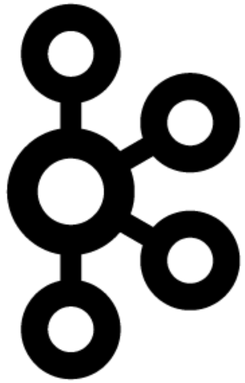
HOW TO FIND THE PERFECT FISHING SPOT IN

CAREFUL MESSAGING WITH PYTHON, KAFKA AND SCALA



kafka

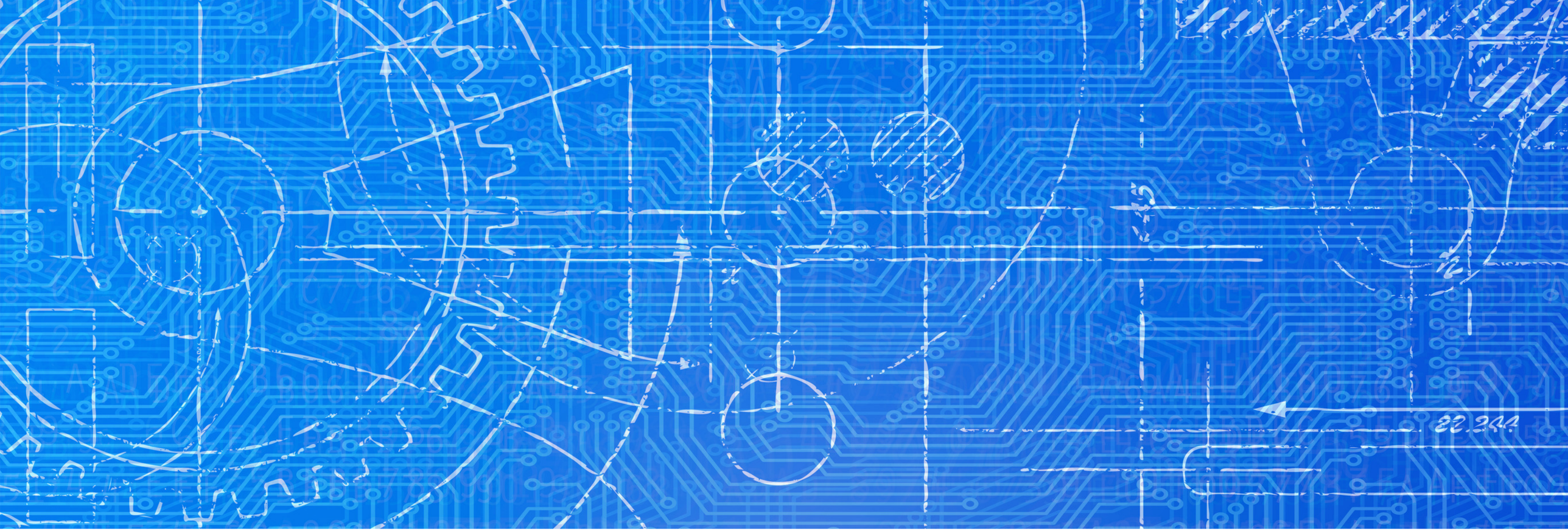
A high-throughput, distributed, publish-subscribe messaging system.

