Reactive Streams, j.u.concurrent, & Beyond!

Konrad `ktoso` Malawski @ QCon San Francisco, 2016

pic: 1:1 scale Gundam model @ Odaiba, Tokyo

Agenda:

Past => Present => Future

“We can do better than that.”

Underlying motto.

For this talk,
and our continued research.

Lightbend

Konrad `ktoso` Malawski

Akka Team,
Reactive Streams TCK,
Persistence, Streams & HTTP, Core, Remoting

Lightbend

polish java user group

geecon

[sckrk]

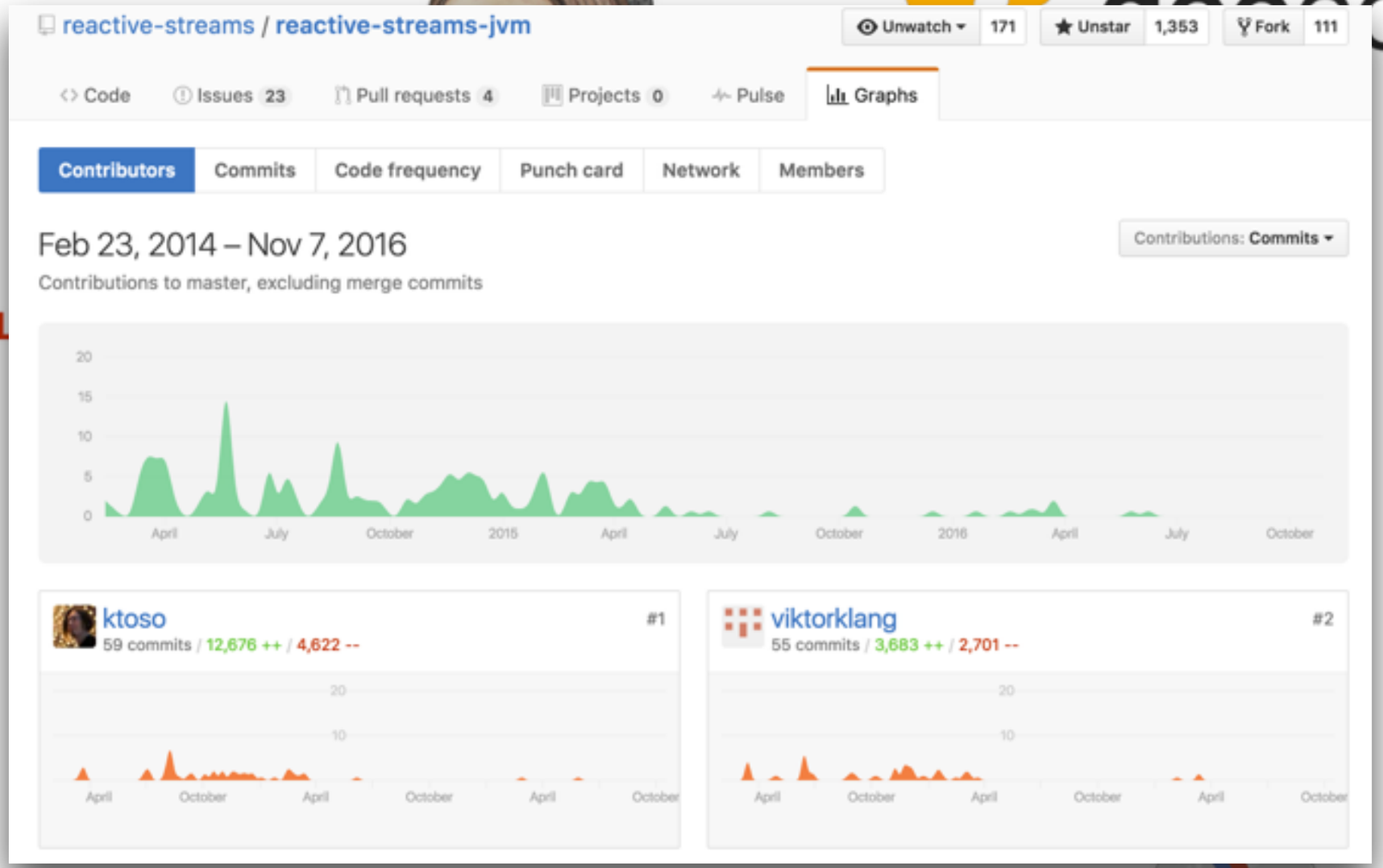
Konrad `@ktosopl` Malawski

KRAKOW SCALA

work: akka.io lightbend.com
personal blog: http://kto.so
communities: geecon.org Java.pl / KrakowScala.pl sckrk.com GDGKrakow.pl lambdakrk.pl

Lightbend

geecon



KRAKOW SCALA

[sckrk]

work: akka.io lightbend.com
personal blog: http://kto.so
communities: geecon.org Java.pl / KrakowScala.pl sckrk.com GDGKrakow.pl lambdakrk.pl

akka

SAMSUNG

Fidelity INVESTMENTS

Scotiabank

HSBC

IBM

BBC

intel

verizon

ebay

THE HUFFINGTON POST

NBC

Time Warner Cable

comcast

workday

Walmart

dropcam

CISCO

amazon web services™

hp

Lightbend

Make building powerful concurrent & distributed applications **simple**.

Akka is a toolkit and runtime for building highly concurrent, distributed, and resilient **message-driven** applications on the JVM

Actors – simple & high performance concurrency
Cluster / **Remoting** – location transparency, resilience
Cluster Sharding – and more prepackaged patterns

Streams – back-pressured stream processing

Persistence – Event Sourcing

HTTP – complete, fully async and reactive HTTP Server
Official **Kafka**, **Cassandra**, **DynamoDB integrations**, tons more in the community

Complete **Java** & **Scala APIs** for all features (since day 1)
Typed coming soon…

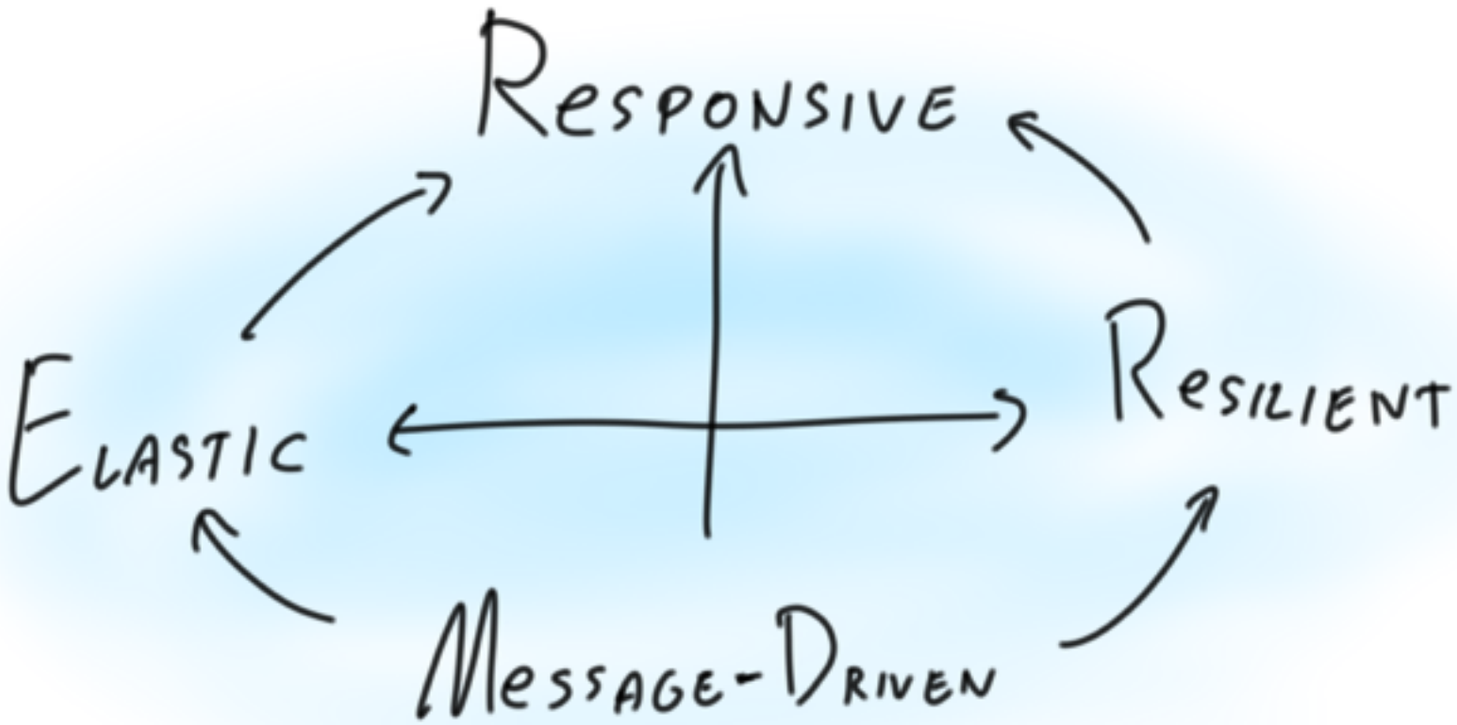
Reactive

And the many meanings it carries.

