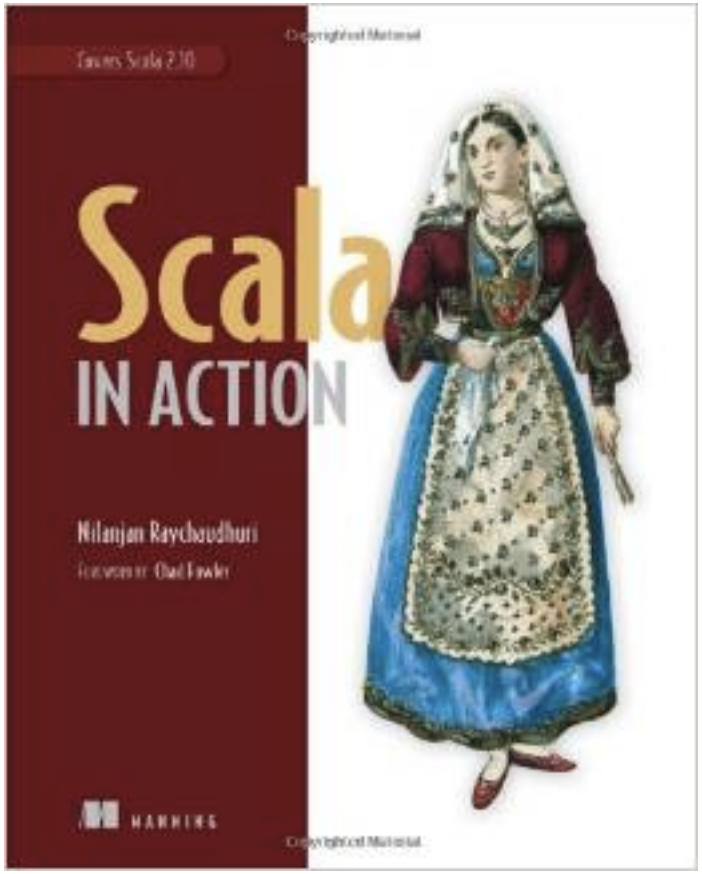

Typesafe

Go Reactive workshop

Nilanjan Raychaudhuri
@nraychaudhuri

@nraychaudhuri



Application requirements have changed

New requirements

Users

Users are demanding richer and more personalized experiences.

Yet, at the same time, expecting blazing fast load time.

Applications

Mobile and HTML5; Data and compute clouds; scaling on demand.

Modern application technologies are fueling the always-on, real-time user expectation.

Businesses

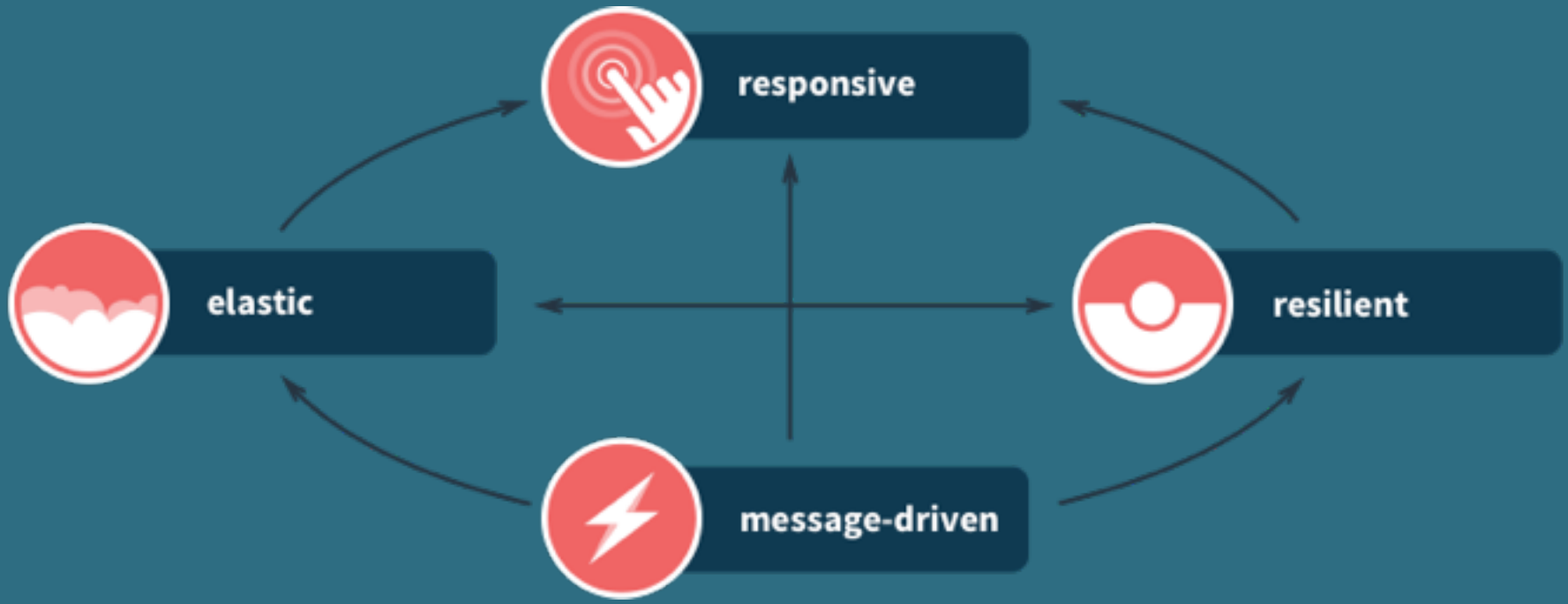
Businesses are being pushed to react to these changing user expectations…

...and embrace modern application requirements.

Apps in the 60s-90s were written for	Apps today are written for
Single machines	Clusters of machines
Single core processors	Multicore processors
Expensive RAM	Cheap RAM
Expensive disk	Cheap disk
Slow networks	Fast networks
Few concurrent users	Lots of concurrent users
Small data sets	Large data sets
Latency in seconds	Latency in milliseconds

As a matter of necessity,
businesses are going Reactive

Reactive Traits



Essence of Reactive applications

Responsiveness

Responsive

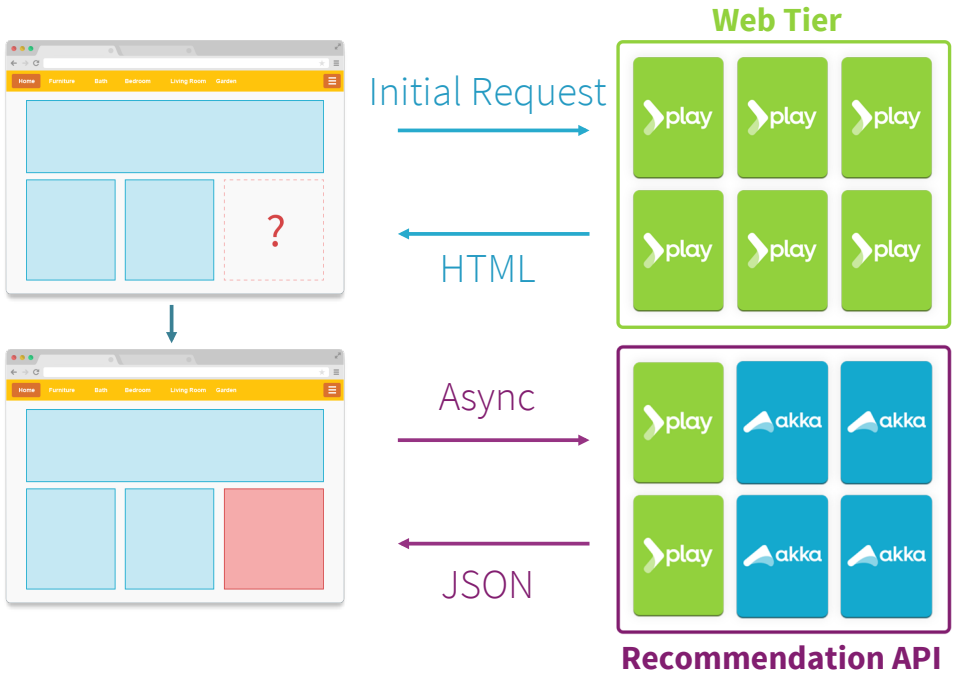
- Real-time, engaging, rich and collaborative
  - Create an open and ongoing dialog with users
  - More efficient workflow; inspires a feeling of connectedness
  - Fully Reactive enabling push instead of pull

"The move to these technologies is already paying off. Response times are down for processor intensive code–such as image and PDF generation–by around 75%."

Brian Pugh, VP of Engineering, Lucid Software

React to **users**

- Imagine **guaranteed response** times.
- **How?** Progressively enhanced UI, modular architecture.



Elastic

Reactive applications scale up and down to meet demand

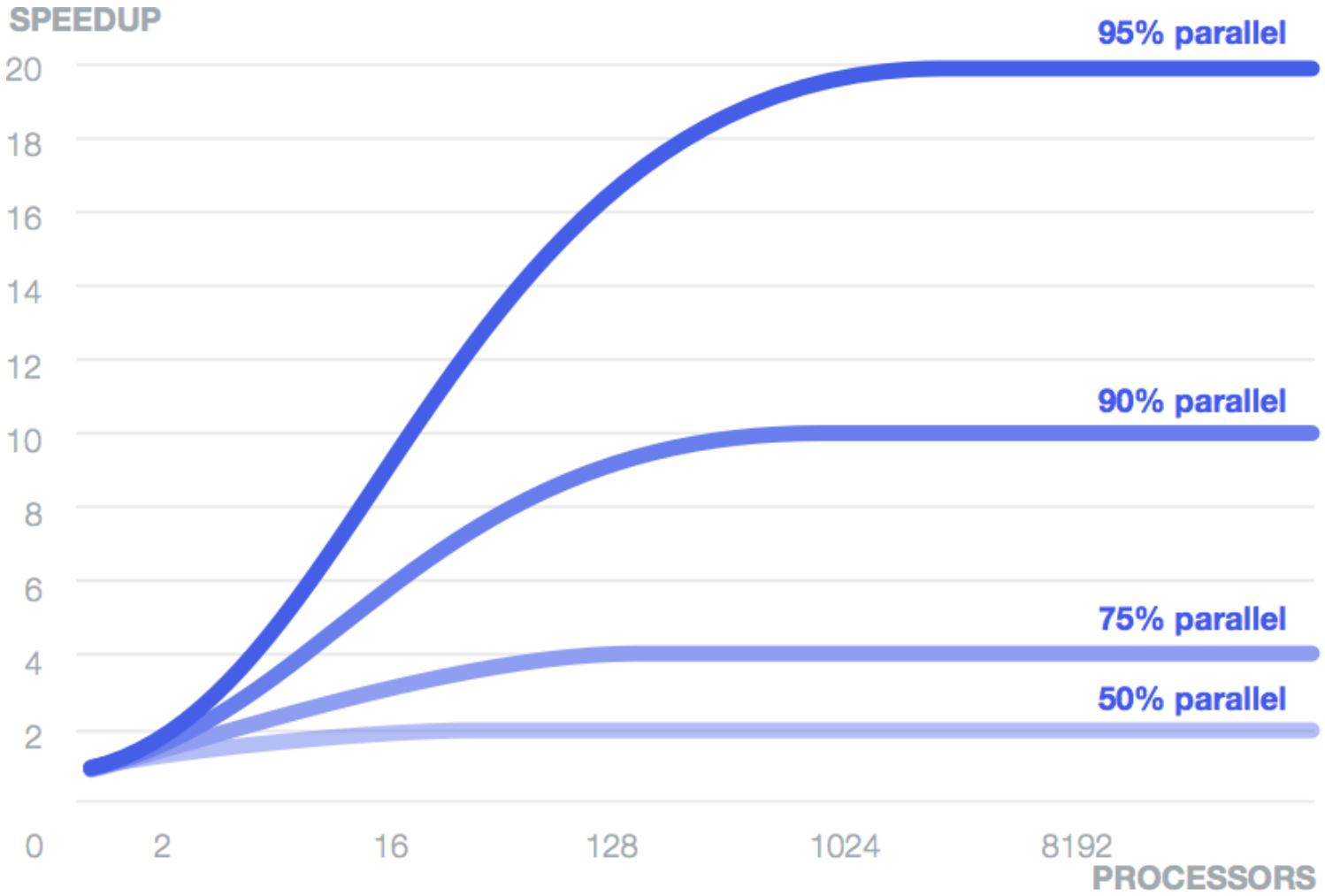
Elastic

- Scalability and elasticity to embrace the Cloud
  - Leverage all cores via asynchronous programming
  - Clustered servers support joining and leaving of nodes
  - More cost-efficient utilization of hardware

“Our traffic can increase by as much as 100x for 15 minutes each day.
Until a couple of years ago, noon was a stressful time.
Nowadays, it’s usually a non-event.”

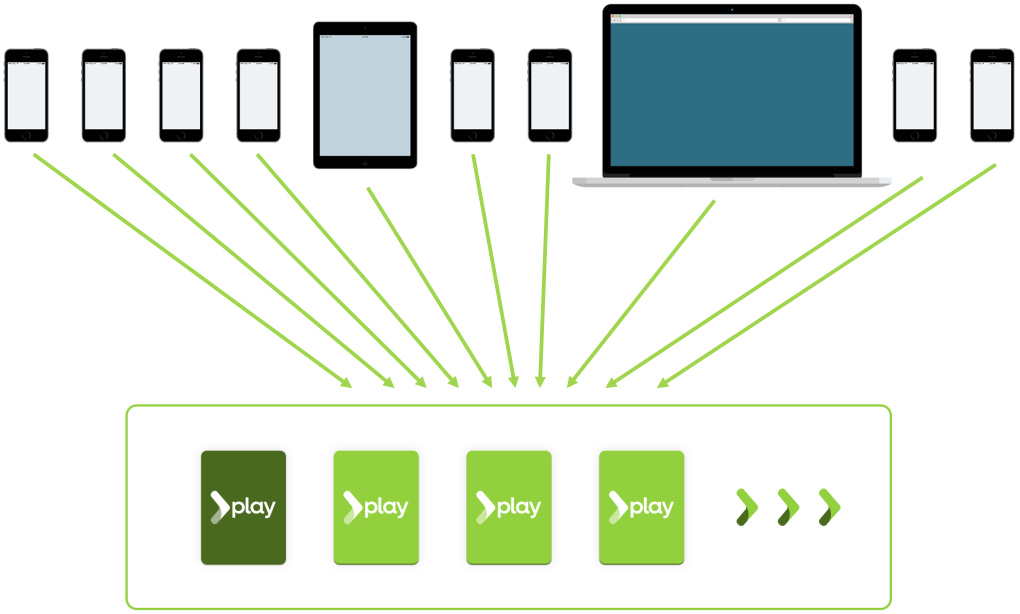
Eric Bowman, VP Architecture, Gilt Groupe

Amdahl's Law


SPEEDUP
20
18
16
14
12
10
8
6
4
2
0
2
16
128
1024
8192
PROCESSORS
95% parallel
90% parallel
75% parallel
50% parallel

React to **load**

- Imagine **guaranteed uptime**, even during spikes on Black Friday and Boxing Day.
- **How?** Scaling out, stateless architecture.



Resilient

Stay responsive in face of failure

Resilient

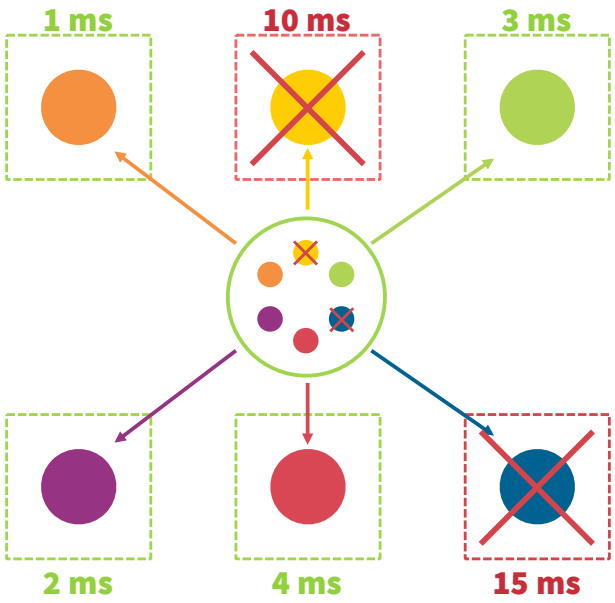
- Failure is embraced as a natural state in the app lifecycle
  - Resilience is a first-class construct
  - Failure is detected, isolated, and managed
  - Applications self heal

“The Typesafe Reactive Platform helps us maintain a very aggressive development and deployment cycle, all in a fail-forward manner. It’s now the default choice for developing all new services.”