python™

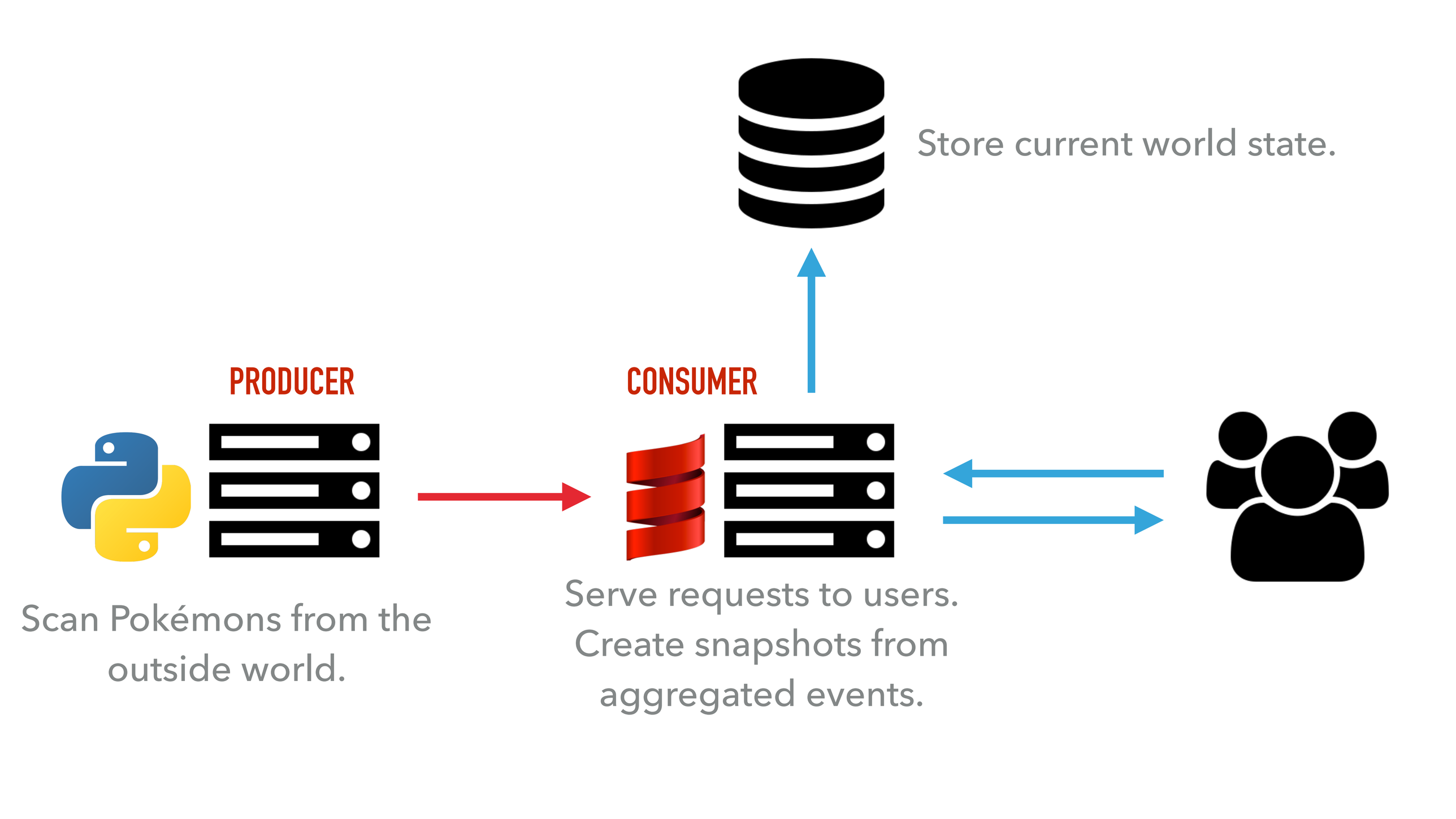
A programing language.

