rubiX

Reactive Streams

Beyond the manifesto

László van den Hoek

- Will code for food since 2008
- Rubix/Lightbend partnership instigator
- Currently @ KLPD  building a data platform

Non-blocking processing with event loops

Non-blocking processing with event loops

Fast producer, slow consumer

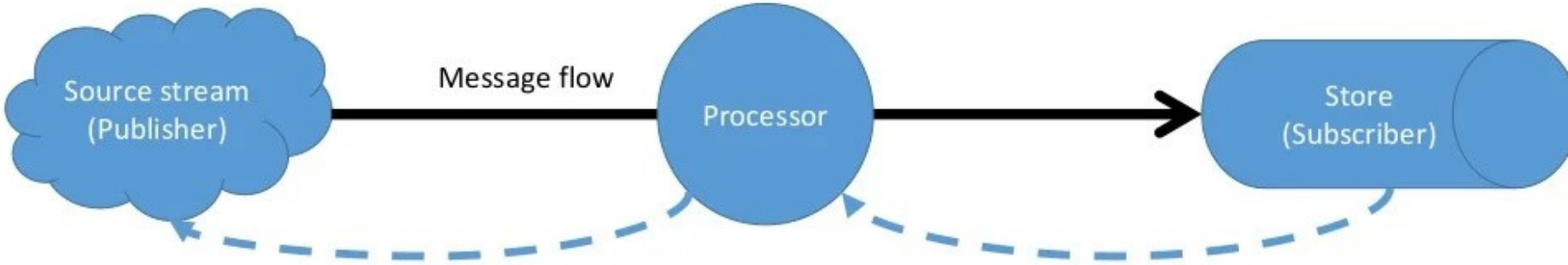
Use cases for managing control flow

- Upstream supply exceeds downstream capacity
  - …constantly, but cannot scale downstream
  - …occasionally; bursty source.
- Messages need to be processed in-order
  - Partitioning and sharding may help, but maybe not enough
- Efficient use of resources
  - Provision for what you need, not for what you (could) get
- Concrete example: sensor data

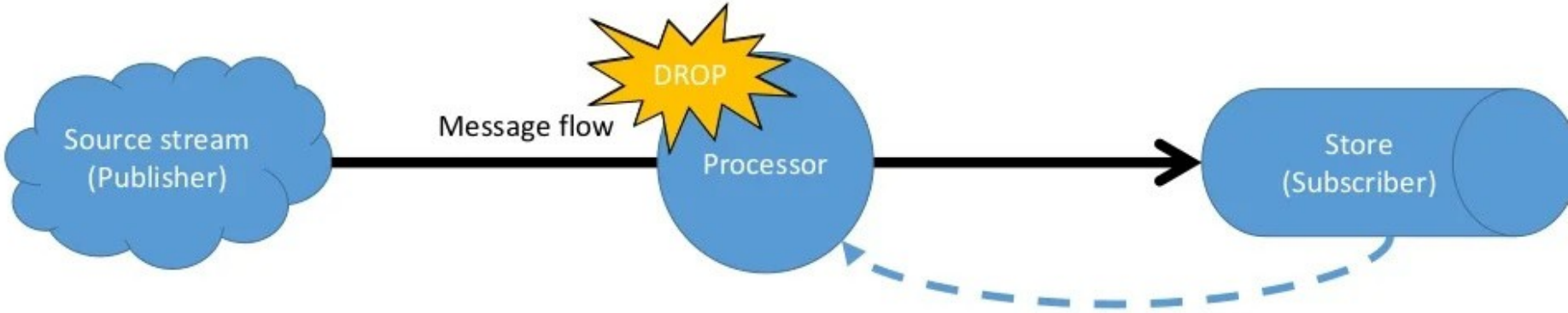
Reactive Streams

- Available since 2013; latest version: 1.0.2 (19-12-2017)
- "asynchronous streams of data with non-blocking **back pressure**"
  - a.k.a. async pull-push flow control
- 4 interfaces, 7 methods, 43 rules for implementors:
  - `Publisher<T>`
    - *`subscribe`*`(Subscriber<? super T> s)`
  - `Subscriber<T>`
    - *`onSubscribe`*`(Subscription s)`, *`onNext`*`(T t)`, *`onError`*`(Throwable t)`, *`onComplete`*`()`
  - `Subscription`
    - *`request`*`(long n)`, *`cancel`*`()`
  - `Processor<T,R> extends Subscriber<T>, Publisher<R>`
- Included in Java SE since JDK9 through JEP-266 as `java.util.concurrent.Flow.*`
- Different implementations can be connected and converted

Explicit demand model propagates backpressure



Dealing with unbackpressured upstream



Akka – a JVM implementation of the actor model

- Actors can:
  - receive messages
  - asynchronously send messages to other actors
  - create and supervise child actors
  - change their own internal state while processing a message

