

KOSSCON 2016

---

**ELIXIR를 이용한 분산시스템**

---

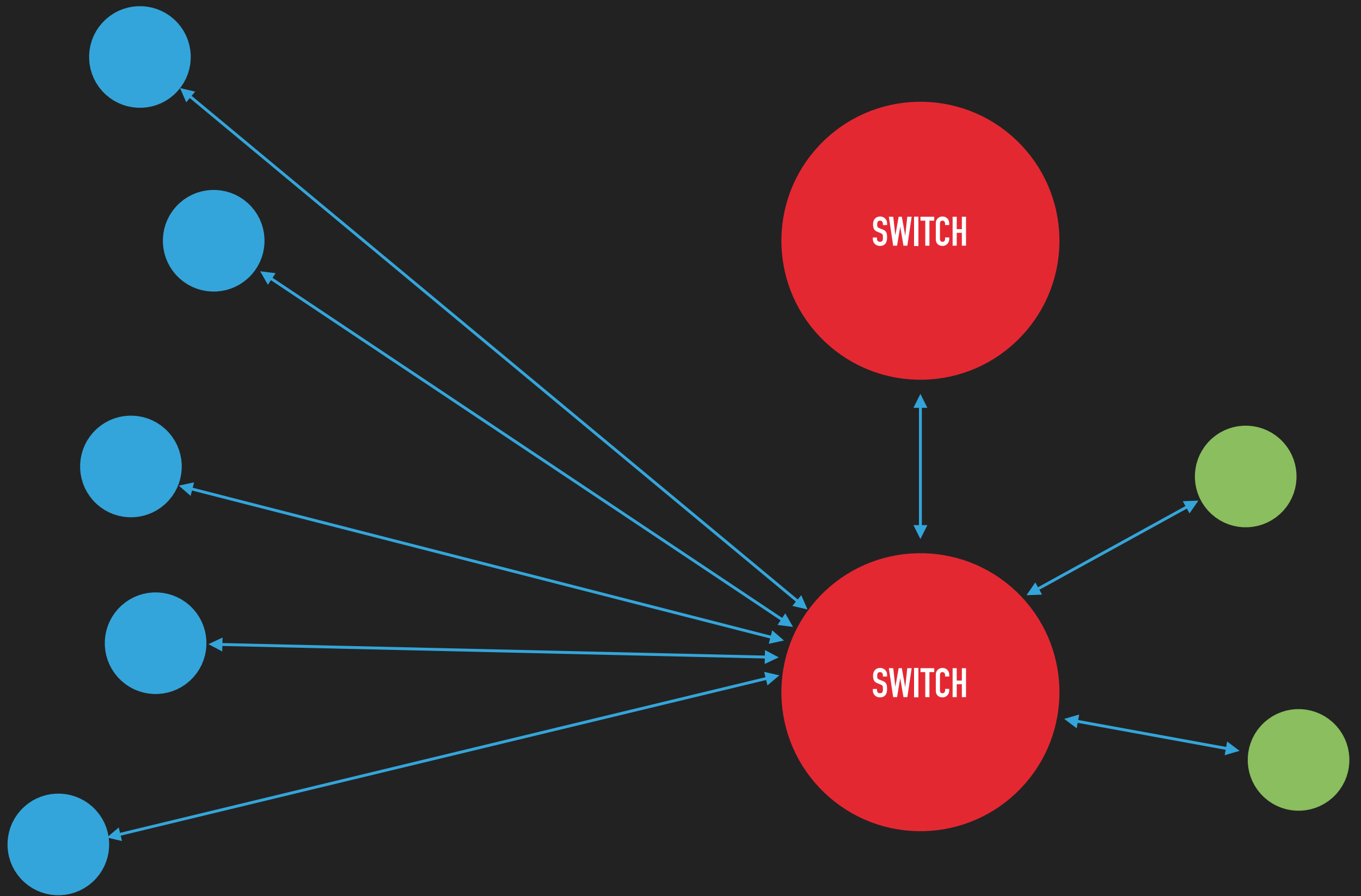
# OVERVIEW

- ▶ Erlang/OTP
- ▶ Elixir
- ▶ Distributed systems
- ▶ Wrapping up

**ERALNG/OTP**



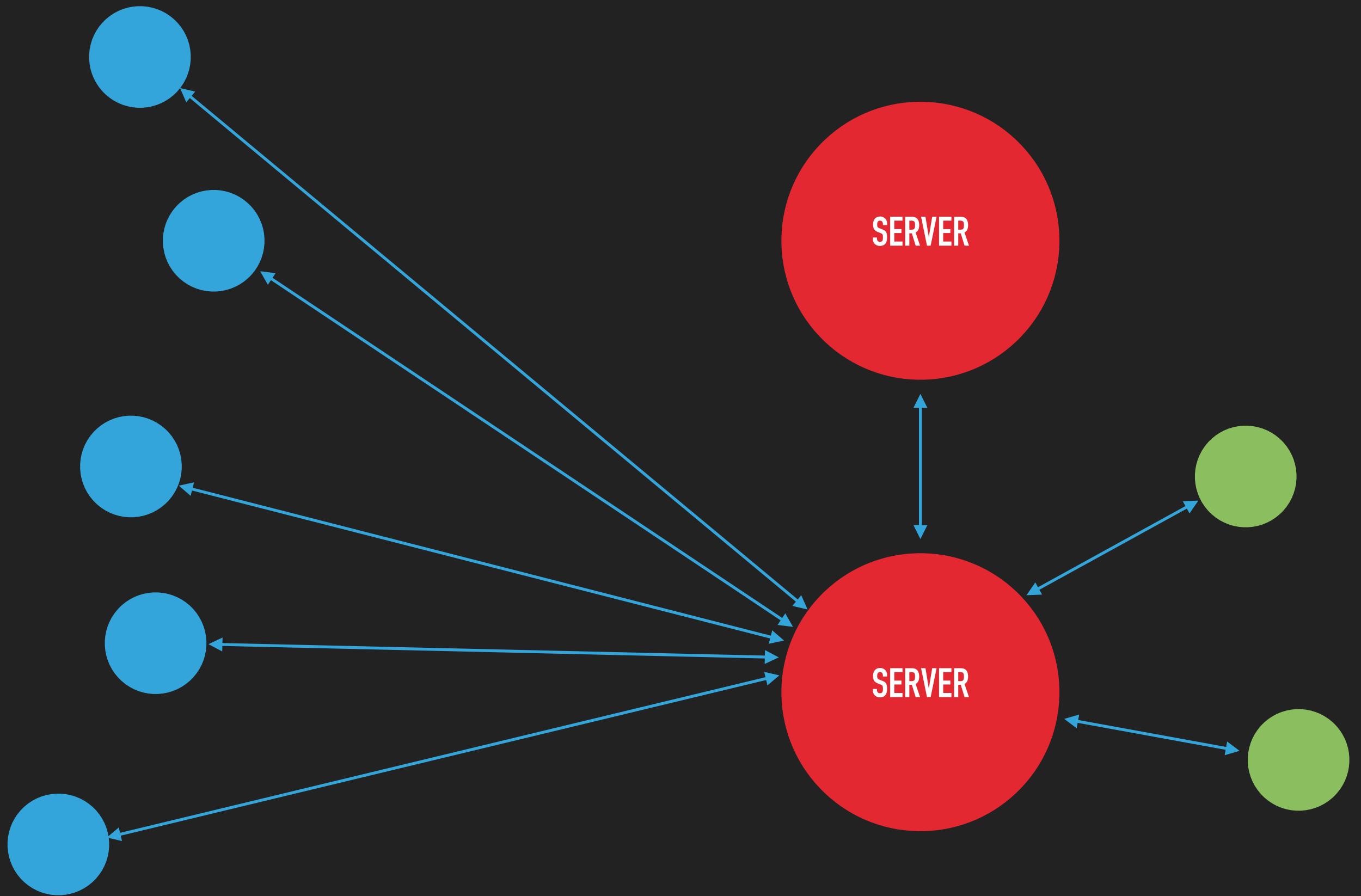




CONCURRENCY.

FAULT TOLERANCE.

DISTRIBUTION.

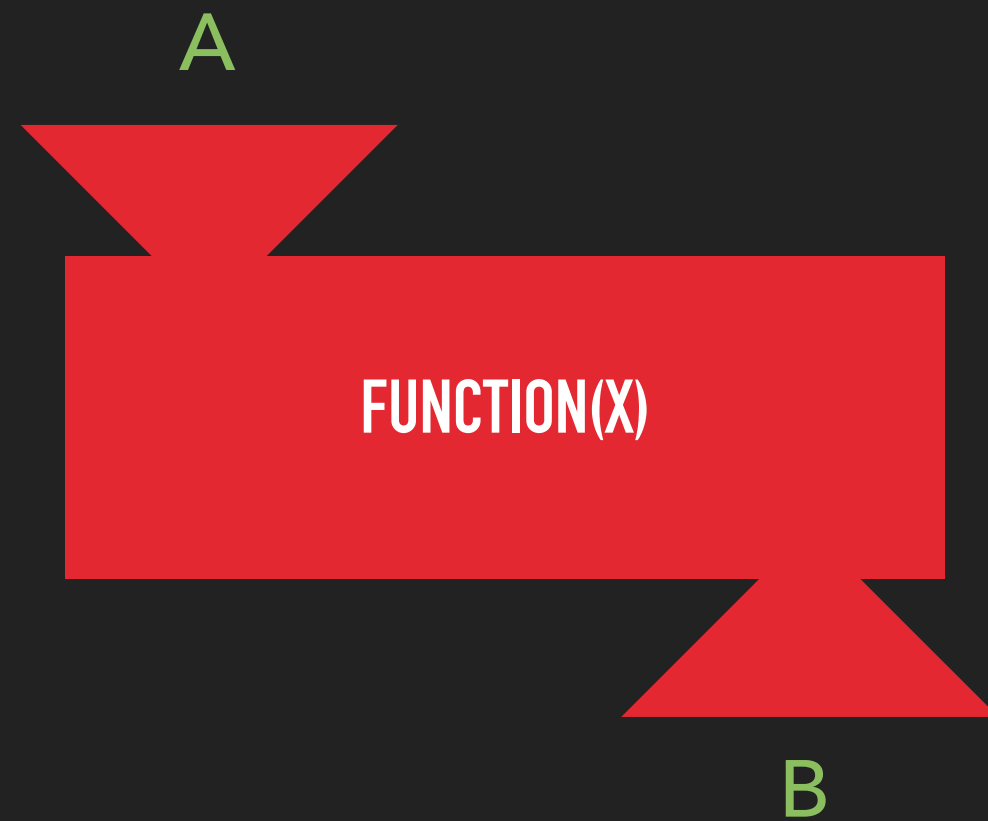




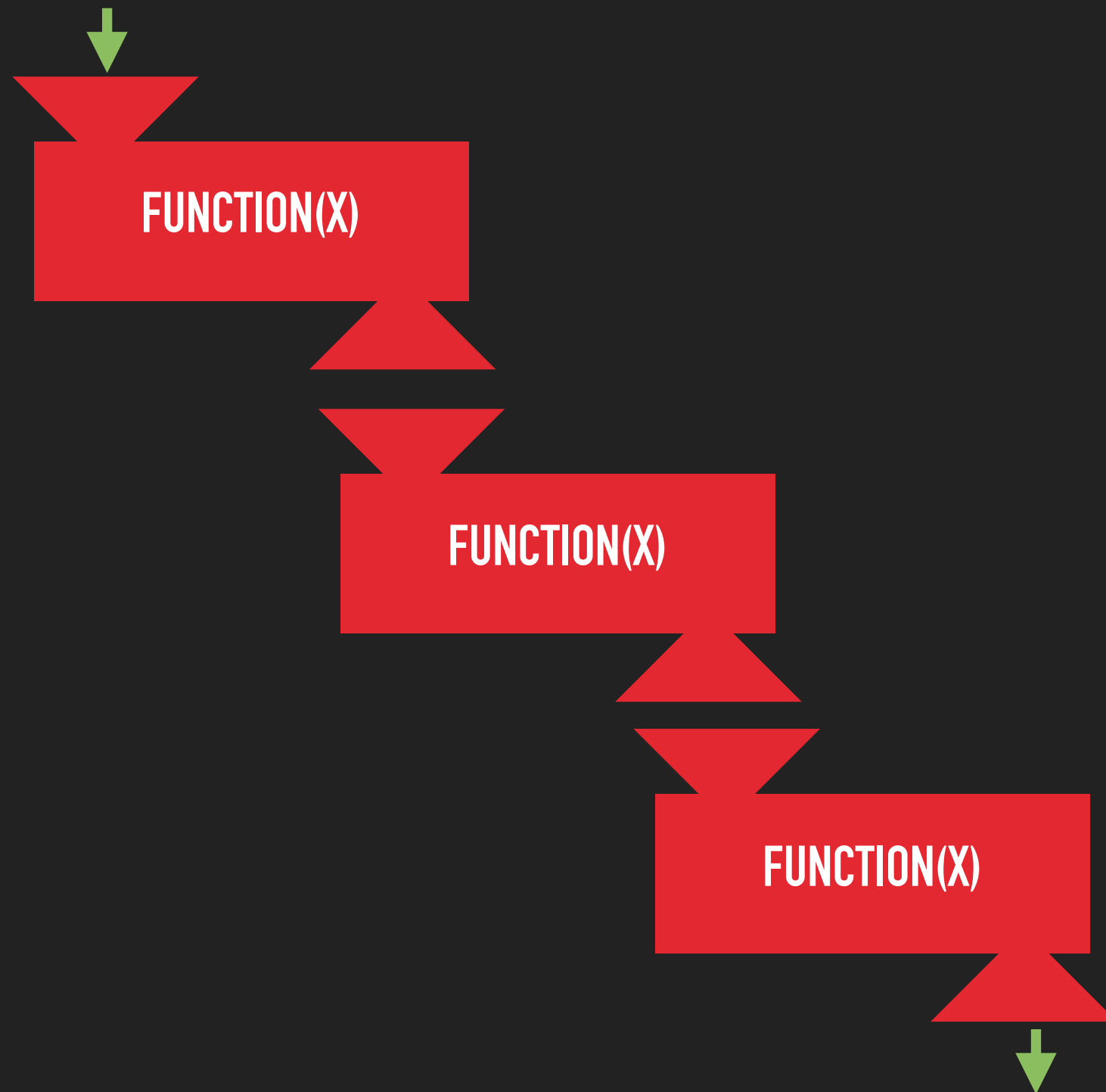
# ERLANG/OTP

- ▶ Functional language
- ▶ Immutable variables
- ▶ Strong, dynamic typing
- ▶ Actor model
- ▶ OTP
  - ▶ Behavior

