sigma, \sigma'') \models \text{Pre}_{\text{merge}} \wedge \text{merge}(\sigma, \sigma'') = \sigma' \Rightarrow \sigma' \models \text{Inv}$$



Eg: Bounded Counter  
 $\text{Total}(\sigma) \geq 0 \wedge \text{Total}(\sigma') \geq 0$

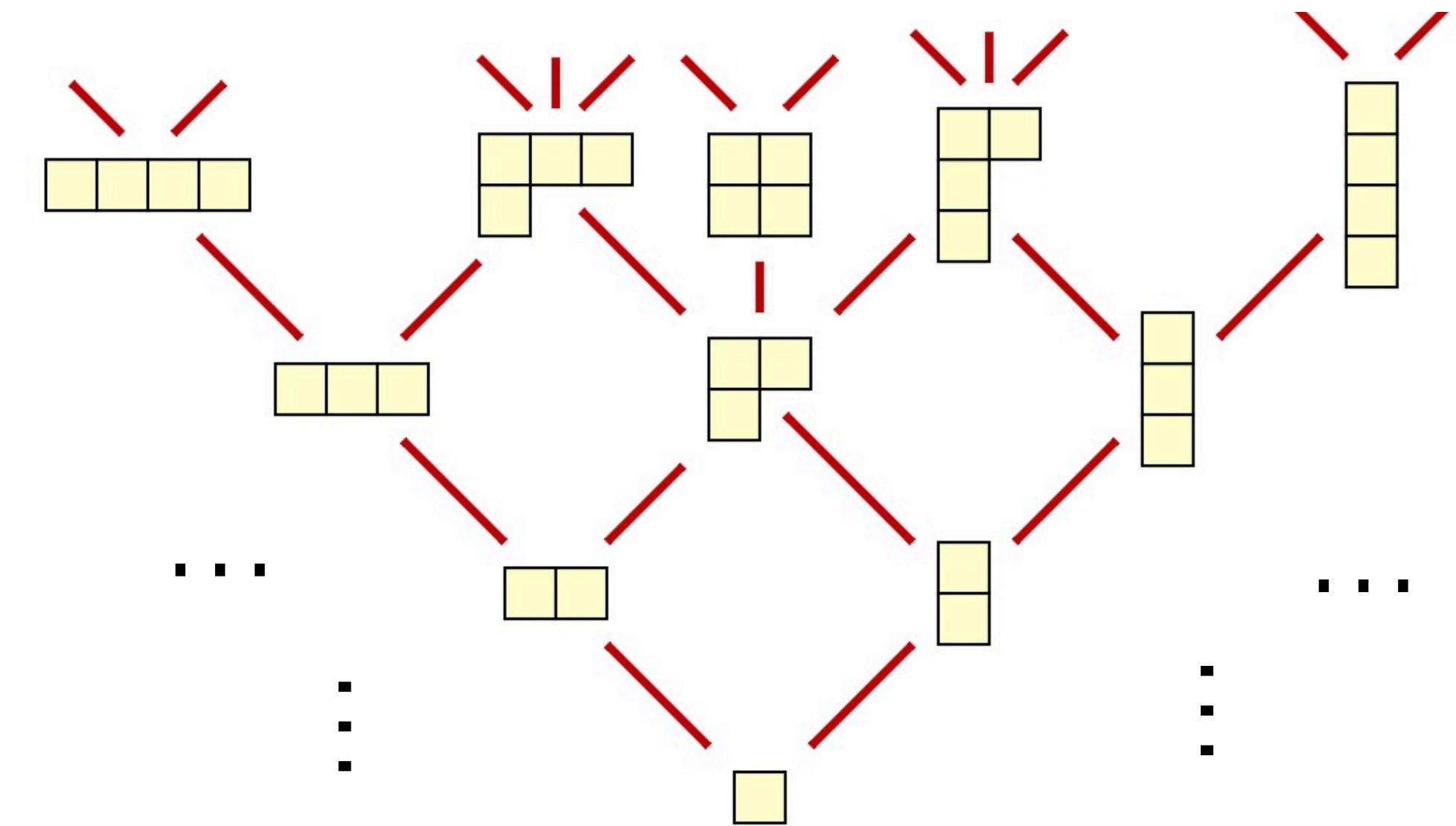
# INVARIANTS FOR SB-CRDTs

- Invariant constraints



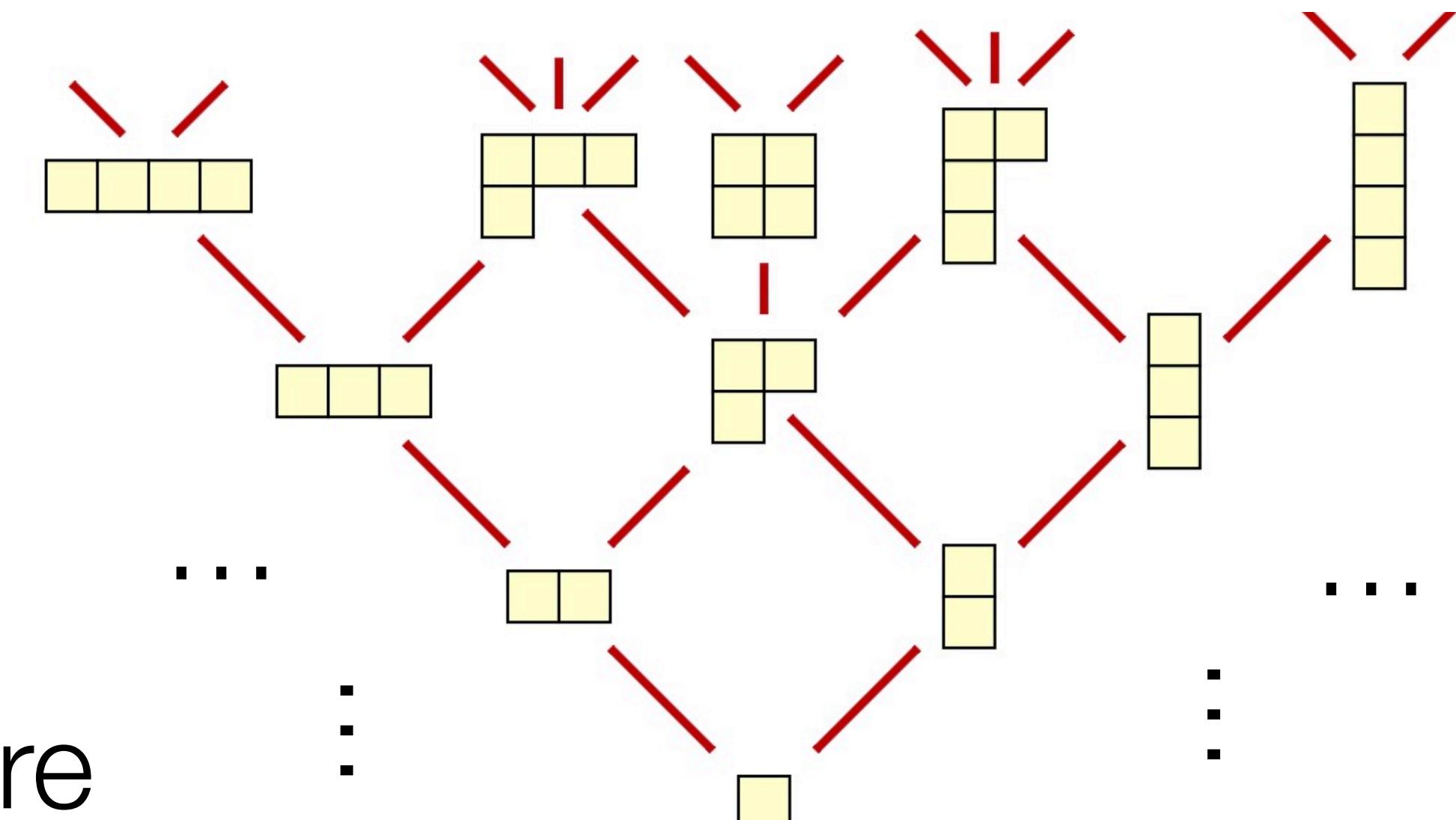
# INVARIANTS FOR SB-CRDTs

- ▶ Invariant constraints
- ▶ **merge** can execute at any time



# INVARIANTS FOR SB-CRDTs

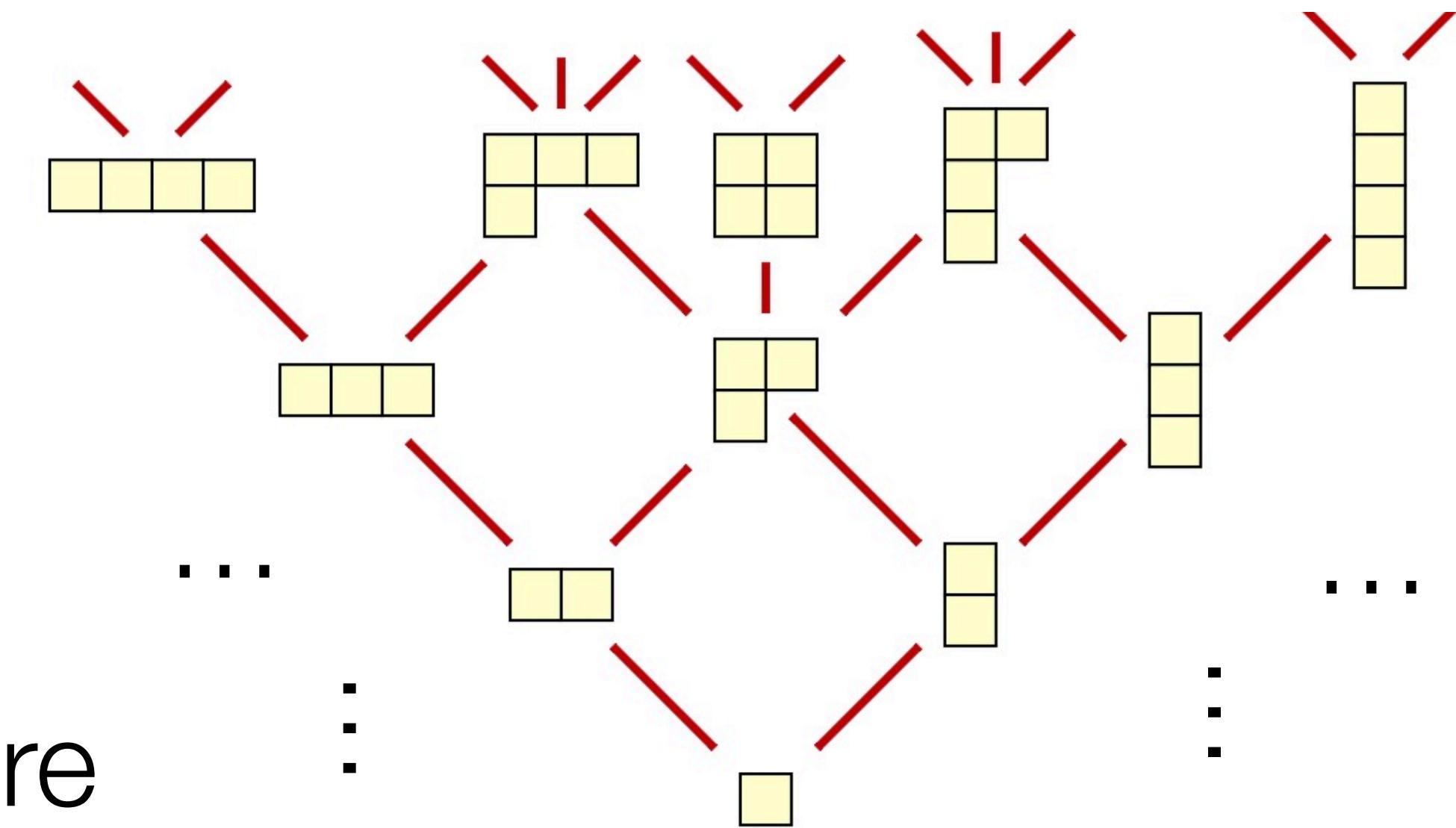
- ▶ Invariant constraints
- ▶ **merge** can execute at any time
- ▶ Operations preserve the **merge** Pre



$$\forall \text{op}, \sigma, \sigma', \sigma'', \left( \begin{array}{l} \sigma \models \text{Pre}_{\text{op}} \wedge \\ (\sigma, \sigma'') \models \text{Pre}_{\text{merge}} \wedge \\ (\sigma, \sigma') \in [\text{op}] \end{array} \right) \Rightarrow (\sigma', \sigma'') \models \text{Pre}_{\text{merge}}$$

# INVARIANTS FOR SB-CRDTs

- ▶ Invariant constraints
- ▶ **merge** can execute at any time
- ▶ Operations preserve the **merge** Pre



$$\forall \text{op}, \sigma, \sigma', \sigma'', \left( \begin{array}{l} \sigma \models \text{Pre}_{\text{op}} \wedge \\ (\sigma, \sigma'') \models \text{Pre}_{\text{merge}} \wedge \\ (\sigma, \sigma') \in [\text{op}] \end{array} \right) \Rightarrow (\sigma', \sigma'') \models \text{Pre}_{\text{merge}}$$

- ▶ **merge** Pre is an invariant

# EXAMPLE: TOKEN IMPLEMENTATION

# (Eg.) MUTUAL EXCLUSION CRDT

# (Eg.) MUTUAL EXCLUSION CRDT

- ▶ At most one replica has rights to the mutual exclusion token

# (Eg.) MUTUAL EXCLUSION CRDT

- ▶ At most one replica has rights to the mutual exclusion token
  - ▶ An array  $V[R]$  to indicate who has the “lock”

# (Eg.) MUTUAL EXCLUSION CRDT

- ▶ At most one replica has rights to the mutual exclusion token
  - ▶ An array  $V[R]$  to indicate who has the “lock”
  - ▶ Escrow
    - ▶ Owner transfer the token to the next owner

# (Eg.) MUTUAL EXCLUSION CRDT

- ▶ At most one replica has rights to the mutual exclusion token
  - ▶ An array  $V[R]$  to indicate who has the “lock”
  - ▶ Escrow
    - ▶ Owner transfer the token to the next owner
- ▶ Comparison function?

# (Eg.) MUTUAL EXCLUSION CRDT

- ▶ At most one replica has rights to the mutual exclusion token
  - ▶ An array  $V[R]$  to indicate who has the “lock”
  - ▶ Escrow
    - ▶ Owner transfer the token to the next owner
- ▶ Comparison function?
- ▶ Add a timestamp  $t$

# (Eg.) MUTUAL EXCLUSION CRDT

- ▶ At most one replica has rights to the mutual exclusion token
  - ▶ An array  $V[R]$  to indicate who has the “lock”
  - ▶ Escrow
    - ▶ Owner transfer the token to the next owner
- ▶ Comparison function?
- ▶ Add a timestamp  $t$

```
transfer((t,V),ro):  
    assert(V[r] = 1 ∧ (∀to ≠ t, t ≥ to))  
    t = t+1  
    V[rs] = 0 # self  
    V[ro] = 1 # other
```

```
merge((t,V),(to,Vo)):  
    t = max(t,to)  
    v = (to<t)?V:Vo
```

# (Eg.) MUTUAL EXCLUSION CRDT

```
transfer((t,V),ro):  
    assert(V[r] = 1 ∧ (∀to ≠ t, t ≥ to))  
    t = t+1  
    V[rs] = 0 # self  
    V[ro] = 1 # other
```

```
merge((t,V),(to,Vo)):  
    t = max(t,to)  
    v = (to<t)?V:Vo
```

# (Eg.) MUTUAL EXCLUSION CRDT

```
transfer((t,V),ro):  
    assert(V[r] = 1 ∧ (∀to ≠ t, t ≥ to))  
    t = t+1  
    V[rs] = 0 # self  
    V[ro] = 1 # other
```

```
merge((t,V),(to,Vo)):  
    t = max(t,to)  
    v = (to<t)?V:Vo
```

- ▶ Order:

# (Eg.) MUTUAL EXCLUSION CRDT

```
transfer((t,V),ro):  
    assert(V[r] = 1 ∧ (∀to ≠ t, t ≥ to))  
    t = t+1  
    V[rs] = 0 # self  
    V[ro] = 1 # other
```

```
merge((t,V),(to,Vo)):  
    t = max(t,to)  
    v = (to<t)?V:Vo
```

► Order:

$$(t_0, V_0) \leq (t_1, V_1) = t_0 \leq t_1 \wedge (t_0 \Rightarrow t_1 \wedge V_0 \leq V_1)$$

# (Eg.) MUTUAL EXCLUSION CRDT

```
transfer((t,V),ro):  
    assert(V[r] = 1 ∧ (∀to ≠ t, t ≥ to))  
    t = t+1  
    V[rs] = 0 # self  
    V[ro] = 1 # other
```

```
merge((t,V),(to,Vo)):  
    t = max(t,to)  
    v = (to<t)?V:Vo
```

- ▶ Order:  $(t_0, V_0) \leq (t_1, V_1) = t_0 \leq t_1 \wedge (t_0 \Rightarrow t_1 \wedge V_0 \leq V_1)$
- ▶ Invariant:

# (Eg.) MUTUAL EXCLUSION CRDT

```
transfer((t,V),ro):  
    assert(V[r] = 1 ∧ (∀to ≠ t, t ≥ to))  
    t = t+1  
    V[rs] = 0 # self  
    V[ro] = 1 # other
```

```
merge((t,V),(to,Vo)):  
    t = max(t,to)  
    v = (to<t)?V:Vo
```

► Order:

$$(t_0, V_0) \leq (t_1, V_1) = t_0 \leq t_1 \wedge (t_0 \Rightarrow t_1 \wedge V_0 \leq V_1)$$

► Invariant:

$$\text{Inv}(t, V) = \sum_r V[r] = 1$$

# (Eg.) MUTUAL EXCLUSION CRDT

```
transfer((t,V),ro):  
    assert(V[r] = 1 ∧ (∀to ≠ t, t ≥ to))  
    t = t+1  
    V[rs] = 0 # self  
    V[ro] = 1 # other
```

```
merge((t,V),(to,Vo)):  
    t = max(t,to)  
    v = (to<t)?V:Vo
```

- ▶ Order:  $(t_0, V_0) \leq (t_1, V_1) = t_0 \leq t_1 \wedge (t_0 \Rightarrow t_1 \wedge V_0 \leq V_1)$
- ▶ Invariant:  $\text{Inv}(t, V) = \sum_r V[r] = 1$
- ▶ Merge precondition:

# (EG.) MUTUAL EXCLUSION CRDT

```
transfer((t,V),ro):  
    assert(V[r] = 1 ∧ (∀to ≠ t, t ≥ to))  
    t = t+1  
    V[rs] = 0 # self  
    V[ro] = 1 # other
```

```
merge((t,V),(to,Vo)):  
    t = max(t,to)  
    v = (to<t)?V:Vo
```

- Order:  $(t_0, V_0) \leq (t_1, V_1) = t_0 \leq t_1 \wedge (t_0 \Rightarrow t_1 \wedge V_0 \leq V_1)$
- Invariant:  $\text{Inv}(t, V) = \sum_r V[r] = 1$
- Merge precondition:  $\text{Pre}_{\text{merge}}((t_s, V_s), (t_o, V_o)) = \text{Inv}(t_s, V_s) \wedge \text{Inv}(t_o, V_o) \wedge (t_s = t_o \Rightarrow V_s = V_o) \wedge (V[r_s] = 1 \Rightarrow t_s \geq t_o)$

# TOOL SUPPORT

- ▶ Input
  - ▶ Definition of Order
  - ▶ Definition of merging function
- ▶ Invariant

# TOOL SUPPORT

- ▶ Input
  - ▶ Definition of Order
  - ▶ Definition of merging function
  - ▶ Invariant
- ▶ Soteria (sister to CISE/CEC)
  - ▶ Implemented on top of Boogie
  - ▶ Performs the lattice and invariant checks

# TOOL SUPPORT

- ▶ Input
  - ▶ Definition of Order
  - ▶ Definition of merging function
  - ▶ Invariant
- ▶ Soteria (sister to CISE/CEC)
  - ▶ Implemented on top of Boogie
  - ▶ Performs the lattice and invariant checks
- ▶ Work in progress

# TOOL SUPPORT

## ► Input

```
~/r/c/c/s/s/soteria (origin/rewriting±) ► python3 soteria.py specs/token.spec
INFO      : **** token ****
INFO      : Checking the syntax
INFO      : Parsing the specification
INFO      : Checking the well-formedness of the specification
INFO      : Checking convergence
INFO      : Checking safety
INFO      : The specification is safe!!!
```

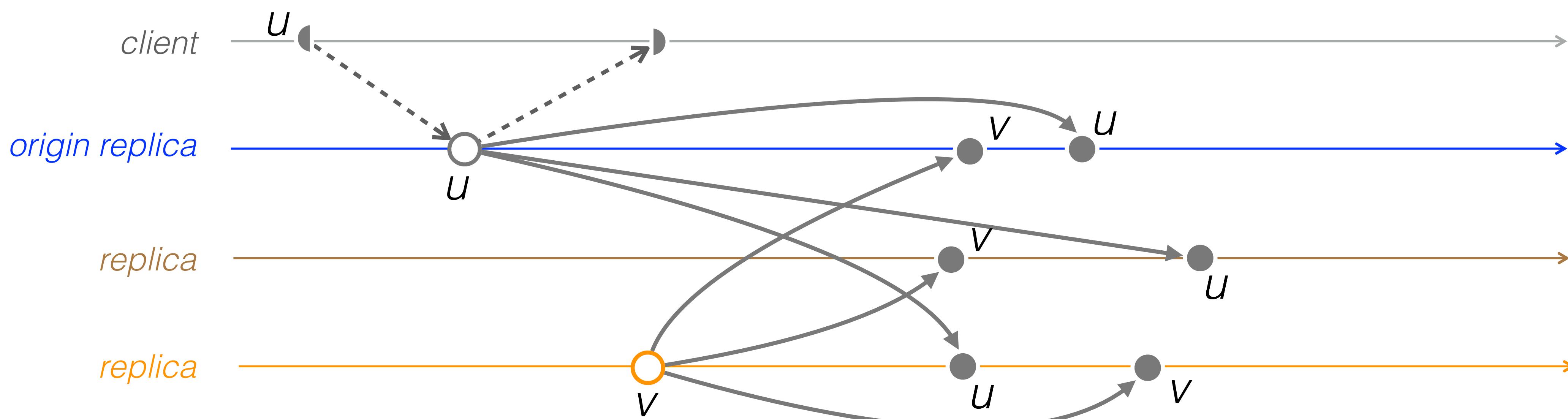
- Implemented on top of Boogie
- Performs the lattice and invariant checks
- Work in progress

# CHECKING INVARIANTS

## OPERATION-BASED CRDTs

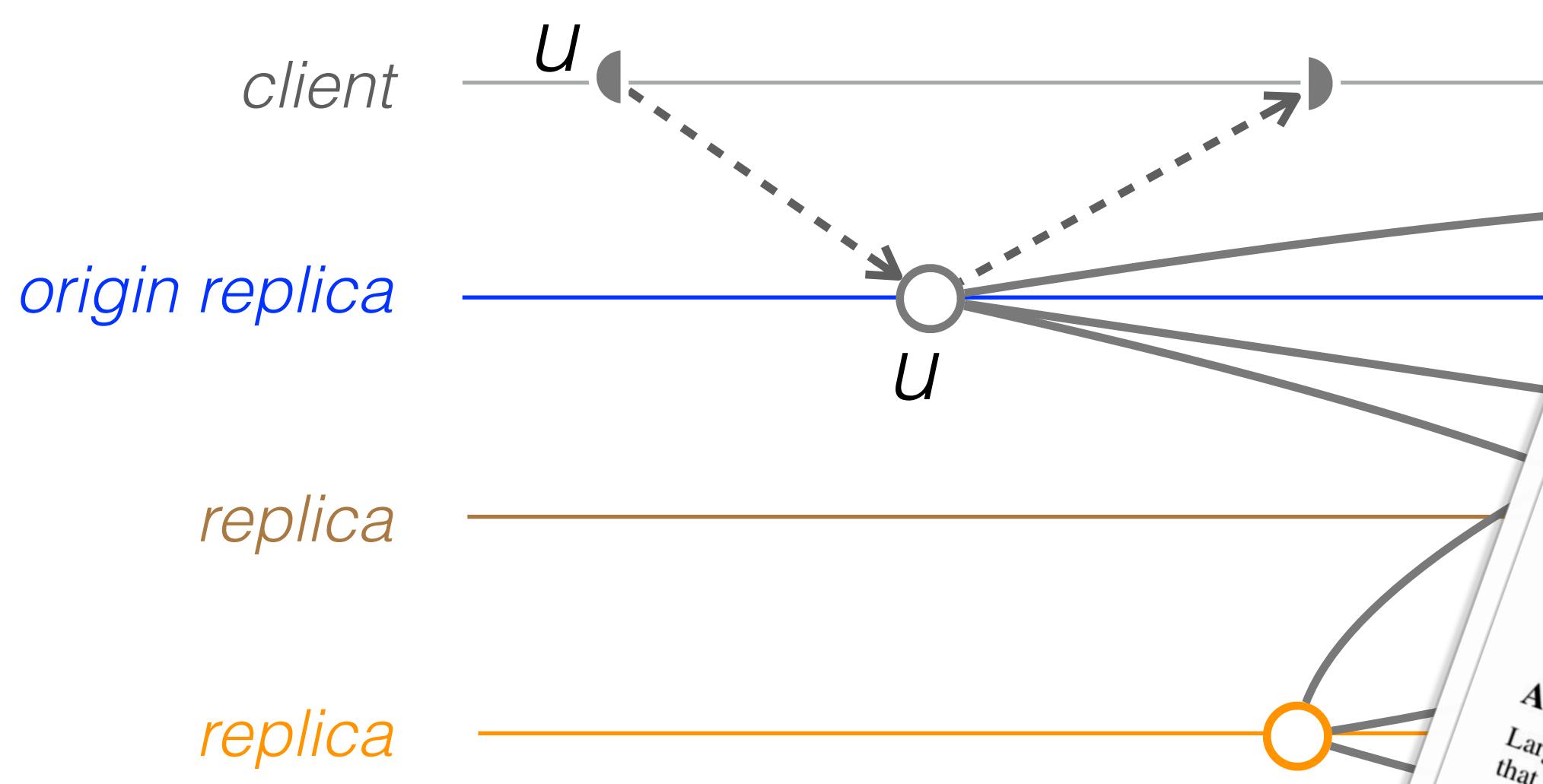
# OPERATION-BASED CRDTs

- ▶ Operation-based CRDTs
- ▶ Each operation is delivered to each replica



# OPERATION-BASED CRDTs

- ▶ Operation-based CRDTs
- ▶ Each operation is delivered to each replica



$100 \text{ €} \geq 0$

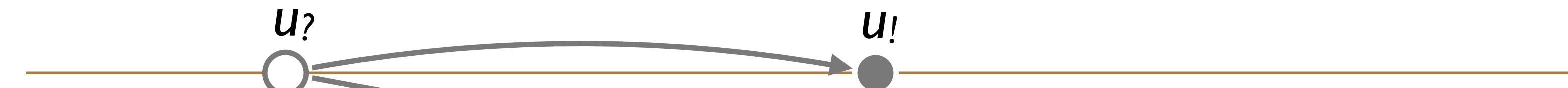
$\sigma: I$



$\sigma: I$

$100 \text{ €} \geq 0$

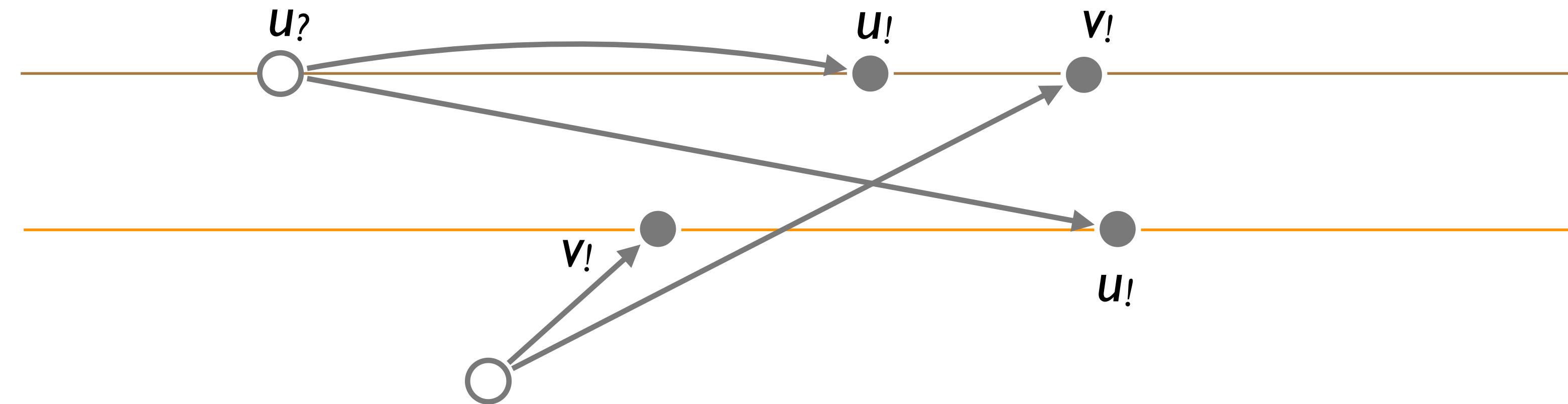
$100 \text{ €} \geq 0$   
 $\sigma: I$



$\sigma: I$   
 $100 \text{ €} \geq 0$

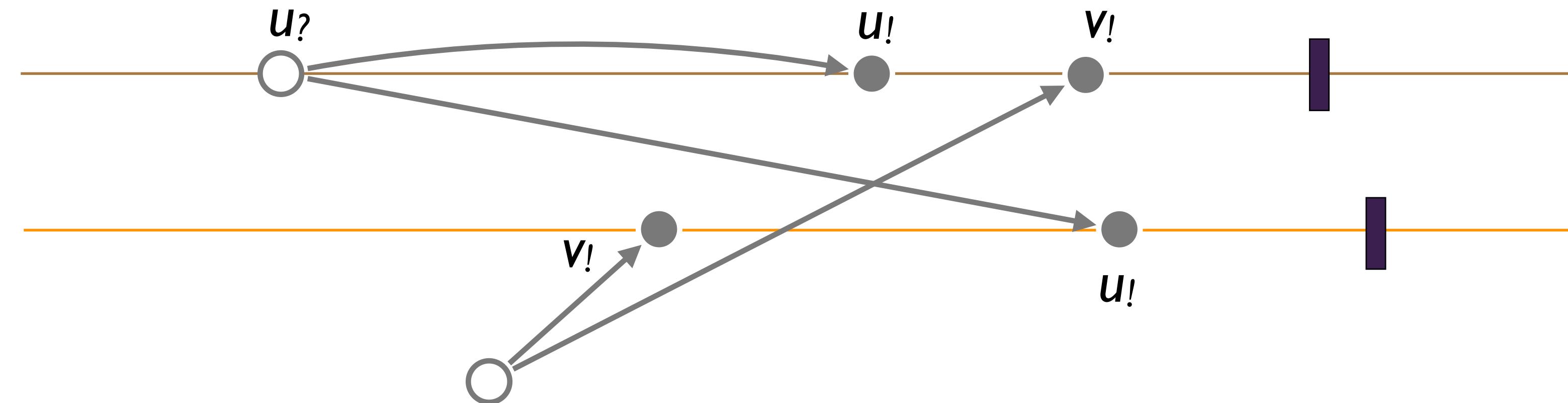
$100 \text{ €} \geq 0$   
 $\sigma: I$

$\sigma: I$   
 $100 \text{ €} \geq 0$



$100 \text{ €} \geq 0$   
 $\sigma: I$

$\sigma: I$   
 $100 \text{ €} \geq 0$

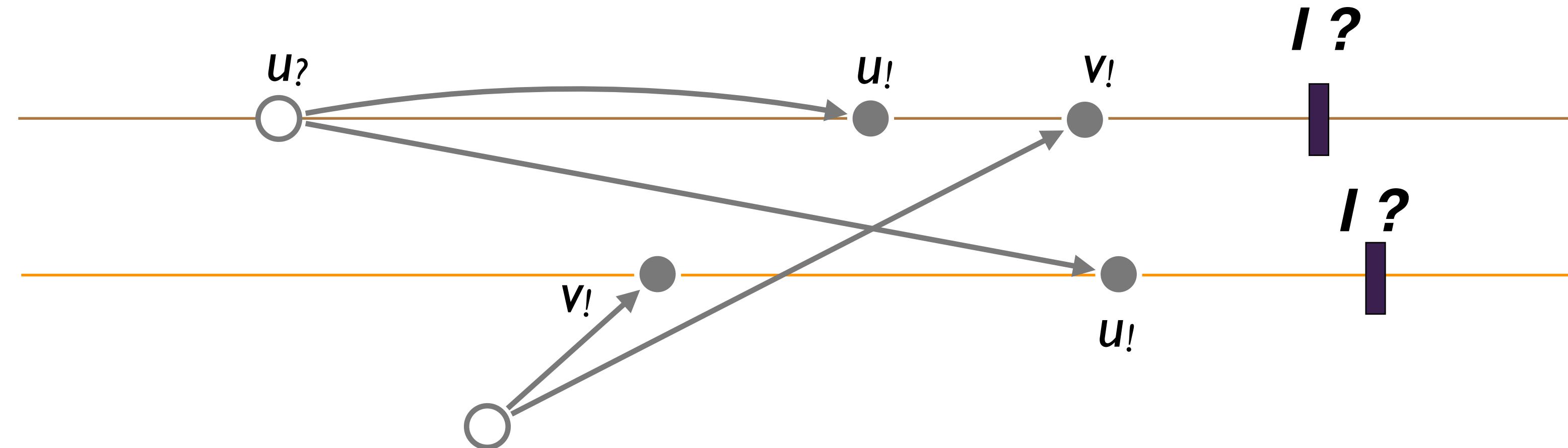


$100 \text{ €} \geq 0$

$\sigma: I$

$\sigma: I$

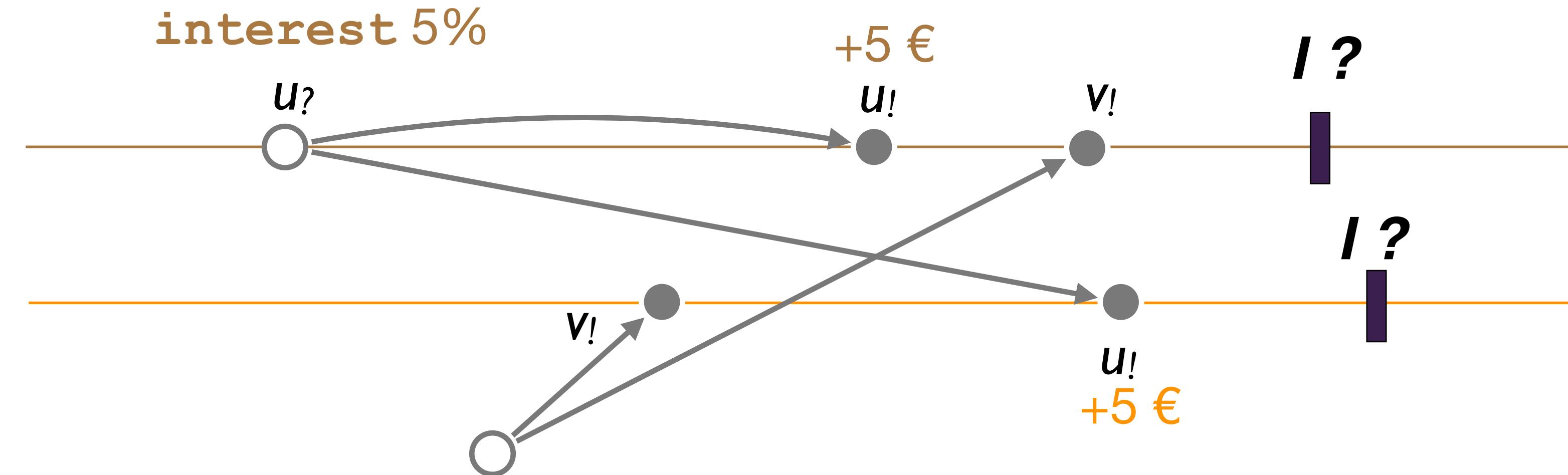
$100 \text{ €} \geq 0$



$100 \text{ €} \geq 0$   
 $\sigma: I$

$\sigma: I$   
 $100 \text{ €} \geq 0$

interest 5%



$100 \text{ €} \geq 0$

$\sigma: I$

$100 \text{ €} \geq 0$

$\sigma: I$

interest 5%

$u?$

+5 €

-100

$I ?$

$u!$

$v!$

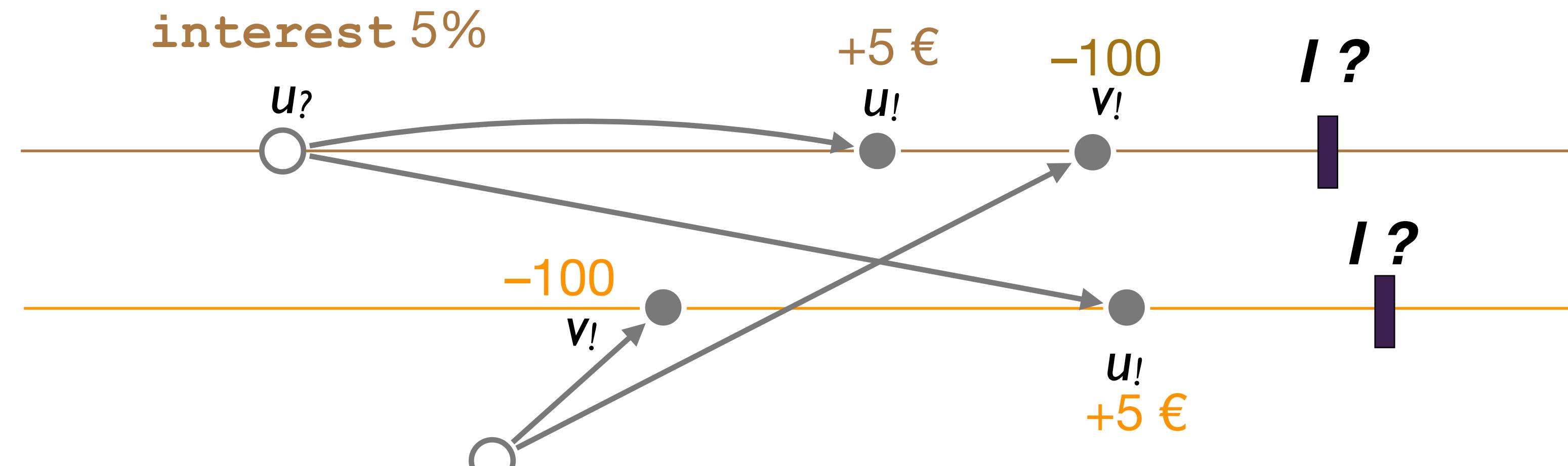
$I ?$

$u!$   
+5 €

withdraw(100)

-100

$v!$



$100 \text{ €} \geq 0$

$\sigma: I$

$100 \text{ €} \geq 0$

$\sigma: I$

interest 5%

$u?$

$+5 \text{ €}$

$u!$

$-100 \text{ €}$

$v!$

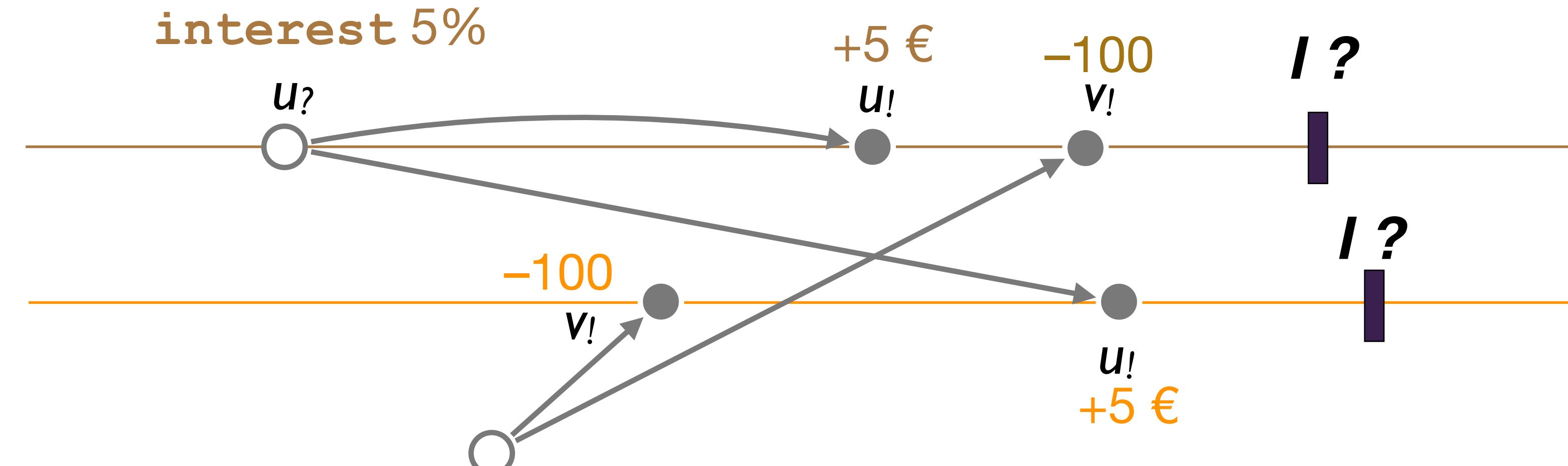
$I ?$

$I ?$

withdraw(100)

$5 \text{ €} \geq 0$

$5 \text{ €} \geq 0$



$100 \text{ €} \geq 0$

$\sigma: I$

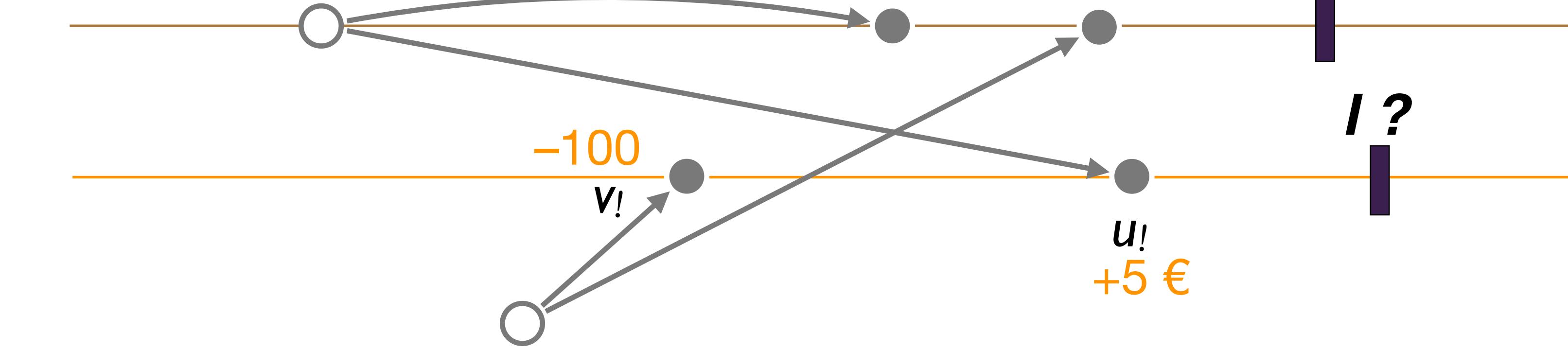
interest 5%

$u?$

+5 €

-100

$I ?$

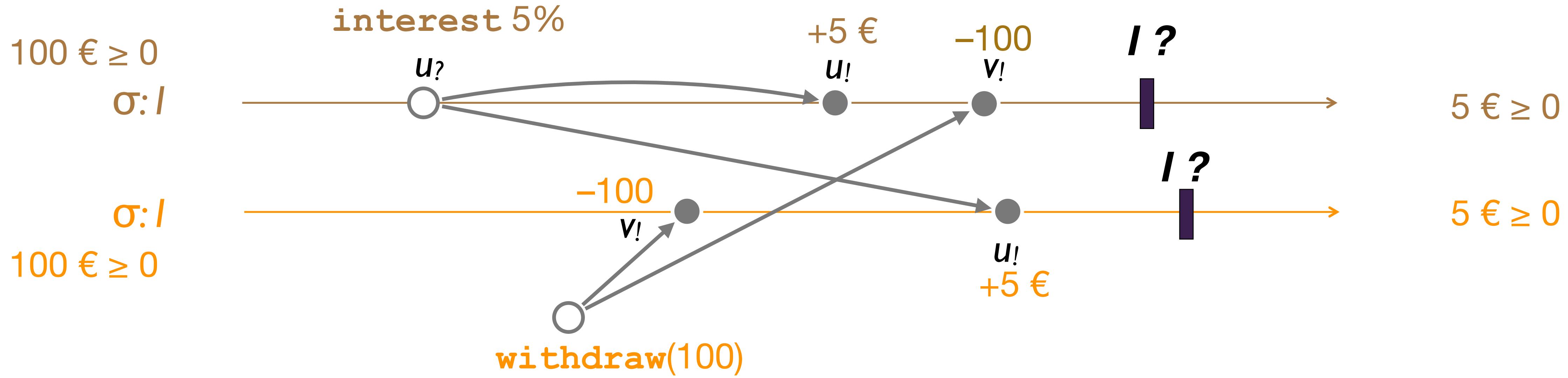


$\sigma: I$

$100 \text{ €} \geq 0$

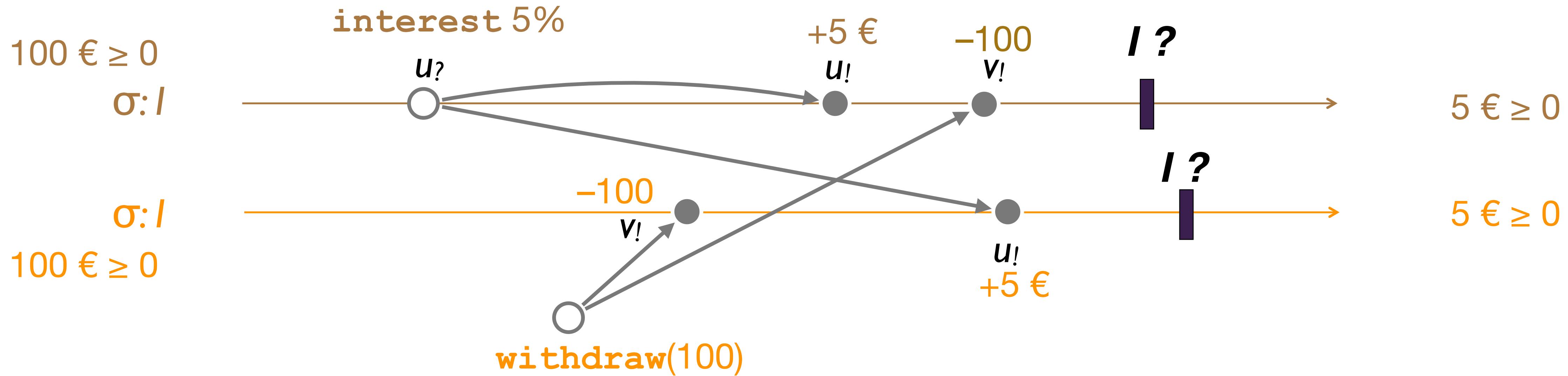
withdraw(100)