Peter Hausel, VP Engineering, Gawker Media

React to **failure**

- Imagine **protecting** back-end heritage systems from cascading failures.
- **How?** Circuit breakers, Reactive application development.



Message-driven

Reactive applications are architected based on loosely coupled design

Message-Driven

- Loosely coupled architecture, easier to extend, maintain, evolve
  - Asynchronous and non-blocking
  - Concurrent by design, immutable state
  - Lower latency and higher throughput

"Clearly, the goal is to do these operations concurrently and non-blocking, so that entire blocks of seats or sections are not locked. We're able to find and allocate seats under load in less than 20ms without trying very hard to achieve it."

Andrew Headrick, Platform Architect, Ticketfly

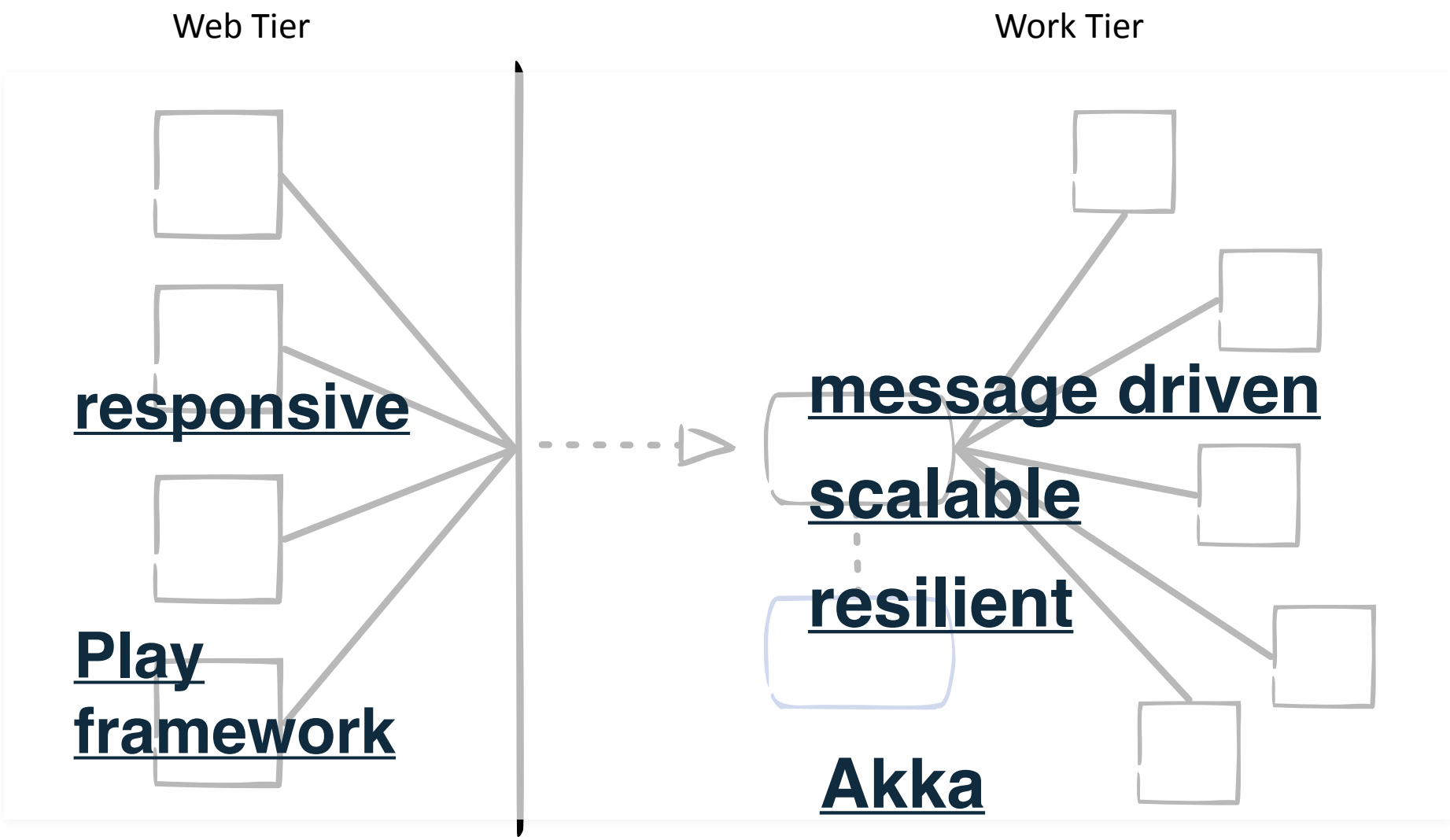
FP

Functional programming is key

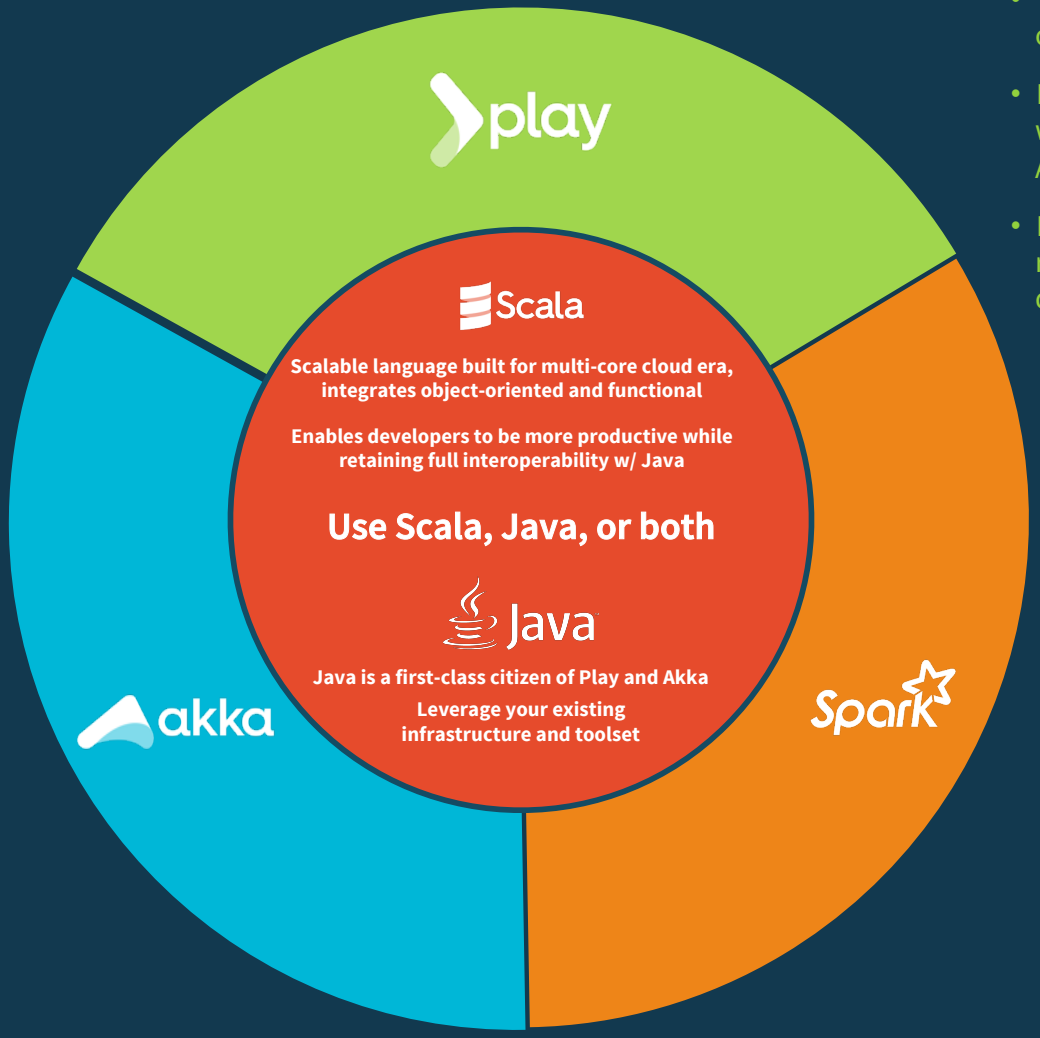
- We want to be asynchronous and non-blocking
- We need to ensure that our data is protected without locks
- We want to compose individual tasks into broader business logic
- Functional programming is critical to meeting these needs
  - Declarative
  - Immutable
  - Referentially transparent
  - Pure functions that only have inputs and outputs

Big picture

Reference Architecture



Typesafe Reactive Platform



Play Framework

- Toolkit for building RESTful APIs on the JVM
- Rails-like developer productivity with first class Java and Scala APIs
- Built on Akka for performance, resilience and scalability on demand

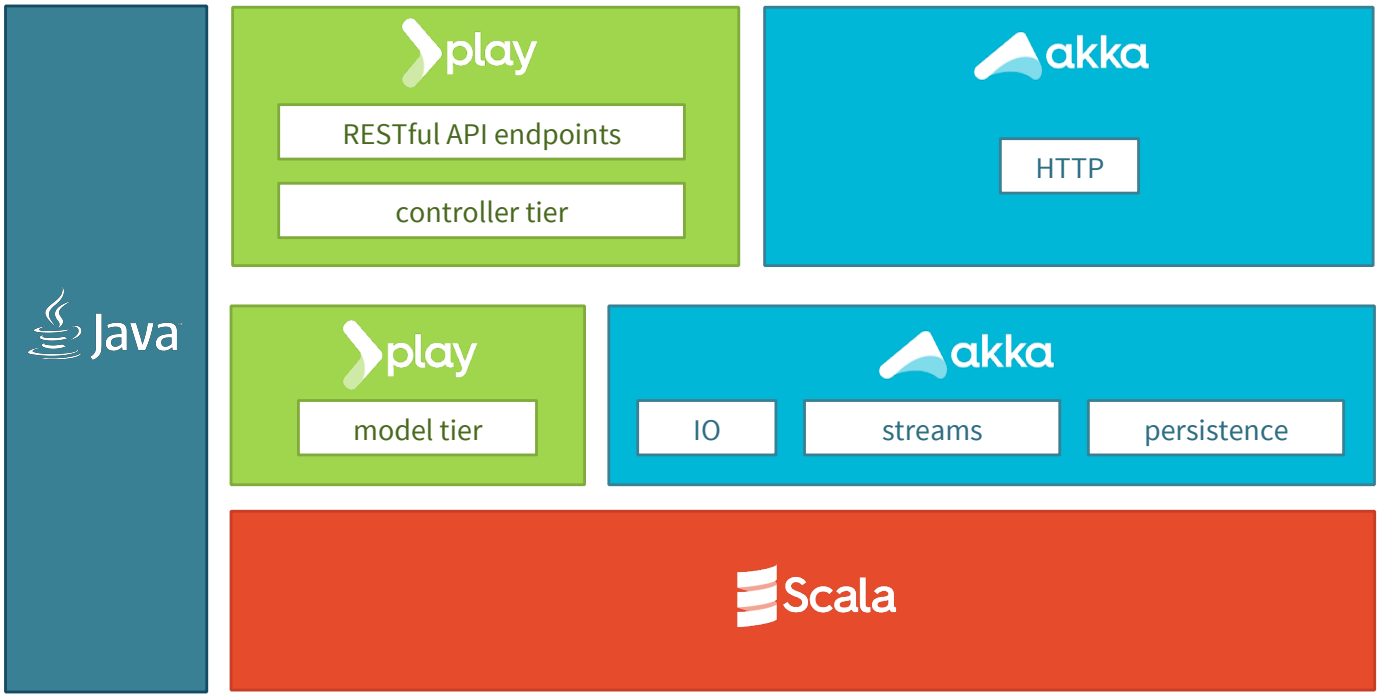
Akka

- Lightweight toolkit for creating message-driven applications on the JVM
- Create loosely coupled, resilient and elastic systems that adapt and scale to meet the needs of your business

Apache Spark

- Typesafe offers developer support for Apache Spark, an open-source cluster computing framework
- Run programs up to 100x faster than Hadoop MapReduce in memory
- Write applications quickly in Java or Scala

Typesafe Reactive Platform



Broad Adoption

Workshop

Case study

Cocktail Bar

http://gxmediagy.com/wp-content/uploads/2012/11/cocktails.jpg

Group exercise

- Code walk through
- Explore the case study project
- Get comfortable with activator
- Build and run

Message-driven

Reactive applications are architected based on loosely coupled design

Message-Driven (Recap)

- Loosely coupled architecture, easier to extend, maintain, evolve
  - Asynchronous and non-blocking
  - Concurrent by design, immutable state
  - Lower latency and higher throughput

"Clearly, the goal is to do these operations concurrently and non-blocking, so that entire blocks of seats or sections are not locked. We're able to find and allocate seats under load in less than 20ms without trying very hard to achieve it."

Andrew Headrick, Platform Architect, Ticketfly

Actors

Actor model

A computational model that embodies:

✓ Processing

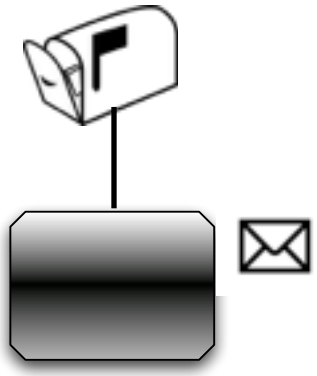
✓ Storage

✓ Communication

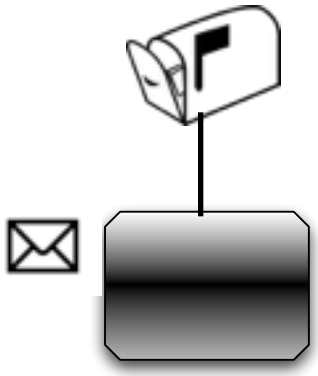
Supports 3 axioms—when an Actor receives a message it can:

1. Create new Actors
2. Send messages to Actors it knows
3. Designate how it should handle the next message it receives

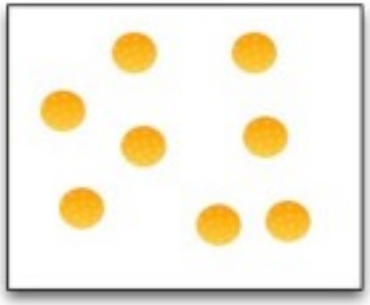
Actor model



Actor

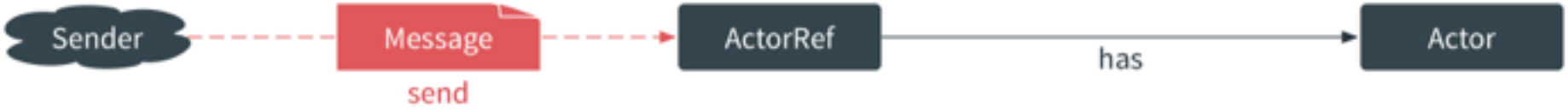


Actor



Thread pool (ExecutionContext)