- When an actor encounters a problem, it will notify its parent, who can either:
  - Resume
  - Restart
  - Stop
  - Escalate

History of the Actor model

- Proposed in 1973 by Carl Hewitt
- Implemented in Erlang in 1980's
  - 99.9999999% (!) uptime in telco setting
- Implemented in Scala in 2006
  - Had problems
- Akka: separate project started in 2009
  - Scala and Java API's
  - Replaced Scala implementation in 2012
  - Java API is fully supported and up to date

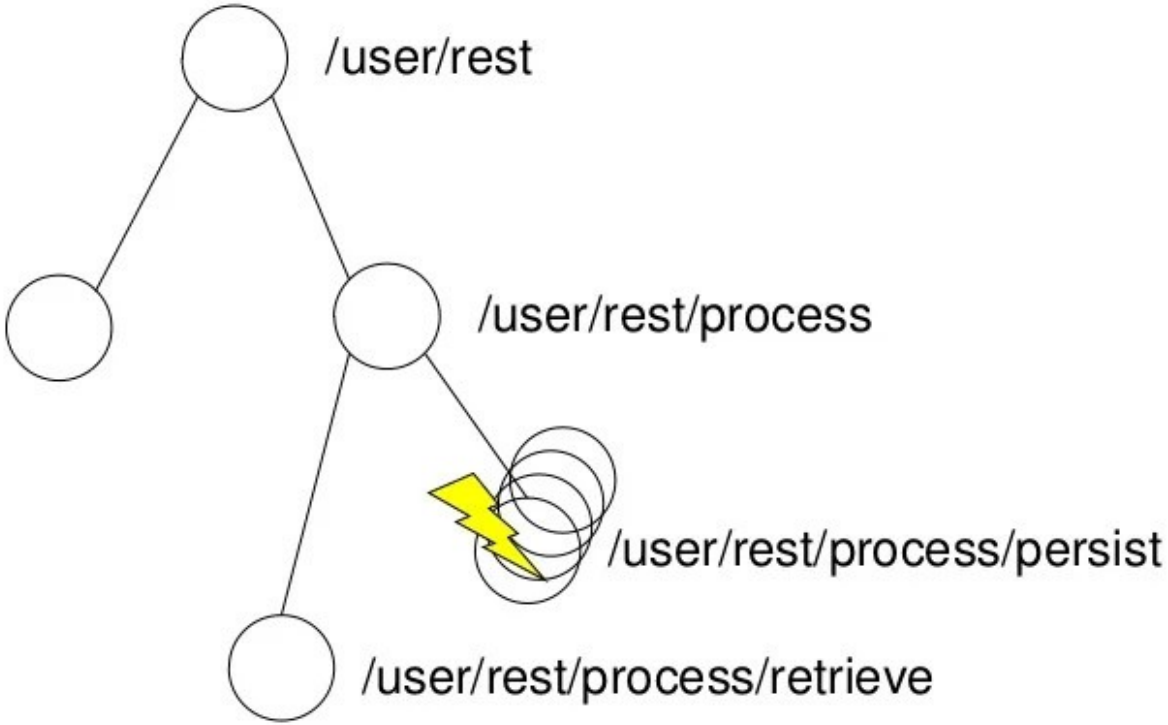
So what does an actor do?

- Receive messages
- Send messages to any actor address
- Create and supervise child actors
- Change the way it will process the next message

Why is this a good abstraction?

- Explicit about unreliability and handling it
  - A slow actor is a dead actor
- Communicate solely by message passing
  - eliminates shared state bottleneck
- Scale up for “free*”
  - Immutability means parallellizability

