ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Failure handling and processing rate

Programming Reactive Systems

Konrad Malawski, Julien Richard-Foy

“Failure” and “rate”

This section has two main sub-topics:

- handling *failure*
- handling differing *processing rates* in a stream

They are slightly related, as they are what differentiates streams from “just” collections.

Failure (in contrast to Error)

The Reactive Manifesto defines *failures* as:

> *A failure is an unexpected event within a service that prevents it from continuing to function normally.*

And contrasts them with *errors*:

> *This is in contrast with an error, which is an expected and coded-for condition—for example an error discovered during input validation, that will be communicated to the client as part of the normal processing of the message.*

Failure (in contrast to Error)

In Reactive Streams, mostly due to historical naming of such methods in Reactive Extensions, the name `onError` is used to talk about stream failure.

Among other reasons why this should be seen as a failure signal is:

- `onError` signals can be sent out-of-bounds (no demand, as well as “overtake” onNext signals),
- `onError` carried failures are not part of the streams’ data model, any exception could be thrown at any point in time,
- an `onError` signal is *terminal*, if a failure happens and `onError` is sent, no further signals may be sent by that upstream.

Carrying errors as values

While we will indeed use the failure handling mechanisms built-into Akka Streams, another important technique to be aware of is carrying errors as values, e.g.:

- `Flow[Validatable, ValidationResult, _]` representing a stream of values where each has to be validated.
- `Flow[Try[T], T, _]`, a “filter Successful” flow.

Failure logging and propagation

Failures flow only *downstream*; if a stage fails this signal is sent down, and a cancellation signal is sent upstream (if any).

Failures usually remain *within* the stream, unless exposed via materialized value for example, by a Sink such as `Sink.seq: Sink[T, Future[Seq[T]]]` which would fail the materialized *failed Future* if a failure is received.

One can also use the `.log().withAttributes(ActorAttributes.logLevels(...))` operator to log all signals at given point.

Recovering from failure

Akka Streams provides a number of ways to recover from failure signals of an upstream:

- `recover[T](pf: PartialFunction[Throwable, T])` operator
- `recoverWith[T](pf: PartialFunction[Throwable, Source[T, _]])` operator
- restart stages with backoff support, including `RestartFlow.withBackoff`

Processing rate

Processing rate is the throughput at which a stage is processing elements.

We already talked about this in the first videos of this week, after all – the need of back-pressure arises only if the processing rates of the streaming stages differ.

In streaming systems it is often refered to as a processing stages *throughput*, which we'll talk about in *elements per second* (or other time unit).

“Global” vs. local rate measurements

It is important to realise, that the processing rate, may be different at various points in the stream.

For example, imagine stream composed of 3 stages:

- an infinite source of numbers
- a flow stage, that only emits an element downstream if and only if it is divisible by 2
- a sink, that accepts such numbers

“Global” vs. local rate measurements

We can observe two kinds distinct throughput values in this system:

- at the source / before the flow
- after the flow / before the sink

“Rate aware” operators

Some operators may take advantage of being aware of the Reactive Streams back-pressure:

- `conflate` - combines elements from upstream *while downstream back-pressures*
- `extrapolate` - in face of faster downstream, allows *extrapolating elements from last seen upstream element*

Summary

- *Failures* of streams are propagated *downstream* through a stream (from `Sources` or `Flows` towards `Sinks`)
- *Recovering from failure* can be done using simple operators or advanced patterns like `Restart*` stages
- *Processing rate* is the throughput at which a stage is processing elements.
- *Rate* must be measured at given points of a stream, since it may differ in various parts of the pipeline