Anatomy of an Actor I

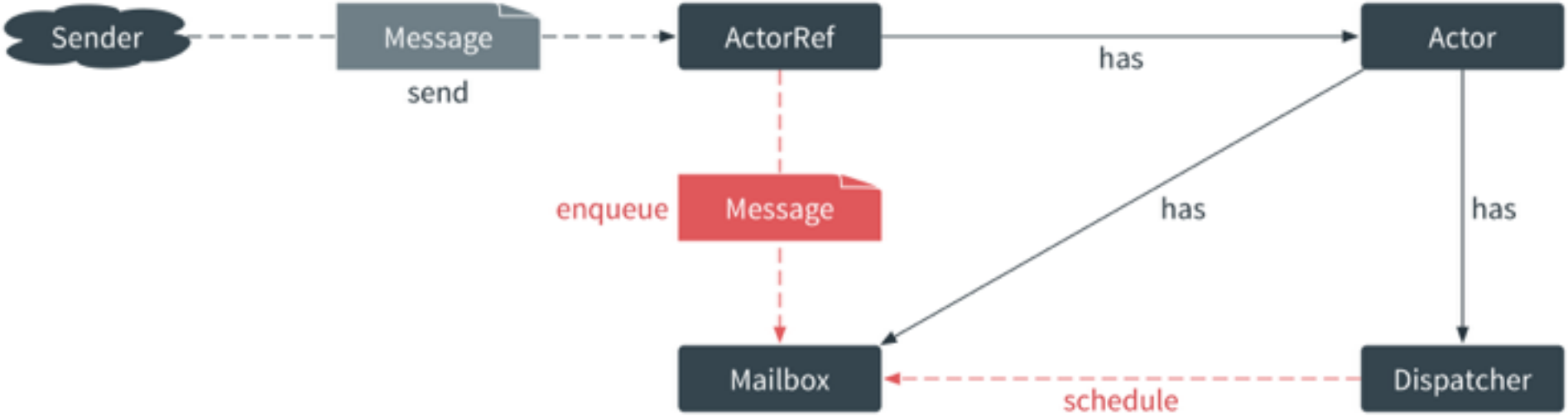


Anatomy of an Actor I

- Each actor is represented by an ActorRef

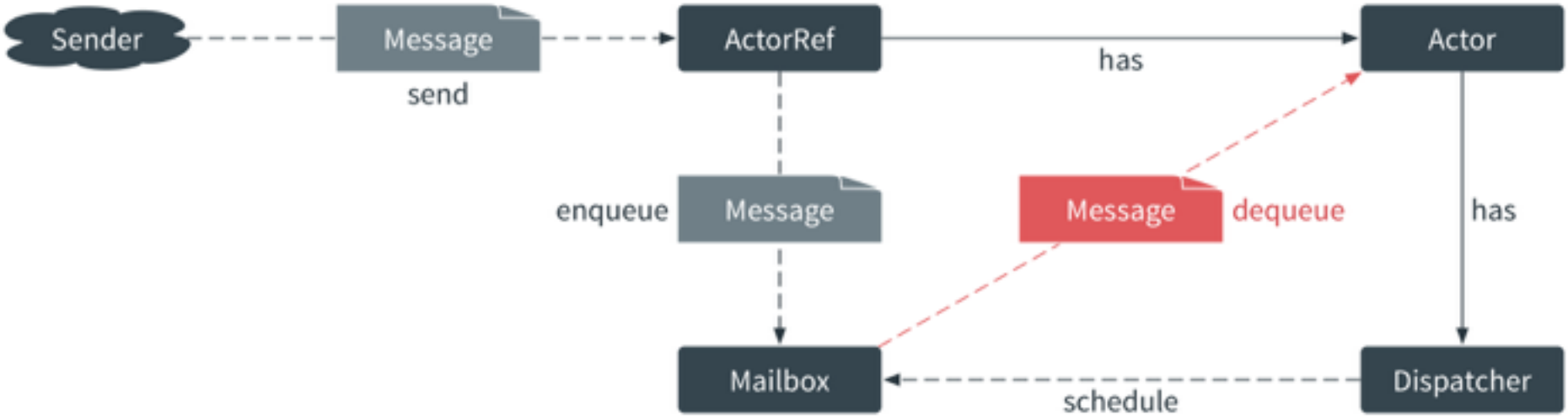
Anatomy of an Actor II

- Each actor is has a mailbox and a dispatcher



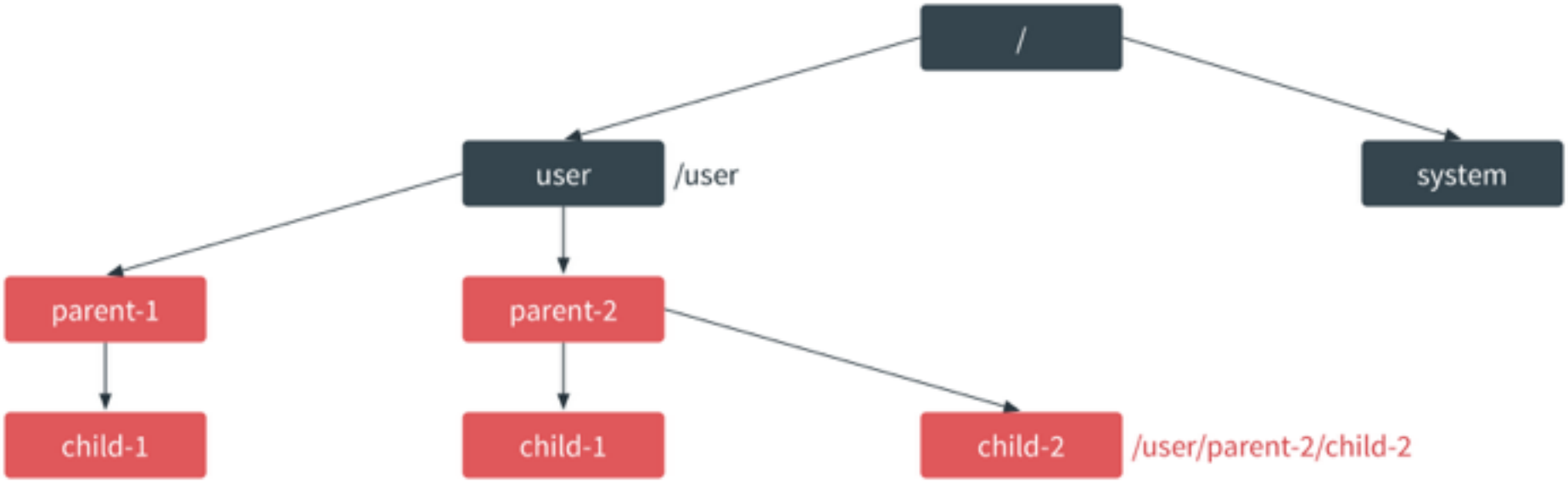
Anatomy of an Actor III

- Only one message at a time is passed to the actor



ActorSystem I

- An actor system is a collaborating ensemble of actors
- Actors are arranged in hierarchy



Group exercise - Create CocktailBar

- Review the actor code
- create an instance of CocktailBar in CocktailBarApp
- Experiment with sending messages

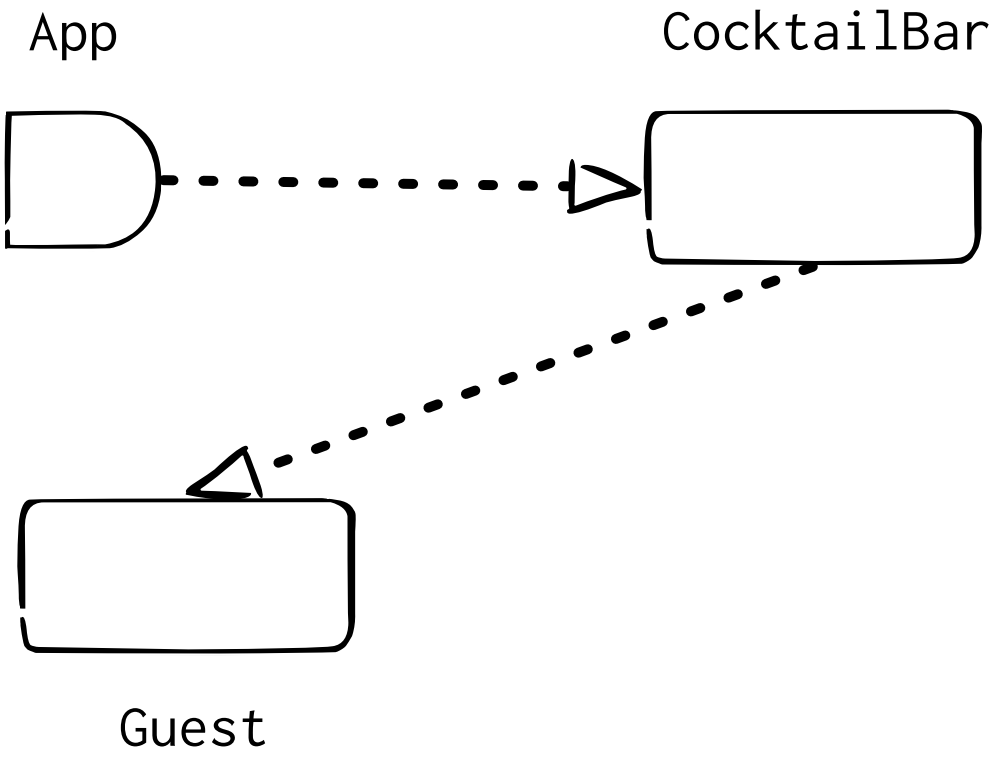
ActorContext

- This provides contextual information and operations
  - Access to self and the current sender
  - Access to parent and children
  - Create child actors and stop actors
  - Death watch, change behavior, etc. (more to come later)

Actors and Mutability

- Actors can have mutable state (don’t share it)
- Actors exclusively communicate with message passing
  - Messages should be immutable

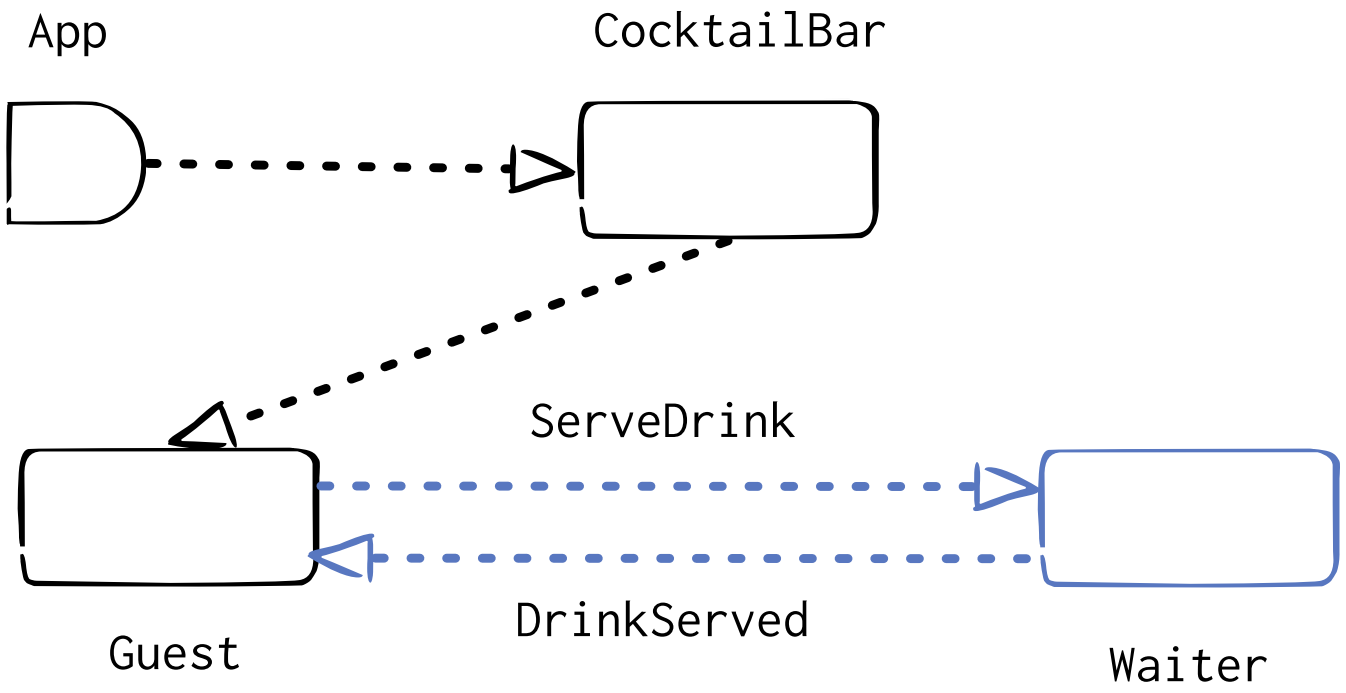
Group exercise - Create Guest I



Group exercise - Create Guest II

- Create a guest actor with an empty behavior
- Add CreateGuest message to CocktailBar
- On receiving create guest message, create new guest child actors
- Run and verify the lifecycle debug messages

Exercise - Create Waiter I



Exercise - Create Waiter II

- **Create the Waiter actor:**
- Add the ServeDrink and DrinkServed case classes to the message protocol, each with a drink parameter of type Drink
- Create a Props factory
- On receiving ServeDrink(drink) respond with DrinkServed(drink)

Exercise - Create Waiter III

- **Change Guest:**
- Add a waiter parameter of type ActorRef
- Add a favoriteDrink parameter of type Drink
- Add a mutable drinkCount field of type Int
- Add the private DrinkFinished message to the message protocol
- On receiving DrinkServed(drink)
- increase drinkCount by one and
- log "Enjoying my {drinkCount}. yummy {drink}!" at info
- On receiving DrinkFinished send ServeDrink(favoriteDrink) to waiter

Exercise - Create Waiter IV

- **Change CocktailBar:**
  - Add a favoriteDrink parameter of type Drink to CreateGuest
  - Create a Waiter with name "waiter"; use a createWaiter factory method
- Run the app, create some Guests and make sure no errors occur
- Hint: Enter g DRINK or guest DRINK where DRINK has to be the first letter of one of the defined drinks, i.e. 'a', 'm' or 'p'; if you omit DRINK, Akkarita will be used by default
- Why don't you see any log messages from Guests enjoying their drinks?

Scheduler Service I

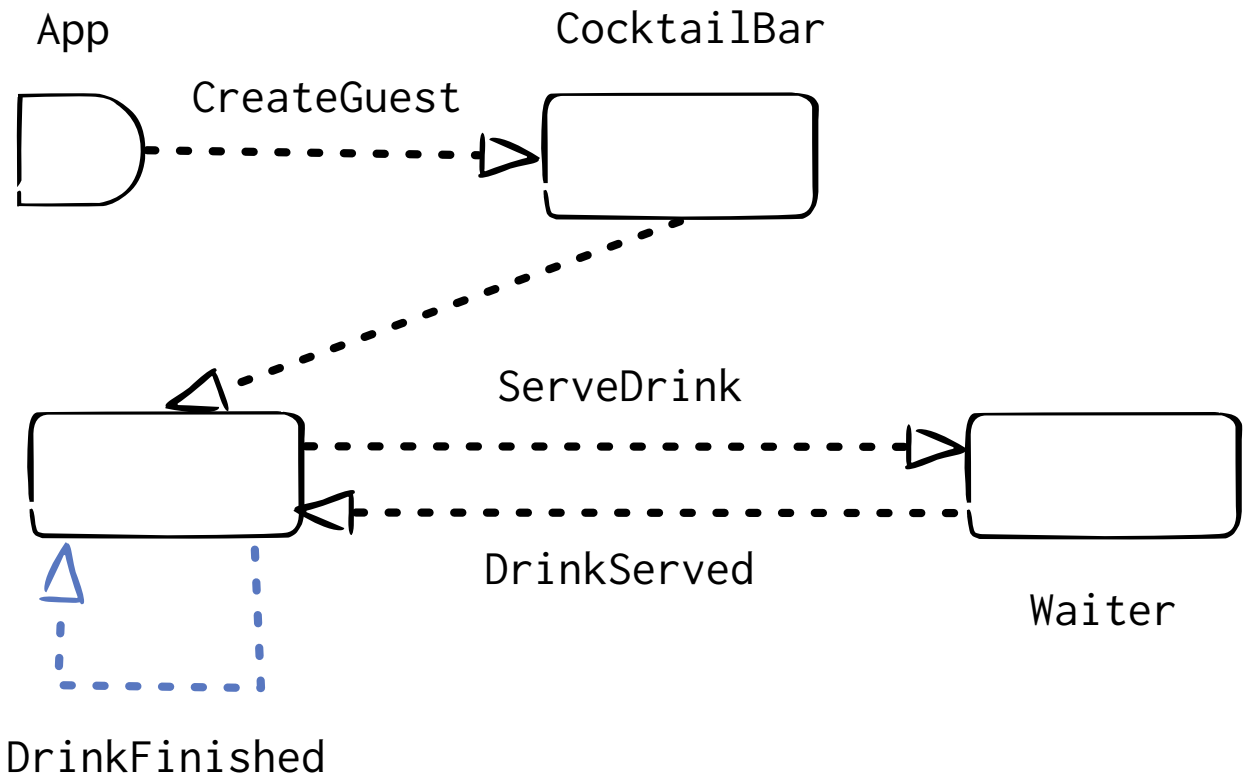
```
import akka.actor.AbstractLoggingActor;
import scala.concurrent.duration.FiniteDuration;

private void scheduleCoffeeFinished(){
  context().system()
  .scheduler()
  .scheduleOnce(finishCoffeeDuration, self(),
      CoffeeFinished.Instance, context().dispatcher(), self());
}
```

Scheduler Service II

- The actor system offers a scheduler:
- You can run a task or send a message to an actor
- This can be scheduled once or periodically
- You need to pass an ExecutionContext, e.g. an actor's dispatcher
- In the the configuration file, you can use expressions like "5 seconds", "2m" for durations

Exercise - Use scheduler service I



Exercise - Use scheduler service II

- **Change Guest:**
- Add a finishDrinkDuration parameter of type scala.concurrent.duration.FiniteDuration
- On receiving DrinkServed schedule sending DrinkFinished to itself after finishDrinkDuration
- What other change is needed to get the drinking started?