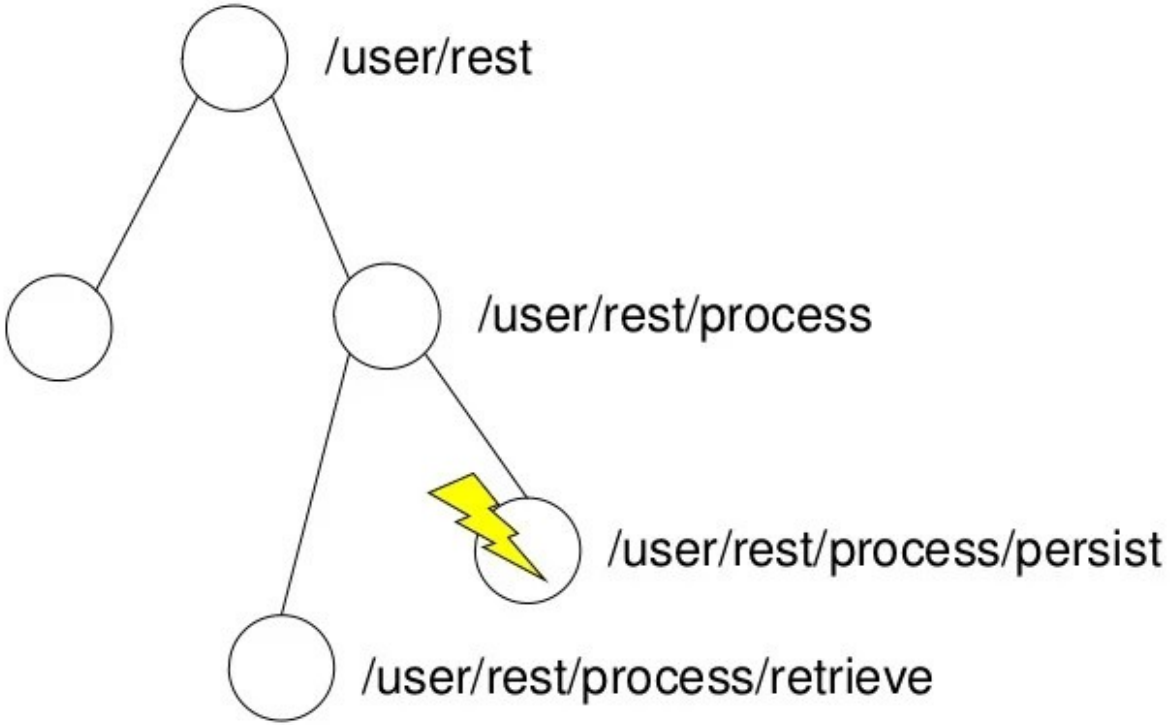
Example Actor System



Supervision (a.k.a. Let It Crash)

- Resume
  - `catch(Exception e) {}`

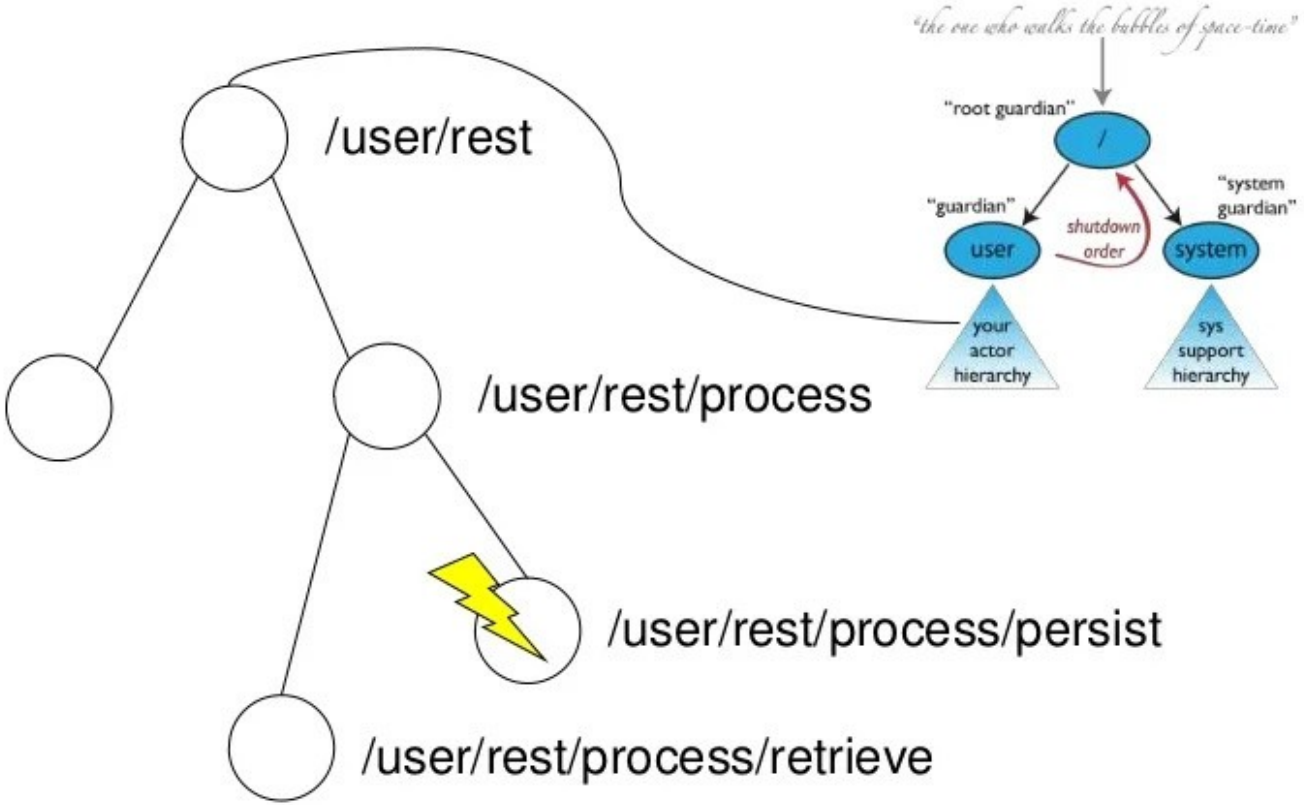
Resume


/user/rest
/user/rest/process
/user/rest/process/persist
/user/rest/process/retrieve