CISE Rules



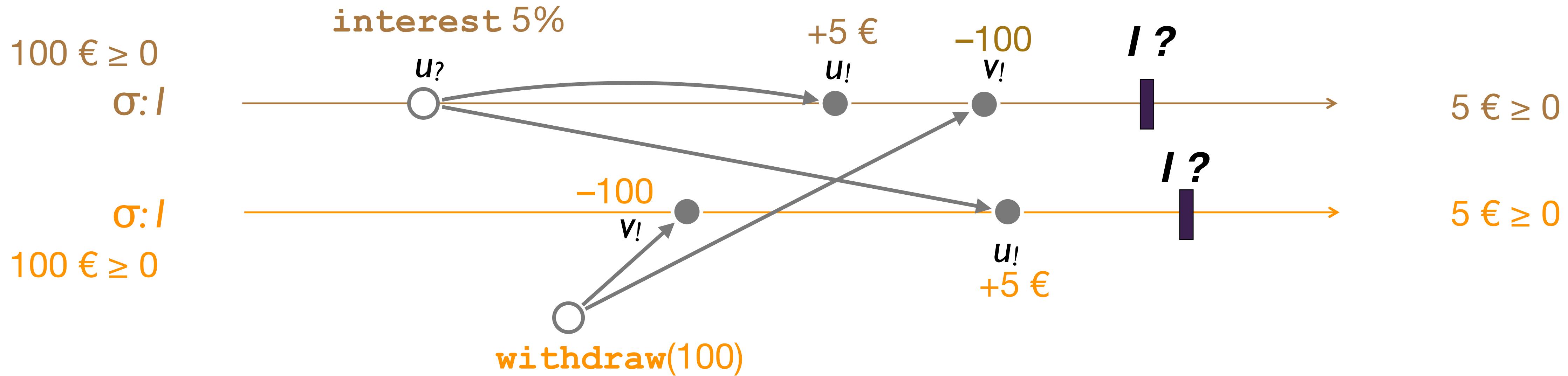
## CISE Rules

- 1: Sequential correctness
  - Single operations preserve the invariant



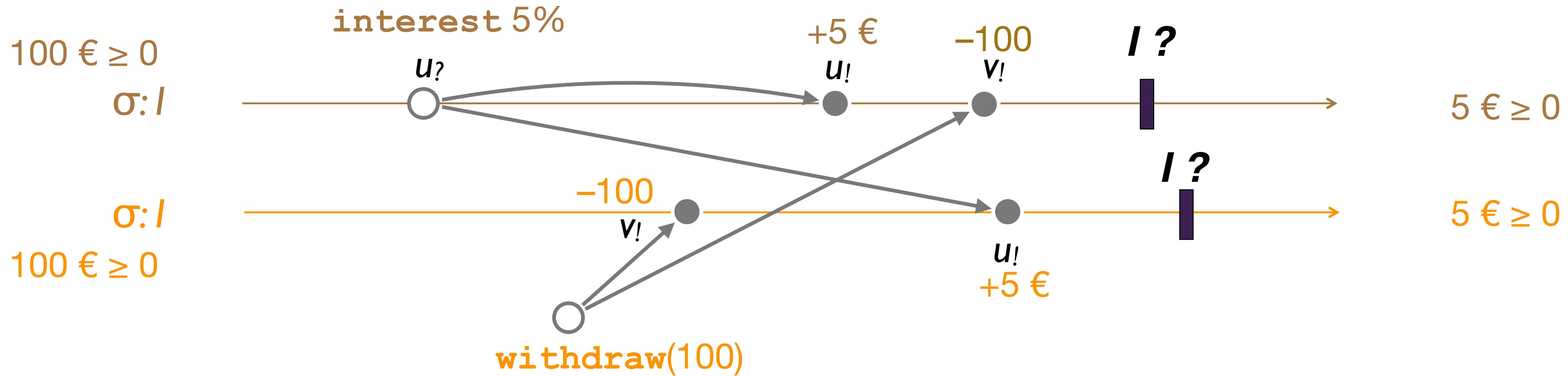
## CISE Rules

- 1: Sequential correctness
  - ▶ Single operations preserve the invariant
- 2: Convergence
  - ▶ Concurrent effectors commute



## CISE Rules

- 1: Sequential correctness
  - ▶ Single operations preserve the invariant
- 2: Convergence
  - ▶ Concurrent effectors commute
- 3: Precondition Stability
  - ▶ Preconditions are stable under concurrent operations



## CISE Rules

- 1: Sequential correctness
  - Single operations preserve the invariant
- 2: Convergence
  - Concurrent effectors commute
- 3: Precondition Stability
  - Preconditions are stable under concurrent operations

}  $\Rightarrow$  Invariant

# SIMPLE EXAMPLE: BANK ACCOUNT

- ▶ Operations: **deposit**(amount), **withdraw**(amount)

# SIMPLE EXAMPLE: BANK ACCOUNT

- ▶ Operations: **deposit** (amount), **withdraw** (amount)
- ▶ Invariant:  $\text{balance} \geq 0$

# SIMPLE EXAMPLE: BANK ACCOUNT

- ▶ Operations: **deposit** (amount), **withdraw** (amount)
- ▶ Invariant:  $\text{balance} \geq 0$
- ▶ Rule 1

1: Sequential correctness

- ▶ Single operations preserve the invariant

# SIMPLE EXAMPLE: BANK ACCOUNT

- ▶ Operations: **deposit** (amount), **withdraw** (amount)
- ▶ Invariant:  $\text{balance} \geq 0$
- ▶ Rule 1
  - ▶ Precondition for withdraw:  $\text{amount} \leq \text{balance}$

1: Sequential correctness

- ▶ Single operations preserve the invariant

# SIMPLE EXAMPLE: BANK ACCOUNT

- ▶ Operations: **deposit** (amount), **withdraw** (amount)
- ▶ Invariant:  $\text{balance} \geq 0$
- ▶ Rule 1
  - ▶ Precondition for withdraw:  $\text{amount} \leq \text{balance}$
- ▶ Rule 2:

2: Convergence

- ▶ Concurrent effectors commute

# SIMPLE EXAMPLE: BANK ACCOUNT

- ▶ Operations: **deposit** (amount), **withdraw** (amount)
- ▶ Invariant:  $\text{balance} \geq 0$
- ▶ Rule 1
  - ▶ Precondition for withdraw:  $\text{amount} \leq \text{balance}$
- ▶ Rule 2: OK

2: Convergence

- ▶ Concurrent effectors commute

# SIMPLE EXAMPLE: BANK ACCOUNT

- ▶ Operations: **deposit** (amount), **withdraw** (amount)
- ▶ Invariant:  $\text{balance} \geq 0$
- ▶ Rule 1
  - ▶ Precondition for withdraw:  $\text{amount} \leq \text{balance}$
- ▶ Rule 2: OK
- ▶ Rule 3

3: Precondition Stability

- ▶ Preconditions are stable under concurrent operations

# SIMPLE EXAMPLE: BANK ACCOUNT

- ▶ Operations: **deposit** (amount), **withdraw** (amount)
- ▶ Invariant:  $\text{balance} \geq 0$
- ▶ Rule 1
  - ▶ Precondition for withdraw:  $\text{amount} \leq \text{balance}$
- ▶ Rule 2: OK
- ▶ Rule 3 : (**withdraw** || **withdraw**) unsafe

3: Precondition Stability

- ▶ Preconditions are stable under concurrent operations

# SIMPLE EXAMPLE: BANK ACCOUNT

- ▶ Operations: **deposit** (amount), **withdraw** (amount)
- ▶ Invariant:  $\text{balance} \geq 0$
- ▶ Rule 1
  - ▶ Precondition for withdraw:  $\text{amount} \leq \text{balance}$
- ▶ Rule 2: OK
- ▶ Rule 3 : (**withdraw** || **withdraw**) unsafe
  - ▶ fixed with concurrency control

$100 \text{ €} \geq 0$

$\sigma: I$

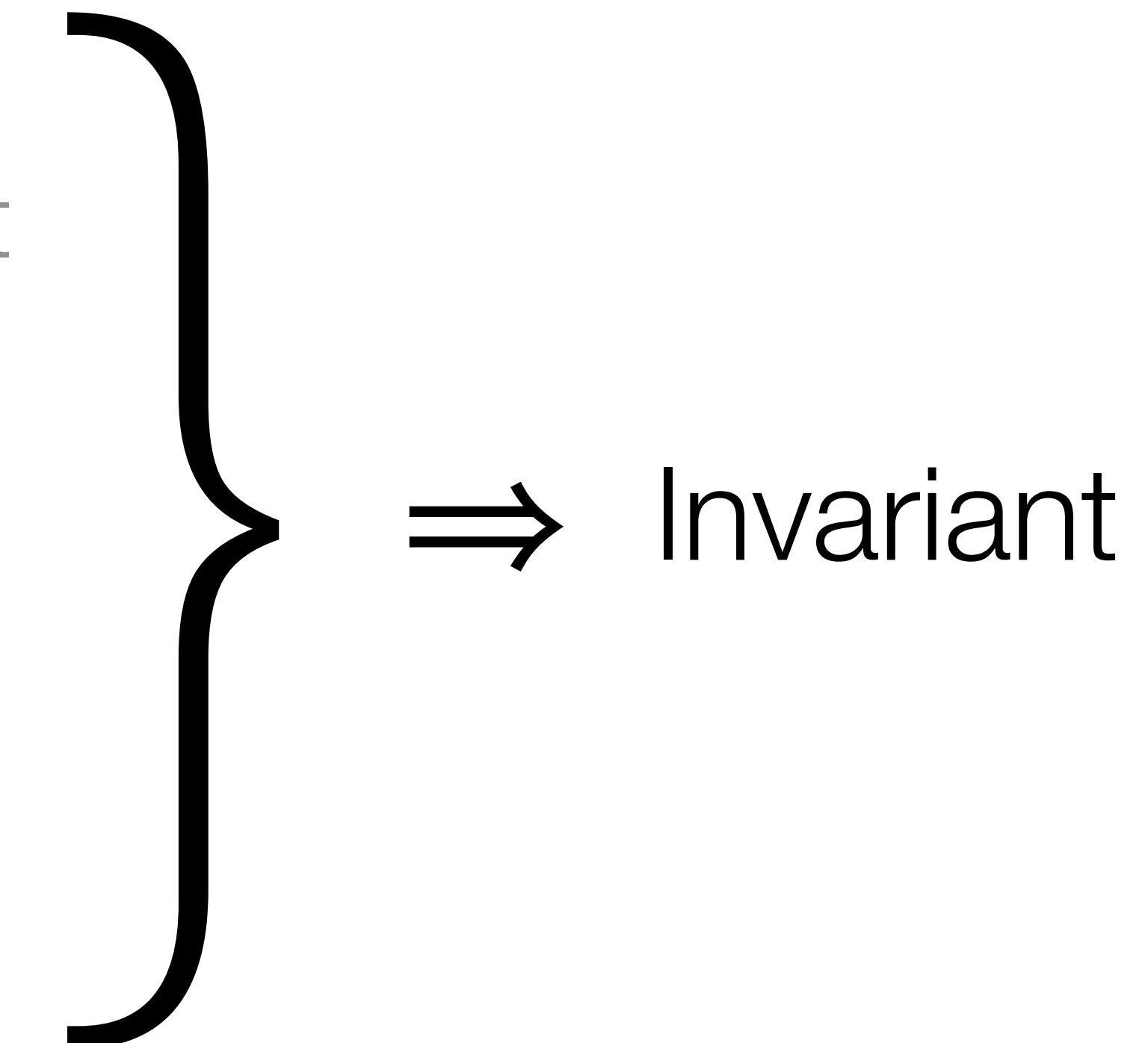


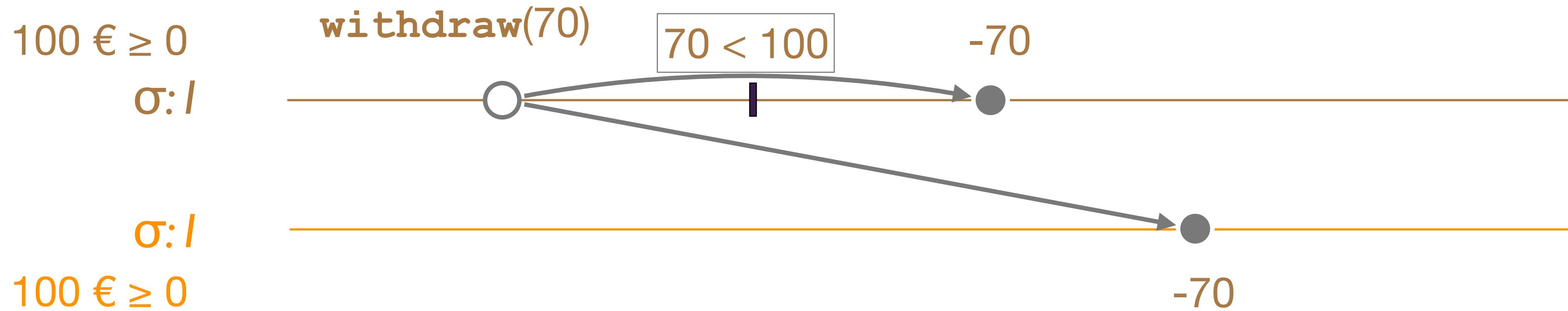
$\sigma: I$

$100 \text{ €} \geq 0$

## CISE Rules

- 1: Sequential correctness
  - ▶ Single operations preserve the invariant
- 2: Convergence
  - ▶ Concurrent effectors commute
- 3: Precondition Stability
  - ▶ Preconditions are stable under concurrent operations

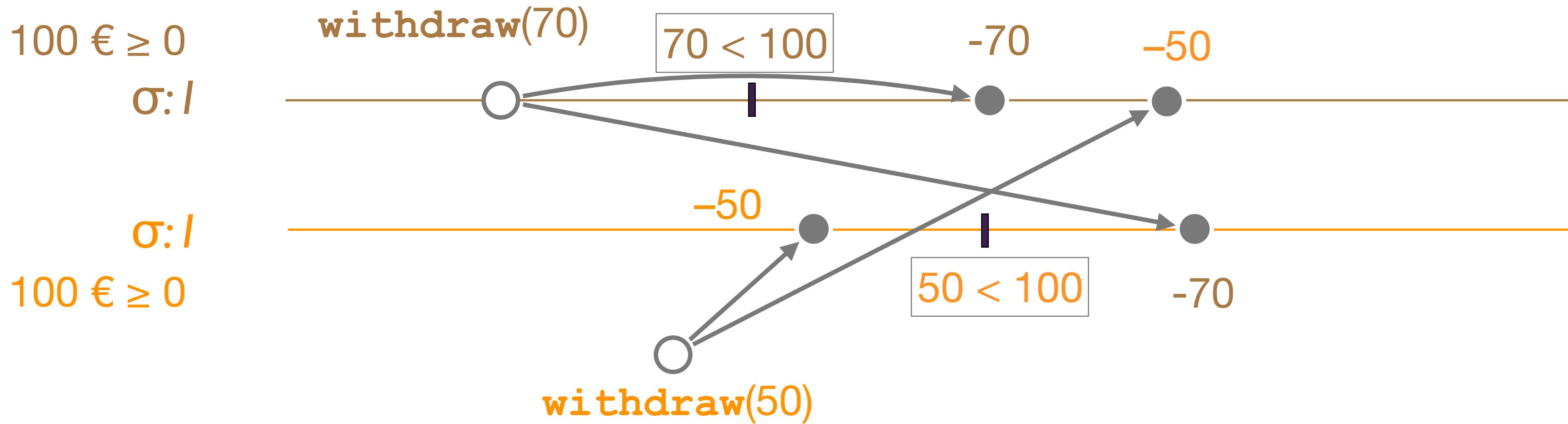




## CISE Rules

- 1: Sequential correctness
  - ▶ Single operations preserve the invariant
- 2: Convergence
  - ▶ Concurrent effectors commute
- 3: Precondition Stability
  - ▶ Preconditions are stable under concurrent operations

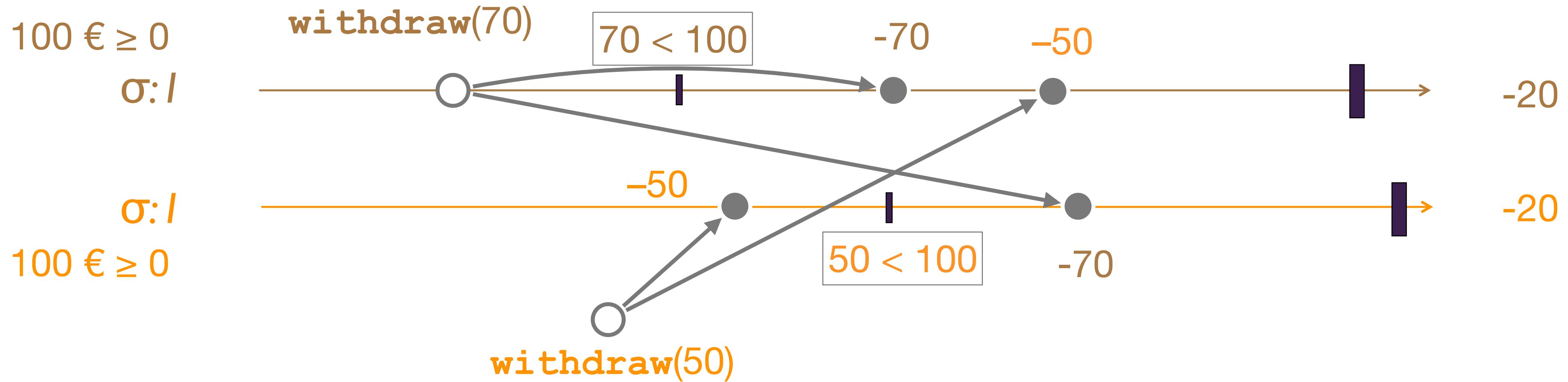
}  $\Rightarrow$  Invariant



## CISE Rules

- 1: Sequential correctness
  - ▶ Single operations preserve the invariant
- 2: Convergence
  - ▶ Concurrent effectors commute
- 3: Precondition Stability
  - ▶ Preconditions are stable under concurrent operations

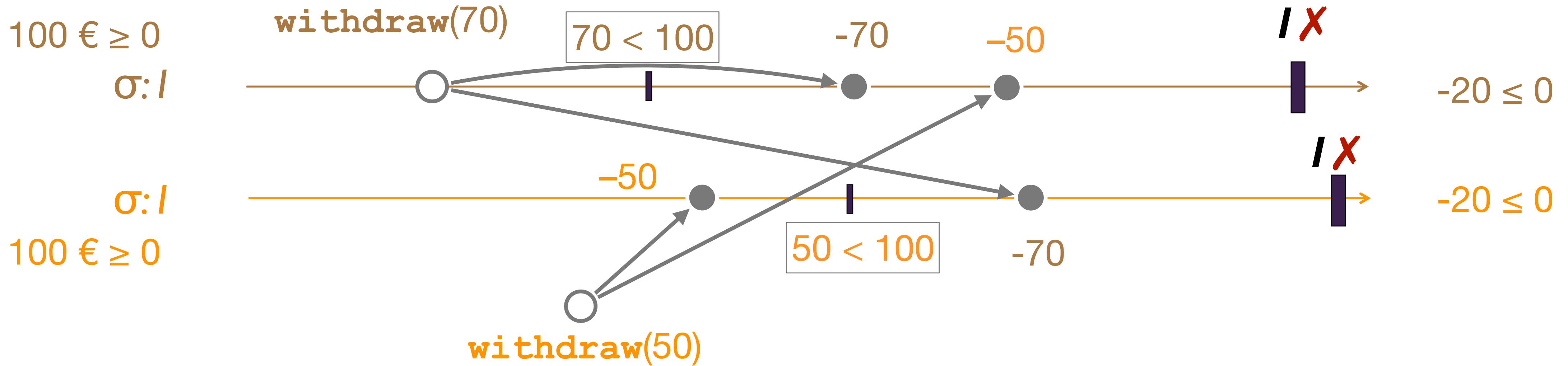
⇒ Invariant



## CISE Rules

- 1: Sequential correctness
  - ▶ Single operations preserve the invariant
- 2: Convergence
  - ▶ Concurrent effectors commute
- 3: Precondition Stability
  - ▶ Preconditions are stable under concurrent operations

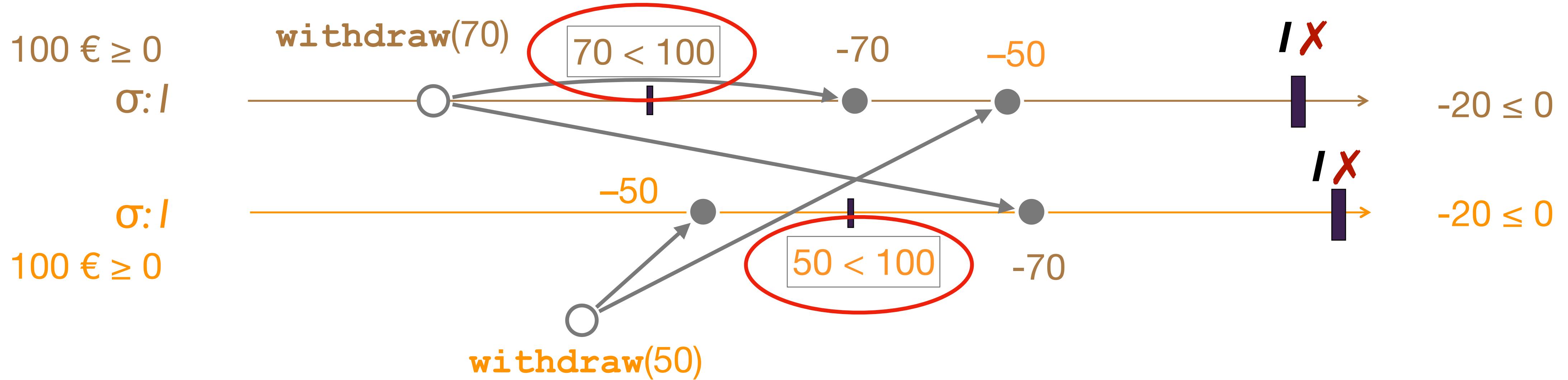
$\Rightarrow$  Invariant



## CISE Rules

- 1: Sequential correctness
  - ▶ Single operations preserve the invariant
- 2: Convergence
  - ▶ Concurrent effectors commute
- 3: Precondition Stability
  - ▶ Preconditions are stable under concurrent operations

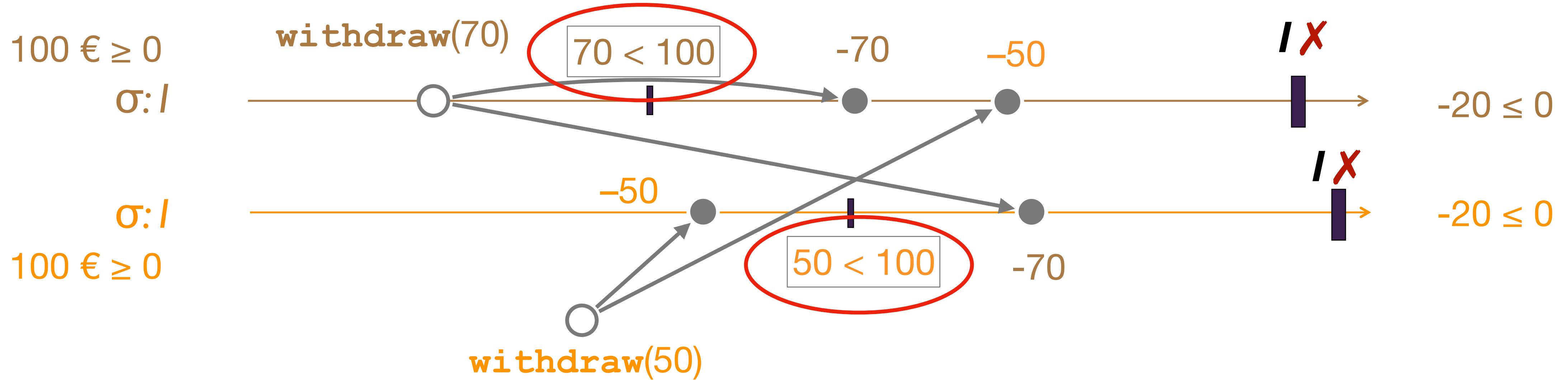
$\Rightarrow$  Invariant



## CISE Rules

- 1: Sequential correctness
  - ▶ Single operations preserve the invariant
- 2: Convergence
  - ▶ Concurrent effectors commute
- 3: Precondition Stability
  - ▶ Preconditions are stable under concurrent operations

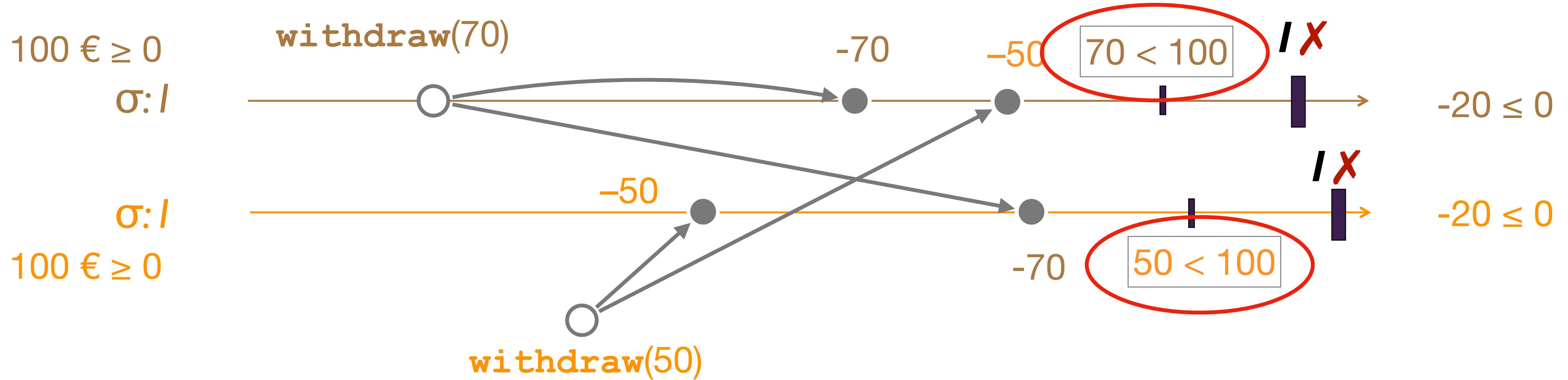
$\Rightarrow$  Invariant



## CISE Rules

- 1: Sequential correctness
  - ▶ Single operations preserve the invariant
- 2: Convergence
  - ▶ Concurrent effectors commute
- 3: Precondition Stability
  - ▶ Preconditions are stable under concurrent operations

$\Rightarrow$  Invariant



## CISE Rules

- 1: Sequential correctness
  - ▶ Single operations preserve the invariant
- 2: Convergence
  - ▶ Concurrent effectors commute
- 3: Precondition Stability
  - ▶ Preconditions are stable under concurrent operations

}  $\Rightarrow$  Invariant

# CISE: THE TOOL

Version of the tool (CEC) by Sreeja Nair

# Research Opportunities

- ▶ Beyond Simple Invariants
  - ▶ Pre/Post conditions of client programs using (1+) CRDTs
- ▶ Transactions + CRDTs
- ▶ Consistency Models: Eventual, Causal, Strong, ...
- ▶ Synchronization?
- ▶ ...

# 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?

# ACTORS FOR THE CLOUD

# ORLEANS (MS)

## Orleans – Virtual Actors

Established: October 14, 2010

*Project "Orleans" invented the **Virtual Actor abstraction**, which provides a straightforward approach to building distributed interactive applications, without the need to learn complex programming patterns for handling concurrency, fault tolerance, and resource management. Orleans applications scale-up automatically and are meant to be deployed in the cloud. It has been used heavily by a number of high-scale cloud services at Microsoft, starting with cloud services for the Halo franchise running in production in Microsoft Azure since 2011. The core Orleans technology was transferred to 343 Industries (<https://www.halowaypoint.com/>) and made available as open source in January 2015.*

The main research paper that describes Orleans Virtual Actors is [here](#).