OO vs Actor vs FP?

Good ideas take long to spread

Defined in 1963 by Carl Hewitt

Popularized by Erlang

Resilient

Stay responsive in face of failure

Resilient

- Failure is embraced as a natural state in the app lifecycle
  - Resilience is a first-class construct
  - Failure is detected, isolated, and managed
  - Applications self heal

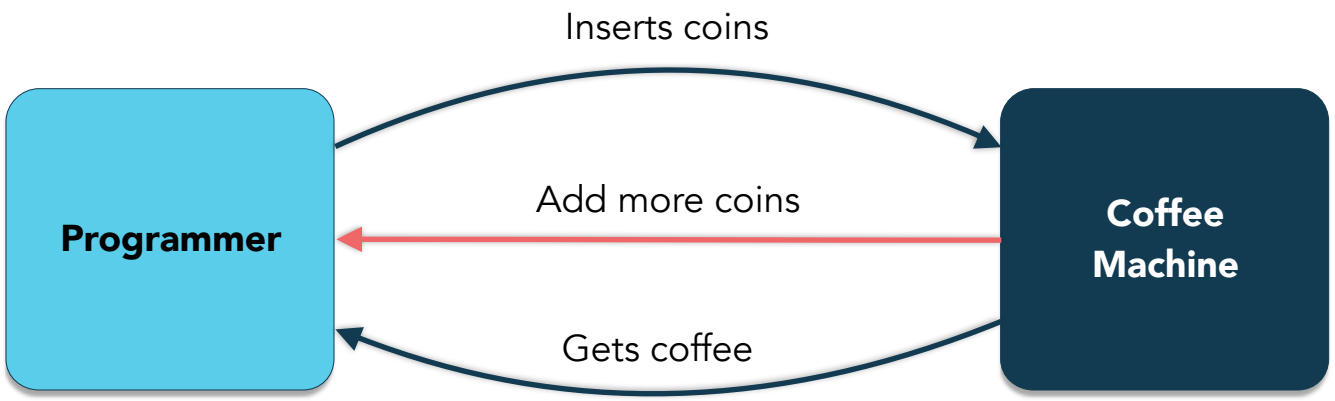
“The Typesafe Reactive Platform helps us maintain a very aggressive development and deployment cycle, all in a fail-forward manner. It’s now the default choice for developing all new services.”

Peter Hausel, VP Engineering, Gawker Media

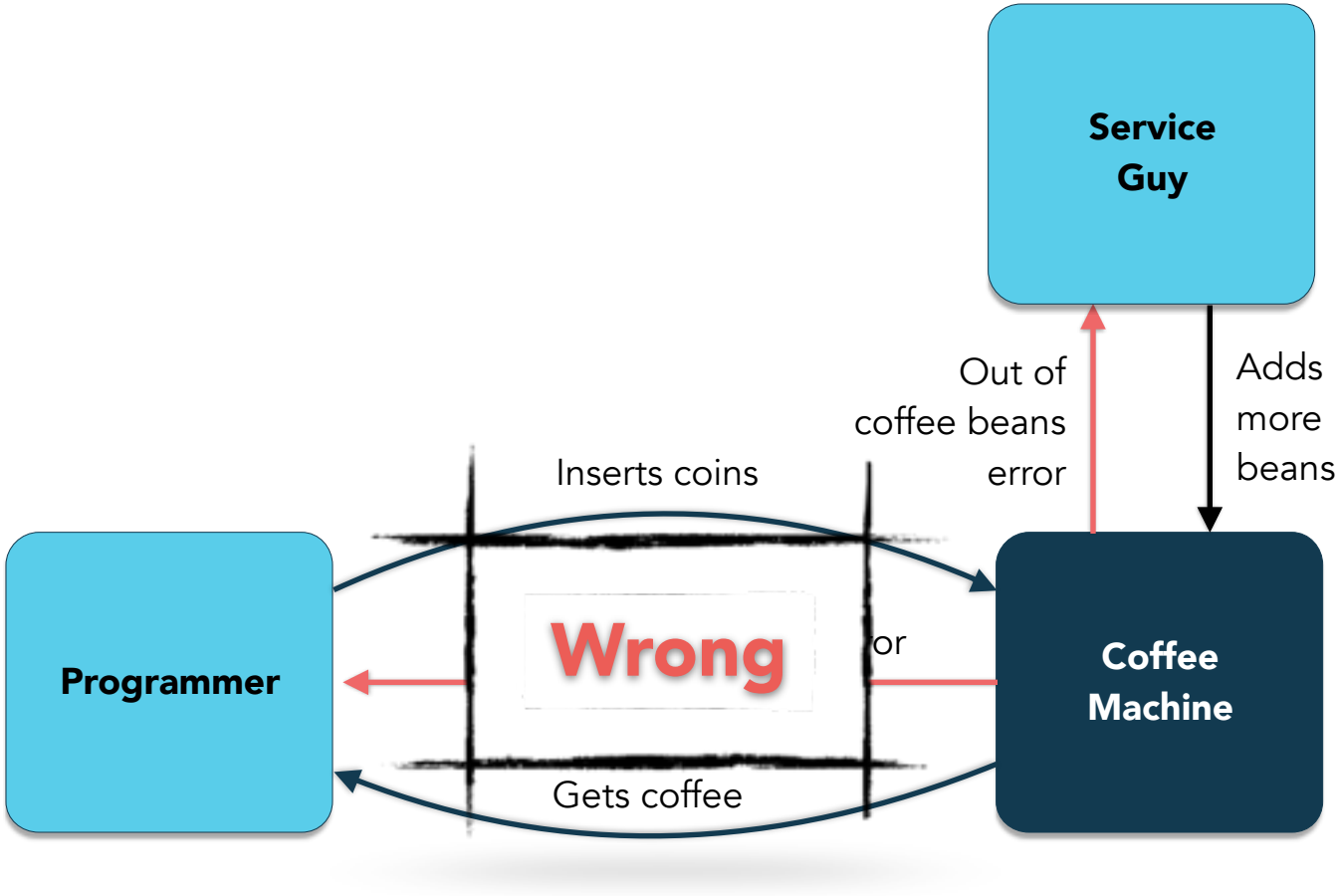

Let it crash

Think Vending Machine

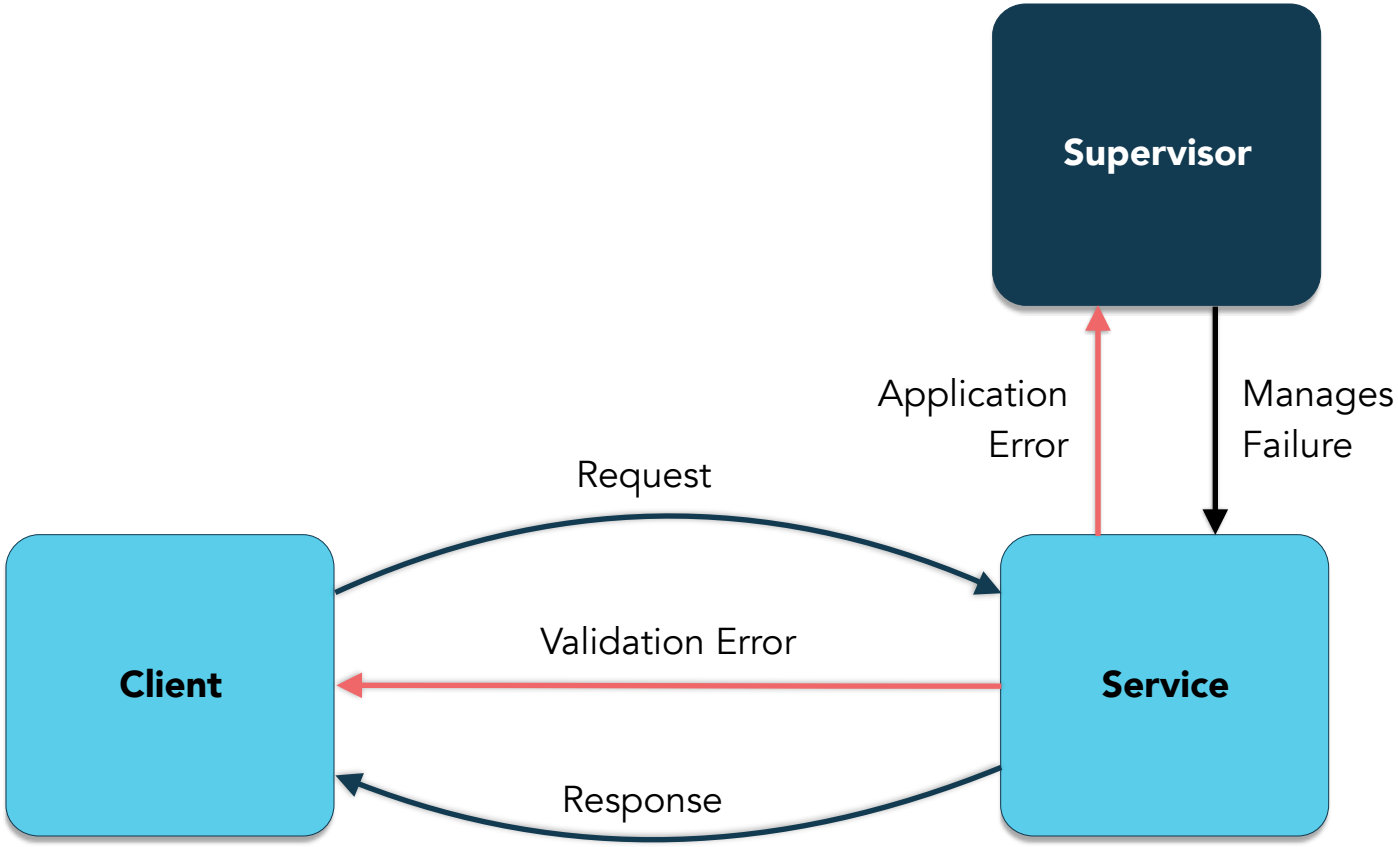
Think Vending Machine



Think Vending Machine

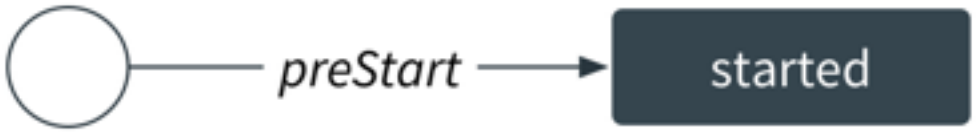


The Right Way



Actor lifecycle I

- Starting actors:
- Creating an actor automatically starts it
- A started actor is fully operable, i.e. it can handle messages
- You can override the preStart hook



Actor lifecycle II

- Stopping actors
- An actor can stop itself or be stopped by another one
- A stopped actor is no longer operable, i.e. it won't process any messages
- You can override the postStop hook, after which the actor is considered terminated and becomes available for garbage collection

Actor lifecycle III

- Stopping actors
- Stopping an actor is an asynchronous and recursive operation
- API:

```
context.stop(self)
context.stop(other)
```

Exercise - Create faulty actor I

- **Change Guest:**
- Add a maxDrinkCount parameter
- Add the DrunkException message extending IllegalStateException to the companion object
- On receiving DrinkFinished throw the DrunkException if drinkCount exceeds maxDrinkCount
- Change CocktailBar and CocktailBarApp: Make it possible to create Guests with a maxDrinkCount
- Run the app, create a Guest with an individual maxDrinkCount less than the global one and watch its lifecycle
- Hint: Enter g 2 or guest 2 to create a Guest with a maxDrinkCount of 2

Fault Tolerance

```
@Override
public SupervisorStrategy supervisorStrategy() {
  return new OneForOneStrategy(false, DeciderBuilder.
    match(SomeException.class, e -> SupervisorStrategy.stop())
    ...
    .build())
}
```

- As you can see, a faulty actor doesn't bring down the whole system
- This fault tolerance is implemented through parental supervision
- Each actor has a supervisor strategy for handling failure of child actors

Supervisor Strategies

- Akka ships with two highly configurable supervisor strategies:
  - OneForOneStrategy: Only the faulty child is affected when it fails
  - AllForOneStrategy: All children are affected when one child fails
- Both are configured with a DeciderBuilder:
  - Decider = PartialFunction<Throwable, Directive>
  - A decider maps specific failure to one of the possible directives
  - If not defined for some failure, the supervisor itself is considered faulty

Supervisor Strategy Directives

- Resume: Simply resume message processing
- Restart:
  - Transparently replace affected actor(s) with new instance(s)
  - Then resume message processing
- Stop: Stop affected actor(s)
- Escalate: Delegate the decision to the supervisor's parent

Exercise - Customize supervision

- Look up the default supervisor strategy in the Akka source code
- Drunk Guests certainly should not be restarted
- Apply a custom supervisor strategy to stop them instead
- Run the app, create a Guest with an individual maxDrinkCount and verify that it is stopped

Elastic

Reactive applications scale up and down to meet demand

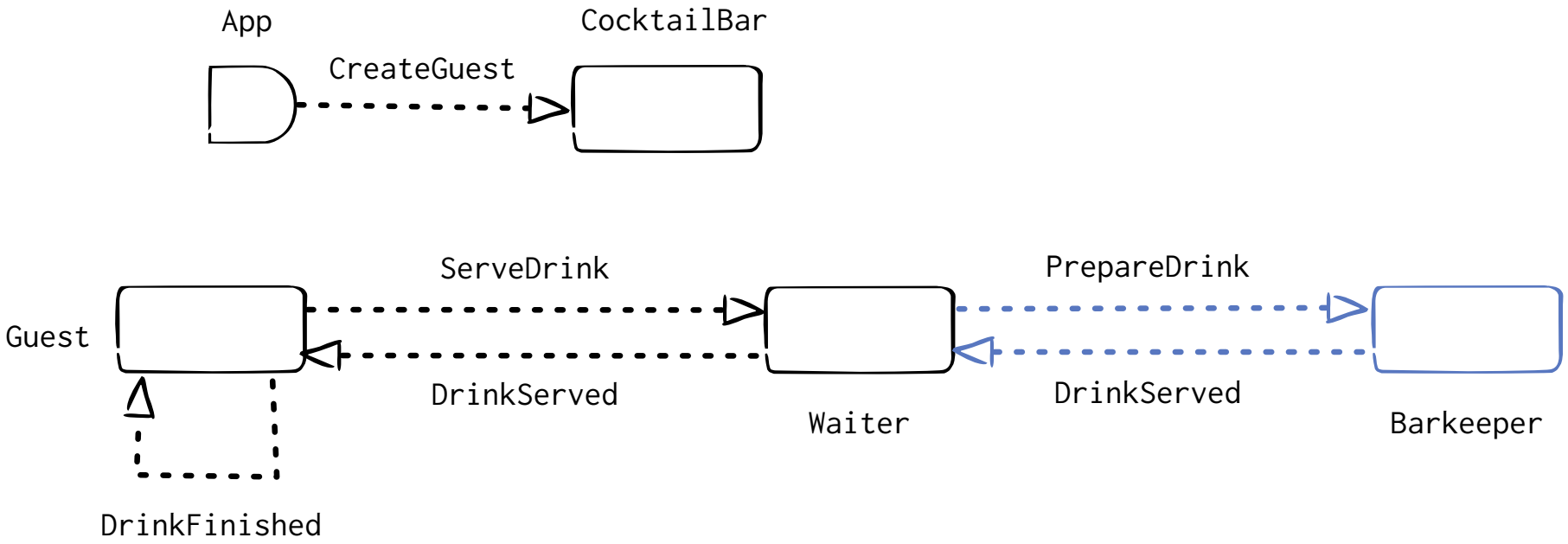
Elastic

- Scalability and elasticity to embrace the Cloud
  - Leverage all cores via asynchronous programming
  - Clustered servers support joining and leaving of nodes
  - More cost-efficient utilization of hardware

“Our traffic can increase by as much as 100x for 15 minutes each day.
Until a couple of years ago, noon was a stressful time.
Nowadays, it’s usually a non-event.”