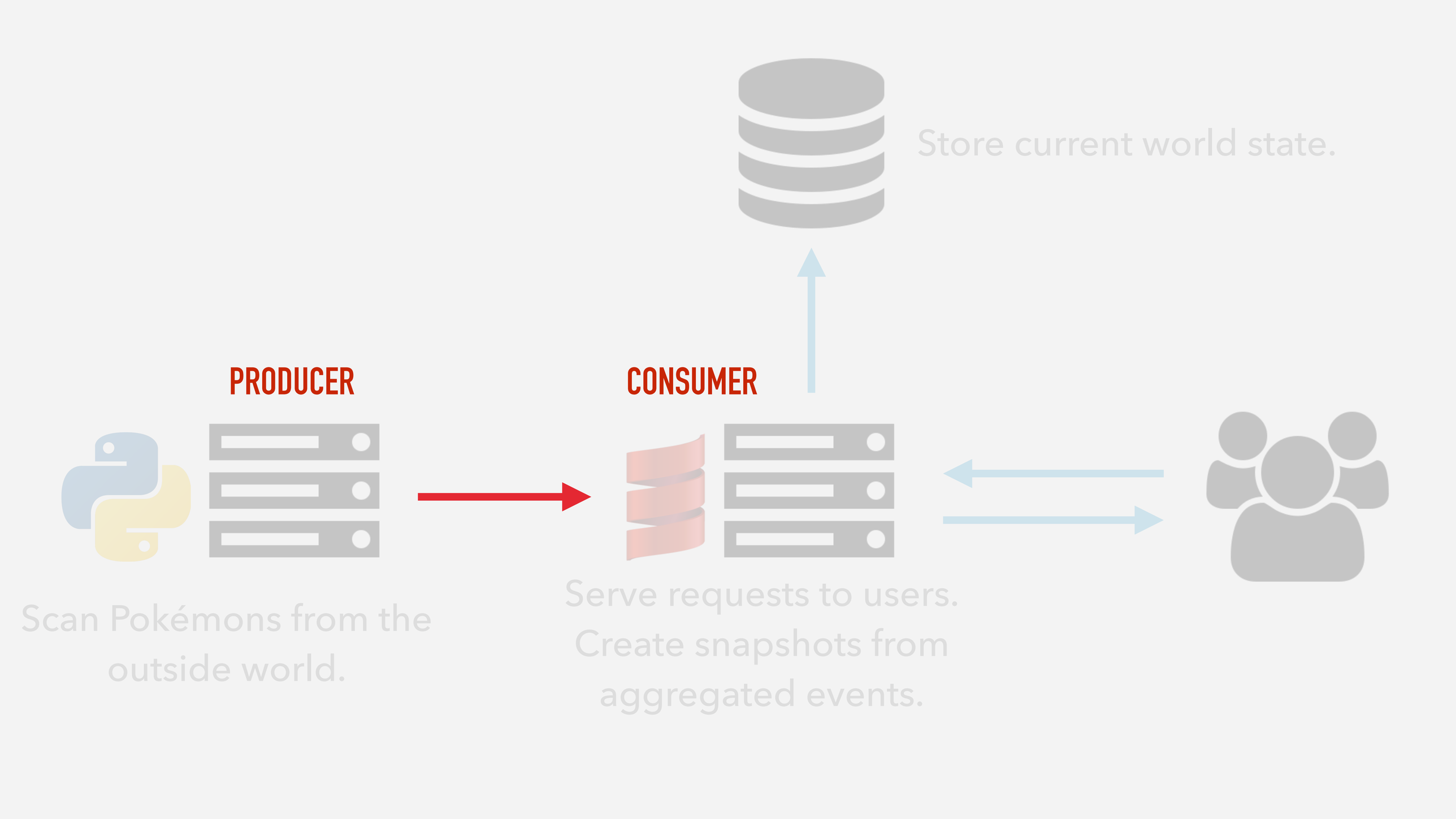
Scala

Another programing language.



DESIGN


Store current world state.
PRODUCER
CONSUMER
Scan Pokémons from the outside world.
Serve requests to users.
Create snapshots from aggregated events.


Store current world state.
PRODUCER
CONSUMER
Scan Pokémons from the outside world.
Serve requests to users.
Create snapshots from aggregated events.

SpawnPoint

id

- Create Pokémons (1-3)
- Destroyed after a random period of time

1

n

Pokemons

id
expireAt
geo (lat, lon)

- Visible on the map
- Can move

INTRODUCTION

DISTRIBUTED STREAMING PLATFORM

Send and receive messages (Message Queue)