Reactive

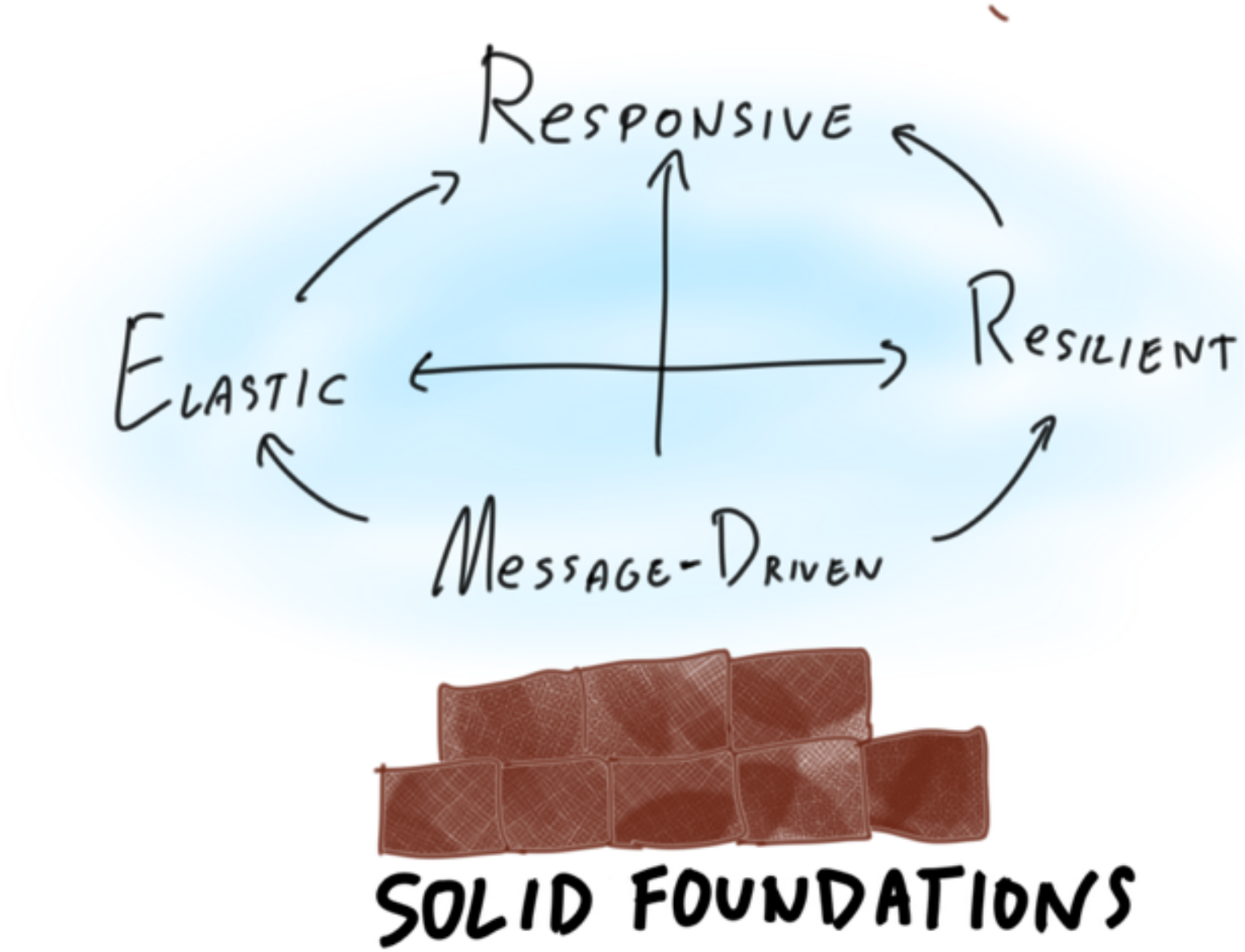
And the many meanings it carries.