Supervision (a.k.a. Let It Crash)

- Resume
  - `catch(Exception e) {}`
- Restart
  - `this = new Child()`

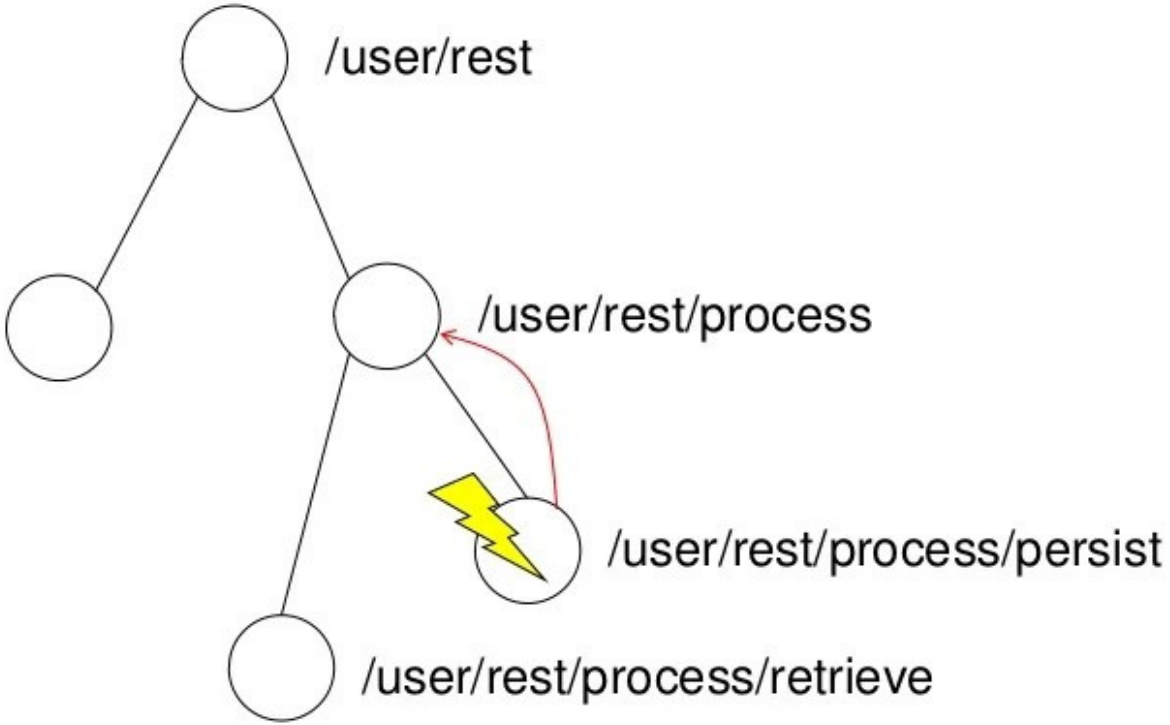
Example Actor System



Supervision (a.k.a. Let It Crash)

- Resume
  - `catch(Exception e) {}`
- Restart
  - `this = new Child()`
- Stop
  - `Parent ! Terminated`