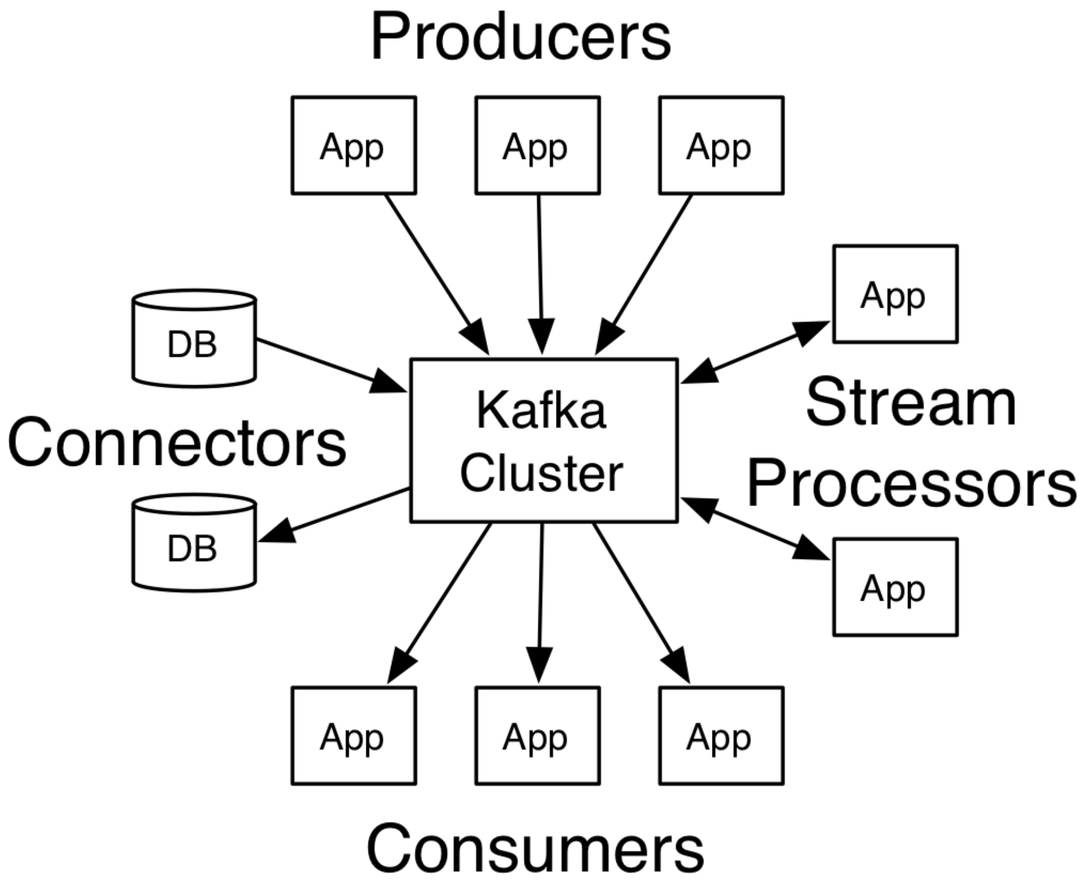
Store and process a stream of records

PRODUCER API

CONSUMER API

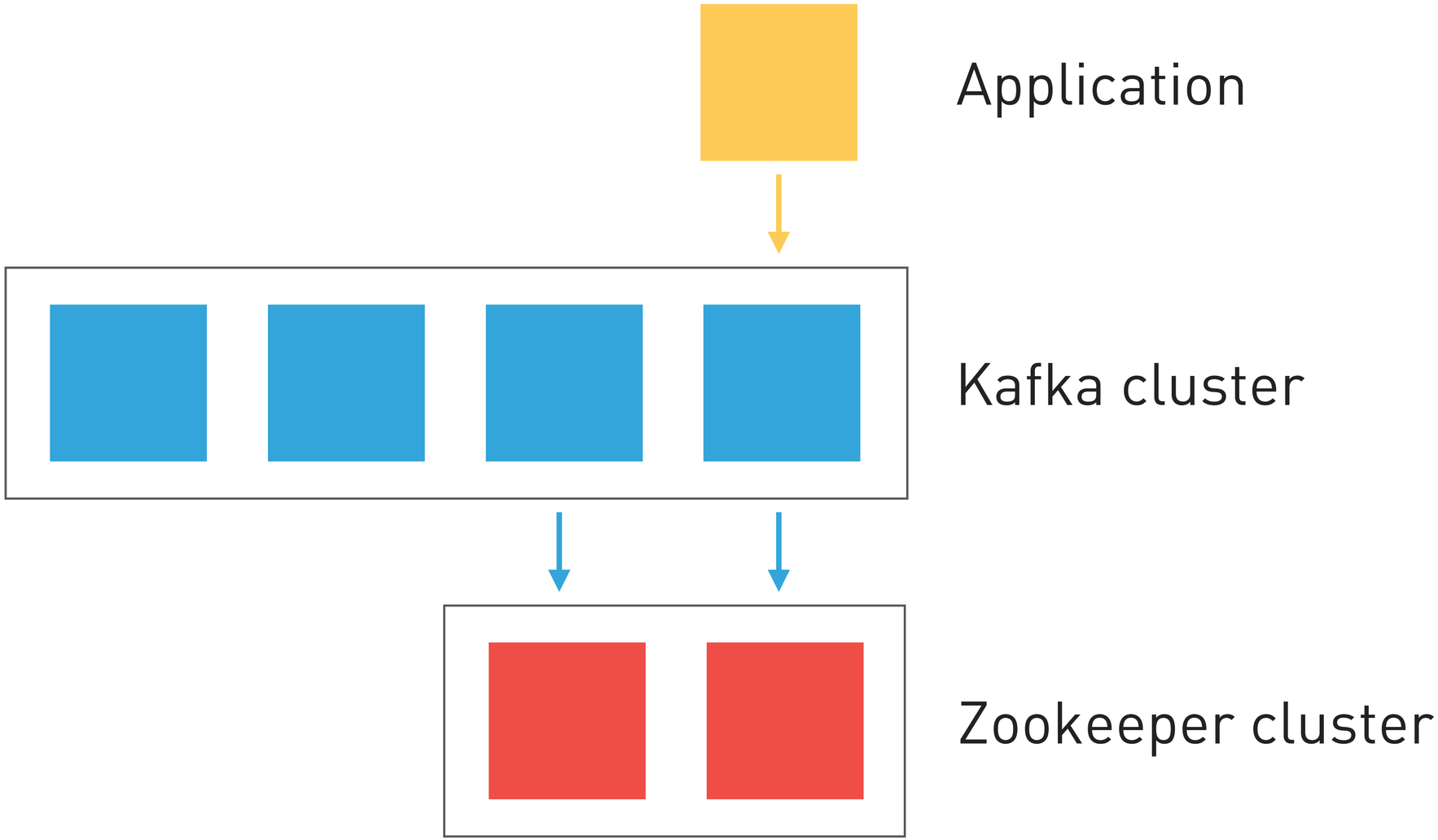
STREAMS API

CONNECTOR API



MESSAGES

KAFKA MESSAGES



ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services.

