Responsiveness

Principles of Reactive Programming

Roland Kuhn

Responsiveness

Responsiveness is the ability to respond to input in time.

A system which does not respond in time is not available.

Responsiveness

Responsiveness is the ability to respond to input in time.

A system which does not respond in time is not available.

The goal of resilience is to be available.

Responsiveness implies resilience to overload scenarios.

Exploit Parallelism

Performing queries sequentially adds up latency:

```
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
  }
}
```

Exploit Parallelism

Performing queries in parallel reduces latency:

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

Load vs. Responsiveness

When incoming request rate rises, latency typically rises.

- Avoid dependency of processing cost on load.
- Add parallelism elastically (resizing routers).

When the rate exceeds the system's capacity requests will pile up:

- Processing gets backlogged.
- Clients timeout, leading to unnecessary work being performed.

Circuit Breaker

```
class Retriever(userService: ActorRef) extends Actor {
  implicit val timeout = Timeout(2.seconds)
  val cb = CircuitBreaker(context.system.scheduler,
    maxFailures = 3,
    callTimeout = 1.second,
    resetTimeout = 30.seconds)

  def receive = {
    case Get(user) =>
      val result = cb.withCircuitBreaker(userService ? user).mapTo[String]
      ...
  }
}
```

Bulkheading (1)

Separate computing intensive parts from client-facing parts.

Actor isolation is not enough: execution mechanism is still shared.

Dedicate disjoint resources to different parts of the system.

```
Props[MyActor].withDispatcher("compute-jobs")
```

Bulkheading (2)

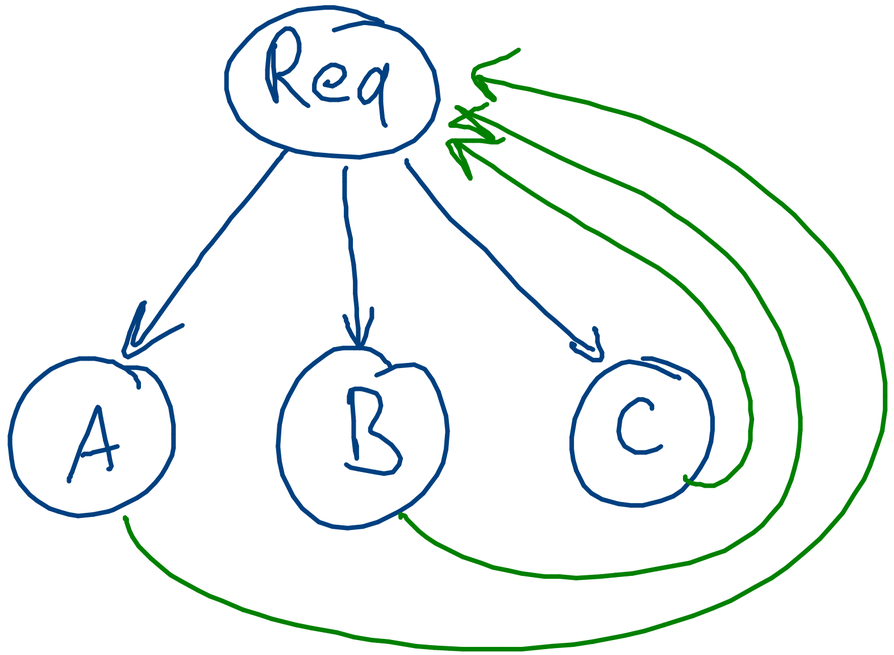
```
akka.actor.default-dispatcher {
  executor = ”fork-join-executor”
  fork-join-executor {
    parallelism-min = 8
    parallelism-max = 64
    parallelism-factor = 3.0
  }
}

compute-jobs.fork-join-executor {
  parallelism-min = 4
  parallelism-max = 4
}
```

Failures vs. Responsiveness

Detecting failure takes time, usually a timeout.

Immediate fail-over requires the backup to be readily available.

Instant fail-over is possible in active–active configurations.



Summary

- Event-driven systems can be scaled horizontally and vertically
- Responsiveness demands resilience and scalability