The many meanings of **Reactive**



reactivemanifesto.org

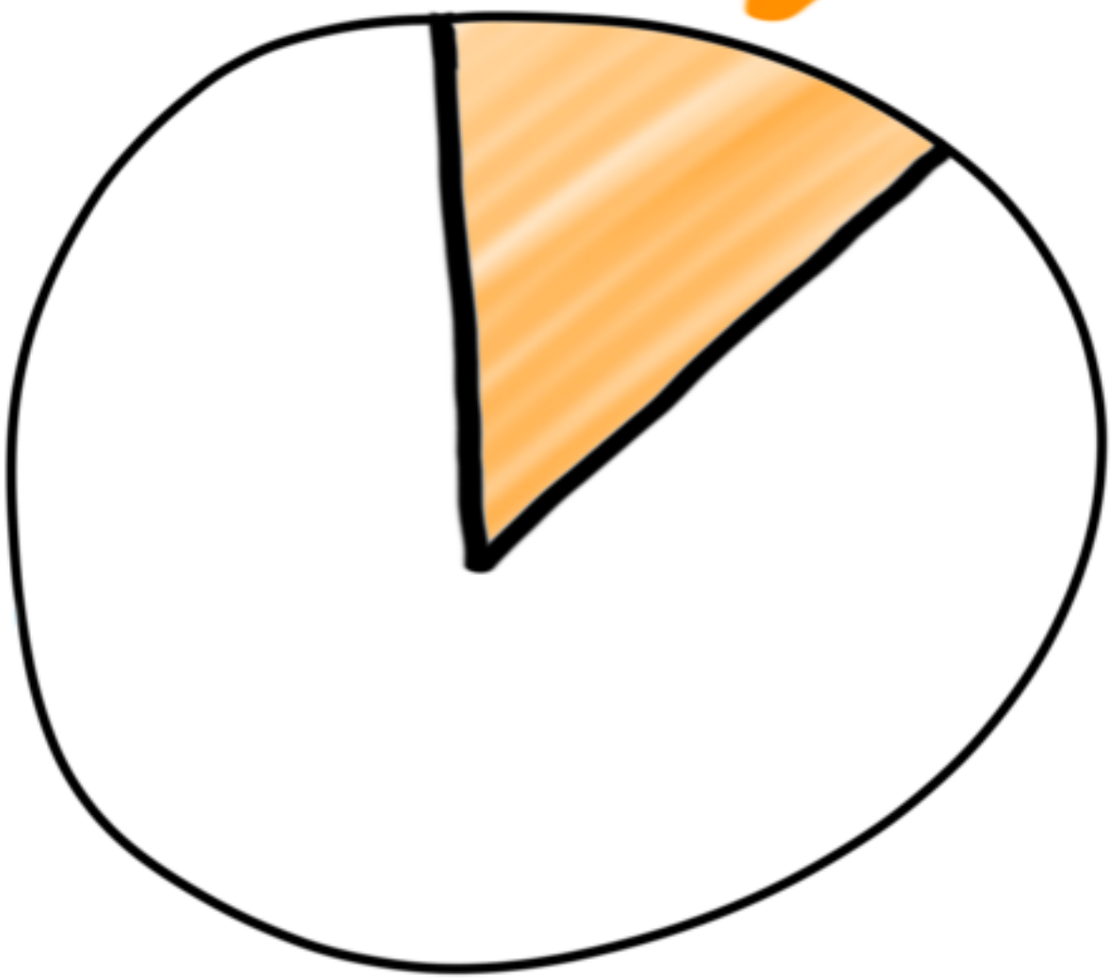
The many meanings of **Reactive**

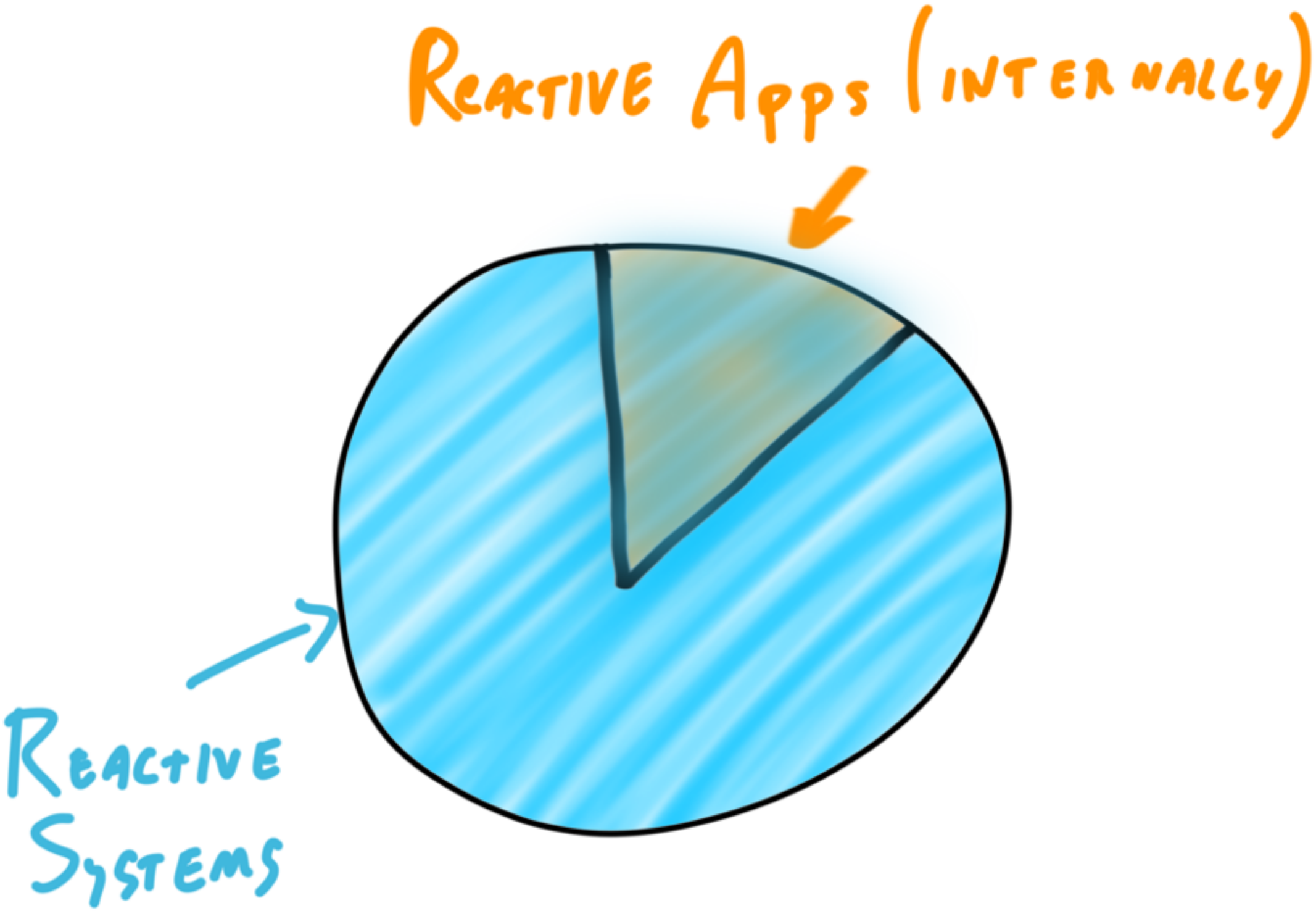


Reactive Apps

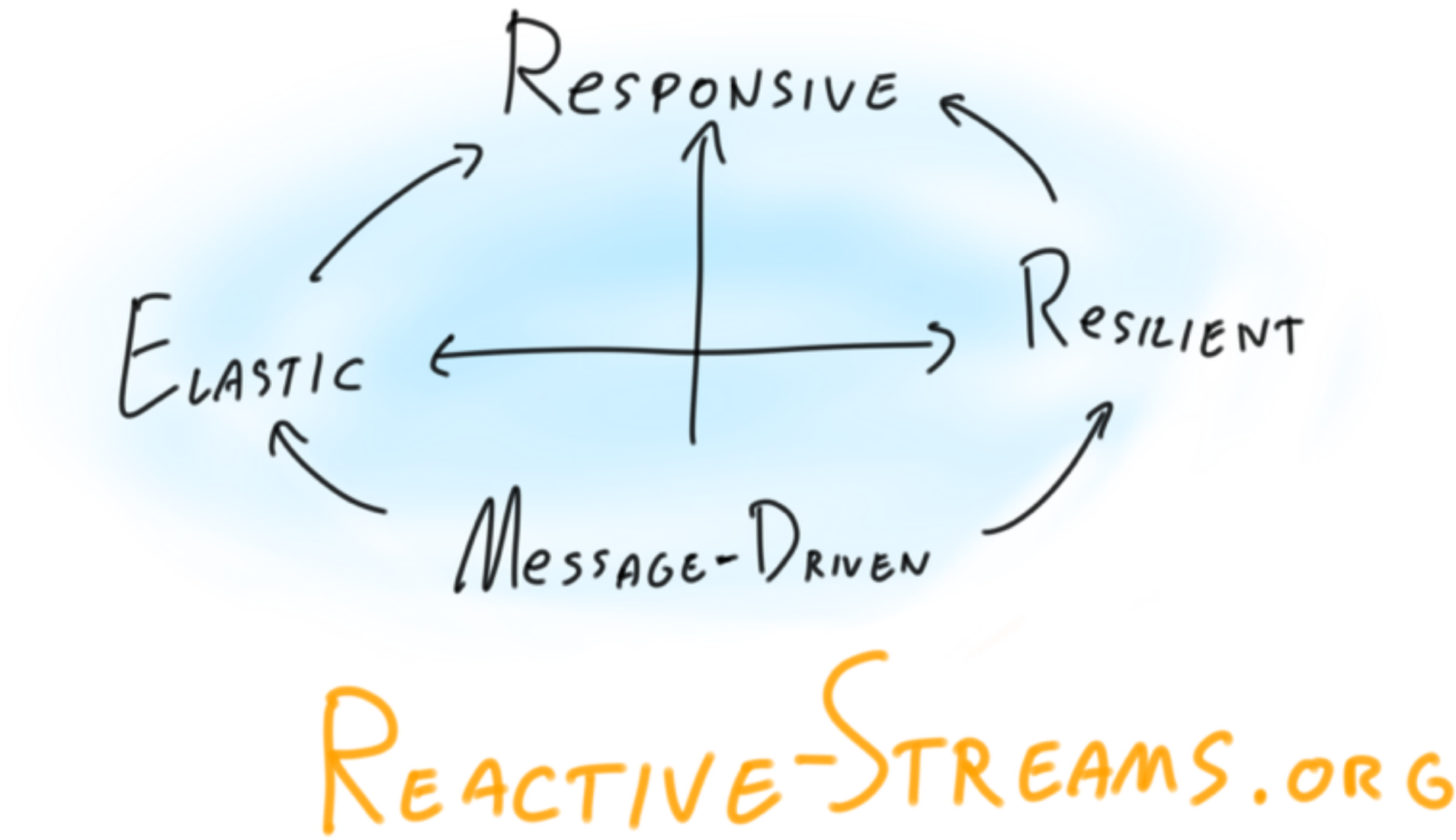
Reactive… on the Application level

Reactive Apps (internally)