Eric Bowman, VP Architecture, Gilt Groupe

Exercise - Keep actors busy



Exercise - Keep actors busy II

- **Create the Barkeeper actor:**
- Add a prepareDrinkDuration parameter of type FiniteDuration
- Add the PrepareDrink and DrinkPrepared case classes to the message protocol, each with
- a drink parameter of type Drink and
- a guest parameter of type ActorRef
- Add a Props factory
- On receiving PrepareDrink(drink, guest) busily prepare the drink for prepareDrinkDuration, then respond with DrinkPrepared(drink, guest)
- Hint: Use busy method from Utils

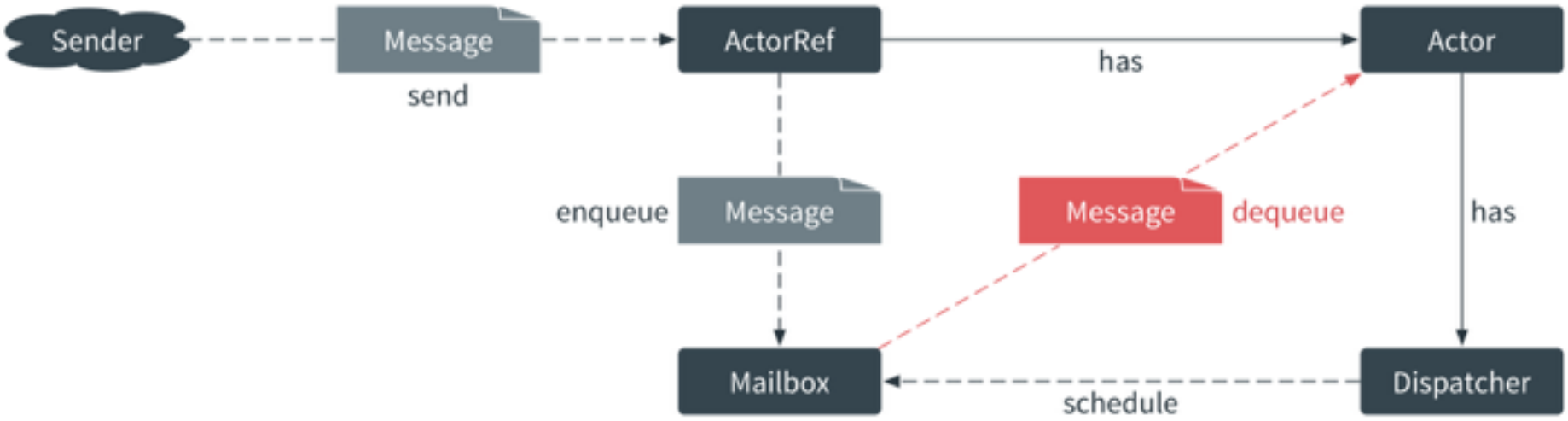
Exercise - Keep actors busy III

- **Change Waiter:**
- Instead of serving drinks immediately, let them be prepared by Barkeeper
- **Change CocktailBar**:
- Create a Barkeeper with name "barkeeper"; use a createBarkeeper factory method
- For prepareDrinkDuration use a configuration value with key cocktail-bar.barkeeper.prepare-drink-duration
- Run the app, create some Guests and verify that everything still works as expected

Exercise - Detect bottleneck

- In the configuration file set
- max-drink-count to 1000 (no more drunk guests)
- prepare-drink-duration to 2 seconds
- finish-drink-duration to 2 seconds
- Run the app and create one Guest, then another one, then some more and watch the throughput per Guest
- Why does the application not scale? Where's the bottleneck?

Message processing revisited

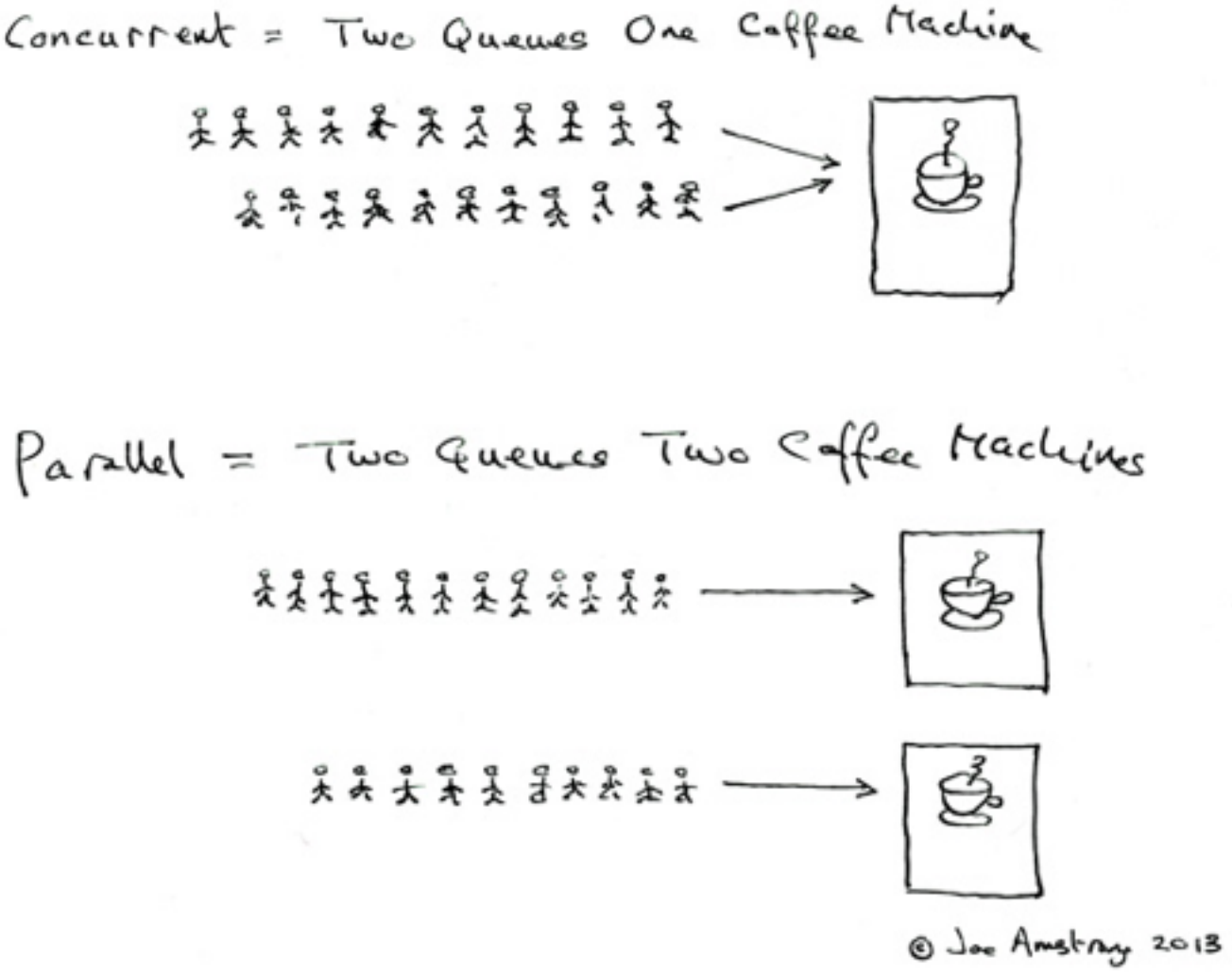


- An actor processes (at most) one message at a time
- If you want to scale up, you have to use multiple actors in parallel

Concurrency vs. Parallelism I

- Definition: Two or more tasks are concurrent, if the order in which they get executed in time is not predetermined
- In other words, concurrency introduces non-determinism
- Concurrent tasks may or may not get executed in parallel
- Hence concurrency is a more general concept than parallelism
- Concurrent programming is primarily concerned with the complexity that arises due to non-deterministic control flow
- Parallel programming aims at improving throughput and making control flow deterministic

Digression: Concurrency and Parallelism II



Routing Strategies provided by Akka

- RandomRoutingLogic
- RoundRobinRoutingLogic
- SmallestMailboxRoutingLogic
- ConsistentHashingRoutingLogic
- BroadcastRoutingLogic
- ScatterGatherFirstCompletedRoutingLogic
- To write your own routing strategy, extend RoutingLogic:
- Attention: The implementation must be thread-safe!

Router Actors

- Akka provides two flavors of self contained router actors:
  - Pool router: creates routees as child actors
  - Group router: routees are provided via actor path
- Message delivery is optimized:
  - Messages don't get enqueed in the mailbox of the router actor
  - Instead, messages are delivered to a routee directly

Creating a Router Actor

```
context.actorOf(
  Props(new SomeActor).withRouter(FromConfig()),
  "some-actor"
)

context.actorOf(
  RoundRobinPool(4).props(Props(new SomeActor)),
  "some-actor"
)
```

- Router actors must be created programmatically

Router Configuration Example

```
akka {
  actor {
    deployment {
      /top-level/child-a {
        router = smallest-mailbox-pool
        nr-of-instances = 4
      }
      /top-level/child-b {
        router = round-robin-pool
        resizer {
          lower-bound = 1
          upper-bound = 4
        }
```

Exercise: Use Routers

- Get rid of the bottleneck by using a router for the Barkeeper
- Use an externally configured round-robin pool router
- Run the app with different router configuration options and watch the throughput

Dispatchers

```
akka.actor.default-dispatcher = ...
```

```
context.actorOf(
  Props[SomeActor].withDispatcher("my-dispatcher")
)
```

- Dispatchers are Akka's engine, they make actors "tick" by
- implementing scala.concurrent.ExecutionContext and
- registering an actor's mailbox for execution
- Dispatchers determine execution time and context and therefore provide the physical capabilities for scaling up
- Each actor system has a dispatcher used as default for all actors
- You can set an externally configured dispatcher for an actor

Dispatchers provided by Akka

- Dispatcher (default): Event-driven dispatcher, sharing threads from a thread pool for its actors
- PinnedDispatcher: Dedicates a unique thread for each actor
- BalancingDispatcher: Event-driven dispatcher redistributing work from busy actors to idle ones
- CallingThreadDispatcher: For testing, runs invocations on the current thread only
- To use your own dispatcher, provide a MessageDispatcherConfigurator

Resilient - *revisited*

Self healing system

Group Exercise: Implement self healing

- Occasionally Barkeeper mixes wrong drink
- Waiter get frustrated
- Keep the bar running

Responsiveness

Responsive

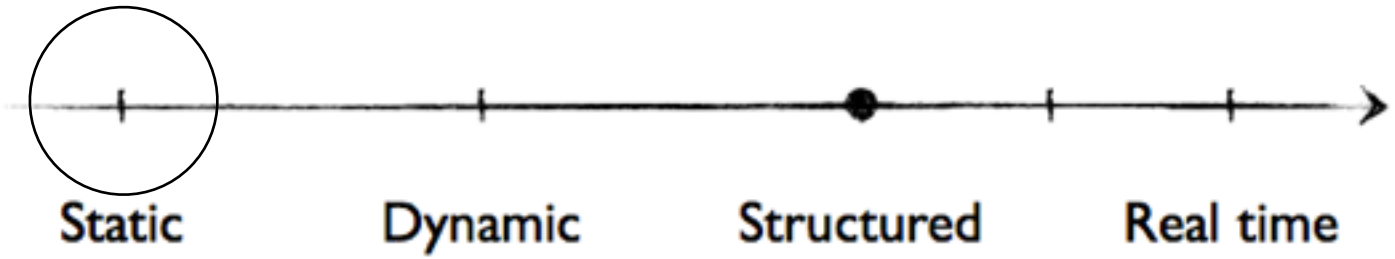
- Real-time, engaging, rich and collaborative
  - Create an open and ongoing dialog with users
  - More efficient workflow; inspires a feeling of connectedness
  - Fully Reactive enabling push instead of pull

"The move to these technologies is already paying off. Response times are down for processor intensive code–such as image and PDF generation–by around 75%."

