DESIGNING A REACTIVE DATA PLATFORM: CHALLENGES, PATTERNS AND ANTI-PATTERNS

Alex Silva
Distributed
Elastic
Location Agnostic
Open
Message Driven
Self-Healing
REACTIVE
The Reactive Manifesto

Responsive

Elastic

Message Driven

Resilient
Elasticity
Scaling OUT vs Scaling UP

Scaling OUT

Scaling UP
Elasticity

Asynchronous
Share Nothing
Divide and Conquer
Location Transparency
Synchronous Messaging

Inherit ordering introduces implicit back pressure on the sender.
Asynchronous Messaging

1

2

Invalid!

3

Asynchronous

Asynchronous Messaging
“The ability of something to return to its original shape, after it has been pulled, stretched, pressed, or bent.”

Merriam-Webster
What about software systems?

Our Disaster Recovery Plan Goes Something Like This...

HELP! HELP!

I'm not stupid, of course I've got a disaster recovery plan! Unfortunately it was on my hard drive when it crashed.
private void foo() throws ApplicationLaunchException {

    URI uri = URI.create("some url");
    Intent intent = new Intent(uri);

    try {
        startActivity(intent.getJson());
    } catch (JSONException anfe) {
        URI weburi = URI.create("another url");
        try {
            Intent webintent = new Intent(weburi);
            startActivity(webintent.getJson());
        } catch (Exception e) {
            throw new ApplicationLaunchException("Something went wrong");
        }
    }

    catch (SQLException e) {
        // the database is down
        try {
            reestablishConnection(true);
        } catch (SQLException ex) {
            throw new ApplicationLaunchException(ex.getMessage());
        }
    }
}
KEEP CALM AND LET IT CRASH
WHAT IF TOLD YOU

IT IS COMPLEX BUT
NOT THAT COMPLICATED
Software Systems are Complex Systems
“Complex systems run in degraded mode.”

“Complex systems run as broken systems.”

Richard Cook
Asynchronous Communication + Eventual Consistency → Resilient Protocols
Failures

- Contained
- Managed
- Observed
- Reified as messages
Messages vs Events

Messages

Events

Specific
Addressable
Topic
Facts
Past
REAL-TIME DATA INGESTION PLATFORM
Why Akka?

- Reactive
- Elastic
- Fault Tolerant
- Load Management
- Location Transparency

Both up and out
Akka Actors

- Lightweight
- Reactive
- Asynchronous
- Resilient
Challenges with Akka

- Learning Curve
- Type Safety
- Debugging
- Dead Letters
Why Kafka?

- Distributed Log
- High Throughput
- Replicated
- Concurrency
Why Spark?

- Fast!
- Unified Platform
- Functional Paradigm
- Rich Library Set
- Active Community
PATTERNS AND ANTI-PATTERNS
GOOD PRACTICE:

DECENTRALIZE THE PROCESSING OF KEY TASKS
MESSAGE HANDLERS
Hydra Ingestion Flow
Handler Registry

Monitors registered handlers for errors/stops

Broadcasts messages

Handler Lifecycle
GOOD PRACTICE:

DESIGN AN INCREMENTAL COMMUNICATION PROTOCOL
Hydra Ingestion Protocol

Publish

Join

Validate

Valid

Ingest

Invalid

STOP

<<Silence>>

MESSAGE HANDLERS
Hey guys!
Check this out!

Huh?!
Nice!
Bring it!
Nah...

Hydra Ingestion Protocol: Publish

Handler Registry

Publish

Message handlers

Join

Join
Hydra Ingestion Protocol: Validation

HOW DOES IT LOOK?

Ingestion Coordinator

Validate

GOOD!

Valid

BAD!

Invalid

Message handlers
Hydra Ingestion Protocol: Invalid Message

Ingestion Coordinator

GOT A BAD ONE

Error Reporter

Ingest

ReportError
Hydra Ingestion Protocol: Ingest

for each handler

Ingest

Encode

Persistence to Kafka

SHIP IT!

JSON
abstract class BaseMessageHandler extends Actor with ActorConfigSupport with ActorLogging with IngestionFlow with ProducerSupport with MessageHandler {

  ingest {
    case Initialize => {
      //nothing required by default
    }
    case Publish(request) => {
      log.info(s"Publish message was not handled by \${self}. Will not join."")
    }
    case Validate(request) => {
      sender ! Validated
    }
    case Ingest(request) => {
      log.warning("Ingest message was not handled by \${self}.")
      sender ! HandlerCompleted
    }
    case Shutdown => {
      //nothing required by default
    }
    case Heartbeat => {
      Health.get(self).getChecks
    }
  }
}
GOOD PRACTICE:

HIDE AN ELASTIC POOL OF RESOURCES BEHIND ITS OWNER
Less of this...
More of this!
akka {
  actor {
    deployment {
      /services-manager/handler_registry/segment_handler {
        router = round-robin-pool
        optimal-size-exploring-resizer {
          enabled = on
          action-interval = 5s
          downsize-after-underutilized-for = 2h
        }
      }
    }
    /services-manager/kafka_producer {
      router = round-robin-pool
      resizer {
        lower-bound = 5
        upper-bound = 50
        messages-per-resize = 500
      }
    }
  }
}
akka {
  actor {
    deployment {
      /services-manager/handler_registry/segment_handler {
        router = round-robin-pool
        optimal-size-exploring-resizer {
          enabled = on
          action-interval = 5s
          downsize-after-underutilized-for = 2h
        }
      }
    }
    provider = "akka.cluster.ClusterRefActorProvider"
  }
  cluster {
    seed-nodes = ["akka.tcp://Hydra@127.0.0.1:2552","akka.tcp://hydra@172.0.0.1:2553"]
  }
}
GOOD PRACTICE:

USE SELF-DESCRIBING MESSAGES
trait HydraRequest {
  def payload: String
  def params: Map[String, String]
  def param(name: String): Option[String]

  // Analogous to HTTP headers
  def metadata: List[HydraRequestMetadata]

  def metadataValue(name: String): Option[String] = {
    metadata.find(m => m.name == name) match {
      case Some(md) => Some(md.value)
      case None => None
    }
  }
}

case class HydraRequestMetadata(name: String, value: String)
private def commitOffsets(offsets: Map[TopicAndPartition, Long], coordinator: Broker, 
config: SimpleConsumerConfig, correlationId: Int): Boolean = {
val offsetsMetadata = offsets.map { case (k, v) => (k, OffsetAndMetadata(v)) }
val commitRequest = OffsetCommitRequest{
  groupId = config.groupId,
  requestInfo = offsetsMetadata,
  versionId = 1,
  correlationId = correlationId,
  clientId = config.clientId
}
try {
  val commitResponse = simpleConsumer(coordinator, config).commitOffsets(commitRequest)
  if (commitResponse.hasError) {
    ErrorMapping.maybeThrowException(commitResponse.commitStatus.find(_._2 != ErrorMapping.NoError).head._2)
    false
  } else {
    true
  }
} catch {
  case NonFatal(e) => {
    log.error("Unable to commit offsets.", e)
    false
  }
}
trait KafkaMessage[K, P] {
    val timestamp = System.currentTimeMillis
    def key: K
    def payload: P
    def retryOnFailure: Boolean = true
}

case class JsonMessage(key: String, payload: JsonNode) extends KafkaMessage[String, JsonNode]

object JsonMessage {
    val mapper = new ObjectMapper()
    def apply(key: String, json: String) = {
        val payload: JsonNode = mapper.readTree(json)
        new JsonMessage(key, payload)
    }
}

case class AvroMessage(val schema: SchemaHolder, key: String, json: String) extends KafkaMessage[String, GenericRecord] {
    def payload: GenericRecord = {
        val converter: JsonConverter[GenericRecord] = new JsonConverter[GenericRecord](schema.schema)
        converter.convert(json)
    }
}
GOOD PRACTICE:

PREFER BINARY DATA FORMATS FOR COMMUNICATION
Why Avro?

- Binary Format
- Space Efficient
- Evolutionary Schemas
- Automatic Tables
GOOD PRACTICE:

DELEGATE AND SUPERVISE! REPEAT!
Error Kernel
Ingestion Actors: Coordinators

Supervises ingestion at the request level

Coordinates protocol flow

Reports errors and metrics
GOOD PRACTICE:

LET IT CRASH
Let it Crash

Components where full restarts are always ok

Transient failures are hard to find

Simplified failure model
override val supervisorStrategy =
  OneForOneStrategy(maxNrOfRetries = 10, withinTimeRange = 1.minute) {
  case _: ActorInitializationException => akka.actor.SupervisorStrategy.Stop
  case _: FailedToSendMessageException => Restart
  case _: ProducerClosedException => Restart
  case _: NoBrokersForPartitionException => Escalate
  case _: KafkaException => Escalate
  case _: ConnectException => Escalate
  case _: Exception => Escalate
}

val kafkaProducerSupervisor = BackoffSupervisor.props(Backoff.onFailure(
  kafkaProducerProps,
  childName = actorName[KafkaProducerActor],
  minBackoff = 3.seconds,
  maxBackoff = 30.seconds,
  randomFactor = 0.2))
class KafkaProducerActor extends Actor with LoggingAdapter with ActorConfigSupport with NotificationSupport[KafkaMessage[Any, Any]] {

  import KafkaProducerActor._

  implicit val ec = context.dispatcher

  override def preRestart(cause: Throwable, message: Option[Any]) = {
    //send it to itself again after the exponential delays, no Ack from Kafka
    message match {
      case Some(rp: RetryingProduce) => {
        notifyObservers(KafkaMessageNotDelivered(rp.msg))
        val nextBackOff = rp.backOff.nextBackOff
        val retry = RetryingProduce(rp.topic, rp.msg)
        retry.backOff = nextBackOff
        context.system.scheduler.scheduleOnce(nextBackOff.waitTime, self, retry)
      }

