Akka Typed

Next generation message driven systems with Akka

Johan Andrén, Akka Team
Devdays Vilnius, 2018-05-23

Lightbend

Johan Andrén

Akka Team
Stockholm Scala User Group

@apnylle
markatta.com/codemonkey/
johan.andren@lightbend.com

Akka

Build powerful reactive, concurrent, and distributed applications more easily

credit karma

UPSIDE

amazon.com

zalando

weightwatchers

What are the tools?

Actors – simple & high performance concurrency

Cluster, Cluster tools – tools for building distributed systems

Streams – reactive streams implementation

Persistence – CQRS + Event Sourcing for Actors

HTTP – fully async streaming HTTP Server

Alpakka – Reactive Streams Integrations a'la Camel

Complete Java & Scala APIs for all features

What is coming soon?

Typed Actors – simple & high performance concurrency

Typed Cluster, Cluster tools – tools for building distributed systems

(already type-safe) **Streams** – reactive streams implementation

Typed Persistence – CQRS + Event Sourcing for Actors

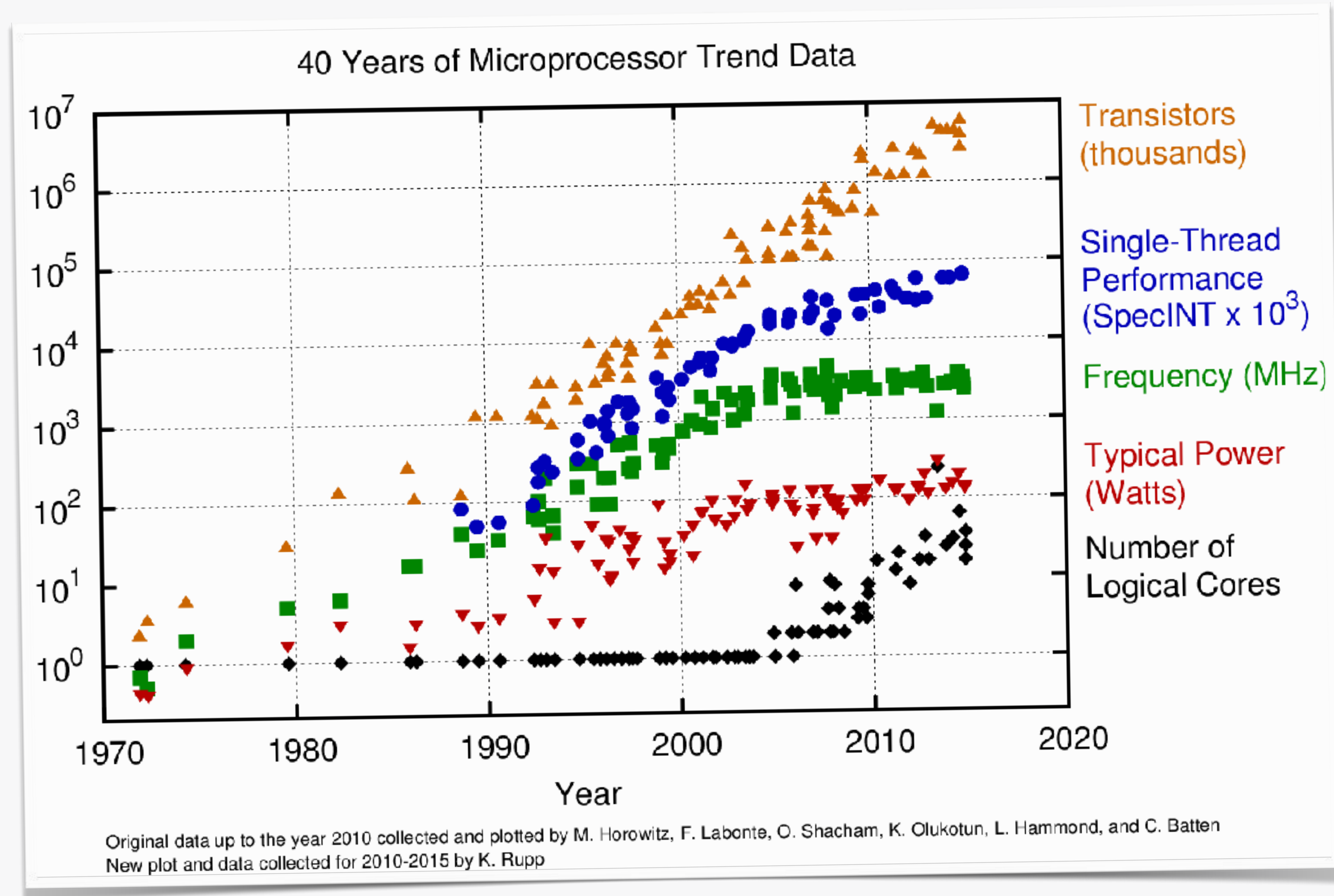
(already type-safe) **HTTP** – fully async streaming HTTP Server

(already type-safe) **Alpakka** – Reactive Streams Integrations a'la Camel

Complete Java & Scala APIs for all features.

All modules ready for preview, final polish during Q2 2018

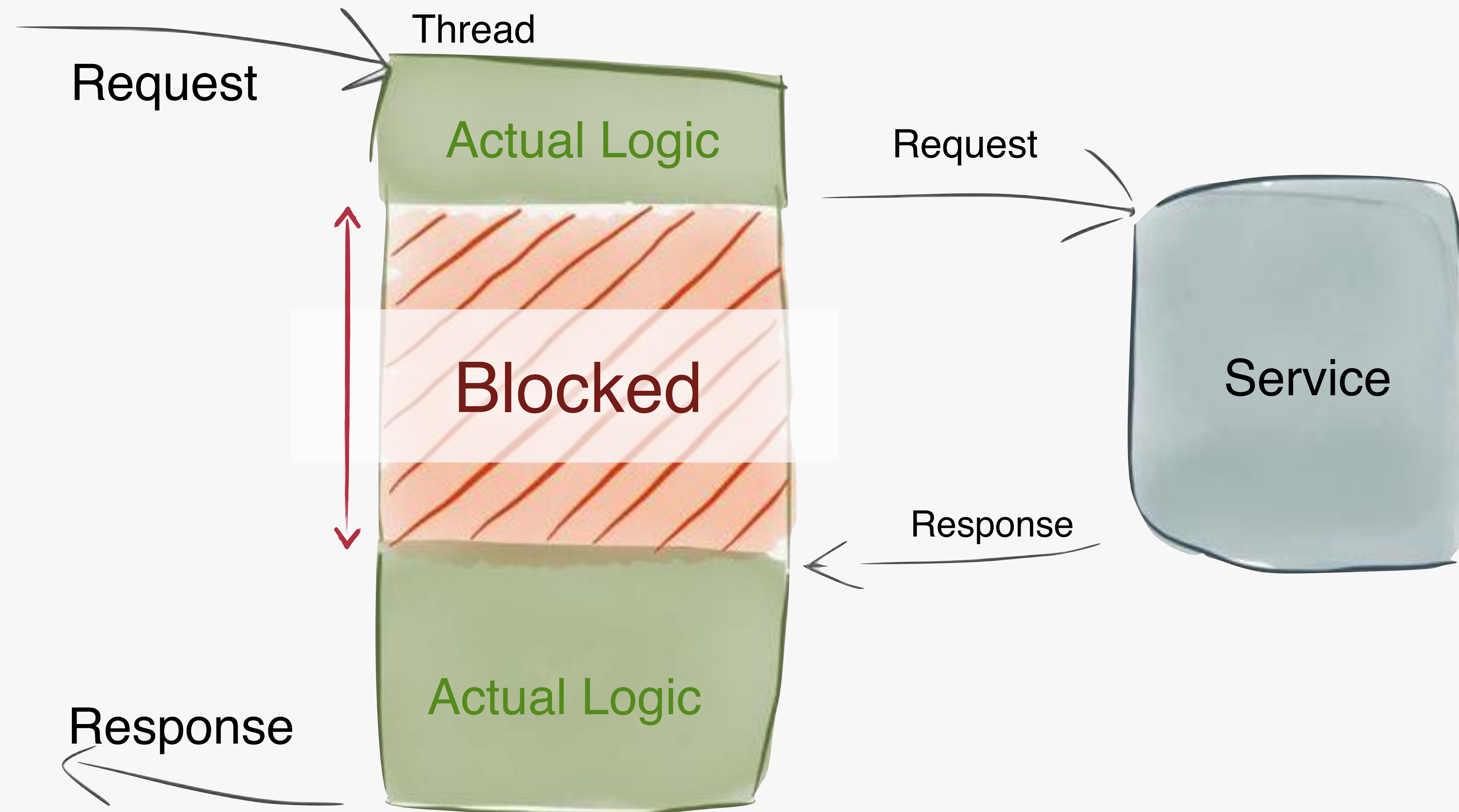
What problem are we solving?