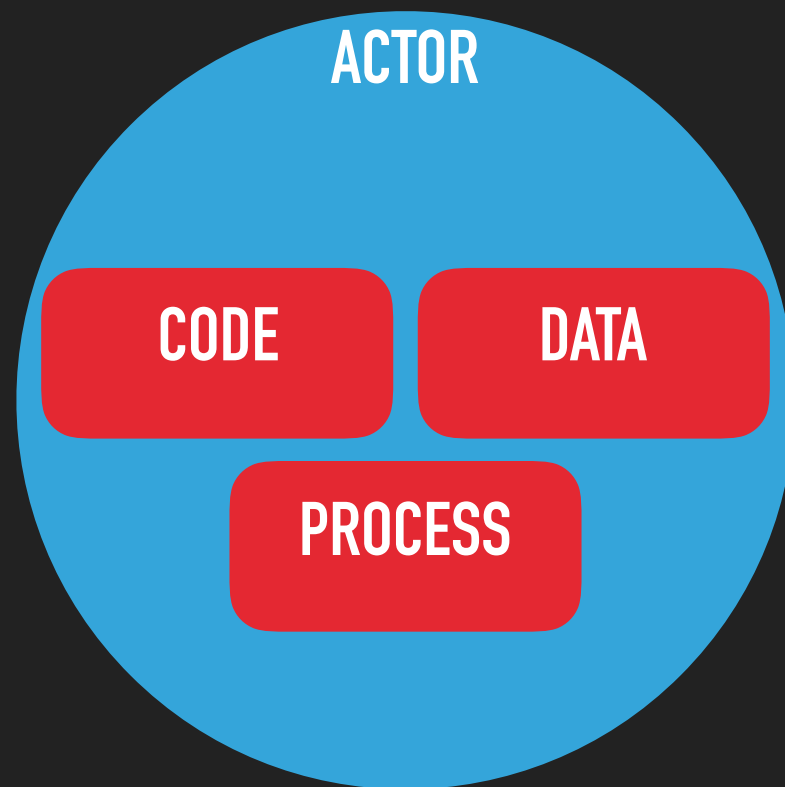
# FUNCTIONAL LANGUAGE



# FUNCTIONAL LANGUAGE

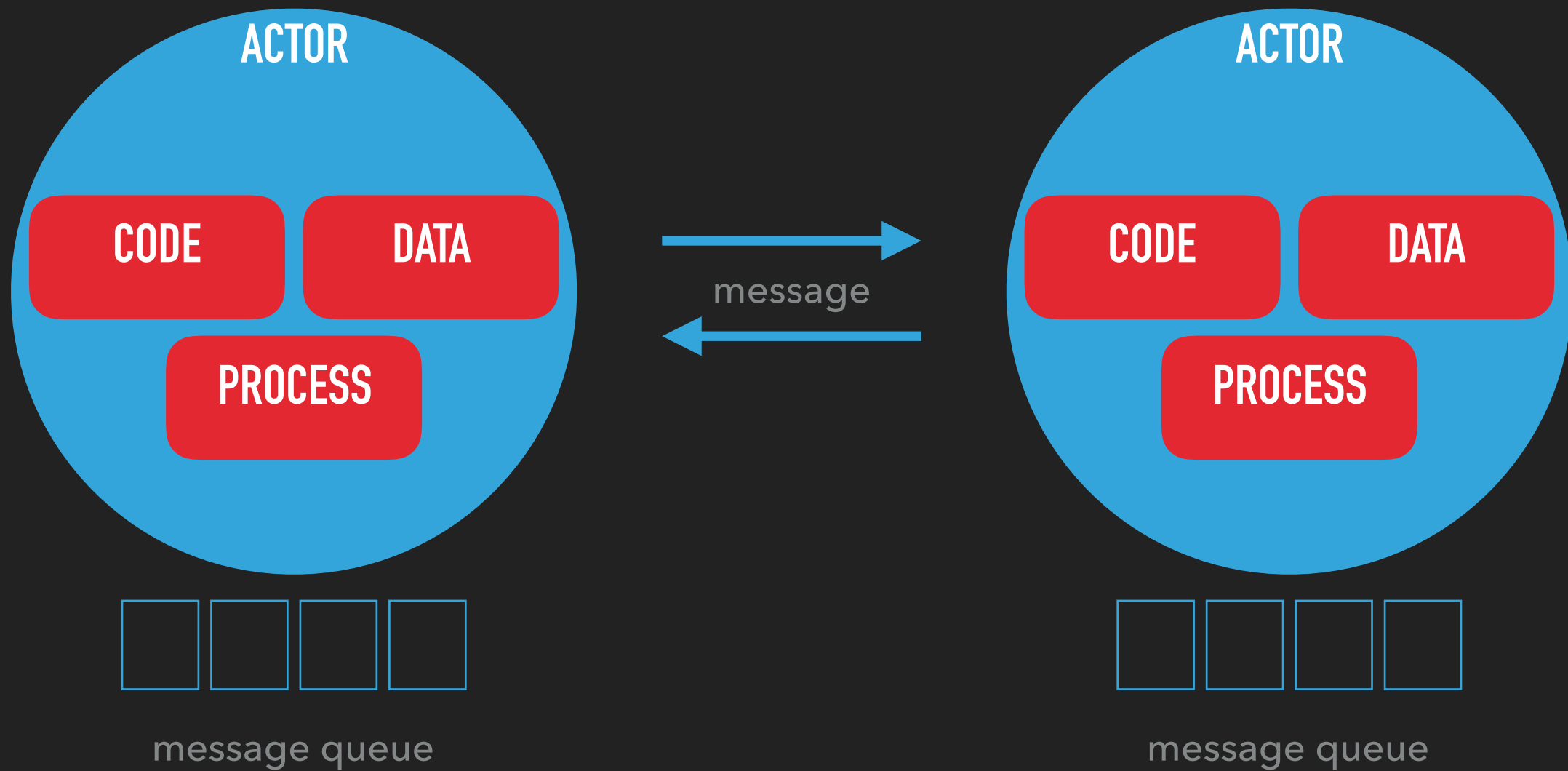


# ACTOR MODEL

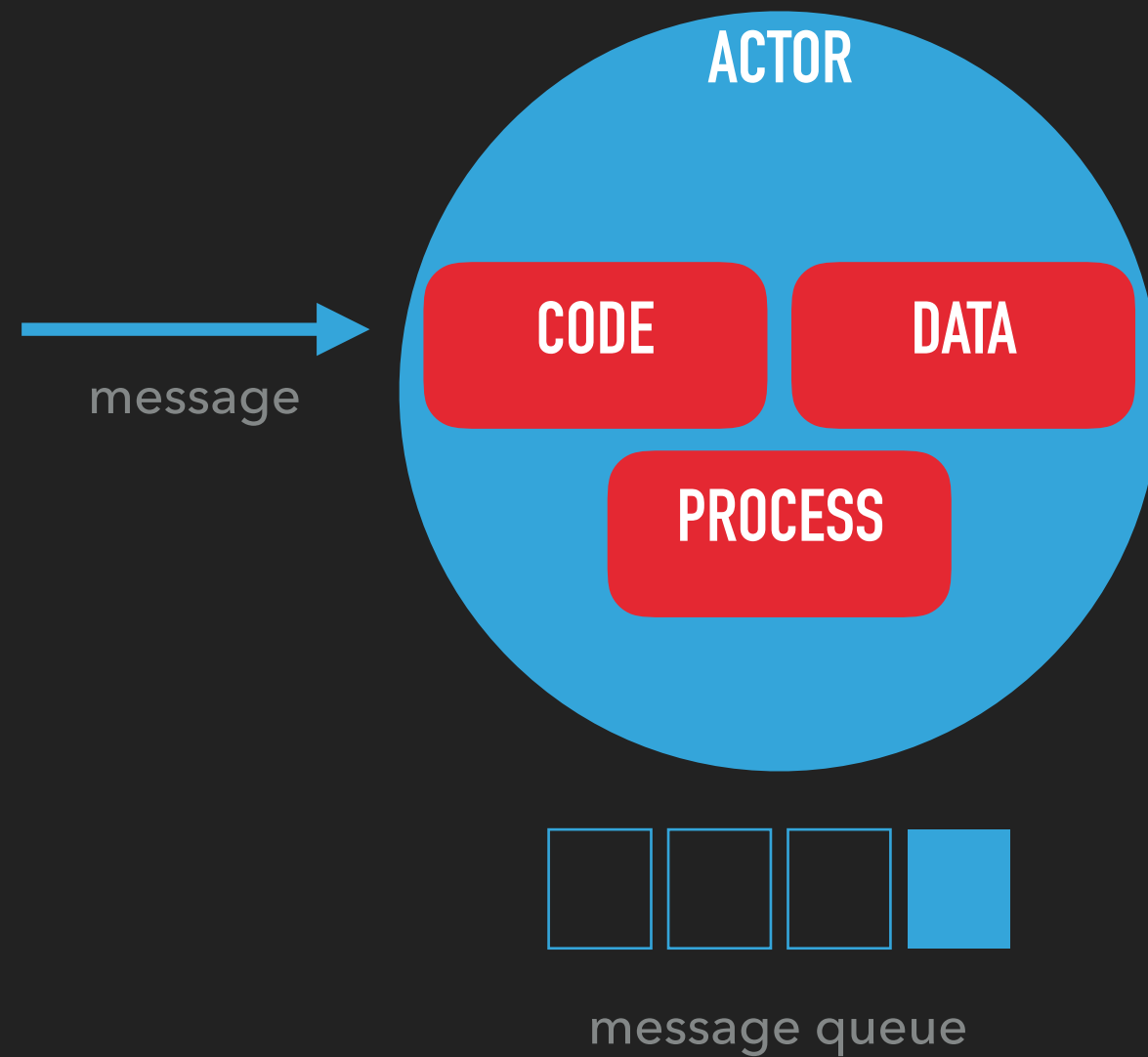


Actor = Lightweight Process + Data + Code

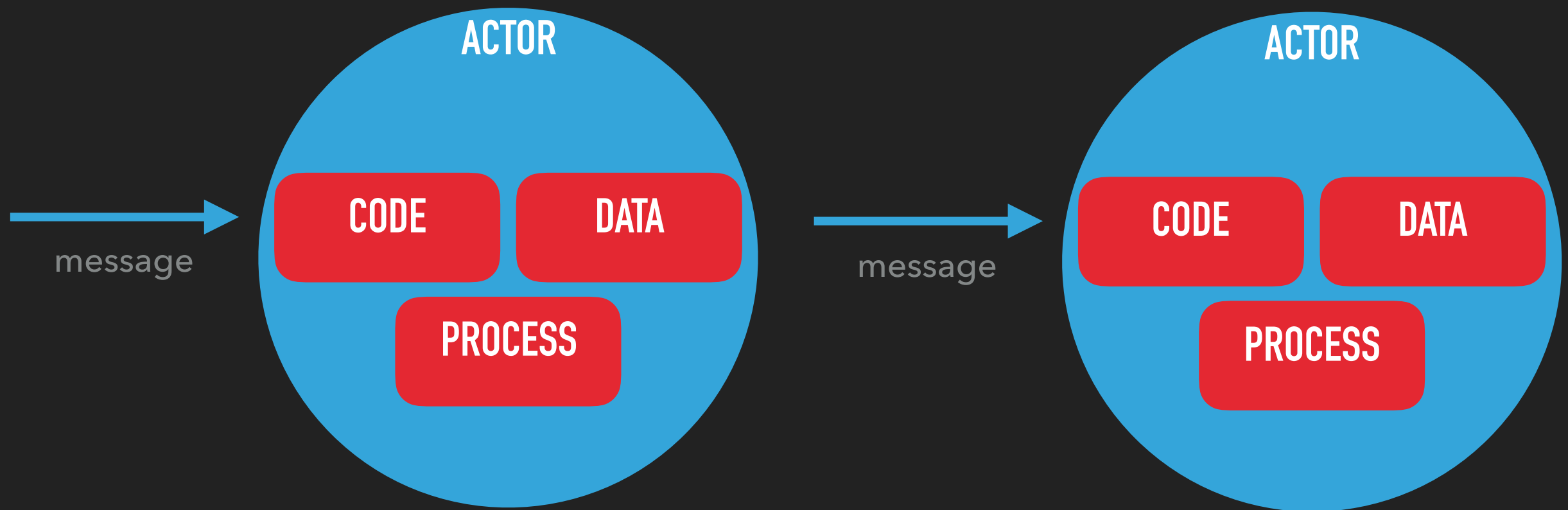
# ACTOR MODEL



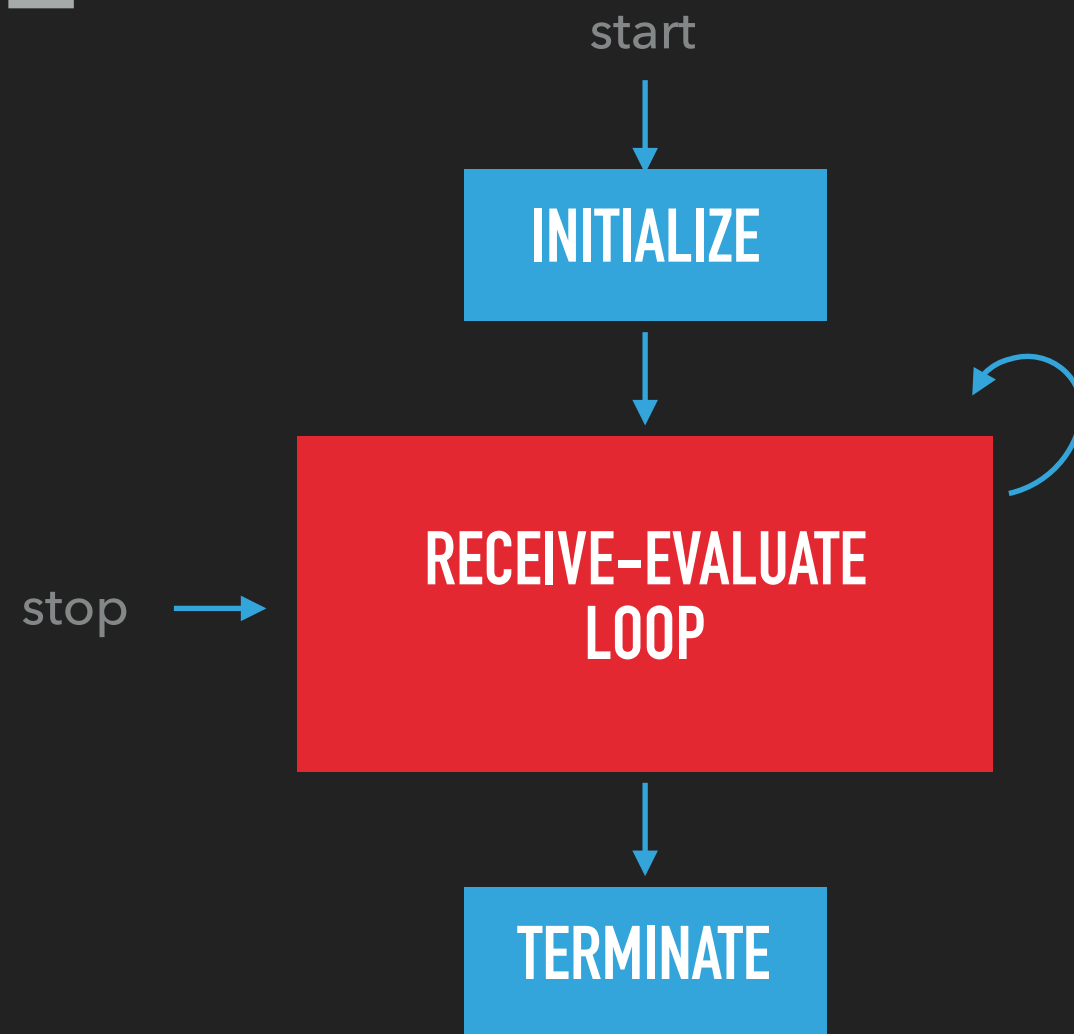
# ACTOR MODEL



# ACTOR MODEL



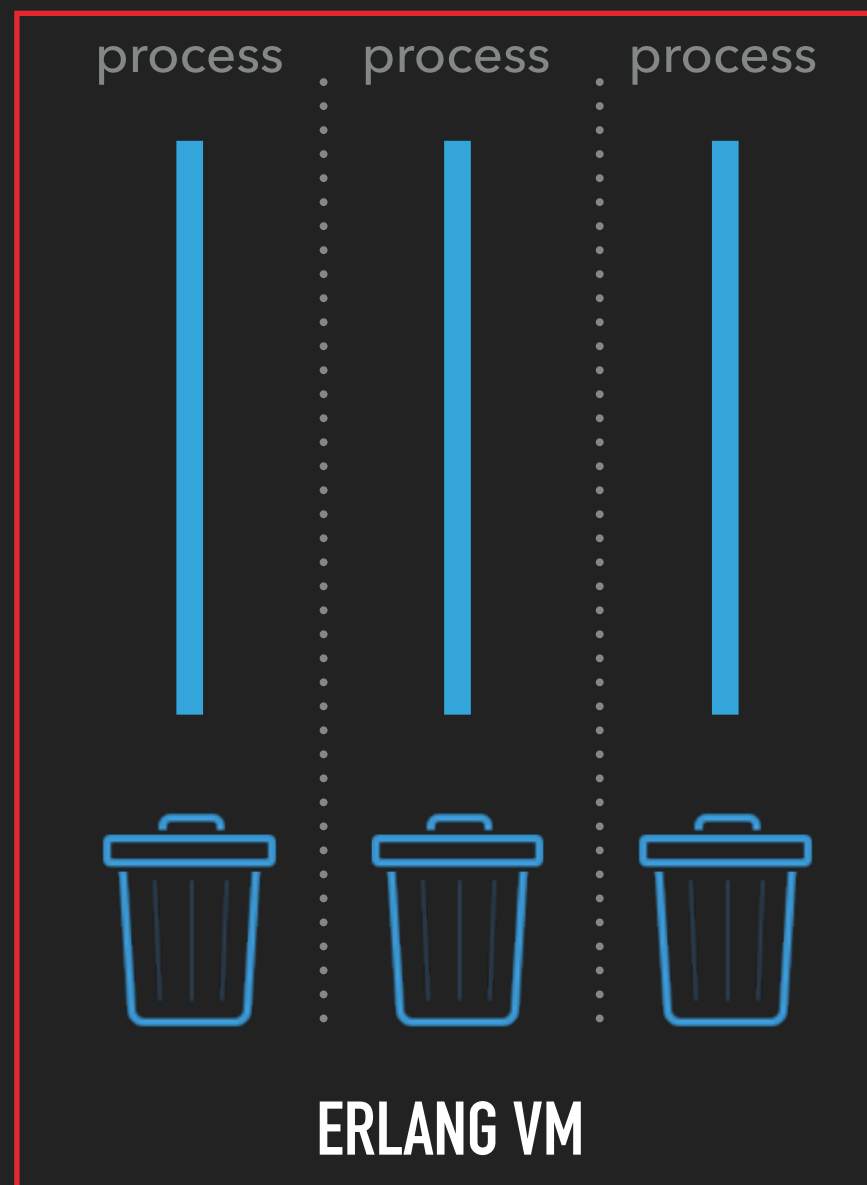
# ACTOR MODEL LIFE CYCLE





# ACTOR MODEL

## GARBAGE COLLECTOR



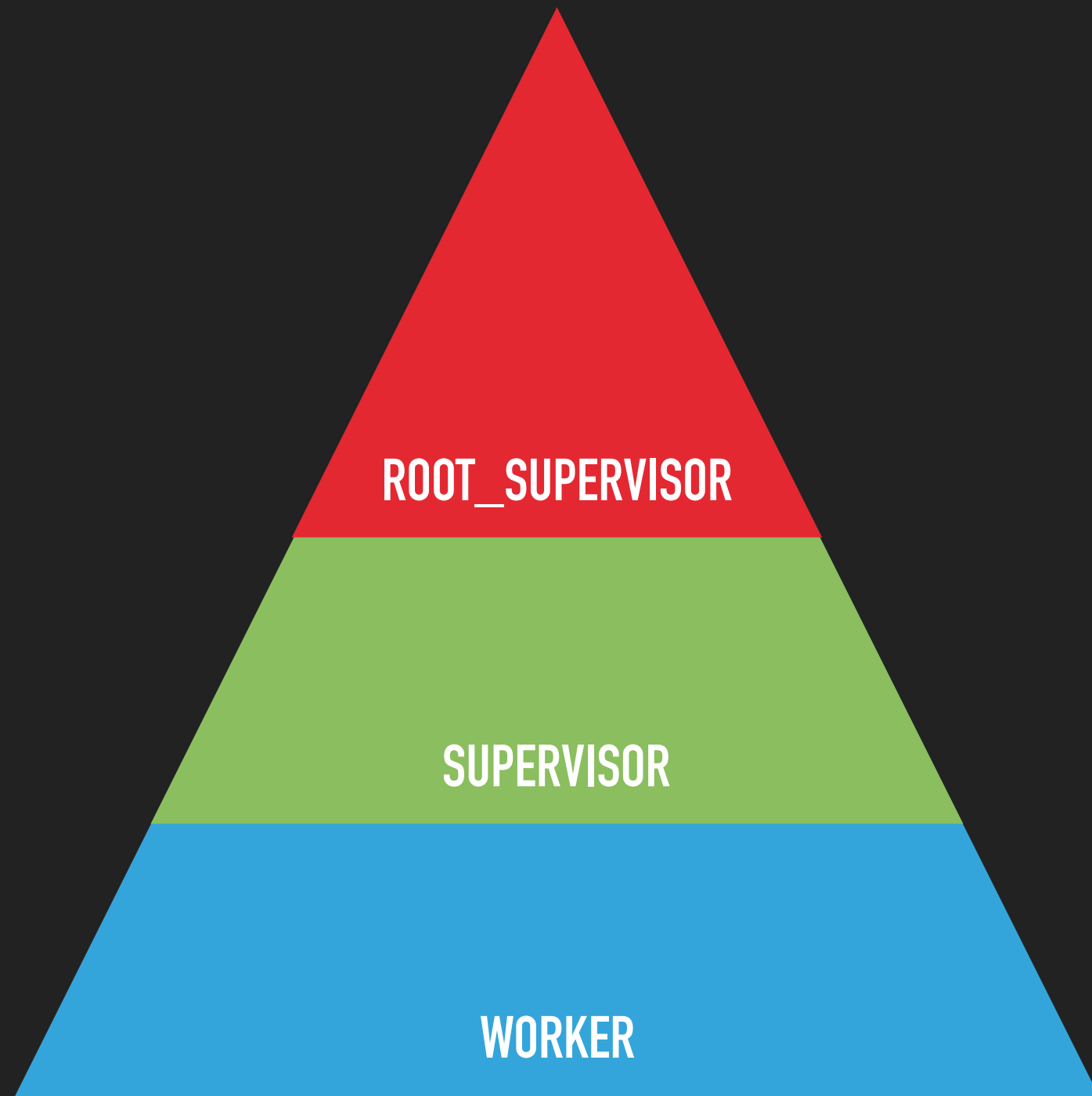
# ACTOR MODEL



# ACTOR MODEL



# ACTOR MODEL



# OTP-BEHAVIOR

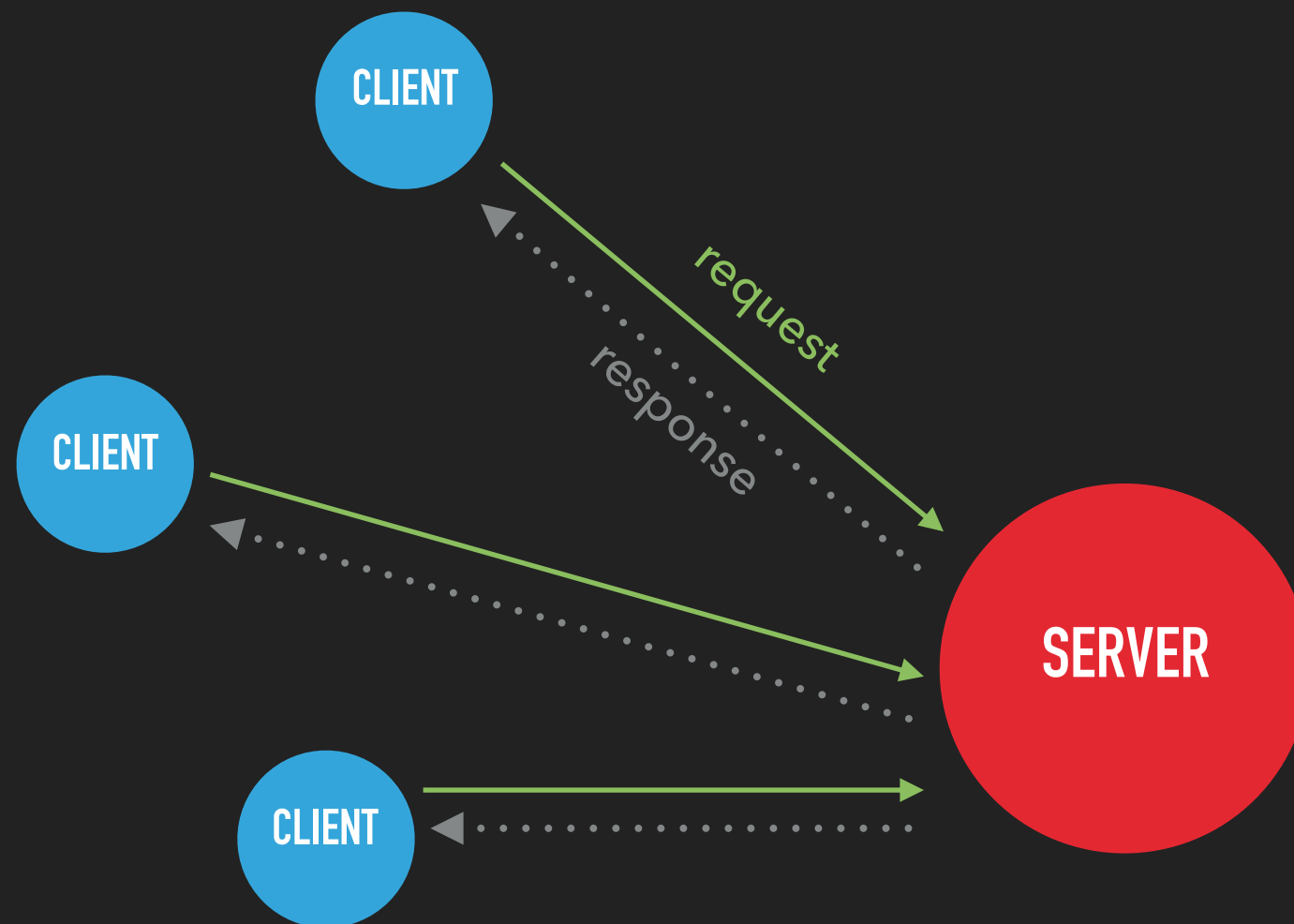
- ▶ GenServer
- ▶ Supervisor
- ▶ GenEvent
- ▶ GenFsm
- ▶ GenStateM

# OTP-BEHAVIOR

- ▶ GenServer
- ▶ Supervisor
- ▶ GenEvent
- ▶ GenStateM

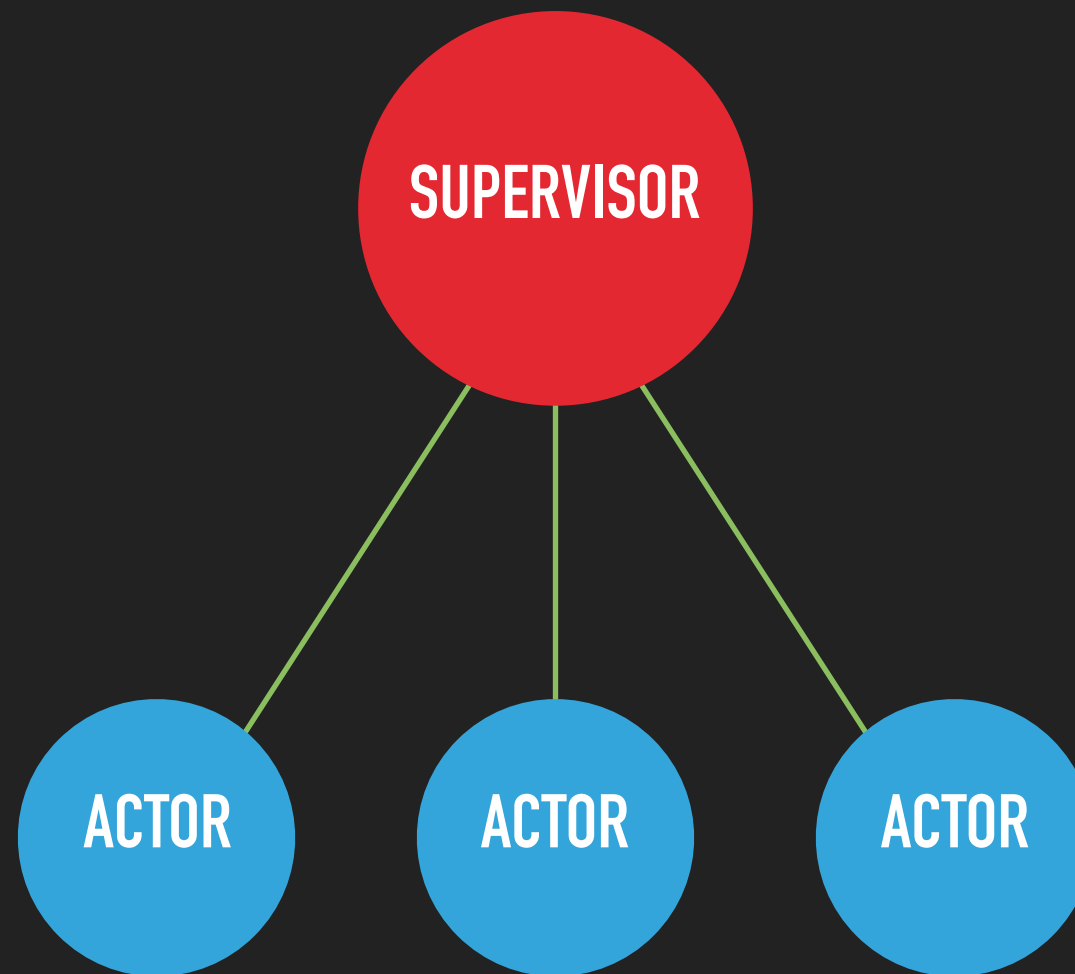
# GENSERVER

## GENERIC SERVER



[http://erlang.org/doc/design\\_principles/gen\\_server\\_concepts.html](http://erlang.org/doc/design_principles/gen_server_concepts.html)

