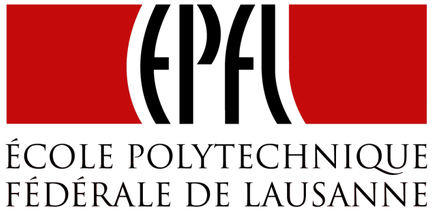

Akka Typed Facilities

Programming Reactive Systems

Roland Kuhn

Speaking multiple protocols

Recall: Modeling protocols with algebraic data types

```
case class RequestQuote(title: String, buyer: ActorRef[Quote])

case class Quote(price: BigDecimal, seller: ActorRef[BuyOrQuit])

sealed trait BuyOrQuit
case class Buy(address: Address, buyer: ActorRef[Shipping]) extends BuyOrQuit
case object Quit extends BuyOrQuit

case class Shipping(date: Date)
```

Akka Typed Adapters

```
sealed trait Secretary
case class BuyBook(title: String,
                   maxprice: BigDecimal,
                   seller: ActorRef[RequestQuote]) extends Secretary

def secretary(address: Address): Behavior[Secretary] =
    Behaviors.receiveMessage {
        case BuyBook(title, maxPrice, seller) =>
            seller ! RequestQuote(title, ???)
            ...
    }
```

Akka Typed Adapters

Solution: wrap foreign protocol to process it

```
sealed trait Secretary
case class BuyBook(title: String,
                   maxprice: BigDecimal,
                   seller: ActorRef[RequestQuote]) extends Secretary
case class QuoteWrapper(msg: Quote) extends Secretary
case class ShippingWrapper(msg: Shipping) extends Secretary
```

Akka Typed Adapters

Use `ActorContext.messageAdapter` to create an `ActorRef`:

```
def secretary(address: Address): Behavior[Secretary] =
    Behaviors.receivePartial {
        case (ctx, BuyBook(title, maxPrice, seller)) =>
            val quote: ActorRef[Quote] = ctx.messageAdapter(QuoteWrapper)
            seller ! RequestQuote(title, quote)
            buyBook(maxPrice, address)
    }
```

Akka Typed Adapters

```
def buyBook(maxPrice: BigDecimal, address: Address): Behavior[Secretary] =
    Behaviors.receivePartial {
        case (ctx, QuoteWrapper(Quote(price, session))) =>
            if (price > maxPrice) {
                session ! Quit // Nay, too expensive.
                Behaviors.stopped
            } else {
                val shipping = ctx.messageAdapter(ShippingWrapper)
                session ! Buy(address, shipping)
                Behaviors.same
            }
        case (ctx, ShippingWrapper(Shipping(date))) =>
            Behaviors.stopped // Yay, a book has been bought!
    }
```

Alternative: declare messages for protocol participants

The protocol definition can also attach roles to messages:

- `RequestQuote` and `BuyOrQuit` extend `BuyerToSeller`
- `Quote` and `Shipping` extend `SellerToBuyer`

This allows more concise message wrappers, e.g.

```
case class WrapFromSeller(msg: SellerToBuyer) extends Secretary
```

Child Actors for protocol sessions

Child Actors for protocol sessions

```
case class BuyBook(title: String, maxprice: BigDecimal,
                   seller: ActorRef[RequestQuote]) extends Secretary
case class Bought(shippingDate: Date) extends Secretary
case object NotBought extends Secretary

def secretary(address: Address): Behavior[Secretary] =
  Behaviors.receive {
    case (ctx, BuyBook(title, maxPrice, seller)) =>
      val session = ctx.spawnAnonymous(buyBook(maxPrice, address, ctx.self))
      seller ! RequestQuote(title, session)
      ctx.watchWith(session, NotBought)
      Behaviors.same
    case (ctx, Bought(shippingDate)) => Behaviors.stopped
    case (ctx, NotBought)            => Behaviors.stopped
  }
```

Child Actors for protocol sessions

```
def buyBook(maxPrice: BigDecimal, address: Address, replyTo: ActorRef[Bought]) =
    Behaviors.receive[SellerToBuyer] {
        case (ctx, Quote(price, session)) =>
            if (price > maxPrice) {
                session ! Quit
                Behaviors.stopped
            } else {
                session ! Buy(address, ctx.self)
                Behaviors.same
            }
        case (ctx, Shipping(date)) =>
            replyTo ! Bought(date)
            Behaviors.stopped
    }
```

Defer handling a message

The order of message reception sometimes is not deterministic but the actor needs to process a particular message first.