### People



**Sergey Bykov**  
PRINCIPAL  
SOFTWARE ENG  
LEAD



**Phil Bernstein**  
Distinguished  
Scientist



**Sebastian  
Burckhardt**  
Researcher

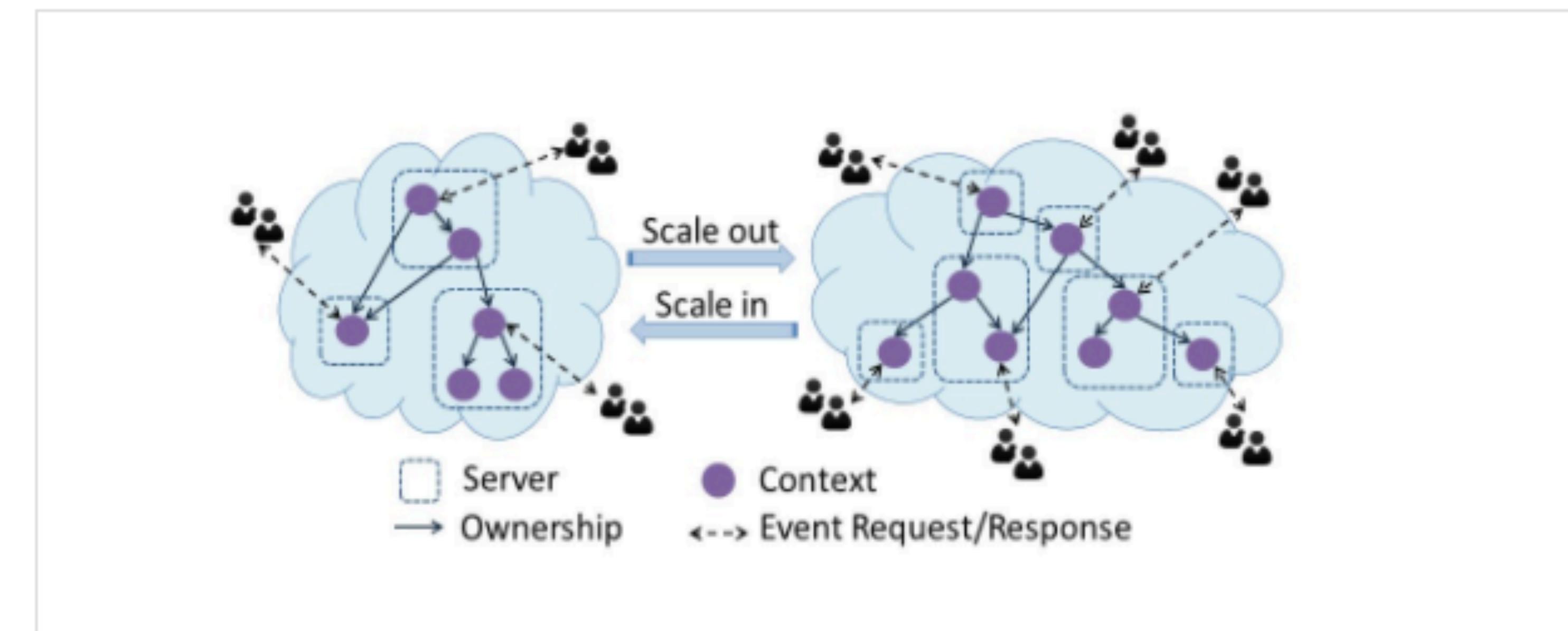
# TRANSACTIONAL

## AEON Project

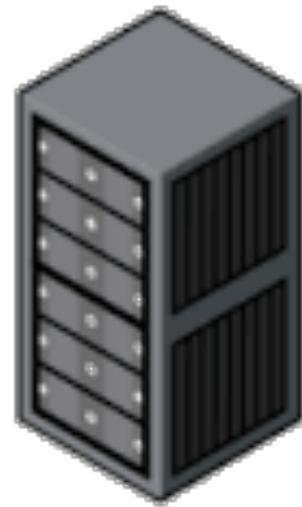


### Atomic Events and Ownership Network

Elastic programming model for cloud application

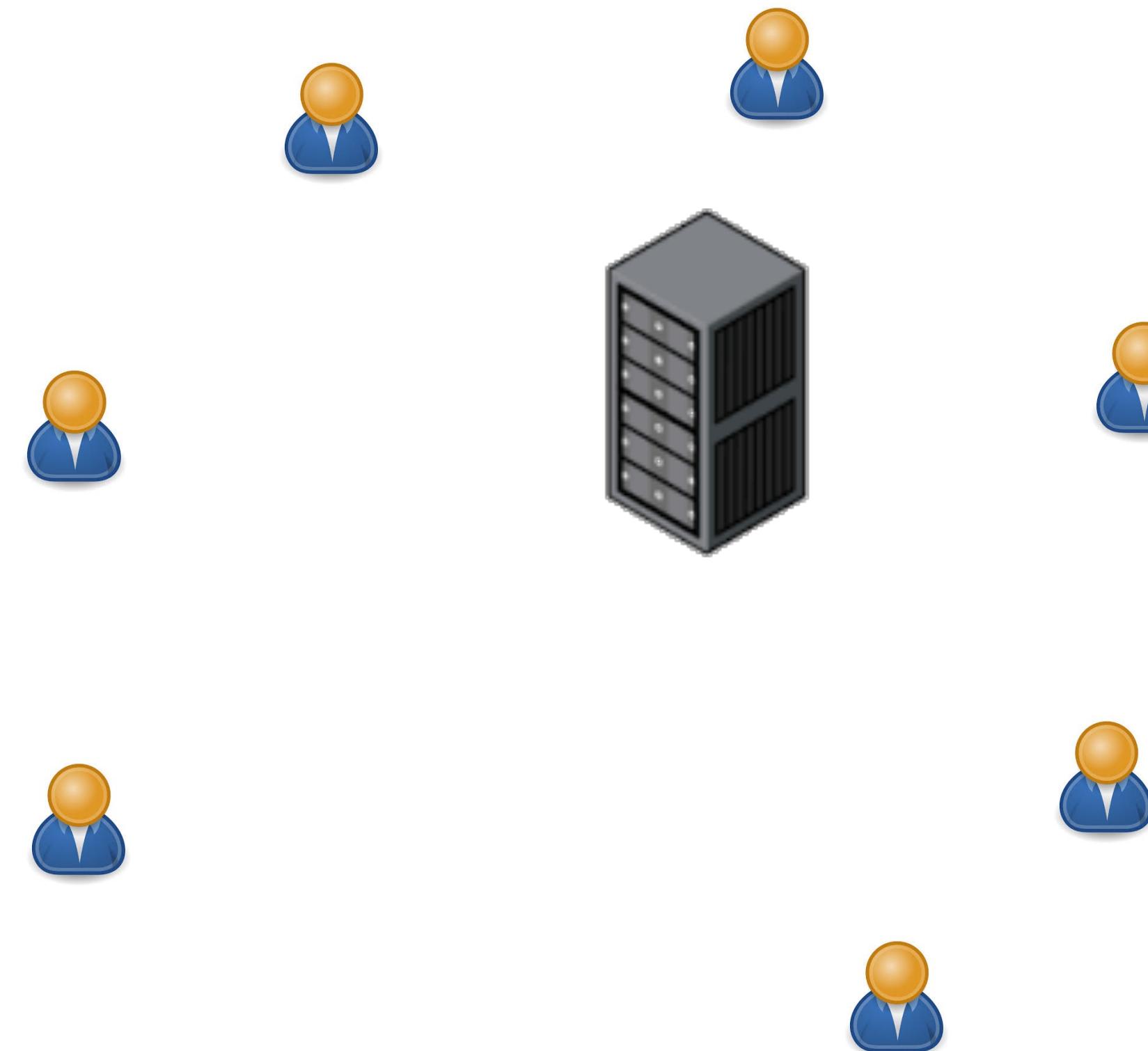


# PROGRAMMING FOR THE CLOUD: AEON



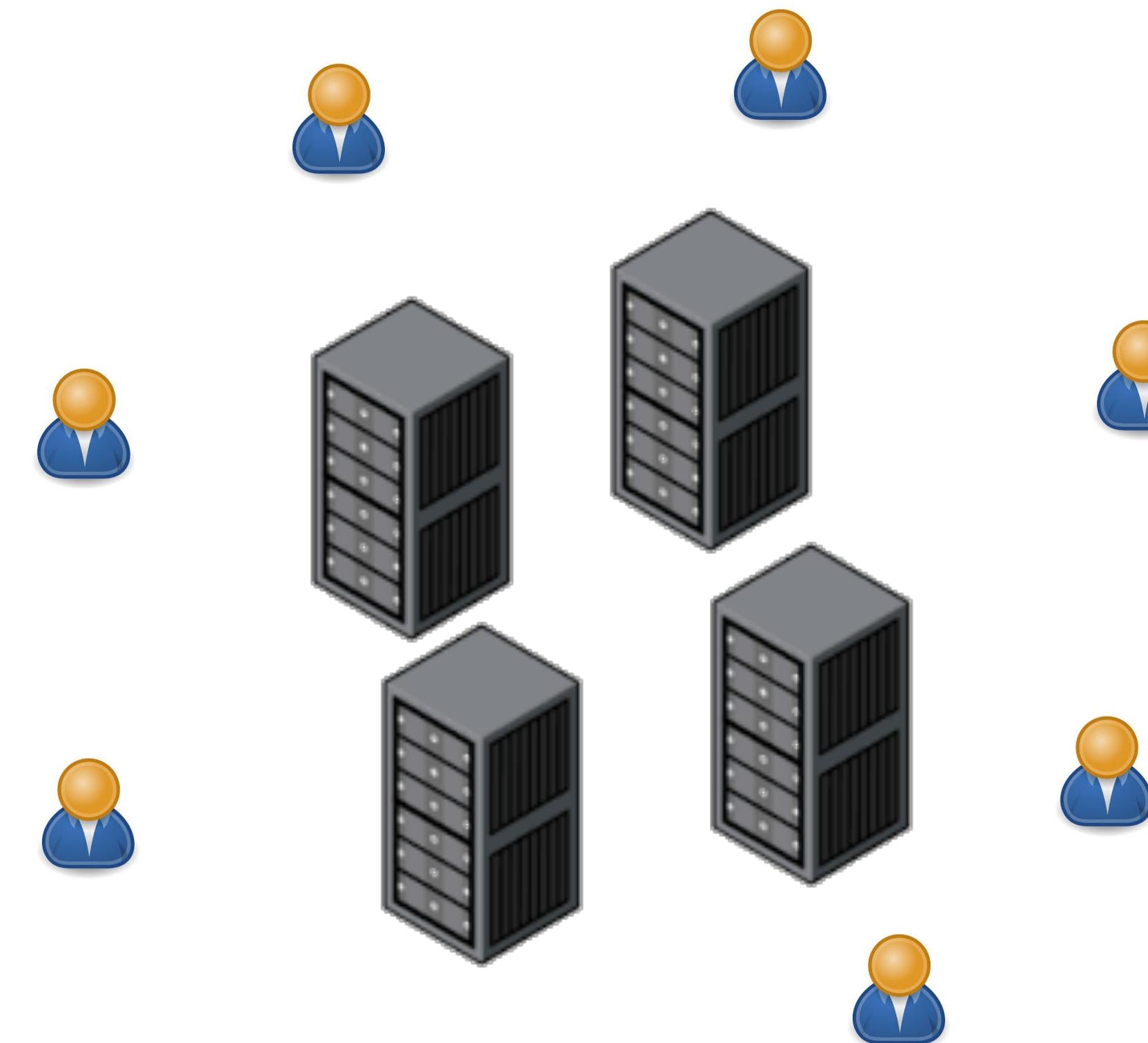
# PROGRAMMING FOR THE CLOUD: AEON

Scalability



# PROGRAMMING FOR THE CLOUD: AEON

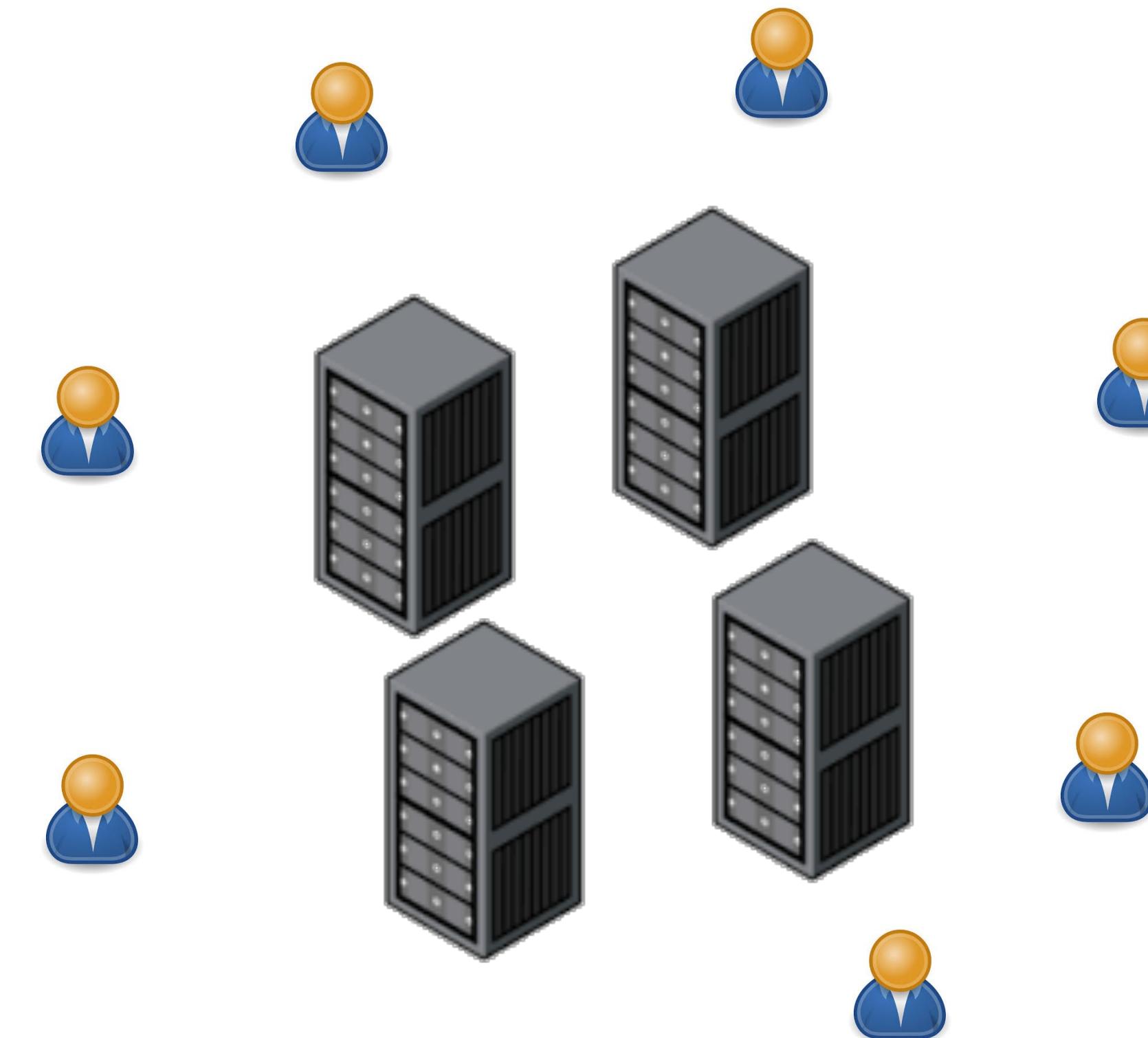
Scalability



# PROGRAMMING FOR THE CLOUD: AEON

Scalability

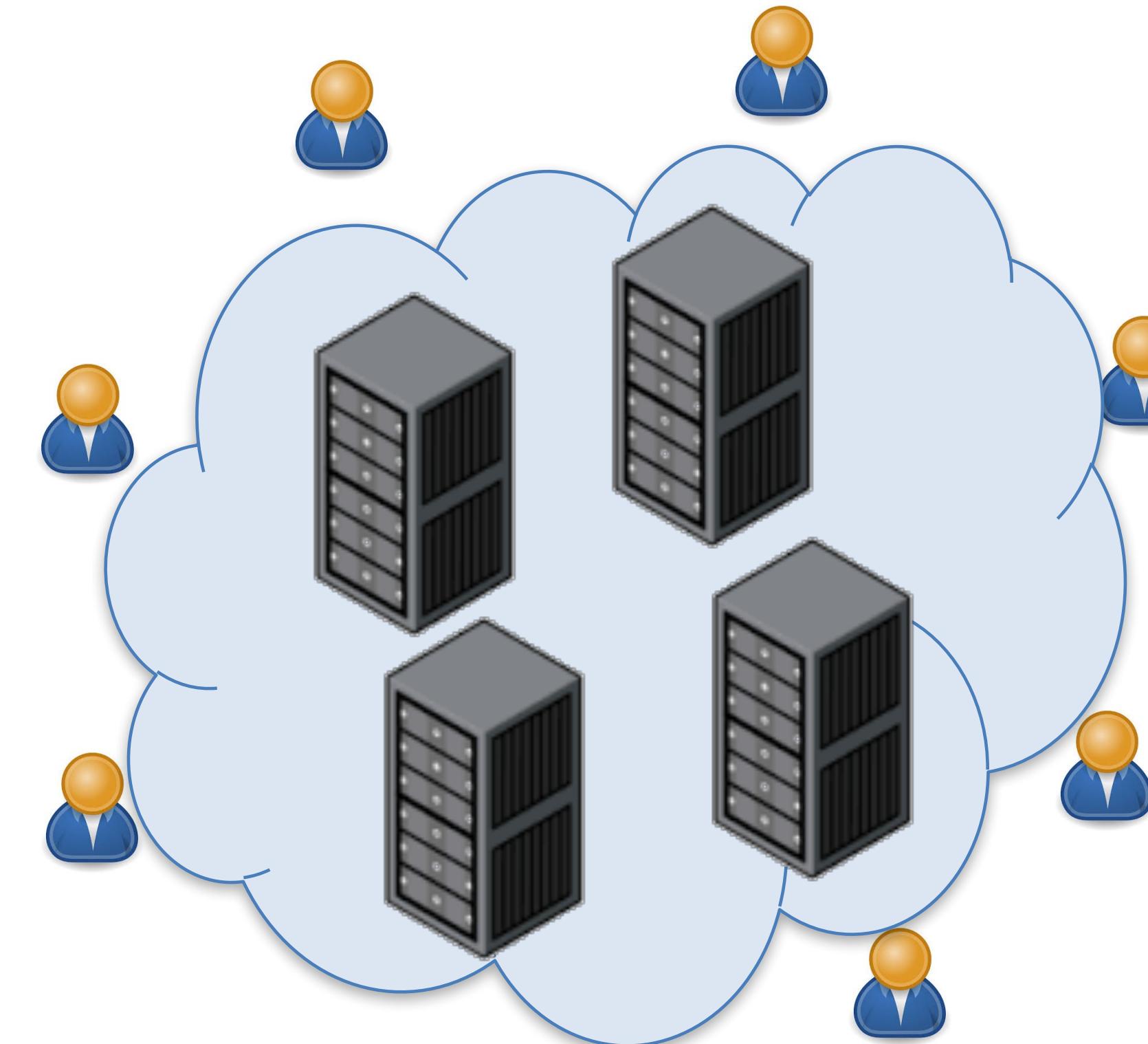
Cost Effectiveness



# PROGRAMMING FOR THE CLOUD: AEON

Scalability

Cost Effectiveness

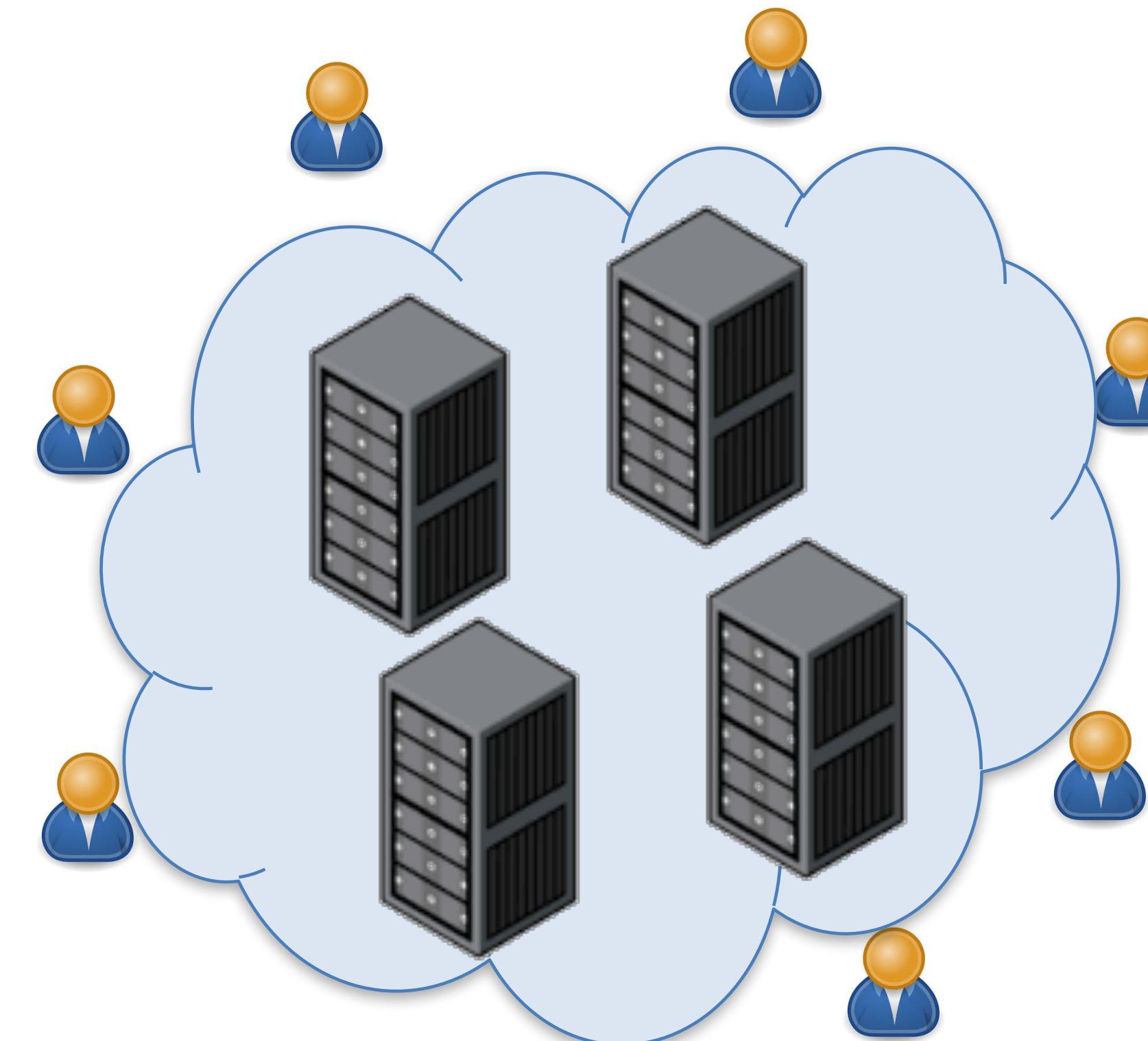


# PROGRAMMING FOR THE CLOUD: AEON

Scalability

Cost Effectiveness

Programmability

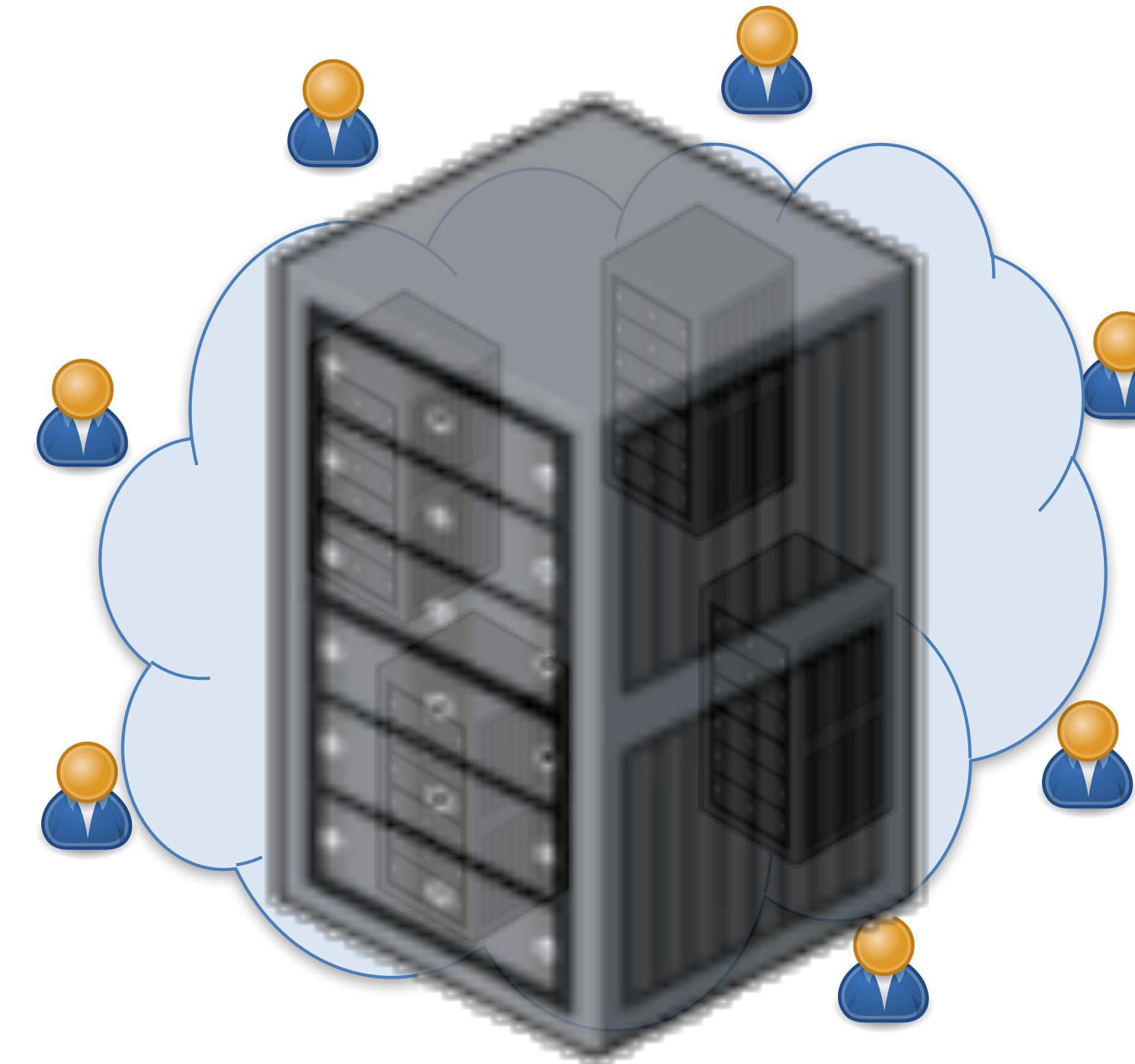


# PROGRAMMING FOR THE CLOUD: AEON

Scalability

Cost Effectiveness

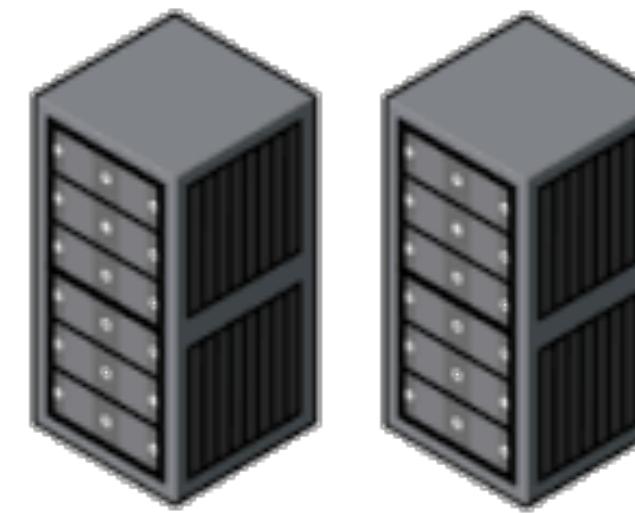
Programmability



# PROGRAMMING FOR THE CLOUD

Elasticity

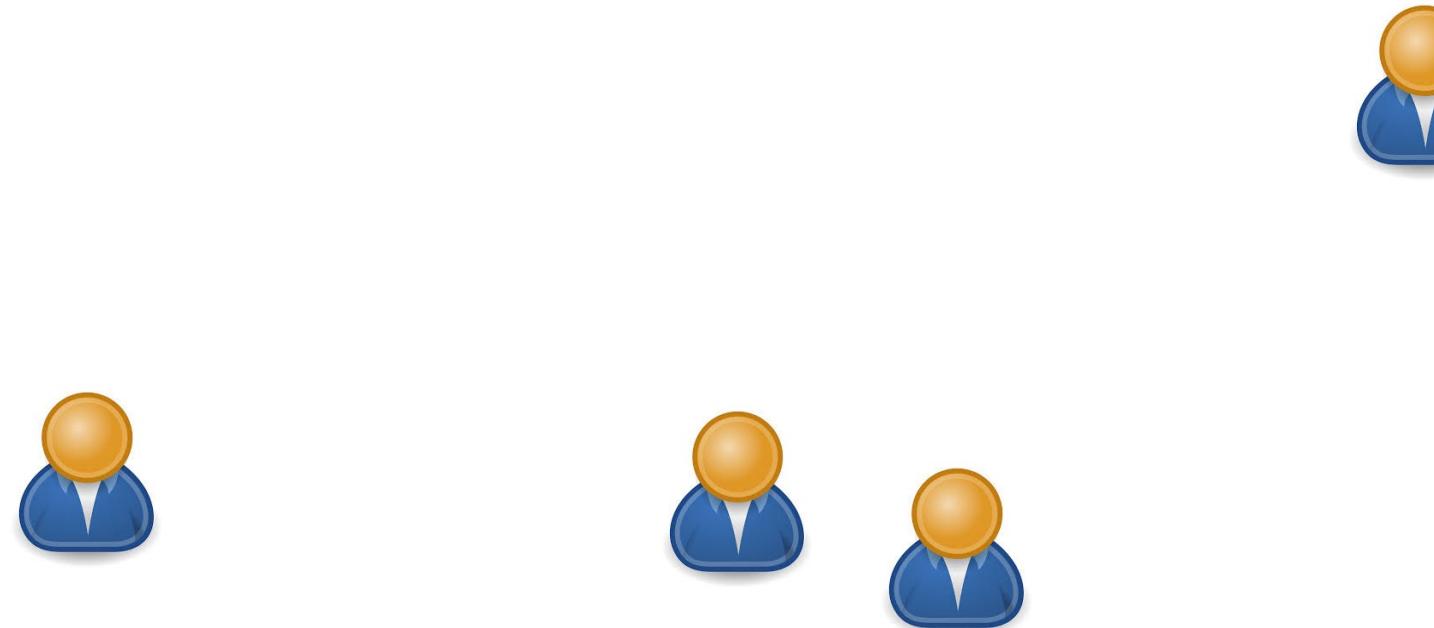
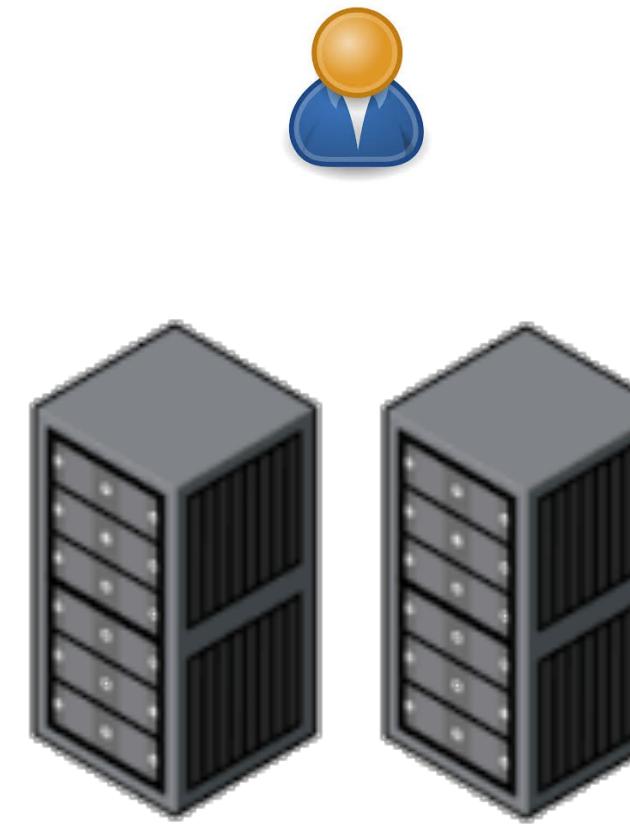
\$



# PROGRAMMING FOR THE CLOUD

Elasticity

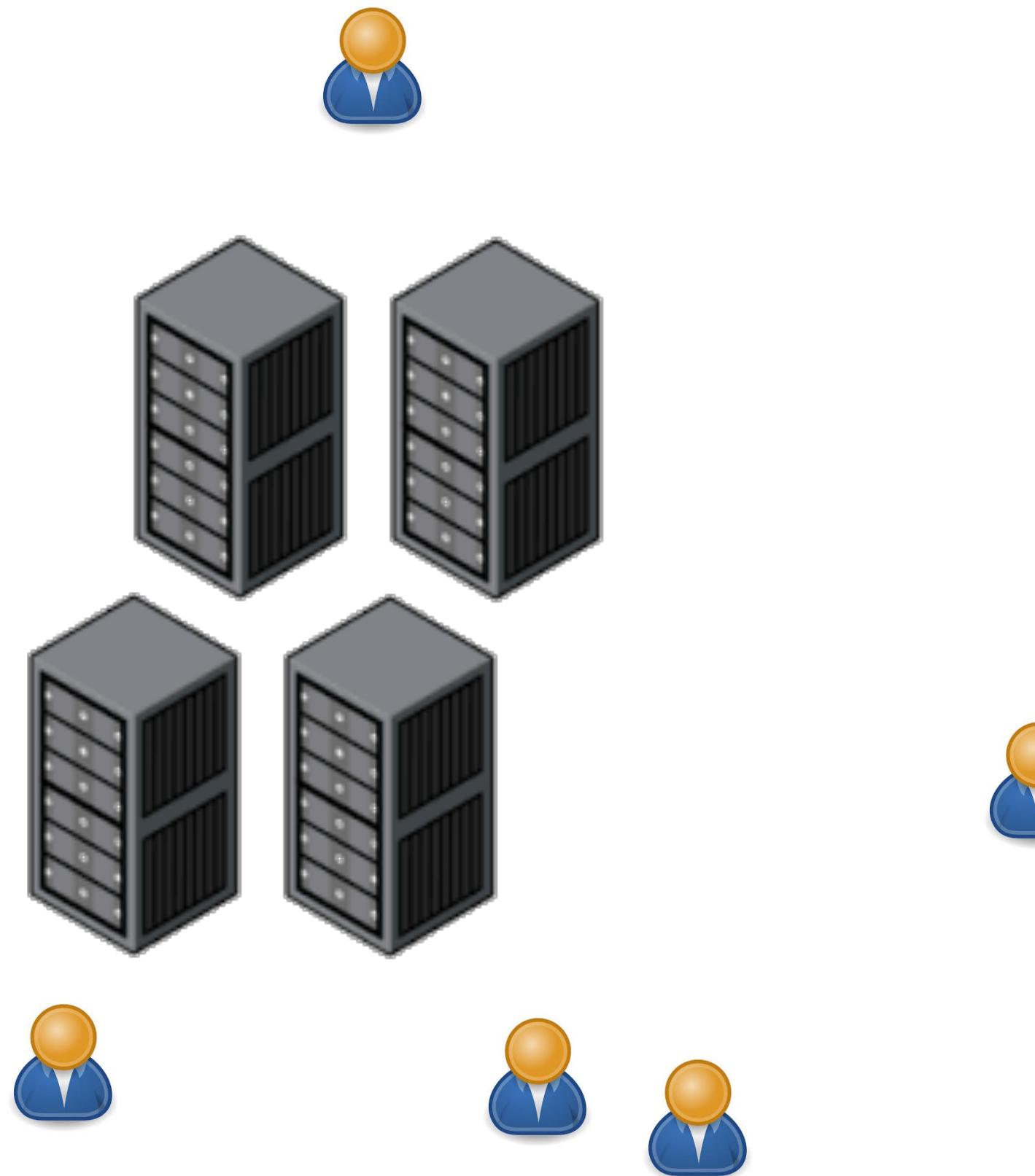
\$



# PROGRAMMING FOR THE CLOUD

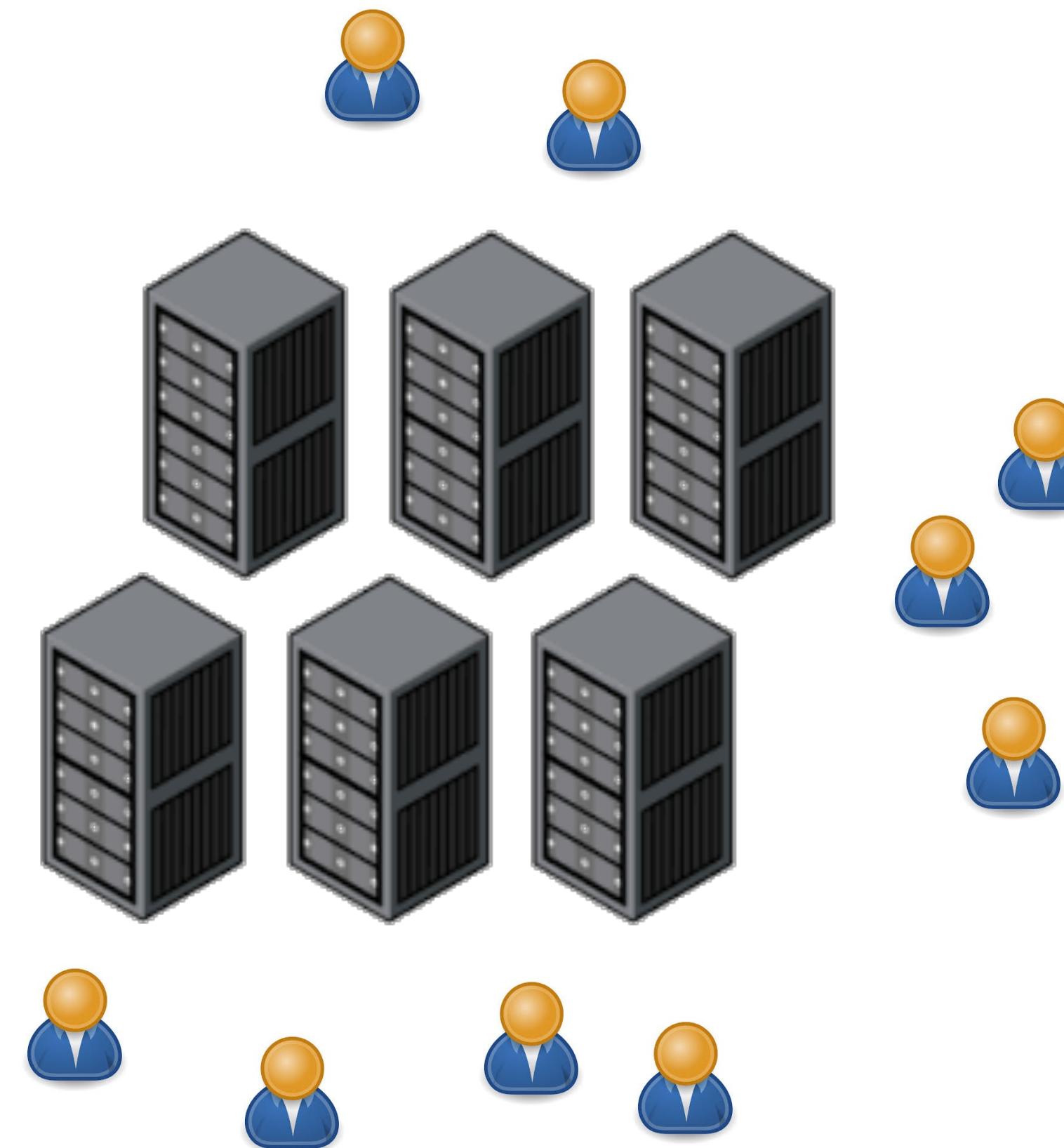
Elasticity

\$



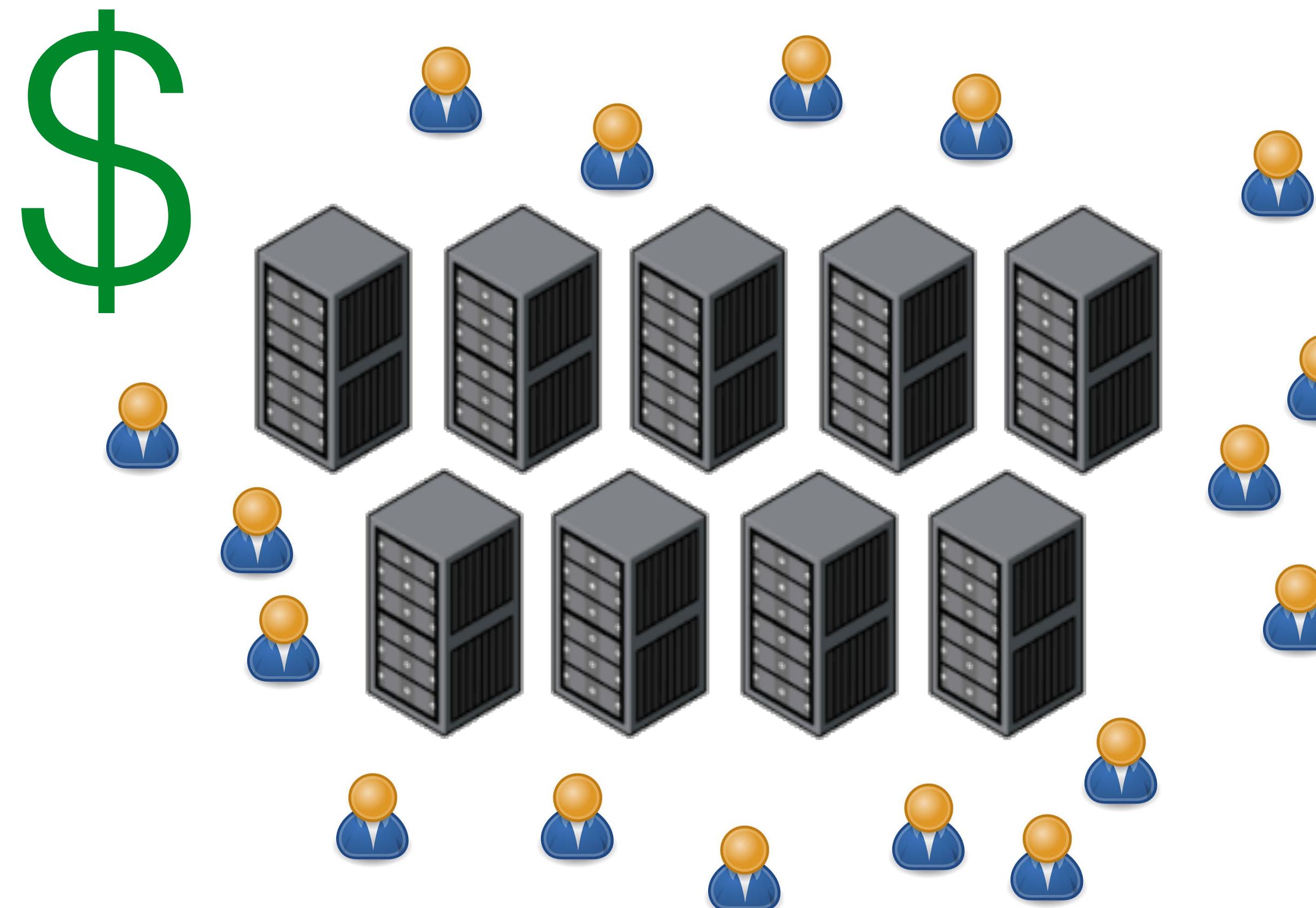
# PROGRAMMING FOR THE CLOUD

Elasticity



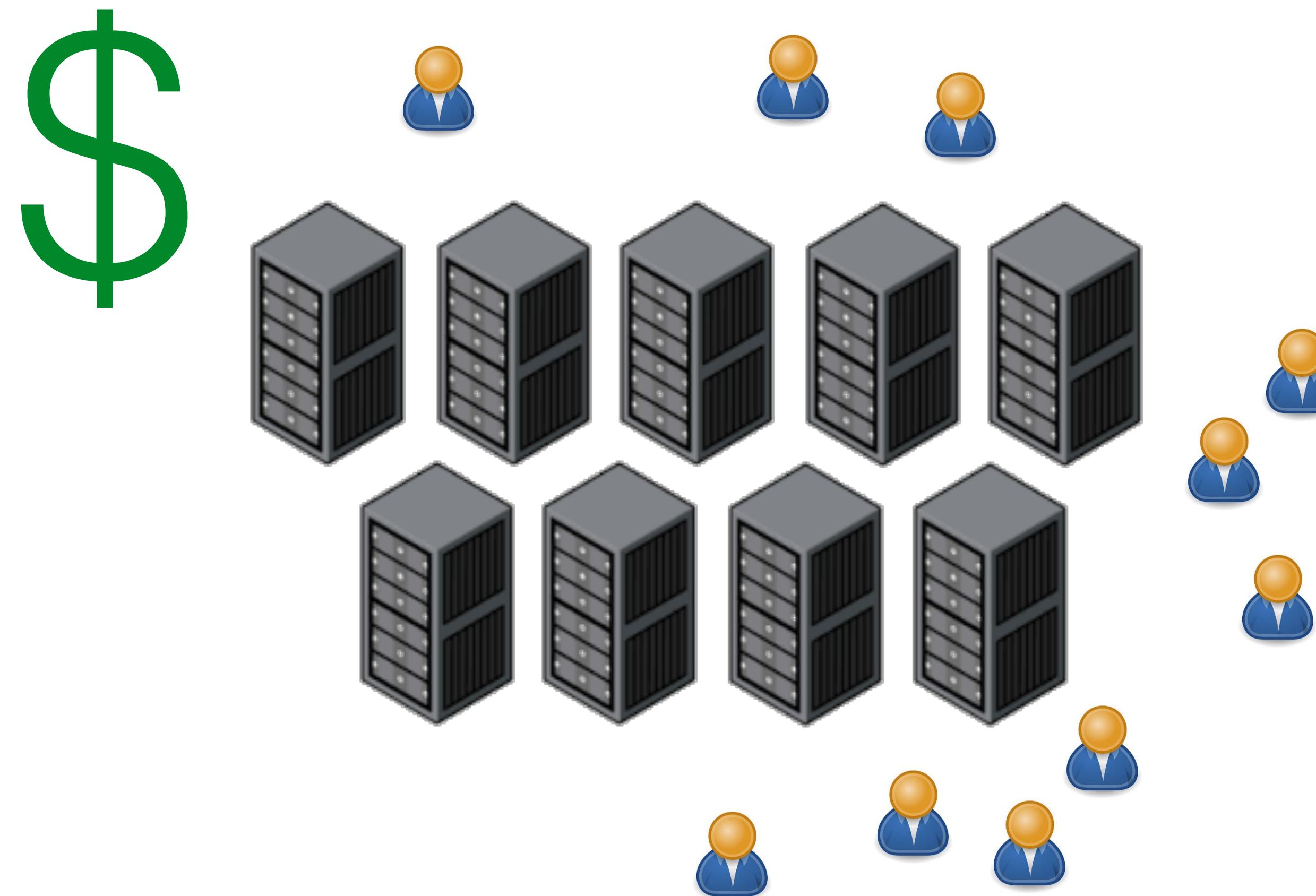
# PROGRAMMING FOR THE CLOUD

Elasticity



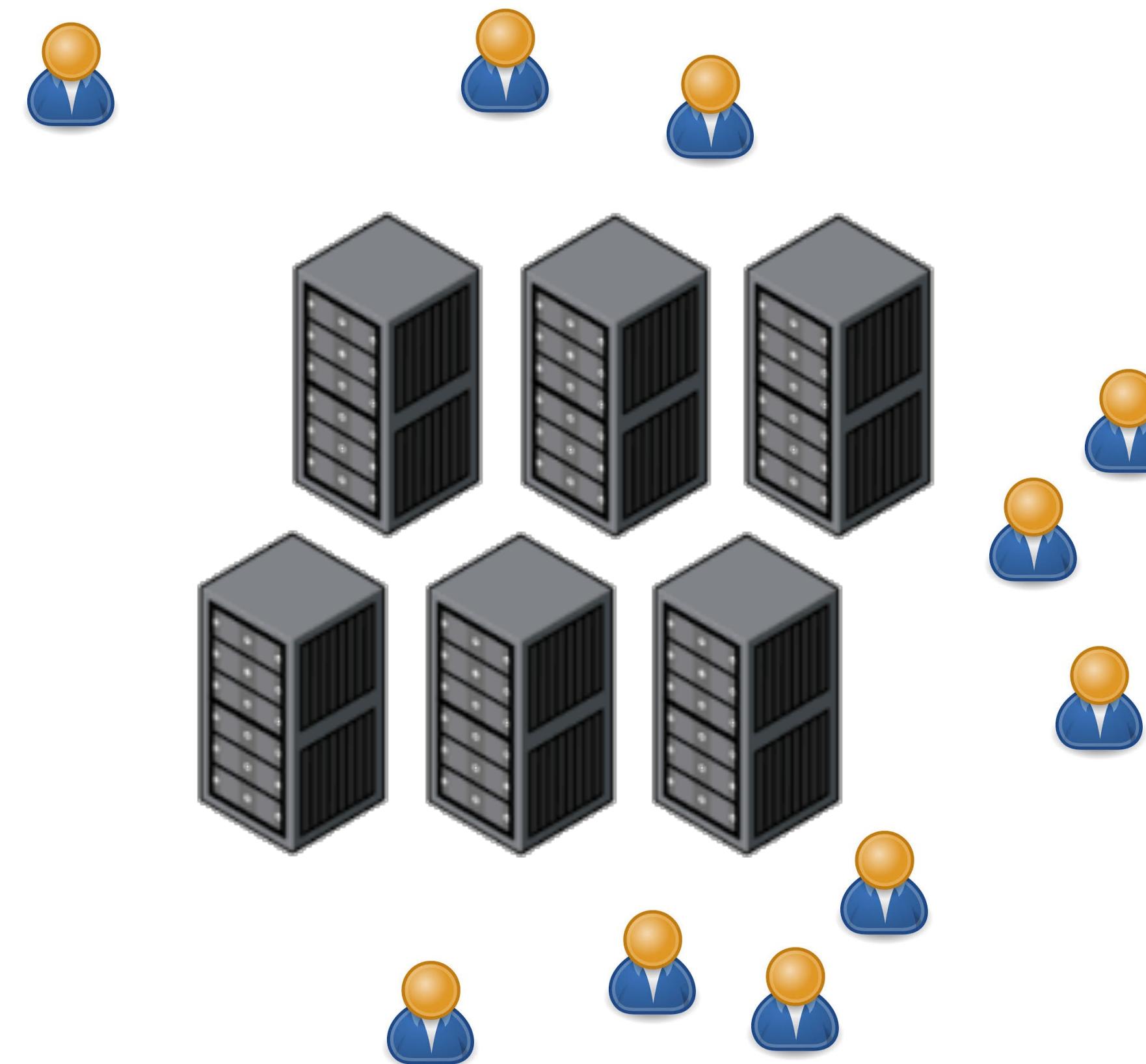
# PROGRAMMING FOR THE CLOUD

Elasticity



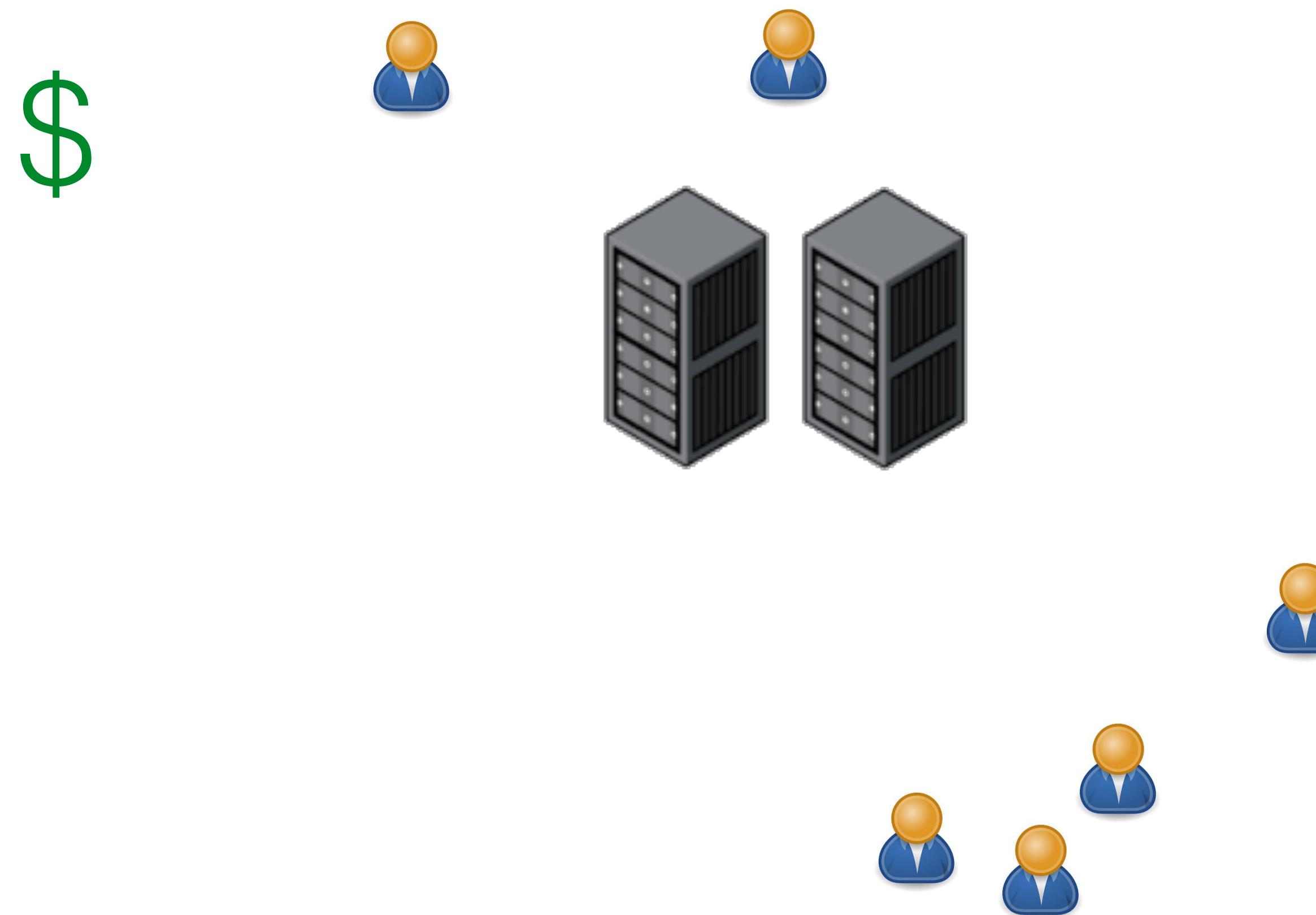
# PROGRAMMING FOR THE CLOUD

Elasticity

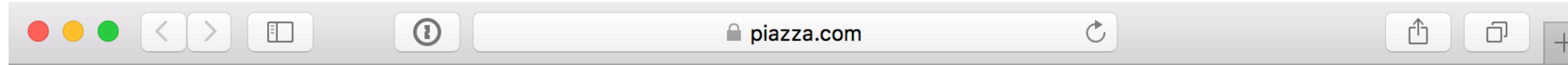


# PROGRAMMING FOR THE CLOUD

Elasticity

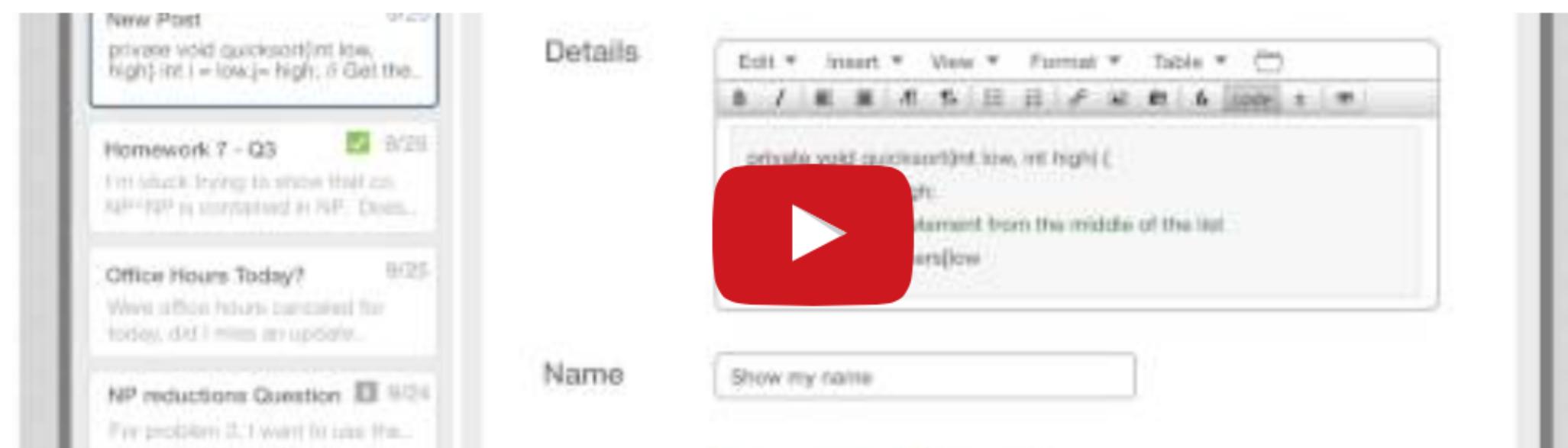


# PIAZZA IN AEON



The incredibly easy, completely free Q&A platform

Save time and help students learn using the power of community



- Wiki style format enables collaboration in a single space
- Features LaTeX editor, highlighted syntax and code blocking
- Questions and posts needing immediate action are highlighted
- Instructors endorse answers to keep the class on track
- Anonymous posting encourages every student to participate
- Highly customizable online polls