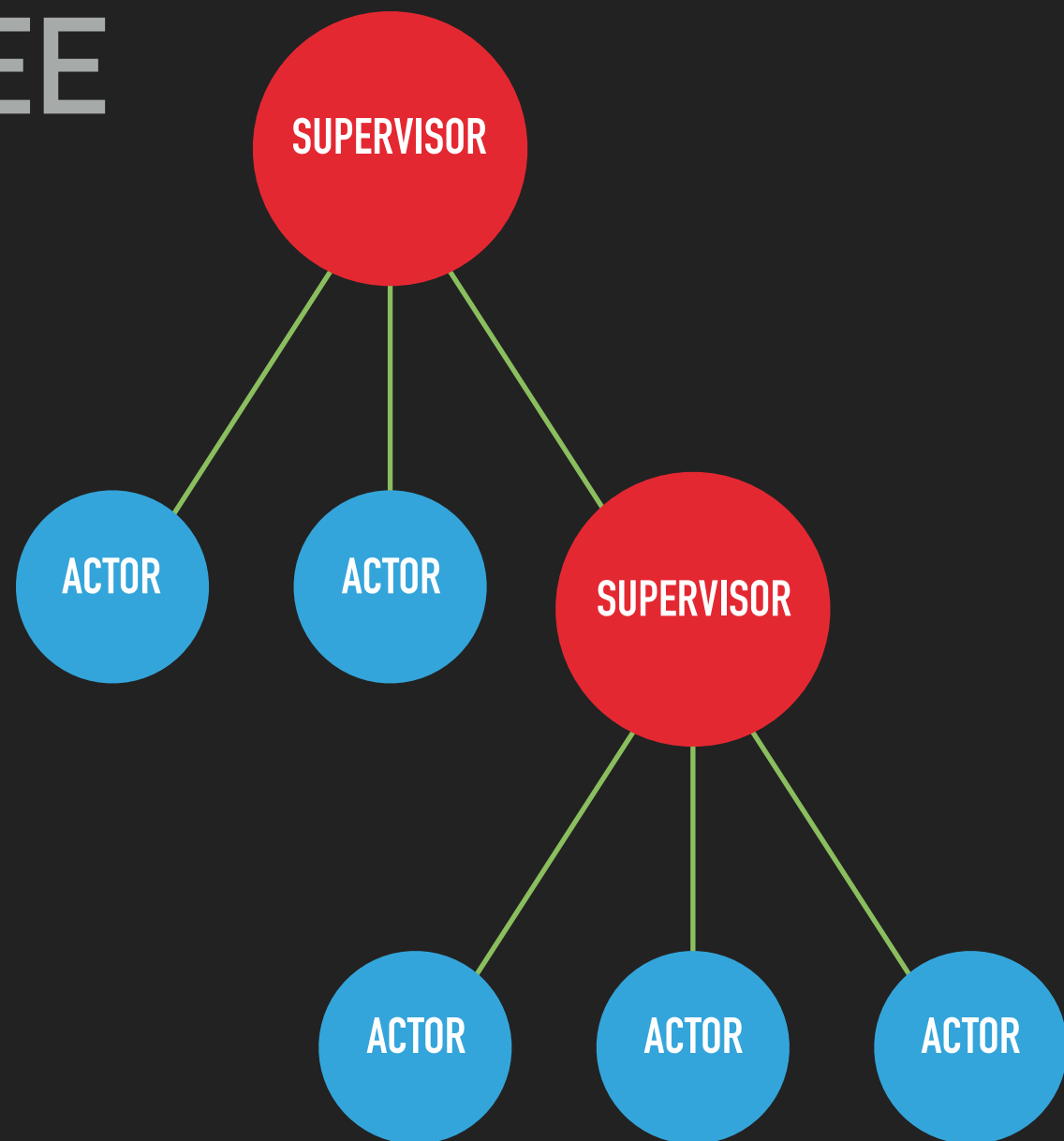
# SUPERVISOR





# SUPERVISOR

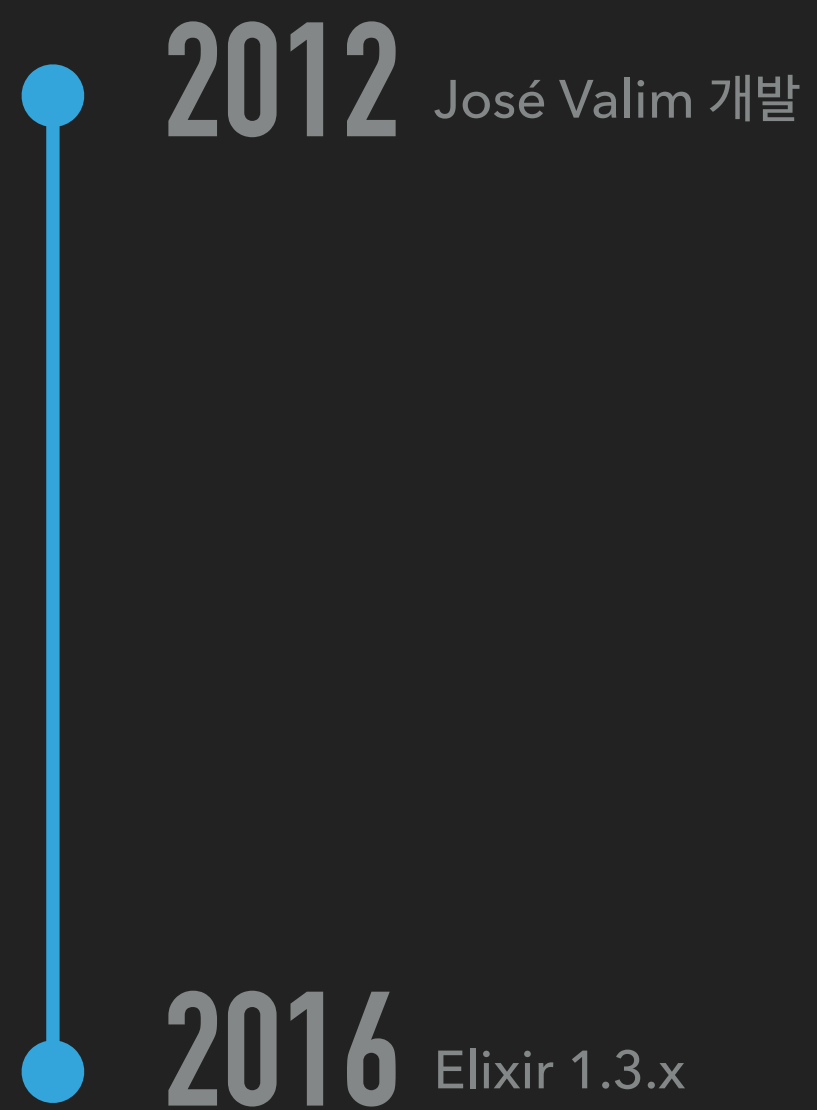
## SUPERVISION TREE



“Erlang was designed for  
writing concurrent programs  
that run forever.”

A History of Erlang – Joe Armstrong

**ELIXIR**



# ELIXIR

- ▶ Functional language
- ▶ Immutable variables
- ▶ Strong, dynamic typing
- ▶ Actor model

ELIXIR

ERLANG | ELIXIR

OTP

BEAM

# ELIXIR

- ▶ Support tools
  - ▶ ExUnit - Unit test
  - ▶ Mix - Build tool
  - ▶ Standard library
  - ▶ Metaprogramming
- ▶ Ecosystem
  - ▶ Hex - Package management
  - ▶ Phoenix - Web framework

# DISTRIBUTED SYSTEMS



“A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable”

Leslie Lamport

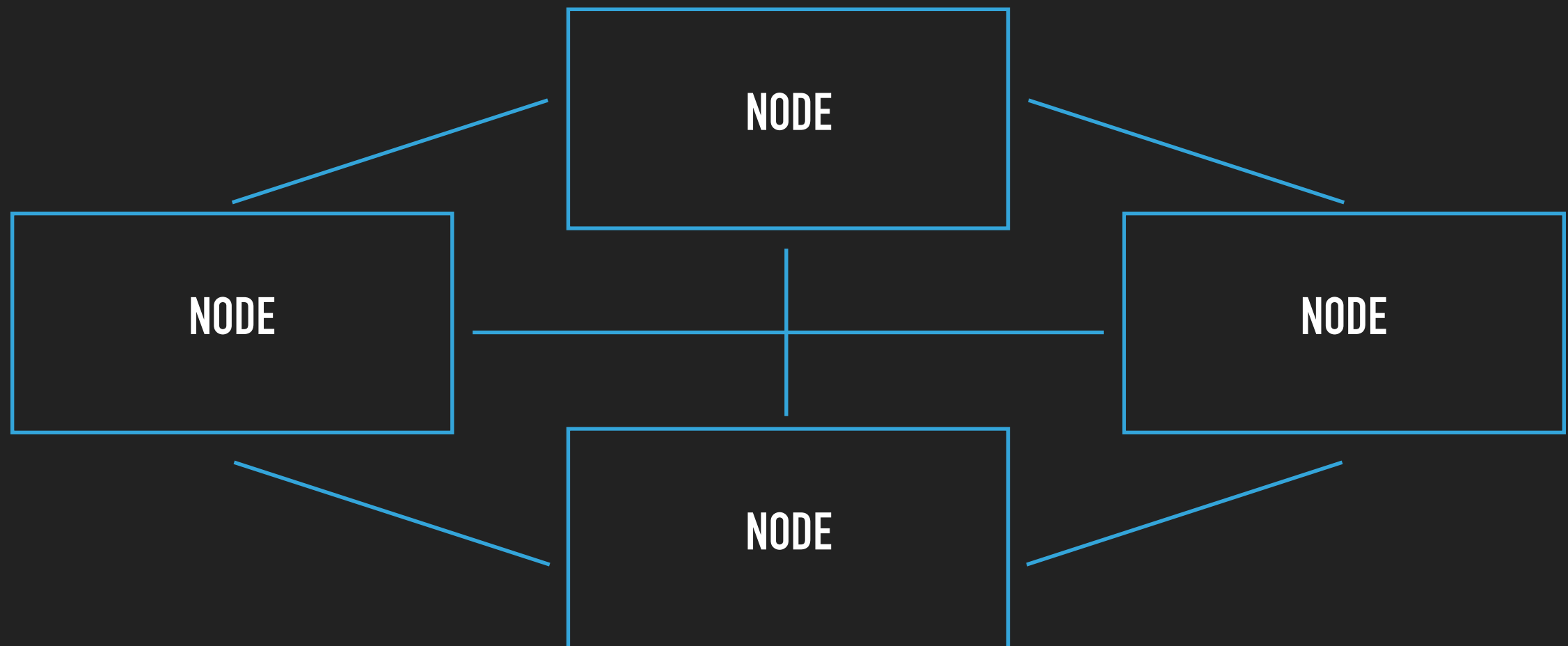
# ERLANG CLUSTER

A distributed Erlang system consists of a number of Erlang runtime systems communicating with each other. Each such runtime system is called a **node**.