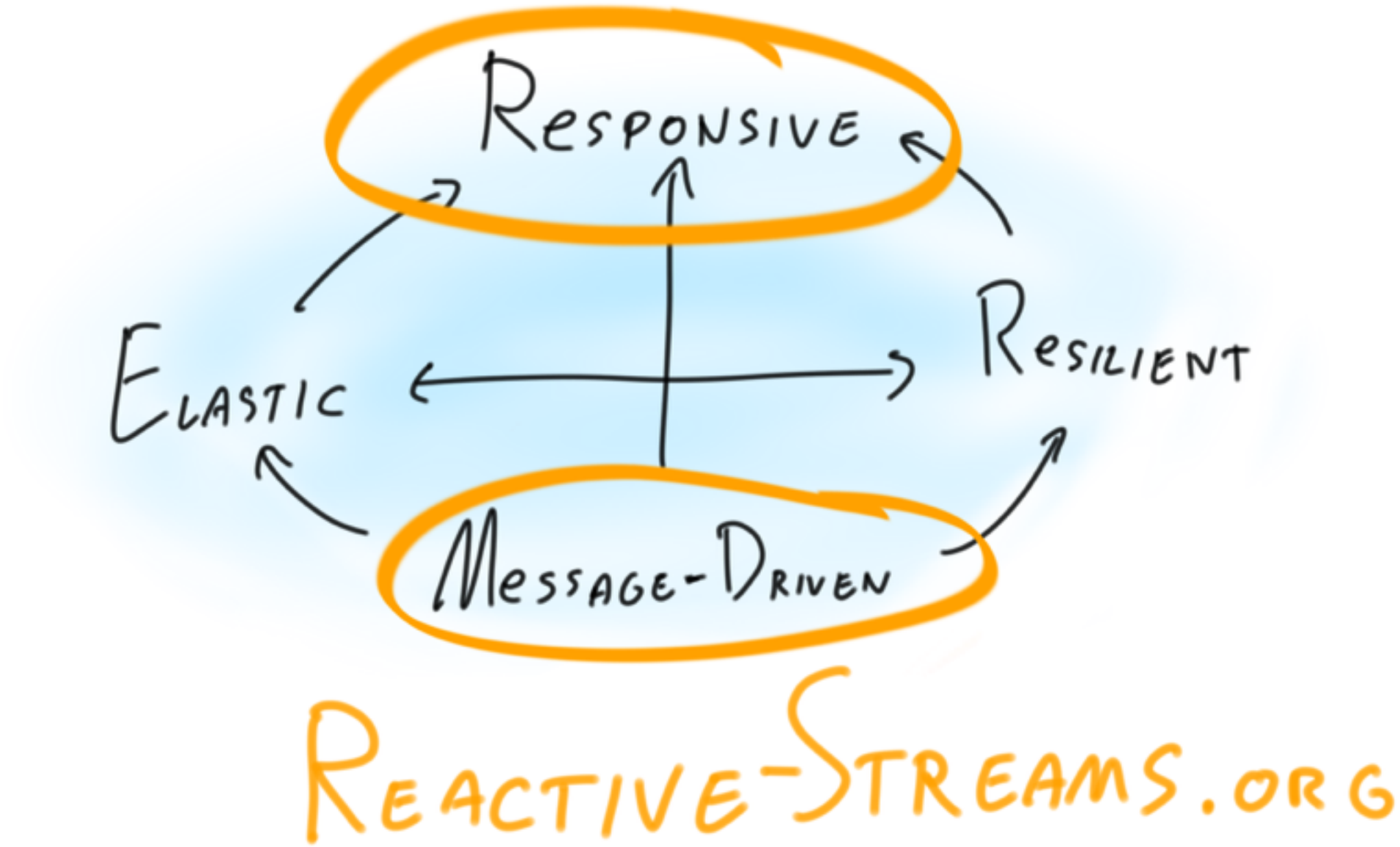



Reactive Apps (internally)
Reactive Systems

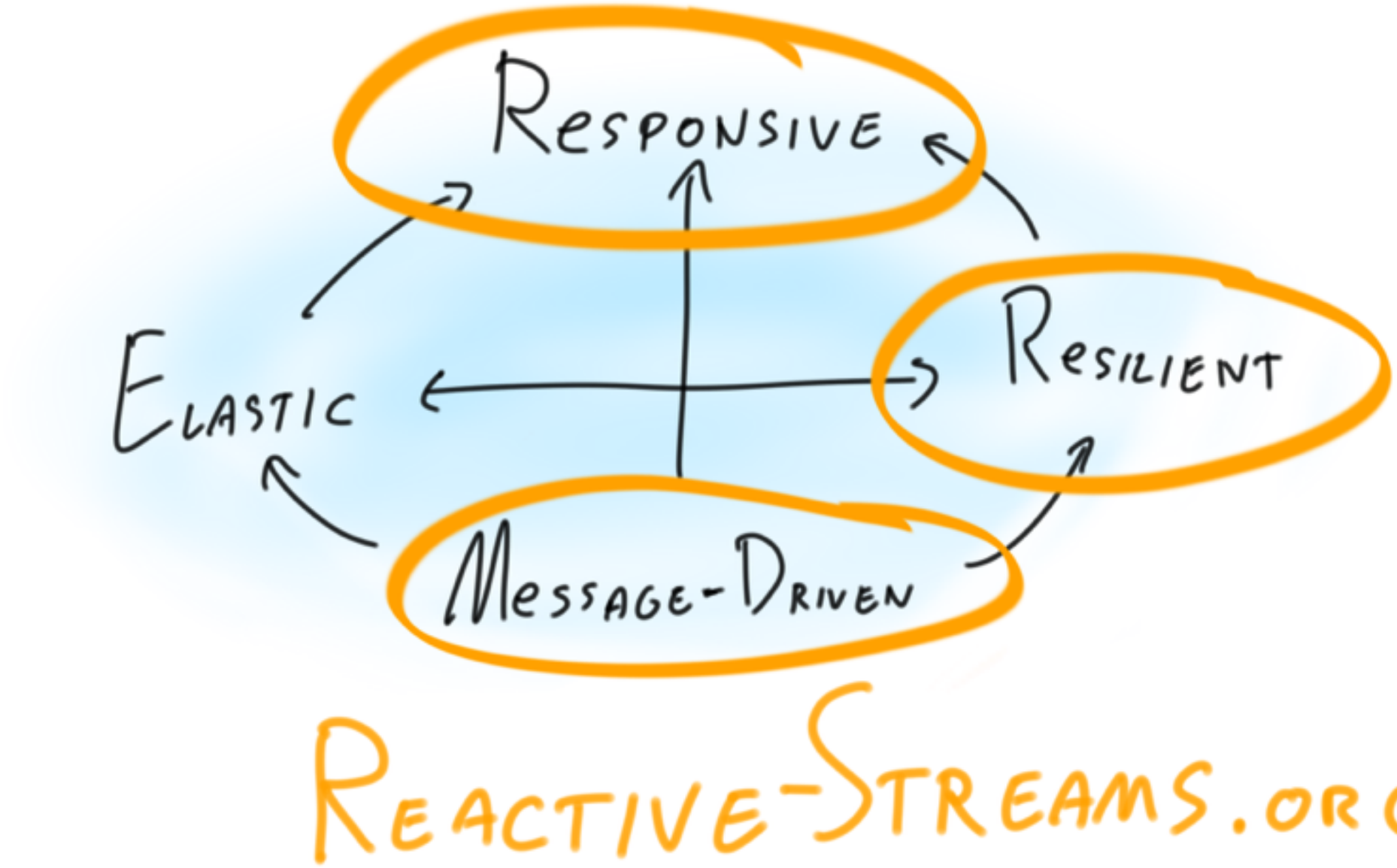
So what are **Reactive Streams** actually?