# PIAZZA IN AEON

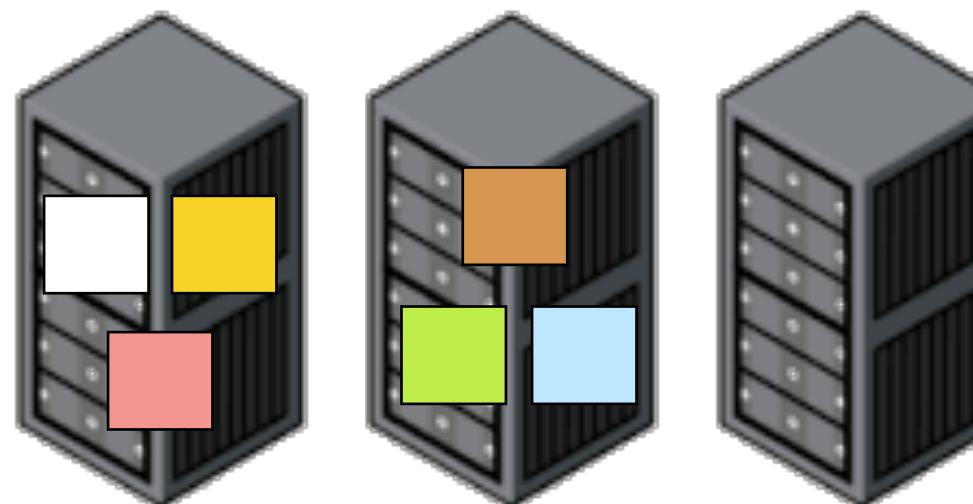
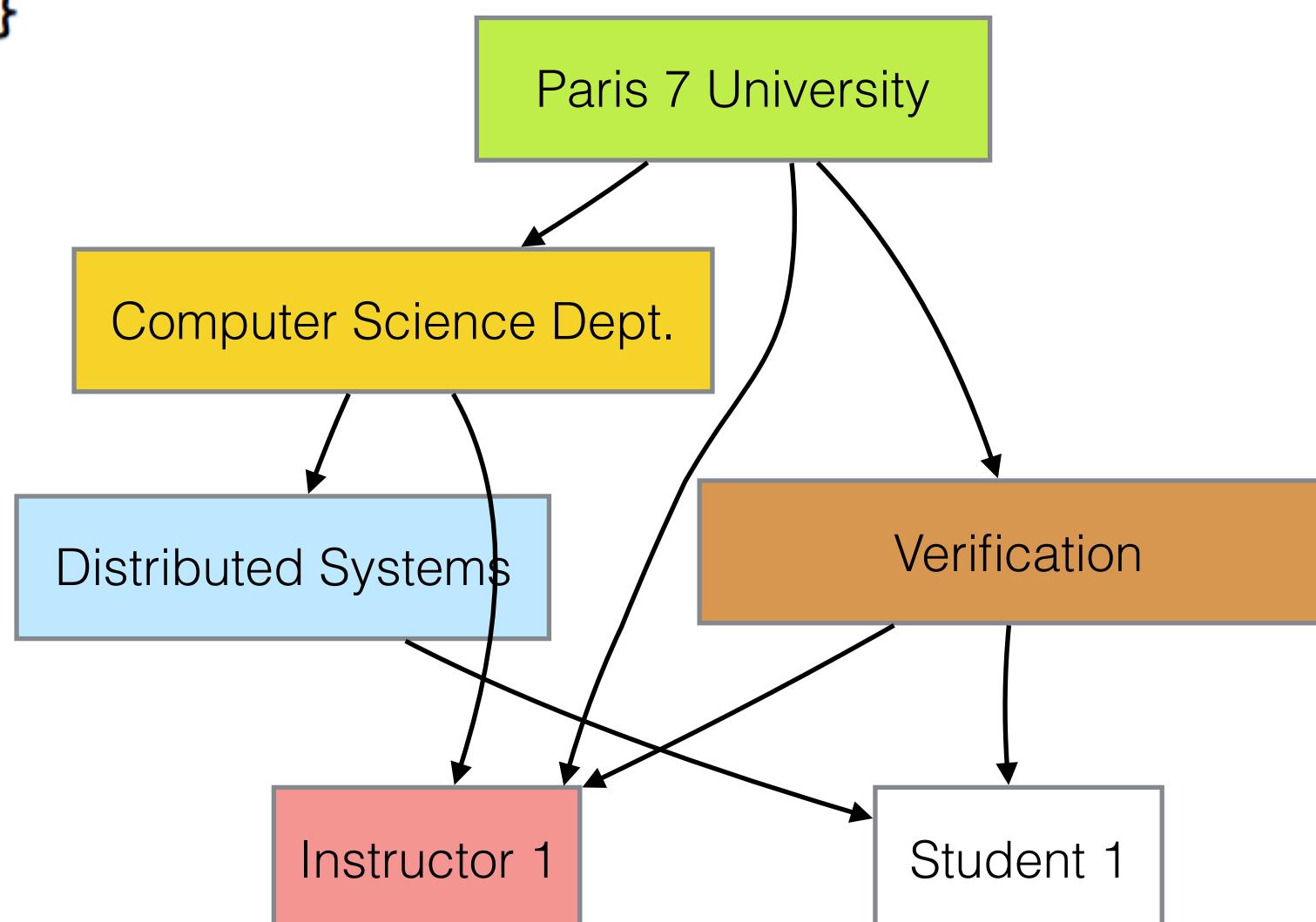
```
1 context Course events <snow_day, ...> {
2     vector<Person> students; // students
3     vector<Person> instructors; // instructor
4     ...
5     void snow_day() {
6         // update student tokens in parallel
7         for(int i=0 upto students.size())
8             async students[i]->add_tokens(1);
9         // notify all instructors atomically
10        for( int i=0 upto instructors.size())
11            instructors[i]->notify("Snow-Day");
12        ...
13    }
14 }
```

```
15 context Person events <use_token, ...> {
16     Role role; // staff or student
17     int tokens; // tokens
18     ...
19     void add_tokens(int n) {
20         tokens += n;
21     }
22     void use_token() {
23         tokens -= 1;
24     }
25     bool notify(String str) { ... }
26
27 }
28 class Role { ... }
```

# PIAZZA IN AEON

```
1  context Course events <snow_day, ...> {
2      vector<Person> students; // students
3      vector<Person> instructors; // instructor
4      ...
5      void snow_day() {
6          // update student tokens in parallel
7          for(int i=0 upto students.size())
8              async students[i]->add_tokens(1);
9          // notify all instructors atomically
10         for( int i=0 upto instructors.size())
11             instructors[i]->notify("Snow-Day");
12         ...
13     }
14 }
```

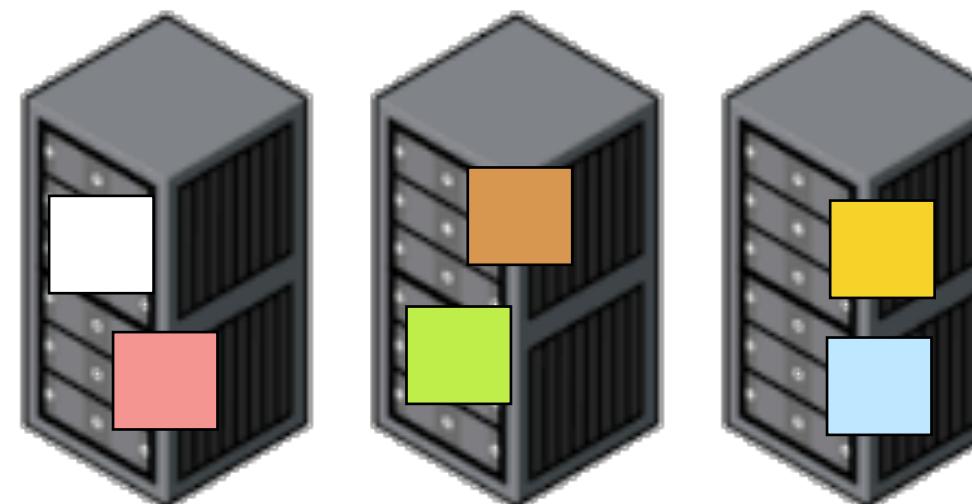
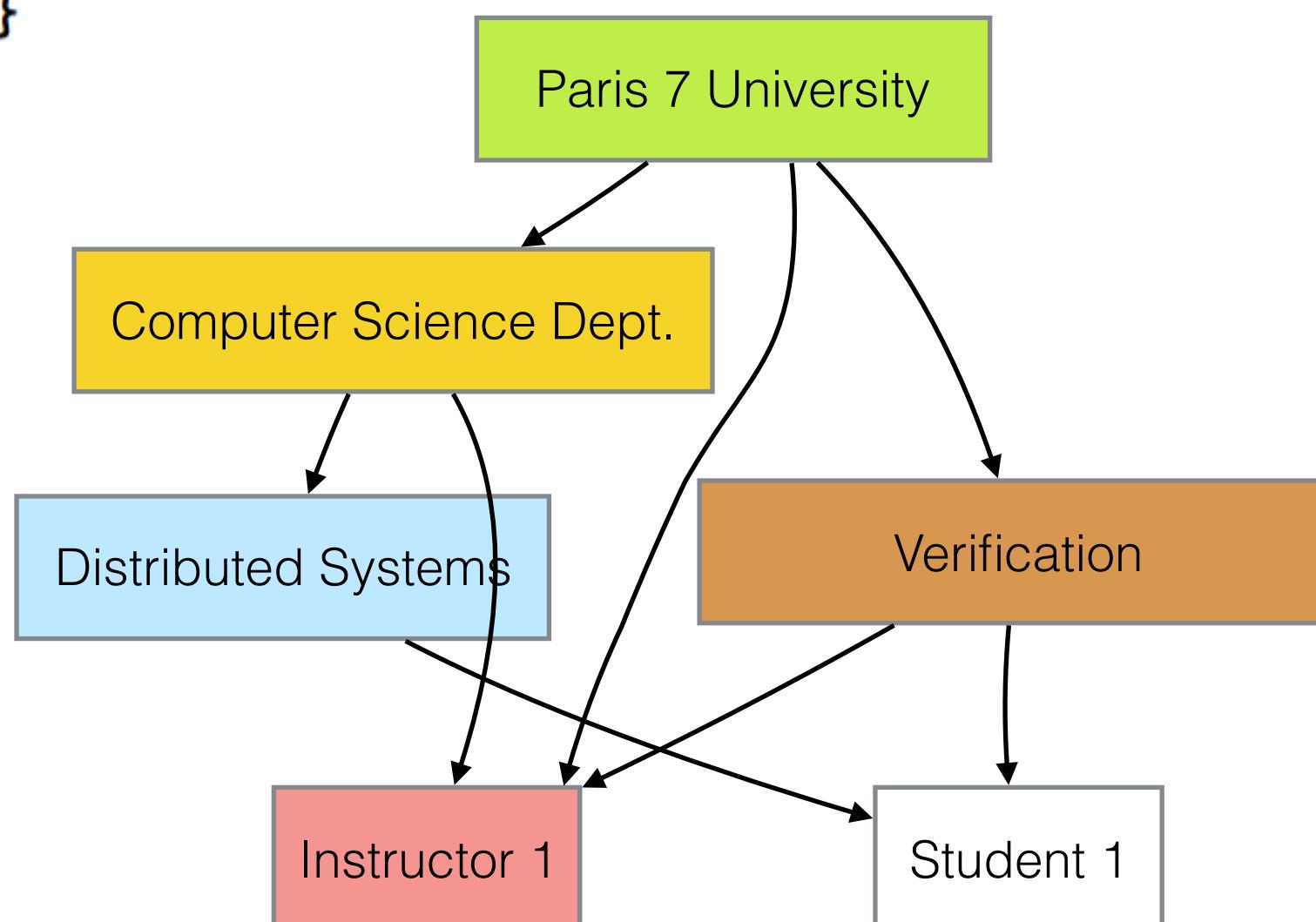
```
15 context Person events <use_token, ...> {
16     Role role; // staff or student
17     int tokens; // tokens
18     ...
19     void add_tokens(int n) {
20         tokens += n;
21     }
22     void use_token() {
23         tokens -= 1;
24     }
25     bool notify(String str) { ... }
26 }
27 }
28 class Role { ... }
```



# PIAZZA IN AEON

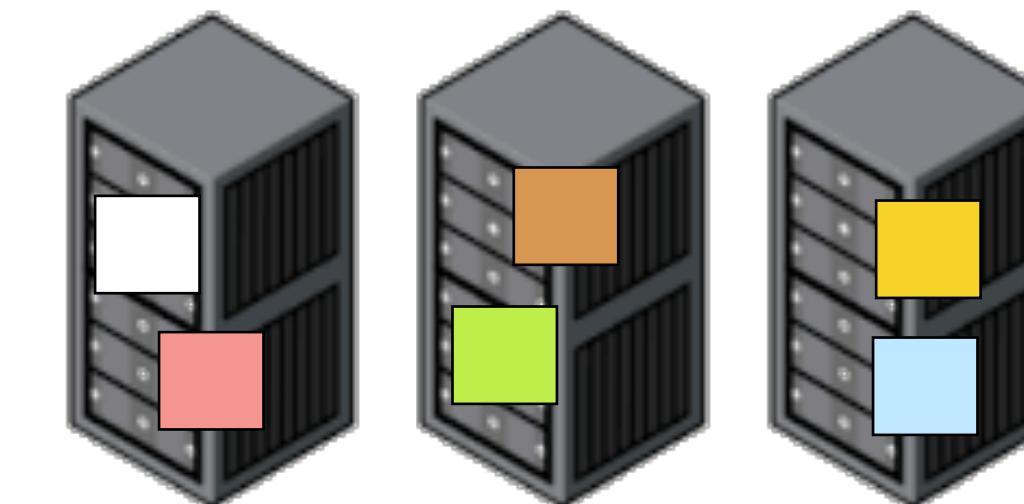
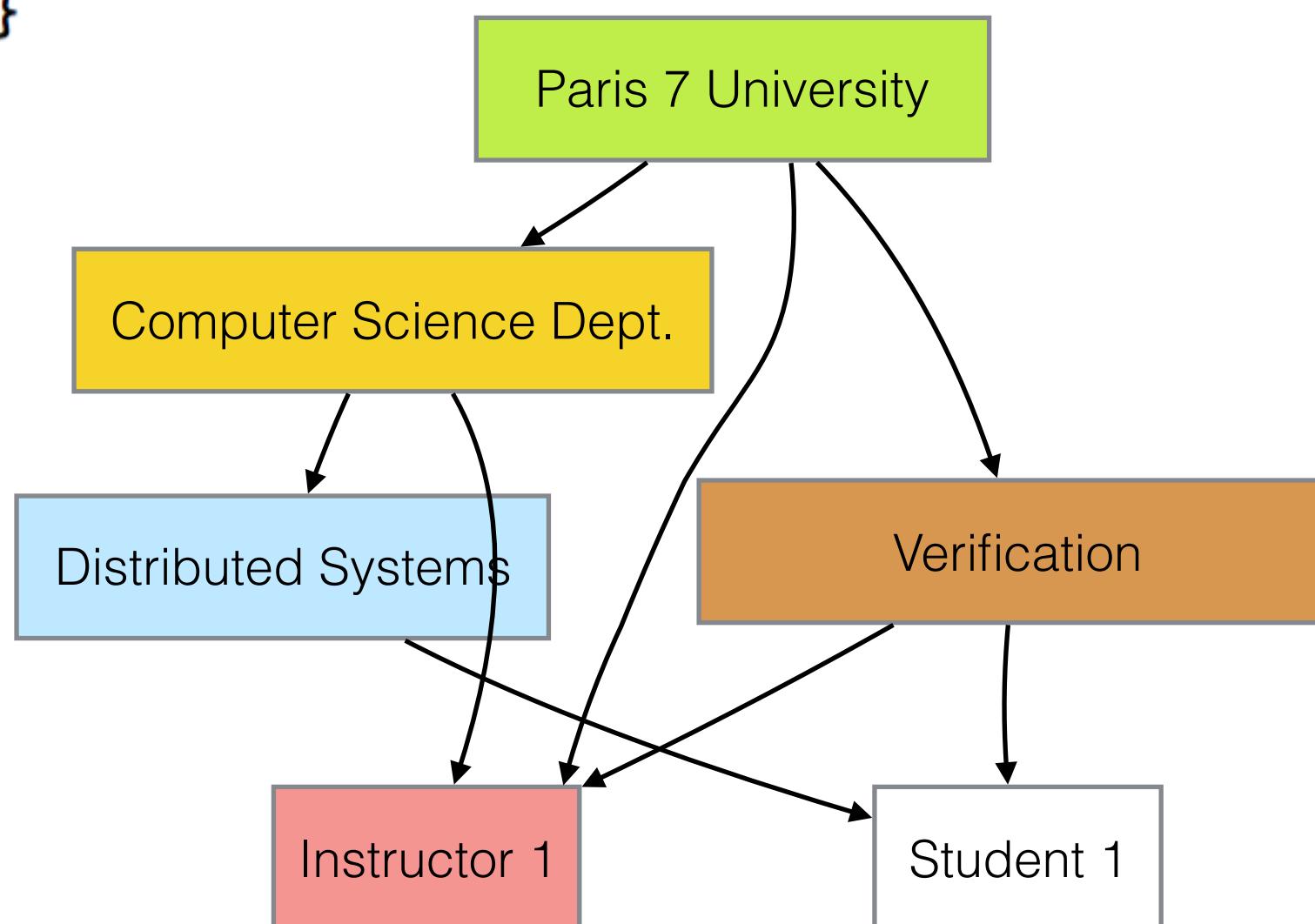
```
1  context Course events <snow_day, ...> {
2      vector<Person> students; // students
3      vector<Person> instructors; // instructor
4      ...
5      void snow_day() {
6          // update student tokens in parallel
7          for(int i=0 upto students.size())
8              async students[i]->add_tokens(1);
9          // notify all instructors atomically
10         for( int i=0 upto instructors.size())
11             instructors[i]->notify("Snow-Day");
12         ...
13     }
14 }
```

```
15 context Person events <use_token, ...> {
16     Role role; // staff or student
17     int tokens; // tokens
18     ...
19     void add_tokens(int n) {
20         tokens += n;
21     }
22     void use_token() {
23         tokens -= 1;
24     }
25     bool notify(String str) { ... }
26 }
27 }
28 class Role { ... }
```



# PIAZZA IN AEON

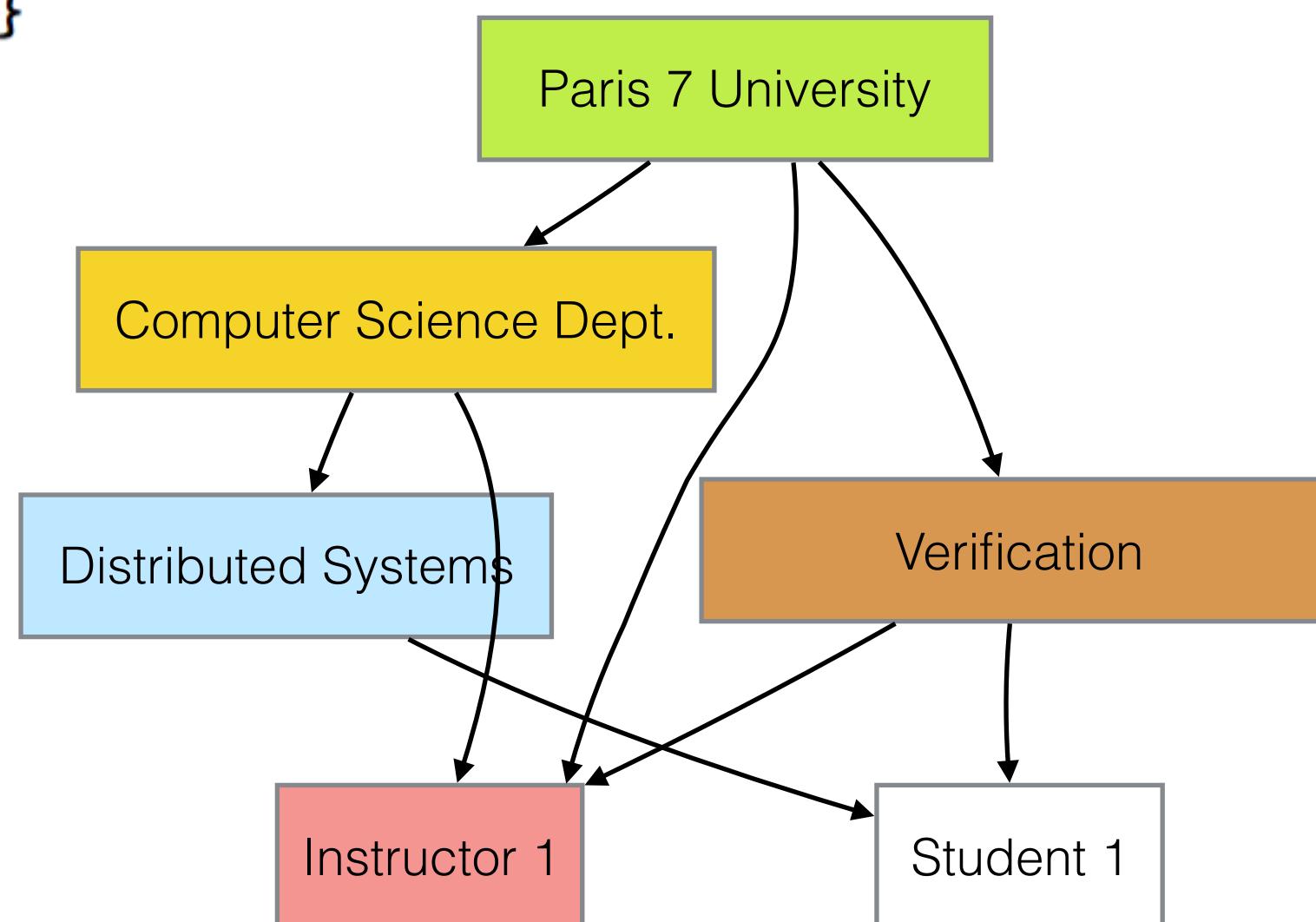
```
1  context Course events <snow_day, ...> {           15 context Person events <use_token, ...> {
2    vector<Person> students; // students             16 Role role; // staff or student
3    vector<Person> instructors; // instructor      17 int tokens; // tokens
4    ...
5    void snow_day() {
6      // update student tokens in parallel
7      for(int i=0 upto students.size())
8        async students[i]->add_tokens(1);
9      // notify all instructors atomically
10     for( int i=0 upto instructors.size())
11       instructors[i]->notify("Snow-Day");
12     ...
13   }
14 }
```



# PIAZZA IN AEON

```
1 context Course events <snow_day, ...> {
2     vector<Person> students; // students
3     vector<Person> instructors; // instructor
4     ...
5     void snow_day() {
6         // update student tokens in parallel
7         for(int i=0 upto students.size())
8             async students[i]->add_tokens(1);
9         // notify all instructors atomically
10        for( int i=0 upto instructors.size())
11            instructors[i]->notify("Snow-Day");
12        ...
13    }
14 }
```

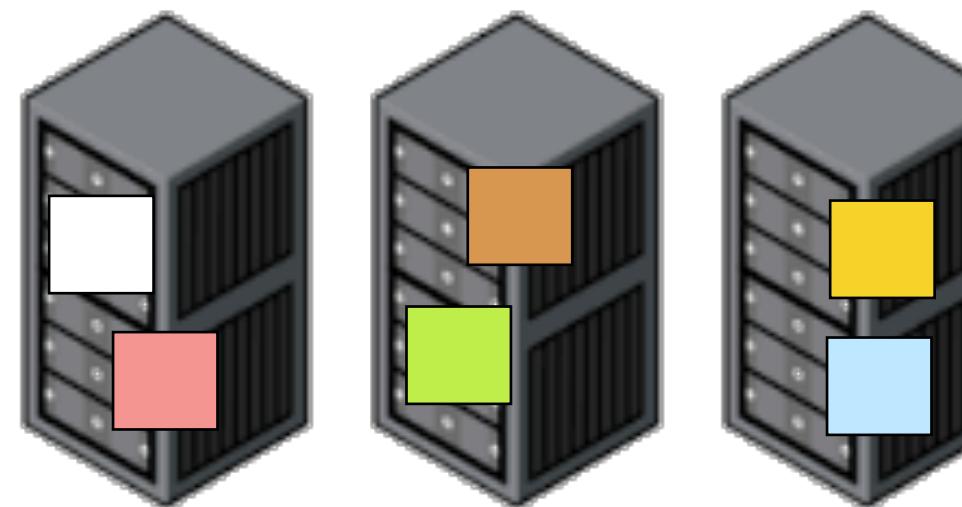
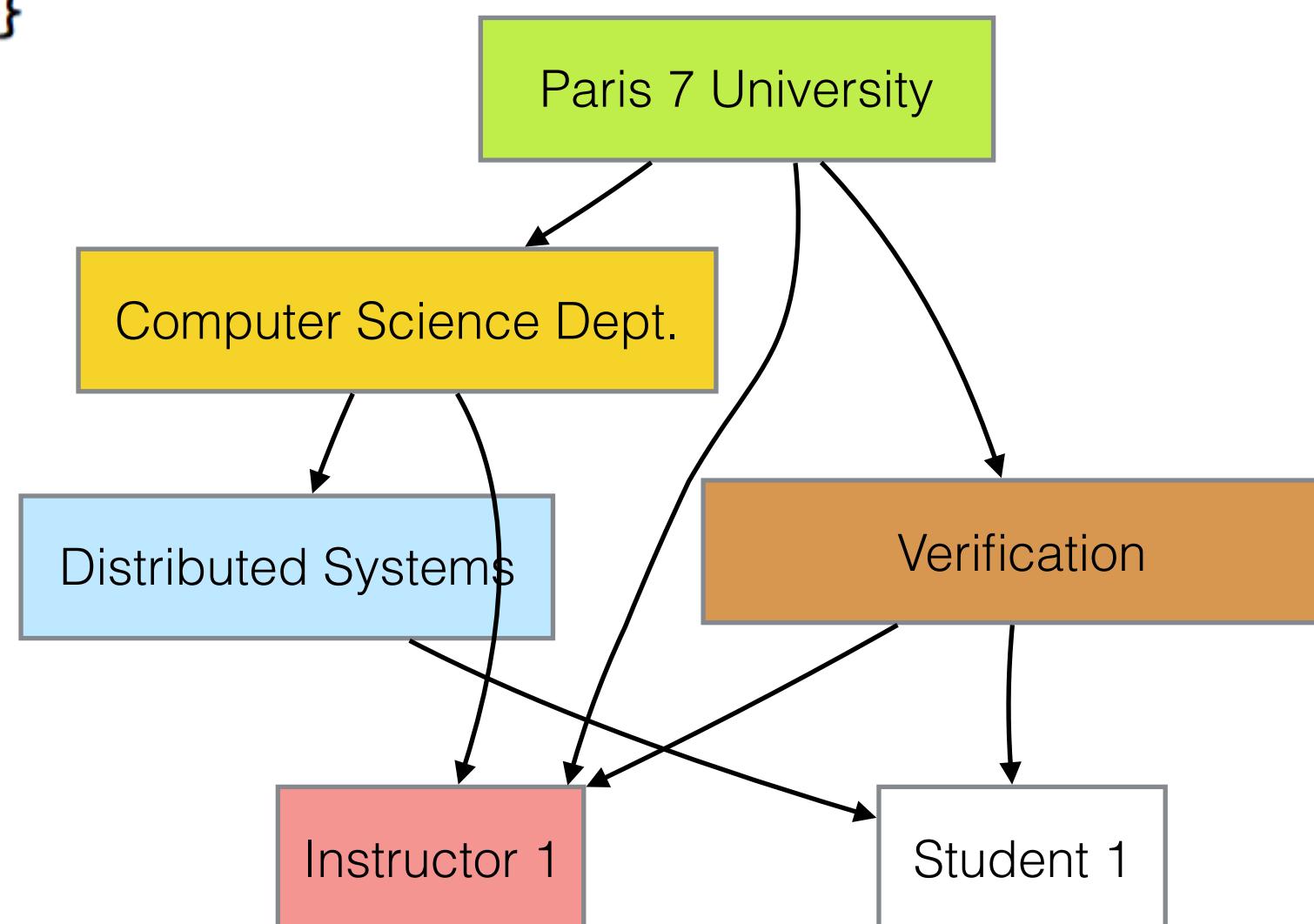
```
15 context Person events <use_token, ...> {
16     Role role; // staff or student
17     int tokens; // tokens
18     ...
19     void add_tokens(int n) {
20         tokens += n;
21     }
22     void use_token() {
23         tokens -= 1;
24     }
25     bool notify(String str) { ... }
26 }
27 class Role { ... }
```



# PIAZZA IN AEON

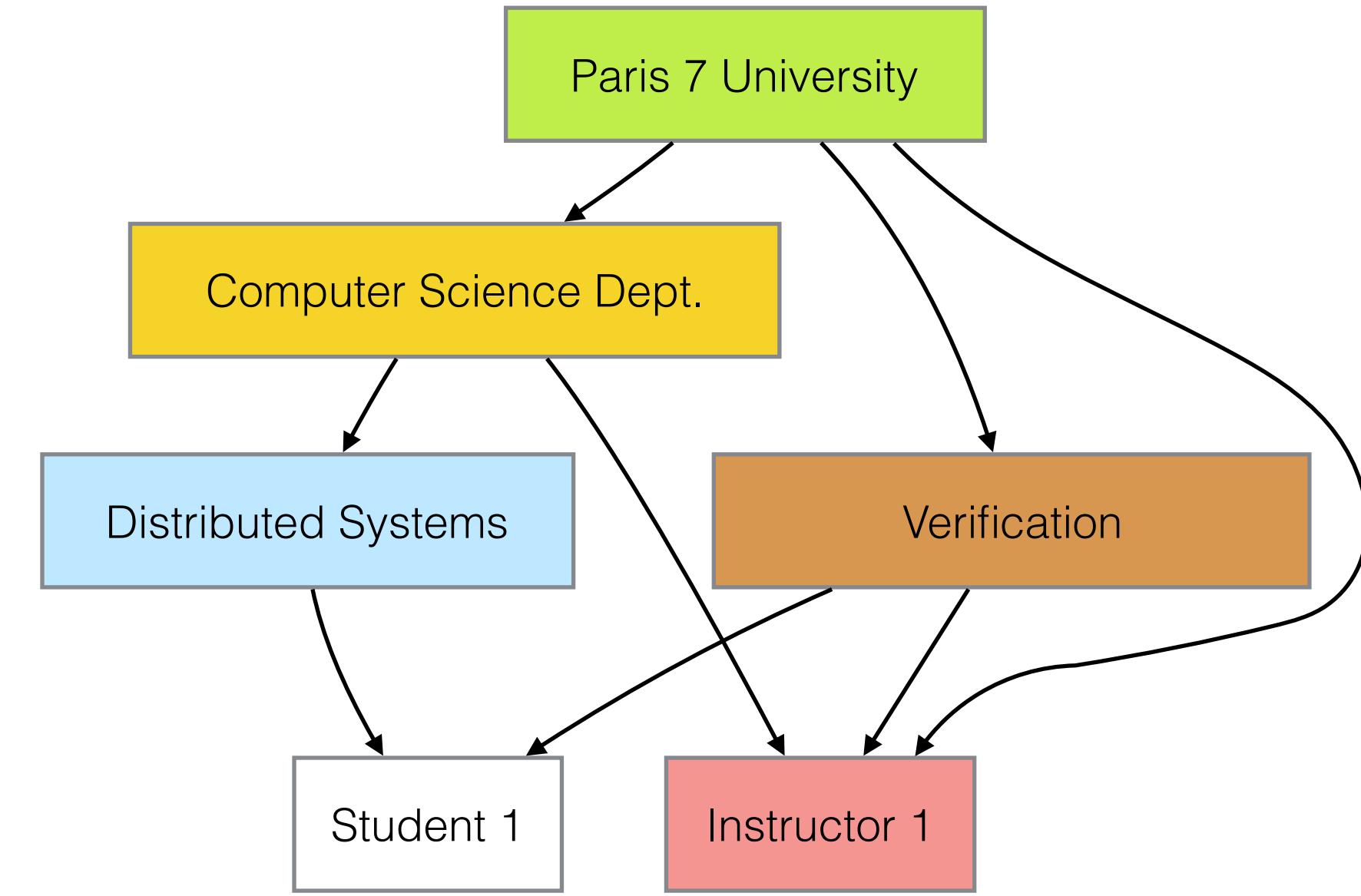
```
1 context Course events <snow_day, ...> {
2     vector<Person> students; // students
3     vector<Person> instructors; // instructor
4     ...
5     void snow_day() {
6         // update student tokens in parallel
7         for(int i=0 upto students.size())
8             async students[i]->add_tokens(1);
9         // notify all instructors atomically
10        for( int i=0 upto instructors.size())
11            instructors[i]->notify("Snow-Day");
12        ...
13    }
14 }
```

```
15 context Person events <use_token, ...> {
16     Role role; // staff or student
17     int tokens; // tokens
18     ...
19     void add_tokens(int n) {
20         tokens += n;
21     }
22     void use_token() {
23         tokens -= 1;
24     }
25     bool notify(String str) { ... }
26 }
27 class Role { ... }
```



# PROGRAMMING DISCIPLINE

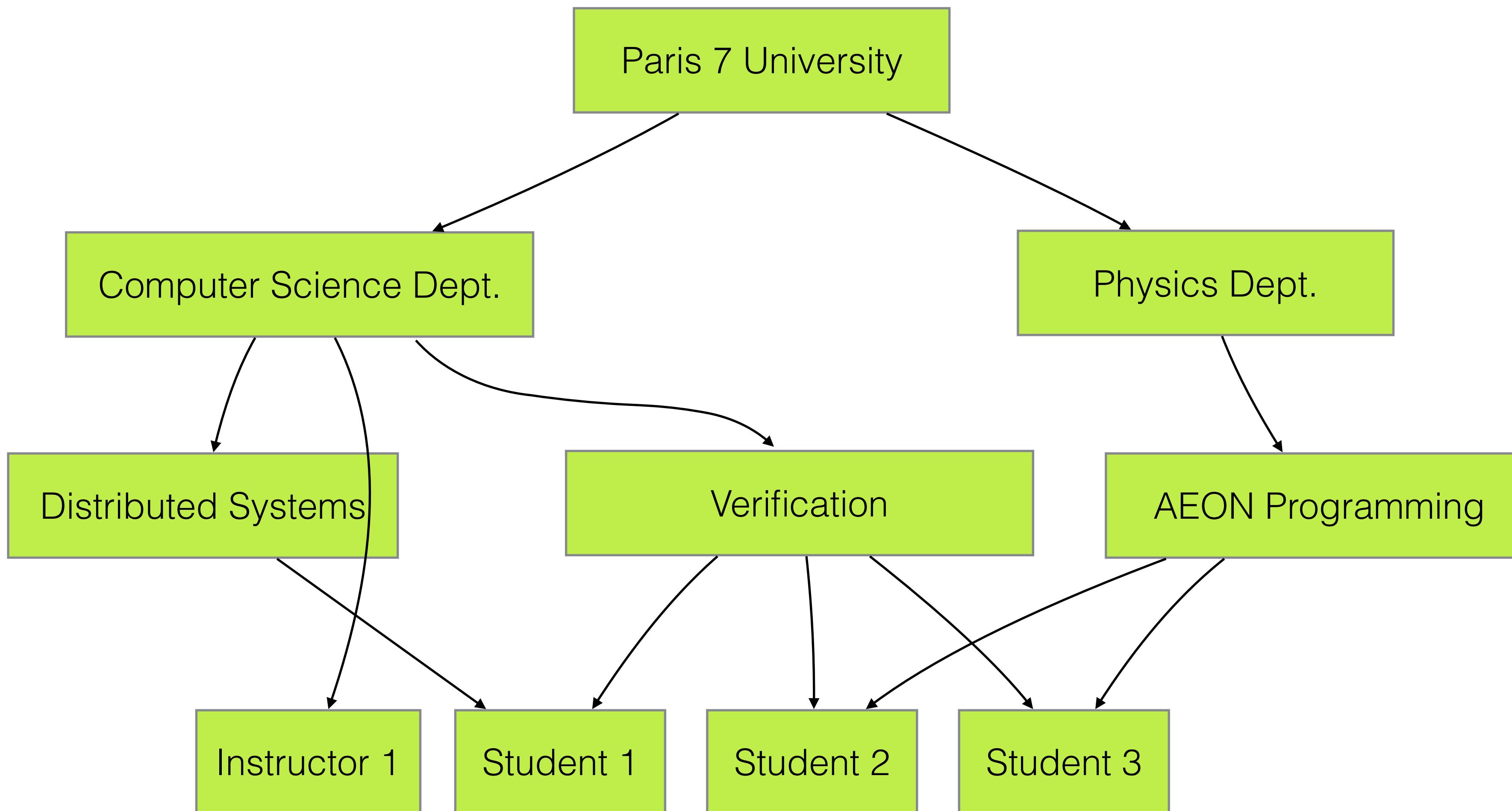
- ▶ Ownership for contexts
- ▶ Context graph is structured as a Directed Acyclic Graph (DAG)
  - ▶ Enforced by typing + runtime checks
- ▶ At runtime contexts are protected by locks
- ▶ Locks are taken top-down
- ▶ Events start at the top-most context they affect
  - ▶ “Conceptually”



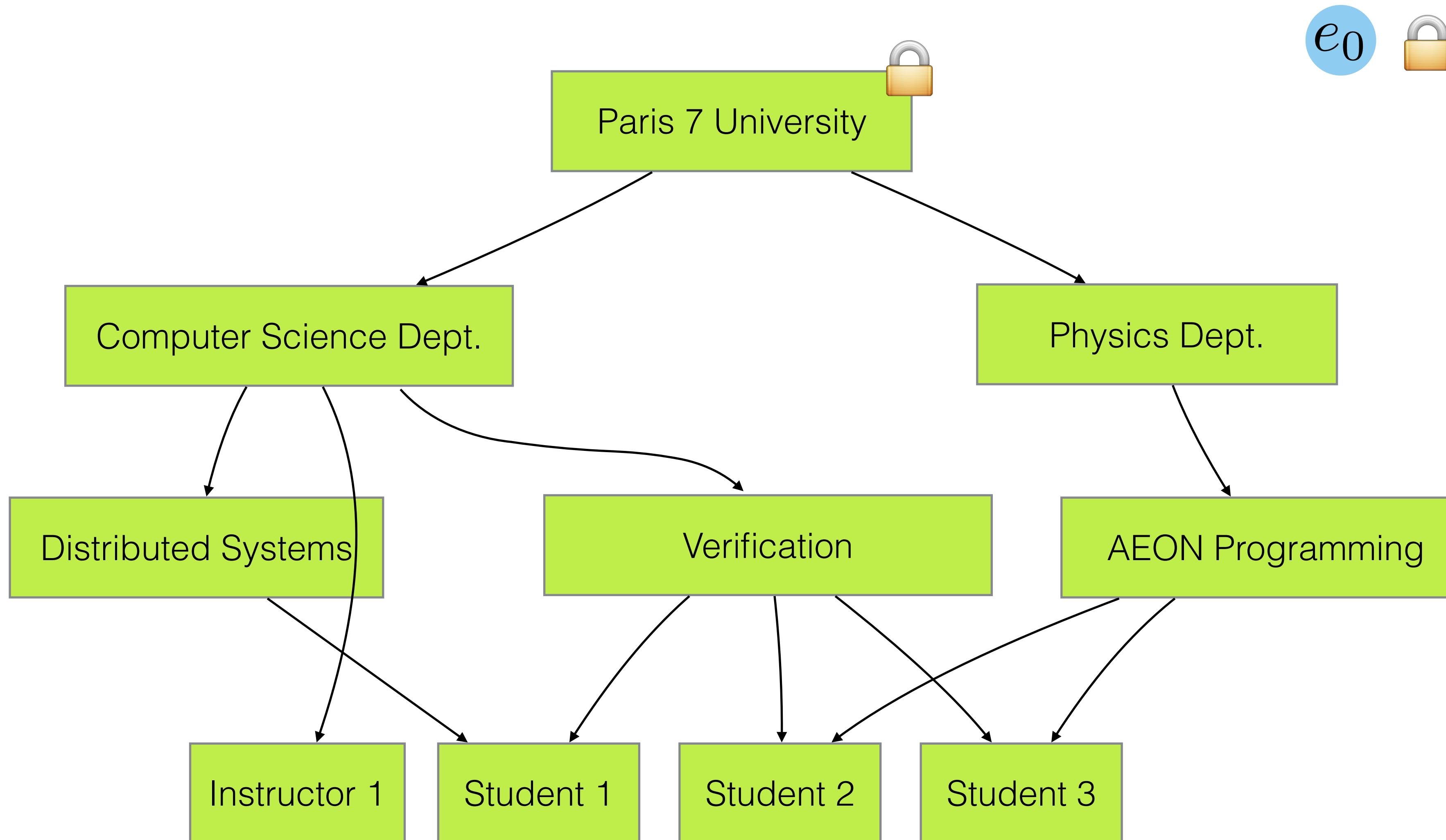
# CALLING MODES

- ▶ Events
    - ▶ Entry point for client requests
  - ▶ Synch Calls
    - ▶ Inter-context call
    - ▶ Calling context waits for result
  - ▶ Asynch
    - ▶ Inter-context call
    - ▶ Calling context does not wait for result (void)
- events <snow\_day, ...>
- instructors[i]->notify("Snow-Day");
- async students[i]->add\_tokens(1);