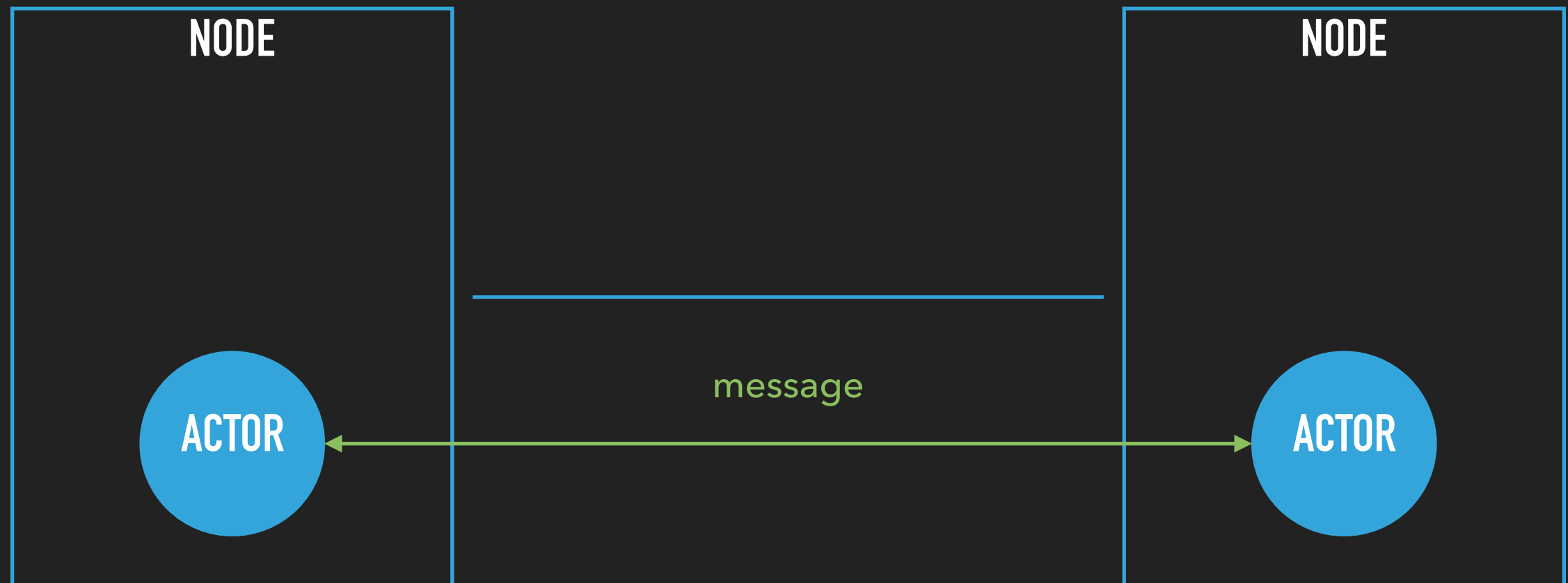


**NODE**

# ERLANG CLUSTER



# ERLANG CLUSTER



# WHY DISTRIBUTED SYSTEMS?

# WHY DISTRIBUTED SYSTEMS?

- ▶ Enhanced Performance
- ▶ Higher Availability

# PERFORMANCE

- ▶ Latency
- ▶ Throughput
- ▶ Computing power

# PERFORMANCE

CLIENT

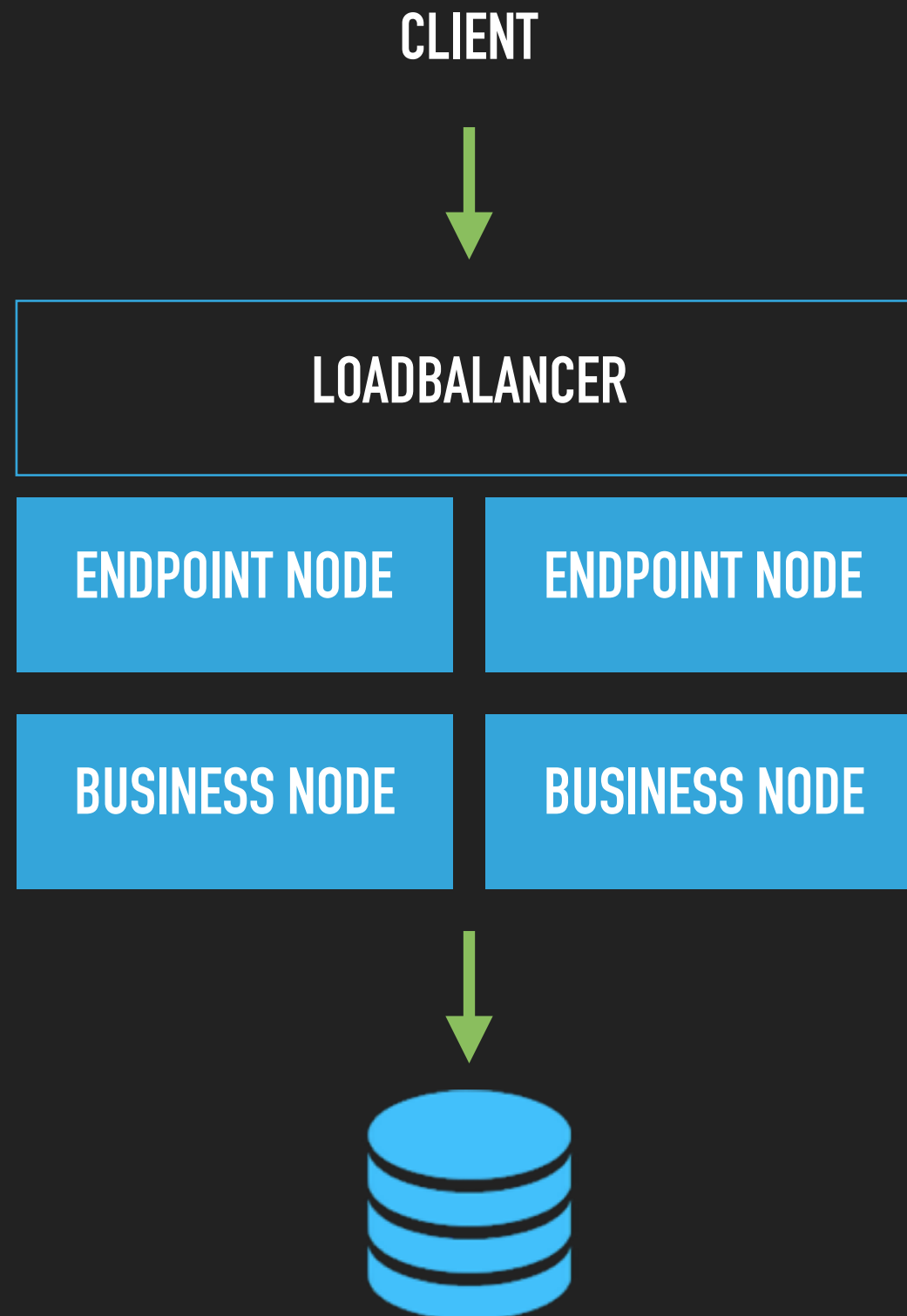


NODE

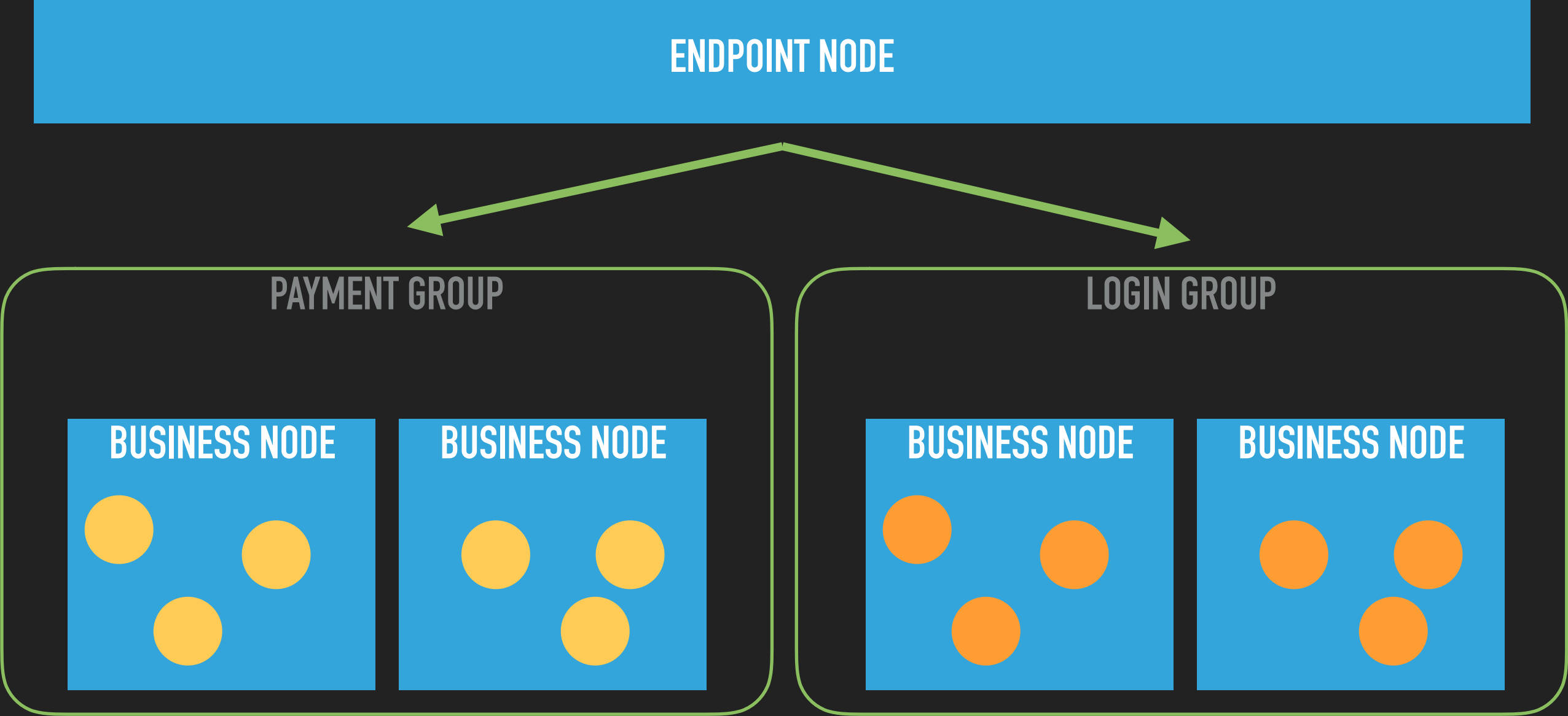




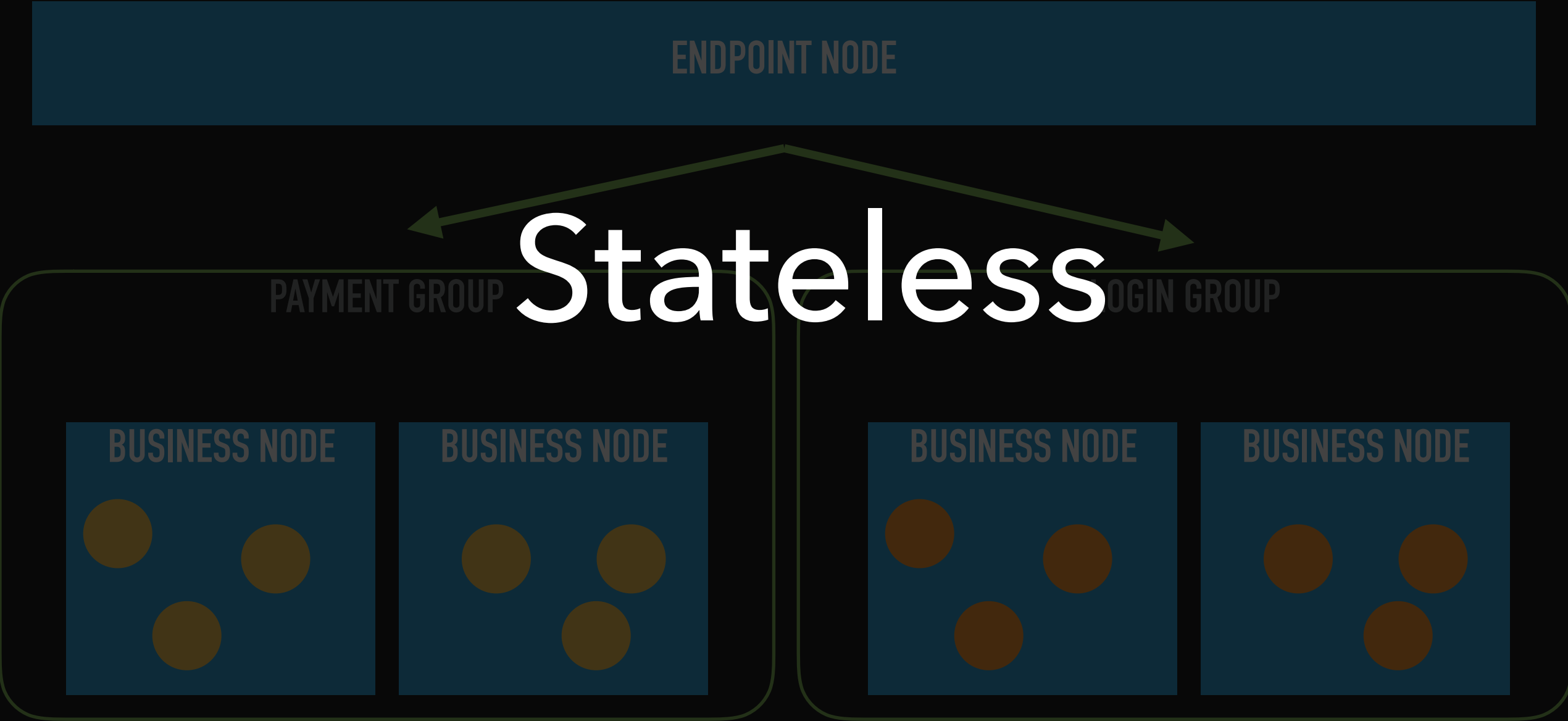
# PERFORMANCE



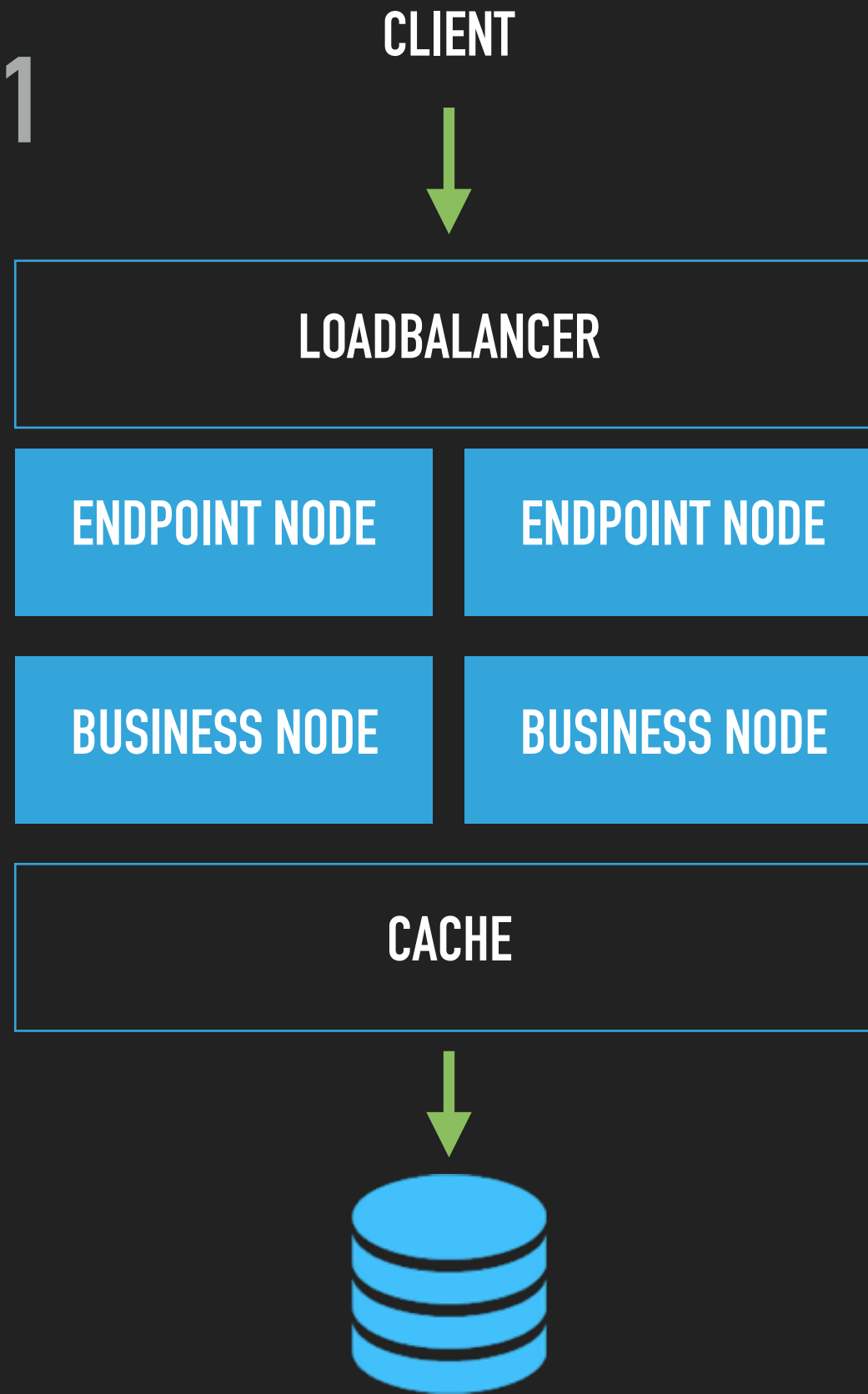
# PERFORMANCE



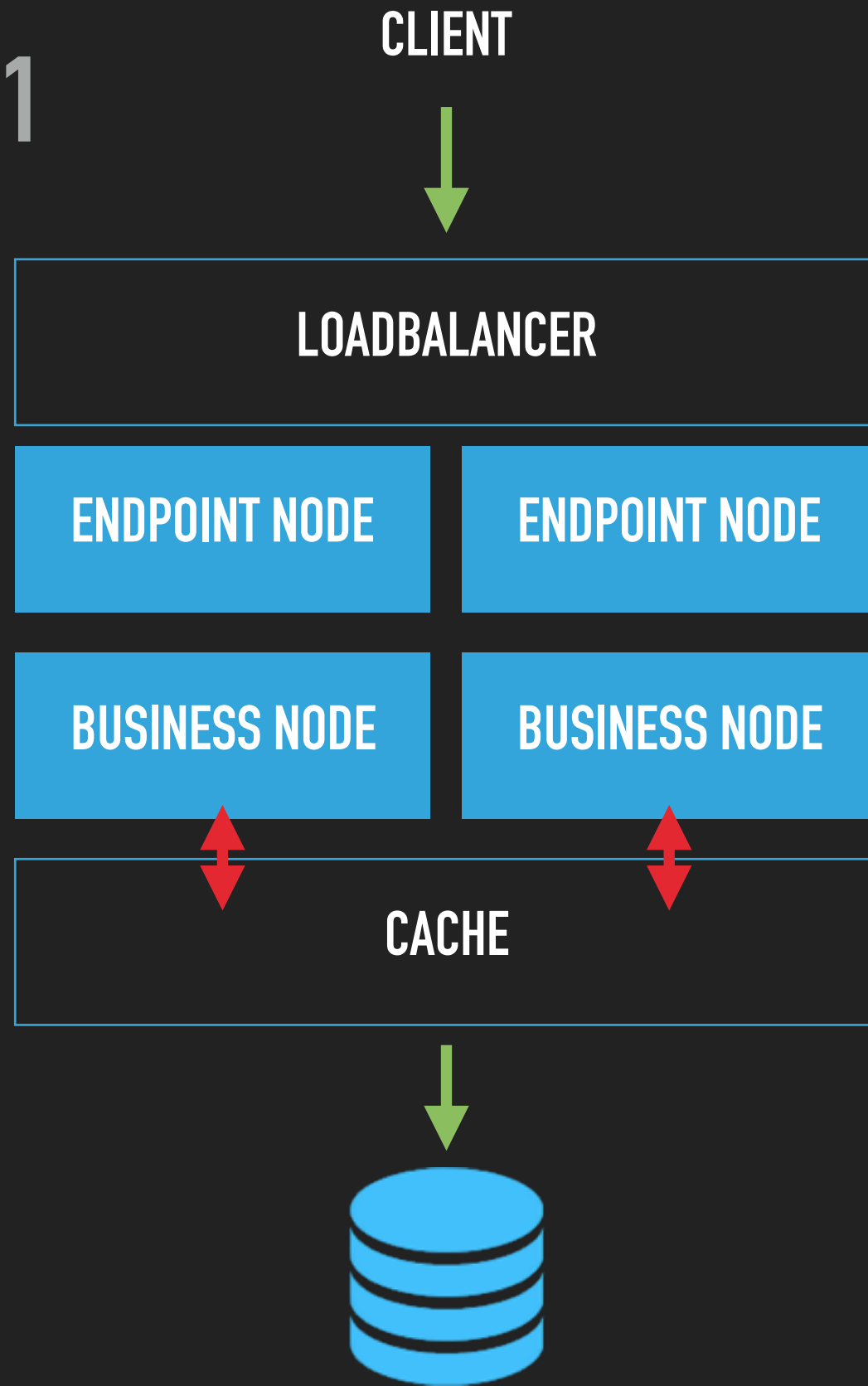