Brian Pugh, VP of Engineering, Lucid Software

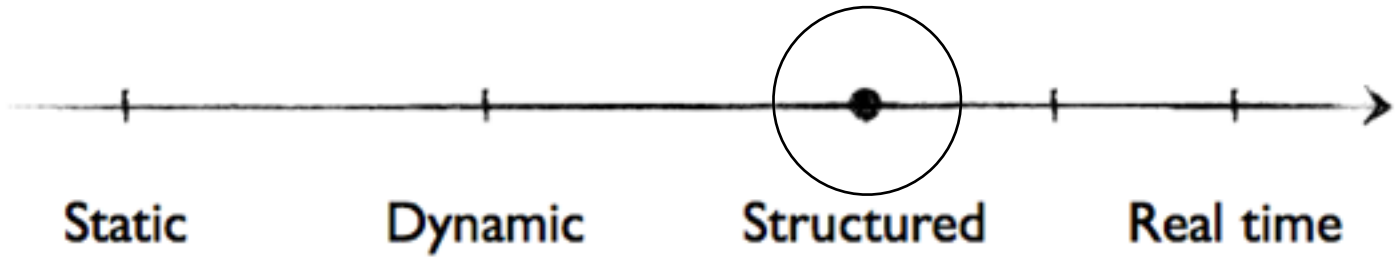

play

Web Evolved

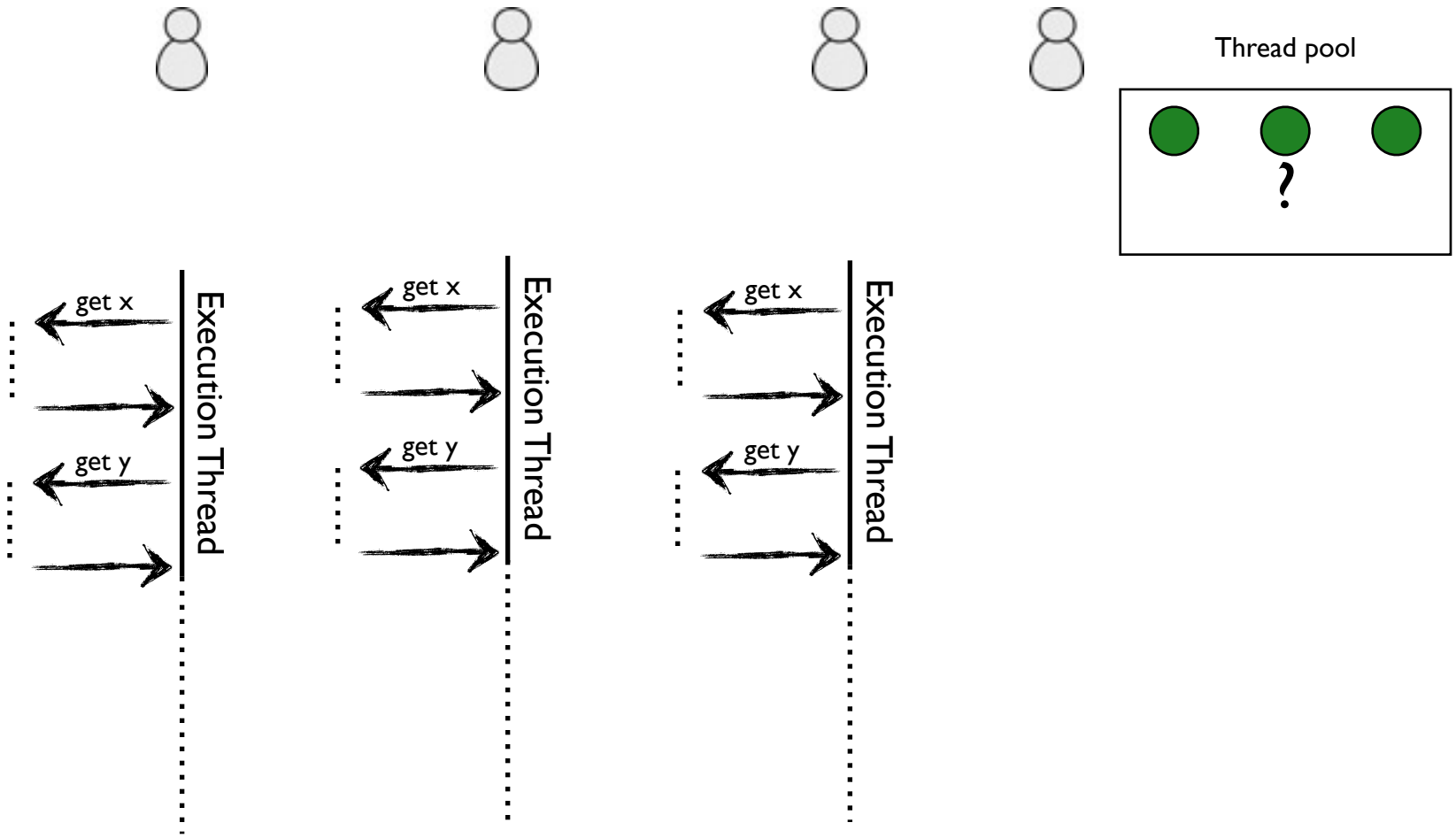


Explore

Present

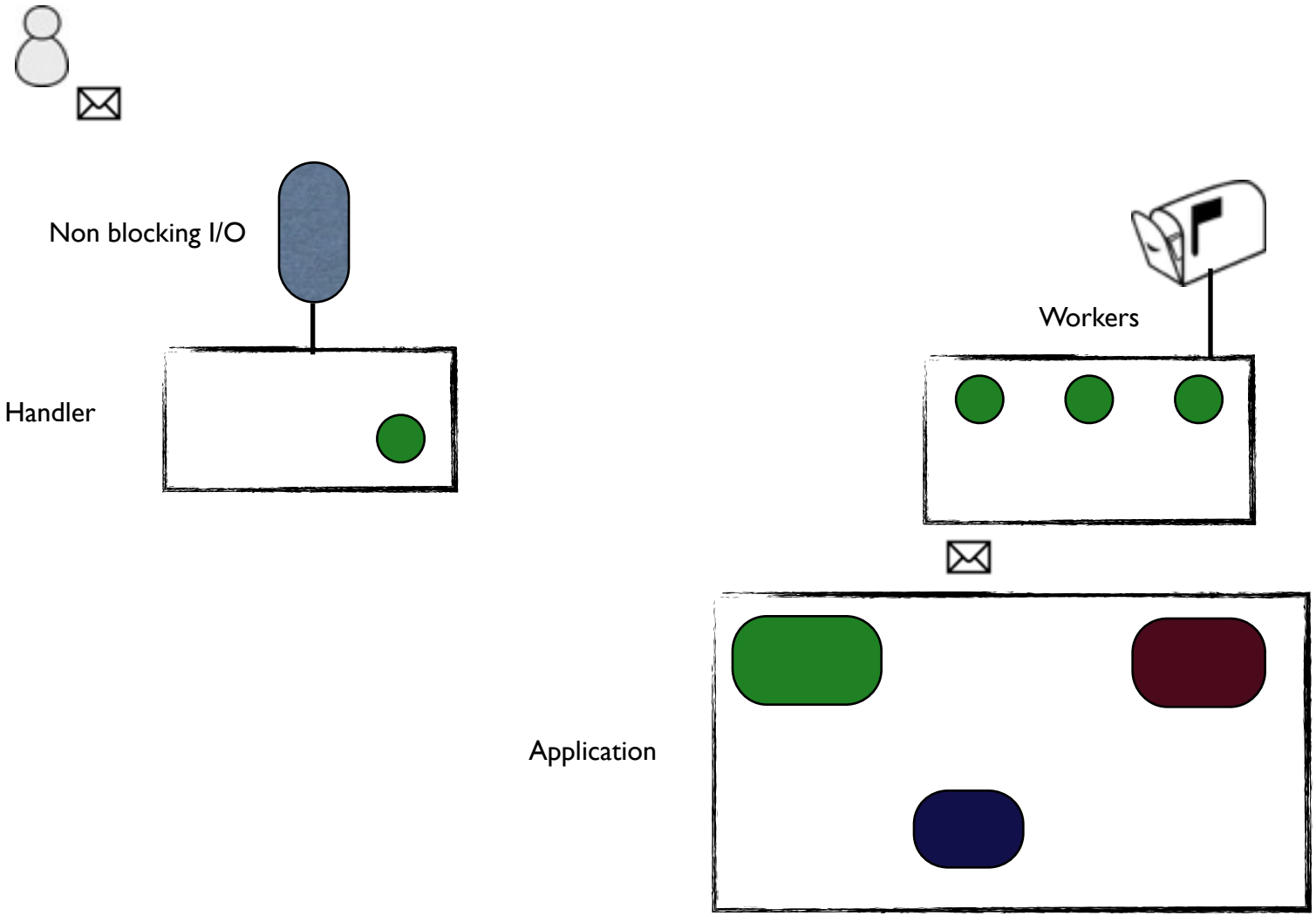


Synchronous Web



Many Data sources…

How async works



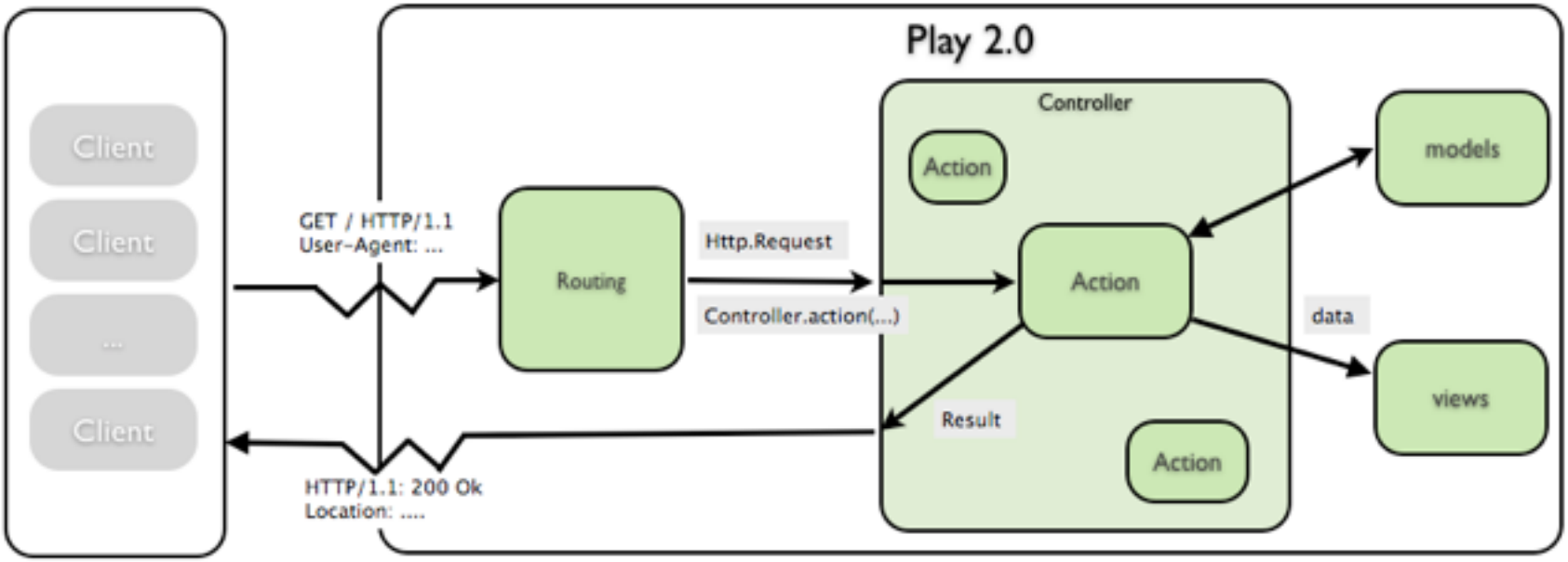
What is Play?

- High velocity Reactive web framework for Java and Scala
- Created by web developers by web developers
- Brings fun back to web development

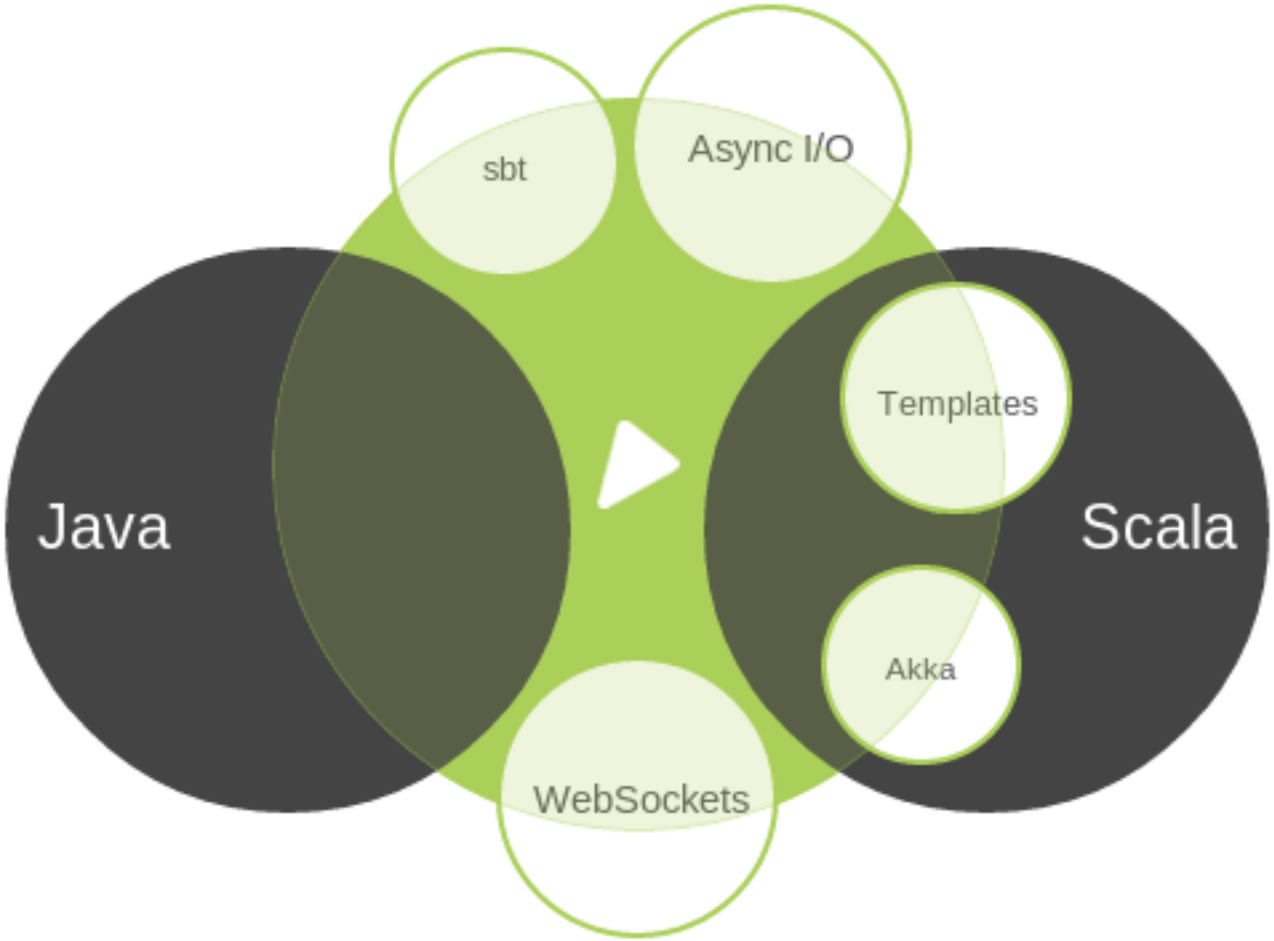
Why Play?

- Play makes building reactive web apps for the JVM simple
- Developer friendly
- Scale predictably
- Reactive web and mobile

