

Actor Composition

Principles of Reactive Programming

Roland Kuhn

The Type of an Actor

The interface of an Actor is defined by its accepted message types, the type of an Actor is structural.

This structure may change over time defined by a protocol.

The Type of an Actor

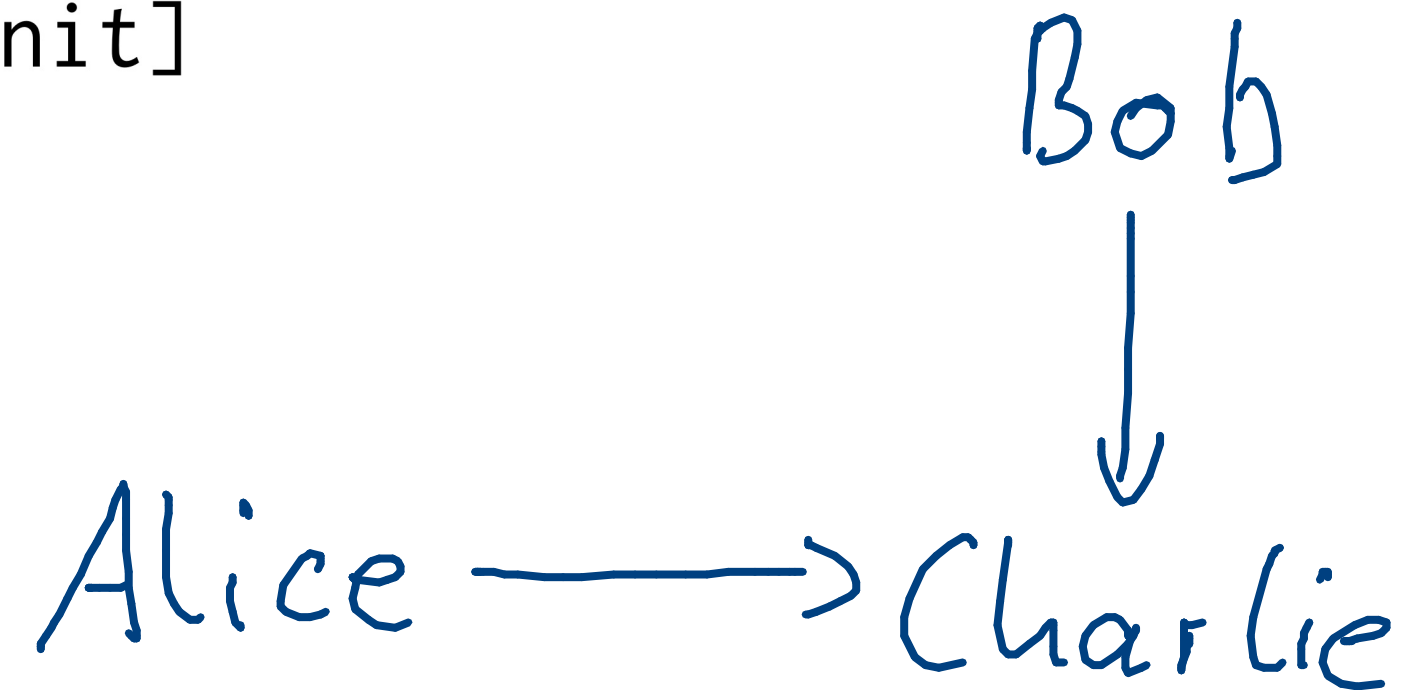
The interface of an Actor is defined by its accepted message types, the type of an Actor is structural.

This structure may change over time defined by a protocol.

Superficially current Actor implementations are untyped:

- ▶ sending a message is `(Any => Unit)`
- ▶ behavior is `PartialFunction[Any, Unit]`

This limitation is not a fundamental.



Actor Composition

Actor Systems are composed like human organizations.

