System Architectures
Actors: Interaction Patterns

Jonathan Thaler

Department of Computer Science

FH Vorarlberg
University of Applied Sciences

Fire and Forget

Fire and Forget

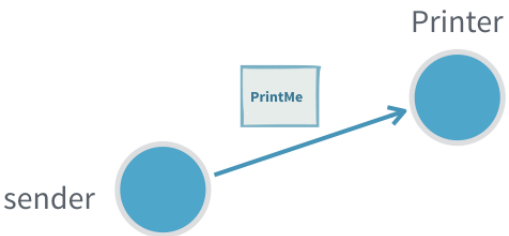

Figure: Fire and Forget interaction

- The fundamental way to interact with an actor is through `actorRef.tell(message)`.
- Sending a message with `tell` can safely be done from **any thread**.
- Tell is **asynchronous** which means that the method returns right away.
- After the statement is executed there is **no guarantee** that the message has been processed by the recipient yet.
- It also means there is no way to know if the message was **received**, the processing succeeded or failed.

Fire and Forget

Useful when:

- It is **not critical** to be sure that the message was processed.
- There is no way to **act** on non successful delivery or processing.
- We want to **minimize** the number of messages created to get higher throughput (sending a response would require creating twice the number of messages)

Problems:

- If the inflow of messages is higher than the actor can process the inbox will **fill up** and can in the worst case cause the JVM crash with an `OutOfMemoryError`.
- If the message gets lost, the sender will **not know**.

,

Request-Response

Request-Response

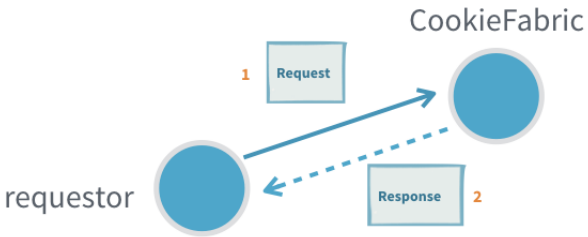

Figure: Request-Response interaction

- Many interactions between actors require one or more **response** message being sent back from the receiving actor.
- A response message can be a result of a **query**, some form of **acknowledgment** that the message was received and processed or events that the request subscribed to.
- In Akka the **recipient** of responses has to be encoded as a **field** in the message itself, which the recipient can then use to send (`tell`) a response back.

Request-Response

Useful when:

- Subscribing to an actor that will send many **response** messages back.

Problems:

- Actors **seldom** have a response message from another actor as a part of their protocol (see *Adapted Response*).
- It is **hard** to detect that a message request was not delivered or processed (see *Request-Response with ask between two actors*).
- Unless the protocol already includes a way to provide **context**, for example a request id that is also sent in the response, it is **not possible** to tie an interaction to some specific context without introducing a **new**, separate, actor (see *Request-Response with ask between two actors* or *Per session child Actor*).

Adapted Response

Adapted Response

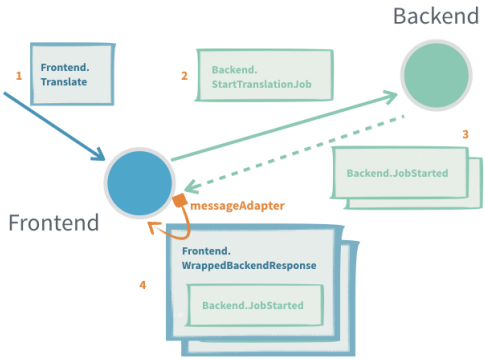

Figure: Adapted Response interaction

- Most often the sending actor does not, and **should not support receiving the response messages of another actor**.
- In such cases we need to provide an `ActorRef` of the right type and **adapt** the response message to a **type** that the sending actor can handle.