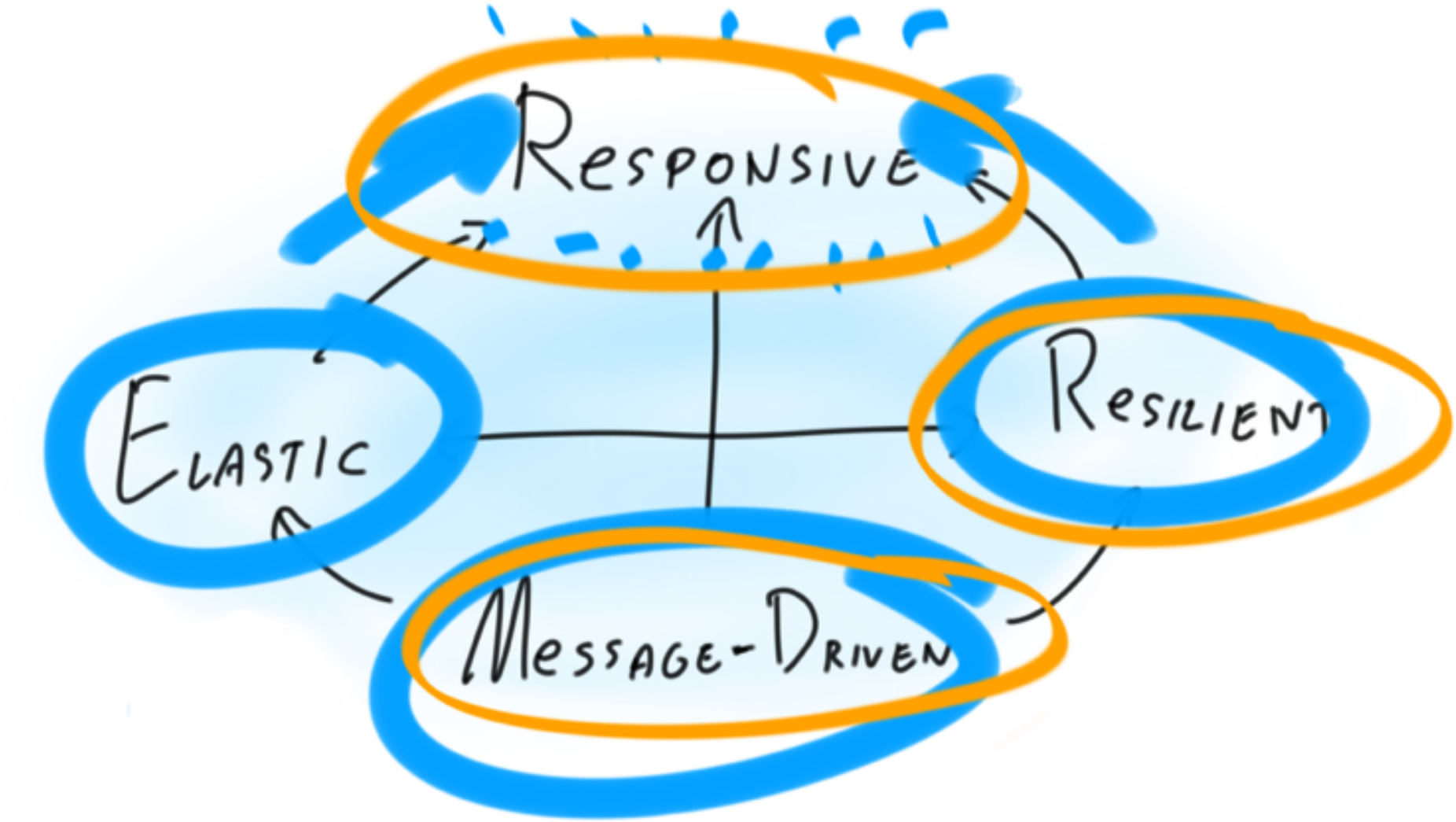
So what are **Reactive Streams** actually?



So what are Reactive Streams actually?

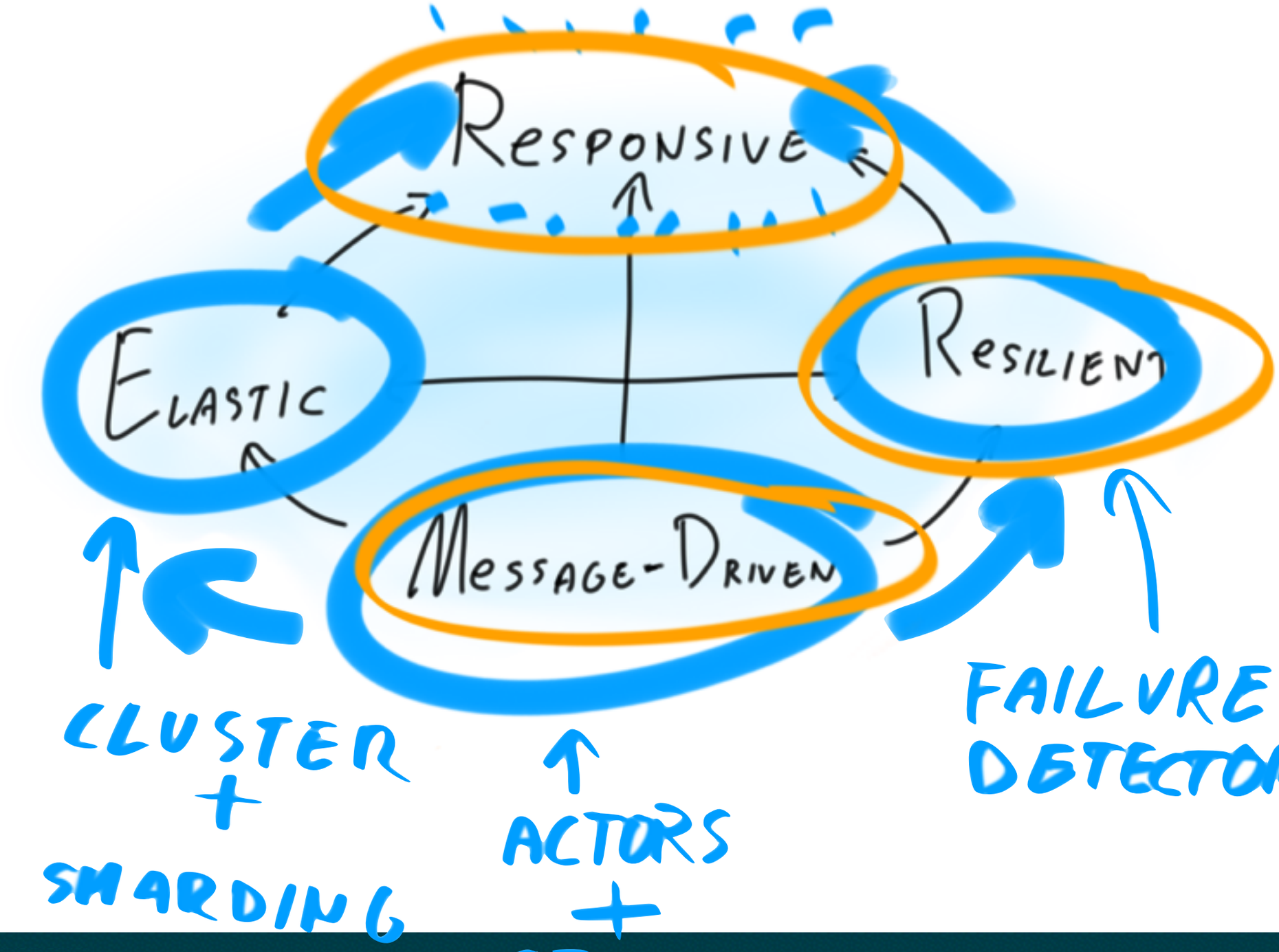


Getting the complete picture



Getting the complete picture

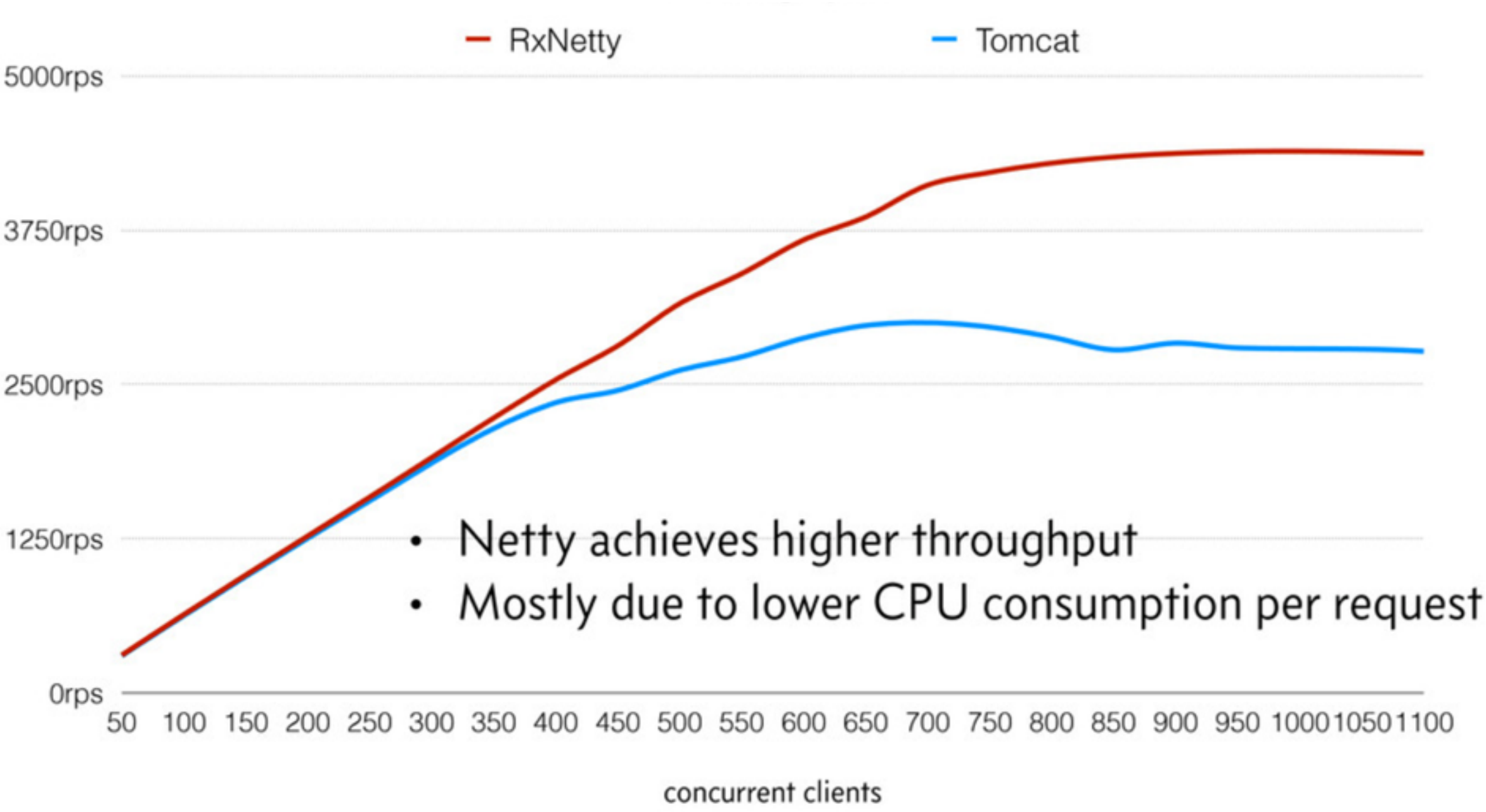
(yet… not the topic of today's talk)