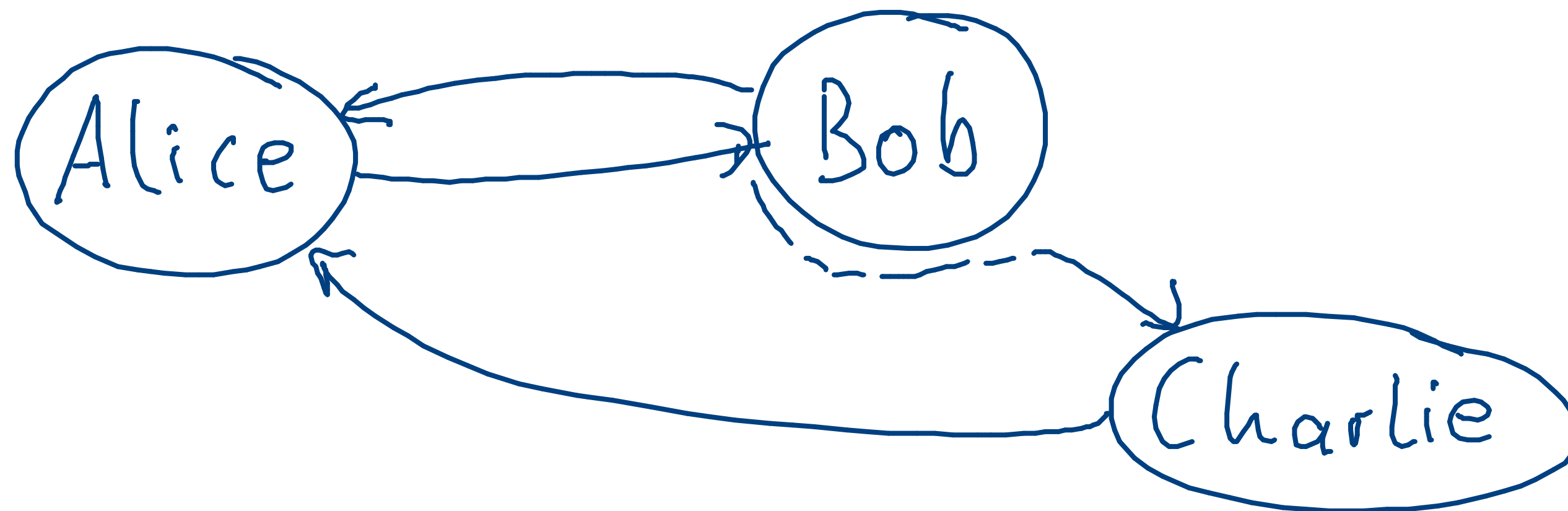
Actors are composed on a protocol level.

An Actor can

- ▶ translate and forward requests
- ▶ translate and forward replies
- ▶ split up requests and aggregate replies

The Customer Pattern

- ▶ fundamental request–reply pattern
- ▶ customer address included in the (original) request
- ▶ allows dynamic composition of actor systems



Interceptors

```
class AuditTrail(target: ActorRef) extends Actor with ActorLogging {  
  def receive = {  
    case msg =>  
      log.info("sent {} to {}", msg, target)  
      target forward msg  
  }  
}
```

A one-way proxy does not need to keep state.