Adapted Response

Backend

```
public class Backend {
  public interface Request {}

  public static class BackendRequestA implements Request {
    ...
  }

  public static class BackendRequestB implements Request {
    ...
  }

  public interface Response {}

  public static class BackendResponseA implements Response {
    ...
  }

  public static class BackendResponseB implements Response {
    ...
  }
}
```

Adapted Response

Frontend

```
public class Frontend {

  public interface Command {}

  public static class Translate implements Command {
    ...
  }

  private static class WrappedBackendResponse implements Command {
    final Backend.Response response;

    public WrappedBackendResponse(Backend.Response response) {
      this.response = response;
    }
  }
```

Adapted Response

Translator

```
public static class Translator extends AbstractBehavior<Command> {
  private final ActorRef<Backend.Request> backend;
  private final ActorRef<Backend.Response> backendResponseAdapter;

  public Translator(ActorContext<Command> context, ActorRef<Backend.Request> backend) {
    super(context);
    this.backend = backend;
    // create an adapter, which translates a Backend.Response message
    // into a WrappedBackendResponse and send it to self (Translator)
    this.backendResponseAdapter =
        context.messageAdapter(Backend.Response.class, WrappedBackendResponse::new);
  }

  @Override
  public Receive<Command> createReceive() {
    return newReceiveBuilder()
      .onMessage(Translate.class, this::onTranslate)
      // this is the message which the messageAdapter will send to the self (Translator)
      .onMessage(WrappedBackendResponse.class, this::onWrappedBackendResponse)
      .build();
  }
```

Adapted Response

Translator

```
  private Behavior<Command> onTranslate(Translate cmd) {
    // IMPORTANT: pass the backendResponseAdapter to the receiver so it can reply
    // with its own protocoll
    backend.tell(new Backend.BackendRequestA(backendResponseAdapter);
    return this;
  }

  private Behavior<Command> onWrappedBackendResponse(WrappedBackendResponse wrapped) {
    Backend.Response response = wrapped.response;
    if (response instanceof Backend.BackendResponseA) {
      // handle BackendResponseA ...

    } else if (response instanceof Backend.BackendResponseB) {
        // handle BackendResponseA ...

    } else {
      return Behaviors.unhandled();
    }

    return this;
  }
}
```

Adapted Response

Useful when:

- **Translating** between different actor message protocols.
- **Subscribing** to an actor that will send many response messages back.

Problems:

- It is **hard** to detect that a message request was not delivered or processed (see *Request-Response with ask between two actors*).
- Only **one** adaption can be made per response message type, if a new one is registered the old one is replaced, for example different target actors can't have different adaption if they use the same response types, unless some correlation is encoded in the messages.
- Unless the protocol already includes a way to provide **context**, for example a request id that is also sent in the response, it is **not possible** to tie an interaction to some specific context without introducing a **new**, separate, actor.

Request-Response with ask between two Actors

Request-Response with ask between two Actors

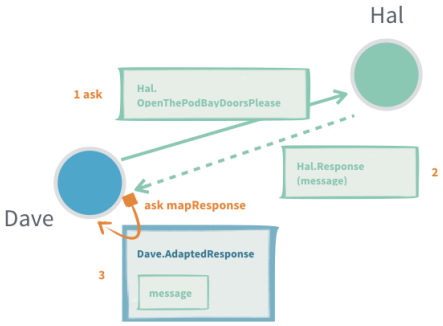

Figure: Request-Response with ask between Actors.

- In an interaction where there is a **1:1 mapping** between a request and a response we can use `ask` on the `ActorContext` to interact with another actor.
- First we need to construct the outgoing message, using the `ActorRef` to put as recipient in the outgoing message.
- The second step is to **transform** the successful response or failure into a message that is part of the protocol of the sending actor.
- The response adapting function is running in the receiving actor and can **safely** access its **state**, but if it throws an exception the actor is stopped.