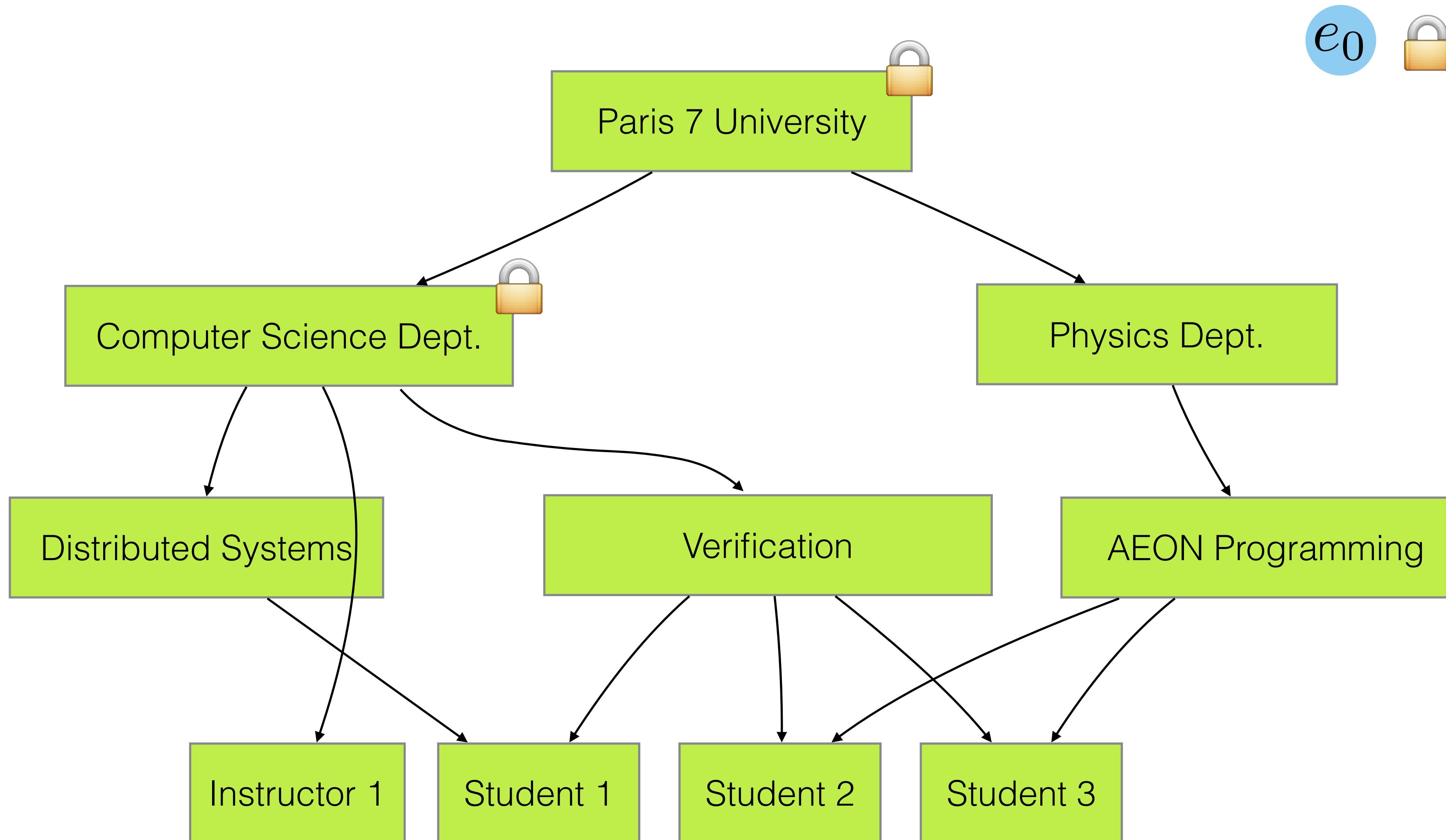
# OWNERSHIP NETWORKS



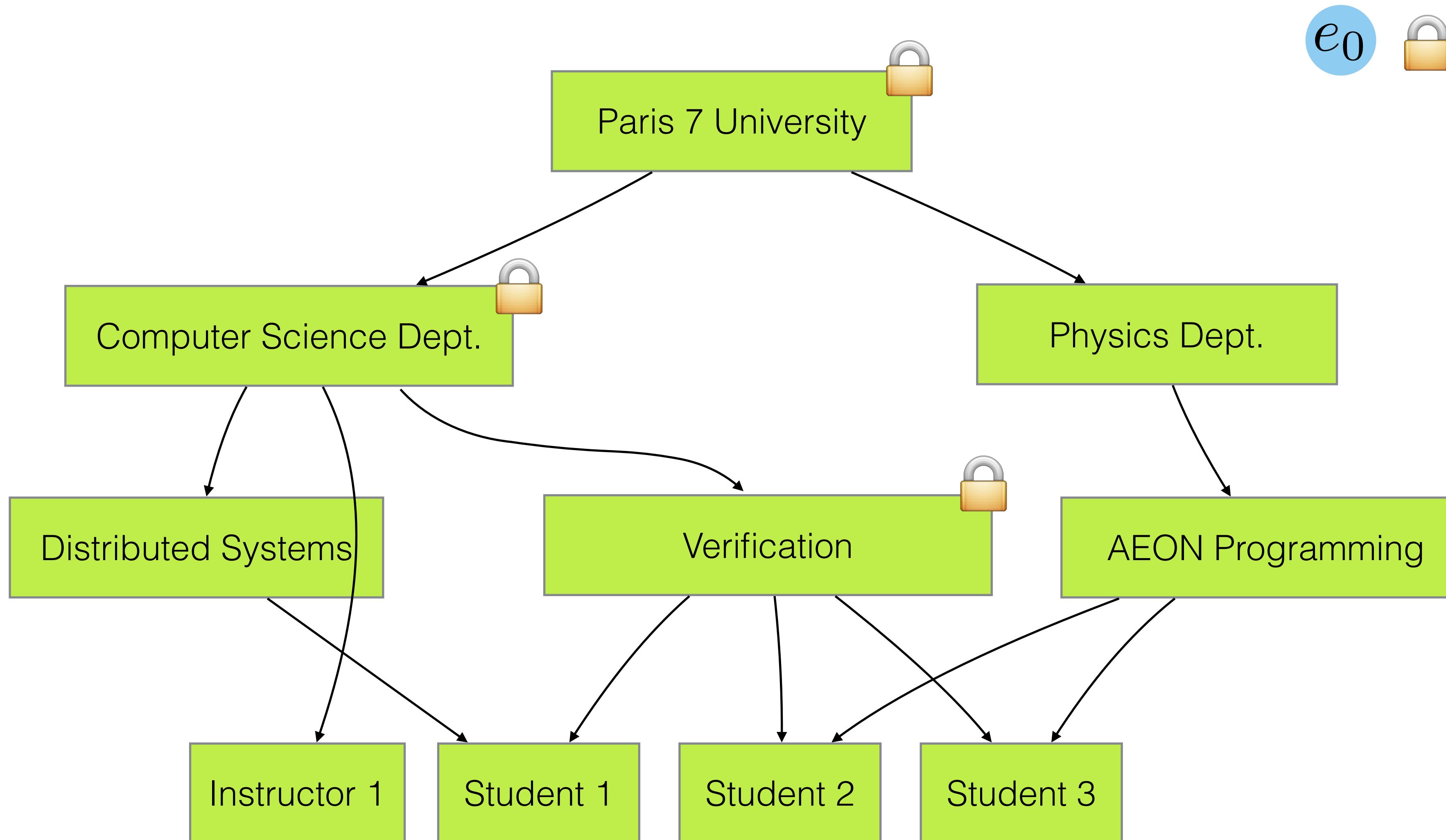
# OWNERSHIP NETWORKS



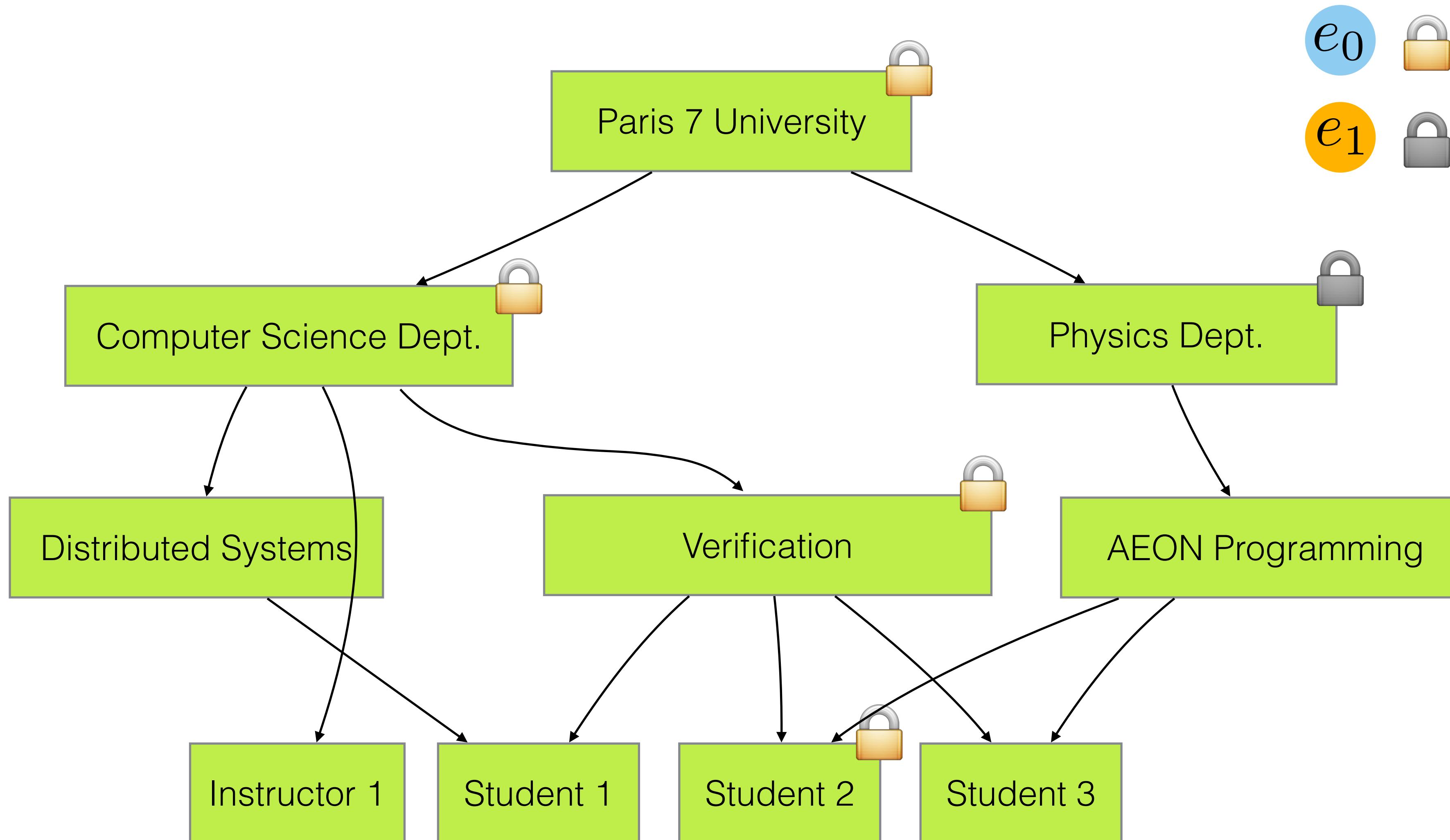
# OWNERSHIP NETWORKS



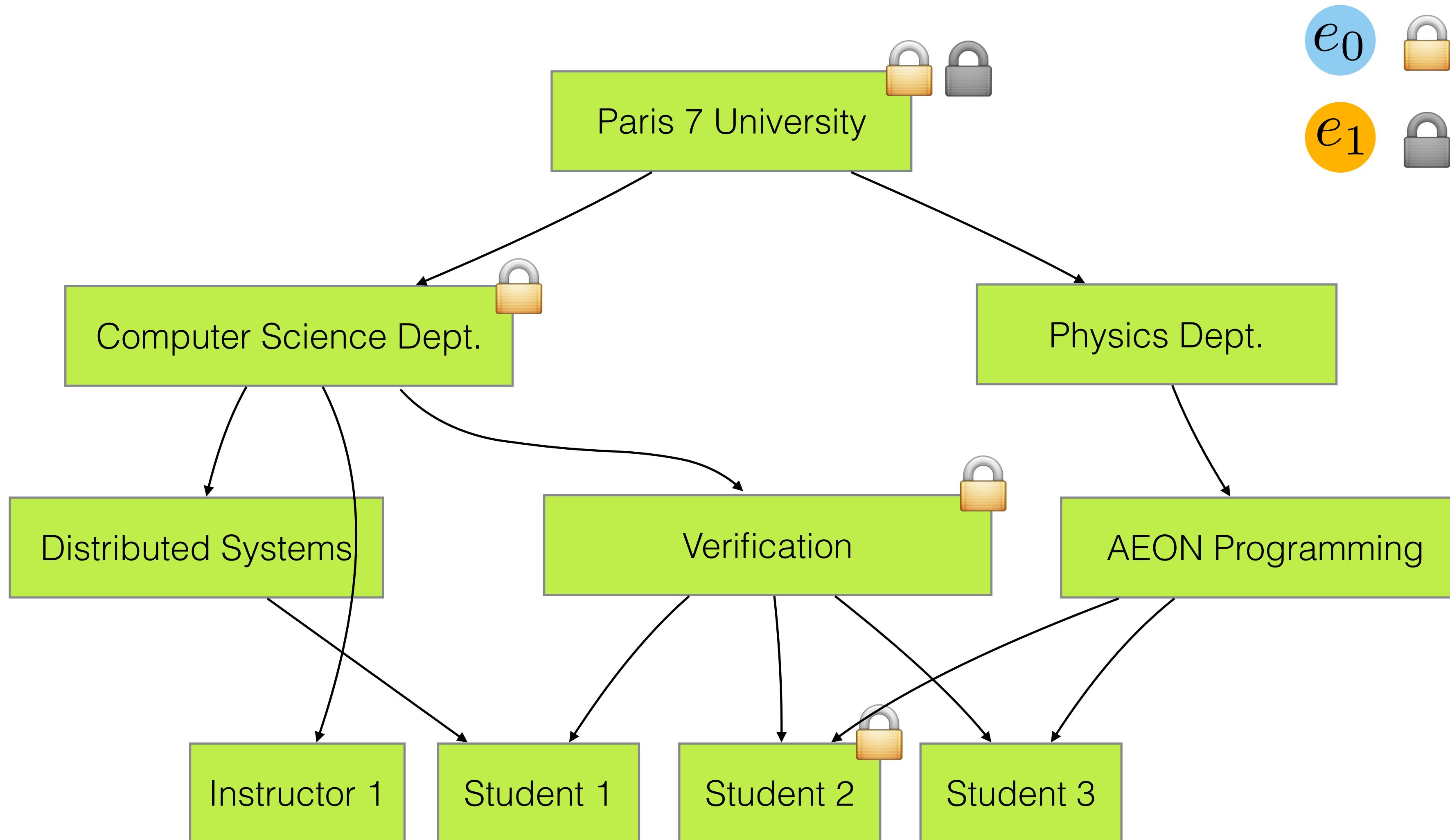
# OWNERSHIP NETWORKS



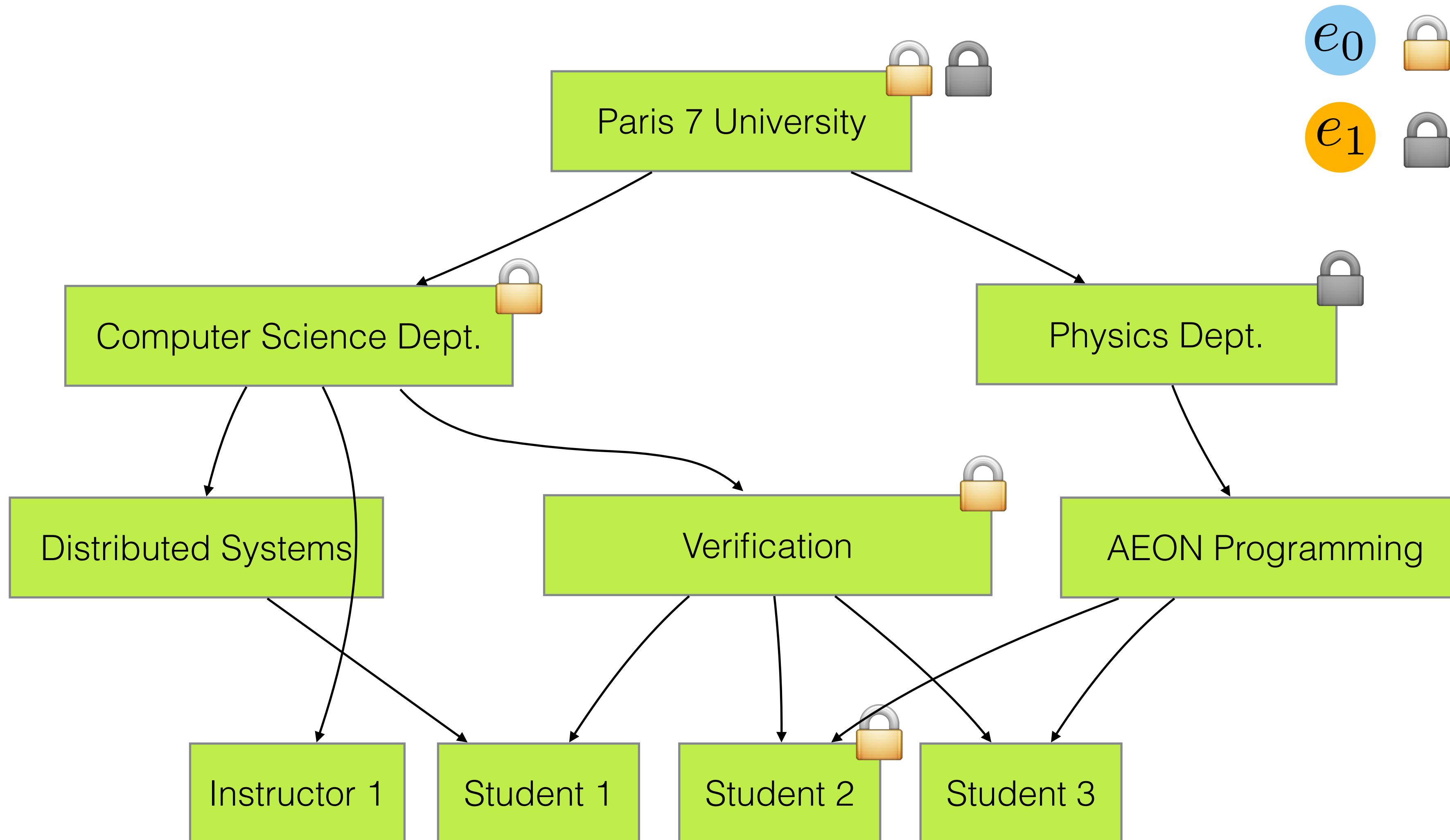
# OWNERSHIP NETWORKS



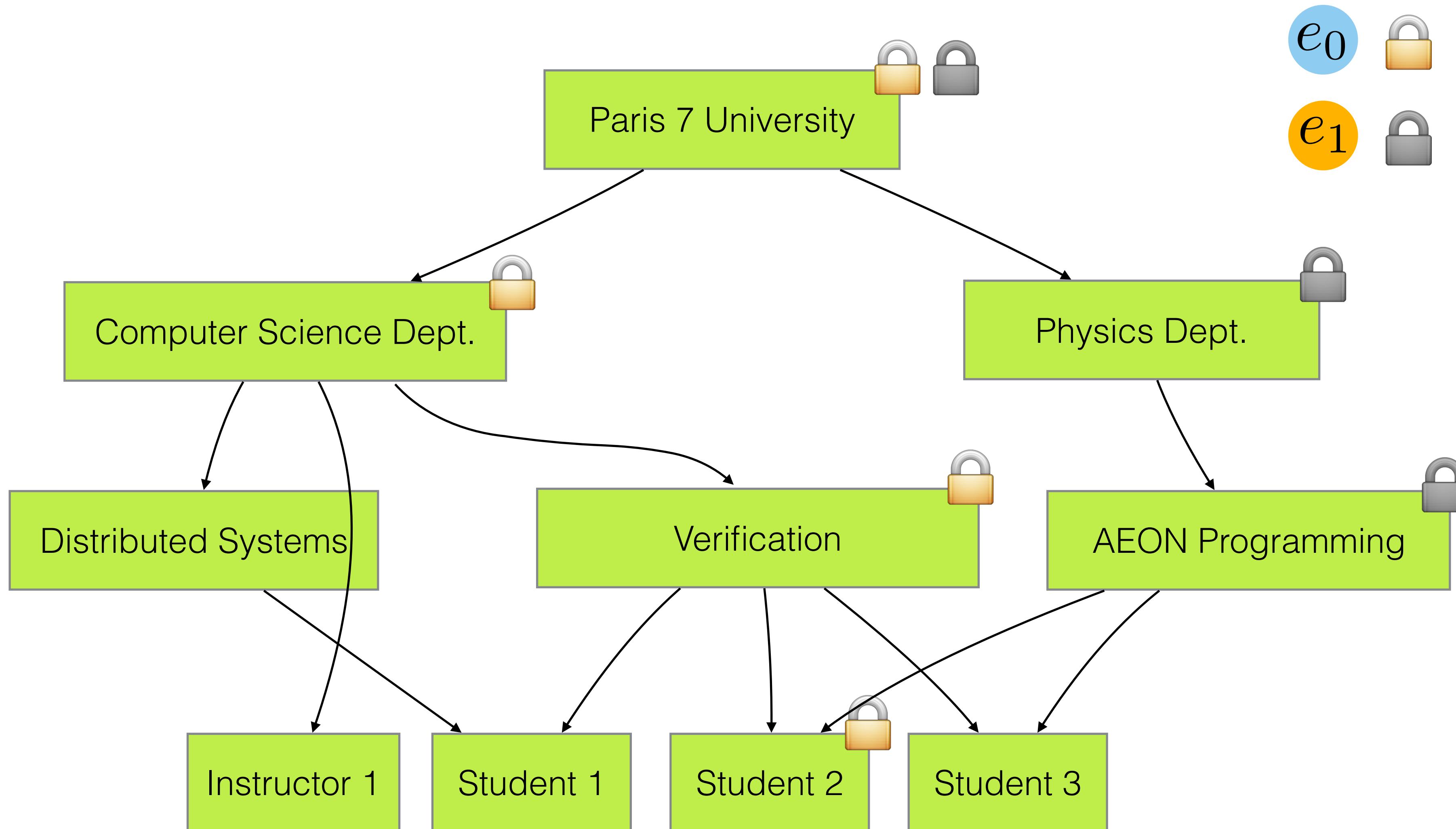
# OWNERSHIP NETWORKS



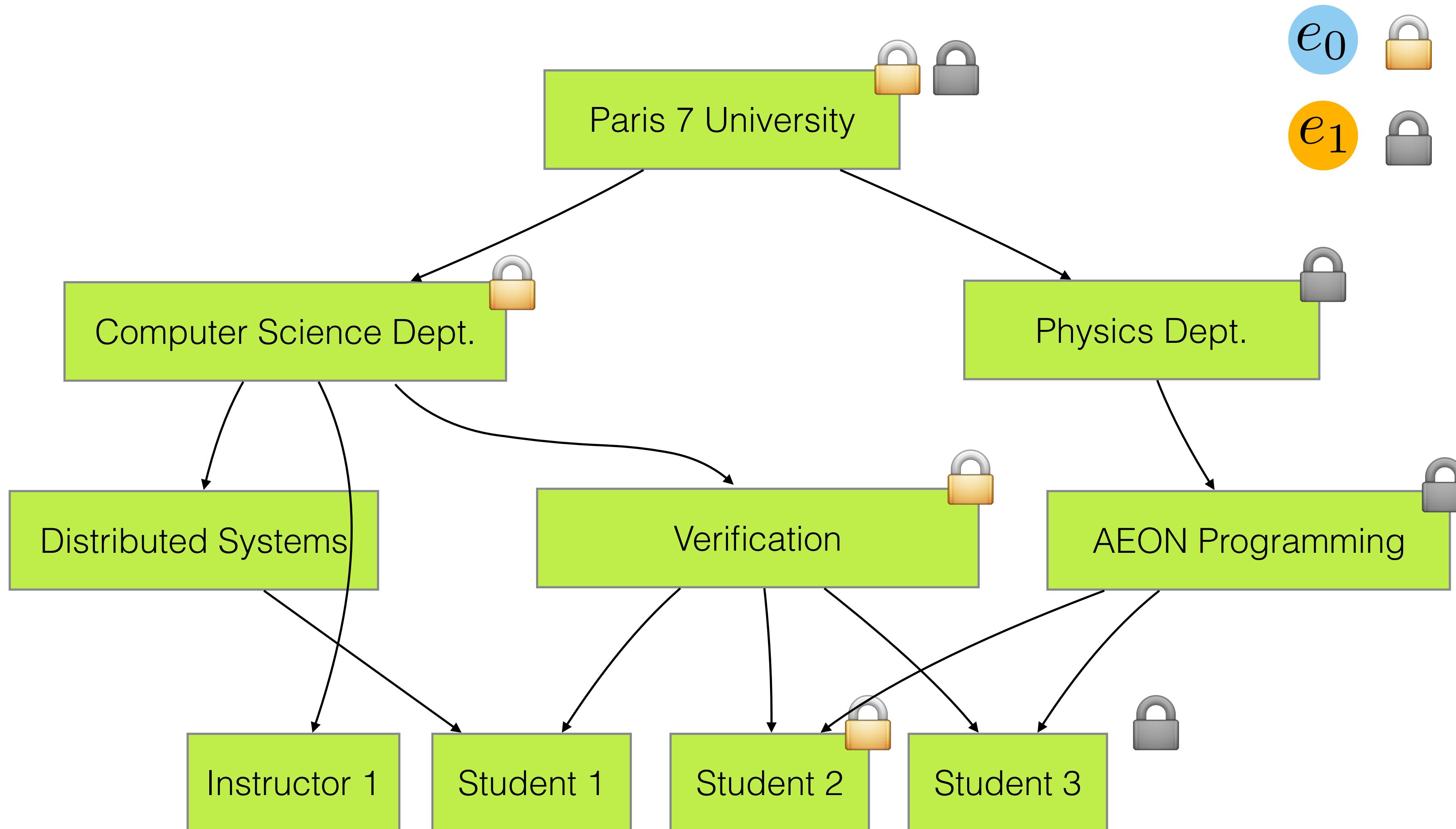
# OWNERSHIP NETWORKS



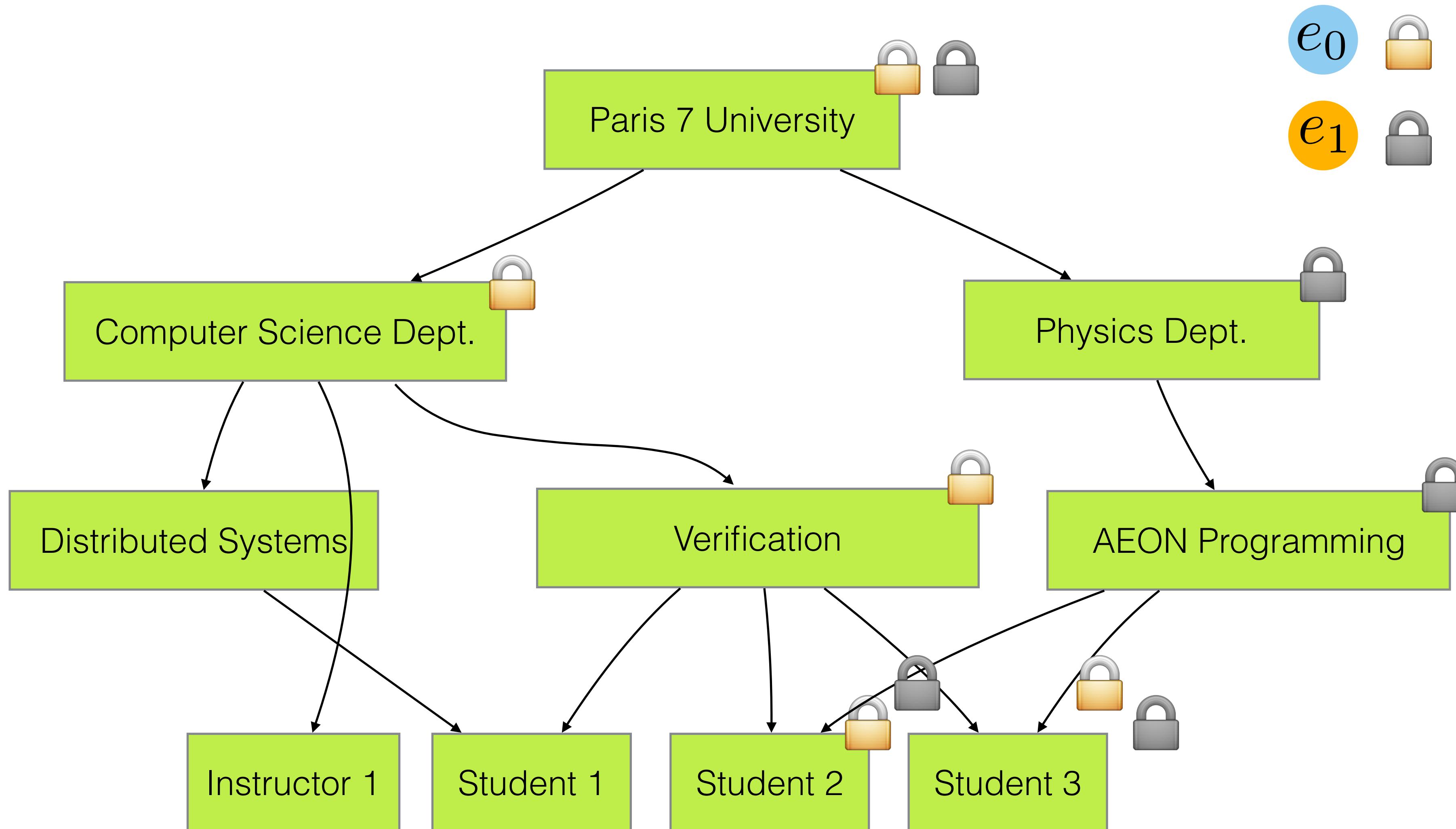
# OWNERSHIP NETWORKS



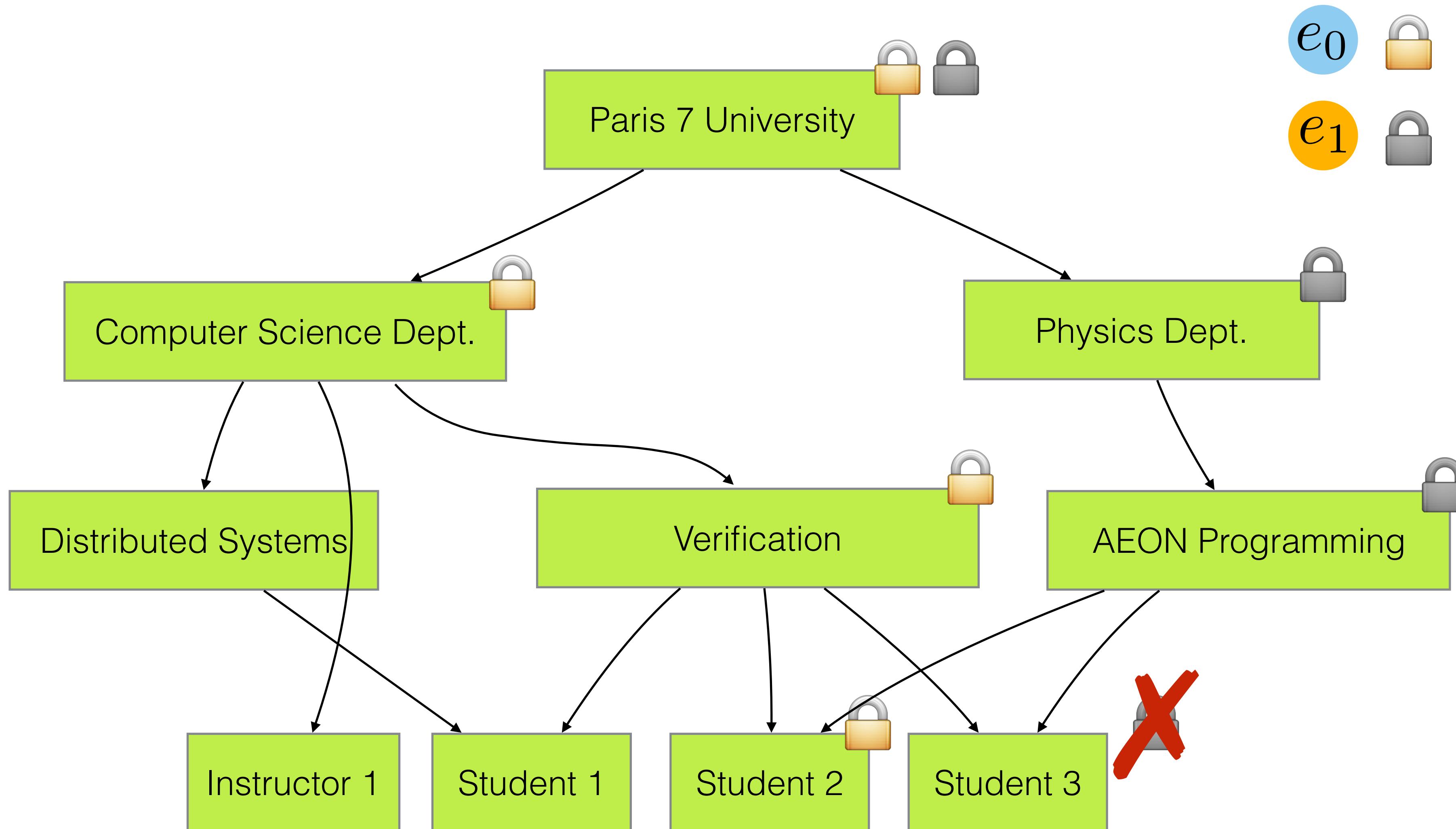
# OWNERSHIP NETWORKS



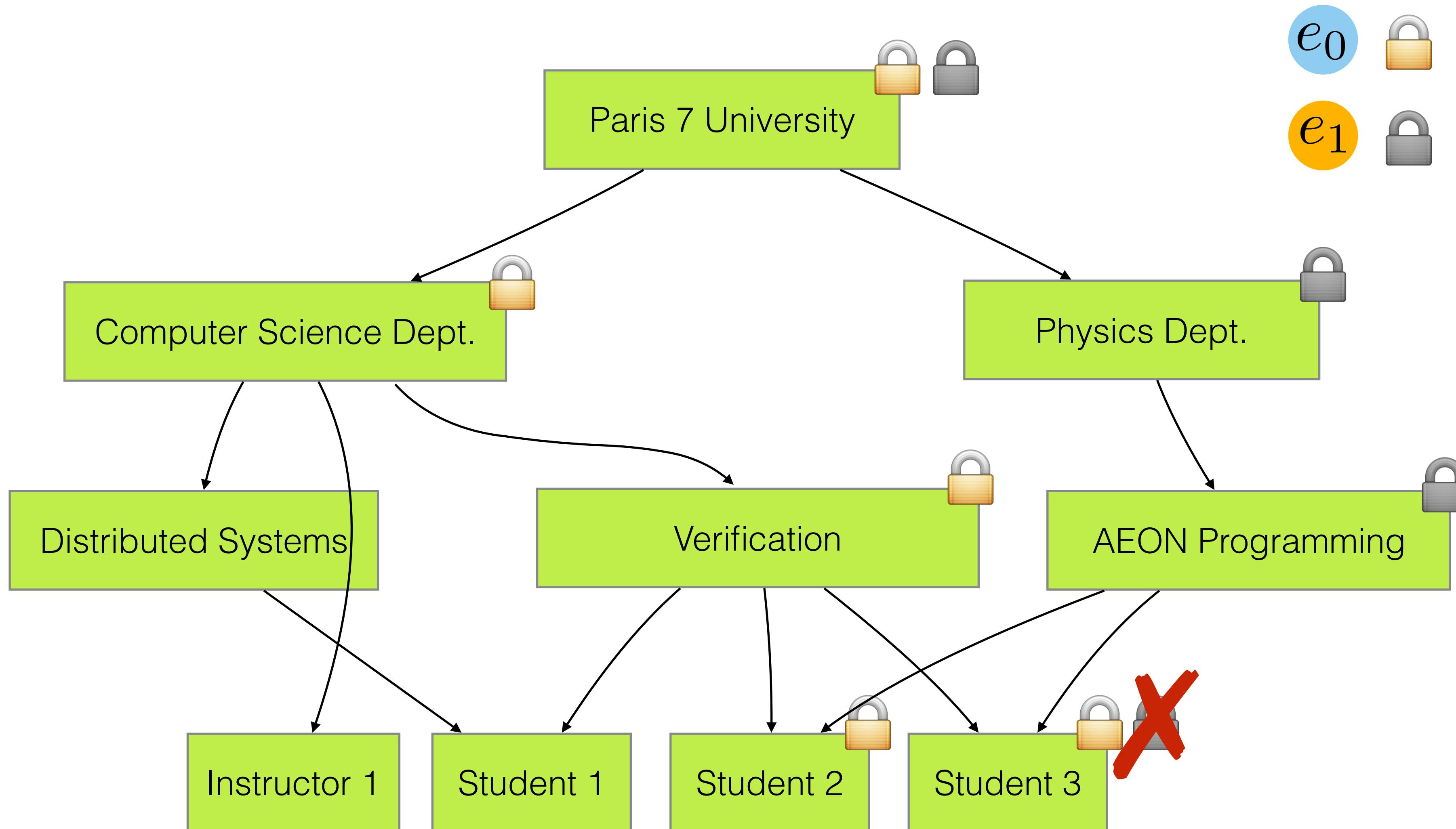
# OWNERSHIP NETWORKS



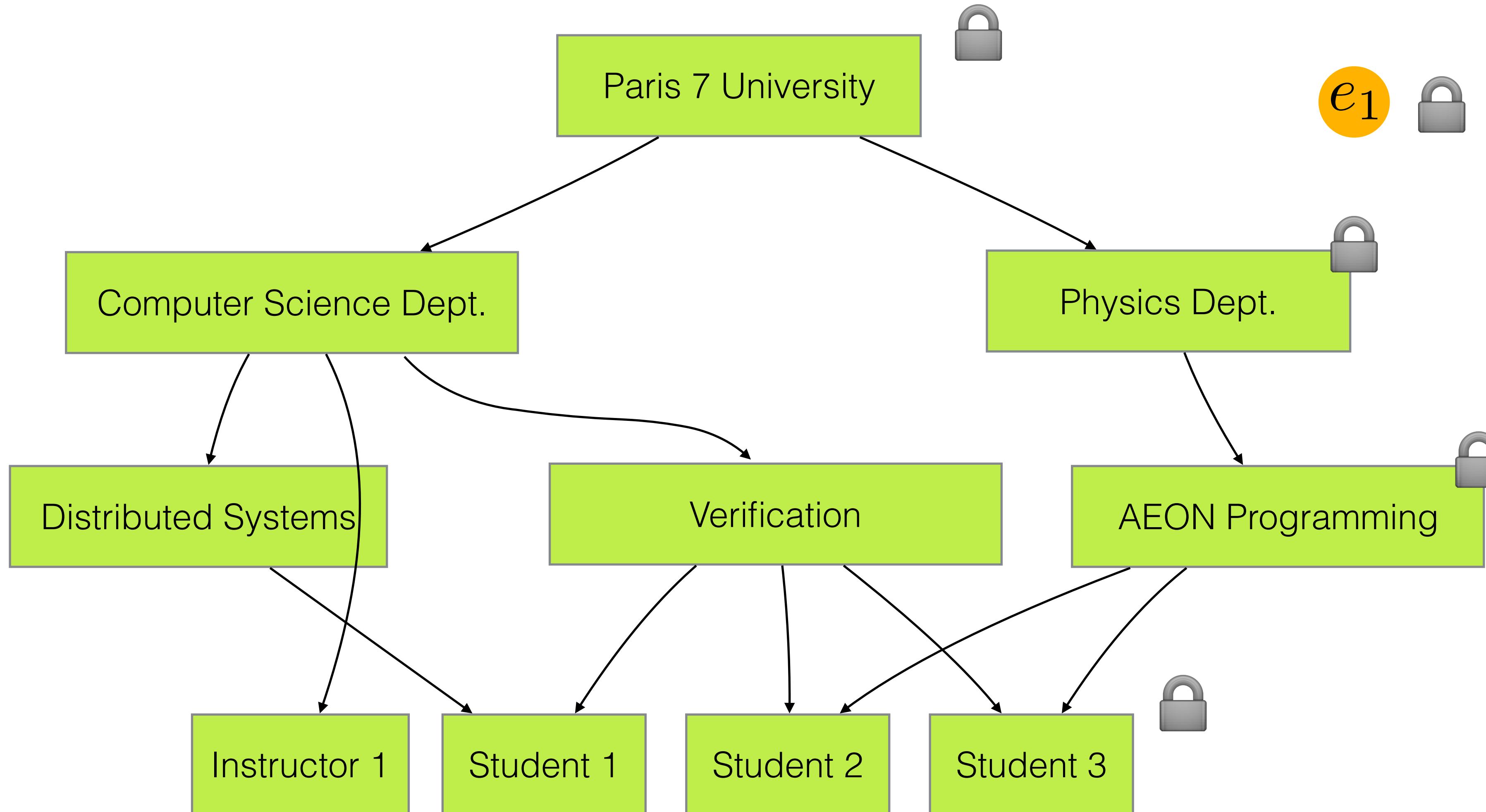
# OWNERSHIP NETWORKS



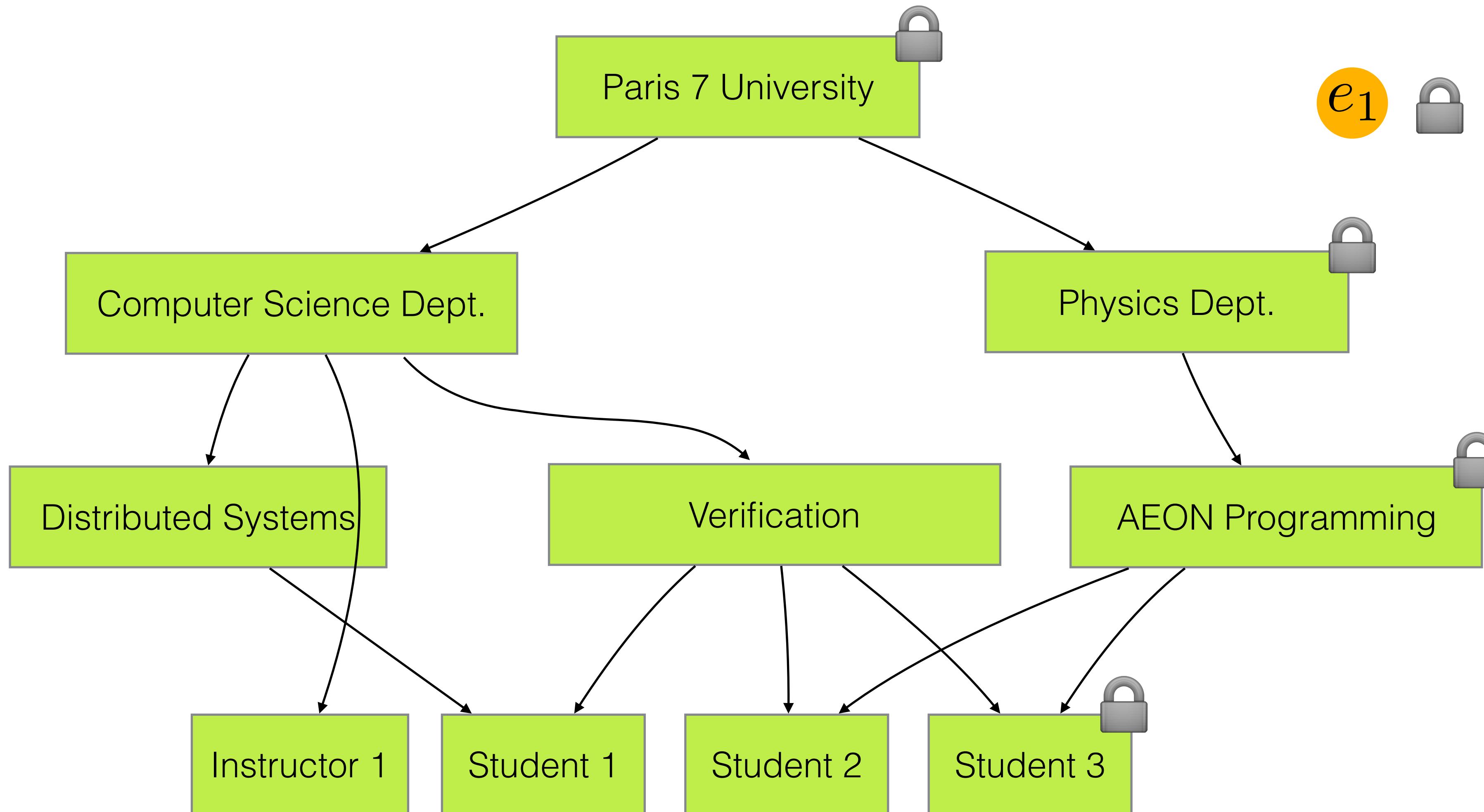
# OWNERSHIP NETWORKS



# OWNERSHIP NETWORKS



# OWNERSHIP NETWORKS



# GUARANTEES

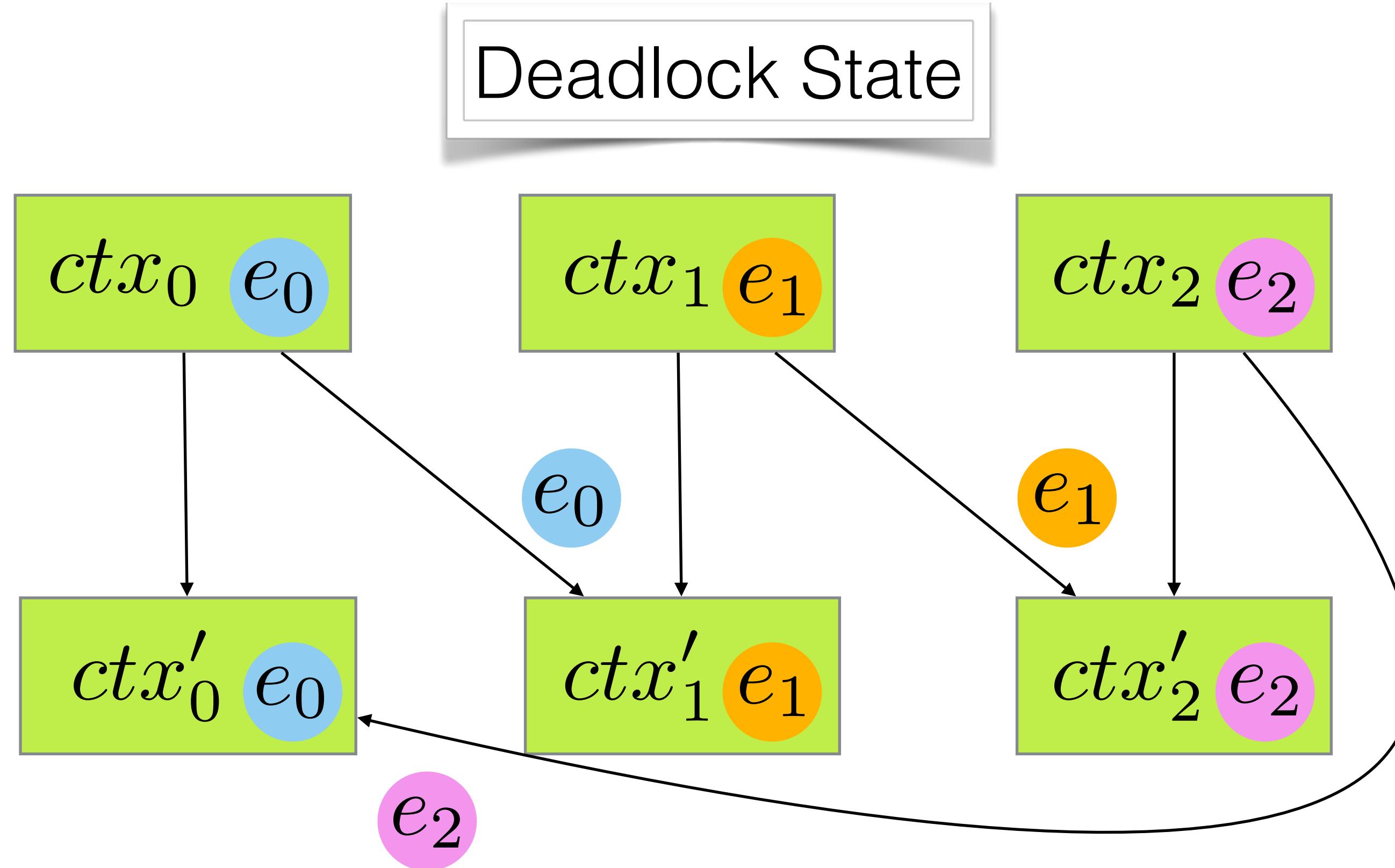
- ▶ Deadlock Freedom
- ▶ Transactional Events