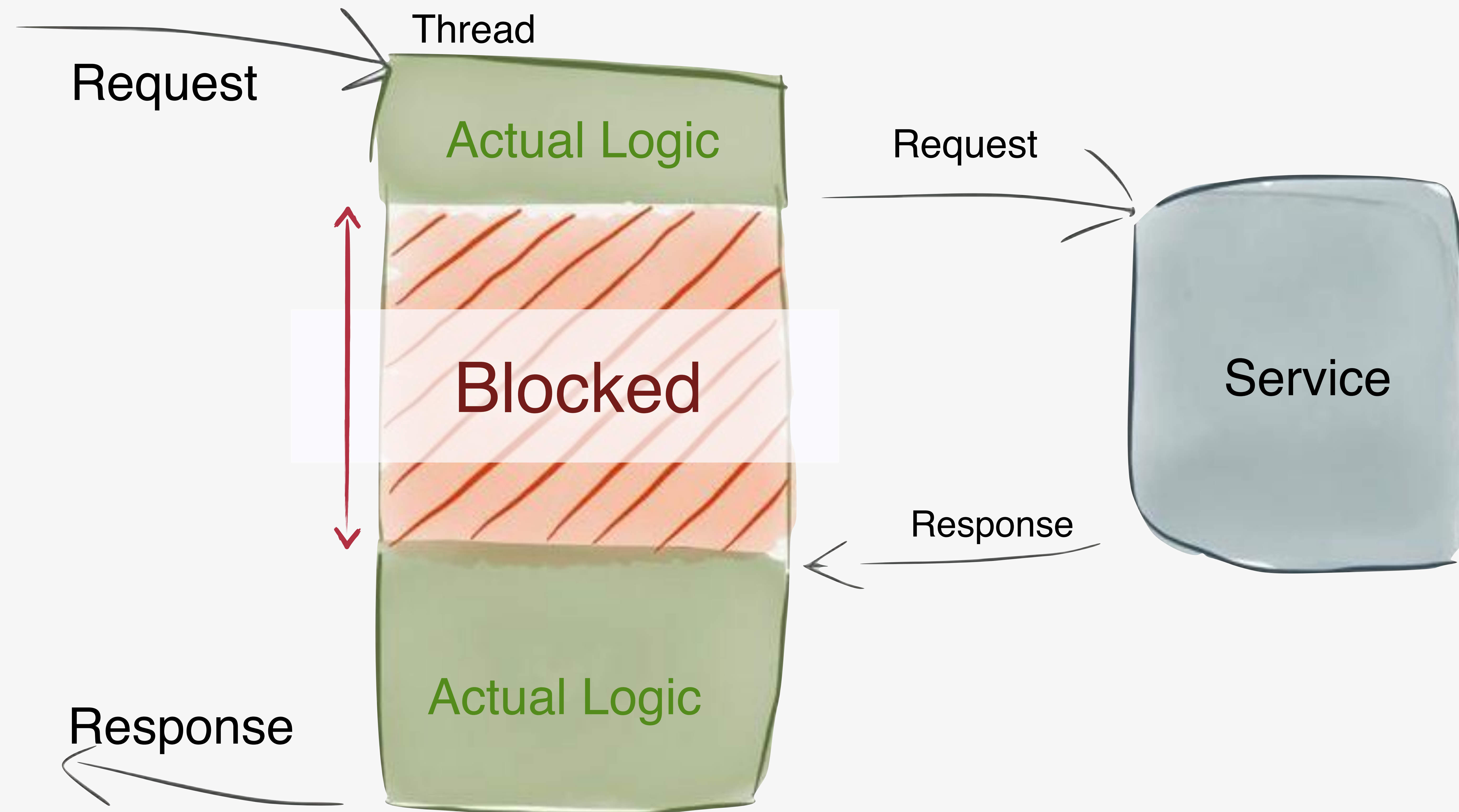
Problems with threads

- Mutable state has to be protected everywhere it is touched
- Atomicity isn’t always what it seems (eg. `counter += 1`)
- Deadlocks
- At scale
- How to know it works
- Cost (stack size, context switches, CPU cache flushes)

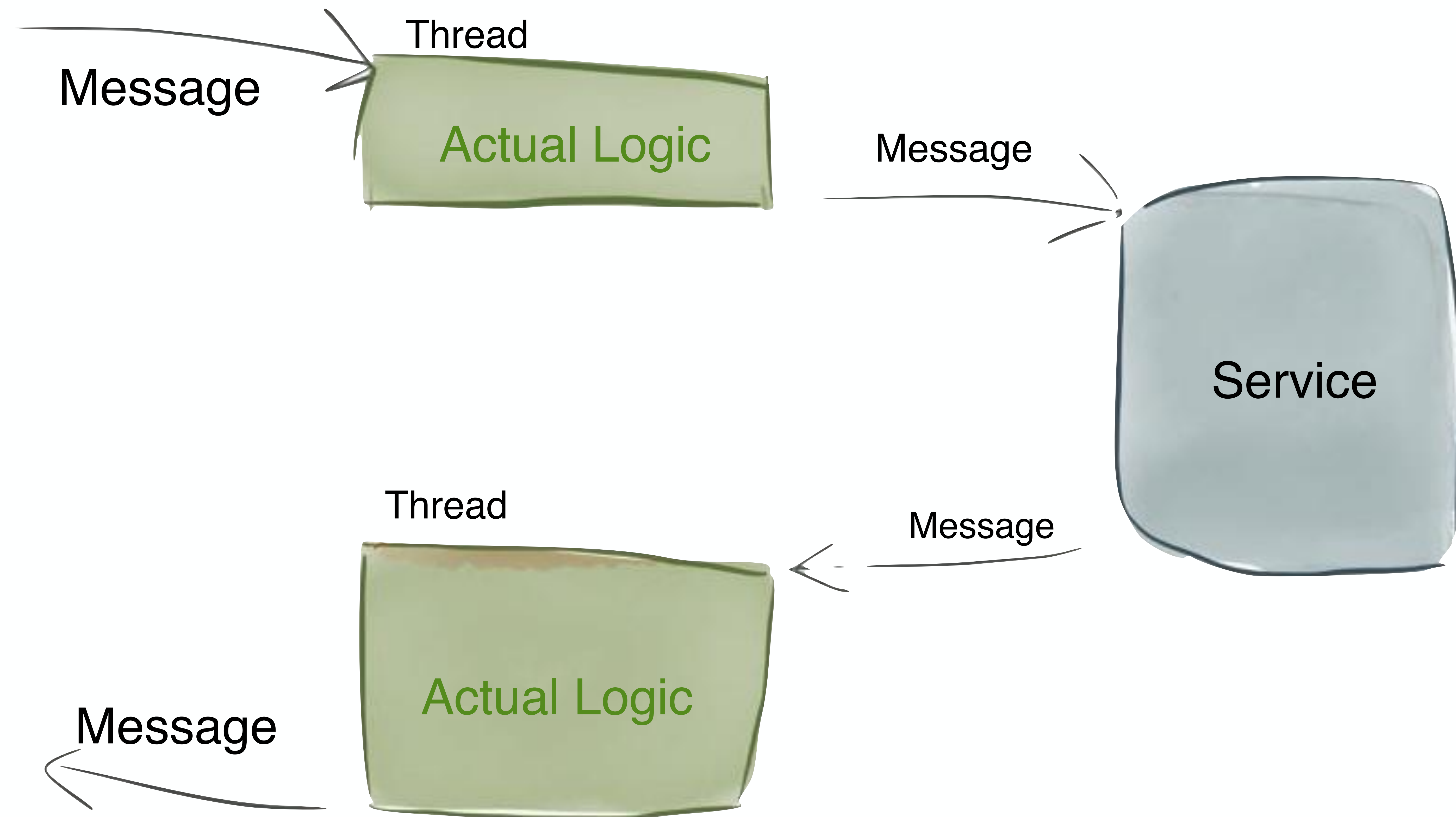
Services that are IO-bound



Services that are IO-bound

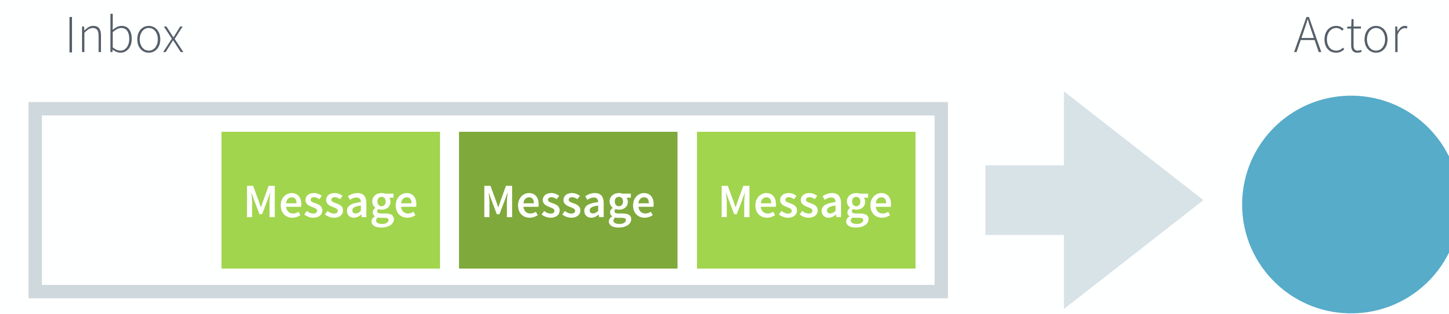


Services that are IO-bound



Akka Actor fundamentals

An Actor can…



- mutate state (including spawning a child actor)
- send messages to other actors
- change its behavior

Na-na na-na na-na na-na
Sample-time!

MVS

(Minimum Viable Sample)