Request-Response with ask between two Actors

`ActorContext.ask` is possible wherever the `ActorContext` is available

```
final Duration timeout = Duration.ofSeconds(3);

context.ask(
  ResponseMessageClass.class, // response class from target actor
  targetActorRef, // target actor
  timeout,
  // construct the outgoing message
  (ActorRef<ResponseMessageClass> ref) -> new ResponseMessageClass(ref),
  // adapt the response (or failure to respond)
  (response, throwable) -> {
    if (response != null) {
      return new AdaptedResponseToSelf(response.message);
    } else {
      return new AdaptedResponseToSelf("Request failed");
    }
  });

@Override
public Receive<Command> createReceive() {
  return newReceiveBuilder()
      // the adapted message ends up being processed like any other message sent to the actor
      .onMessage(AdaptedResponseToSelf.class, this::onAdaptedResponse)
      .build();
}

private Behavior<Command> onAdaptedResponse(AdaptedResponseToSelf response) {
  getContext().getLog().info("Got adapted response: ", response.message);
  return this;
}
```

Request-Response with ask between two Actors

Useful when:

- Single response queries.
- An actor needs to know that the message was **processed before continuing**.
- To allow an actor to **resend** if a timely response is not produced.
- To **keep track** of outstanding requests and **not** overwhelm a recipient with messages ("backpressure").
- **Context** should be attached to the interaction but the protocol does not support that (request id, what query the response was for).

Request-Response with ask between two Actors

Problems:

- There can only be a **single** response to one ask (see *Per session child Actor*).
- When `ask` times out, the **receiving** actor does **not** know and may still process it to completion, or even start processing it after the fact.
- Finding a **good** value for the **timeout**, especially when `ask` triggers chained `asks` in the receiving actor. You want a short timeout to be responsive and answer back to the requester, but at the same time you do not want to have many false positives.

Request-Response with ask from outside an Actor

Request-Response with ask from outside an Actor

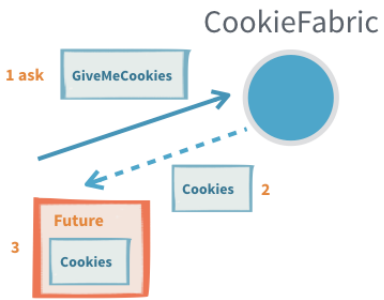

Figure: Request-Response with ask from outside an Actor.

- Sometimes you need to interact with actors from the **outside** of the actor system.
- This can be done with *fire-and-forget* as described in the `Hello World` example or through the **external ask pattern** from the `Factorial` example.
- The external ask returns a `CompletionStage<Response>` that is either completed with a successful response or failed with a `TimeoutException` if there was no response within the specified timeout.

Request-Response with ask from outside an Actor

External Ask Pattern

```
ActorSystem<Void> system = ...;
ActorRef<Command> targetActor = ...;

CompletionStage<CookieFabric.Reply> result =
    AskPattern.ask(
        // can also use the system => sends the message to the guardian top-level actor
        targetActor,
        replyTo -> new RequestMessage(replyTo),
        // asking someone requires a timeout and a scheduler, if the timeout hits without
        // response the ask is failed with a TimeoutException
        Duration.ofSeconds(3),
        system.scheduler());

result.whenComplete(
    (reply, failure) -> {
      if (reply instanceof CommandA)
        System.out.println("Received CommandA");
      else if (reply instanceof CommandB)
        System.out.println("Received CommandA");
      else
        System.out.println("Failed: " + failure);
    });
```

Request-Response with ask from outside an Actor

Useful when:

- Querying an actor from **outside** of the actor system.

Problems:

- It is easy to **accidentally** close over and unsafely **mutate** state with the callbacks on the returned `CompletionStage` as those will be executed on a different thread.
- There can only be a **single** response to one ask (see *Per session child Actor*).
- When `ask` times out, the **receiving** actor does not know and may still process it to completion, or even start processing it after the fact.

Ignoring Replies

Ignoring Replies

In some situations an actor has a response for a particular request message but you are **not interested in the response**.