The Ask Pattern

```
import akka.pattern.ask

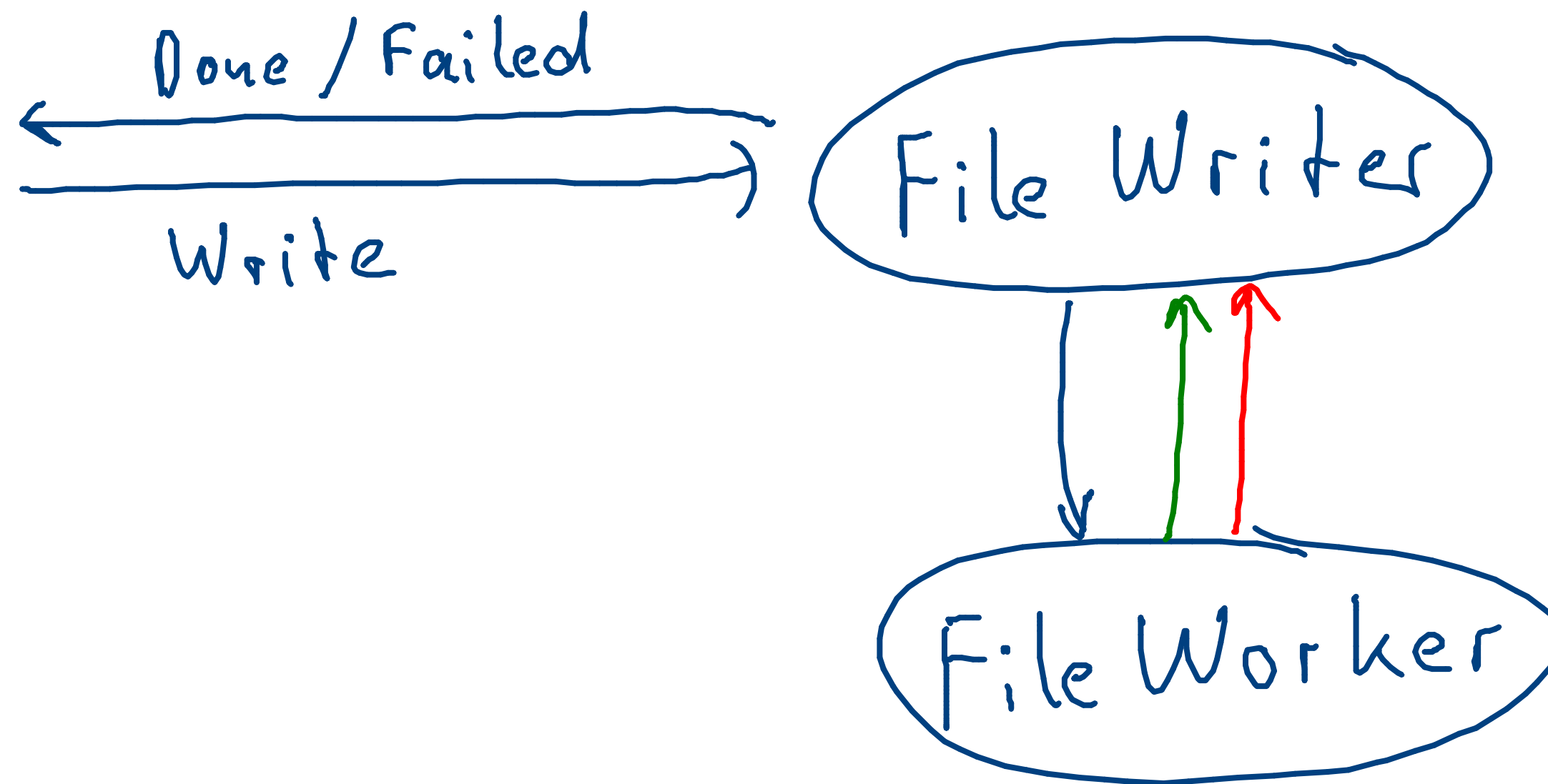
class PostsByEmail(userService: ActorRef) extends Actor {
  implicit val timeout = Timeout(3.seconds)
  def receive = {
    case Get(email) =>
      (userService ? FindByEmail(email)).mapTo[UserInfo]
        .map(info => Result(info.posts.filter(_.email == email)))
        .recover { case ex => Failure(ex) }
        .pipeTo(sender)
  }
}
```

Result Aggregation

```
class PostSummary(...) extends Actor {  
  implicit val timeout = Timeout(500.millis)  
  def receive = {  
    case Get(postId, user, password) =>  
      val response = for {  
        status <- (publisher ? GetStatus(postId)).mapTo[PostStatus]  
        text    <- (postStore ? Get(postId)).mapTo[Post]  
        auth    <- (authService ? Login(user, password)).mapTo[AuthStatus]  
      } yield  
        if (auth.successful) Result(status, text)  
        else Failure("not authorized")  
      response pipeTo sender  
  }  
}
```

Risk Delegation

- ▶ create subordinate to perform dangerous task
- ▶ apply lifecycle monitoring
- ▶ report success/failure back to requestor
- ▶ ephemeral actor shuts down after each task



Example: File Writer

```
class FileWriter extends Actor {  
  var workerToCustomer = Map.empty[ActorRef, ActorRef]  
  override val supervisorStrategy = SupervisorStrategy.stoppingStrategy  
  def receive = {  
    case Write(contents, file) =>  
      val worker = context.actorOf(Props(new FileWorker(contents, file, self)))  
      context.watch(worker)  
      workerToCustomer += worker -> sender  
    case Done => workerToCustomer.get(sender).foreach(_ ! Done)  
                workerToCustomer -= sender  
    case Terminated(worker) => workerToCustomer.get(worker).foreach(_ ! Failed)  
                                workerToCustomer -= worker  
  }  
}
```

Façade

- ▶ translation
- ▶ validation
- ▶ rate limitation
- ▶ access control