PERFORMANCE



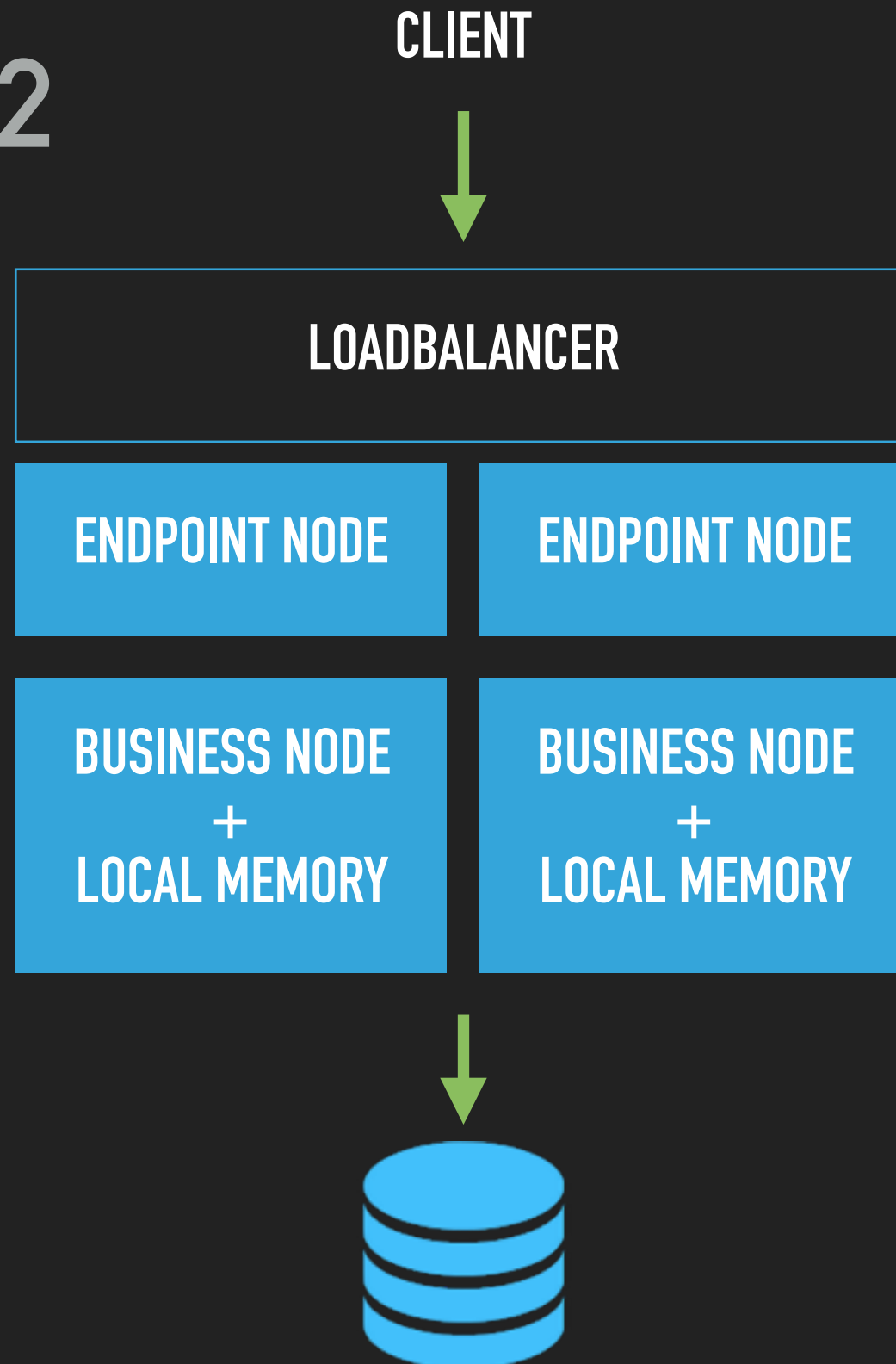
# PERFORMANCE STATEFUL #1



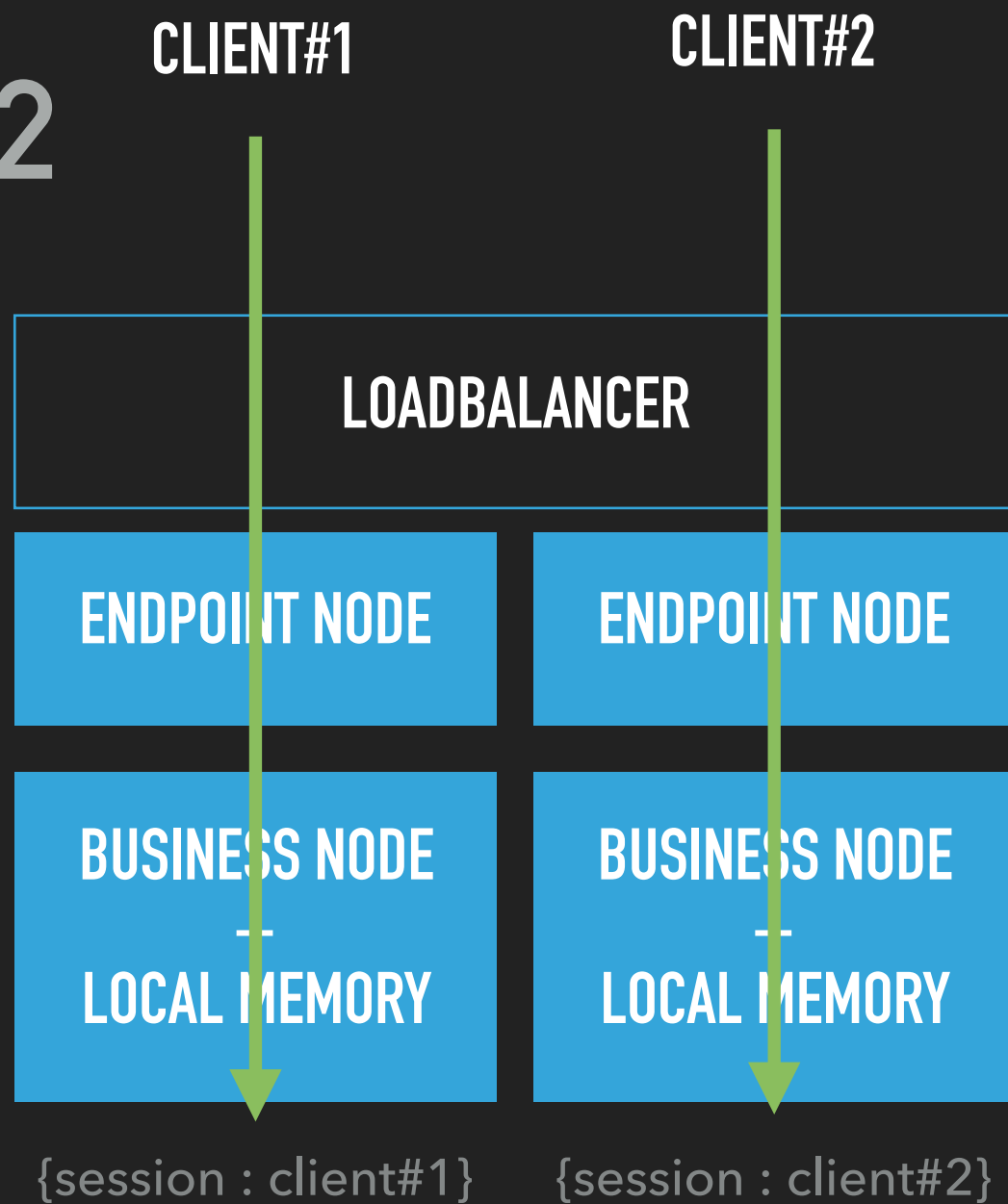
# PERFORMANCE STATEFUL #1



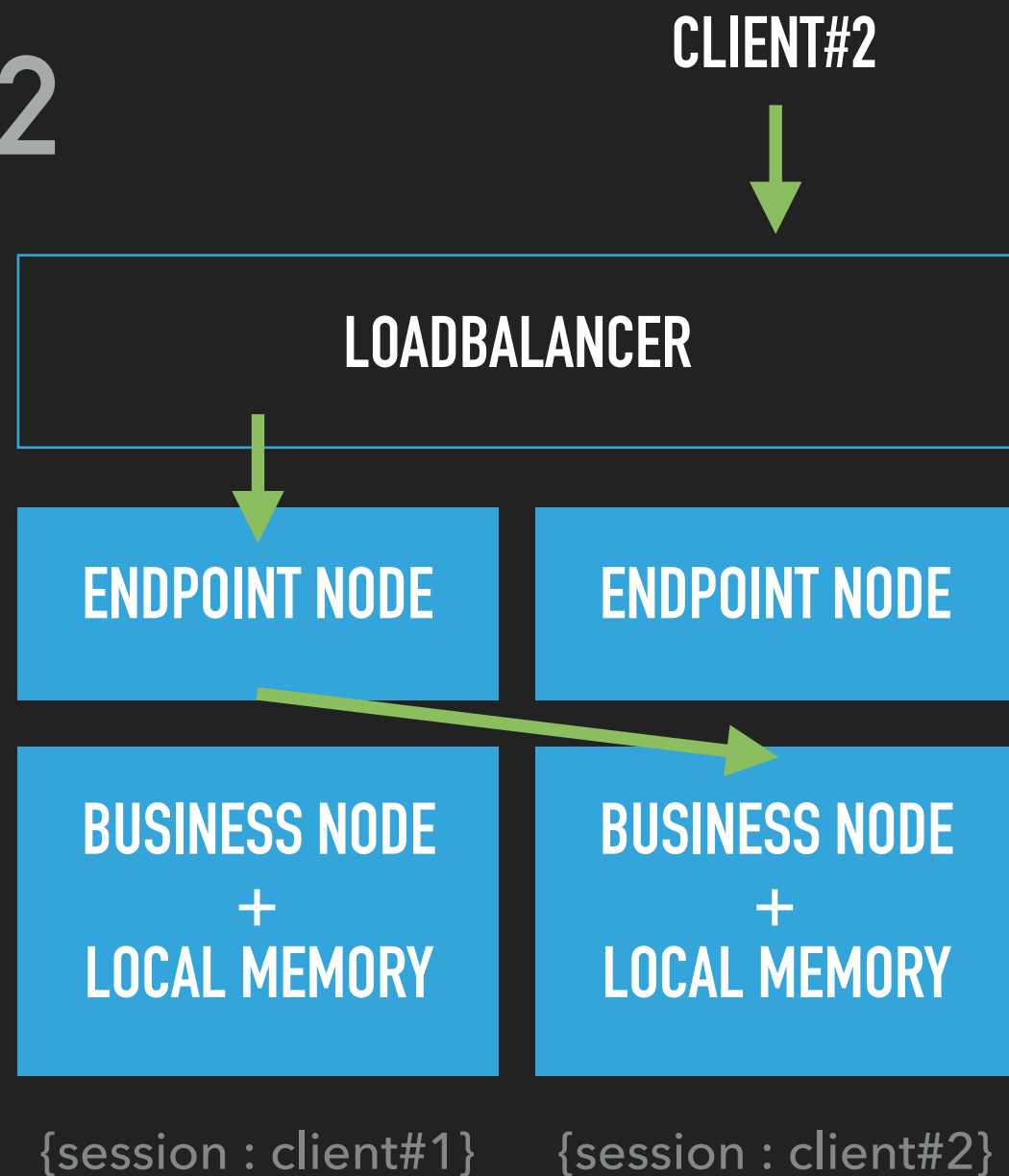
# PERFORMANCE STATEFUL #2



# PERFORMANCE STATEFUL #2



# PERFORMANCE STATEFUL #2

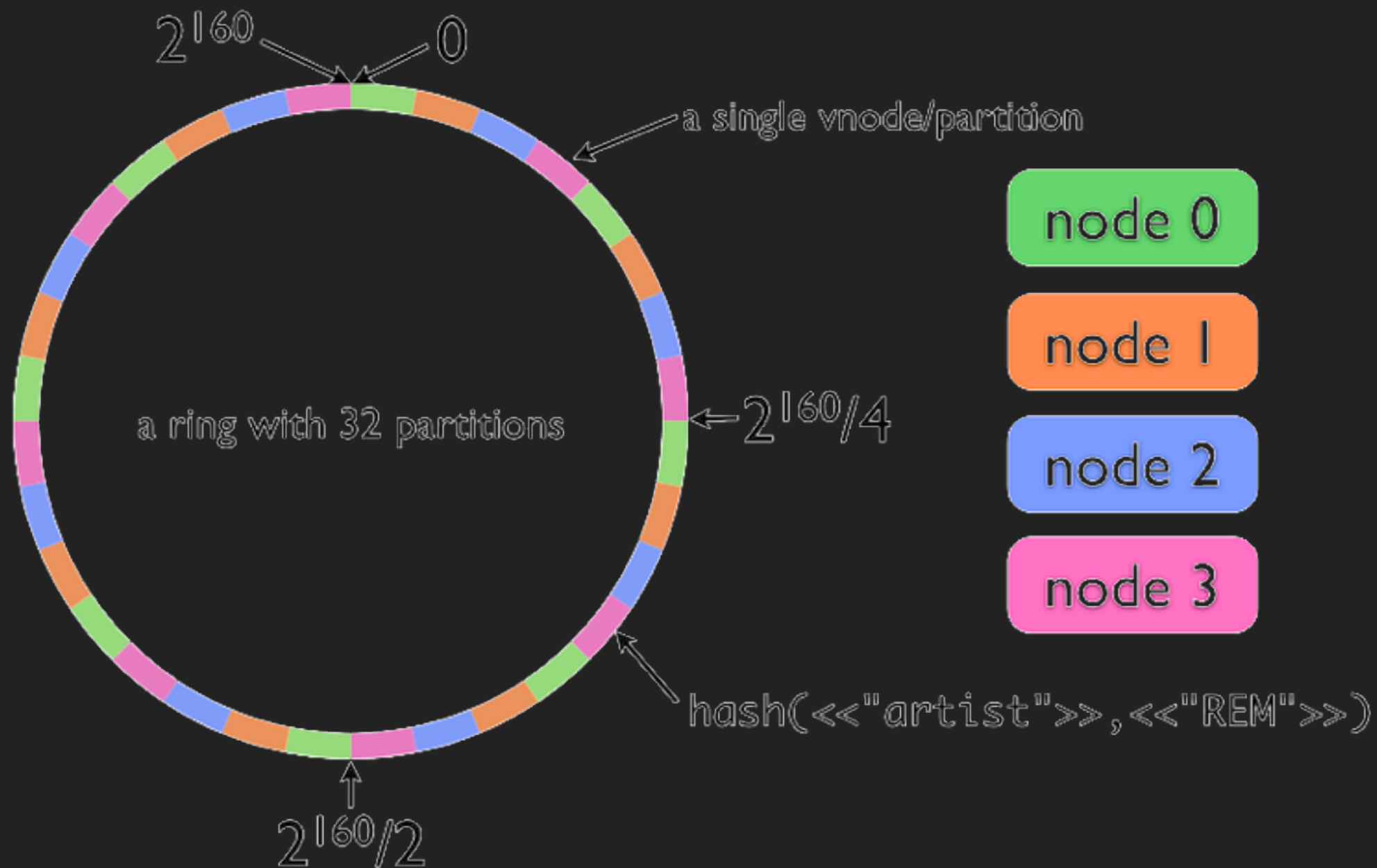




# PERFORMANCE STATEFUL #2

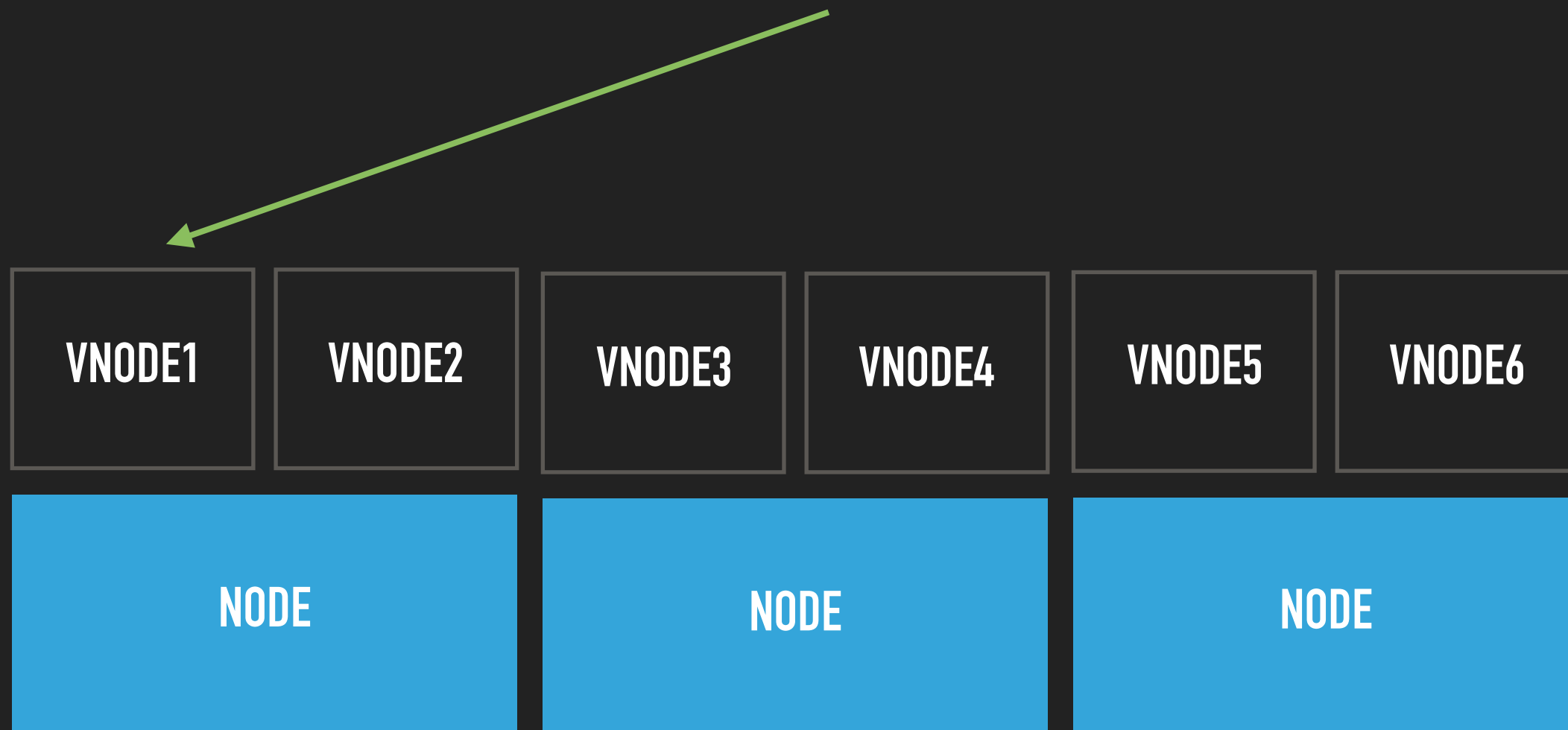
RIAK\_CORE

# PERFORMANCE STATEFUL #2

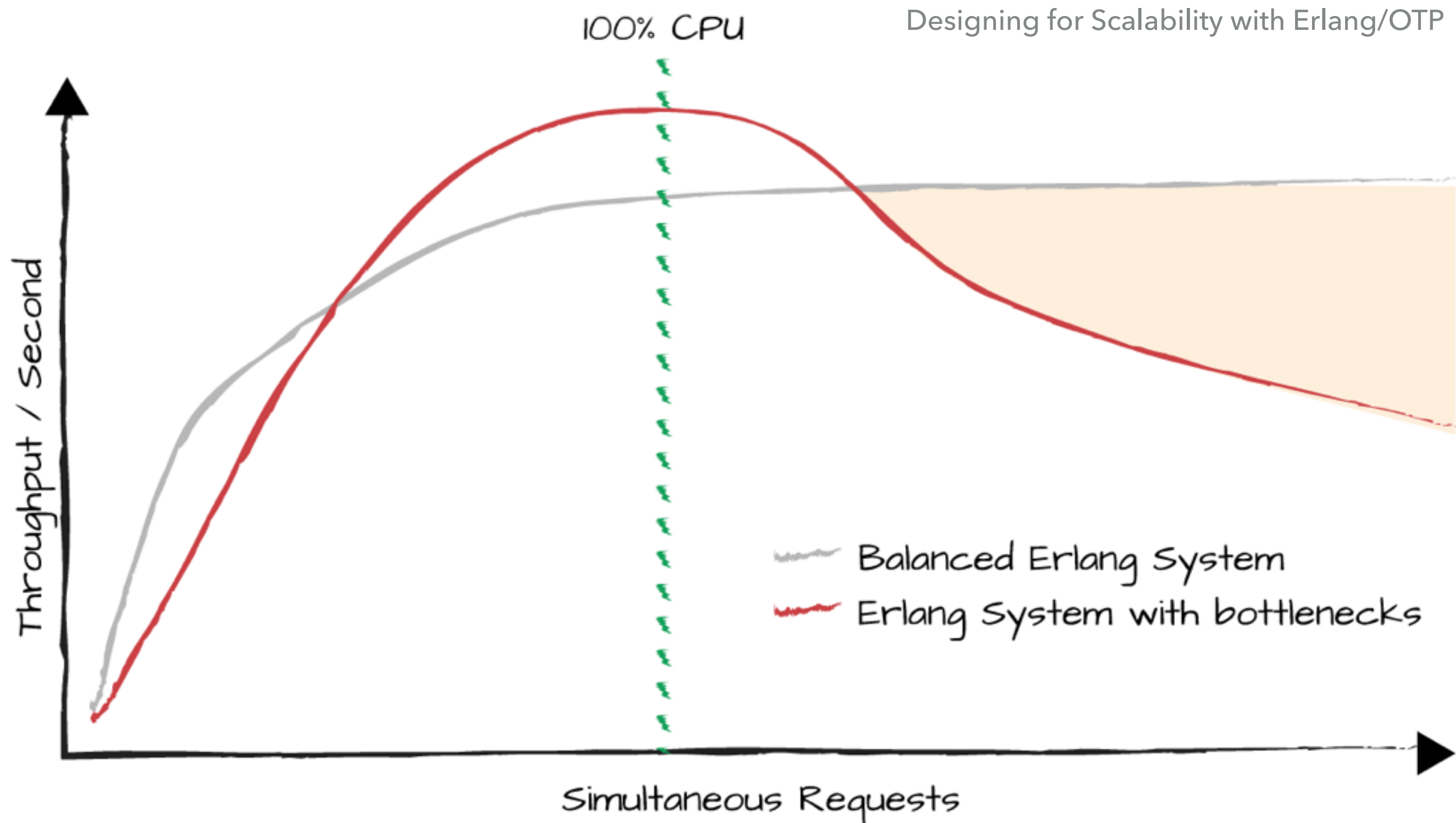


# PERFORMANCE STATEFUL #2

$\text{hash}(\text{session\_id}) = 1$