In this case you can pass `system.ignoreRef()` turning the request-response into a *fire-and-forget*. `system.ignoreRef()`, as the name indicates, returns an `ActorRef` that ignores any message sent to it.

```
targetActor.tell(new RequestCommandWithResponseRef("SomeData", context.getSystem().ignoreRef()));
```

Ignoring Replies

Useful when:

- Sending a message for which the protocol defines a reply, but you are **not interested** in getting the reply.

Problems:

- The returned `ActorRef` **ignores all** messages sent to it, therefore it should be used carefully.
- **Passing it around** inadvertently as if it was a normal `ActorRef` may result in broken actor-to-actor interactions.
- Using it when performing an `ask` from outside the Actor System will cause the `CompletionStage` returned by the `ask` to **timeout** since it will **never** complete.
- Finally, it's legal to `watch` it, but since it's of a special kind, it **never** terminates and therefore you will **never** receive a `Terminated` signal from it.

Send Future result to self

Send Future result to self

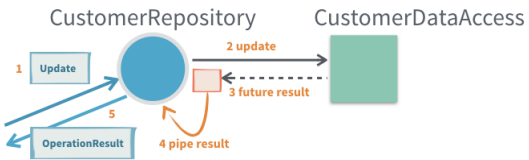

Figure: Send Future result to self.

- When using an API that returns a `CompletionStage` from an actor it's common that you would like to use the **value of the response** in the actor when the `CompletionStage` is **completed**.
- For this purpose the `ActorContext` provides a `pipeToSelf` method,

Send Future result to self

Using `pipeToSelf`

It could be **tempting** to just use a callback on the CompletionStage, but that introduces the **risk** of accessing internal **state** of the actor that is **not thread-safe** from an **external** thread. Therefore it is better to **map** the result to a **message** and perform further processing when receiving that message.

```
private Behavior<Command> onUpdate(Command cmd) {
  CompletionStage<ServiceData> futureResult = someservice.someoperation(cmd.data);
  getContext()
      .pipeToSelf(
          futureResult,
          (ok, exc) -> {
            if (exc == null)
              return new ResultFailure(exc);
            else
              return new ResultSuccess(ok.data, cmd.data);
          });

  return this;
}

private Behavior<Command> onResultFailure(ResultFailure res) {
    // handle failure...
  return this;
}

private Behavior<Command> onResultSuccess(ResultSuccess res) {
    // handle success...
  return this;
}
```

Send Future result to self

Useful when:

- Accessing APIs that are returning `CompletionStage` from an actor, such as a database or an **external** service.
- The actor needs to **continue** processing when the `CompletionStage` has **completed**.
- Keep **context** from the original request and use that when the `CompletionStage` has **completed**, for example a `replyTo` actor reference.

Problems:

- **Boilerplate** of adding wrapper messages for the results.

Per session child Actor

Per session child Actor

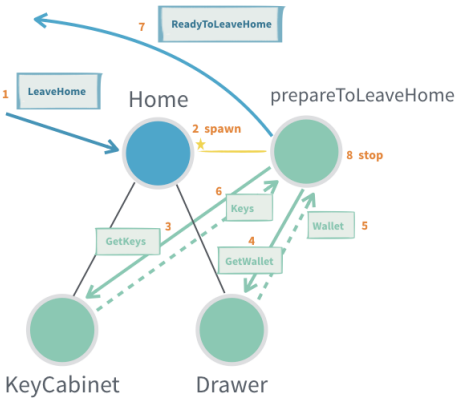

Figure: Per session child Actor interaction.

- In some cases a **complete response** to a request can only be created and sent back after collecting **multiple answers** from other actors.
- For these kinds of interaction it can be good to **delegate** the work to a *per session* child actor.
- This was used in the `Factorial` example, where new actors are **spawned** on-the-fly.
- The protocol of the session actor is **not a public API** but rather an implementation detail of the parent actor: simply make the session actor receive any message (`Object`).