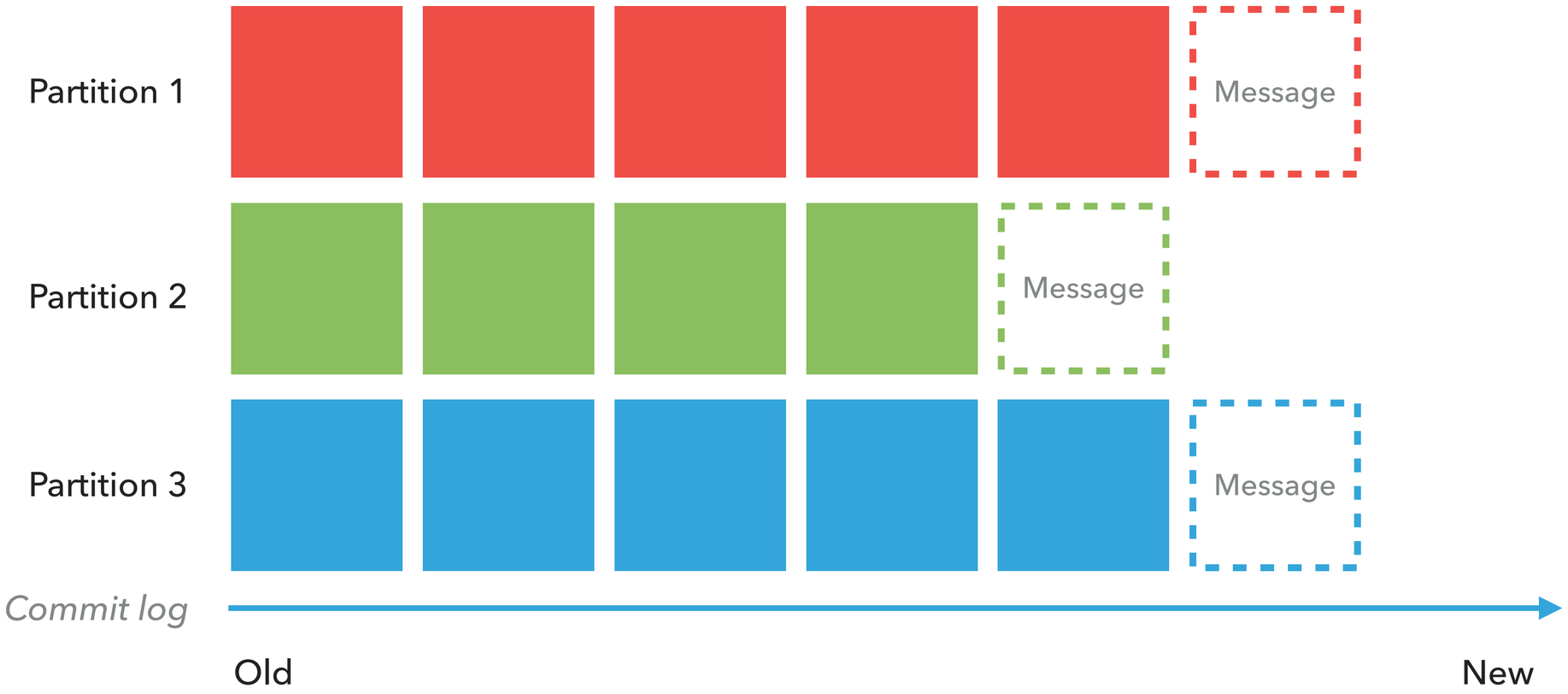
=

etcd

CONSUL

KAFKA MESSAGES

ANATOMY OF A TOPIC



KAFKA MESSAGES

ANATOMY OF A MESSAGE

Optional. Used for partitioning

KEY Message

Custom serializer for BOTH

KAFKA MESSAGES

CREATE {{SpawnPointId}}

DELETE {{SpawnPointId}}

{{SPAWNPOINTID}} {{Pokemon}}


```
{
  "geo": {
    "lat": 48.877449,
    "lon": 2.264464
  },
  "id": 75,
  "expireAt": "2016-09-15T00:08:12"
}
```

KAFKA MESSAGES

spawnpoints Topic

CREATE {{SpawnPointId}}

spawnpoints Topic

DELETE {{SpawnPointId}}

{{SPAWNPOINTID}} {{Pokemon}}

pokemons Topic


```
{
  "geo": {
    "lat": 48.877449,
    "lon": 2.264464
  },
  "id": 75,
  "expireAt": "2016-09-15T00:08:12"
}
```

PRODUCER

PYTHON PRODUCER

```
while True:
    pokemons = scan()
    for pokemon in pokemons:
        kafka.send('pokemons', key=pokemon.spawn_id, value=pokemon)
    kafka.flush()
    time.sleep(CYCLE_SPEED)
```

THE ONLY PYTHON SNIPPET IN THESE SLIDES

CONSUMER

SCALA CONSUMER

```
def start = {
  // Starting our consumer
  val consumer = Consumer.create(config)
```

SCALA CONSUMER

```
def start = {
  // Starting our consumer
  val consumer = Consumer.create(config)

  // Topics -> Streams count
  val topics = Map(
    "pokemons" -> 1,
    "spawnpoints" -> 1
  )
```

SCALA CONSUMER

```
def start = {
  // Starting our consumer
  val consumer = Consumer.create(config)

  // Topics -> Streams count
  val topics = Map(
    "pokemons" -> 1,
    "spawnpoints" -> 1
  )

  val streams = consumer.createMessageStreams(topics)
```

SCALA CONSUMER

```
def start = {
  // Starting our consumer
  val consumer = Consumer.create(config)

  // Topics -> Streams count
  val topics = Map(
    "pokemons" -> 1,
    "spawnpoints" -> 1
  )

  val streams = consumer.createMessageStreams(topics)

  // Start the consumer asynchronously
  Future {
    streams.get("pokemons").get.foreach(PokemonService.cycle(system))
  }
  Future {
    streams.get("spawnpoints").get.foreach(SpawnService.cycle(system))
  }
}
```

"FUTURES"?

FROM THE OFFICIAL DOCUMENTATION :

A Future is a placeholder object for a value
that may not yet exist.

WHICH REALLY MEANS :

Stuff that run magically in parallel.

OR

Non-blocking block of code that will return something later (maybe).