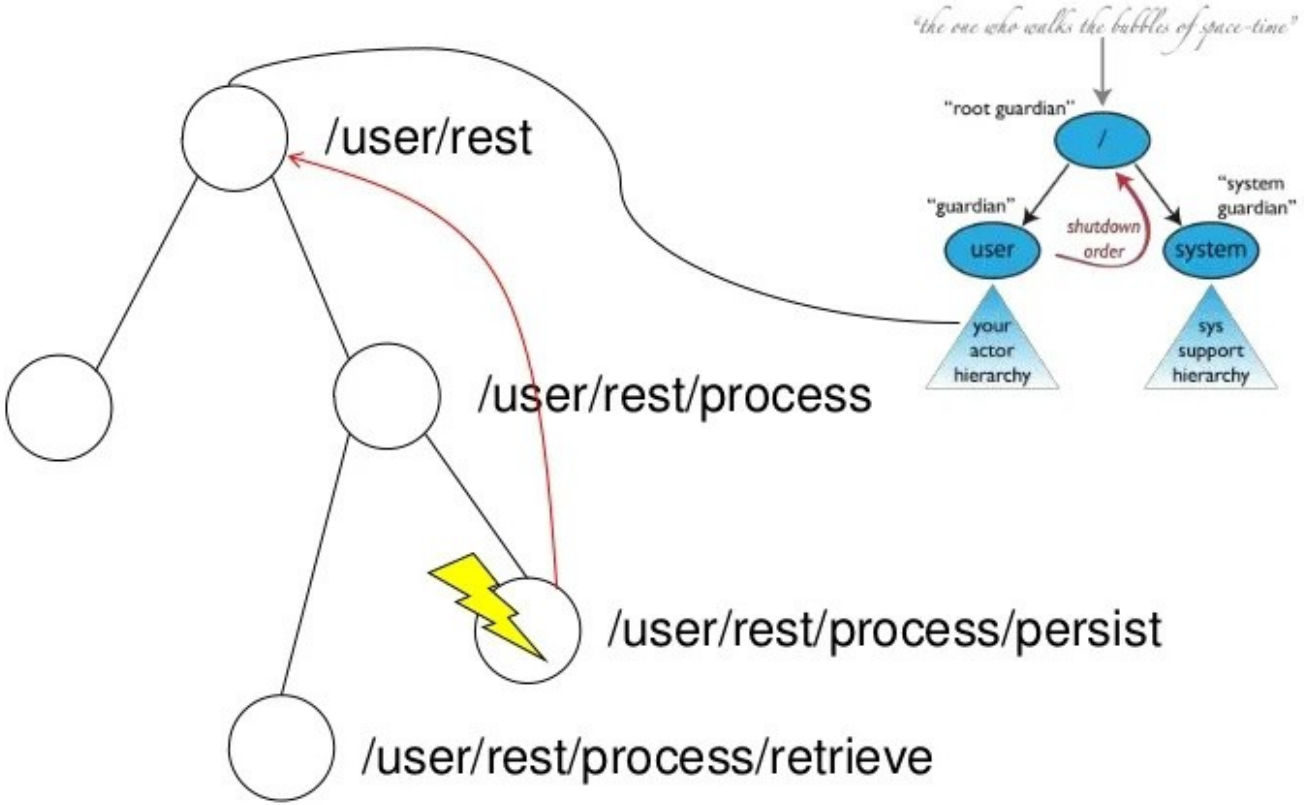
Stop


/user/rest
/user/rest/process
/user/rest/process/persist
/user/rest/process/retrieve

Supervision (a.k.a. Let It Crash)

- Resume
  - `catch(Exception e) {}`
- Restart
  - `this = new Child()`
- Stop
  - `Parent ! Terminated`
- Escalate
  - `throw(e)`

Example Actor System



Akka-specific features

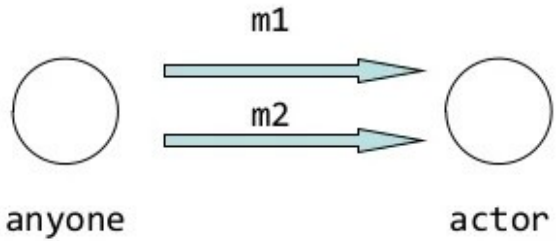
- On top of the actor model, Akka provides:
  - Ordered message arrival
  - Scaling out with location transparency
  - Tool box with high-level components
    - Circuit breaker

Akka-specific features

- On top of the actor model, Akka provides:
  - Ordered message arrival
  - Scaling out with location transparency
  - Tool box with high-level components
    - Circuit breaker

Ordered message arrival

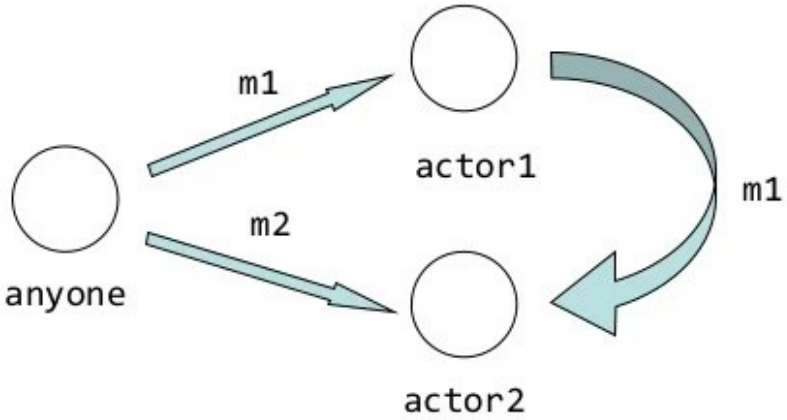
- `actor.send(m1)`
- `actor.send(m2)`



- In `receive()` method of `actor`, you will never get `m2` before `m1`

Doesn't hold transitively

- `actor1.send(m1)`
- `actor2.send(m2)`
- In `receive()` method of `actor1`:
  - `actor2.forward(m1)`