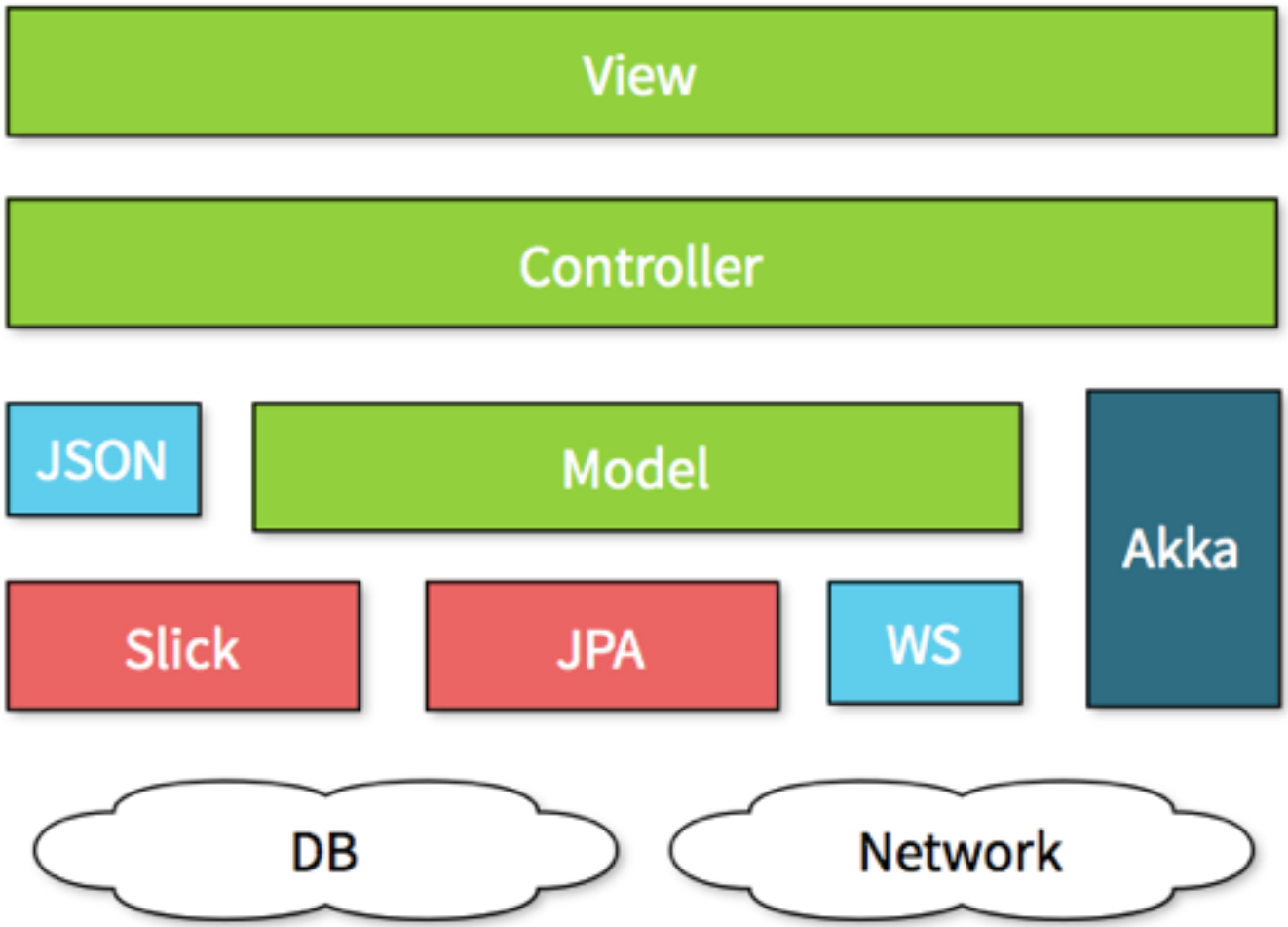
Request Flow



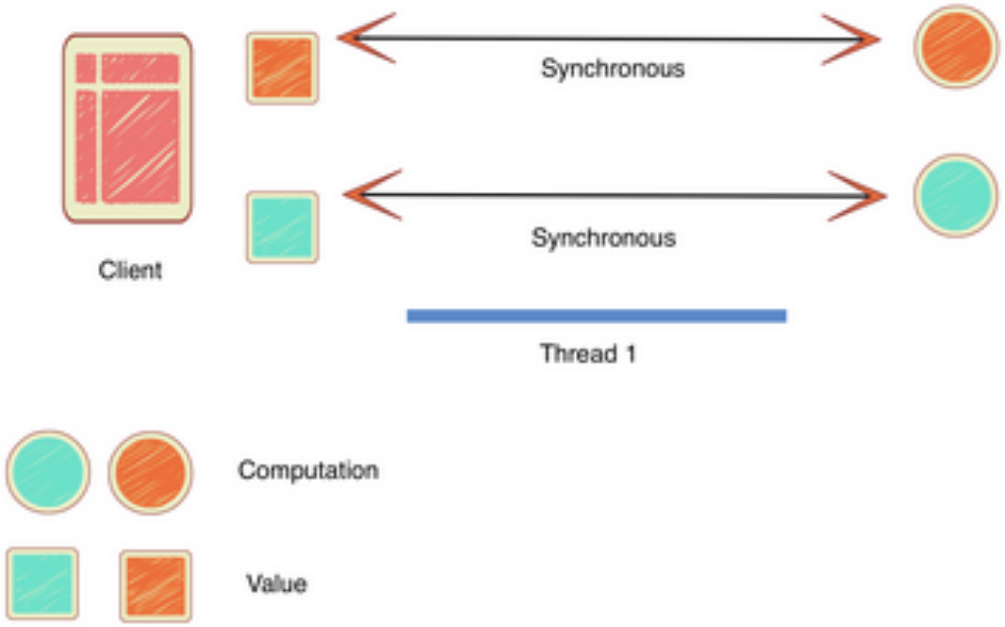
Under the covers



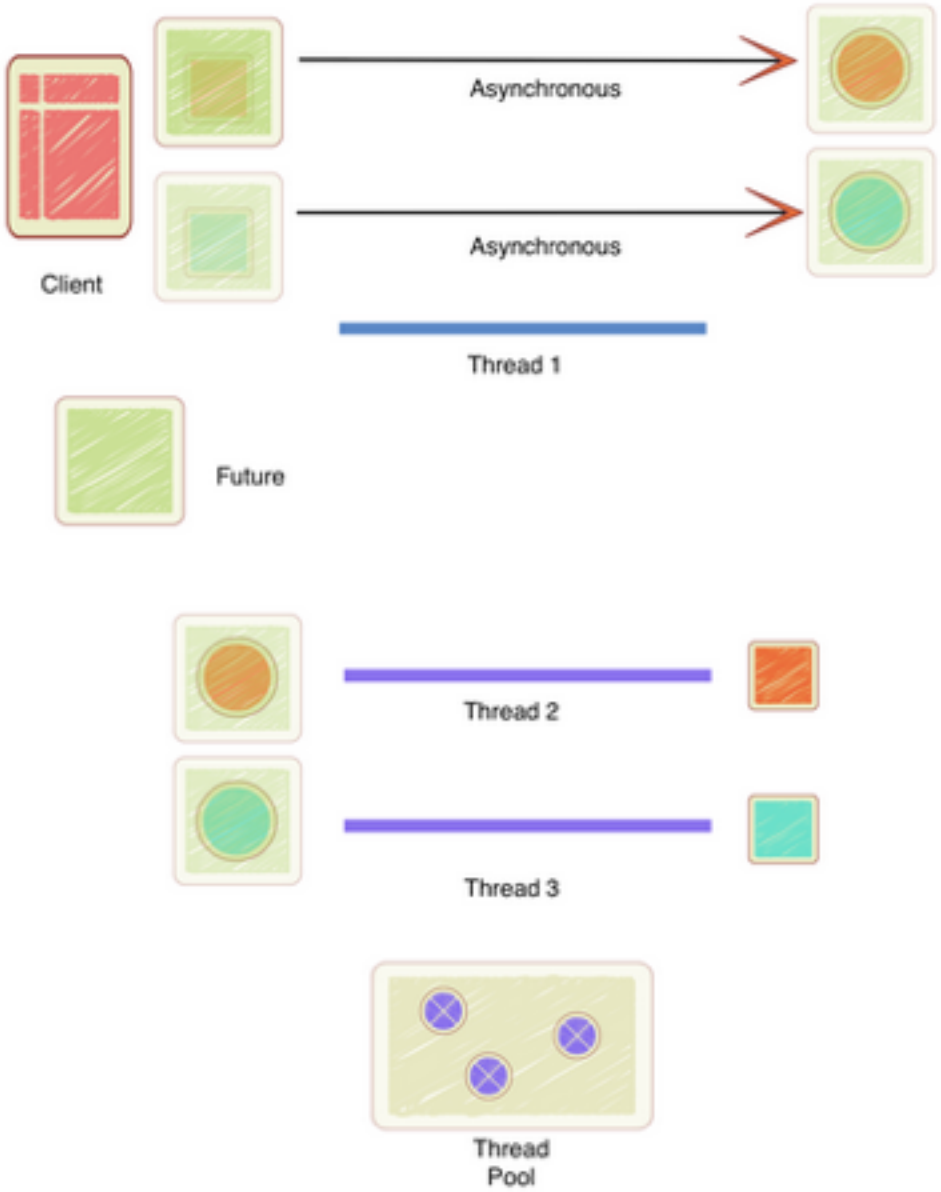
Play application overview



Sequential operations



Using Promise/Future

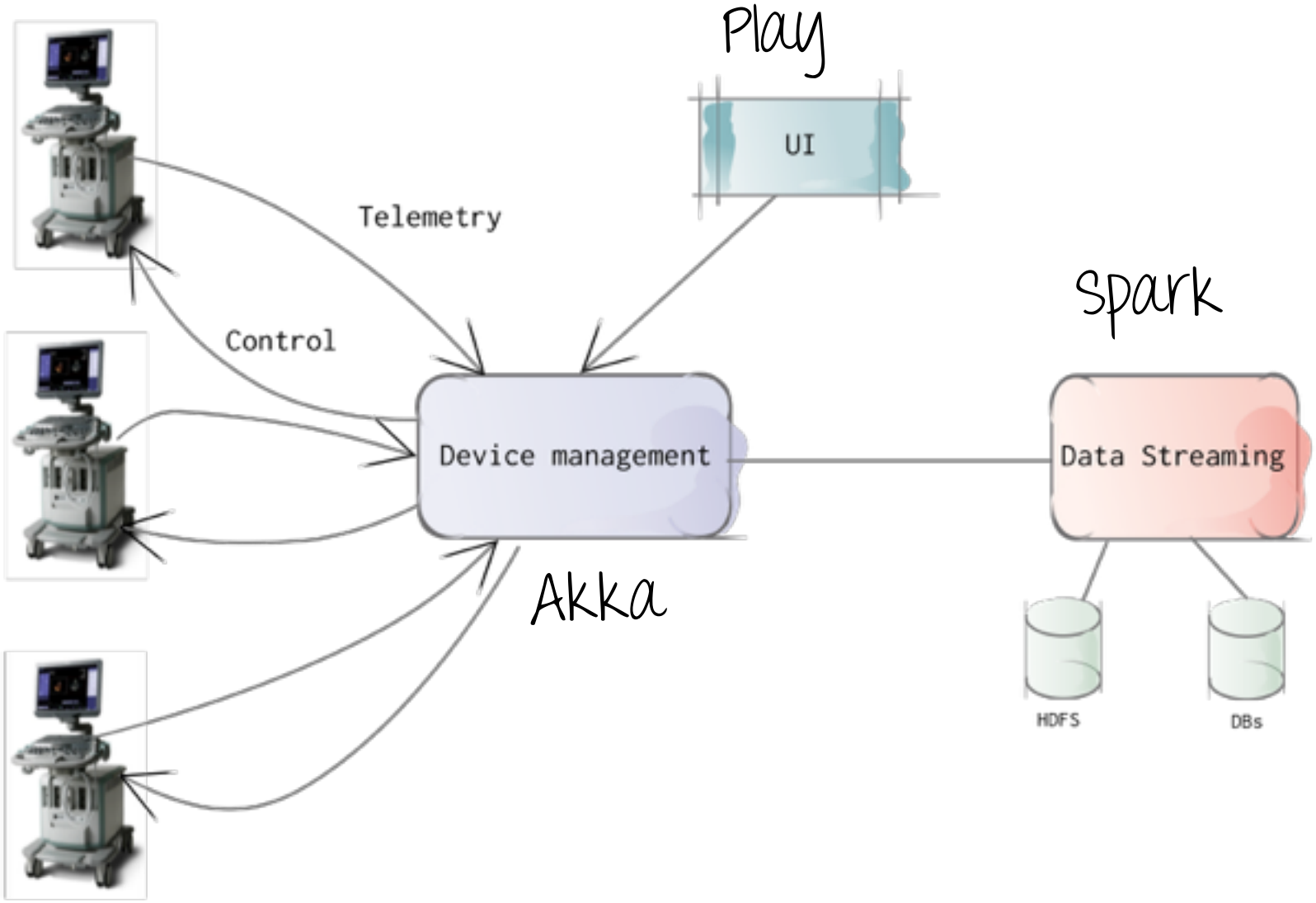


Play code examples

Real world case studies

Precision Farming

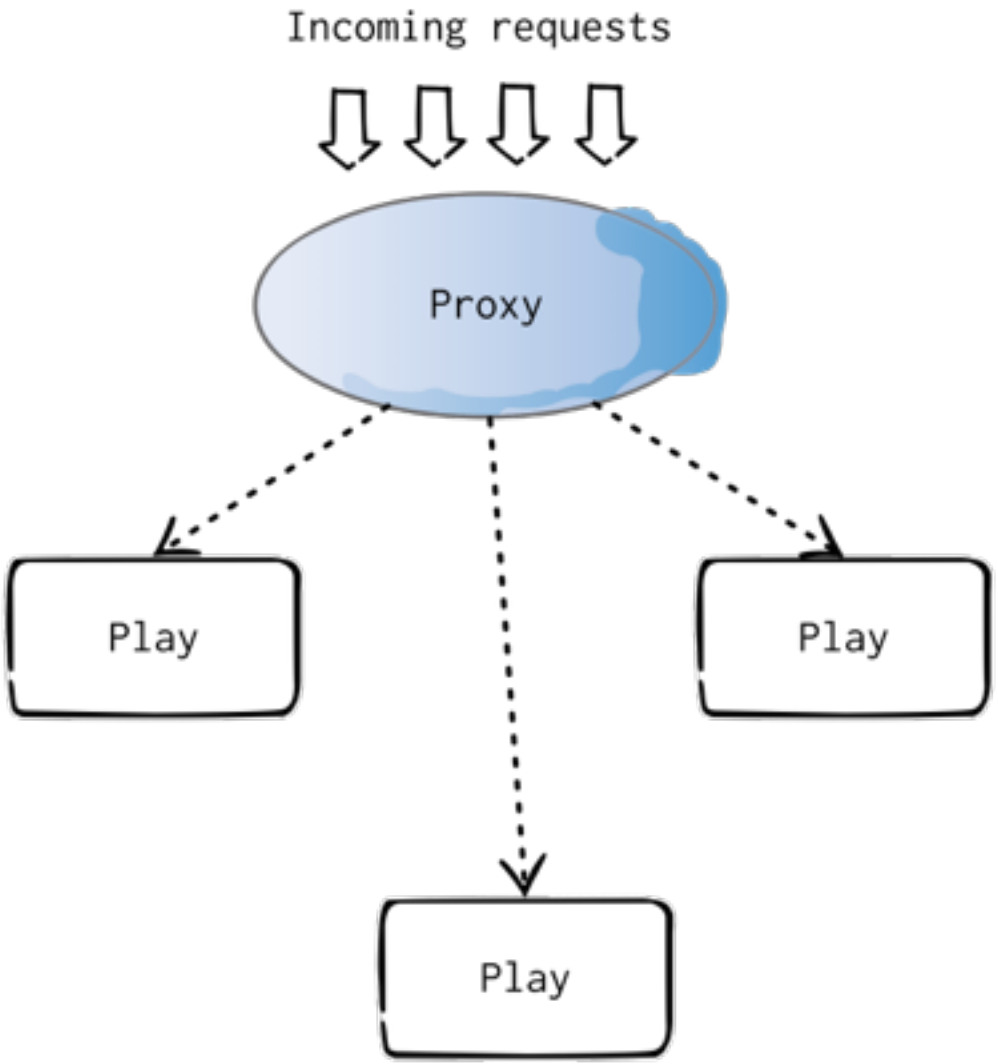
Reactive IoT



Trucks, Farm Equipment

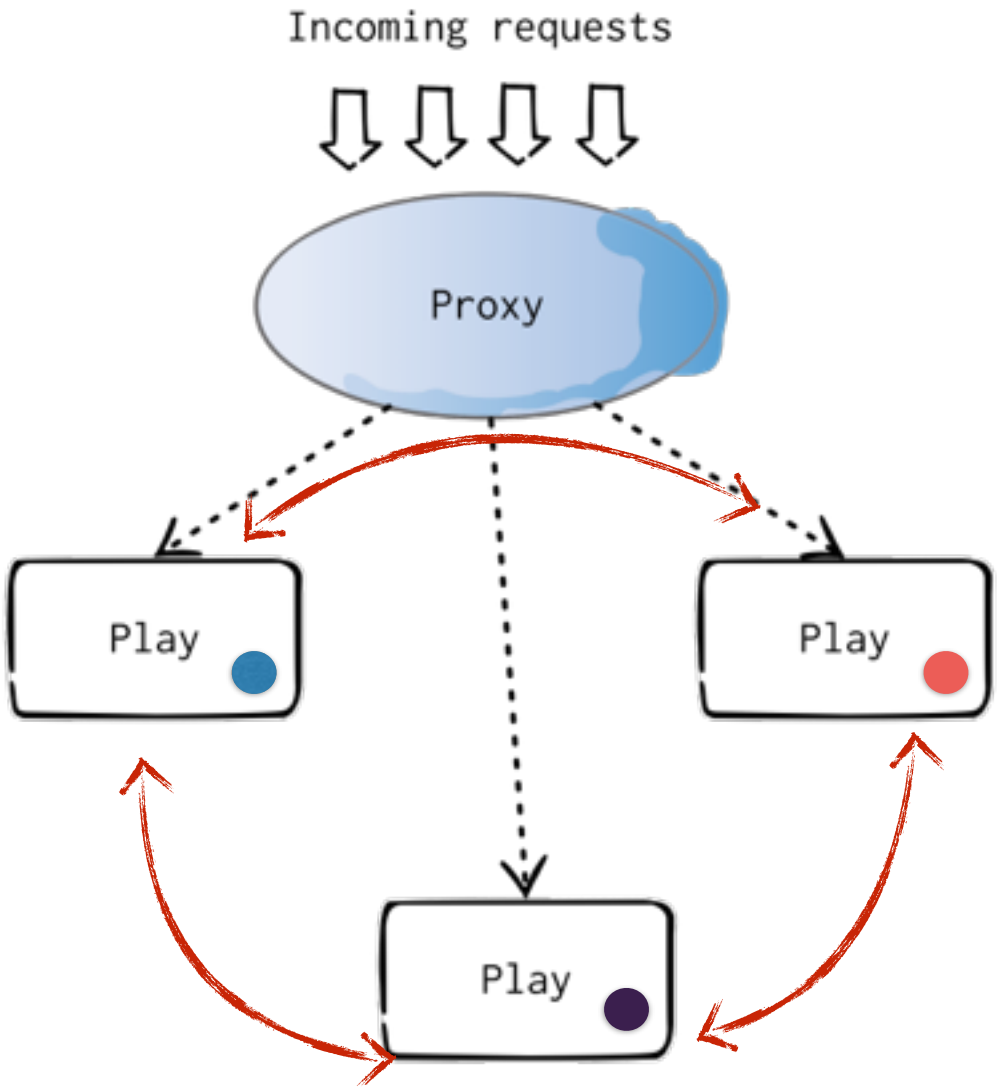
- Phone Home to report movements determined using GPS.
  - Optimize routing.
  - Spy on drivers?
- Occasional network.