NOTE :

If you try to access a Future's value before it's actually ready, it'll block.

```
Future {
    1 + 1 // return 2
}
```

```
val v = Future {
    while (true) {}
}

v.get // uh oh, bloqué
```

Thread 2

Thread 1

NOTE :

Execution Context

Threadpool

Promise

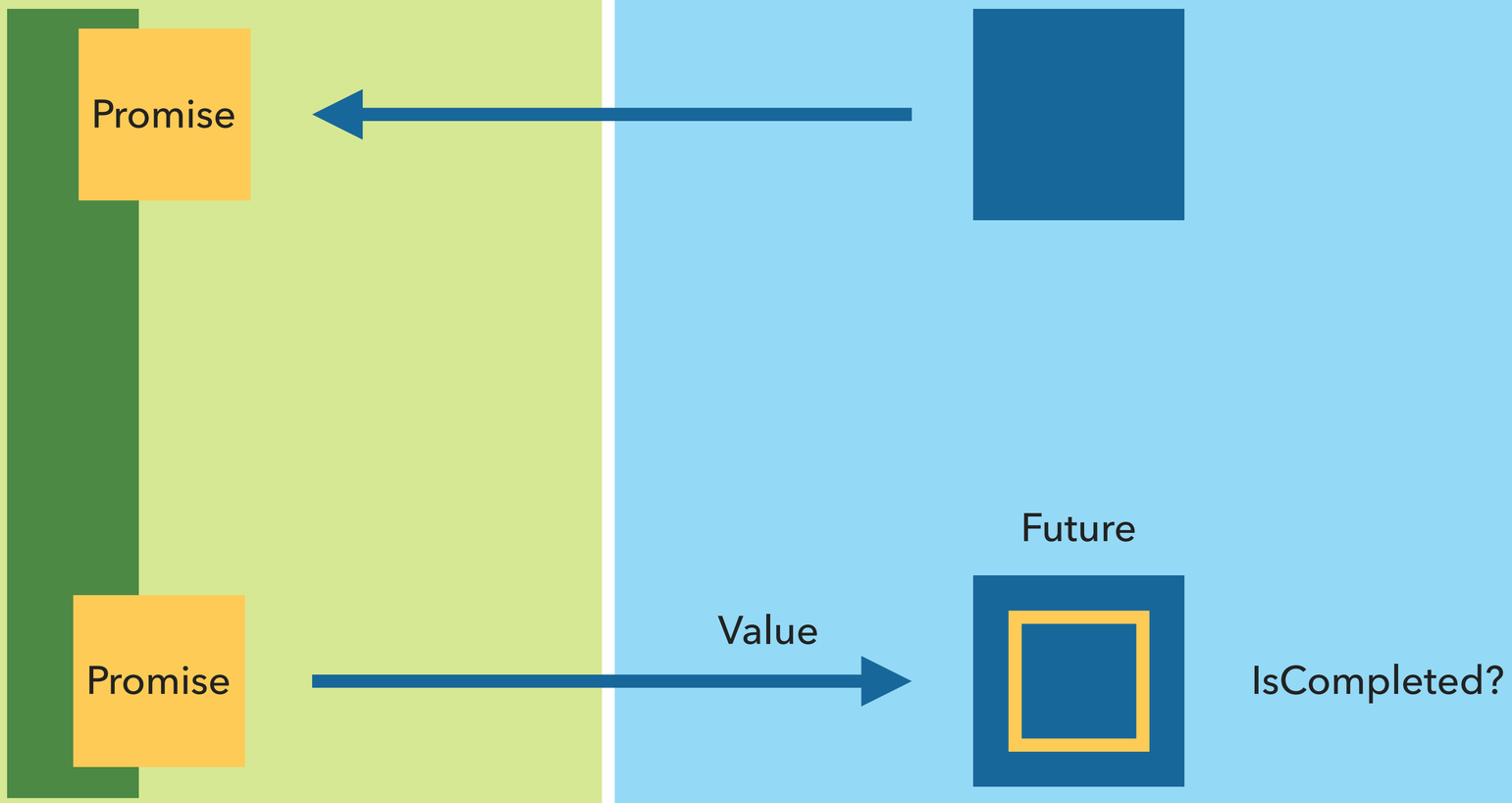
Promise

Future

Future

Value

IsCompleted?