Also known as "hello world"

```java
import akka.actor.typed.ActorRef;
import akka.actor.typed.ActorSystem;
import akka.actor.typed.Behavior;
import akka.actor.typed.javadsl.Behaviors;

import java.io.IOException;

public class Sample1 {

  static class Hello {
    public final String who;
    public Hello(String who) {
      this.who = who;
    }
  }

  final static Behavior<Hello> greetingBehavior =
    Behaviors.receive(Hello.class)
      .onMessage(Hello.class, (context, message) -> {
        context.getLog().info("Hello {}!", message.who);
        return Behavior.same();
      }).build();

  public static void main(String[] args) throws IOException {
    ActorSystem<Hello> actorSystem =
        ActorSystem.create(greetingBehavior, "my-system");
    ActorRef<Hello> rootActor = actorSystem;
    rootActor.tell(new Hello("Johan"));
    rootActor.tell(new Hello("Devdays Vilnius audience"));

    System.out.println("Press that any-key to terminate");
    System.in.read();
    actorSystem.terminate();
  }
}
```

```
public class Sample1 {

  static class Hello {
    public final String who;
    public Hello(String who) {
      this.who = who;
    }
  }

  final static Behavior<Hello> greetingBehavior =
    Behaviors.receive(Hello.class)
      .onMessage(Hello.class, (context, message) -> {
        context.getLog().info("Hello {}!", message.who);
        return Behavior.same();
      }).build();

  public static void main(String[] args) throws IOException {
    ActorSystem<Hello> actorSystem =
        ActorSystem.create(greetingBehavior, "my-system");
    ActorRef<Hello> rootActor = actorSystem;
    rootActor.tell(new Hello("Johan"));
    rootActor.tell(new Hello("Devdays Vilnius audience"));
```

```java
public class Sample1 {

  static class Hello {
    public final String who;
    public Hello(String who) {
      this.who = who;
    }
  }

  final static Behavior<Hello> greetingBehavior =
    Behaviors.receive(Hello.class)
      .onMessage(Hello.class, (context, message) -> {
        context.getLog().info("Hello {}!", message.who);
        return Behavior.same();
      }).build();

  public static void main(String[] args) throws IOException {
    ActorSystem<Hello> actorSystem =
        ActorSystem.create(greetingBehavior, "my-system");
    ActorRef<Hello> rootActor = actorSystem;
    rootActor.tell(new Hello("Johan"));
    rootActor.tell(new Hello("Devdays Vilnius audience"));
```

```java
    }
  }

  final static Behavior<Hello> greetingBehavior =
    Behaviors.receive(Hello.class)
      .onMessage(Hello.class, (context, message) -> {
        context.getLog().info("Hello {}!", message.who);
        return Behavior.same();
      }).build();

  public static void main(String[] args) throws IOException {
    ActorSystem<Hello> actorSystem =
        ActorSystem.create(greetingBehavior, "my-system");
    ActorRef<Hello> rootActor = actorSystem;
    rootActor.tell(new Hello("Johan"));
    rootActor.tell(new Hello("Devdays Vilnius audience"));

    System.out.println("Press that any-key to terminate");
    System.in.read();
    actorSystem.terminate();
  }
}
```

```
    }
  }

  final static Behavior<Hello> greetingBehavior =
    Behaviors.receive(Hello.class)
      .onMessage(Hello.class, (context, message) -> {
        context.getLog().info("Hello {}!", message.who);
        return Behavior.same();
      }).build();

  public static void main(String[] args) throws IOException {
    ActorSystem<Hello> actorSystem =
        ActorSystem.create(greetingBehavior, "my-system");
    ActorRef<Hello> rootActor = actorSystem;
    rootActor.tell(new Hello("Johan"));
    rootActor.tell(new Hello("Devdays Vilnius audience"));

    System.out.println("Press that any-key to terminate");
    System.in.read();
    actorSystem.terminate();
  }
}
```

```scala
import akka.actor.typed.{ActorRef, ActorSystem, Behavior}
import akka.actor.typed.scaladsl.Behaviors

import scala.io.StdIn

object Sample1 {

  case class Hello(who: String)

  val greetingBehavior: Behavior[Hello] =
    Behaviors.receive { (ctx, hello) =>
      ctx.log.info(s"Hello ${hello.who}!")
      Behaviors.same
    }

  def main(args: Array[String]): Unit = {
    val system = ActorSystem(greetingBehavior, "my-system")
    val rootActor: ActorRef[Hello] = system

    rootActor ! Hello("Johan")
    rootActor ! Hello("Devdays Vilnius audience")

    println("Press the any-key to terminate")
    StdIn.readLine()
    system.terminate()
  }
}
```

Let’s do another one right away

```
interface Command {}

static class ChangeGreeting implements Command {
  public final String newGreeting;
  public ChangeGreeting(String newGreeting) {
    this.newGreeting = newGreeting;
  }
}

static class Hello implements Command {
  public final String who;
  public Hello(String who) {
    this.who = who;
  }
}

public static Behavior<Command> dynamicGreetingBehavior(String greeting) {
  return Behaviors.receive(Command.class)
      .onMessage(Hello.class, (context, message) -> {
        context.getLog().info(greeting + " " + message.who + "!");
        return Behavior.same();
      }).onMessage(ChangeGreeting.class, (context, changeGreeting) ->
        dynamicGreetingBehavior(changeGreeting.newGreeting)
      ).build();
}

public static void main(String[] args) throws IOException {
  var actorSystem =
      ActorSystem.create(dynamicGreetingBehavior("Hello"), "my-system");
  actorSystem.tell(new Hello("Johan"));
  actorSystem.tell(new ChangeGreeting("Sveiki"));
  actorSystem.tell(new Hello("Devdays Vilnius audience"));
}
```

```java
interface Command {}

static class ChangeGreeting implements Command {
  public final String newGreeting;
  public ChangeGreeting(String newGreeting) {
    this.newGreeting = newGreeting;
  }
}

static class Hello implements Command {
  public final String who;
  public Hello(String who) {
    this.who = who;
  }
}

public static Behavior<Command> dynamicGreetingBehavior(String greeting) {
  return Behaviors.receive(Command.class)
      .onMessage(Hello.class, (context, message) -> {
        context.getLog().info(greeting + " " + message.who + "!");
        return Behavior.same();
      }).onMessage(ChangeGreeting.class, (context, changeGreeting) ->
        dynamicGreetingBehavior(changeGreeting.newGreeting)
      ).build();
}
```

```java
static class Hello implements Command {
  public final String who;
  public Hello(String who) {
    this.who = who;
  }
}

public static Behavior<Command> dynamicGreetingBehavior(String greeting) {
  return Behaviors.receive(Command.class)
      .onMessage(Hello.class, (context, message) -> {
        context.getLog().info(greeting + " " + message.who + "!");
        return Behavior.same();
      }).onMessage(ChangeGreeting.class, (context, changeGreeting) ->
        dynamicGreetingBehavior(changeGreeting.newGreeting)
      ).build();
}

public static void main(String[] args) throws IOException {
  var actorSystem =
      ActorSystem.create(dynamicGreetingBehavior("Hello"), "my-system");
  actorSystem.tell(new Hello("Johan"));
  actorSystem.tell(new ChangeGreeting("Sveiki"));
  actorSystem.tell(new Hello("Devdays Vilnius audience"));
}
```

```java
public static Behavior<Command> dynamicGreetingBehavior(String greeting) {
  return Behaviors.receive(Command.class)
      .onMessage(Hello.class, (context, message) -> {
        context.getLog().info(greeting + " " + message.who + "!");
        return Behavior.same();
      }).onMessage(ChangeGreeting.class, (context, changeGreeting) ->
        dynamicGreetingBehavior(changeGreeting.newGreeting)
      ).build();
}

public static void main(String[] args) throws IOException {
  var actorSystem =
      ActorSystem.create(dynamicGreetingBehavior("Hello"), "my-system");
  actorSystem.tell(new Hello("Johan"));
  actorSystem.tell(new ChangeGreeting("Sveiki"));
  actorSystem.tell(new Hello("Devdays Vilnius audience"));
}
```

```scala
sealed trait Command
case class Hello(who: String) extends Command
case class ChangeGreeting(newGreeting: String) extends Command

def dynamicGreetingBehavior(greeting: String): Behavior[Command] =
  Behaviors.receive { (ctx, message) =>
    message match {
      case Hello(who) =>
        ctx.log.info(s"$greeting ${who}!")
        Behaviors.same
      case ChangeGreeting(newGreeting) =>
        dynamicGreetingBehavior(newGreeting)
    }
  }

def main(args: Array[String]): Unit = {
  val system = ActorSystem(dynamicGreetingBehavior("Hello"), "my-system")

  system ! Hello("Johan")
  system ! ChangeGreeting("Sveiki")
  system ! Hello("Devdays Vilnius audience")
}
```

But I don’t like it

(the FP style, that is)

```java
static class MutableGreetingBehavior extends MutableBehavior<Command> {

  private final ActorContext<Command> context;
  private String greeting;

  public MutableGreetingBehavior(String initialGreeting, ActorContext<Command> context) {
    this.context = context;
    greeting = initialGreeting;
  }

  @Override
  public Behaviors.Receive<Command> createReceive() {
    return receiveBuilder()
        .onMessage(Hello.class, this::onHello)
        .onMessage(ChangeGreeting.class, this::onChangeGreeting)
        .build();
  }

  private Behavior<Command> onHello(Hello hello) {
    context.getLog().info(greeting + " " + hello.who + "!");
    return Behaviors.same();
  }

  private Behavior<Command> onChangeGreeting(ChangeGreeting changeGreeting) {
    greeting = changeGreeting.newGreeting;
    return Behaviors.same();
  }
}
```

```java
static class MutableGreetingBehavior extends MutableBehavior<Command> {

  private final ActorContext<Command> context;
  private String greeting;

  public MutableGreetingBehavior(String initialGreeting, ActorContext<Co
    this.context = context;
    greeting = initialGreeting;
  }

  @Override
  public Behaviors.Receive<Command> createReceive() {
    return receiveBuilder()
        .onMessage(Hello.class, this::onHello)
        .onMessage(ChangeGreeting.class, this::onChangeGreeting)
        .build();
  }

  private Behavior<Command> onHello(Hello hello) {
    context.getLog().info(greeting + " " + hello.who + "!");
    return Behaviors.same();
  }
```

```java
static class MutableGreetingBehavior extends MutableBehavior<Command> {

  private final ActorContext<Command> context;
  private String greeting;

  public MutableGreetingBehavior(String initialGreeting, ActorContext<Co
    this.context = context;
    greeting = initialGreeting;
  }

  @Override
  public Behaviors.Receive<Command> createReceive() {
    return receiveBuilder()
        .onMessage(Hello.class, this::onHello)
        .onMessage(ChangeGreeting.class, this::onChangeGreeting)
        .build();
  }

  private Behavior<Command> onHello(Hello hello) {
    context.getLog().info(greeting + " " + hello.who + "!");
    return Behaviors.same();
  }
```

```java
static class MutableGreetingBehavior extends MutableBehavior<Command> {

  private final ActorContext<Command> context;
  private String greeting;

  public MutableGreetingBehavior(String initialGreeting, ActorContext<Co
    this.context = context;
    greeting = initialGreeting;
  }

  @Override
  public Behaviors.Receive<Command> createReceive() {
    return receiveBuilder()
        .onMessage(Hello.class, this::onHello)
        .onMessage(ChangeGreeting.class, this::onChangeGreeting)
        .build();
  }

  private Behavior<Command> onHello(Hello hello) {
    context.getLog().info(greeting + " " + hello.who + "!");
    return Behaviors.same();
  }
```

```java
    greeting = initialGreeting;
  }

  @Override
  public Behaviors.Receive<Command> createReceive() {
    return receiveBuilder()
        .onMessage(Hello.class, this::onHello)
        .onMessage(ChangeGreeting.class, this::onChangeGreeting)
        .build();
  }

  private Behavior<Command> onHello(Hello hello) {
    context.getLog().info(greeting + " " + hello.who + "!");
    return Behaviors.same();
  }

  private Behavior<Command> onChangeGreeting(ChangeGreeting changeGreeti
    greeting = changeGreeting.newGreeting;
    return Behaviors.same();
  }
}
```

```java
public static void main(String[] args) {
  var system = ActorSystem.<Command>create(
      Behaviors.setup((context) ->
        new MutableGreetingBehavior("Hello", context)),
      "my-system"
  );
  system.tell(new Hello("Johan"));
  system.tell(new ChangeGreeting("Sveiki"));
  system.tell(new Hello("Devdays Vilnius audience"));
}
```

The need for strong/strict typing

“Sending the wrong message to an actor is actually quite uncommon”

– Myself, last week

- Discoverability, how are things related
- More compile time type-safety, less runtime debugging

Types helps productivity!