Solution: stash messages in a buffer.

Defer handling a message

The order of message reception sometimes is not deterministic but the actor needs to process a particular message first.

Solution: stash messages in a buffer.

```
val initial = Behaviors.setup[String] { ctx =>
    val buffer = StashBuffer[String](100)

    Behaviors.receiveMessage {
        case "first" =>
            buffer.unstashAll(ctx, running)
        case other =>
            buffer.stash(other)
            Behaviors.same
    }
}
```

Type-safe service discovery

Actor A provides protocol P.
Actor B needs to speak with an actor implementing protocol P.

- in a local system dependencies can be injected by first creating A and then pass an `ActorRef[P]` to B
- dependency graph can become unwieldy
- this approach does not work for a cluster

Type-safe service discovery

Actor A provides protocol P.
Actor B needs to speak with an actor implementing protocol P.

- in a local system dependencies can be injected by first creating A and then pass an `ActorRef[P]` to B
- dependency graph can become unwieldy
- this approach does not work for a cluster

Solution: cluster-capable well-known service registry at each `ActorSystem`:

```
val ctx: ActorContext[_] = ???
ctx.system.receptionist: ActorRef[Receptionist.Command]
```

Registering a service provider

Create a serializable cluster-wide identifier for the protocol:

```
val key = ServiceKey[Greeter]("greeter")
```

Registering a service provider

Create a serializable cluster-wide identifier for the protocol:

```
val key = ServiceKey[Greeter]("greeter")
```

Obtain an `ActorRef[Greeter]`, for example by creating an actor:

```
val greeter = ctx.spawn(Greeter.behavior, "greeter")
```

Registering a service provider

Create a serializable cluster-wide identifier for the protocol:

```
val key = ServiceKey[Greeter]("greeter")
```

Obtain an `ActorRef[Greeter]`, for example by creating an actor:

```
val greeter = ctx.spawn(Greeter.behavior, "greeter")
```

Register the reference with the receptionist:

```
ctx.system.receptionist ! Register(key, greeter)
```

Looking up a service implementation

Assuming a friendly actor like this one:

```
sealed trait FriendlyCommand
case class Intro(friend: String) extends FriendlyCommand
case class SetGreeter(listing: Listing) extends FriendlyCommand
```

Looking up a service implementation

Assuming a friendly actor like this one:

```
sealed trait FriendlyCommand
case class Intro(friend: String) extends FriendlyCommand
case class SetGreeter(listing: Listing) extends FriendlyCommand
```

When starting, first query the service registry:

```
val friendly = Behaviors.setup[FriendlyCommand] { ctx =>
    val receptionist = ctx.system.receptionist
    val listingRef = ctx.messageAdapter(SetGreeter)
    receptionist ! Find(key, listingRef)

    ...
}
```

Looking up a service implementation

```
val friendly = Behaviors.setup[FriendlyCommand] { ctx =>
    ...
    val buffer = StashBuffer[FriendlyCommand](100)

    Behaviors.receiveMessage {
        case SetGreeter(key.Listing(refs)) if refs.isEmpty =>
            ctx.schedule(3.seconds, receptionist, Find(key, listingRef))
            Behaviors.same
        case SetGreeter(key.Listing(refs)) =>
            buffer.unstashAll(ctx, friendlyRunning(refs.head))
        case other =>
            buffer.stash(other)
            Behaviors.same
    }
}
```

Summary

In this video we have seen:

- using adapters to incorporate other protocols into an actor's type
- stashing messages for later consumption to compensate for non-deterministic ordering
- service discovery with the receptionist