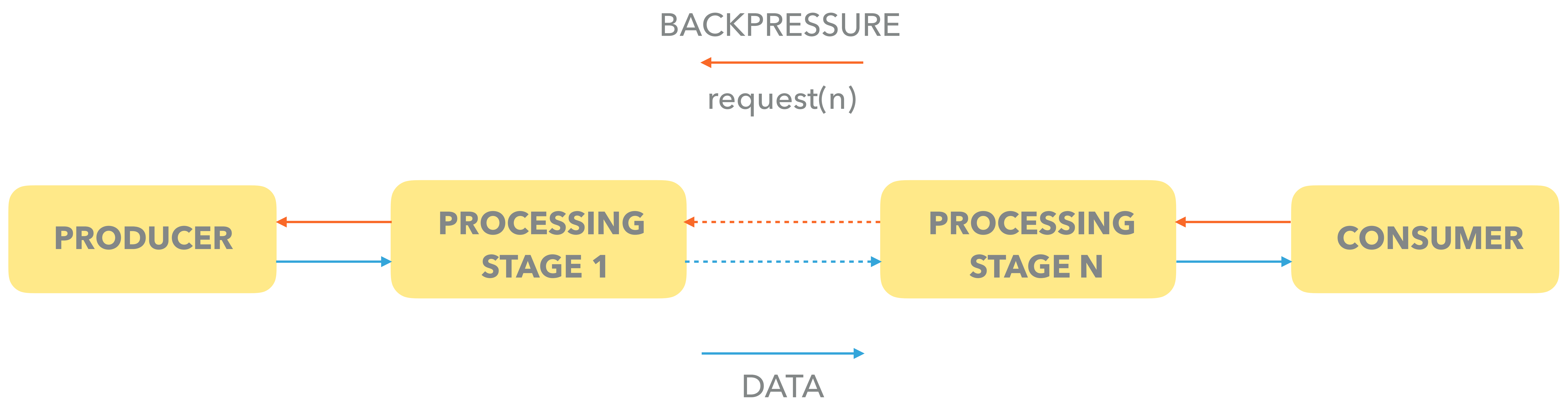
JACEK KUNICKI
@rucek

SOFTWAREMILL

HOW (NOT) TO USE REACTIVE STREAMS IN JAVA 9+

STREAM PROCESSING



Publisher Processor Processor Subscriber

REACTIVE STREAMS

- asynchronous
- non-blocking backpressure
  - reuse threads whenever possible
- slow consumers are represented in the domain model, e.g.
  - Twitter API can tell you that you're consuming too slow
  - `conflate()` in Akka Streams - aggregates when the downstream is slow

`java.util.concurrent.Flow`

`j.u.c.Flow.Publisher<T>`

- produces items of type T that subscribers are going to consume
- subscribers are registered via `subscribe(Subscriber<? super T>)`

`j.u.c.Flow.Subscriber<T>`

- subscribes to a producer in order to receive:
  - subscription confirmation via `onSubscribe(Subscription)`
  - items via `onNext(T)`
  - errors via `onError(Throwable)`
  - completion signal via `onComplete()`

`j.u.c.Flow.Subscription`

- connects a single producer to a single subscriber, allows to:
  - backpressure with `request(long)`
  - signal termination with `cancel()`

`j.u.c.Flow.Processor<T, R>`

- a combination of a `Subscriber<T>` and a `Publisher<R>`

```
publisher.subscribe(subscriber)
```

```
onSubscribe
onNext*
(onComplete | onError)?
```

Talk is cheap
show me the
CODE
SOFTWAREMILL

FURTHER CHALLENGES

- unbounded recursion through `request() -> onNext() -> request() -> …`
- handling infinite demand
  - just calling `onNext()` for each of `MAX_VALUE` elements will exhaust the threads
  - `long demand` + incrementing is not enough - overflow

http://www.rhfleet.org/sites/default/files/images/RHF_DTTAH_Clogo_DEC14.jpg

SPI

SERVICE PROVIDER INTERFACE

EXISTING STREAMING ABSTRACTIONS

- `java.io.InputStream / OutputStream`
- `java.util.Iterator`
- `java.nio.channels.*`
- `javax.servlet.ReadListener/WriteListener`
- `java.sql.ResultSet`
- `java.util.Stream`
- `java.util.concurrent.Flow.*`

```
publisher.subscribe(subscriber)
```

MINIMUM OPERATION SET

- only interfaces at the moment
- no basic operations like `filter`, `map` etc.
- https://github.com/lightbend/reactive-streams-utils
- basic operations built-in, others pluggable like `-Djava.flow.provider=akka`

but politics: vs

HTTP

- async Servlet IO (since 3.1)
- JDK 9+ HTTP client provides a `POST(Publisher<ByteBuffer>)`
- if the `HttpServletRequest` provided a body publisher, file upload would become:

```
POST(BodyPublisher.fromPublisher(req.getPublisher())
```

DATABASE ACCESS

- ADBA (Asynchronous Database Access API)
- existing vendor-specific async drivers
- JPA - what if…

```
Publisher<User> users = entityManager
  .createQuery("select u from users")
  .getResultPublisher()
```

AND MORE

- reactive file IO (like a `Publisher<Byte>`)
- JMS
- websockets
- AWS - on the way
- Alpakka?

DEMO 2 – SIMPLE INTEGRATION

- Project Reactor's `Flux` as a publisher
- Akka Streams `Flow` as a processor
- RxJava as a subscriber

SUMMARY

- not a full Reactive Streams implementation
- an SPI that allows for interoperability between other implementations
- implementing it yourself is at least non-trivial
  - use the TCK

RESOURCES

- pluggable runtime: https://github.com/lightbend/reactive-streams-utils
- TCK: https://github.com/reactive-streams/reactive-streams-jvm#specification
- ADBA: https://blogs.oracle.com/java/jdbc-next:-a-new-asynchronous-api-for-connecting-to-a-database
- Advanced Reactive Java: http://akarnokd.blogspot.com/

THANK YOU!

@rucek

SOFTWAREMILL