Deadlock-Free Runtime

Transactional Execution of Events

Starvation-Free Runtime

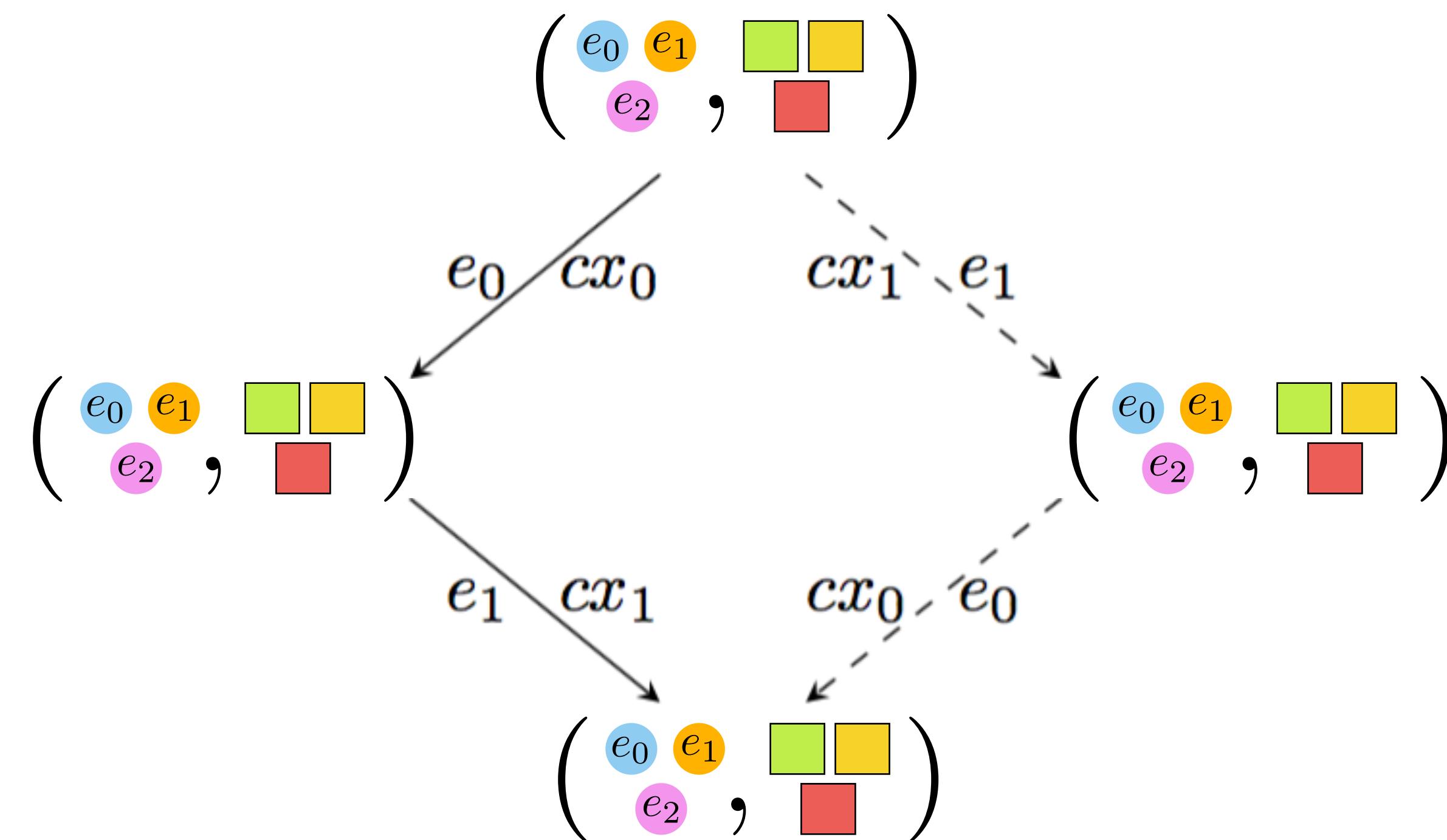
# DEADLOCK FREEDOM



**Theorem 0.1 (Deadlock-Freedom)** *For every execution of AEON  $(E, \Omega) \xrightarrow{*} (E', \Omega')$ , starting from a well-formed initial state  $(E, \Omega)$ , we have that  $(E', \Omega')$  is deadlock-free.*

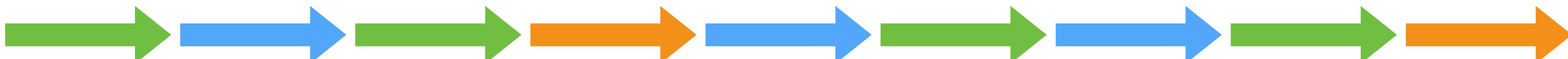
# TRANSACTIONAL EVENTS

Commutativity of independent events



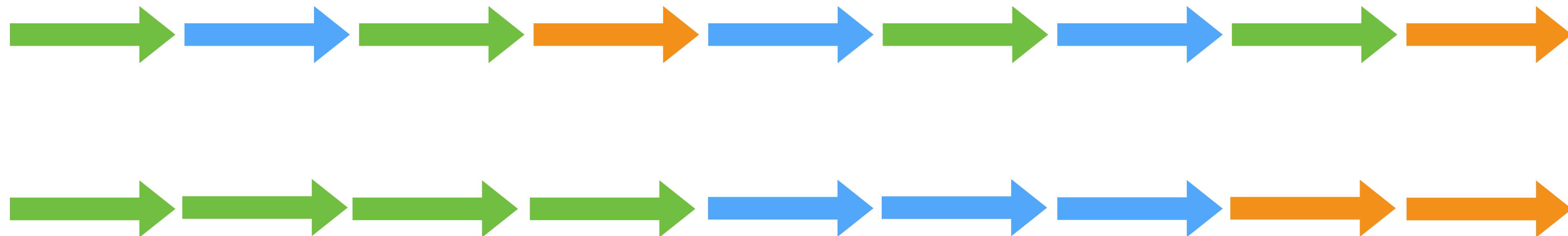
# TRANSACTIONAL EVENTS

- ▶ We build a semantics with no concurrence: **single-event**  
 $(\begin{smallmatrix} \text{green} & \text{yellow} \\ \text{red} & \end{smallmatrix}, e_0)$
- ▶ We show a *simulation* between our semantics and the single-event one
- ▶ Induction on commutativity and properties of locking



# TRANSACTIONAL EVENTS

- ▶ We build a semantics with no concurrence: **single-event**  
 $(\begin{smallmatrix} \text{green} & \text{yellow} \\ \text{red} & \end{smallmatrix}, e_0)$
- ▶ We show a *simulation* between our semantics and the single-event one
- ▶ Induction on commutativity and properties of locking



# STARVATION FREEDOM

FIFO Queues + Deadlock-Free

=>

Starvation Freedom

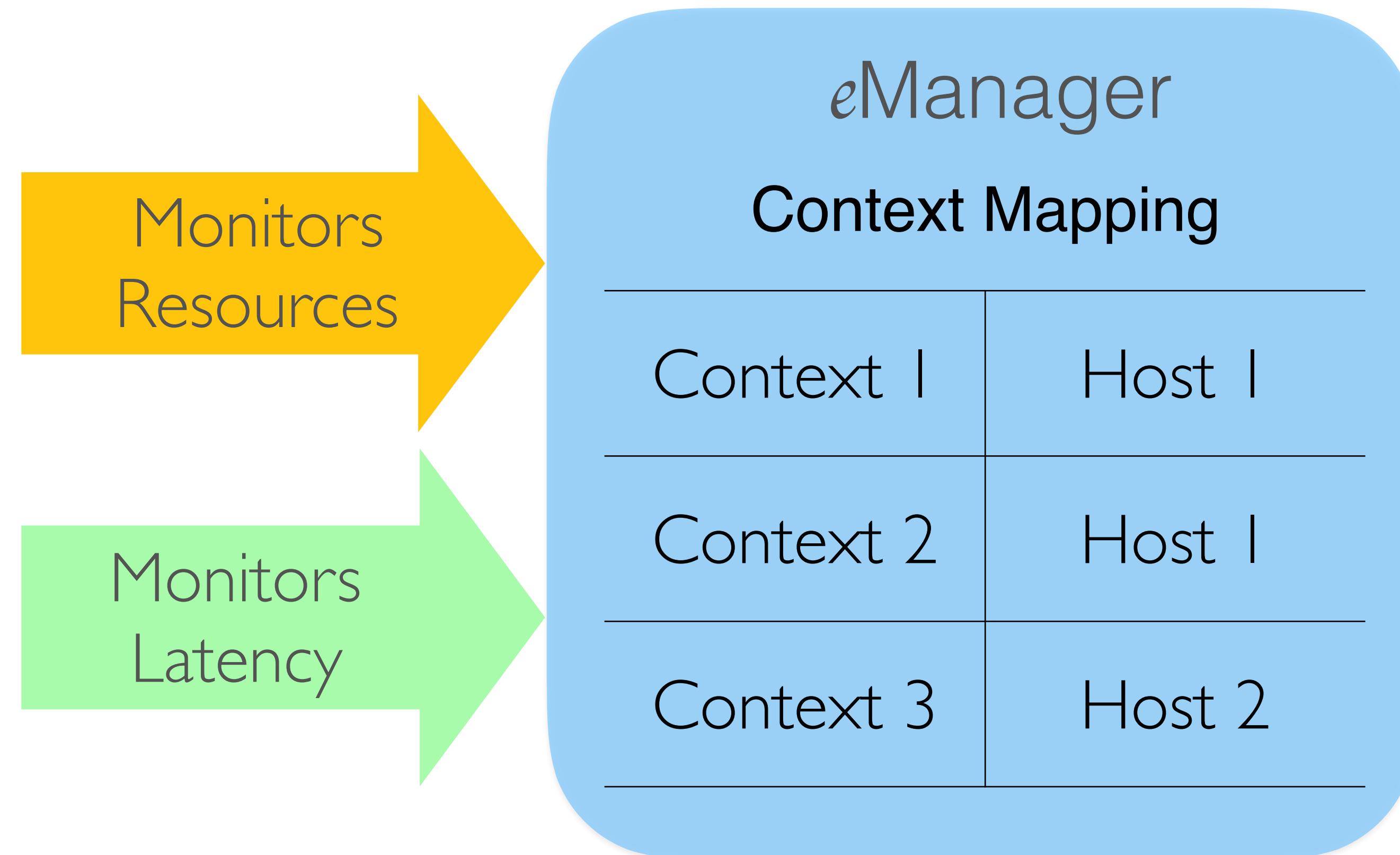
# ELASTICITY MANAGEMENT

eManager

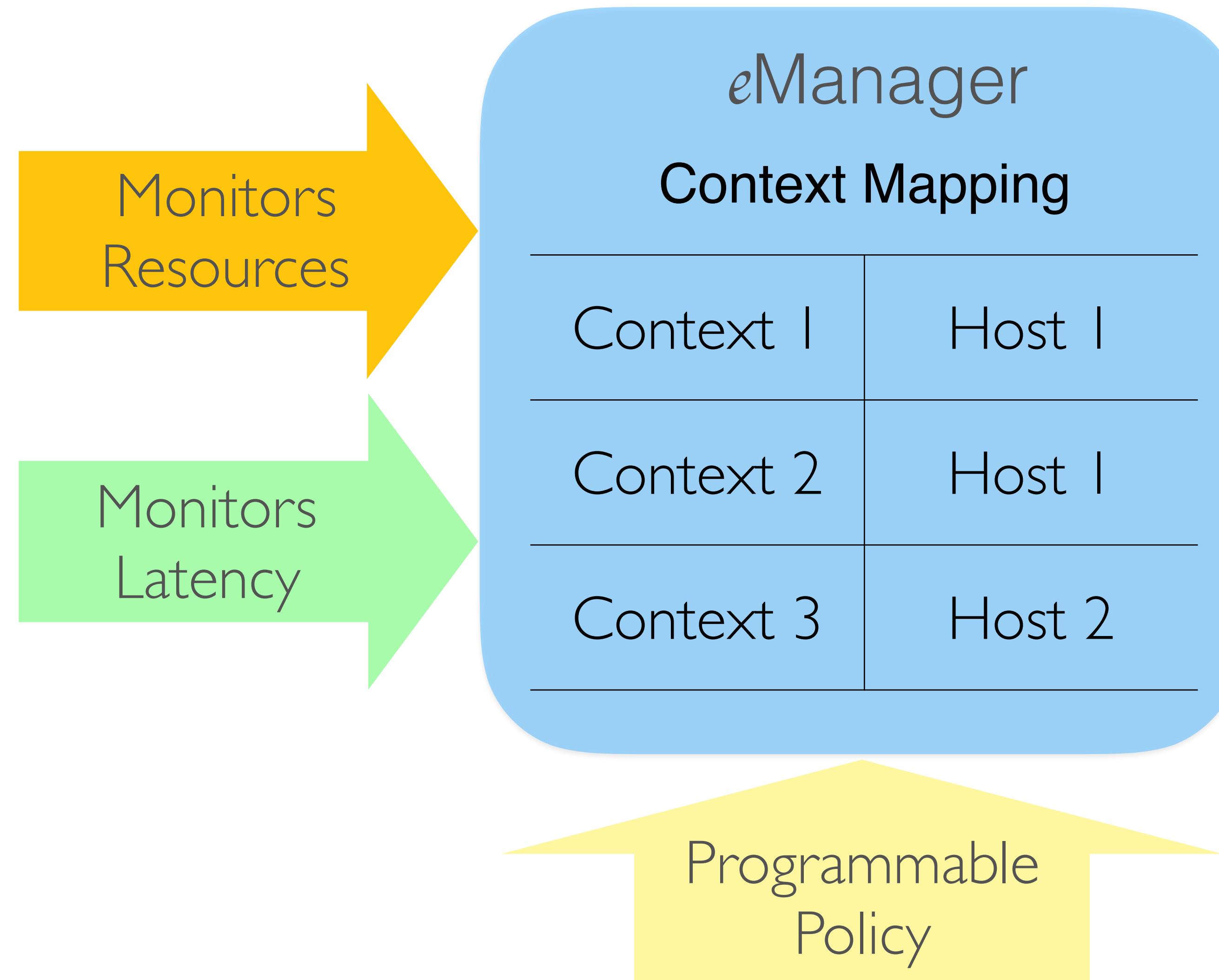
Context Mapping

Context 1	Host 1
Context 2	Host 1
Context 3	Host 2

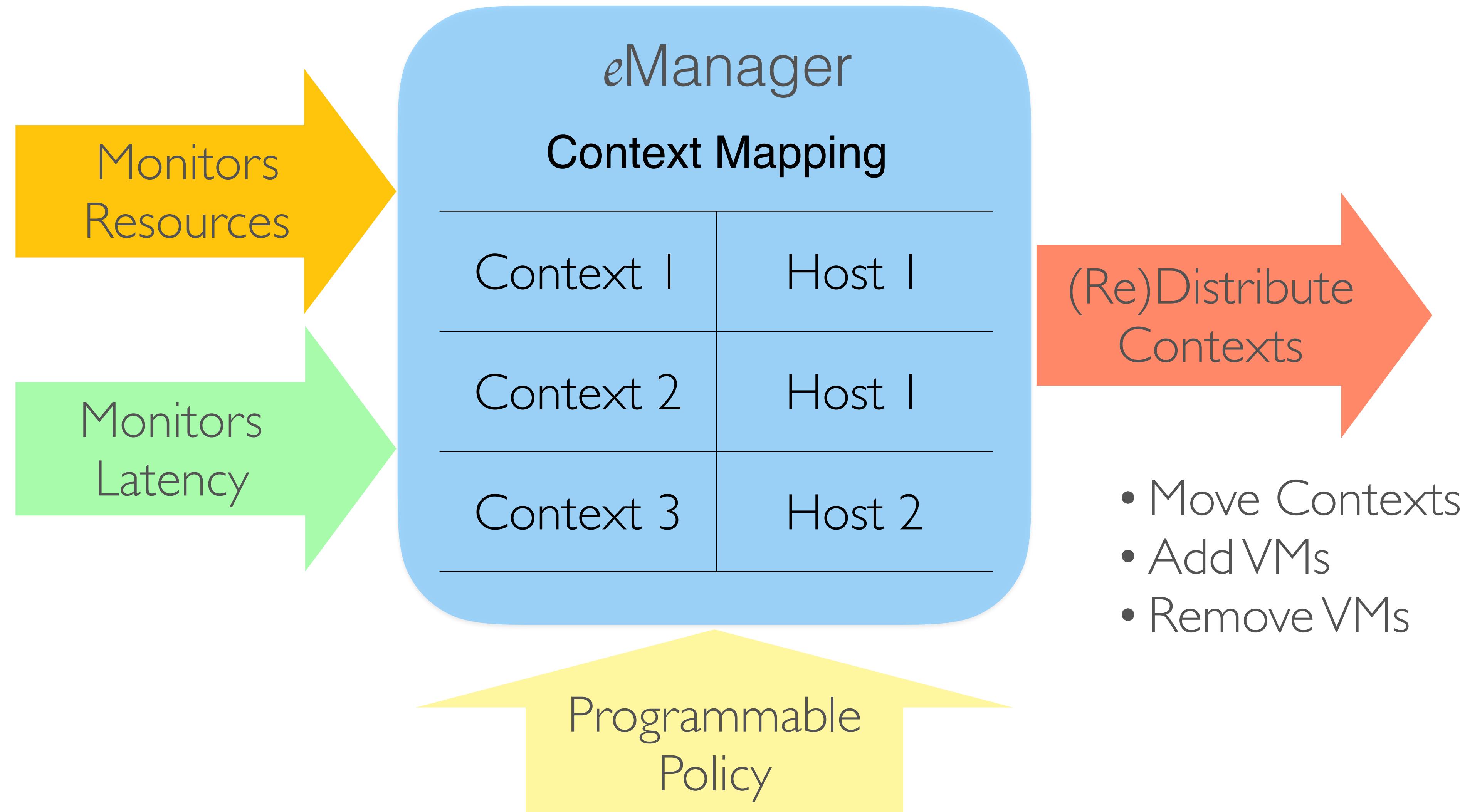
# ELASTICITY MANAGEMENT



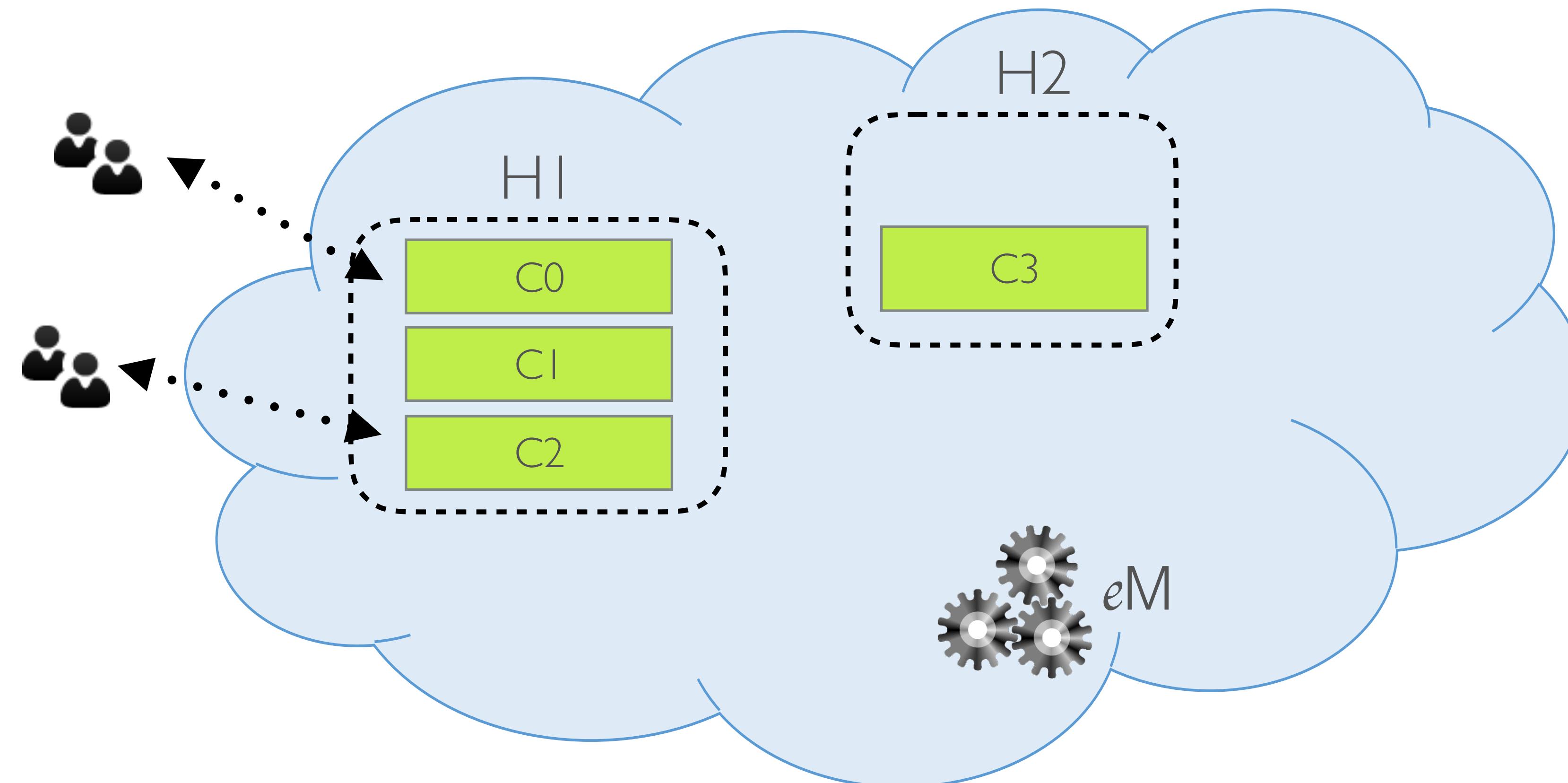
# ELASTICITY MANAGEMENT



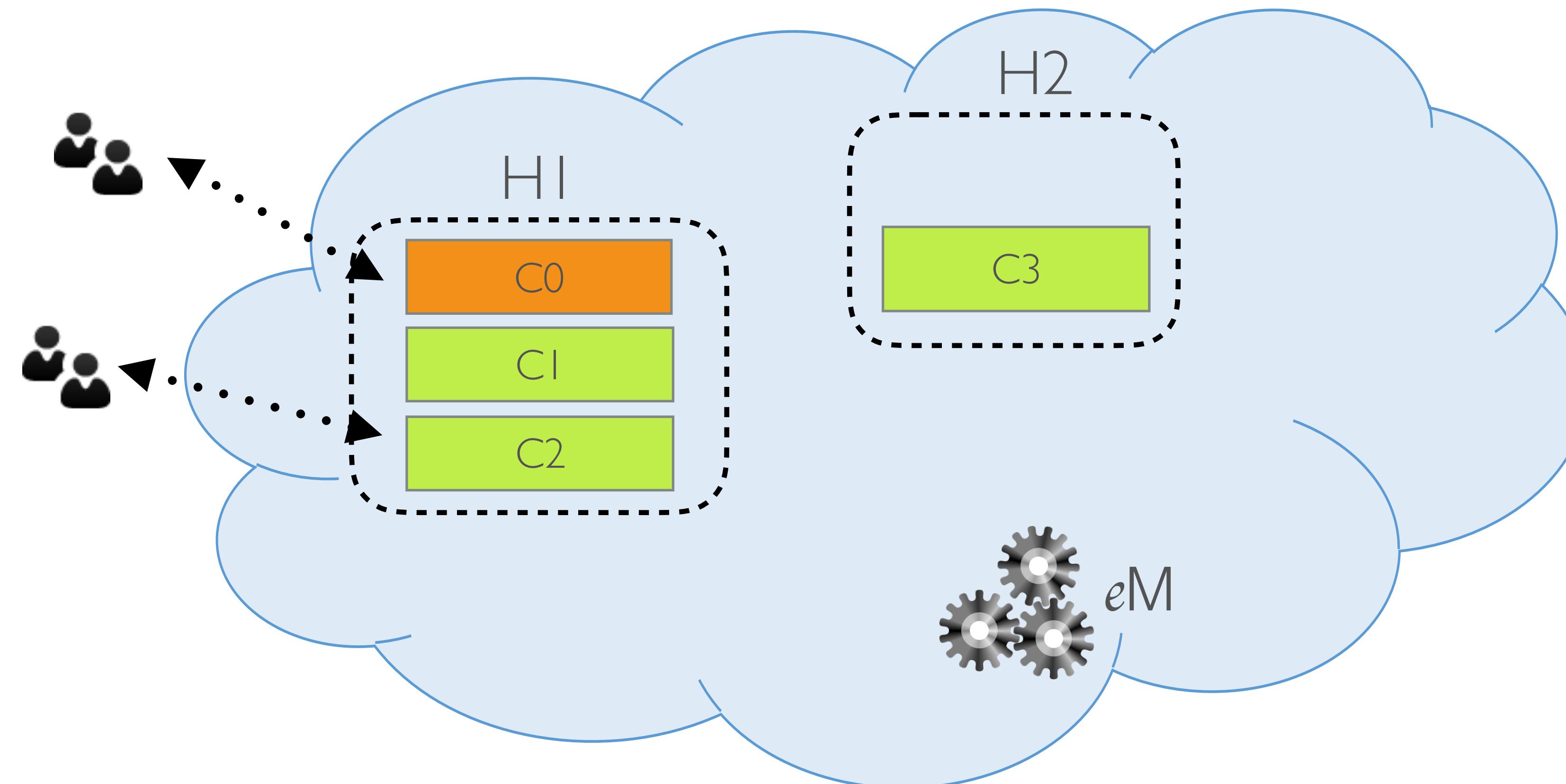
# ELASTICITY MANAGEMENT



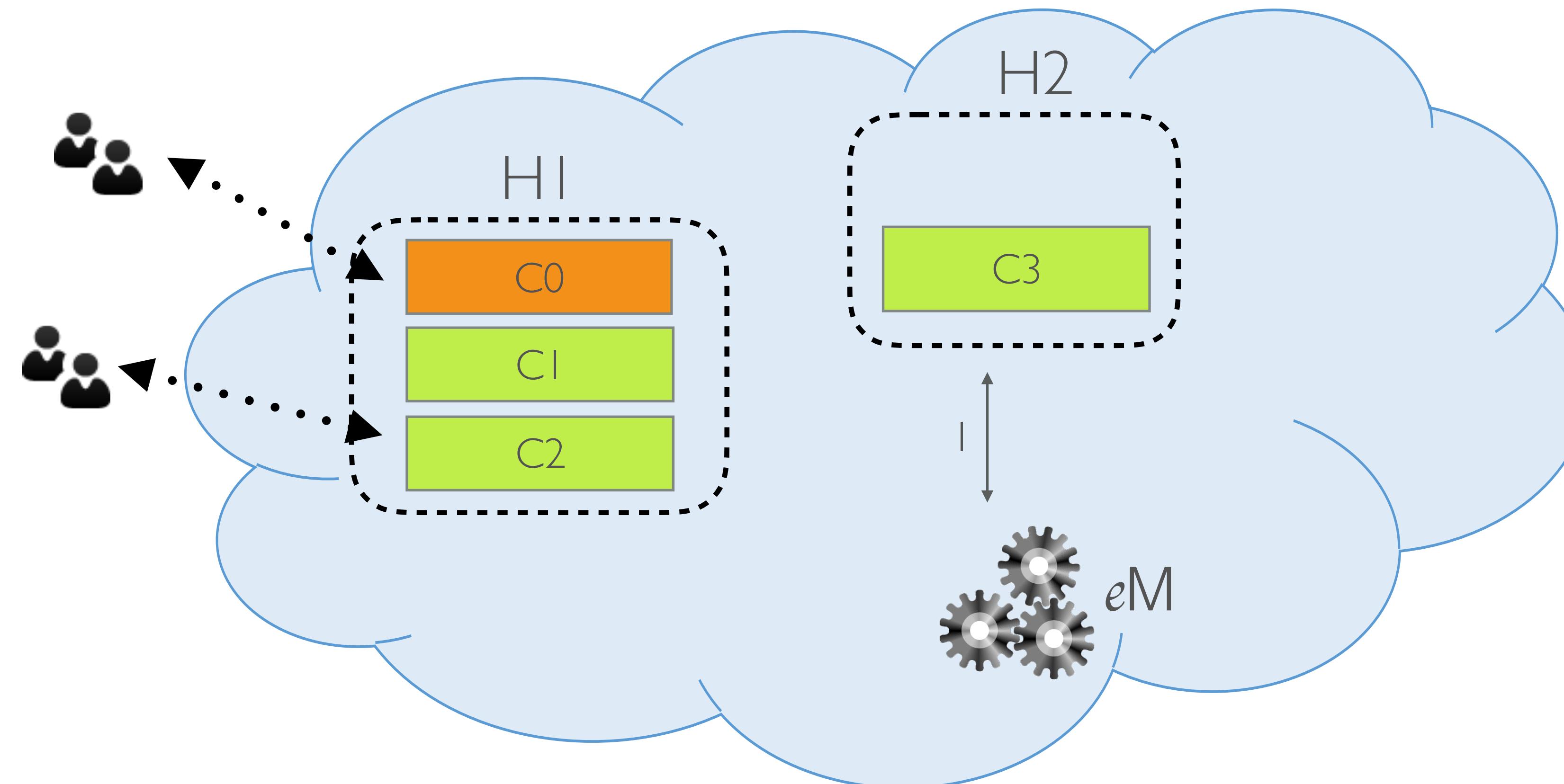
# ELASTICITY MANAGEMENT



# ELASTICITY MANAGEMENT

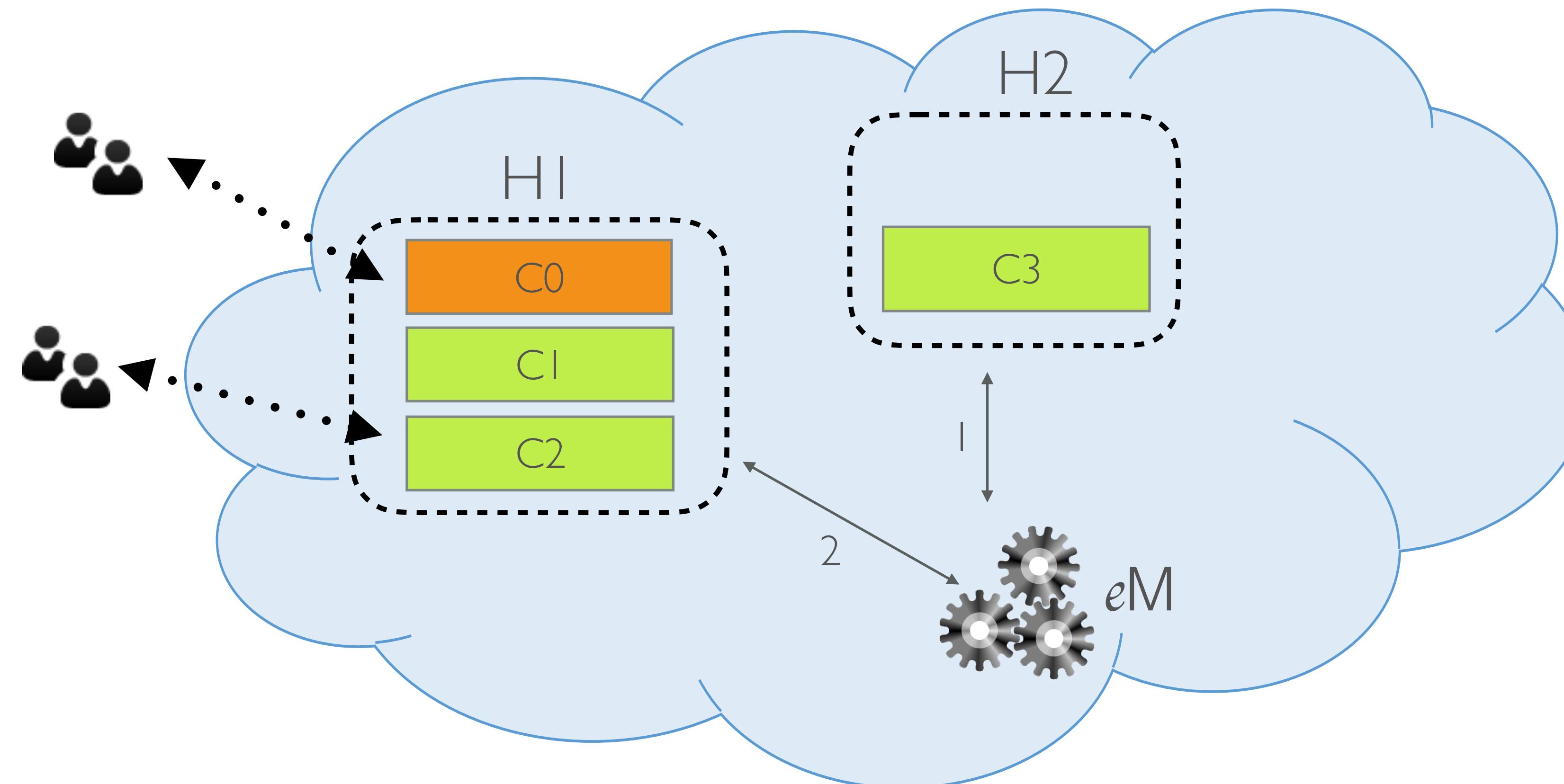


# ELASTICITY MANAGEMENT



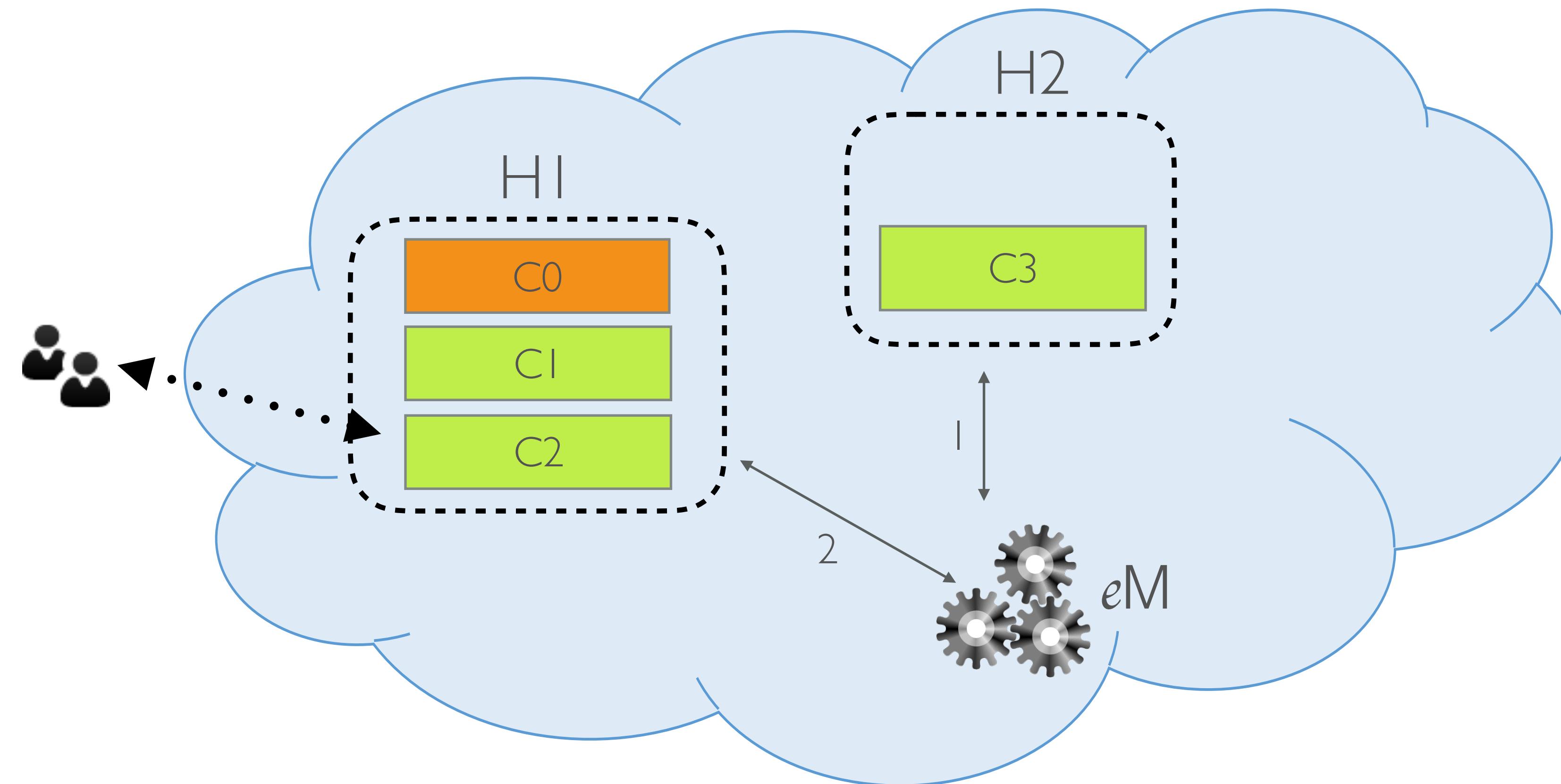
I.  $eM \rightarrow H2$ : Prepare to receive C0

# ELASTICITY MANAGEMENT



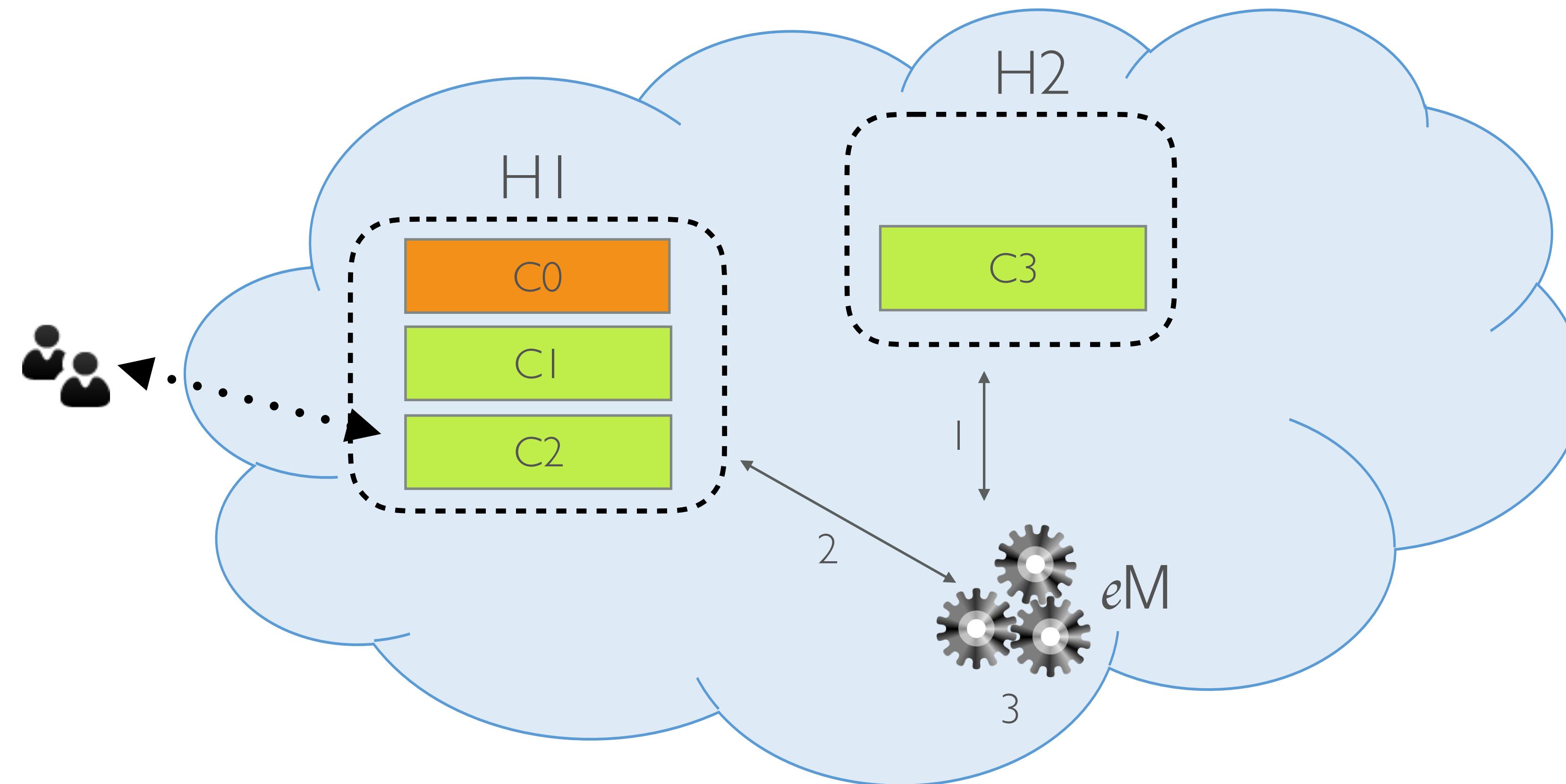
1.  $eM \rightarrow H_2$ : Prepare to receive  $C_0$
2.  $eM \rightarrow H_1$ : Stop receiving events for  $C_0$