# PERFORMANCE

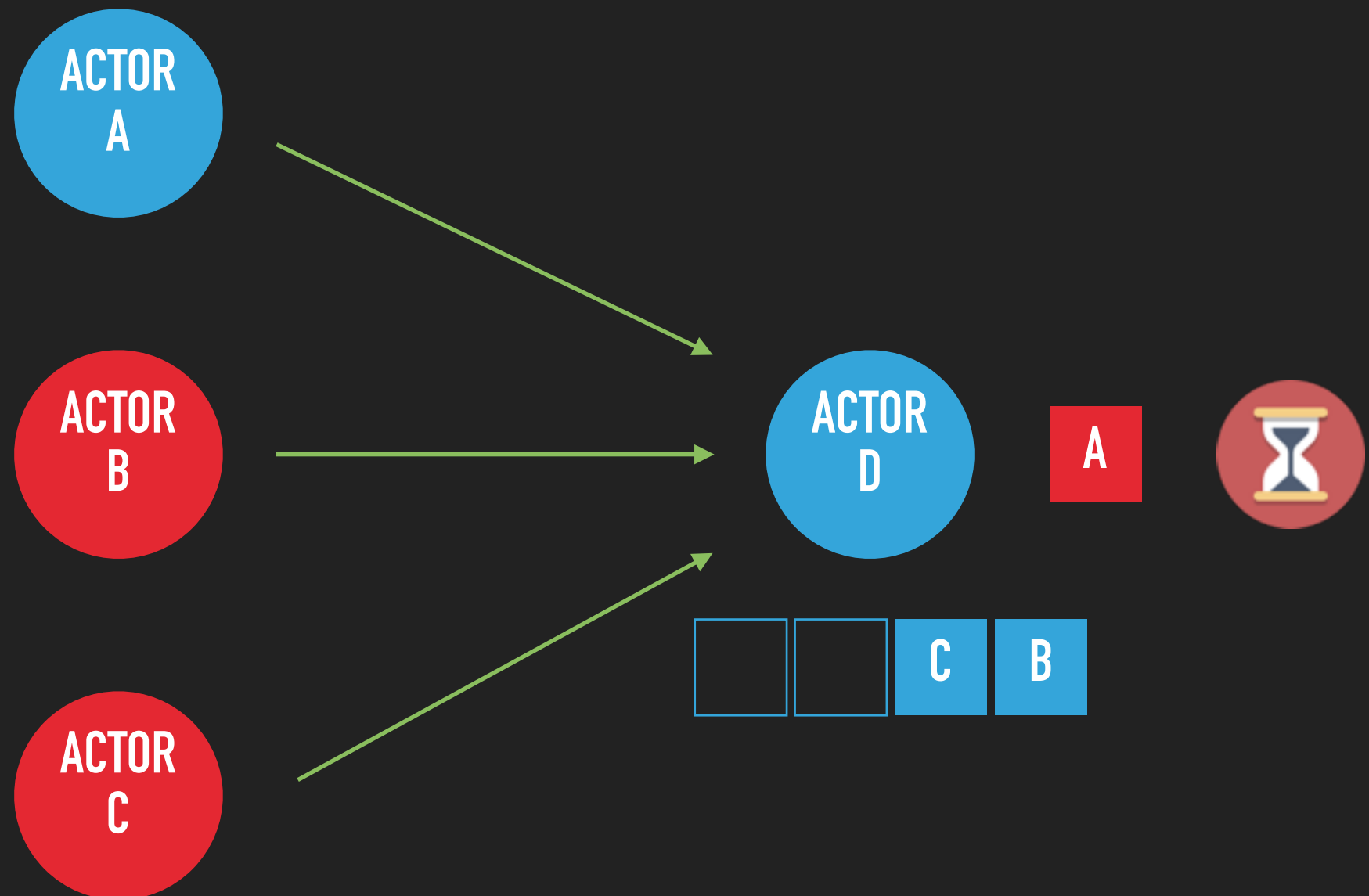


**PERFORMANCE**

**SYNCHRONOUS CALLS**

# PERFORMANCE

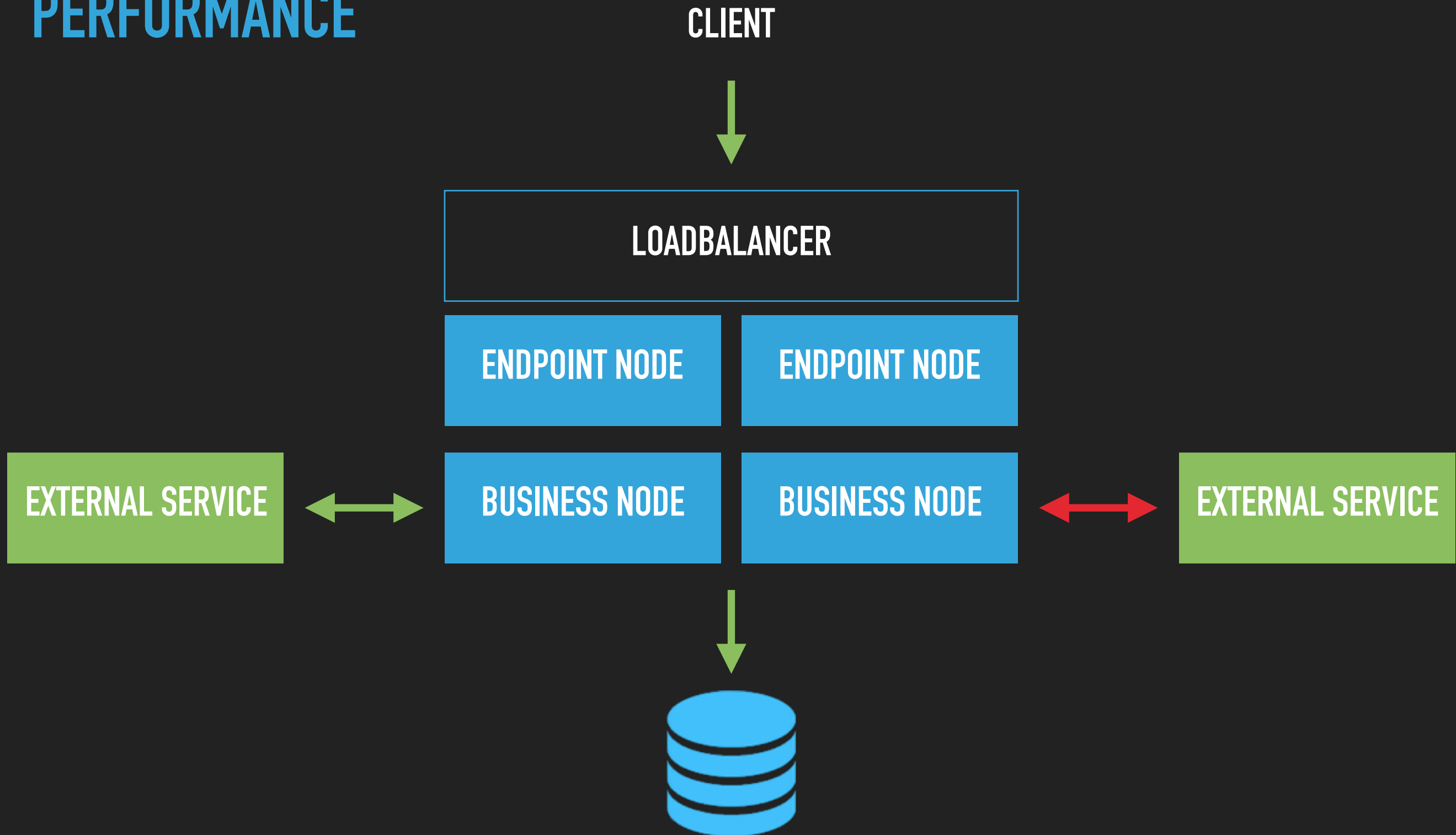
## SYNCHRONOUS CALLS



**PERFORMANCE**

**EXTERNAL SERVICE**

# PERFORMANCE

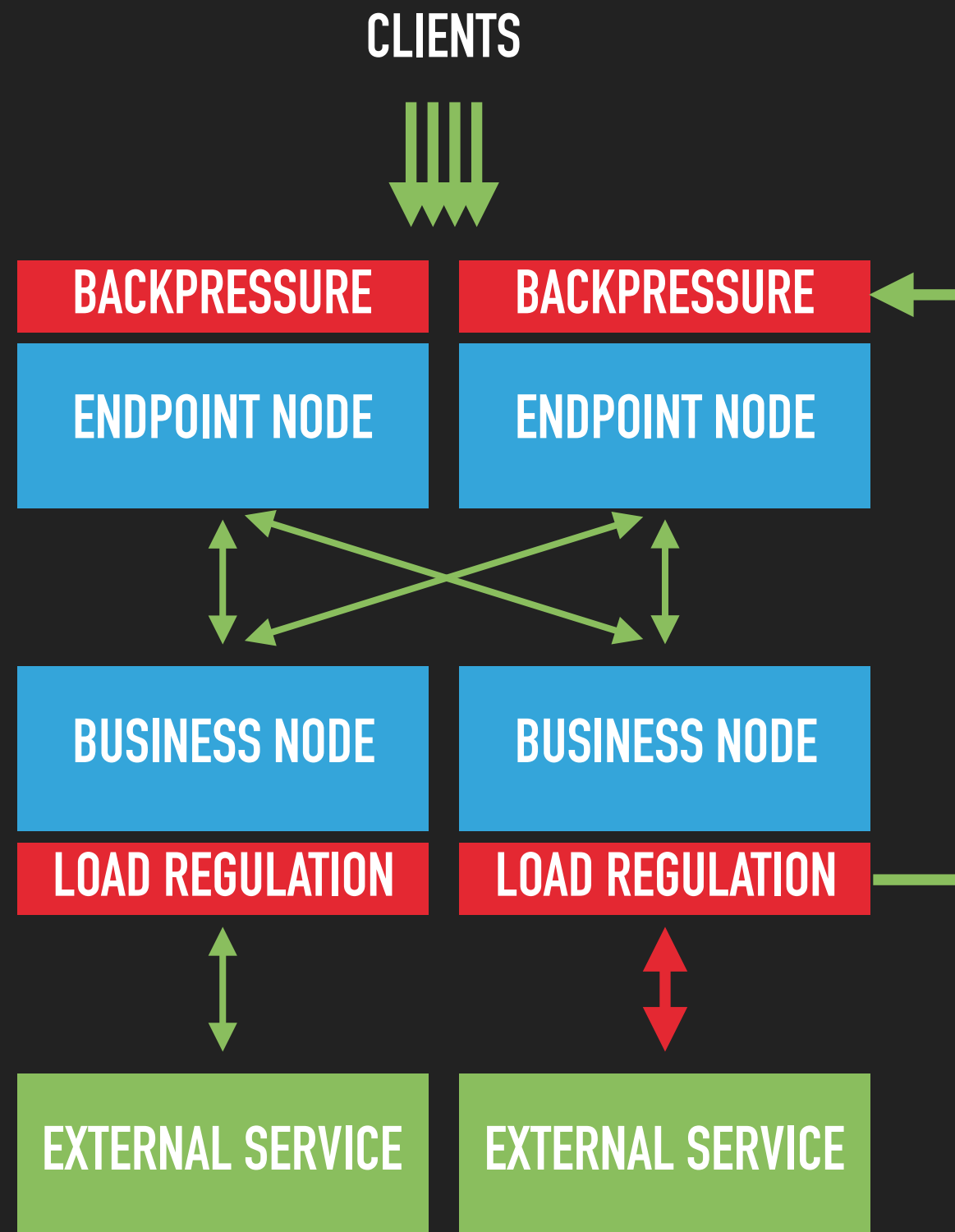




**PERFORMANCE**

**LOAD REGULATION AND BACKPRESSURE**

# PERFORMANCE

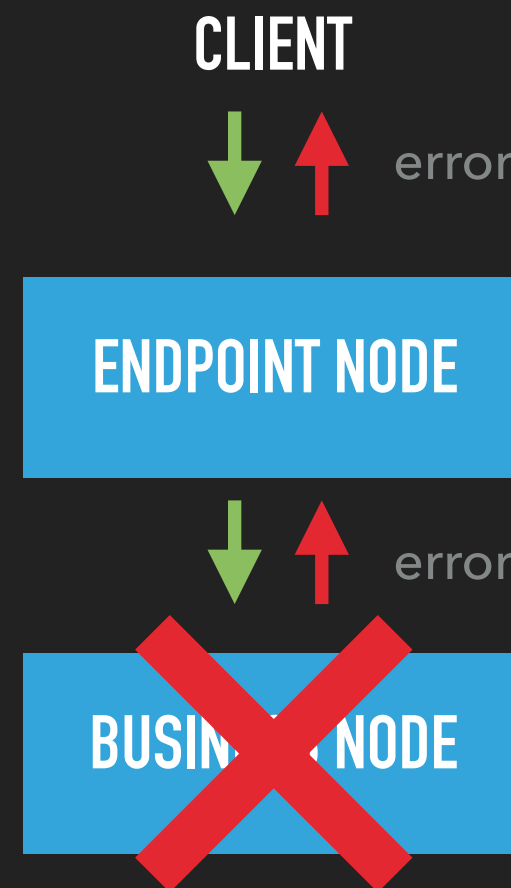
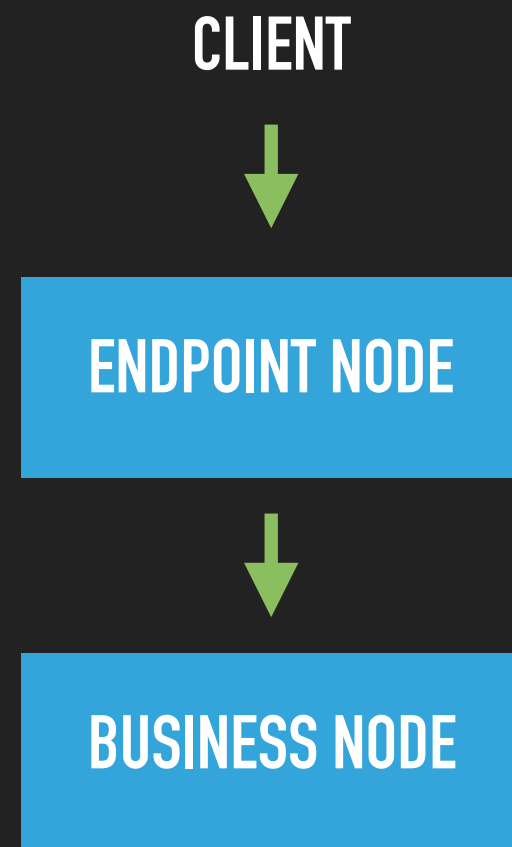


# AVAILABILITY

- ▶ Fault Tolerance
- ▶ Resilience
- ▶ Reliability

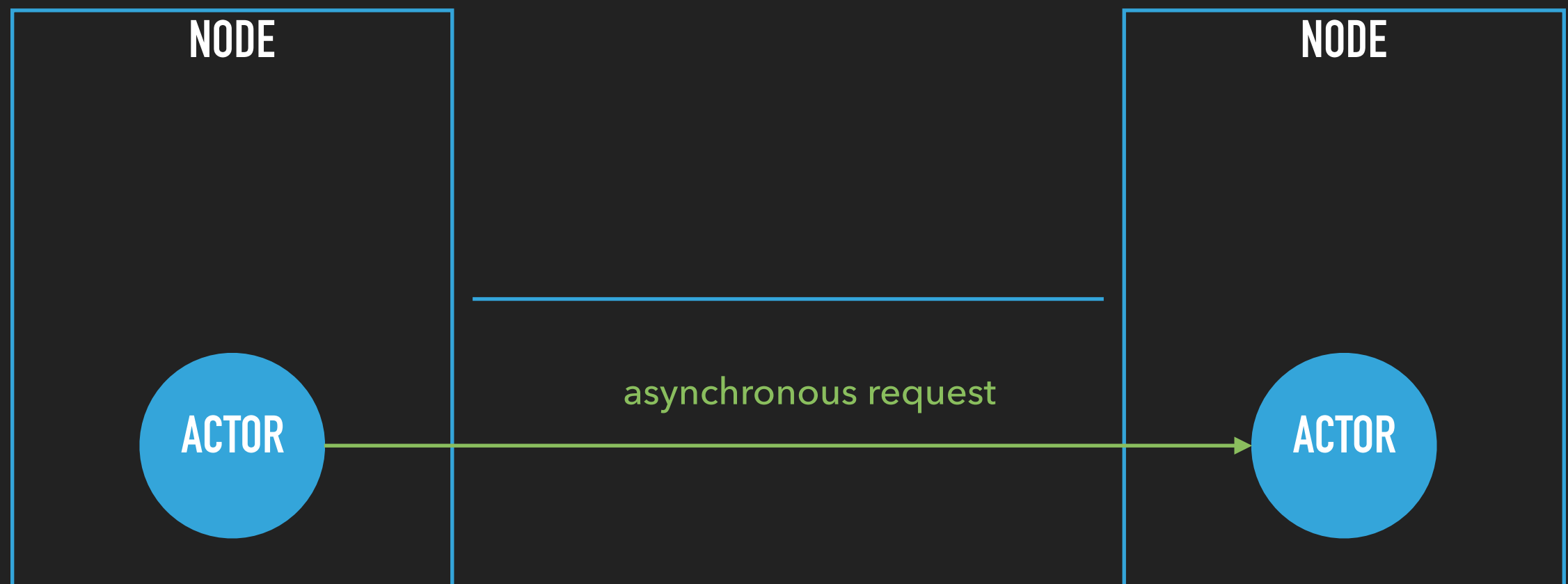