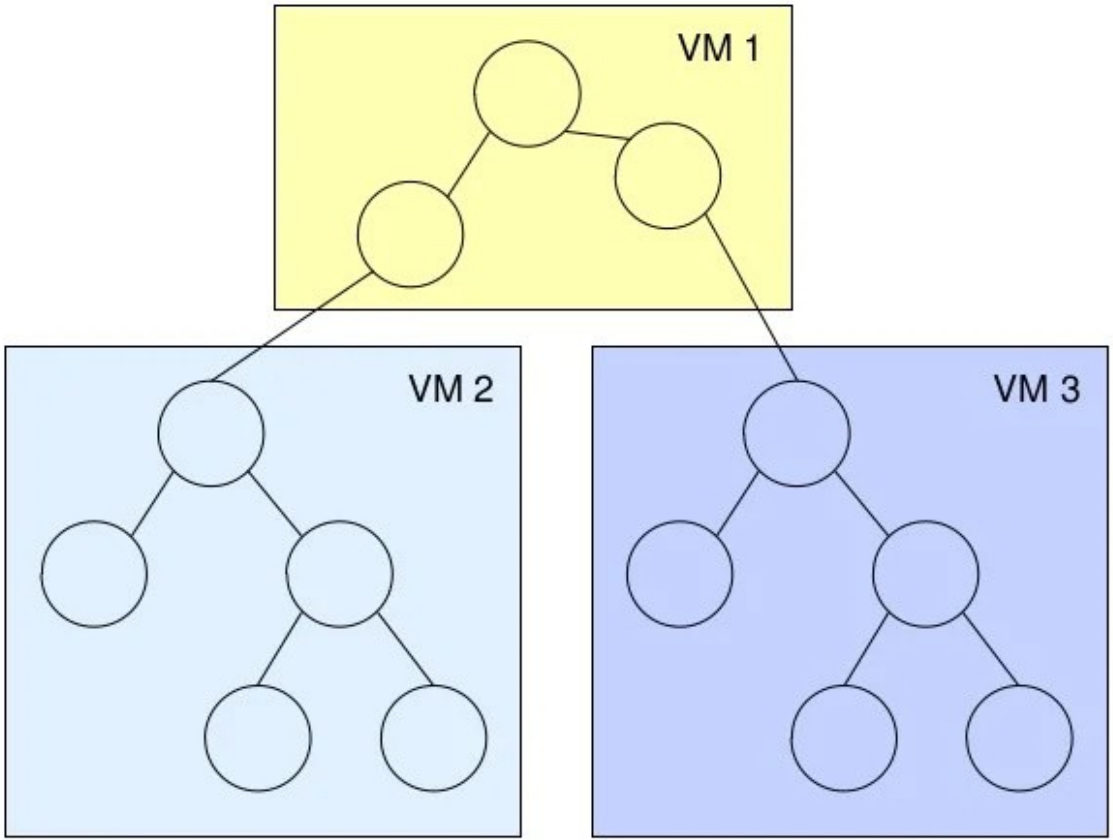


No way to tell
what `actor2` will see

Akka-specific features

- On top of the actor model, Akka provides:
  - Ordered message arrival
  - Scaling out with location transparency
  - Tool box with high-level components
    - Circuit breaker