Single Reactive App

Any benefits?

Reactive on the Application level

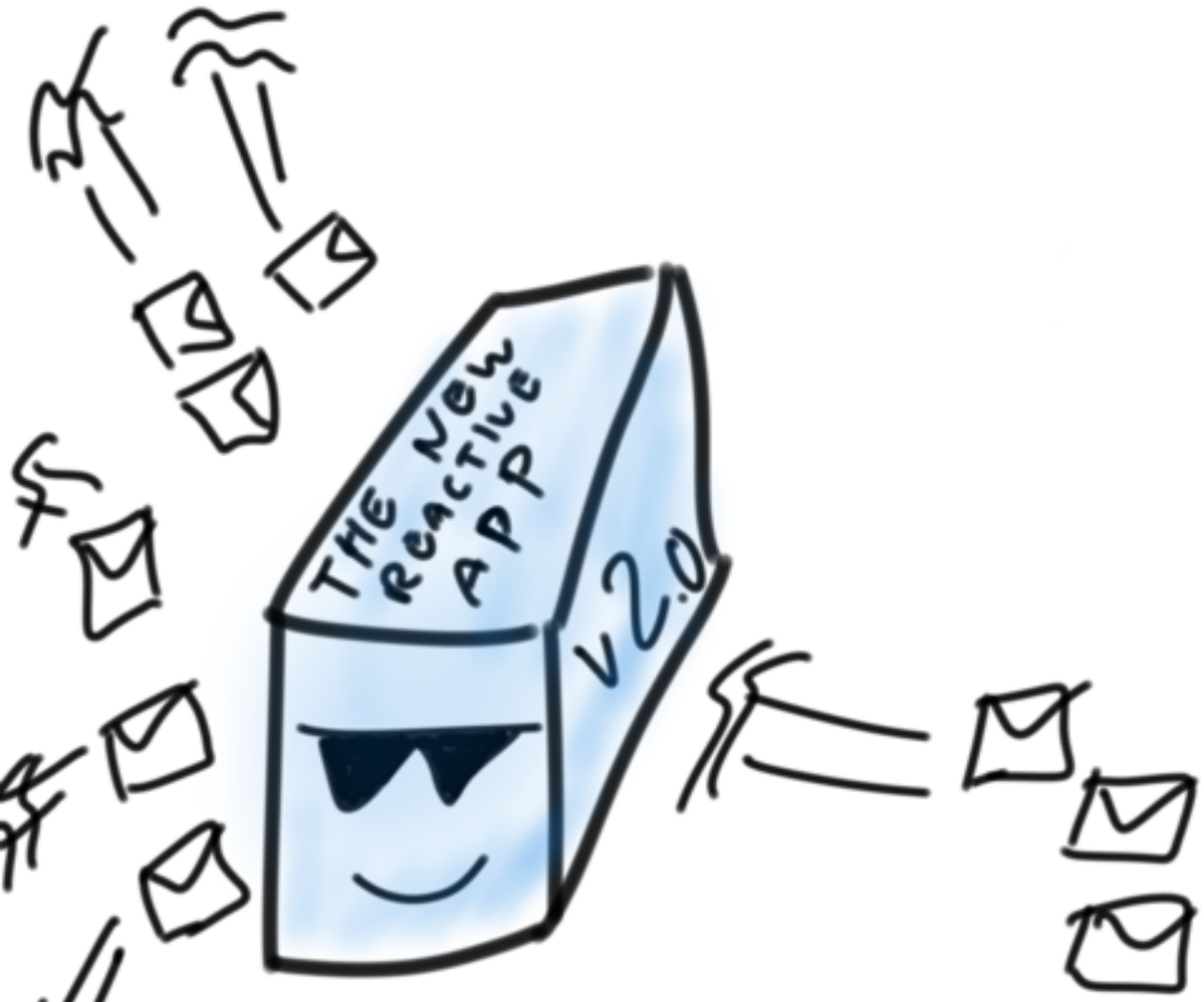


https://speakerdeck.com/benjchristensen/applying-rxjava-to-existing-applications-at-philly-ete-2015

“Not-quite-Reactive-System”

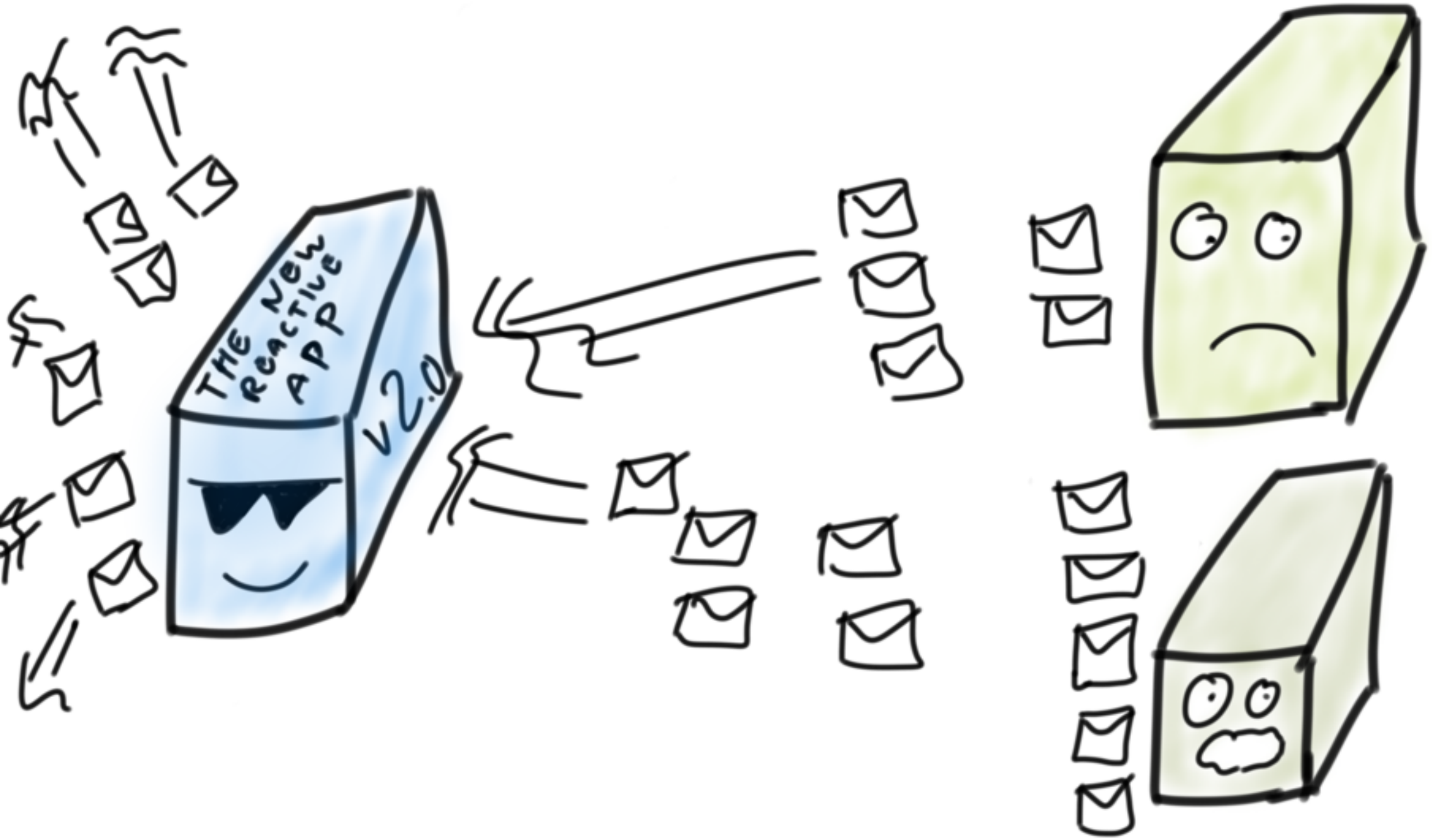
The reason we started researching into transparent to users flow control.