Where to put the state?



Challenges

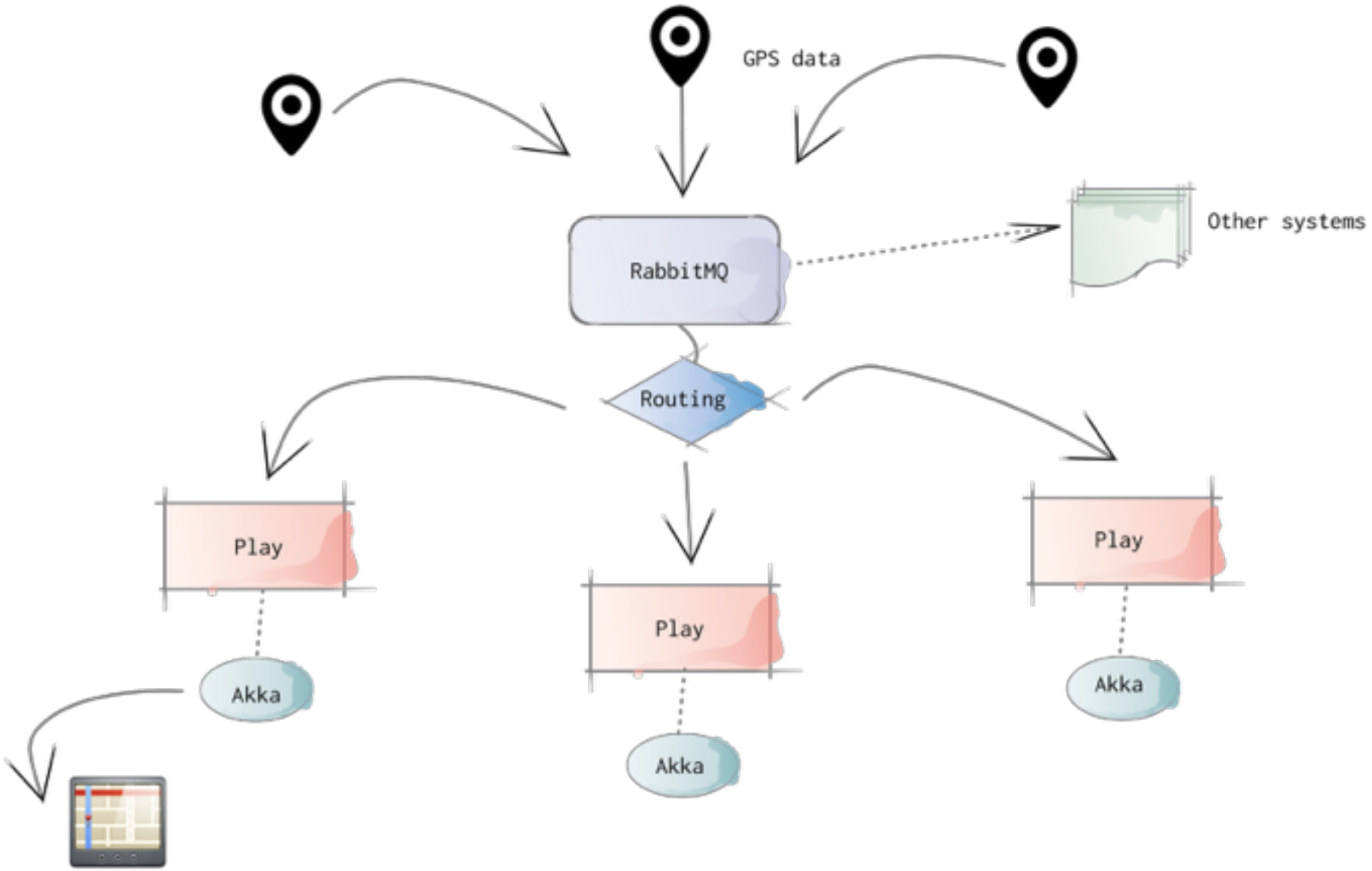
- In memory state
- Location
- Node crashes


Incoming requests
Proxy
Play
Play
Play

Code

Realtime traffic

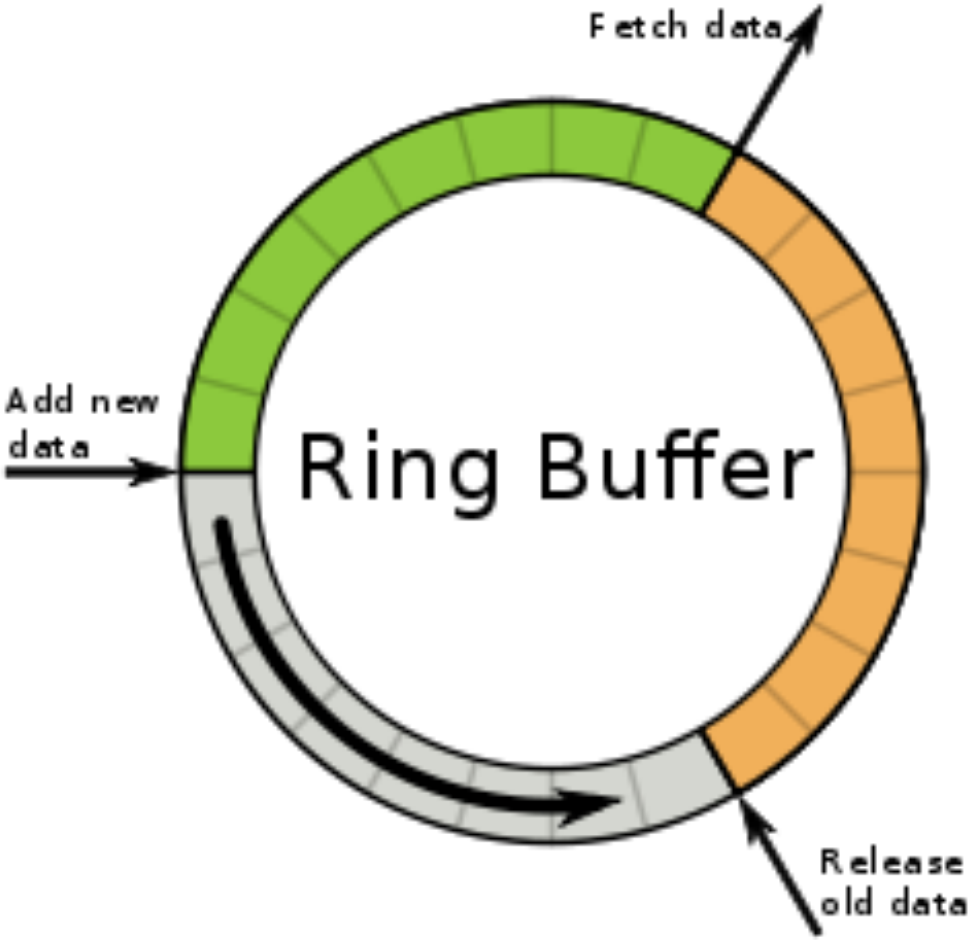
real time traffic



Challenges

- Cannot block
- Burst scenarios

nbbq



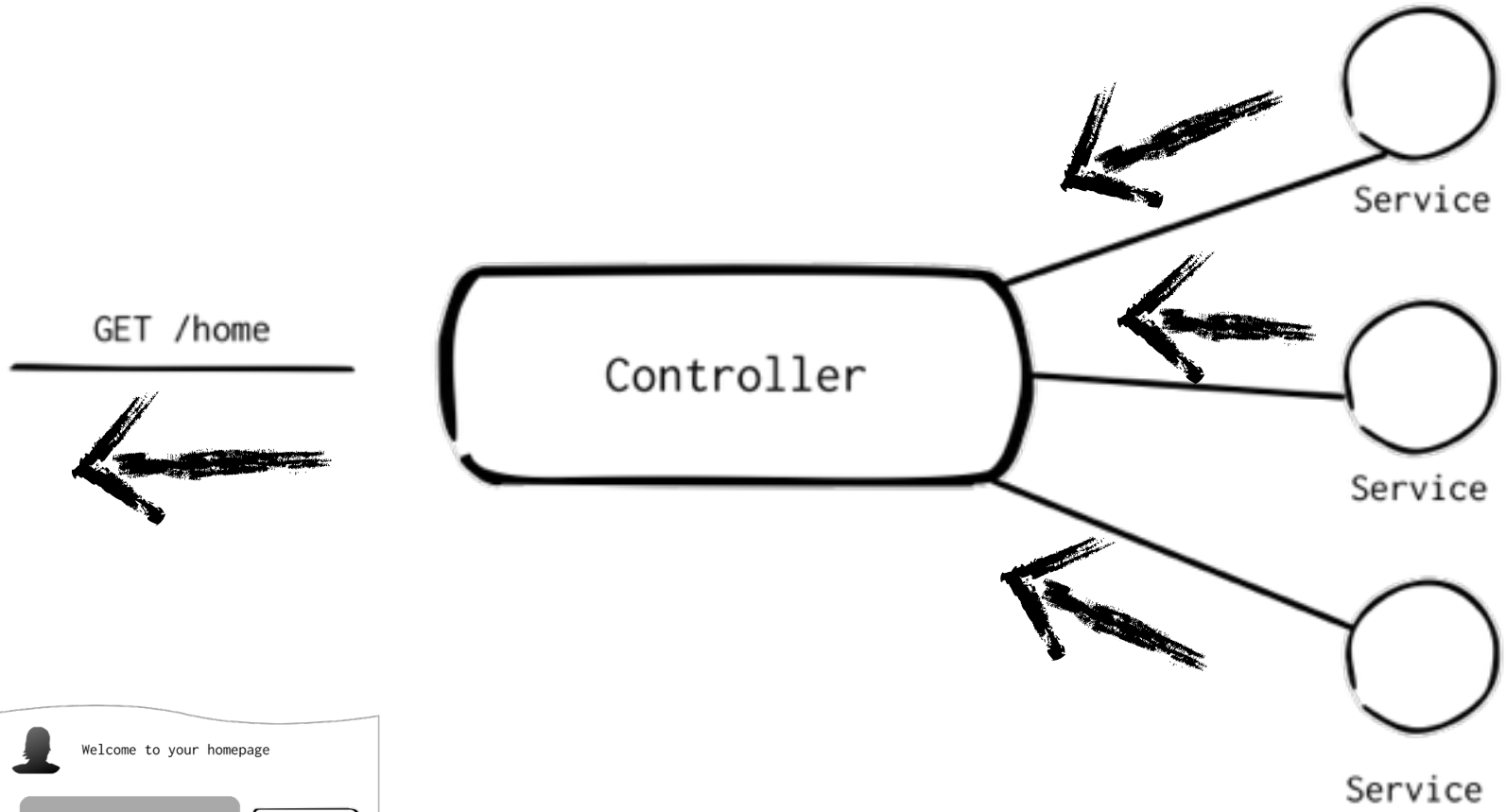
http://en.wikipedia.org/wiki/Circular_buffer#mediaviewer/File:Ring_buffer.svg

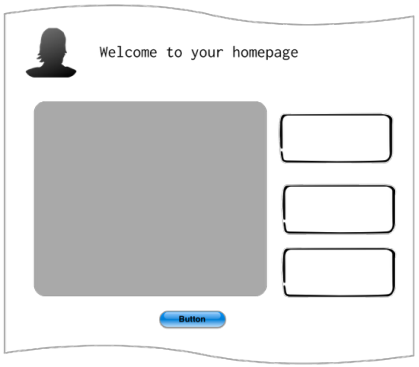
Code

Responsive UI

Welcome to your homepage

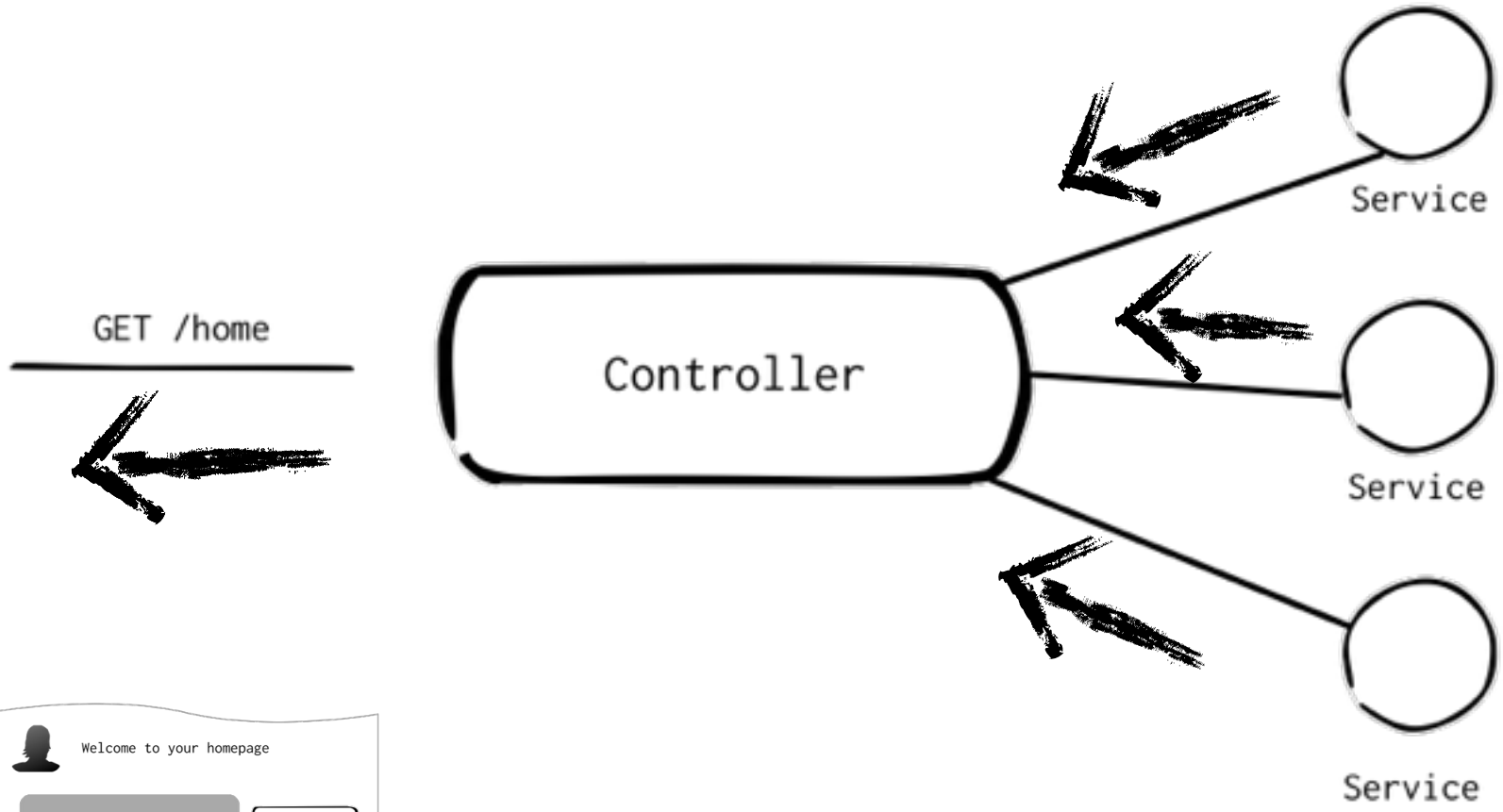
Button

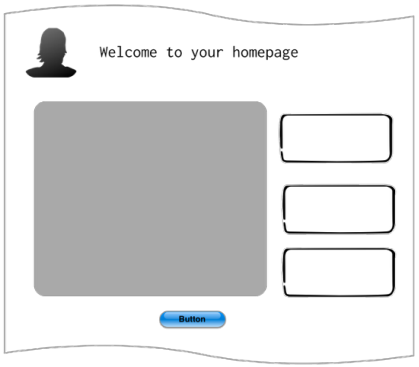

GET /home
Controller
Service
Service
Service


Welcome to your homepage
Button

Challenges

- Reduce time to first byte
- Latency variations
- No Ajax


GET /home
Controller
Service
Service
Service


Welcome to your homepage
Button

Code

Wrap up