(and results in more reliable systems in production)

“Realistic” Example

Burglar Alarm

- enabled/disabled with a pin code
- accepts notifications about “activity”
- if enabled on activity, sound the alarm

```java
interface AlarmMessage {}

static class EnableAlarm implements AlarmMessage {
  public final String pinCode;
  public EnableAlarm(String pinCode) {
    this.pinCode = pinCode;
  }
}

static class DisableAlarm implements AlarmMessage {
  public final String pinCode;
  public DisableAlarm(String pinCode) {
    this.pinCode = pinCode;
  }
}

static class ActivityEvent implements AlarmMessage { }
```

```java
public static Behavior<AlarmMessage> enabledAlarm(String pinCode) {
  return Behaviors.receive(AlarmMessage.class)
      .onMessage(
        DisableAlarm.class,
        // predicate
        (disable) -> disable.pinCode.equals(pinCode),
        (context, message) -> {
          context.getLog().info("Correct pin entered, disabling alarm");
          return disabledAlarm(pinCode);
        }
      ).onMessage(ActivityEvent.class, (context, activityEvent) -> {
        context.getLog().warning("EOEOEOEOEOE ALARM ALARM!!!");
        return Behaviors.same();
      }).build();

}
```

```java
public static Behavior<AlarmMessage> disabledAlarm(String pinCode) {
  return Behaviors.receive(AlarmMessage.class)
      .onMessage(EnableAlarm.class,
        // predicate
        (enable) -> enable.pinCode.equals(pinCode),
        (context, message) -> {
          context.getLog().info("Correct pin entered, enabling alarm");
          return enabledAlarm(pinCode);
        }
      ).build();
}
```

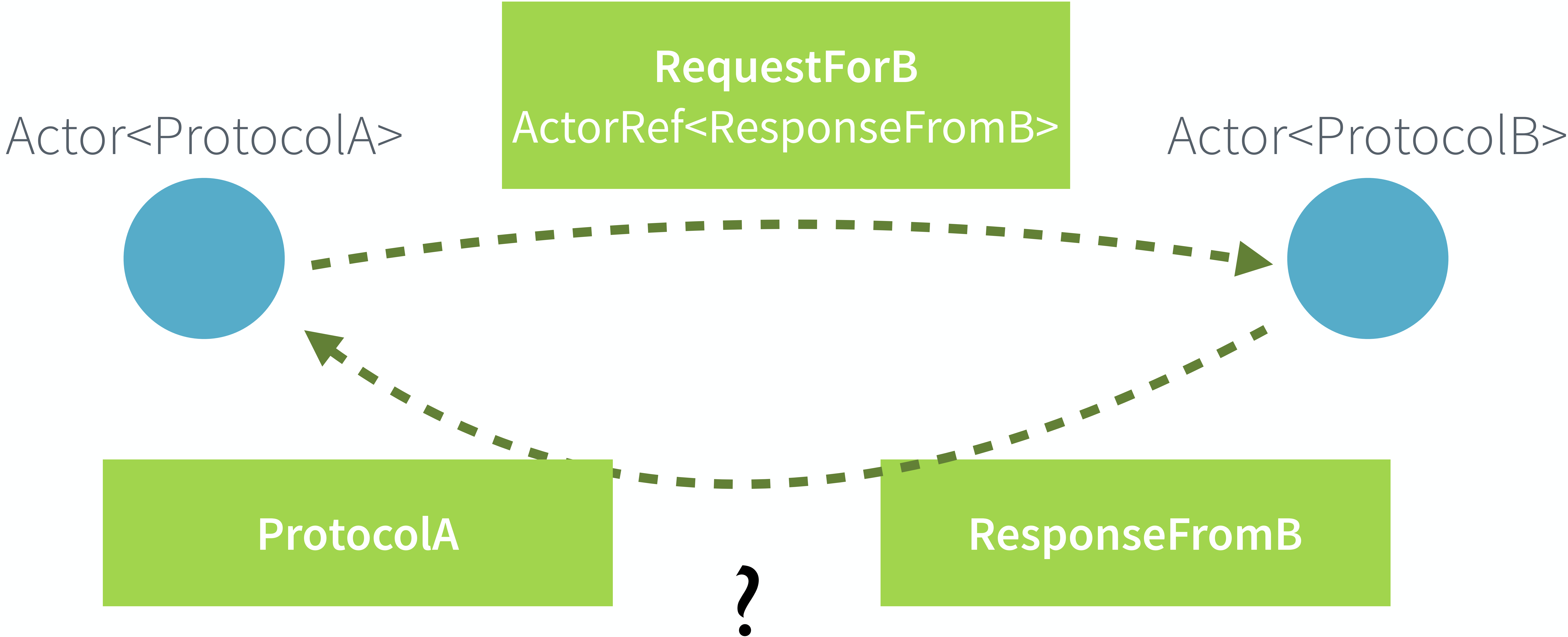
```java
public static void main(String[] args) {
  var system = ActorSystem.create(enabledAlarm("0000"), "my-system");
  system.tell(new ActivityEvent());
  system.tell(new DisableAlarm("1234"));
  system.tell(new ActivityEvent());
  system.tell(new DisableAlarm("0000"));
  system.tell(new ActivityEvent());
  system.tell(new EnableAlarm("0000"));
}
```

```scala
sealed trait AlarmMessage
case class EnableAlarm(pinCode: String) extends AlarmMessage
case class DisableAlarm(pinCode: String) extends AlarmMessage
case object ActivityEvent extends AlarmMessage

def enabledAlarm(pinCode: String): Behavior[AlarmMessage] =
  Behaviors.receive { (context, message) =>
    message match {
      case ActivityEvent =>
        context.log.warning("EOEOEOEOEOE ALARM ALARM!!!")
        Behaviors.same

      case DisableAlarm(`pinCode`) =>
        context.log.info("Correct pin entered, disabling alarm");
        disabledAlarm(pinCode)
      case _ => Behaviors.unhandled
    }

  }

def disabledAlarm(pinCode: String): Behavior[AlarmMessage] =
  Behaviors.receivePartial {
    case (context, EnableAlarm(`pinCode`)) =>
      context.log.info("Correct pin entered, enabling alarm")
      enabledAlarm(pinCode)
  }

def main(args: Array[String]): Unit = {
  val system = ActorSystem.create(enabledAlarm("0000"), "my-system")
  system.tell(ActivityEvent)
  system.tell(DisableAlarm("1234"))
  system.tell(ActivityEvent)
  system.tell(DisableAlarm("0000"))
  system.tell(ActivityEvent)
  system.tell(EnableAlarm("0000"))
  system.tell(ActivityEvent)
}
```

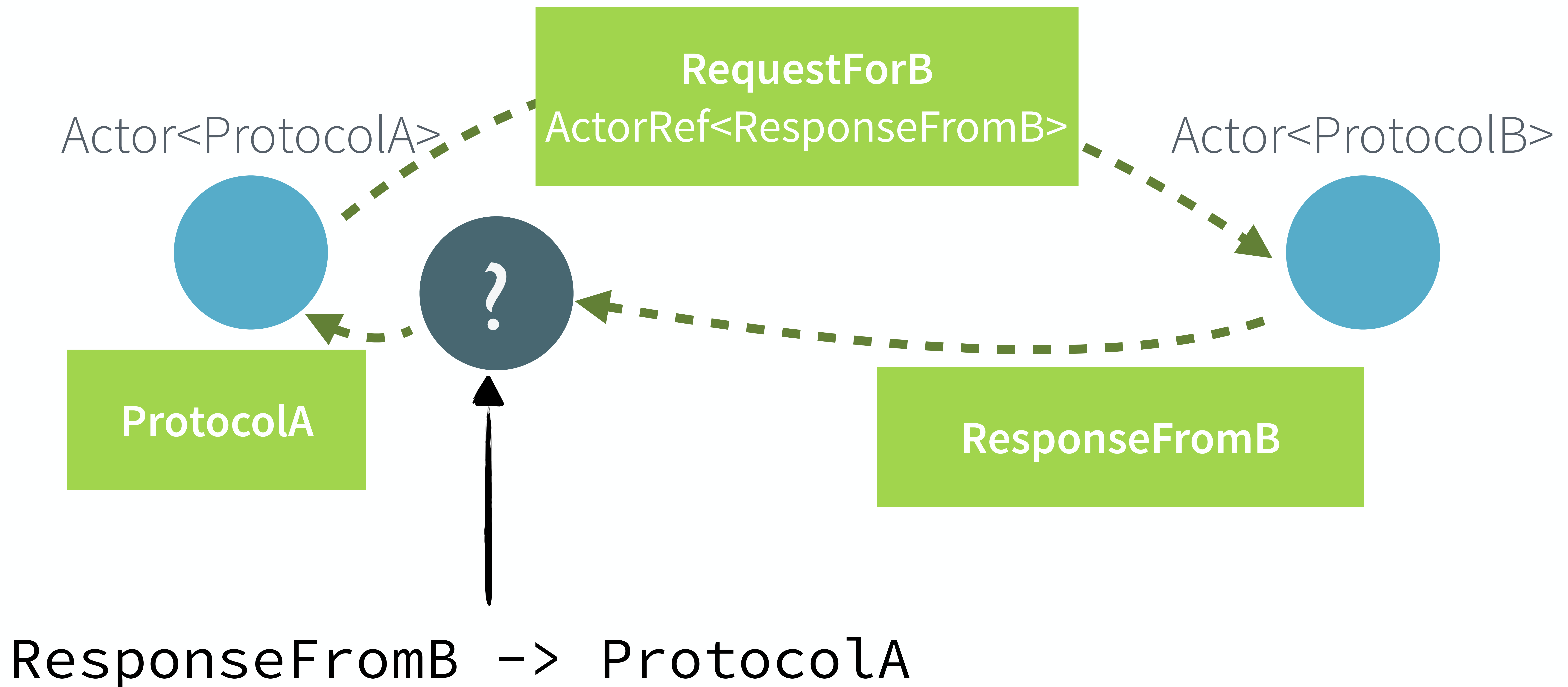
Request-response with other actors

- In untyped actors the sender was auto-magically available
- In Akka Typed the recipient of response has to be encoded in message *(why?)*