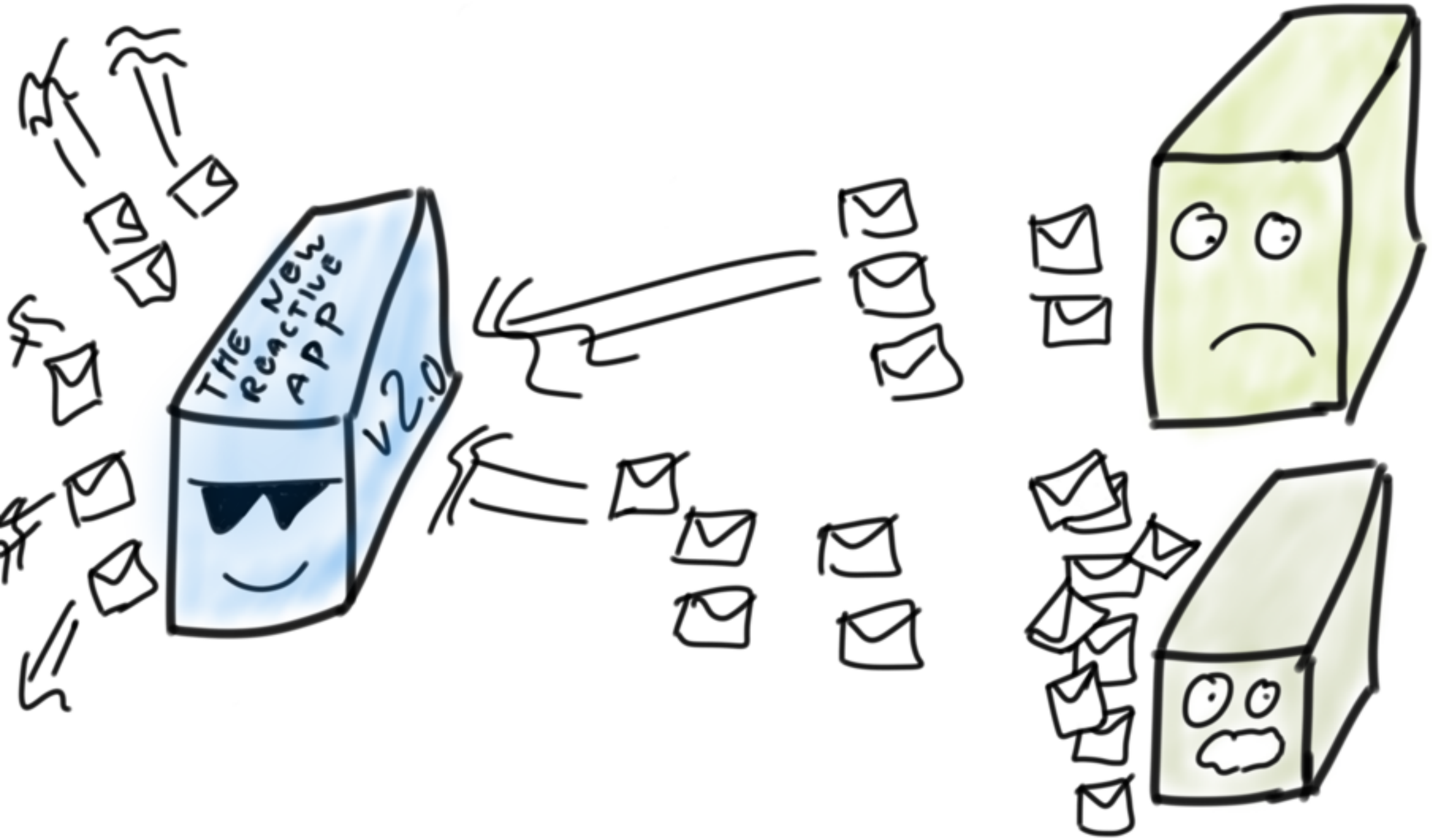

THE NEW REACTIVE APP
V2.0

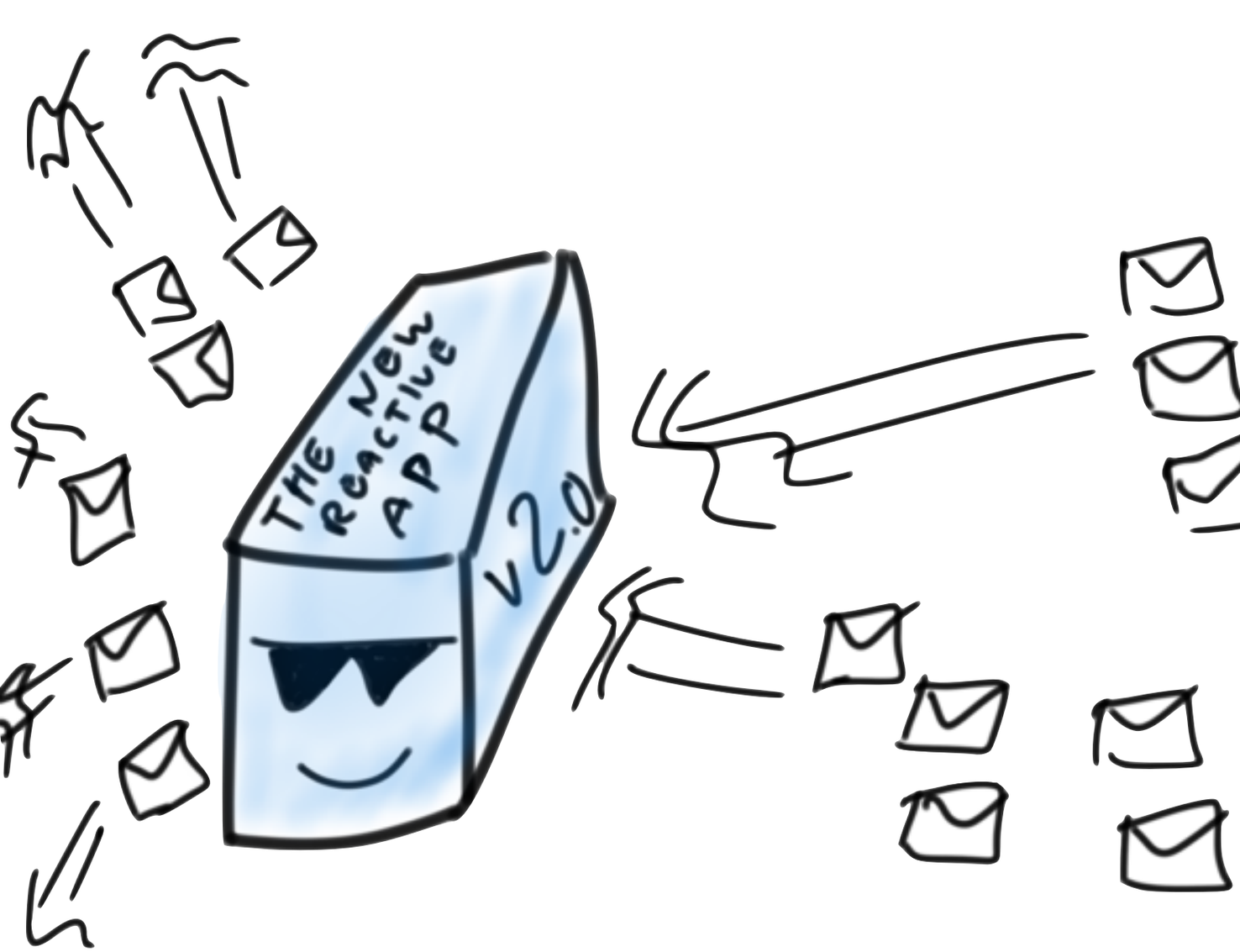


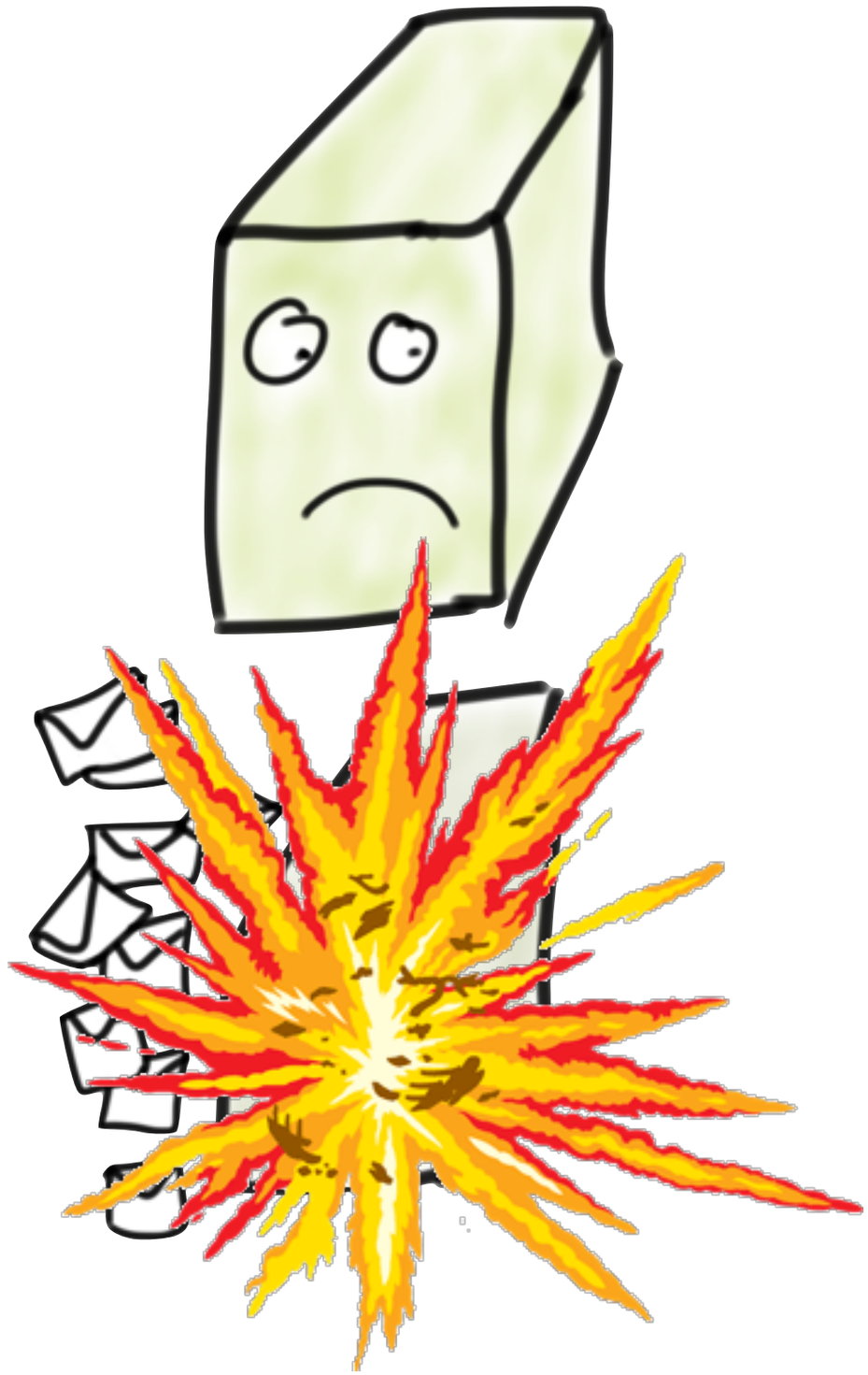
HANDLES
TRAFFIC