Per session child Actor

Useful when:

- A single incoming request should result in **multiple interactions** with other actors before a result can be built, for example **aggregation** of several results.
- You need to handle **acknowledgement** and **retry** messages for at-least-once delivery.

Problems:

- Children have **life cycles** that must be **managed** to not create a **resource leak**, it can be easy to miss a scenario where the session actor is not stopped.
- It **increases complexity**, since each such child can execute **concurrently** with other children and the parent.

General purpose response Aggregator

General purpose response Aggregator

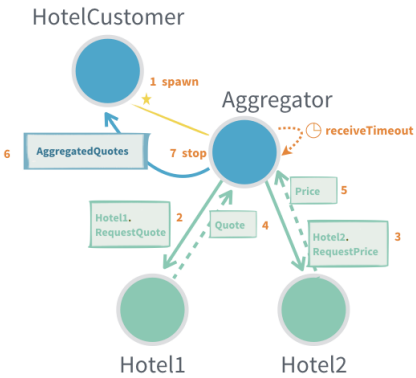

Figure: General purpose response Aggregator.

- Sometimes you might end up **repeating** the same way of **aggregating** replies and want to extract that to a **reusable** actor.
- Technical details are very similar to *Per session child Actor* but an implementation can be developed into a very **general purpose Aggregator** (this goes beyond the scope of this course, and we refer interested students to `https://doc.akka.io/docs/akka/current/typed/interaction-patterns.html`).

General purpose response Aggregator

Useful when:

- **Aggregating** replies are performed in the same way at multiple places and should be extracted to a more **general purpose actor**.
- A single incoming request should result in **multiple interactions** with other actors before a result can be built, for example **aggregation** of several results.
- You need to handle **acknowledgement** and **retry** messages for at-least-once delivery.

Problems:

- Message protocols with generic types are **difficult** since the generic types are **erased** in runtime.
- Children have **life cycles** that must be **managed** to not create a resource leak, it can be easy to miss a scenario where the session actor is not stopped.
- It **increases complexity**, since each such child can execute **concurrently** with other children and the parent.

Latency tail chopping

Latency tail chopping

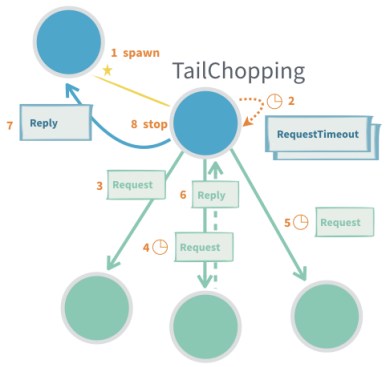

Figure: Latency tail chopping interaction.

- The goal of this algorithm is to **decrease** tail latencies ("chop off the tail latency") in situations where multiple destination actors can perform the same piece of work, and where an actor may occasionally respond more **slowly** than expected.
- In this case, sending the same work request (also known as a "backup request") to **another actor** results in better response time - because it's less probable that multiple actors are under heavy load **simultaneously**.

Latency tail chopping

Useful when:

- **Reducing** higher **latency** percentiles and **variations** of latency are important.
- The "work" can be done more than **once** with the **same result**, e.g. a request to retrieve information.

Problems:

- **Increased load** since more messages are sent and "work" is performed more than once.
- Can't be used when the "work" is **not idempotent** and must only be performed once.
- Message protocols with generic types are difficult since the generic types are erased in runtime.
- Children have life cycles that must be managed to not create a resource leak, it can be easy to miss a scenario where the session actor is not stopped.

Scheduling messages to self

Scheduling messages to self

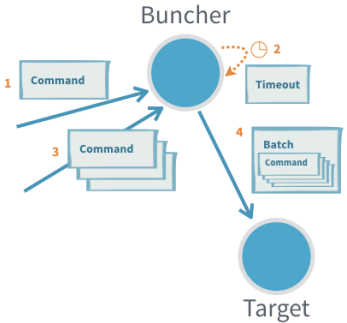

Figure: Timer interaction.