Request-response with other actors



Request-response with other actors



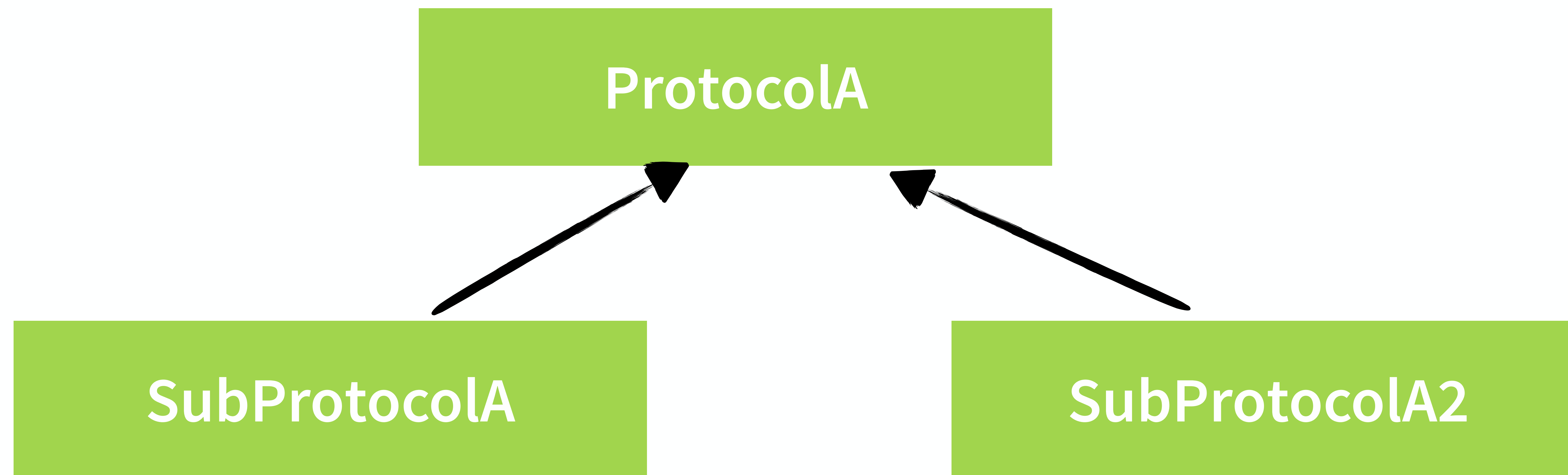
One off with Ask

```
context.ask(
  ResponseFromB.class,
  actorB,
  Timeout.create(Duration.ofSeconds(5)),
  (ActorRef<ResponseFromB> respondTo) -> new RequestForB(respondTo),
  (ResponseFromB res, Throwable failure) -> {
    if (res != null) return new ProtocolAMessage(res);
    else throw new RuntimeException("Request failed", failure);
  }
);
```

Long live the adapter

```
ActorRef<ResponseFromB> adapter = context.messageAdapter(
    ResponseFromB.class,
    (responseFromB) -> new ProtocolAMessage(responseFromB)
);

actorB.tell(new RequestForB(adapter));
```

Keeping parts of the protocol to ourselves




```
ActorRef<ProtocolA> actorRef = …
ActorRef<SubProtocolA> moreNarrowActorRef = actorRef.narrow();
```

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

Why is it so hard?

The Joys of Computer Networks:

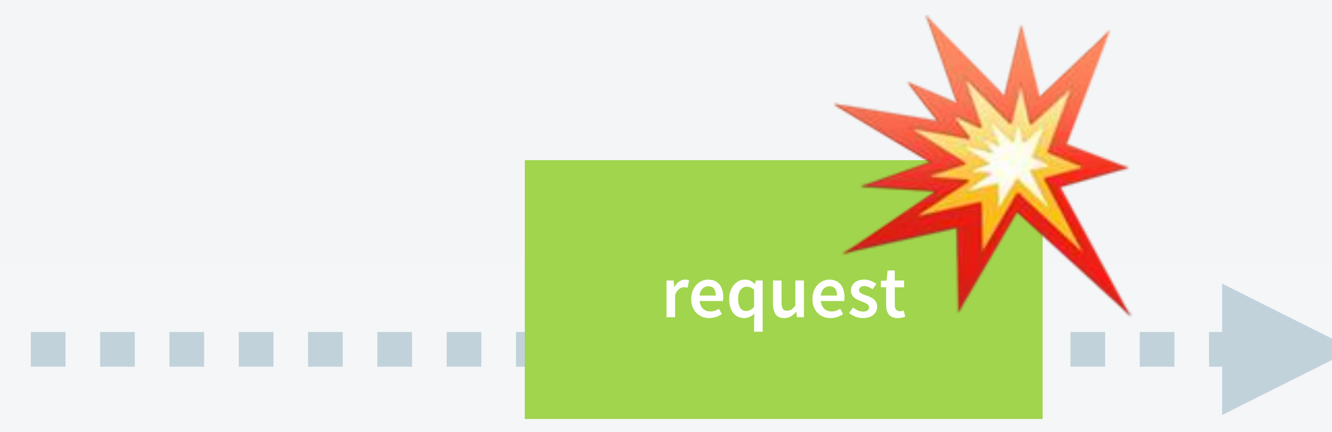
Reliability: power failure, old network equipment, network congestion, coffee in router, rodents, that guy in the IT dept., DDOS attacks…

Latency: loopback vs local net vs shared congested local net vs internet

Bandwidth: again loopback vs local vs shared local vs internet

Partial Failure

Partial Failure



Partial Failure

Partial Failure

Why do it, if it is so hard?

Data or processing doesn’t fit a single machine

Many objects, that should be kept in memory. Many not so powerful servers can be cheaper than a supercomputer.

Elasticity

Being able to scale in (less servers) and out (more servers) depending on load. Not paying for servers unless you need them.

Resilience

Building systems that will keep working in the face of failures or degrade gracefully.

Actor Model vs Network

Interaction already modelled as immutable messages

Data travels over the network in packages, changes has to be explicitly sent back.

At most once

Data reaches a node on the other side at most once, but can be lost, already part of model!

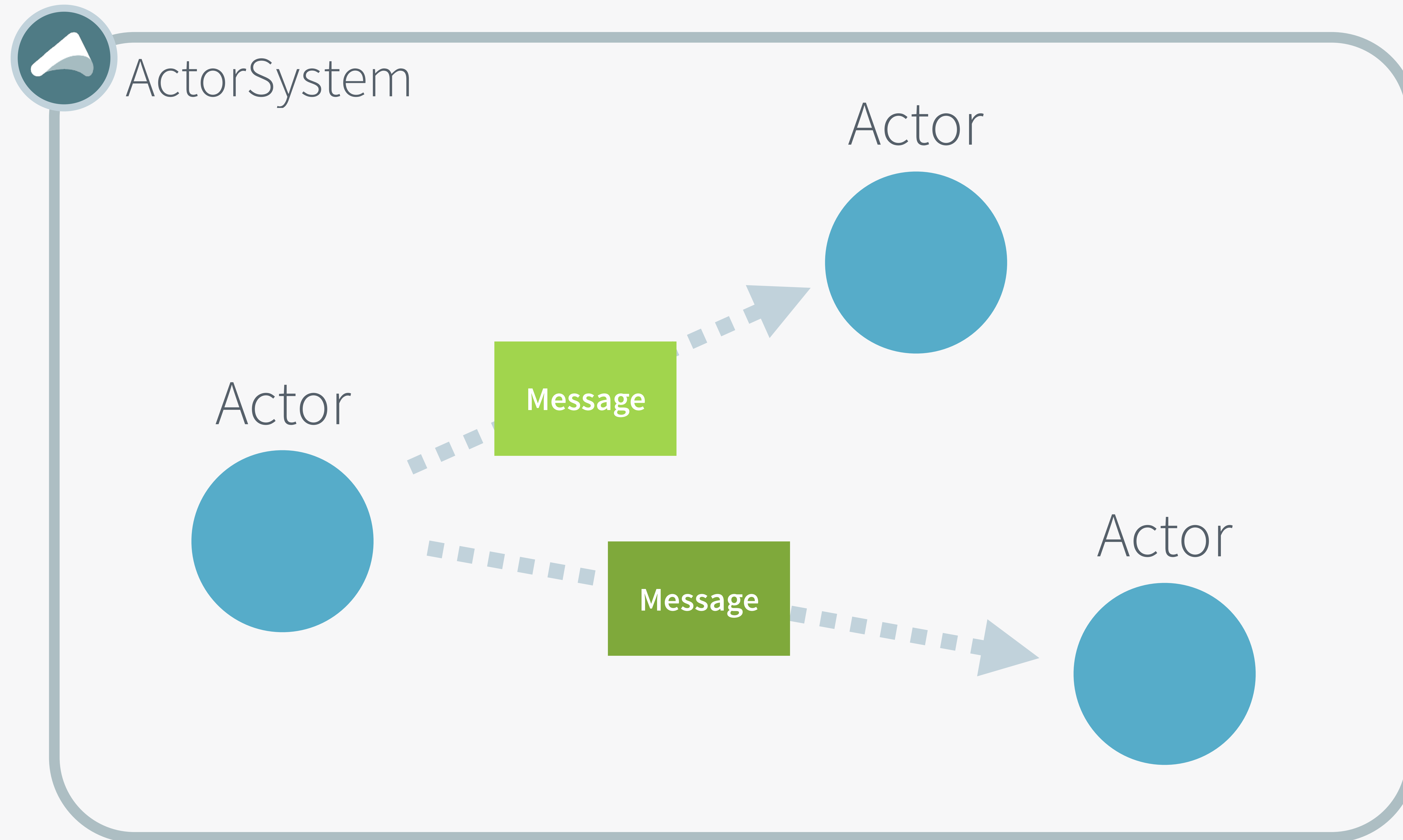
A recipient of a message can reply directly to sender