SCALA CONSUMER

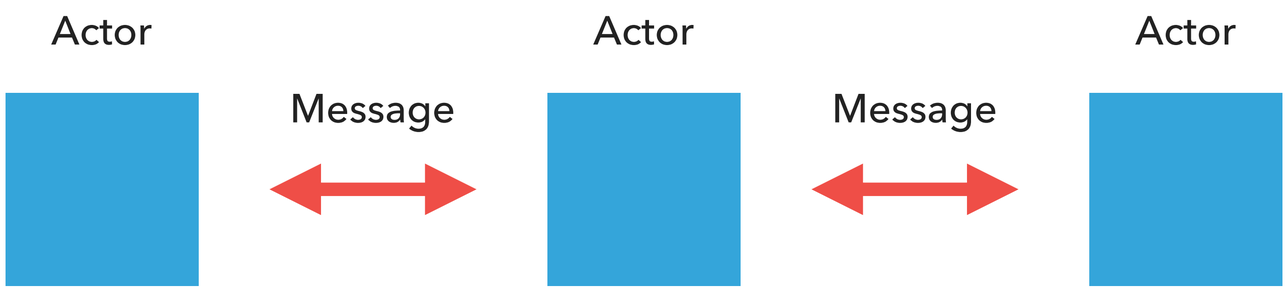
```
key match {
  case "create" => system.actorOf(Props[IndexService], msg)
  case "delete" => sendPoisonPill(msg)
  case _        => println(s"Unknown operation: '$key'")
}
```

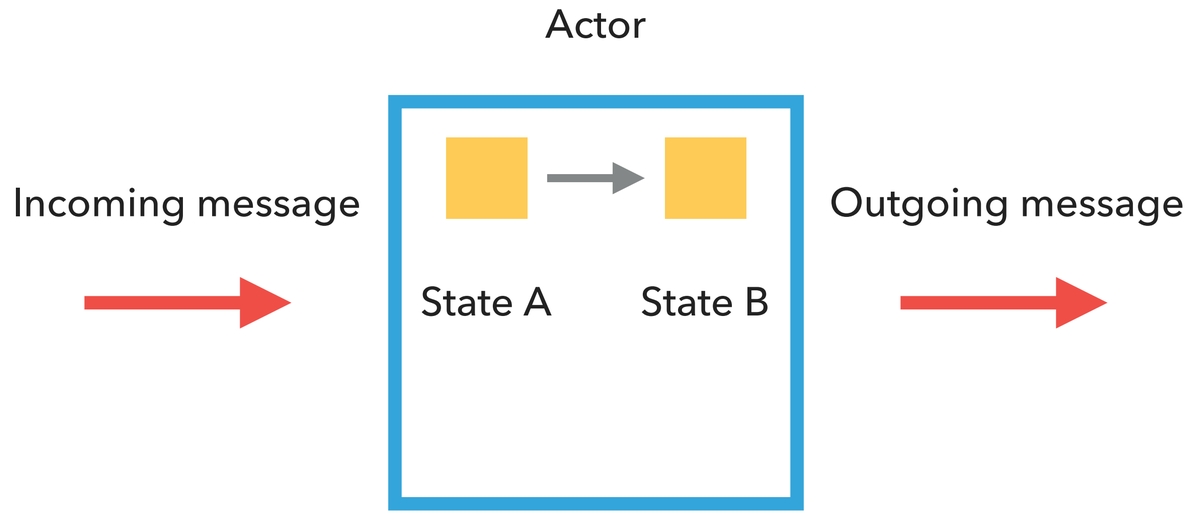
SCALA ♥ PATTERN MATCHING

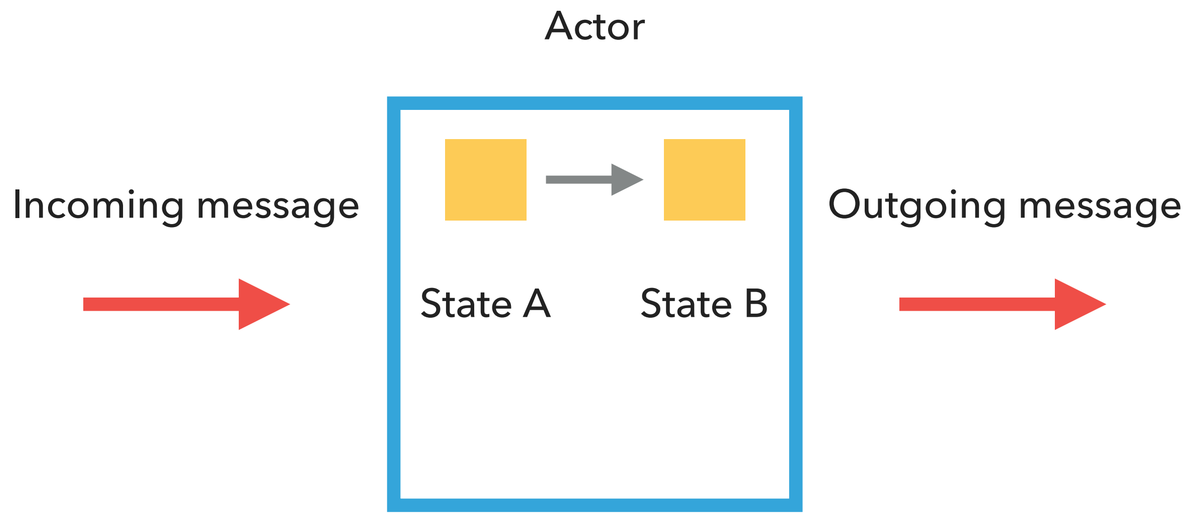
"ACTORS"?

FROM WIKIPEDIA :

The actor model in computer science is a mathematical model of concurrent computation that treats "actors" as an universal primitives.