# ELASTICITY MANAGEMENT



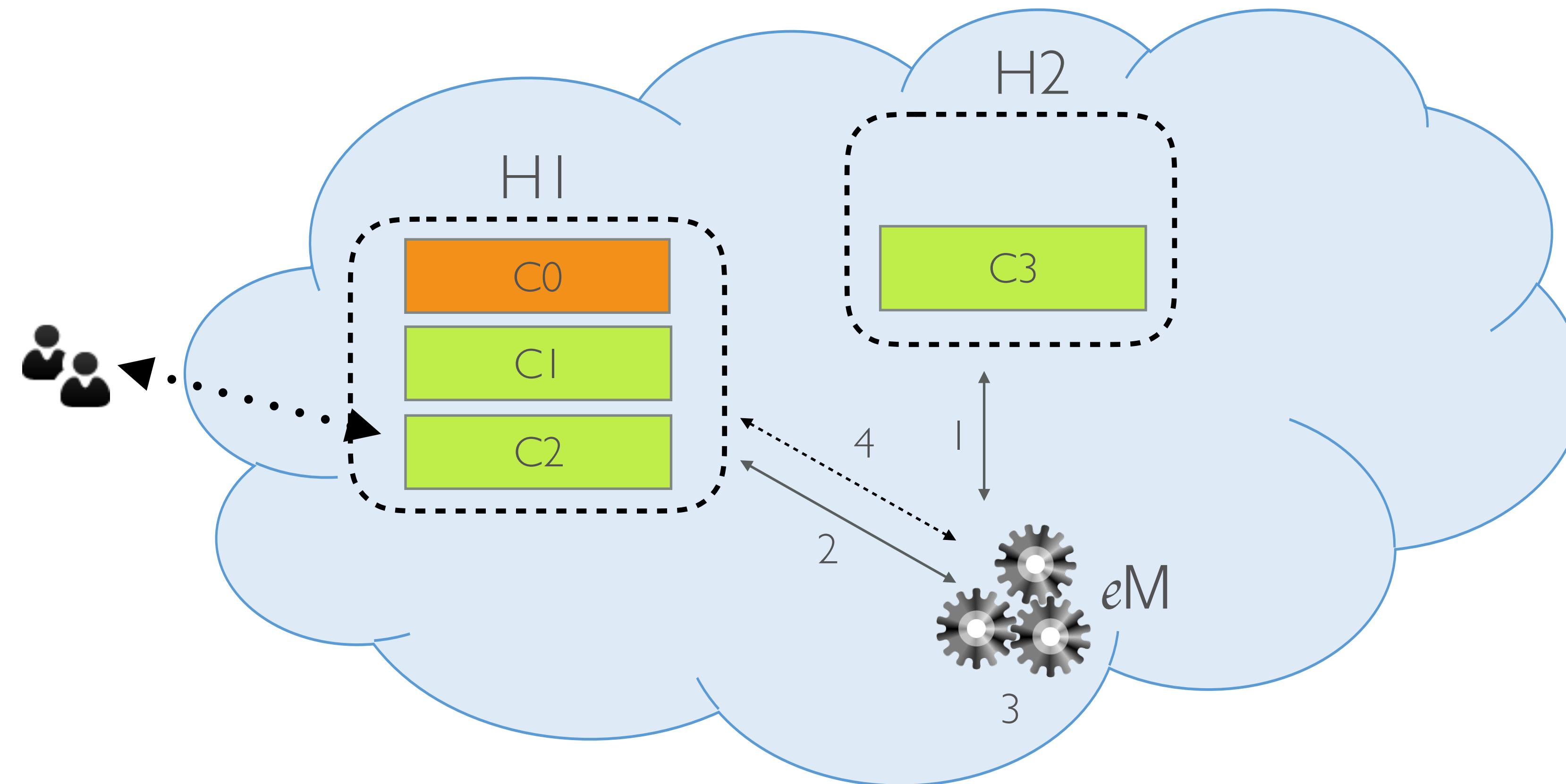
1.  $eM \rightarrow H2$ : Prepare to receive C0
2.  $eM \rightarrow H1$ : Stop receiving events for C0

# ELASTICITY MANAGEMENT



1.  $eM \rightarrow H2$ : Prepare to receive C0
2.  $eM \rightarrow H1$ : Stop receiving events for C0
3.  $eM$ : Assign C0 to H2

# ELASTICITY MANAGEMENT



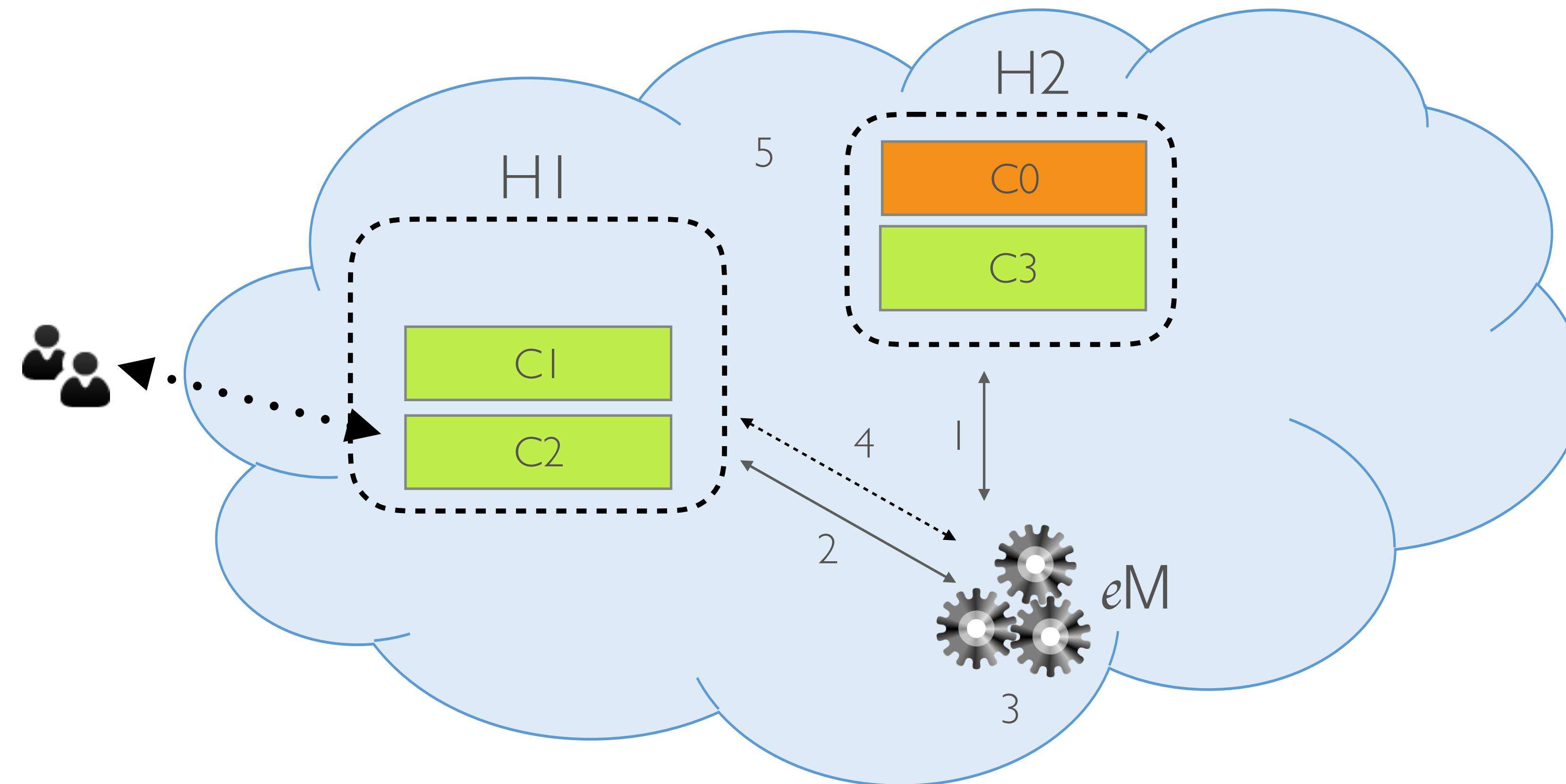
1.  $eM \rightarrow H2$ : Prepare to receive C0

2.  $eM \rightarrow H1$ : Stop receiving events for C0

3.  $eM$ : Assign C0 to H2

4.  $eM \rightarrow H1$ : Migrate C0 to H2

# ELASTICITY MANAGEMENT



1.  $eM \rightarrow H2$ : Prepare to receive C0

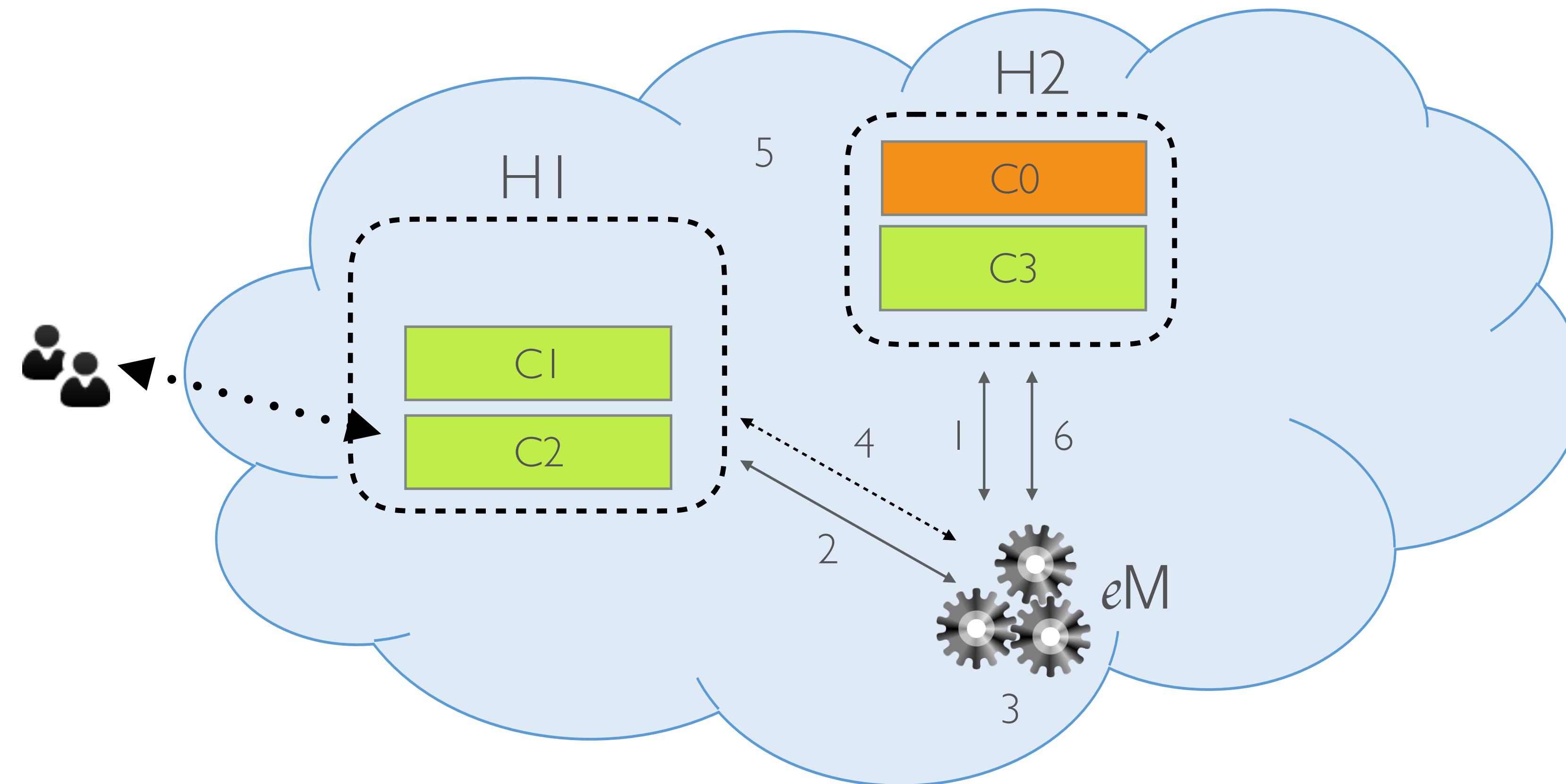
2.  $eM \rightarrow H1$ : Stop receiving events for C0

3.  $eM$ : Assign C0 to H2

4.  $eM \rightarrow H1$ : Migrate C0 to H2

5.  $H1 \rightarrow H2$ : Migrate C0

# ELASTICITY MANAGEMENT

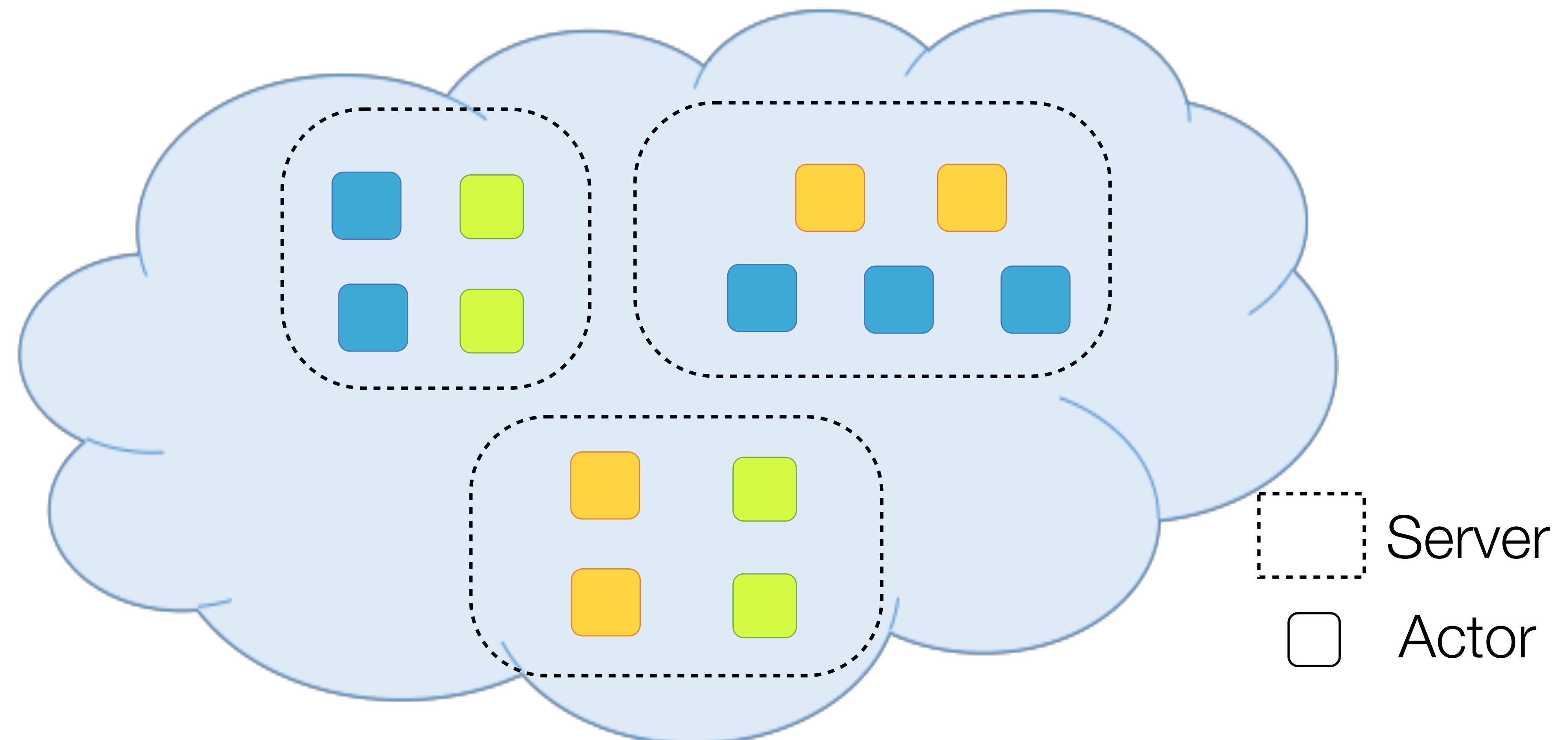


1.  $eM \rightarrow H2$ : Prepare to receive C0
2.  $eM \rightarrow H1$ : Stop receiving events for C0
3.  $eM$ : Assign C0 to H2

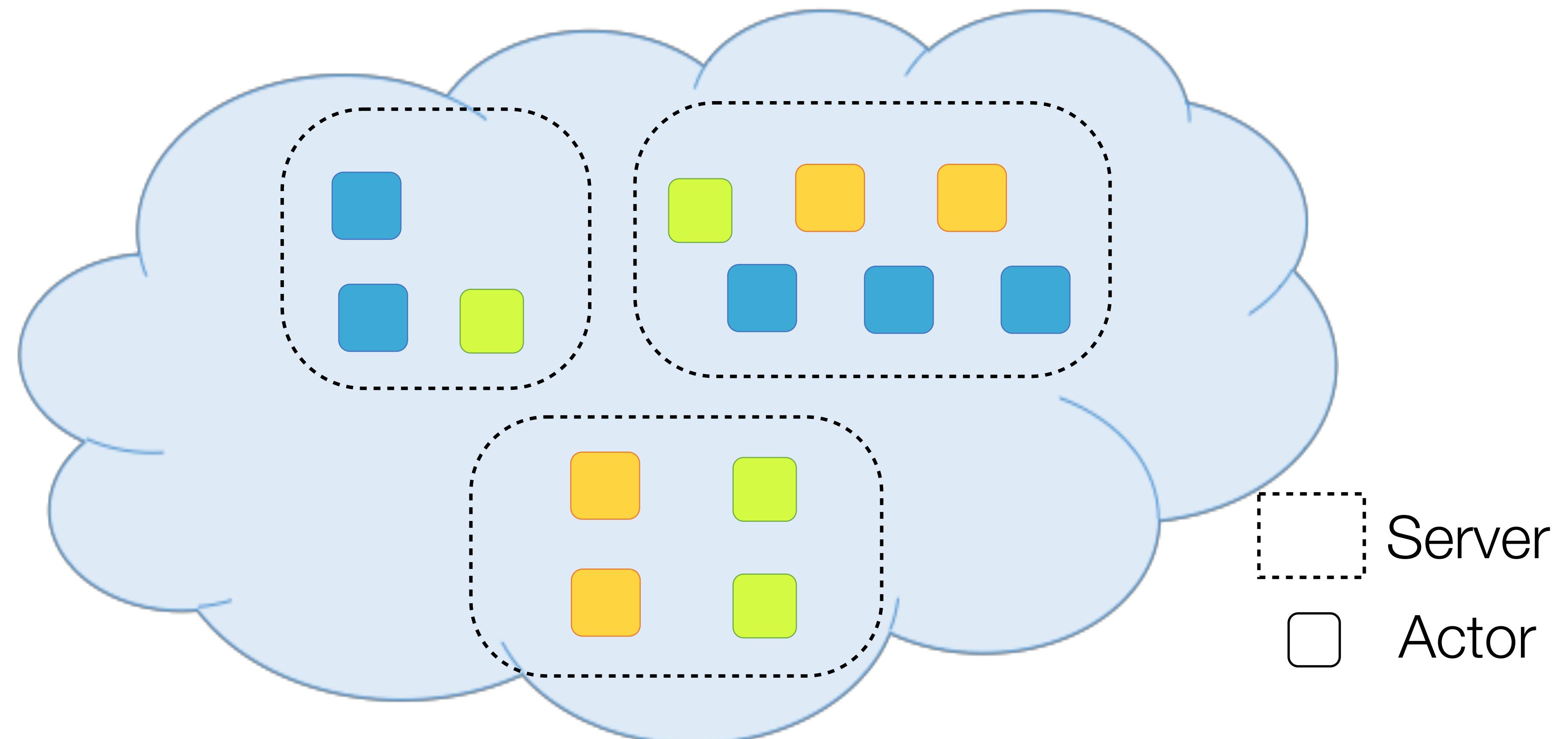
4.  $eM \rightarrow H1$ : Migrate C0 to H2
5.  $H1 \rightarrow H2$ : Migrate C0
6.  $H2 \rightarrow eM$ : Migration is complete

# AEON: PROGRAMMABLE ELASTICITY

# ACTOR-BASED ELASTICITY

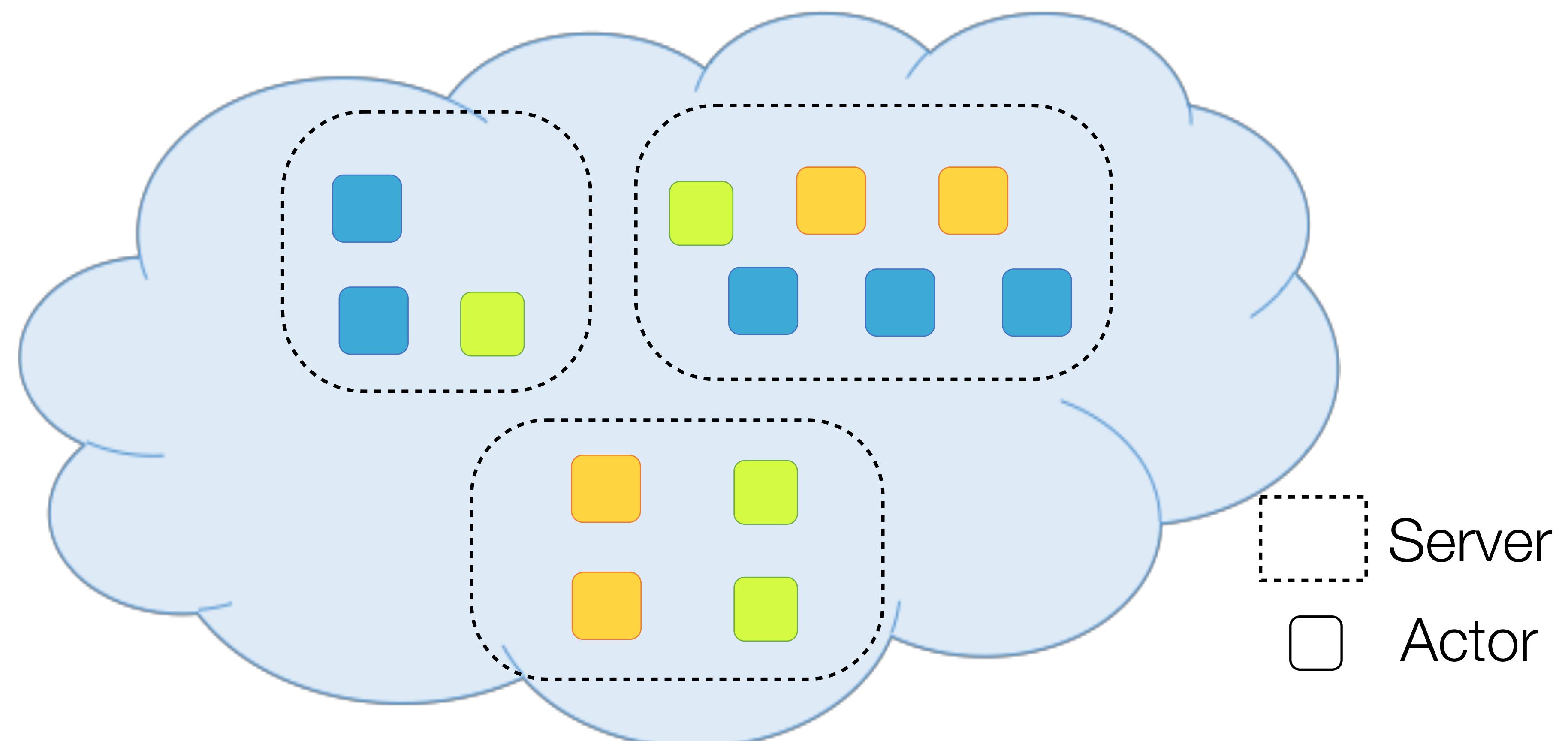


# ACTOR-BASED ELASTICITY



# ACTOR-BASED ELASTICITY

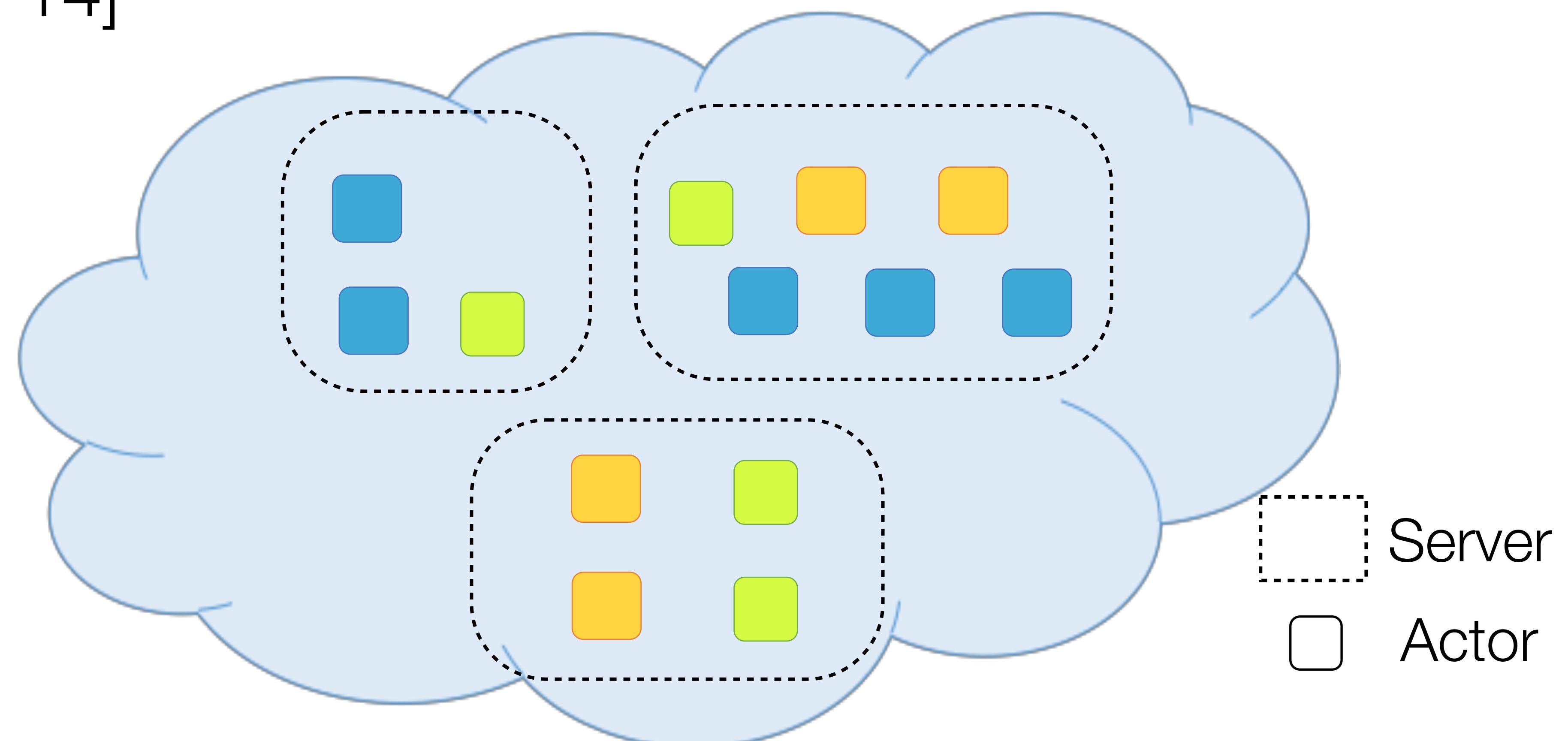
Orleans [SoCC'13, Eurosys'16]



# ACTOR-BASED ELASTICITY

Orleans [SoCC'13, Eurosys'16]

EventWave [SoCC'14]

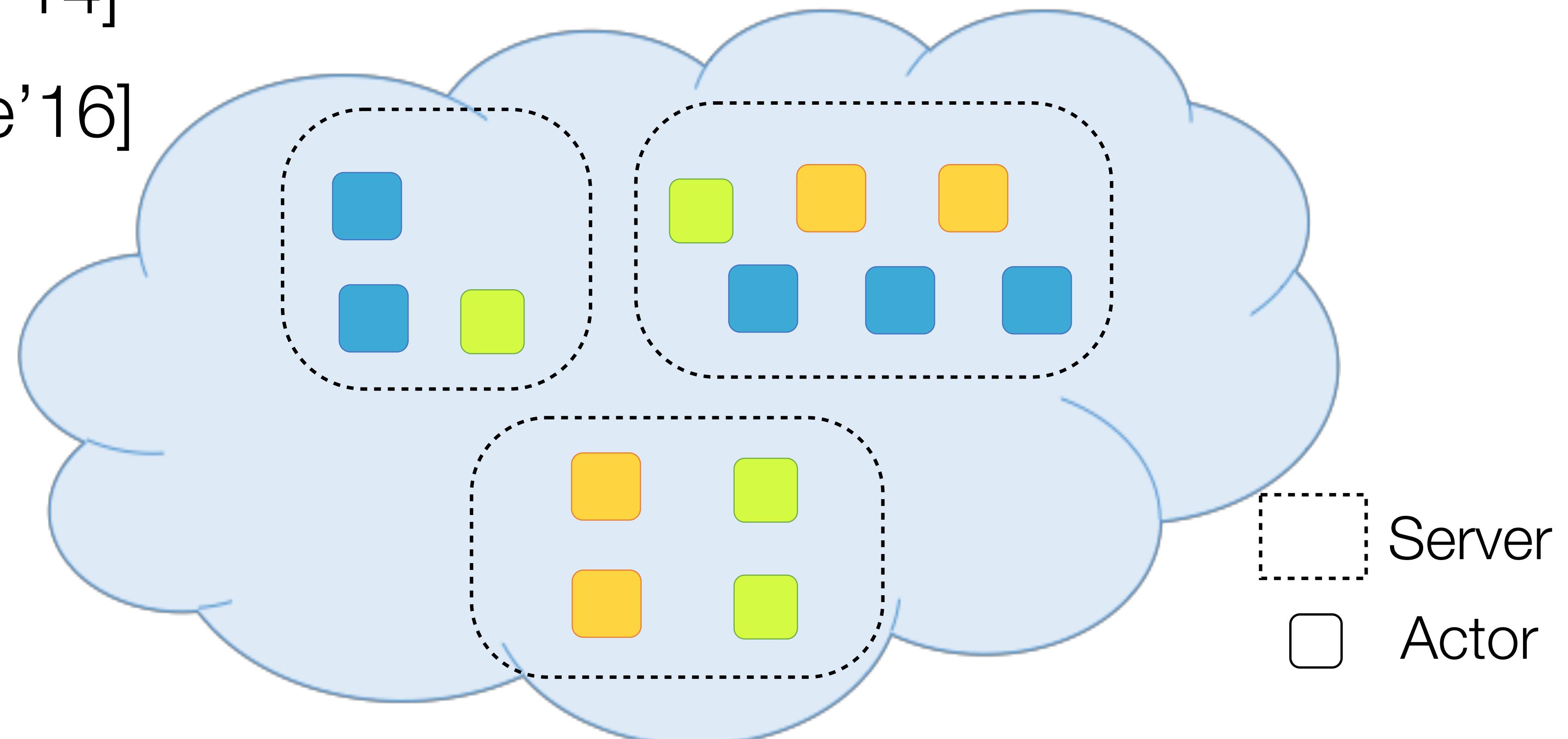


# ACTOR-BASED ELASTICITY

Orleans [SoCC'13, Eurosys'16]

EventWave [SoCC'14]

AEON [Middleware'16]

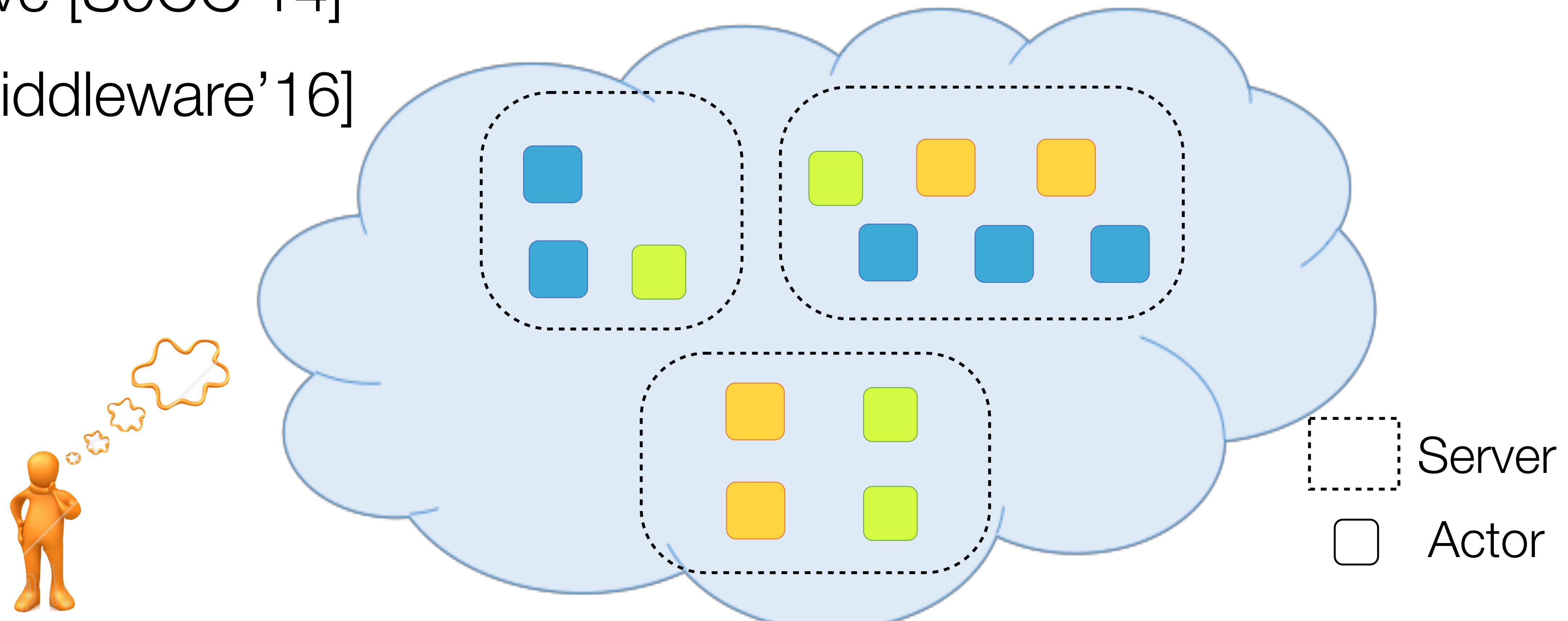


# ACTOR-BASED ELASTICITY

Orleans [SoCC'13, Eurosys'16]

EventWave [SoCC'14]

AEON [Middleware'16]

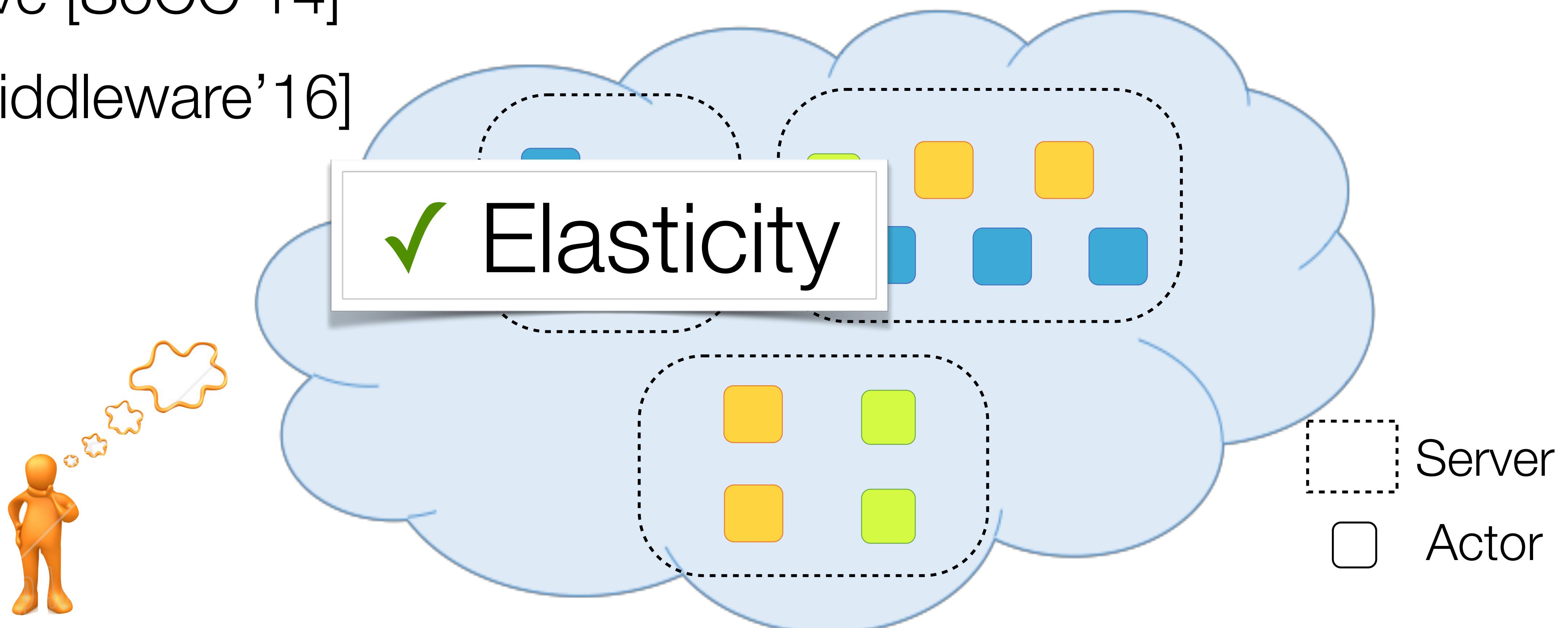


# ACTOR-BASED ELASTICITY

Orleans [SoCC'13, Eurosys'16]

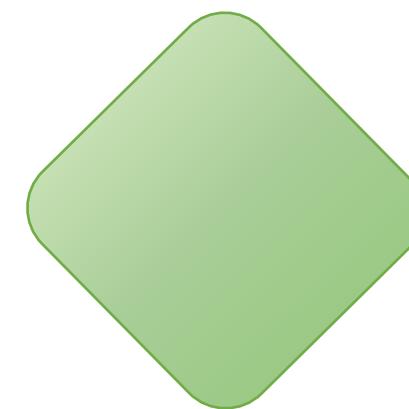
EventWave [SoCC'14]

AEON [Middleware'16]



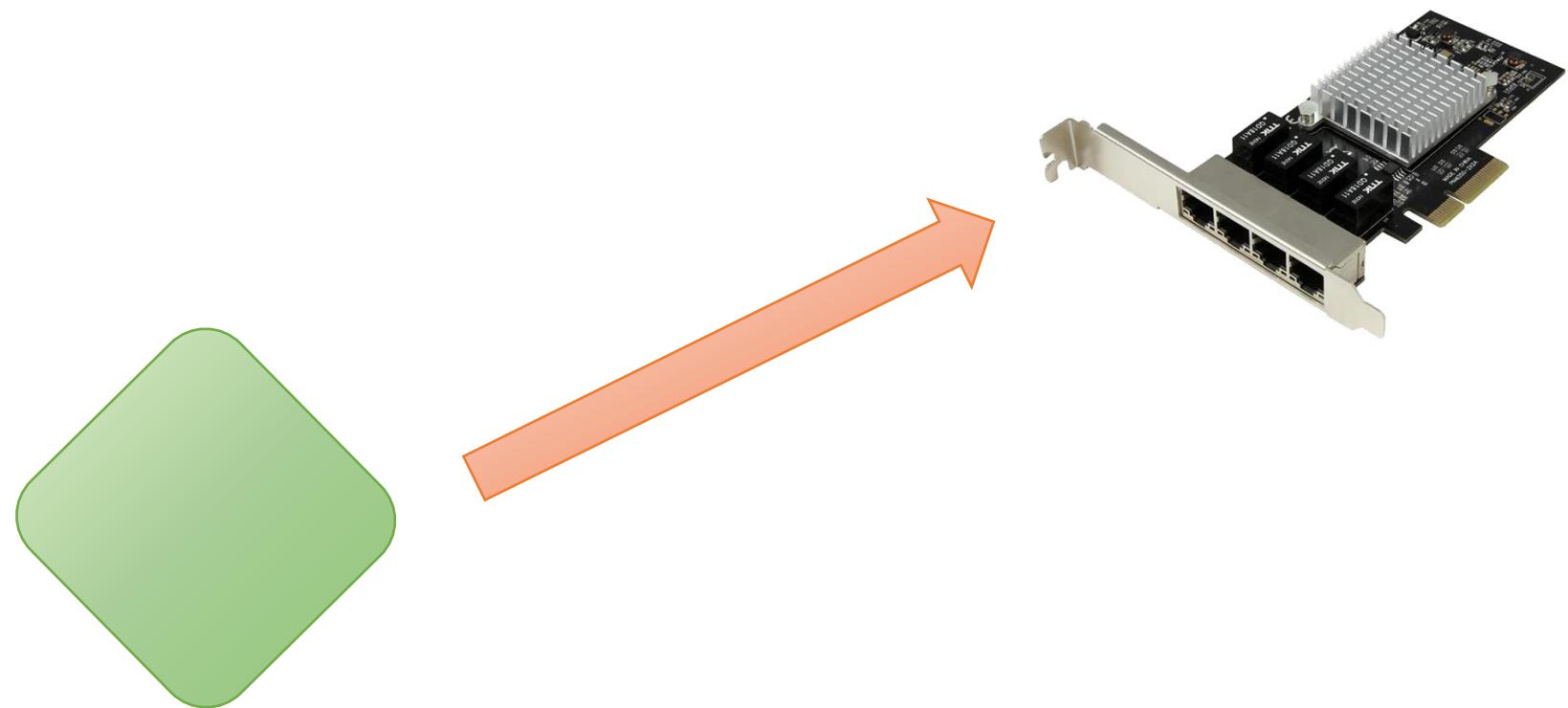
# WHERE TO PLACE ACTORS?

Resources requirements?



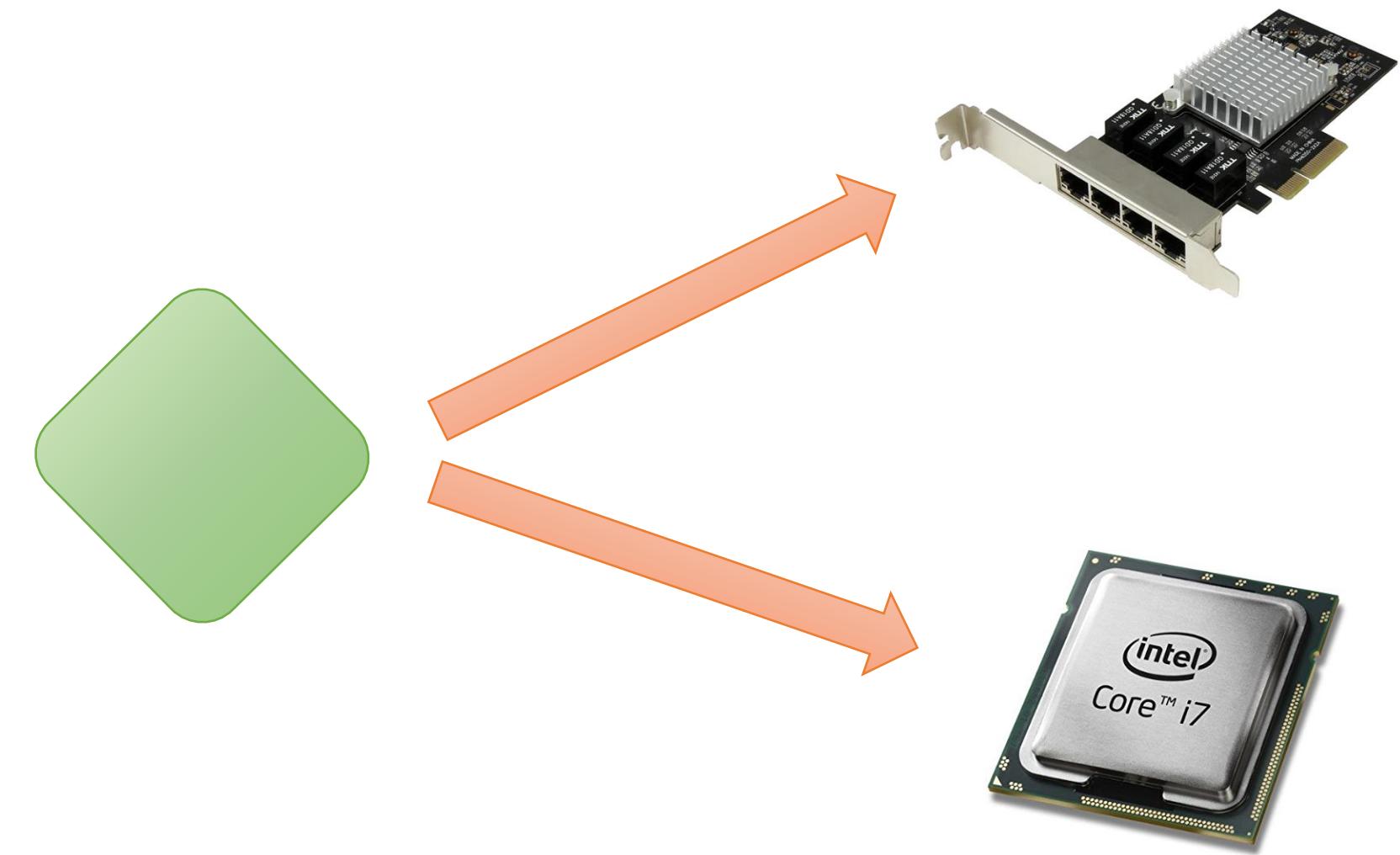
# WHERE TO PLACE ACTORS?