Regardless if there were intermediate recipients of the message

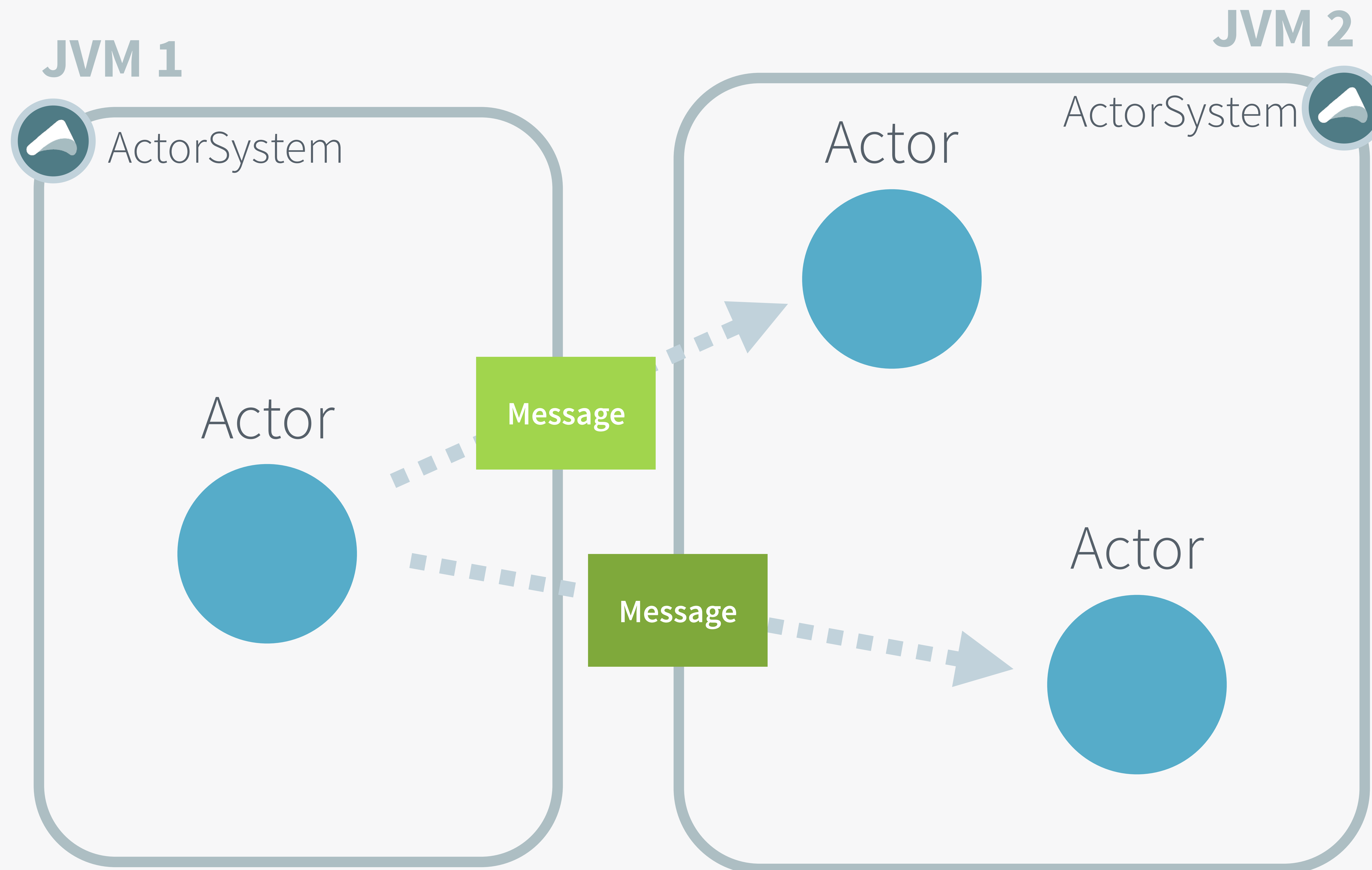
Messages not limited to request response

Messages can flow in either direction when two systems are connected.