# AVAILABILITY

## FAULT TOLERANCE



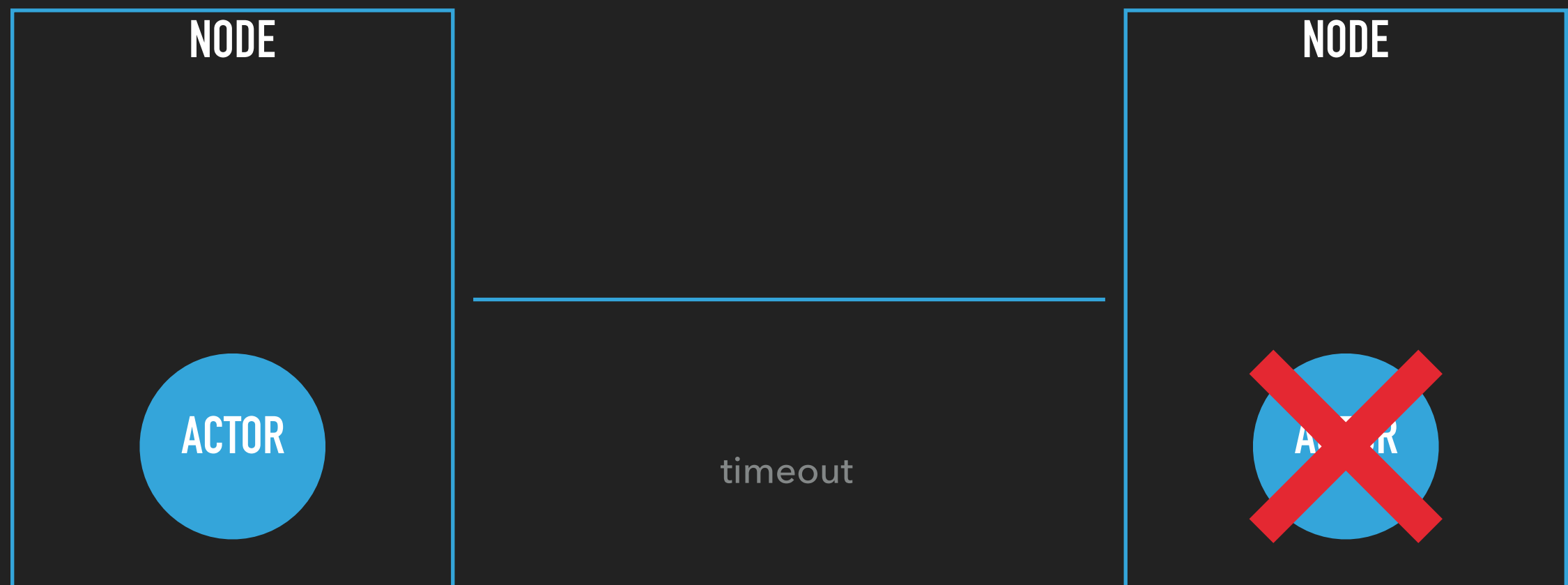
AVAILABILITY

FAULT TOLERANCE



AVAILABILITY

FAULT TOLERANCE



AVAILABILITY

RESILIENCE - NODE

ENDPOINT NODE

PAYMENT GROUP

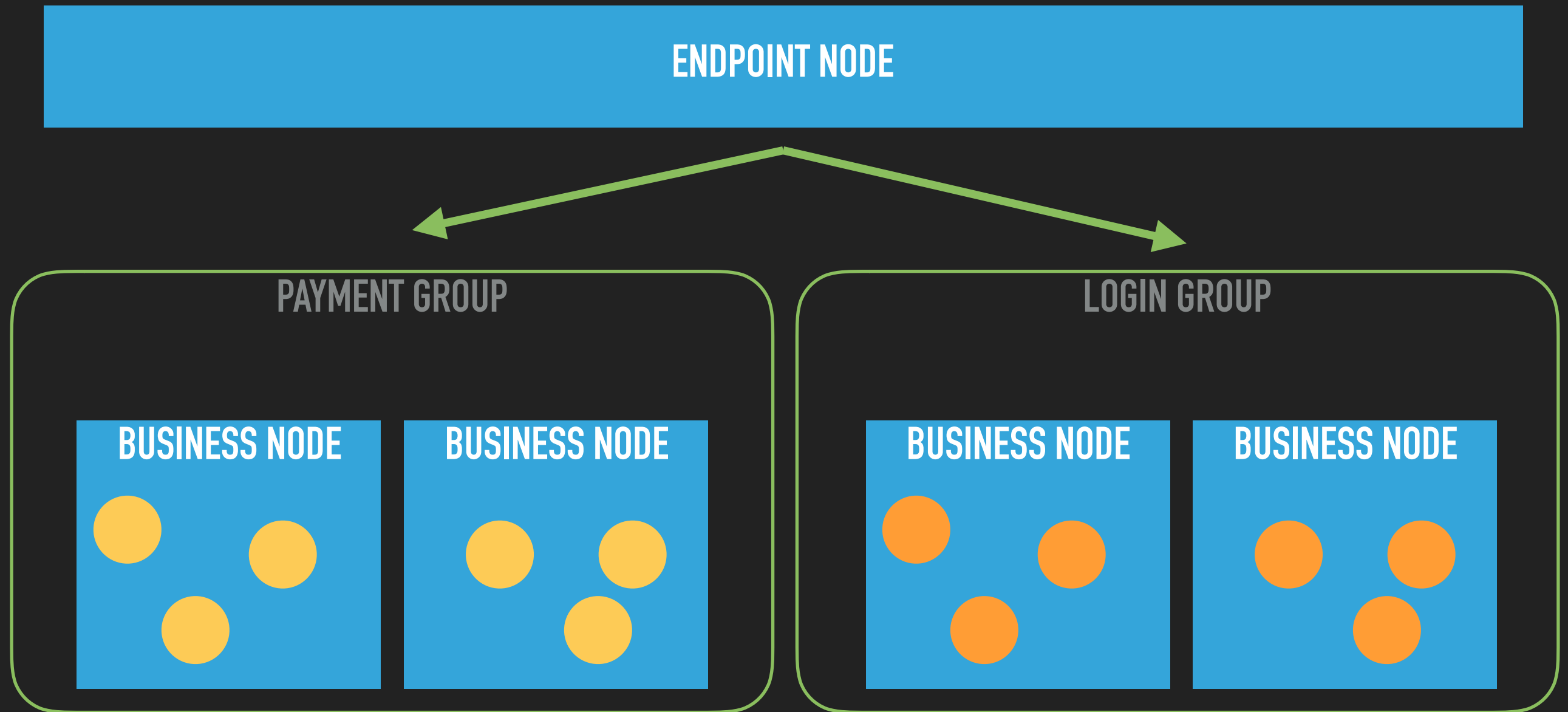
BUSINESS NODE

BUSINESS NODE

LOGIN GROUP

BUSINESS NODE

BUSINESS NODE



AVAILABILITY

# RESILIENCE - NODE

ENDPOINT NODE

PAYMENT GROUP

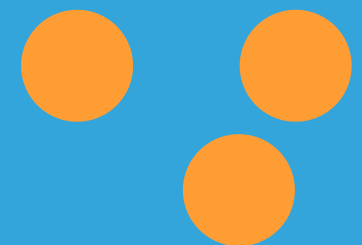
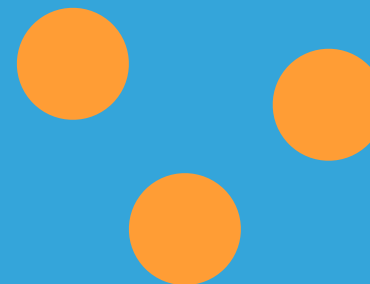
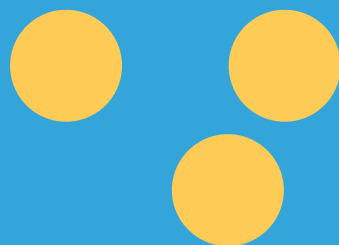
LOGIN GROUP