Resources requirements?



# WHERE TO PLACE ACTORS?

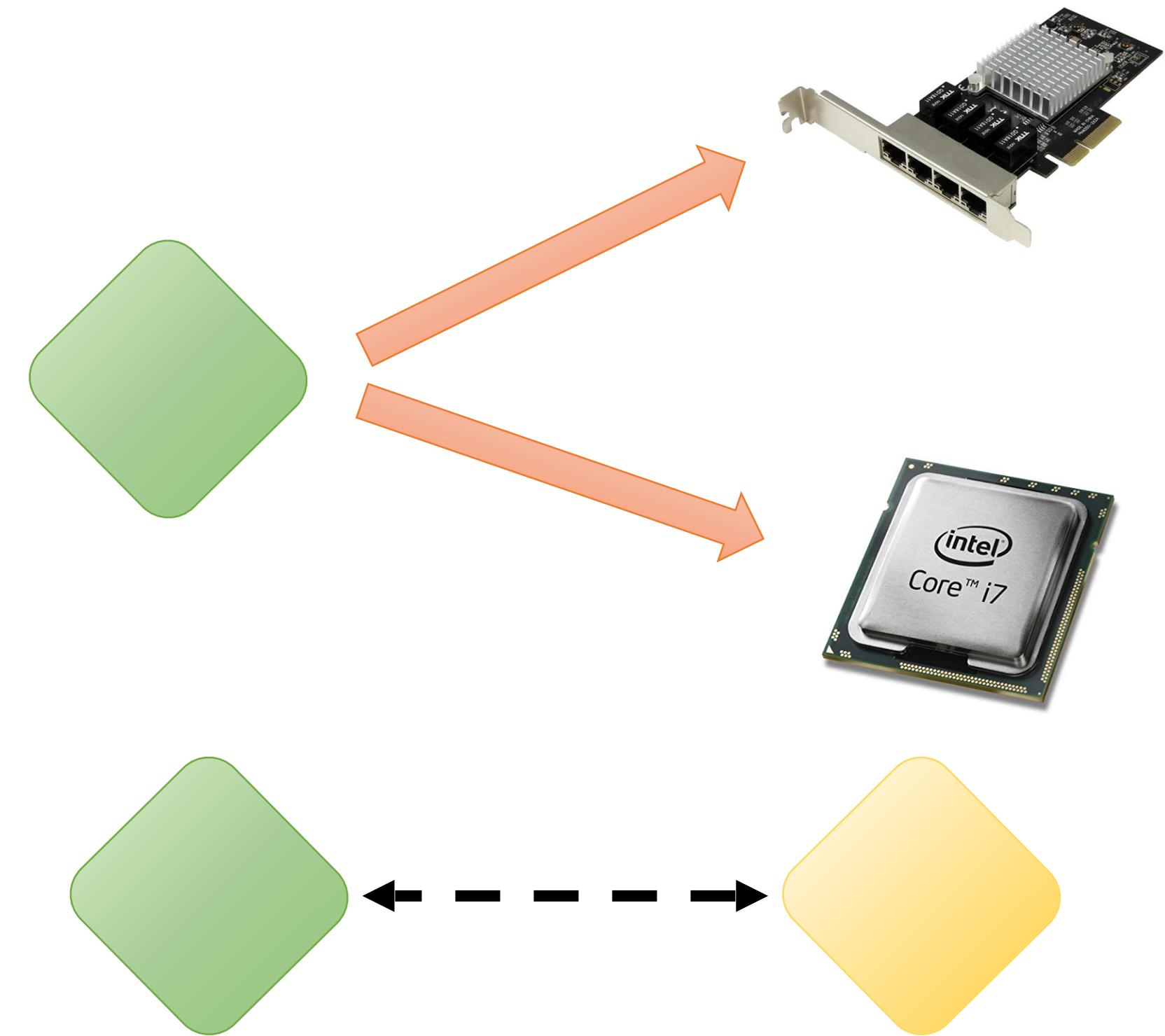
Resources requirements?



# WHERE TO PLACE ACTORS?

Resources requirements?

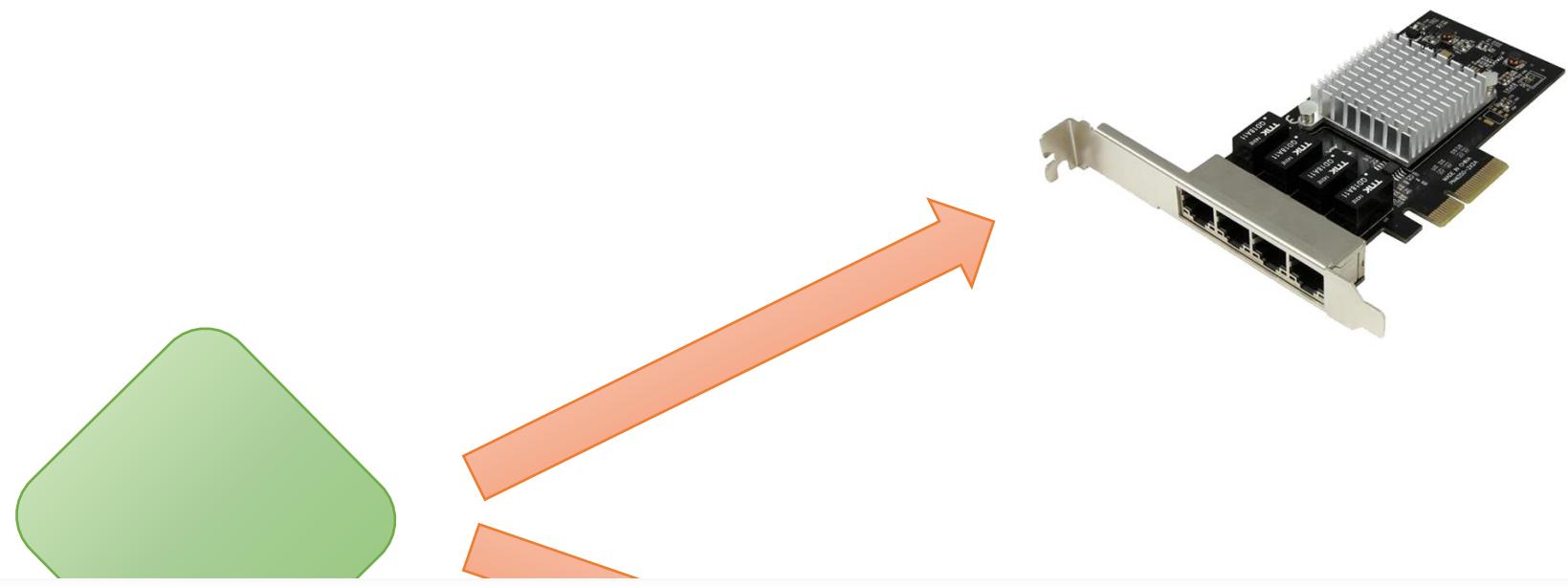
Actor Interactions?



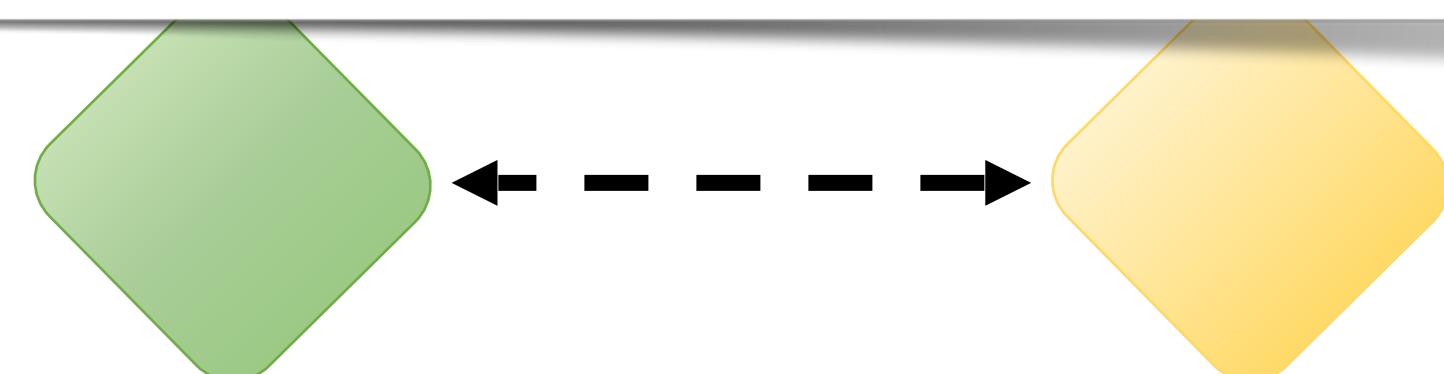
# WHERE TO PLACE ACTORS?

Resources requirements?

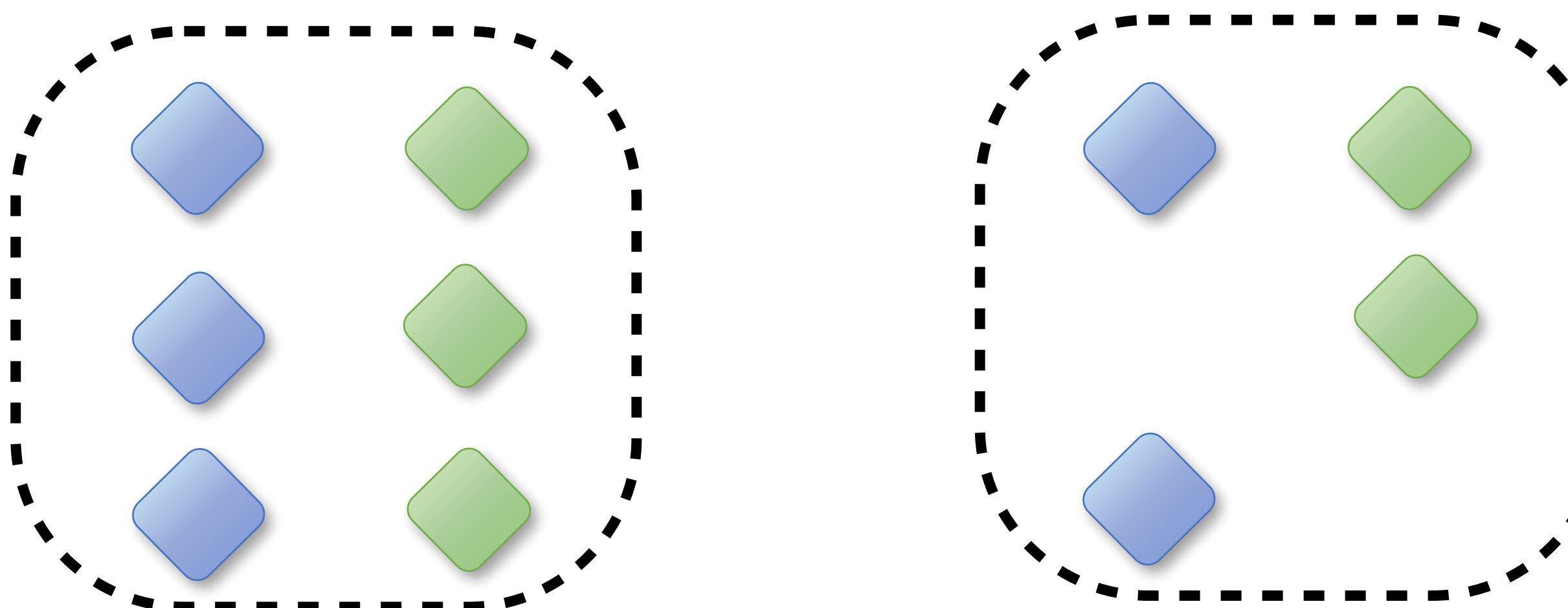
- ▶ Programmers can't control actor placement
- ▶ All actors are treated equally



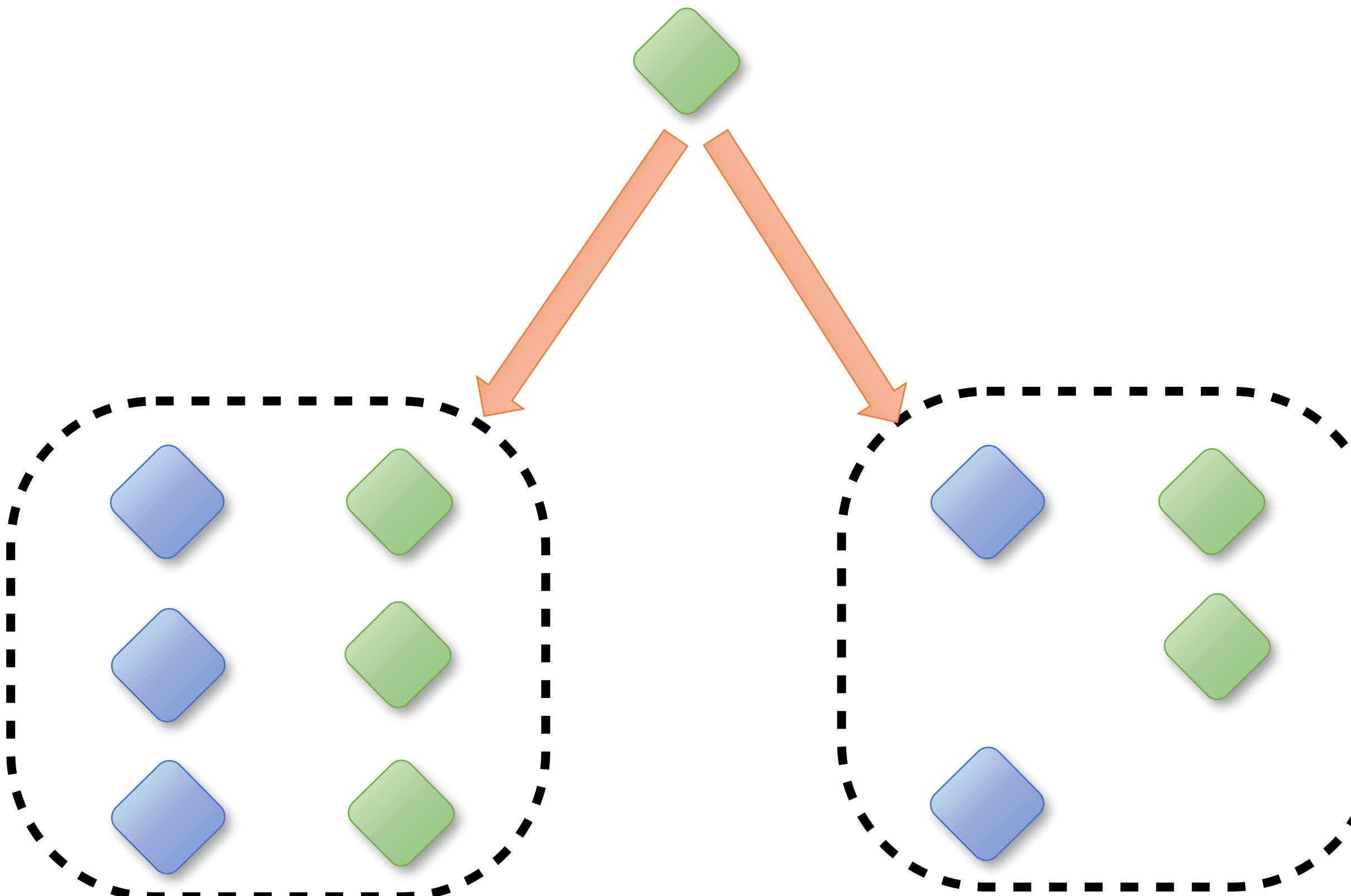
Actor Interactions?



# ACTOR PLACEMENT CONSIDERATIONS



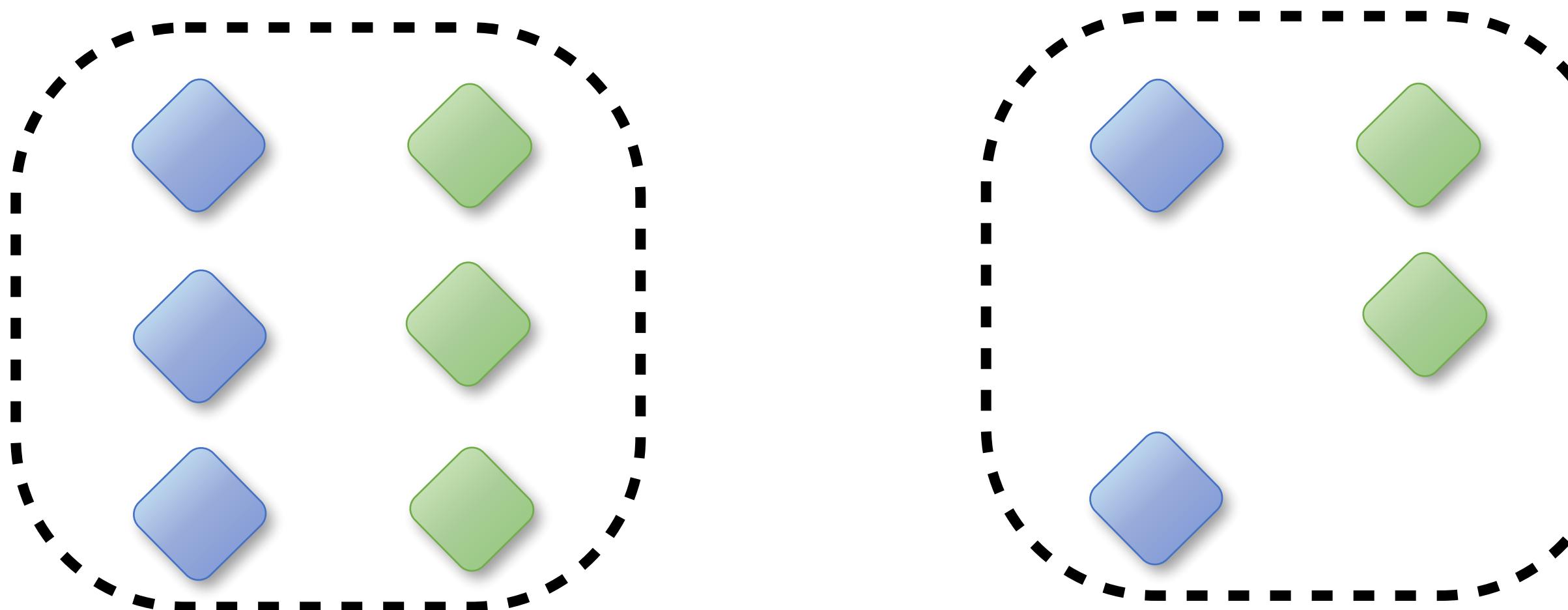
# ACTOR PLACEMENT CONSIDERATIONS



✓ Resource requirements

# ACTOR PLACEMENT CONSIDERATIONS

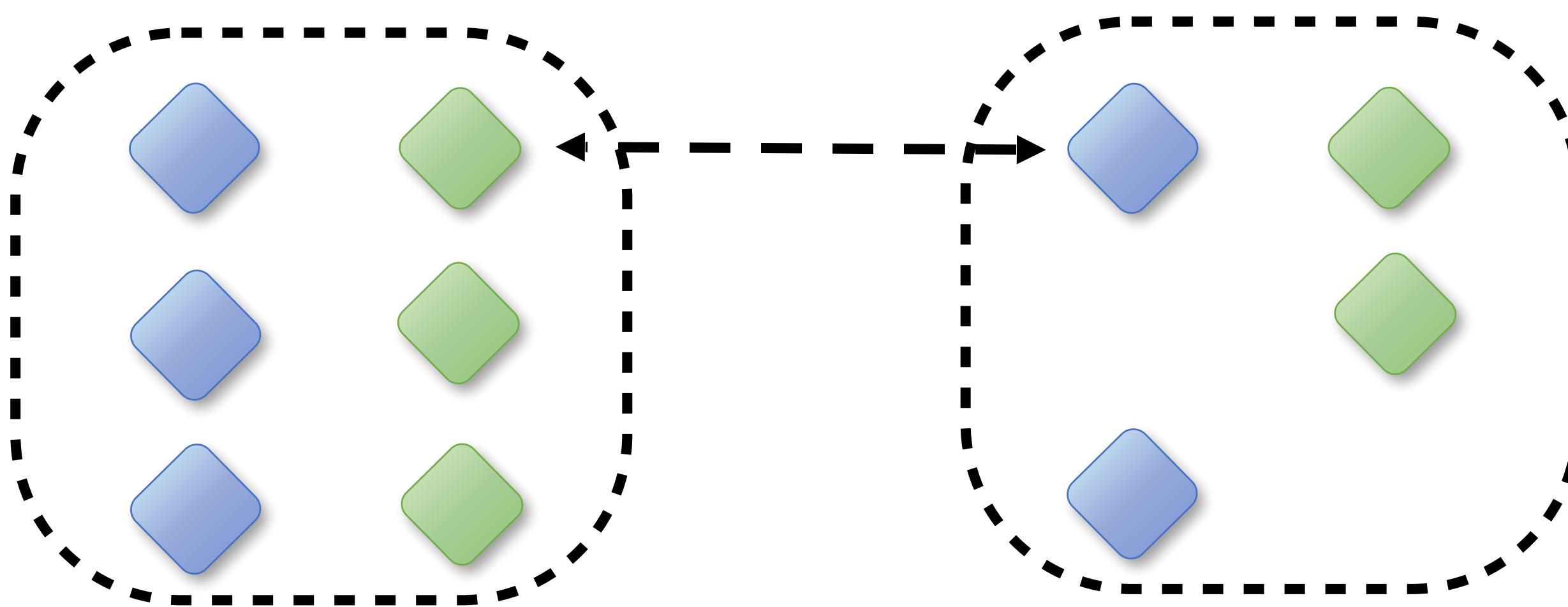
✓ Resource requirements



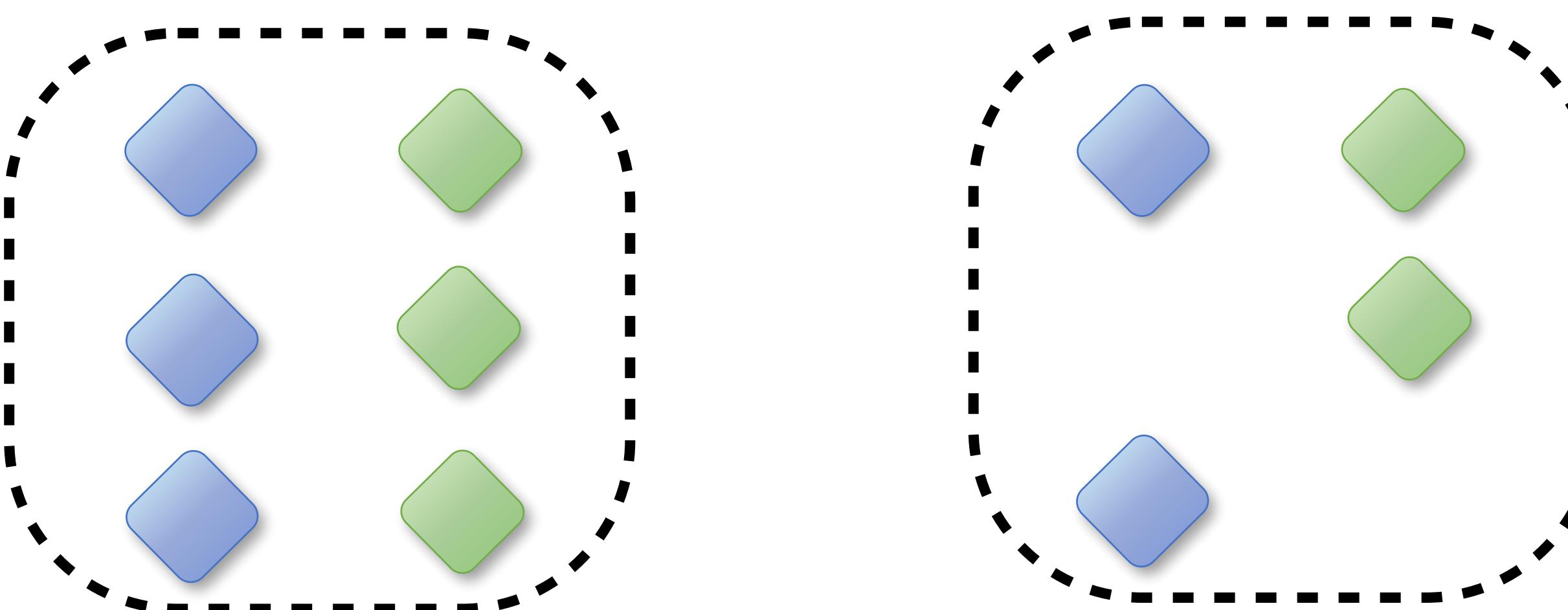
# ACTOR PLACEMENT CONSIDERATIONS

✓ Resource requirements

✓ Message passing



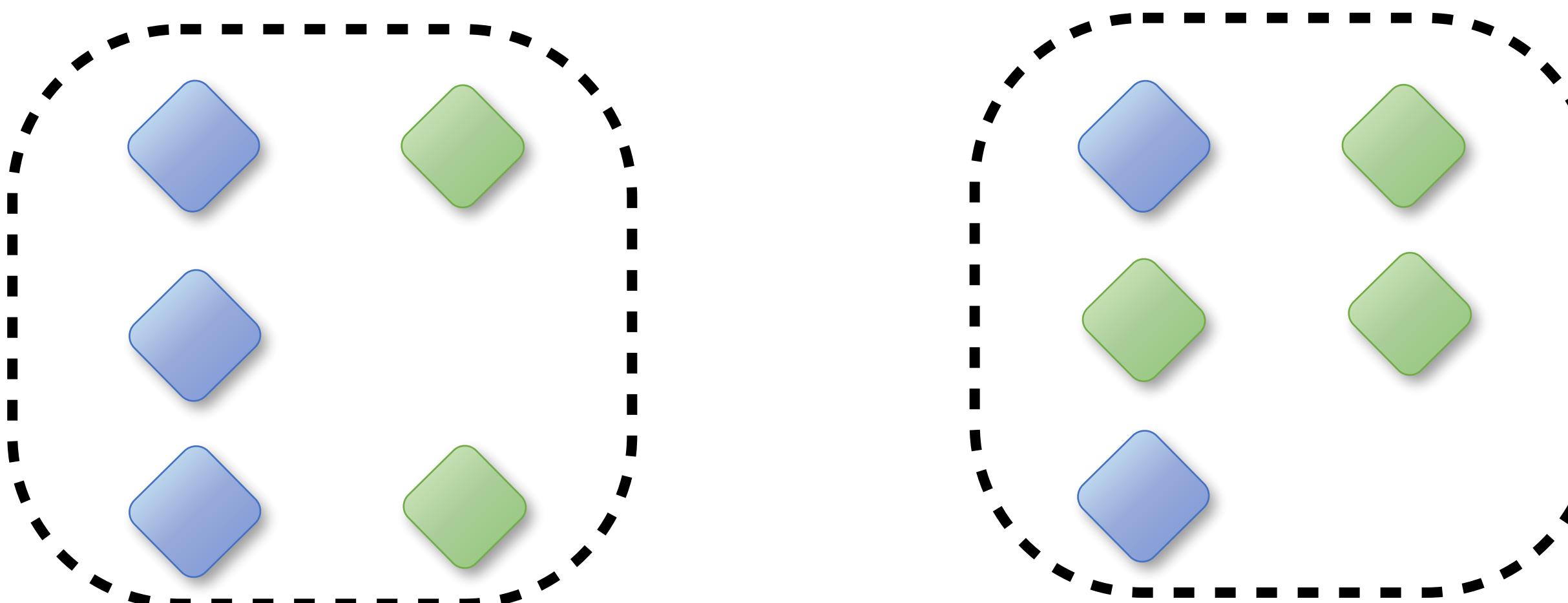
# ACTOR PLACEMENT CONSIDERATIONS



✓ Resource requirements

✓ Message passing

# ACTOR PLACEMENT CONSIDERATIONS



✓ Resource requirements

✓ Message passing

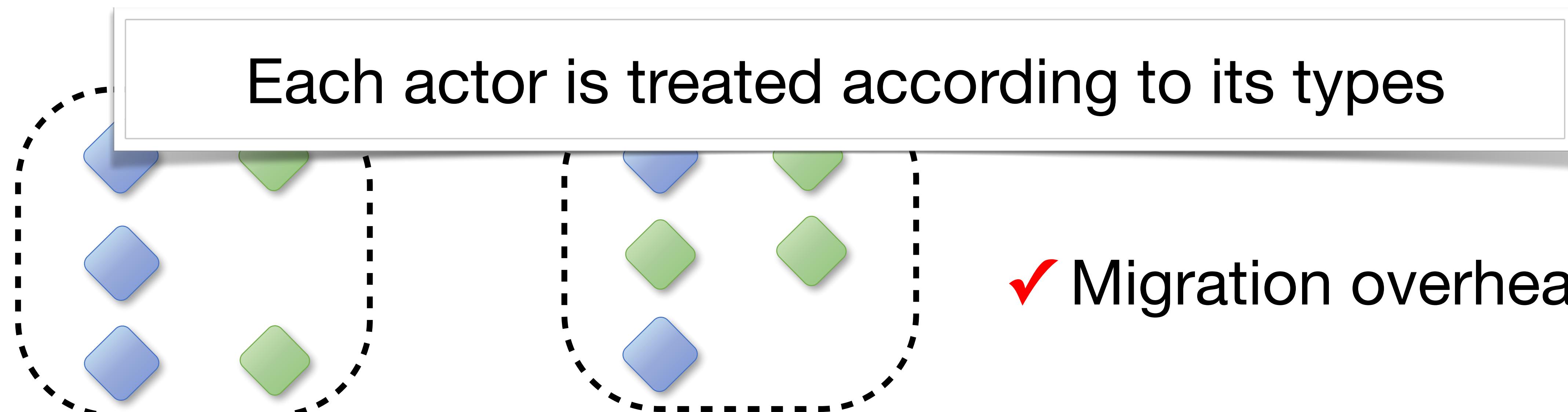
✓ Migration overhead

# ACTOR PLACEMENT CONSIDERATIONS

✓ Resource requirements

Each actor is treated according to its types

✓ Migration overhead



# PROGRAMMABLE ELASTICITY

- ▶ Why programmable?

# PROGRAMMABLE ELASTICITY

- ▶ Why programmable?
- ▶ Programmer knows application requirements

# PROGRAMMABLE ELASTICITY

- ▶ Why programmable?
  - ▶ Programmer knows application requirements
  - ▶ Cost can be considered at application design time

# PROGRAMMABLE ELASTICITY

- ▶ Why programmable?
  - ▶ Programmer knows application requirements
  - ▶ Cost can be considered at application design time
- ▶ Design principles

# PROGRAMMABLE ELASTICITY

- ▶ Why programmable?
  - ▶ Programmer knows application requirements
  - ▶ Cost can be considered at application design time
- ▶ Design principles
  - ▶ Preserve transparent placement of actors

# PROGRAMMABLE ELASTICITY

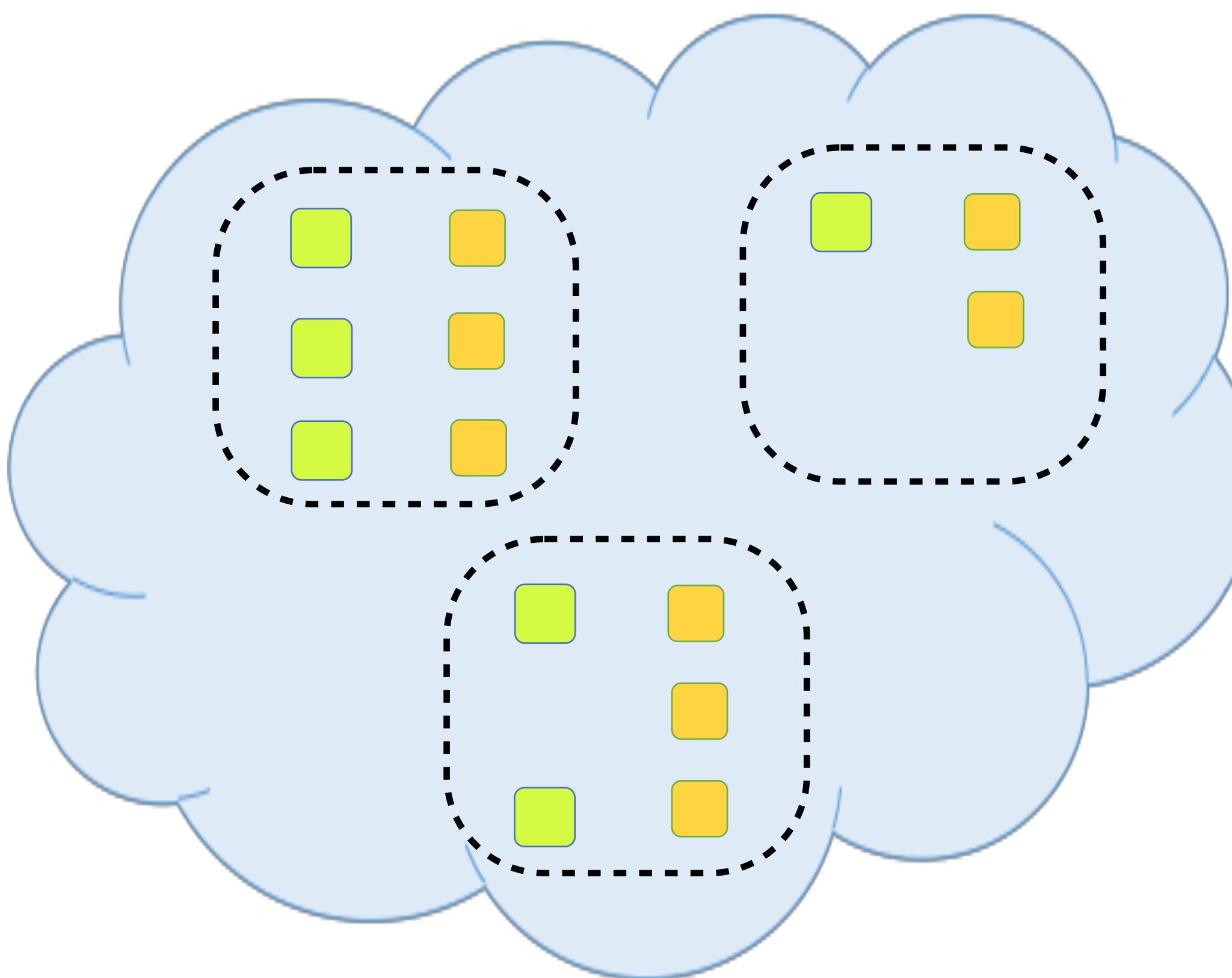
- ▶ Why programmable?
  - ▶ Programmer knows application requirements
  - ▶ Cost can be considered at application design time
- ▶ Design principles
  - ▶ Preserve transparent placement of actors
  - ▶ Loose coupling of elasticity policy from implementation

# PROGRAMMABLE ELASTICITY

- ▶ Why programmable?
  - ▶ Programmer knows application requirements
  - ▶ Cost can be considered at application design time
- ▶ Design principles
  - ▶ Preserve transparent placement of actors
  - ▶ Loose coupling of elasticity policy from implementation
  - ▶ Rules at the type level rather than at individual actors

# ELASTICITY PROGRAM MODEL

Runtime monitors activity



Conditions

$\text{cpu} > 50\%$

$f.\text{count} > 10^3$

$\text{ref}(\text{Child})$

Actions

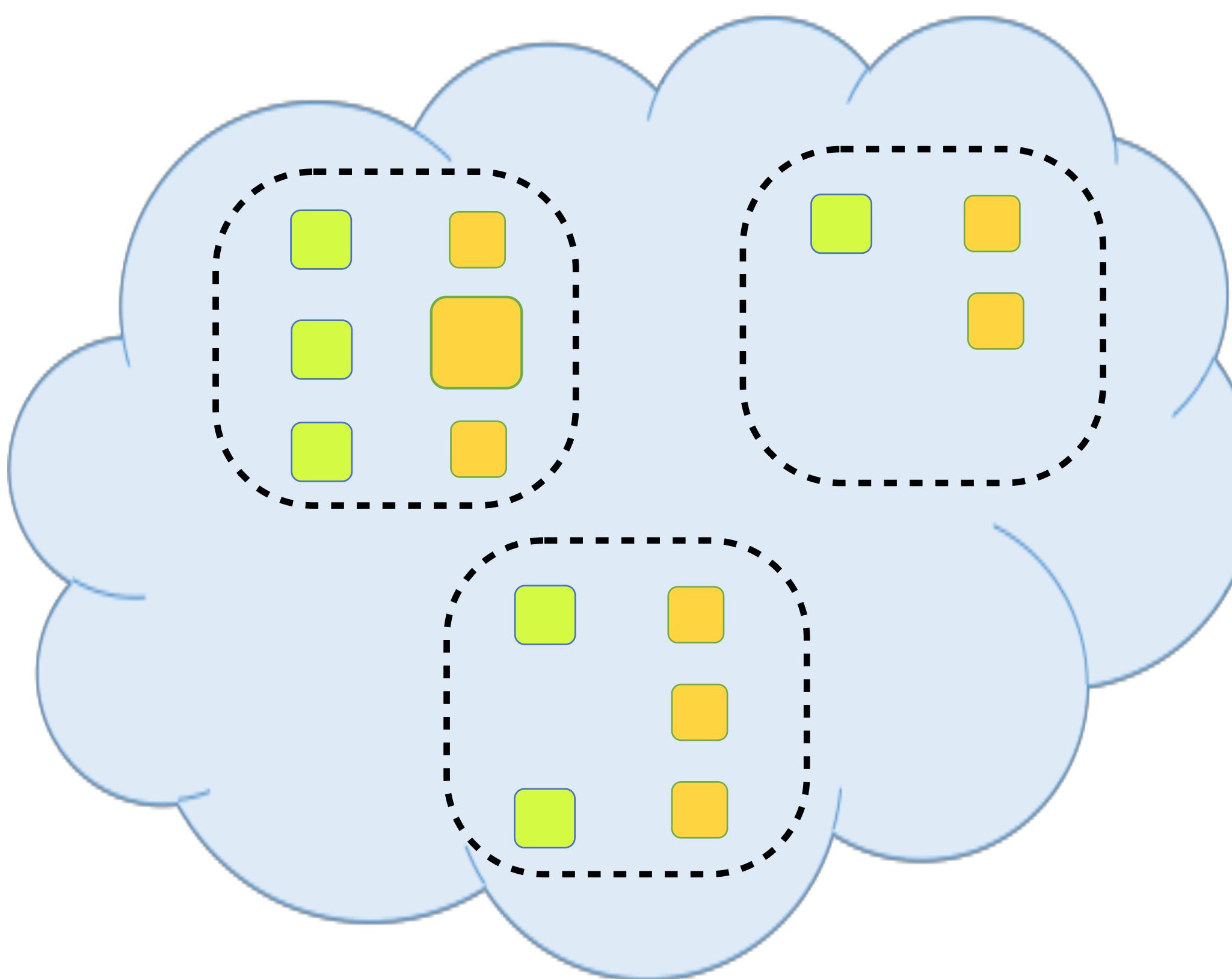
separate

isolate

collocate

# ELASTICITY PROGRAM MODEL

Runtime monitors activity

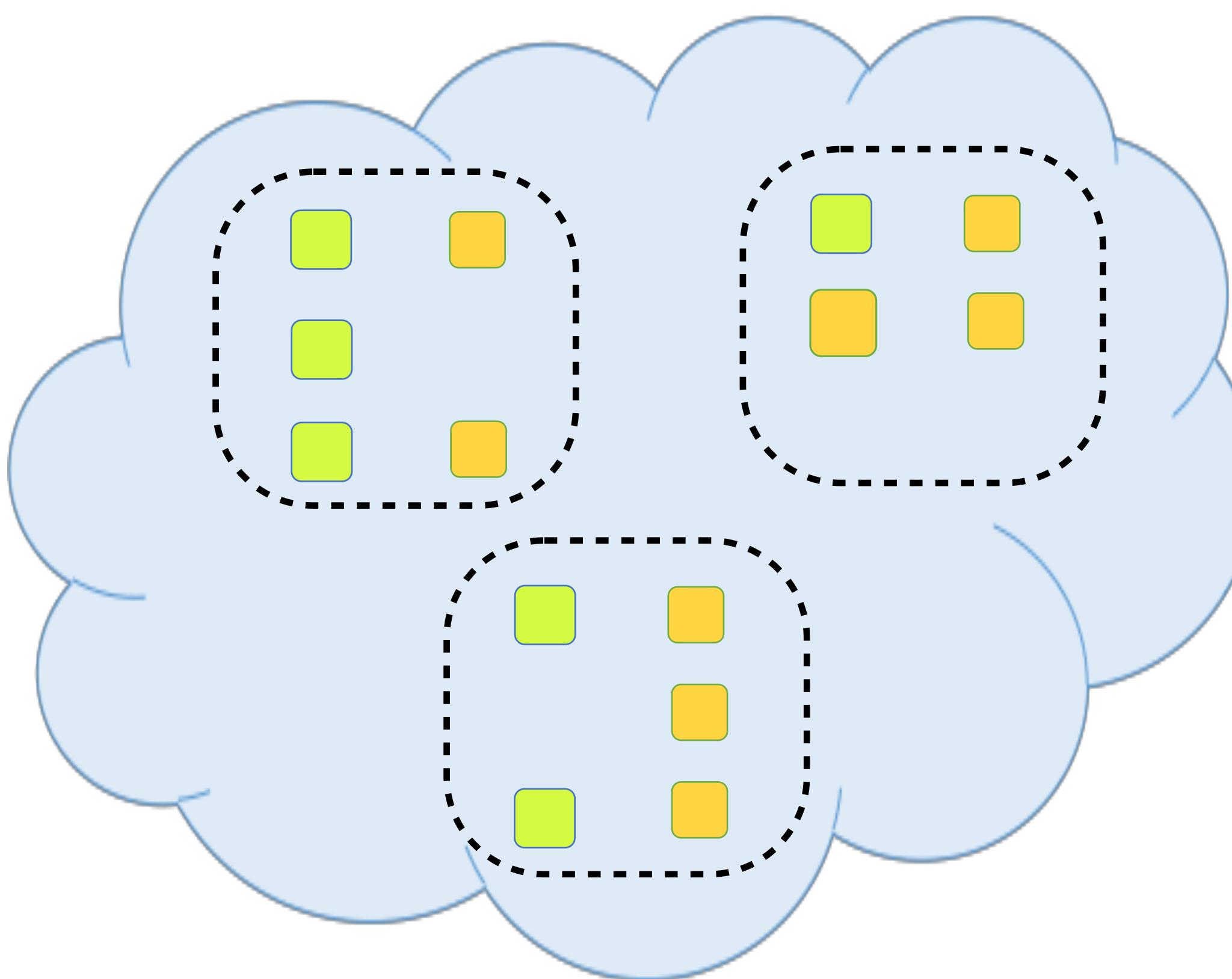


Conditions

Conditions	Actions
$\text{cpu} > 50\%$	separate
$f.\text{count} > 10^3$	isolate
$\text{ref}(\text{Child})$	collocate

# ELASTICITY PROGRAM MODEL

Runtime monitors activity



Conditions

Conditions	Actions
$\text{cpu} > 50\%$	separate
$f.\text{count} > 10^3$	isolate
$\text{ref}(\text{Child})$	collocate

# ELASTICITY SYNTAX

## Conditions

$at$ , any	actor type
$f.\text{count}$	# of calls to func. $f$
$\text{ref}(at)$	references with type $at$
cpu	cpu usage
mem	memory usage
net	network usage

## Actions

$\text{colocate}(at_1, at_2, cd)$
$\text{separate}(at_1, at_2, cd)$
$\text{pin}(at)$
$\text{isolate}(at, cd, re)$

# ELASTICITY SYNTAX

## Conditions

$at$ , any	actor type
$f.\text{count}$	# of calls to func. $f$
$\text{ref}(at)$	
cpu	cpu usage
mem	memory usage
net	network usage

## Actions

$\text{colocate}(at_1, at_2, cd)$
$\text{separate}(at_1, at_2, cd)$
$\text{pin}(at)$
$\text{unpin}(at, cd, re)$

Work in progress!

# WRAPPING UP

- ▶ Actors for the cloud (FTW!)
- ▶ Actors play well with the reactive manifesto
- ▶ Cloud computing offers easy deployment at just-the-right cost
- ▶ Actors help in managing:
  - ▶ Cost
  - ▶ Faults
  - ▶ Availability