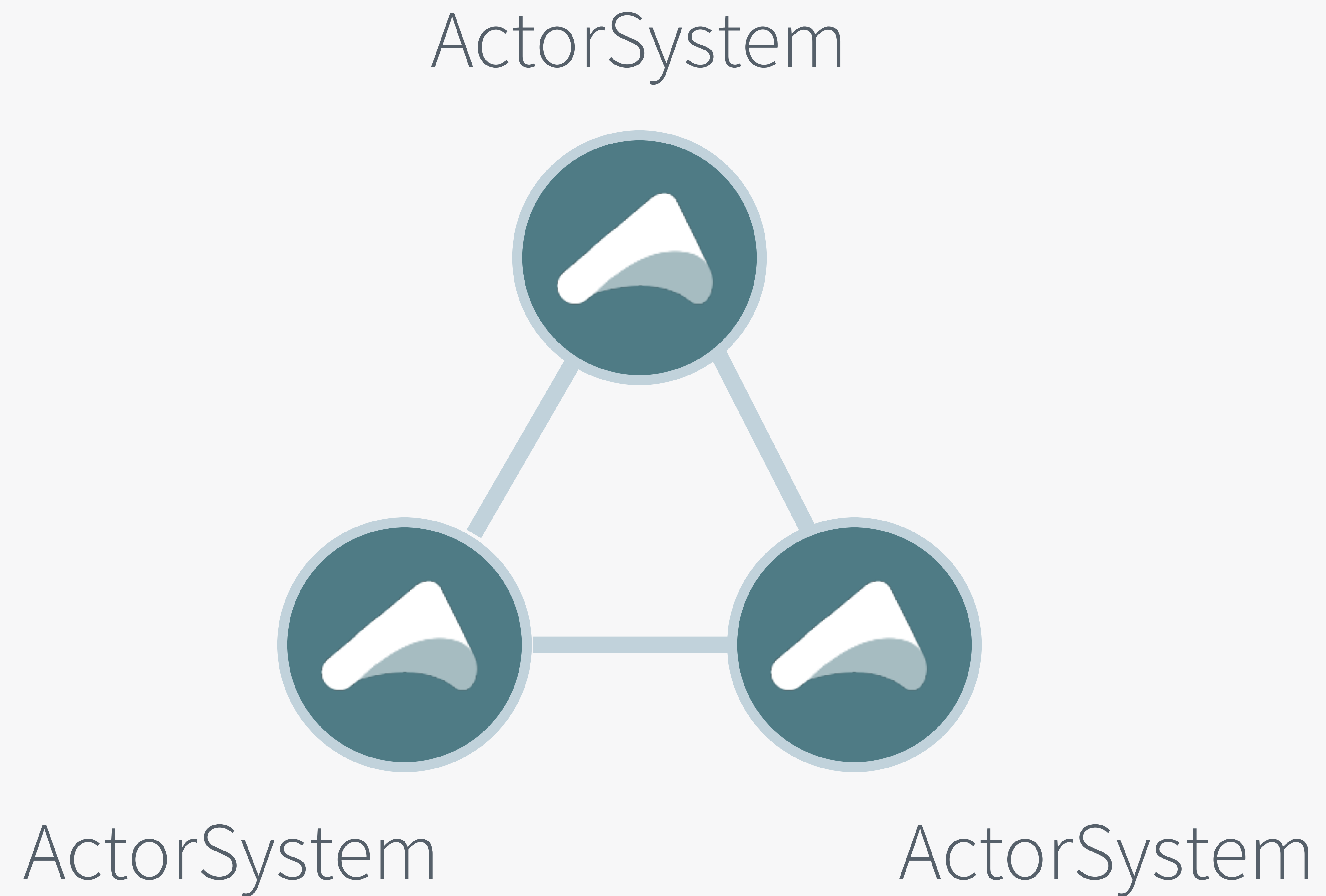
Local



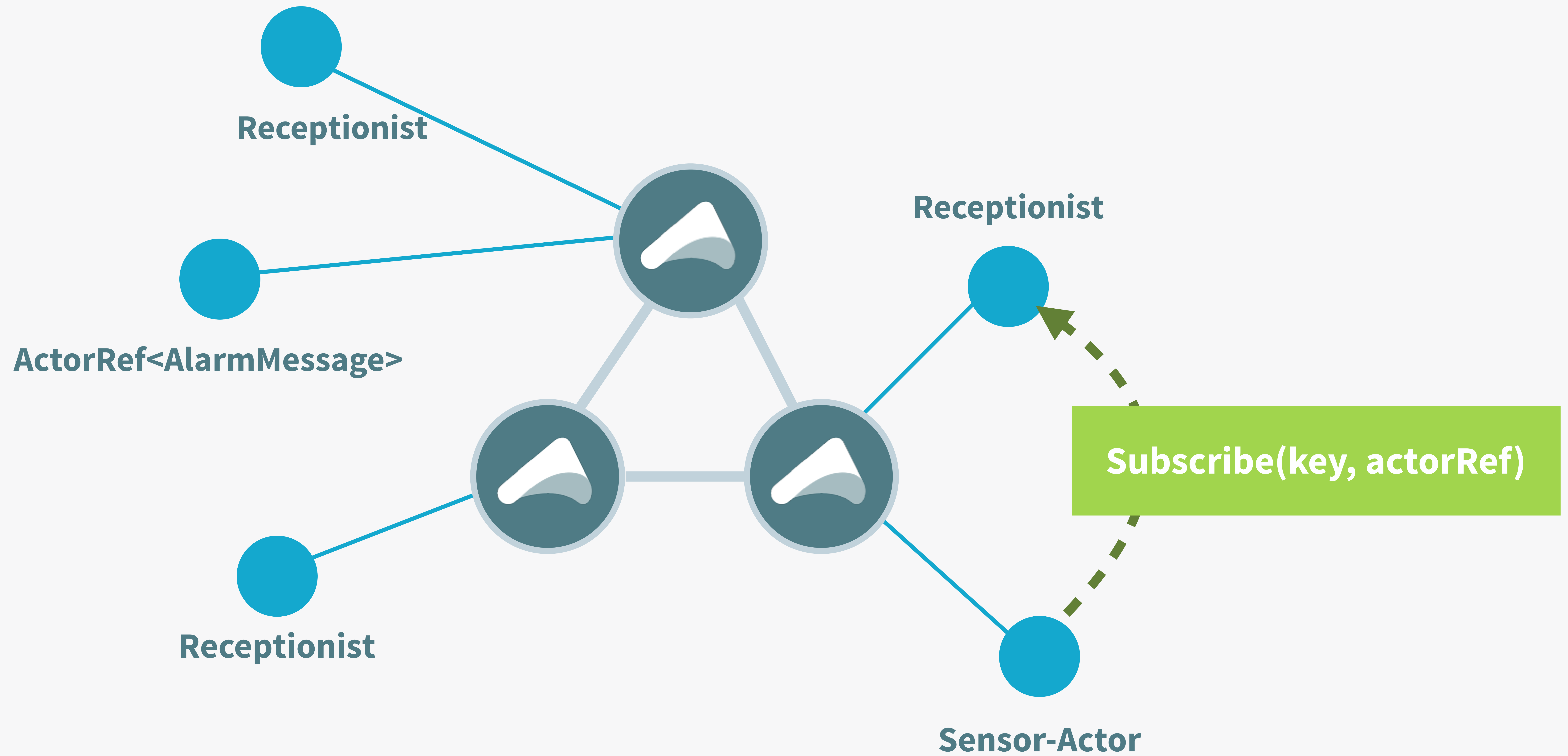
Distributed



Akka Cluster + Akka Typed

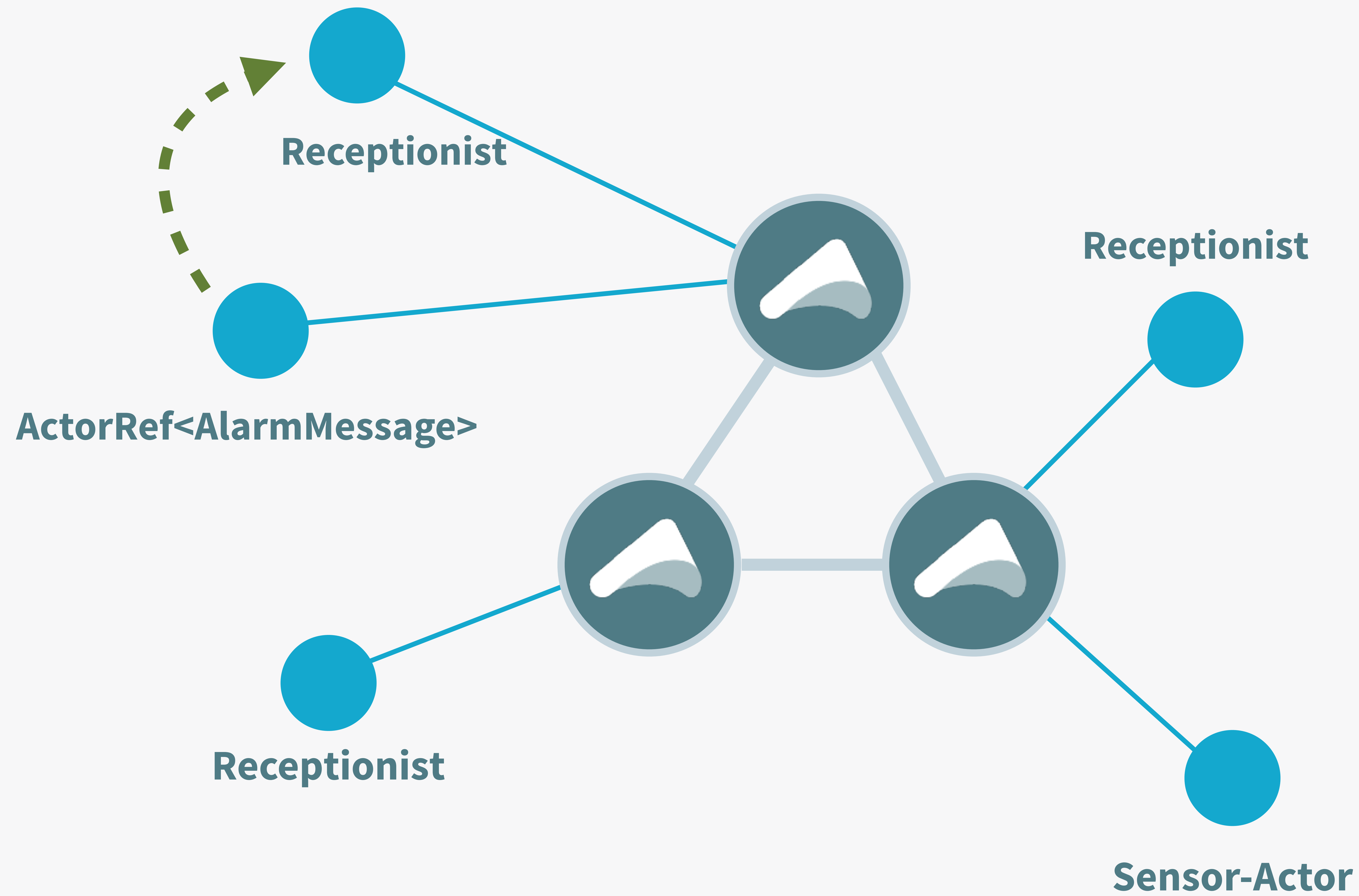


Typed Receptionist

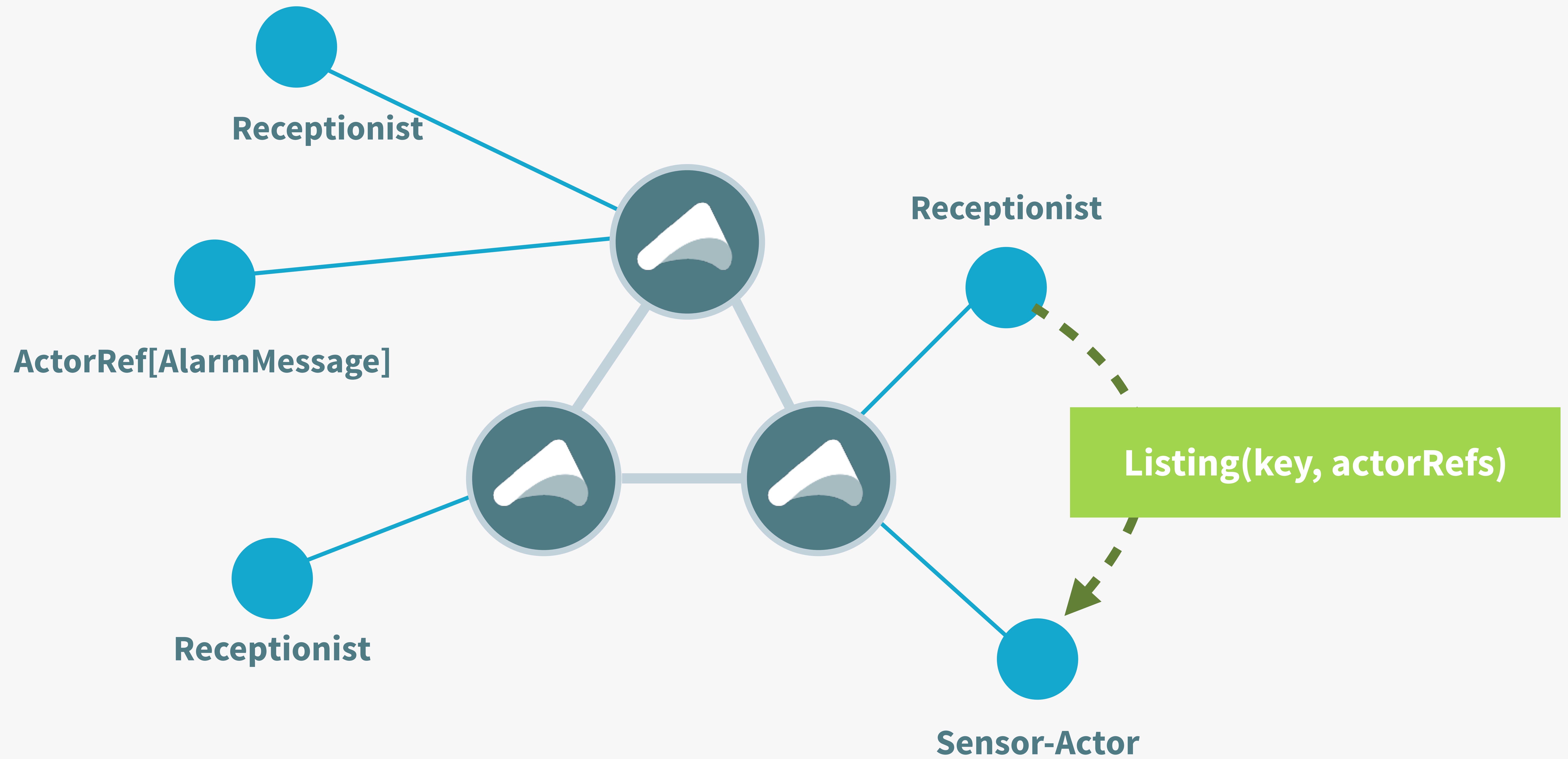


Typed Receptionist

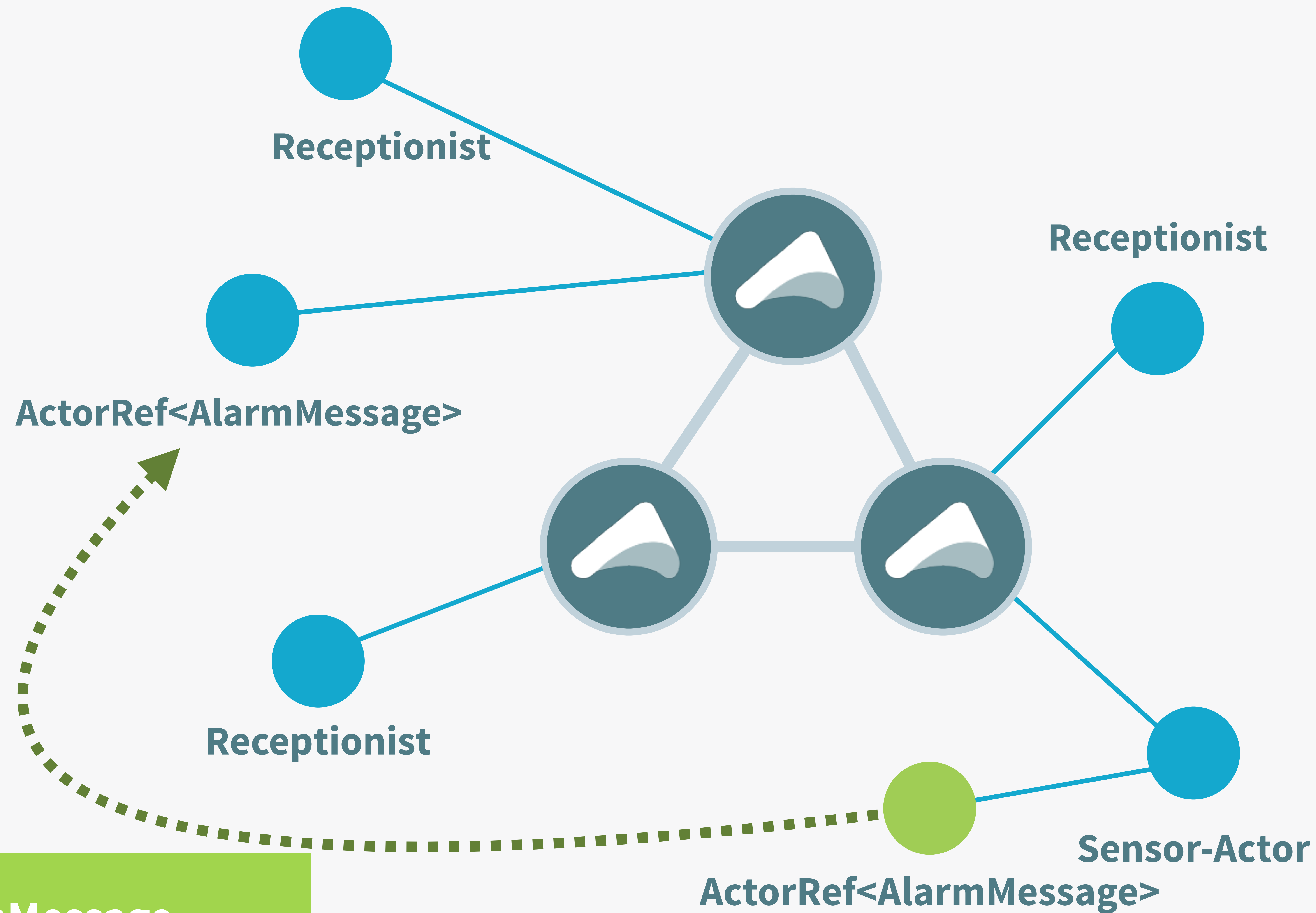
Register(Key<AlarmMessage>, ActorRef<AlarmMessage>)



Typed Receptionist



Typed Receptionist



Distributed Burglar Alarm

- we can have any number of nodes
- a sensor on any node can report activity to the alarm

```java
static final ServiceKey<ActivityEvent> serviceKey =
  ServiceKey.create(ActivityEvent.class, "alarm");

interface AlarmMessage extends Serializable {}

static class EnableAlarm implements AlarmMessage {
  public final String pinCode;
  public EnableAlarm(String pinCode) {
    this.pinCode = pinCode;
  }
}

static class DisableAlarm implements AlarmMessage {
  public final String pinCode;
  public DisableAlarm(String pinCode) {
    this.pinCode = pinCode;
  }
}

static class ActivityEvent implements AlarmMessage { }
```

```java
static final ServiceKey<ActivityEvent> serviceKey =
  ServiceKey.create(ActivityEvent.class, "alarm");

interface AlarmMessage extends Serializable {}

static class EnableAlarm implements AlarmMessage {
  public final String pinCode;
  public EnableAlarm(String pinCode) {
    this.pinCode = pinCode;
  }
}

static class DisableAlarm implements AlarmMessage {
  public final String pinCode;
  public DisableAlarm(String pinCode) {
    this.pinCode = pinCode;
  }
}

static class ActivityEvent implements AlarmMessage { }
```

```java
public static Behavior<AlarmMessage> alarm(String pinCode) {
  return Behaviors.setup((context) -> {
    ActorRef<Receptionist.Command> receptionist =
      context.getSystem().receptionist();
    receptionist.tell(
      Receptionist.register(serviceKey, context.getSelf().narrow()));
    return enabledAlarm(pinCode);
  });
}
```

```java
public static class SensorBehavior extends MutableBehavior<TriggerCommand> {

  private final ActorContext<TriggerCommand> context;
  private Set<ActorRef<ActivityEvent>> alarms = Collections.EMPTY_SET;

  public SensorBehavior(ActorContext<TriggerCommand> context) {
    this.context = context;

    // we need to transform the messages from the receptionist protocol
    // into our own protocol:
    ActorRef<Receptionist.Listing> adapter = context.messageAdapter(
        Receptionist.Listing.class,
        (listing) -> new AlarmActorUpdate(listing.getServiceInstances(serviceKey))
    );
    ActorRef<Receptionist.Command> receptionist = context.getSystem().receptionist();
    receptionist.tell(Receptionist.subscribe(serviceKey, adapter));
  }

  @Override
  public Behaviors.Receive<TriggerCommand> createReceive() {
    return receiveBuilder()
        .onMessage(TriggerSensor.class, this::onTrigger)
        .onMessage(AlarmActorUpdate.class, this::onAlarmActorUpdate)
        .build();
  }

  private Behavior<TriggerCommand> onTrigger(TriggerSensor trigger) {
    final ActivityEvent activityEvent = new ActivityEvent();
    if (alarms.isEmpty()) context.getLog().warning("Saw trigger but no alarms known yet");
    alarms.forEach((alarm) ->
      alarm.tell(activityEvent)
    );
    return Behaviors.same();
  }

  private Behavior<TriggerCommand> onAlarmActorUpdate(AlarmActorUpdate update) {
    context.getLog().info("Got alarm actor list update");
    alarms = update.alarms;
    return Behaviors.same();
  }
}
```

```java
public static class SensorBehavior extends MutableBehavior<TriggerCommand> {

  private final ActorContext<TriggerCommand> context;
  private Set<ActorRef<ActivityEvent>> alarms = Collections.EMPTY_SET;

  public SensorBehavior(ActorContext<TriggerCommand> context) {
    this.context = context;

    // we need to transform the messages from the receptionist protocol
    // into our own protocol:
    ActorRef<Receptionist.Listing> adapter = context.messageAdapter(