      case Some(produce: Produce) => {
        notifyObservers(KafkaMessageNotDelivered(produce.msg))
        if (produce.msg.retryOnFailure) {
          context.system.scheduler.scheduleOnce(initialDelay, self, RetryingProduce(produce.topic, self, produce.msg))
        }
      }
    }
  }
}
Monitoring through Death Watches

```scala
override def receive: Receive = {
  case RegisterHandler(info) => {
    val actor = context.actorOf(info.props(self.path), info.name)
    val handler = handlers.getOrElseUpdate(actor.path.name, actor)
    context.watch(actor)
    sender ! RegisteredHandler(info.name, handler)
  }
  case RemoveHandler(path) => {
    handlers.remove(path.name) match {
      case Some(handler) => {
        context.unwatch(handler)
        context.stop(handler)
      }
      case None => log.info("Handler %s not found", format(path.name))
    }
    case p@Publish(_) => broadcaster.forward(p)
  case Terminated(handler) => {
    log.error(s"Handler $handler terminated.")
  }
```

```
WHAT ABOUT SOME ANTI-PATTERNS?
NOT SO GOOD PRACTICE:

BUILDING NANO SERVICES
Ingestion
Hydra Core
Ingestors
HTTP
Spark (Batch and Streaming)
Hydra Core
Dispatchers
HTTP
RDBMS
HDFS
Persistence :: Kafka
Persistence
Hydra Core
HTTP
Conductors
Hydra Core
Conductors
HTTP
AKKA Remoting
Spark (Batch and Streaming)
Hydra Core
Dispatchers
HTTP
HDFS
RDBMS
Conductors
Hydra Core
Conductors
HTTP
NOT SO GOOD PRACTICE:

TREATING LOCATION TRANSPARENCY AS A FREE-FOR-ALL
Guaranteed Delivery in Hydra

What does guaranteed delivery mean?

At most once semantics

Can be made stronger
Akka Remoting

- Peer-to-Peer
- Serialization
- Delivery Reliability
- Latency
The Reliable Proxy Pattern
@throws(classOf[Exception])
override def init: Future[Boolean] = Future {
  val useProxy = config.getBoolean("message.proxy", false)
  val ingestorPath = config.getRequiredString("ingestor.path")

  ingestionActor =
  if (useProxy) context.actorSelection(ingestorPath) else
  context.actorOf(ReliableIngestionProxy.props(ingestorPath))

  val cHeaders = config.getOptionalList("headers")
  topic = config.getRequiredString("kafka.topic")
  headers = cHeaders match {
    case Some(ch) => List(
      ch.unwrapped.asScala.map {
        header => {
          val sh = header.toString.split(":")
          RawHeader(sh(0), sh(1))
        }
      }:_*
    )
    case None => List.empty[HttpHeader]
  }
  true
}
NOT SO GOOD PRACTICE:

NOT KEEPING MESSAGE PROTOCOL BOUND TO THEIR CONTEXTS
object Messages {

  case object ServiceStarted
  case class RegisterHandler(info: ActorRef)
  case class RegisteredHandler(name: String, handler: ActorRef)
  case class RemoveHandler(path: ActorPath)
  case object GetHandlers
  case object InitiateIngestion extends HydraMessage
  case class RequestCompleted(s: IngestionSummary) extends HydraMessage
  case class IngestionSummary(name: String)
  case class Produce(topic: String, msg: KafkaMessage[_, _], ack: Option[ActorRef]) extends HydraMessage
  case object HandlerTimeout extends HydraMessage
  case class Validate(req: HydraRequest) extends HydraMessage
  case class Validated(req: HydraRequest) extends HydraMessage
  case class NotValid(req: HydraRequest, reason: String) extends HydraMessage
  case object HandlingCompleted extends HydraMessage
  case class Publish(request: HydraRequest)
  case class Ingest(request: HydraRequest)
  case class Join(r: HydraRequest) extends HydraMessage
}

class HandlerRegistry extends Actor with LoggingAdapter with ActorConfigSupport {

    override def receive: Receive = {
        ...
    }

    override val supervisorStrategy = OneForOneStrategy() {
        case e: Exception => {
            report(e)
            Restart
        }
    }

    object HandlerRegistry {

        case class RegisterHandler(info: HandlerInfo)
        case class RegisteredHandler(name: String, handler: ActorRef)
        case class RemoveHandler(path: ActorPath)
        case object GetHandlers
    }
}
NOT SO GOOD PRACTICE:

DEVELOPING OVERLY CHATTY PROTOCOLS
What’s next?
What’s streaming into Hydra today?

Conductors:
- YouBora
- RabbitMQ
- Hackhands

Webhooks:
- Segment.io
- WebEngage
Average Ingestions Per Second

Dec-15

Jan-16

Jan-16

Jan-16

1-Feb

3/1/16

Requests
Some Facts

9,730 lines of Scala code

Production Platform Since Jan 2016

C.I. through Jenkins and Salt
OPEN SOURCE
ALL THE THINGS
QUESTIONS?

Thank You!

http://linkedin.com/in/alexysilva
@thealexsilva