- Sometimes it is useful to **schedule** messages with a **delay** to self.
- To be able to **schedule** messages with a **delay**, access to a `TimeScheduler` is needed. This can be aquired upon `Behavior` creation using `Behaviors.withTimers`.
- It is also possible to schedule **periodically**.

Scheduling messages to self

Accessing the `TimerScheduler`

```
class SchedulerBehavior extends AbstractBehavior<Command> {
  private final TimerScheduler<Command> timers;

  public static Behavior<Command> create() {
    return Behaviors.setup(ctx -> Behaviors.withTimers(timers -> new SchedulerBehavior(ctx, timers)));
  }

  private SchedulerBehavior(ActorContext<Command> context, TimerScheduler<Command> timers) {
    super(context);
    this.timers = timers;
    this.timers.startTimerAtFixedRate(new Command(), Duration.ofSeconds(10));
  }
```

Scheduling messages to self

Handling Timeout Messages

```
private static final Object TIMER_KEY = new Object();

  // We use an enum as timeout message as it does not carry any
  // parameters and only indicates the passing of the time
  private enum Timeout implements Command {
    INSTANCE
  }

  @Override
  public Receive<Command> createReceive() {
    return newReceiveBuilder()
        .onMessageEquals(Timeout.class, this::onTimeout)
        .onMessage(Command.class, this::onCommand)
        .build();
  }

  private Behavior<Command> onCommand(Command msg) {
    Duration delay = Duration.ofSeconds(1);
    // Each timer has a key and if a new timer with the same key is started,
    // the previous is cancelled. It is guaranteed that a message from the
    // previous timer is not received, even if it was already enqueued in
    // The mailbox when the new timer was started.
    timers.startSingleTimer(TIMER_KEY, Timeout.INSTANCE, delay);
    return this;
  }

  private Behavior<Command> onTimeout() {
    // handle timeout message
    return this;
  }
```

Scheduling messages to self

Fixed-Rate

Use `startTimerAtFixedRate` if the frequency of execution over time should meet the given interval.

- When using fixed-rate it will **compensate** the delay for a subsequent task if the previous messages were delayed too long.
- Fixed-rate execution is appropriate for recurring activities that are **sensitive** to **absolute** time or where the **total** time to perform a fixed number of executions is important, such as a **countdown** timer that ticks once every second for ten seconds.
- For example, `scheduleAtFixedRate` with an interval of 1 second and the process is suspended for 30 seconds will result in 30 messages being sent in rapid succession to **catch up**.

Scheduling messages to self

Fixed-Delay

Use `startTimerWithFixedDelay` for a delay between sending subsequent messages that will always be (at least) the given delay.

- When using fixed-delay it will **not compensate** the delay between messages if the scheduling is **delayed** longer than specified for some reason.
- Fixed-delay execution is appropriate for **recurring** activities that require "smoothness".
- It is appropriate for **activities** where it is more important to keep the **frequency** accurate in the **short run** than in the long run.

Best Practices

Best Practices

Well behaved

Actors should do their job **efficiently** without bothering everyone else needlessly and **avoid hogging resources**. This means they should process events and generate responses (or more requests) in an event-driven manner. Actors should **not block** (i.e. passively wait while occupying a Thread) on some external entity - which might be a lock, a network socket,...

Best Practices

No sharing of mutable state

Do not pass mutable objects between actors. In order to ensure that, prefer **immutable** messages. If the encapsulation of actors is broken by exposing their mutable state to the outside, you are back in **normal** Java concurrency land with all the **drawbacks**.

Best Practices

Hierarchical Systems

The top-level actor of the actor system is the **innermost** part of your Error Kernel. It should only be responsible for **starting** the various sub systems of your application, and not contain much **logic** in itself, prefering truly **hierarchical** systems. This has benefits with respect to fault-handling and it also reduces the **strain** on the guardian actor, which is a single point of **contention** if over-used.