        Receptionist.Listing.class,
        (listing) -> new AlarmActorUpdate(listing.getServiceInstances(serviceKey))
    );
    ActorRef<Receptionist.Command> receptionist = context.getSystem().receptionist();
    receptionist.tell(Receptionist.subscribe(serviceKey, adapter));
  }

  @Override
  public Behaviors.Receive<TriggerCommand> createReceive() {
    return receiveBuilder()
        .onMessage(TriggerSensor.class, this::onTrigger)
        .onMessage(AlarmActorUpdate.class, this::onAlarmActorUpdate)
        .build();
  }

  private Behavior<TriggerCommand> onTrigger(TriggerSensor trigger) {
    final ActivityEvent activityEvent = new ActivityEvent();
    if (alarms.isEmpty()) context.getLog().warning("Saw trigger but no alarms known yet");
    alarms.forEach((alarm) ->
```

```java
    ActorRef<Receptionist.Command> receptionist = context.getSystem().receptionist();
    receptionist.tell(Receptionist.subscribe(serviceKey, adapter));
  }

  @Override
  public Behaviors.Receive<TriggerCommand> createReceive() {
    return receiveBuilder()
        .onMessage(TriggerSensor.class, this::onTrigger)
        .onMessage(AlarmActorUpdate.class, this::onAlarmActorUpdate)
        .build();
  }

  private Behavior<TriggerCommand> onTrigger(TriggerSensor trigger) {
    final ActivityEvent activityEvent = new ActivityEvent();
    if (alarms.isEmpty()) context.getLog().warning("Saw trigger but no alarms known yet");
    alarms.forEach((alarm) ->
      alarm.tell(activityEvent)
    );
    return Behaviors.same();
  }

  private Behavior<TriggerCommand> onAlarmActorUpdate(AlarmActorUpdate update) {
    context.getLog().info("Got alarm actor list update");
    alarms = update.alarms;
    return Behaviors.same();
  }
}
```

```java
public static void main(String[] args) throws Exception {

  ActorSystem<AlarmMessage> system1 =
    ActorSystem.create(alarm("0000"), "my-cluster");
  ActorSystem<TriggerCommand> system2 =
    ActorSystem.create(sensorBehavior(), "my-cluster");
  ActorSystem<TriggerCommand> system3 =
      ActorSystem.create(sensorBehavior(), "my-cluster");

  // first join the first node to itself to form a cluster
  Cluster node1 = Cluster.get(system1);
  node1.manager().tell(Join.create(node1.selfMember().address()));

  // then have node 2 and 3 join that cluster
  Cluster node2 = Cluster.get(system2);
  node2.manager().tell(Join.create(node1.selfMember().address()));
  Cluster node3 = Cluster.get(system3);
  node3.manager().tell(Join.create(node1.selfMember().address()));


  Scanner in = new Scanner(System.in);
  while (true) {
```

```
// then have node 2 and 3 join that cluster
Cluster node2 = Cluster.get(system2);
node2.manager().tell(Join.create(node1.selfMember().address()));
Cluster node3 = Cluster.get(system3);
node3.manager().tell(Join.create(node1.selfMember().address()));


Scanner in = new Scanner(System.in);
while (true) {
  try {
    System.out.println("Enter 2 or 3 to trigger activity on that node");
    int chosenNode = Integer.parseInt(in.next());
    System.out.println("Triggering sensor on node " + chosenNode);
    if (chosenNode == 2) system2.tell(new TriggerSensor());
    else if (chosenNode == 3) system3.tell(new TriggerSensor());
  } catch (Exception ex) {
    // we don't care, loop!
  }
}
}
```

Summary

- On message - message another actor, change state or behavior
- Types gives better guardrails and makes understanding the system easier
- We can use the same abstraction for local and distributed

Further reading

GitHub repo with all samples in this talk

bit.ly/2LnOU4L

Akka typed docs

doc.akka.io/docs/akka/2.5/typed/index.html

More resources

blog.akka.io - news and articles

discuss.akka.io - forums

developer.lightbend.com - samples

github.com/akka/akka - sources and getting involved

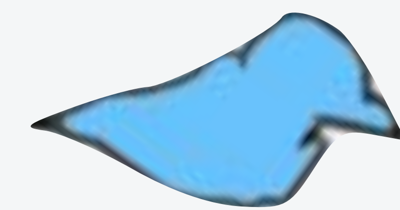
Questions?

GitHub repo with all samples in this talk

bit.ly/2LnOU4L

Thanks for listening!

http://akka.io



@apnylle
johan.andren@lightbend.com