Actor
Actor
Actor
Message
Message

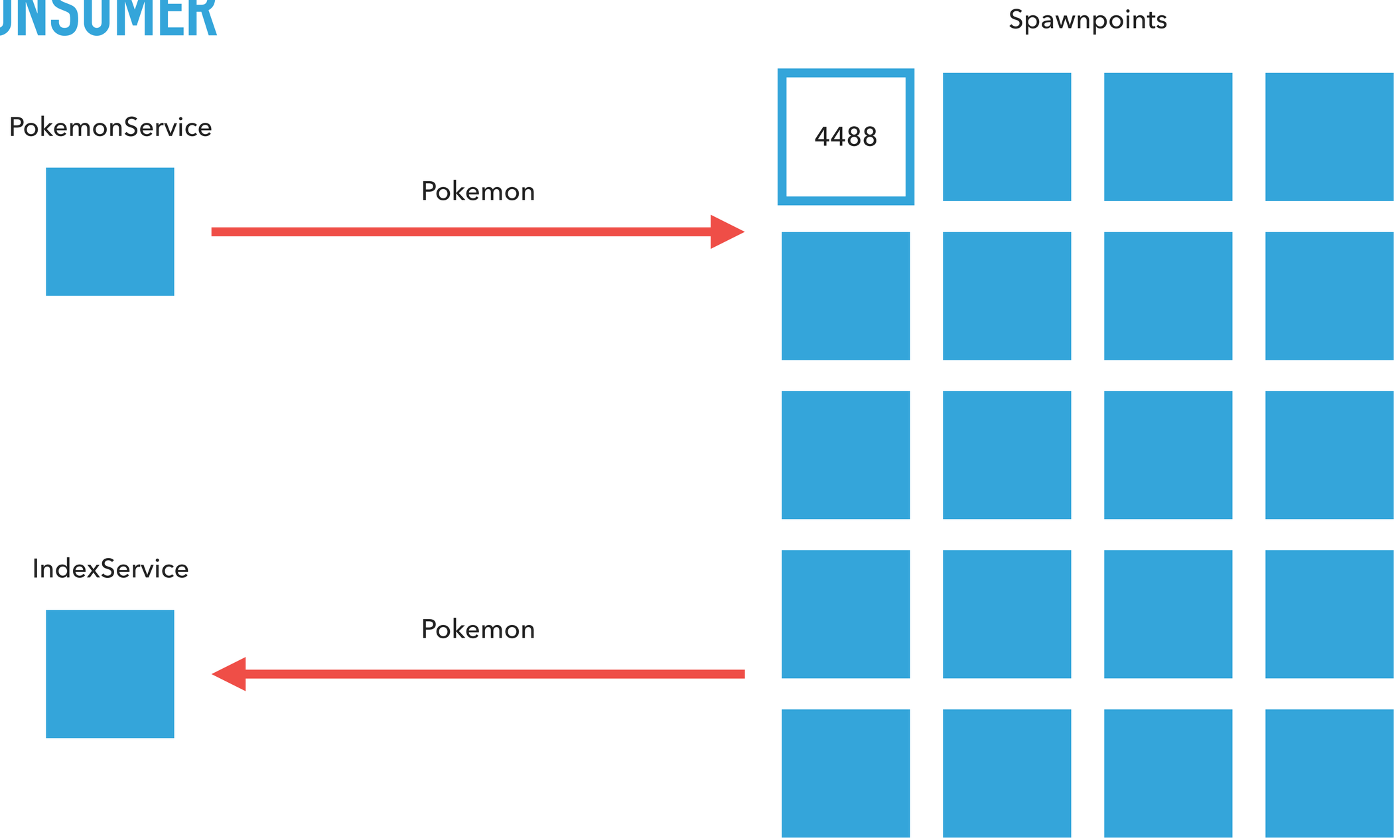

Actor
Incoming message
Outgoing message
State A
State B



WHAT IF WE STORED STATE TRANSITIONS FOR FUTURE REPLAY ?

Sure. It’s called Event Sourcing.

SCALA CONSUMER



SCALA CONSUMER

```
def indexWithTTL(pokemon: Pokemon) = {
  // Index pokemon into ES and schedule another message containing the ID
  // for a deferred removal
  IndexInto.pokemon(pokemon).map { id =>
    val duration = DateTime.now to pokemon.expireAt

    system.scheduler.scheduleOnce(duration) {
      self ! id
    }          <– This (bang) is a Scala function sending a message (id) to an actor (self).
  }
}
```

SCALA ACTORS SUPPORTS DEFERRED AND INTERVAL SCHEDULING

WE LOVE FUNCTIONS WITH MEANINGFUL NAMES.

```
:: -> |@| <<< ++= ??? \/ ∧ _* /: >| (>_>) (_+_)
```

WE LOVE FUNCTIONS WITH MEANINGFUL NAMES.

`(_+_)` → Anonymous function with two parameters

`(>_>)` → A smiley

DEMO

Should work.
Don’t leave now.

QUESTIONS?

GITHUB.COM/FIAHIL/TALKS