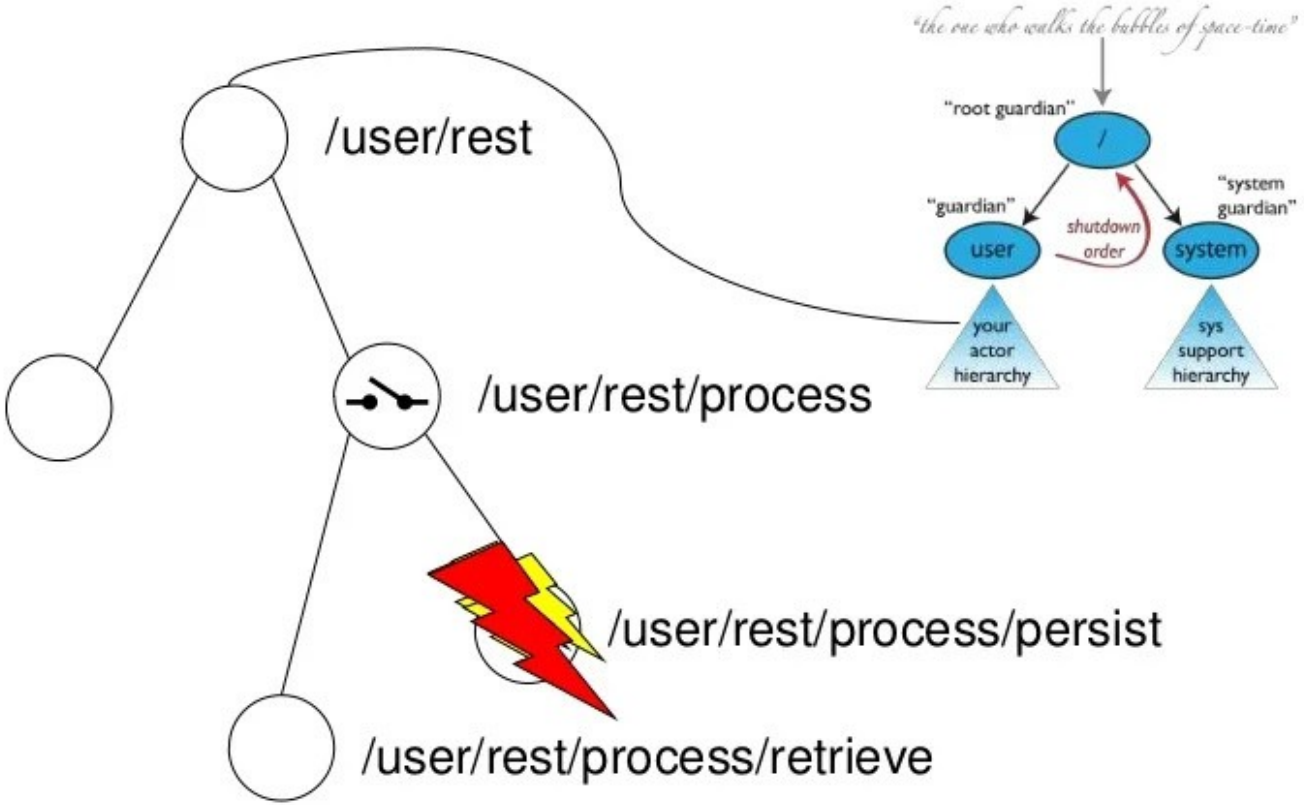
Location transparency



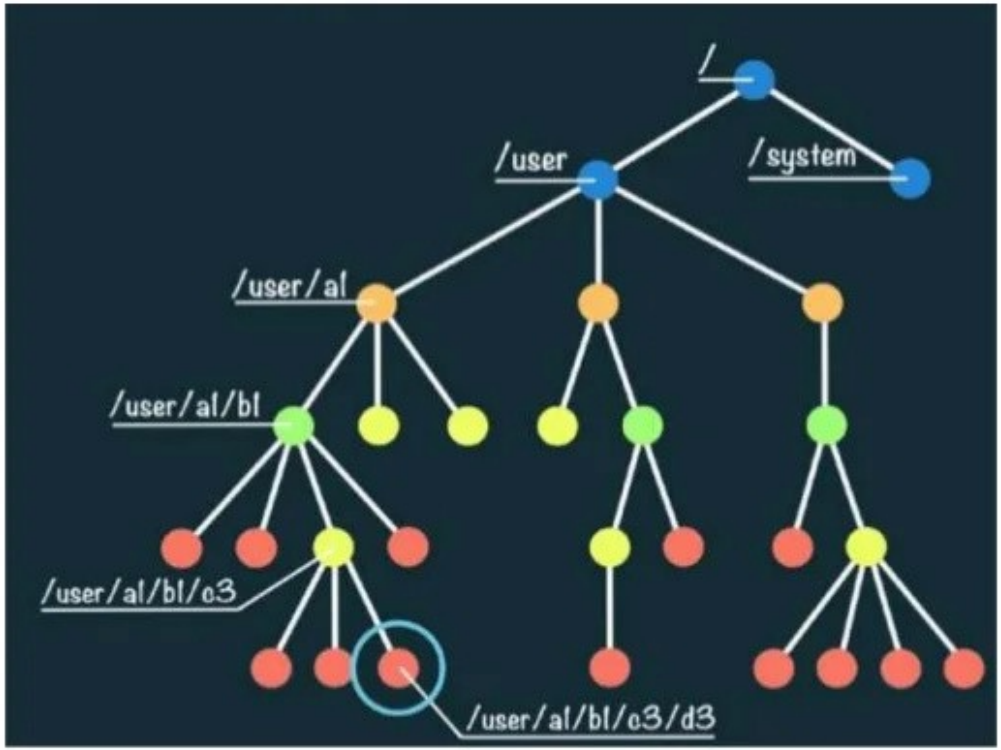
Akka-specific features

- On top of the actor model, Akka provides:
  - Ordered message arrival
  - Scaling out with location transparency
  - Tool box with high-level components
    - Circuit breaker