BUSINESS NODE

BUSINESS NODE

BUSINESS NODE

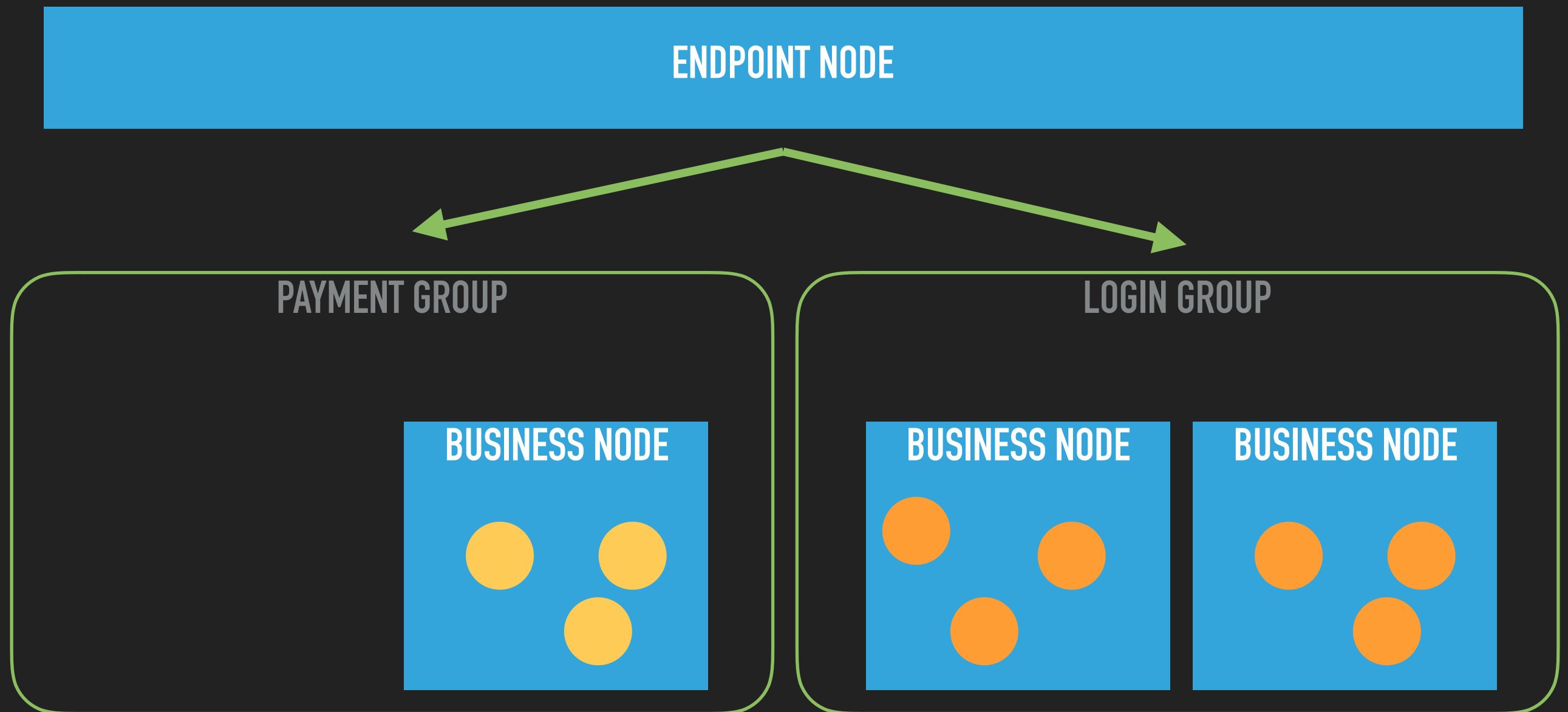
BUSINESS NODE





AVAILABILITY

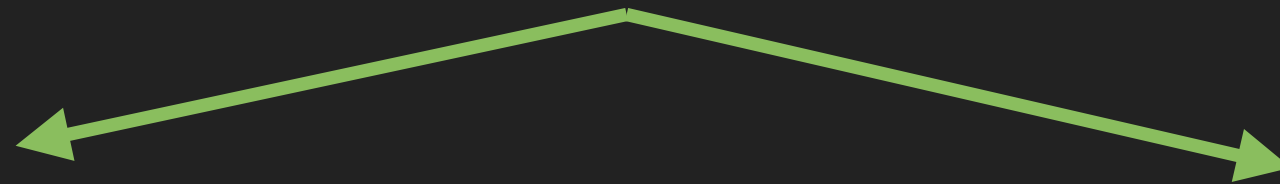
# RESILIENCE - NODE



AVAILABILITY

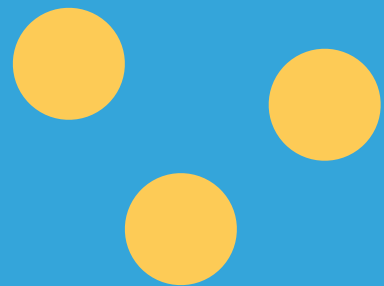
# RESILIENCE - NODE

ENDPOINT NODE

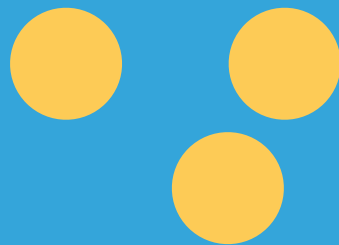


PAYMENT GROUP

BUSINESS NODE

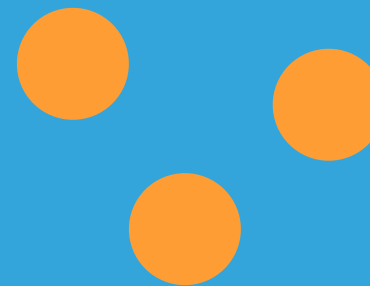


BUSINESS NODE

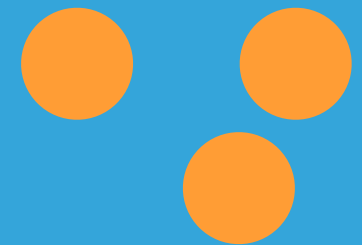


LOGIN GROUP

BUSINESS NODE

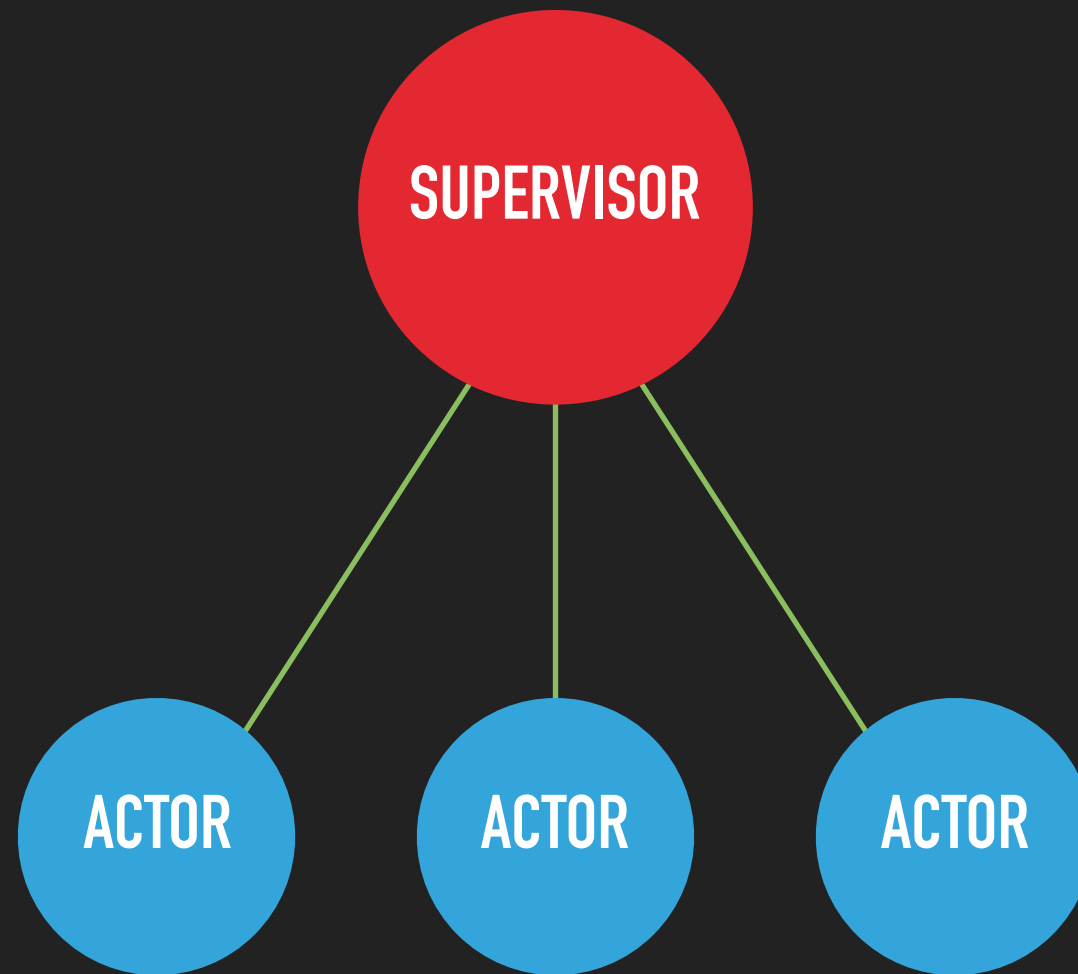


BUSINESS NODE



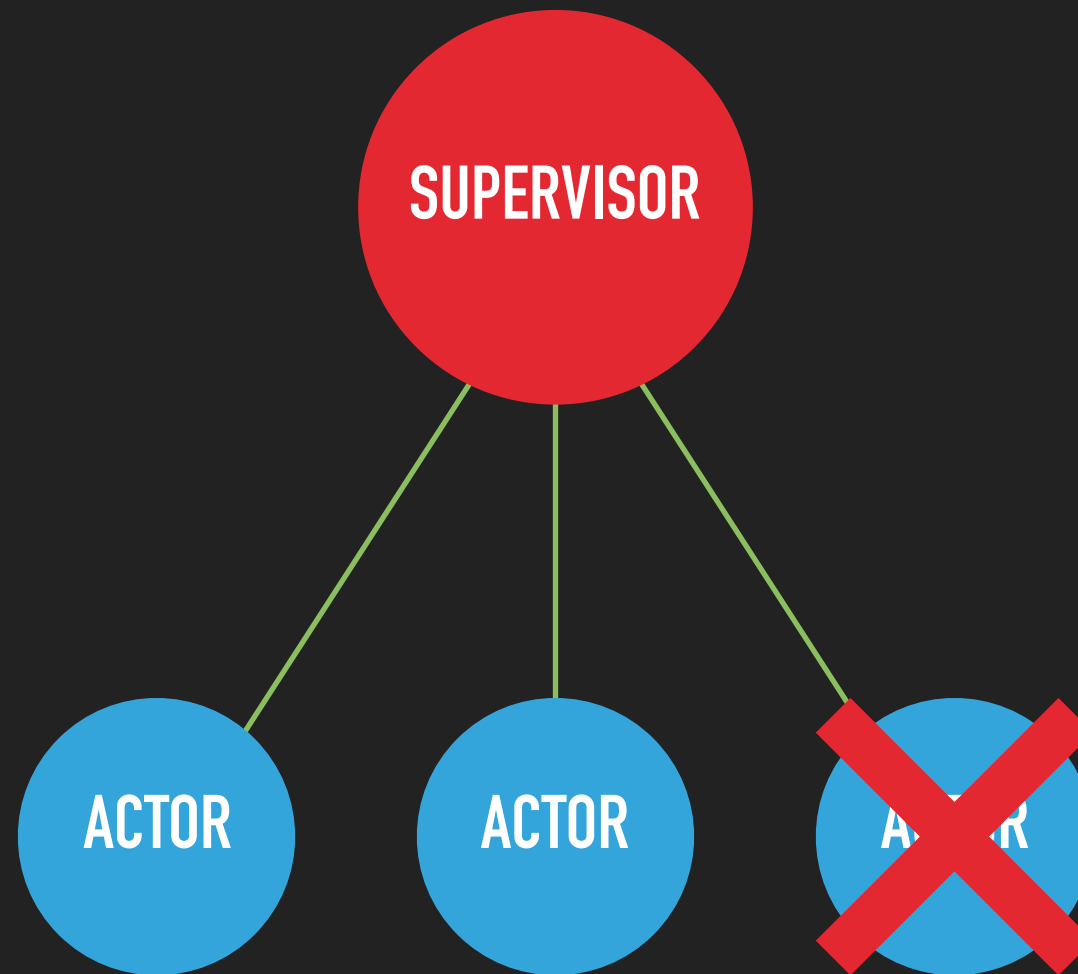
AVAILABILITY

RESILIENCE - ACTOR



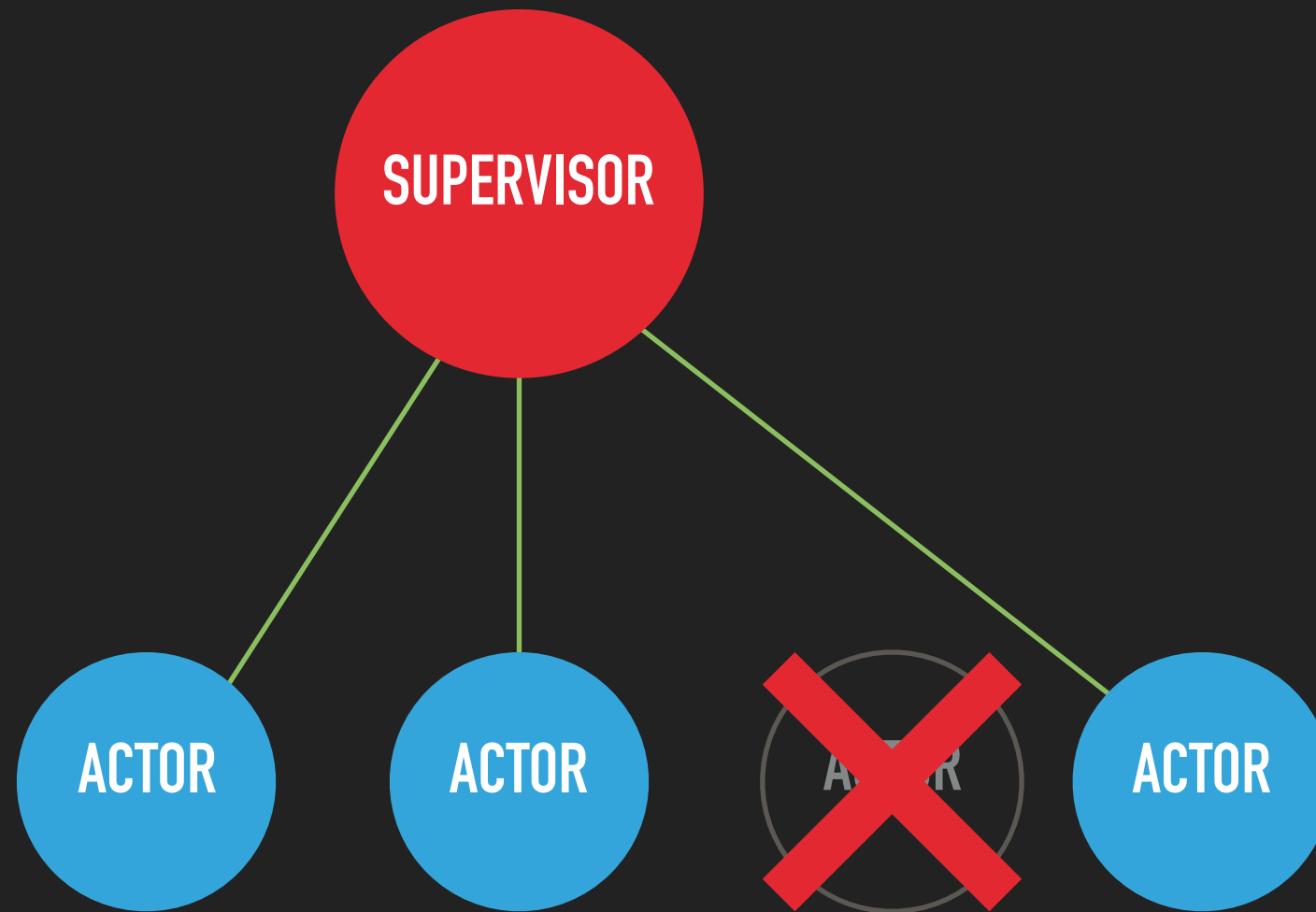
AVAILABILITY

RESILIENCE - ACTOR



AVAILABILITY

RESILIENCE - ACTOR



**AVAILABILITY**

**RELIABILITY**

**CLIENT**



**ENDPOINT NODE**



**BUSINESS NODE**

**BUSINESS NODE**

AVAILABILITY

RELIABILITY

CLIENT

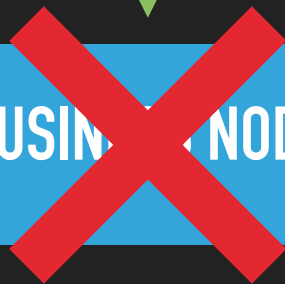


ENDPOINT NODE



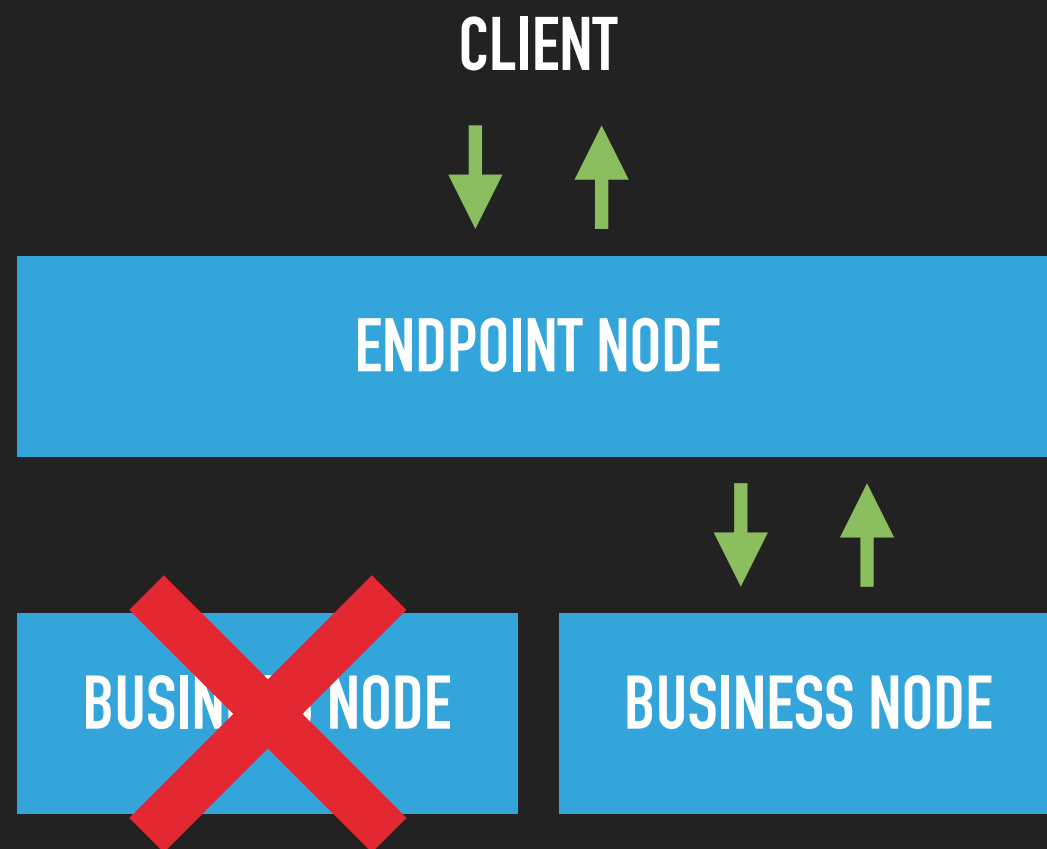
BUSINESS NODE

BUSINESS NODE



AVAILABILITY

RELIABILITY





# WRAPPING UP

**NO SILVER BULLET**



O'REILLY

# Designing for Scalability with Erlang/OTP

IMPLEMENTING ROBUST, FAULT-TOLERANT SYSTEMS



Francesco Cesarini & Steve Vinoski

Designing for Scalability with Erlang/OTP

# Programming Elixir 1.3

Functional

|> Concurrent

|> Pragmatic

|> Fun

Dave Thomas

Foreword by  
José Valim,  
Creator of Elixir



# Programming Elixir 1.3

# Erlang/Elixir Korea User Group

<https://www.facebook.com/groups/elixir.korea/>

# REFERENCES

## ▶ Books

- ▶ Designing for Scalability with Erlang/OTP
- ▶ Programming Elixir 1.3 Reliability
- ▶ The\_Little\_Elixir\_&\_OTP\_Guidebook

## ▶ Videos

- ▶ Experimenting with Superpowered Web Services: Phoenix on Riak\_Core  
<https://www.youtube.com/watch?v=sYYOLaJ-VDQ>
- ▶ Erlang Factory SF 2016 - Concurrency + Distribution = Scalability + Availability, a Journey Architecting Erlang Systems  
[https://www.youtube.com/watch?v=\\_IZMQMuphfo](https://www.youtube.com/watch?v=_IZMQMuphfo)

**THANK YOU...**