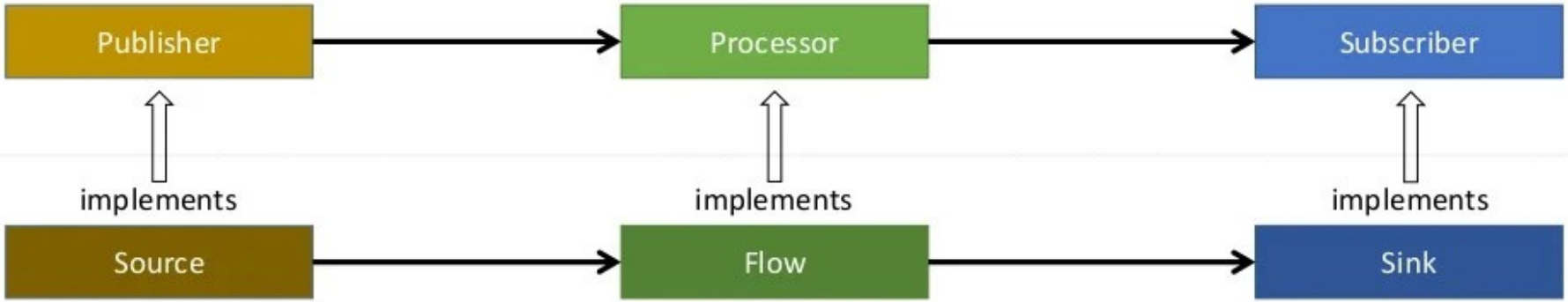
Circuit breaker



Delegate “risky” tasks (e.g. calls over network)



Akka Streams implements Reactive Streams



Akka Streams code example: split/aggregate

```
CompletionStage<List<Integer>> ret =
        Source.from(Arrays.asList("1-2-3", "2-3", "3-4"))
                .map(s -> Arrays.asList(s.split("-")))
                //split all messages into sub-streams
                .splitWhen(a -> true)
                //now split each collection
                .mapConcat(f -> f)
                //Sub-streams logic
                .map(s -> Integer.valueOf(s))
                //aggregate each sub-stream
                .reduce((a, b) -> a + b)
                //and merge back the result into the original stream
                .mergeSubstreams()
                .runWith(Sink.seq(), materializer);
                //result: List(6, 5, 7)
```

Akka HTTP

- Built on top of Akka Streams
- HTTP requests and responses are represented with Akka Streams types:
  - Requests as `Source<HttpRequest, ?>`
  - Responses as `Sink<HttpResponse, ?>`
  - Message bodies as `Source<ByteString, ?>`

Reactive Enterprise Integration using Akka

- System integration in a nutshell:
  - Input
  - Transform
  - Output
- Camel lets you define these separately, but no native streaming support
- RxJava lets you define streams, but not in a reusable way
- Akka Streams lets you define reusable, reactive components
  - Akka HTTP is an async HTTP client built on top of Akka Streams
  - Alpakka provides a set of backpressure-aware components…


Alpaca?

```scala
val jmsSource: Source[String, KillSwitch] =
 JmsConsumer.textSource(                                                    // (1)
  JmsConsumerSettings(connectionFactory).withBufferSize(10).withQueue("test")
 )

val webSocketFlow: Flow[ws.Message, ws.Message, Future[WebSocketUpgradeResponse]] = //
 Http().webSocketClientFlow(WebSocketRequest("ws://localhost:8080/webSocket/ping"))

val (runningSource, wsUpgradeResponse): (KillSwitch, Future[WebSocketUpgradeResponse]) =
// stream element type
 jmsSource                                  //: String
  .map(ws.TextMessage(_))                   //: ws.TextMessage              (3)
  .viaMat(webSocketFlow)(Keep.both)         //: ws.TextMessage                  (4)
  .mapAsync(1)(wsMessageToString)           //: String                      (5)
  .map("client received: " + _)             //: String                  (6)
  .toMat(Sink.foreach(println))(Keep.left)  //                            (7)
  .run()
```

Alpakka connectors

- AMQP Connector
- Apache Geode connector
- Apache Solr Connector
- AWS DynamoDB Connector
- AWS Kinesis Connector
- AWS Lambda Connector
- AWS S3 Connector
- AWS SNS Connector
- AWS SQS Connector
- Azure Storage Queue Connector
- Cassandra Connector
- Elasticsearch Connector
- File Connectors
- FTP Connector
- Google Cloud Pub/Sub
- Google Firebase Cloud Messaging
- HBase connector
- IronMq Connector
- JMS Connector
- MongoDB Connector
- MQTT Connector
- OrientDB Connector
- Server-sent Events (SSE) Connector
- Slick (JDBC) Connector
- Spring Web
- Unix Domain Socket Connector
- File IO
- Azure
- AWS Kinesis
- Camel
- Eventuate
- FS2
- HTTP Client
- MongoDB
- Kafka
- Pulsar
- TCP


Alpakka
Akka HTTP
Akka Streams
Reactive Streams
Akka Actors

Vert.X : Akka

≡

Containers : λ

RxJava is backwards compatible

- Android does not fully support Java 8; support for `java.util.function.*` only since Android 7.0
  - RxJava (including 2.x) targets Android 2.3+ → JDK 6
  - Own set of interfaces to support FP
  - Extra adapter classes needed for Reactive Streams

Links

- Konrad Malawski - Why Reactive? http://www.oreilly.com/programming/free/files/why-reactive.pdf