LIKE A BOSS.


THE NEW REACTIVE APP
V2.0


THE NEW REACTIVE APP
V2.0


THE NEW REACTIVE APP
V2.0

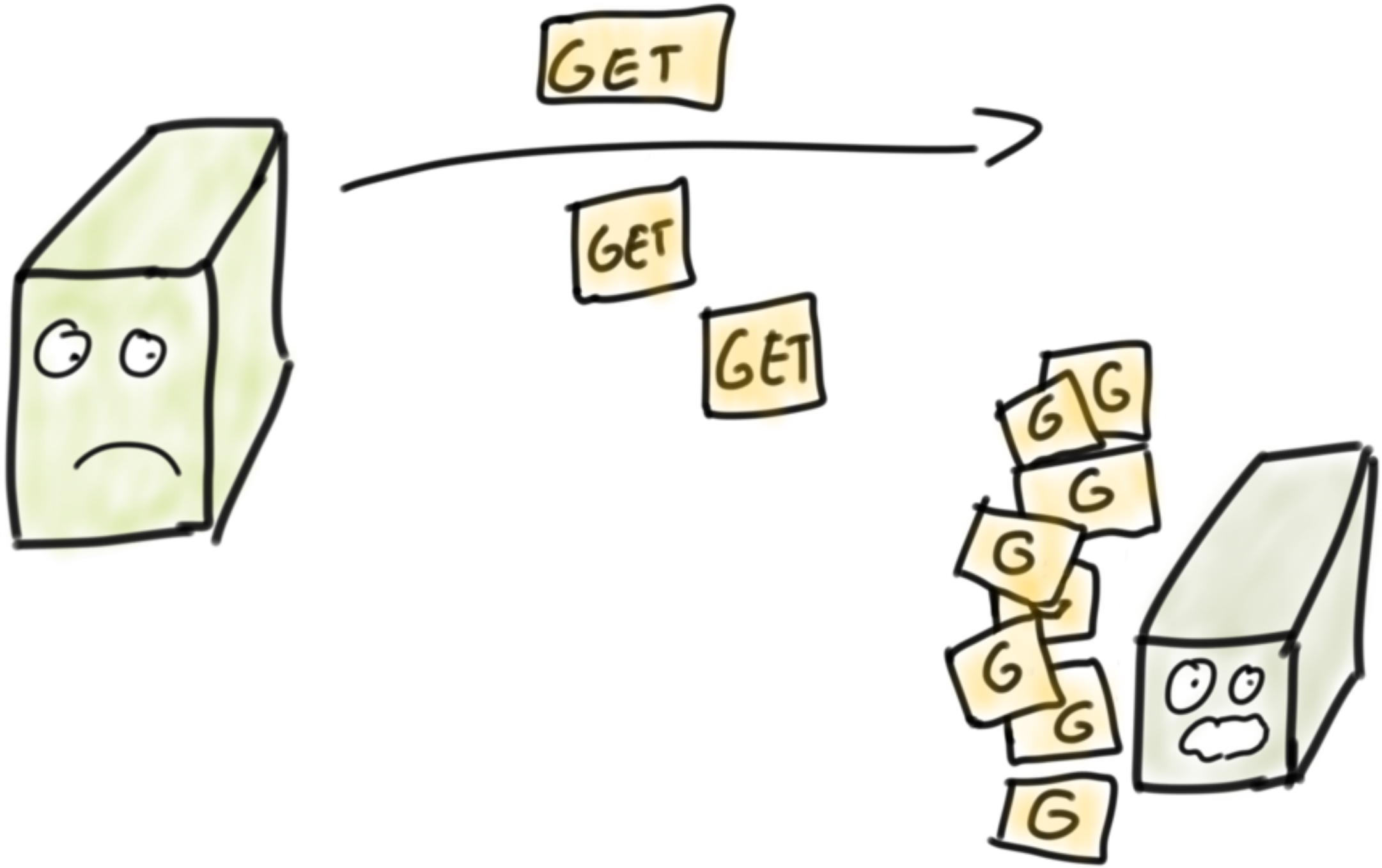

THE NEW REACTIVE APP
V2.0



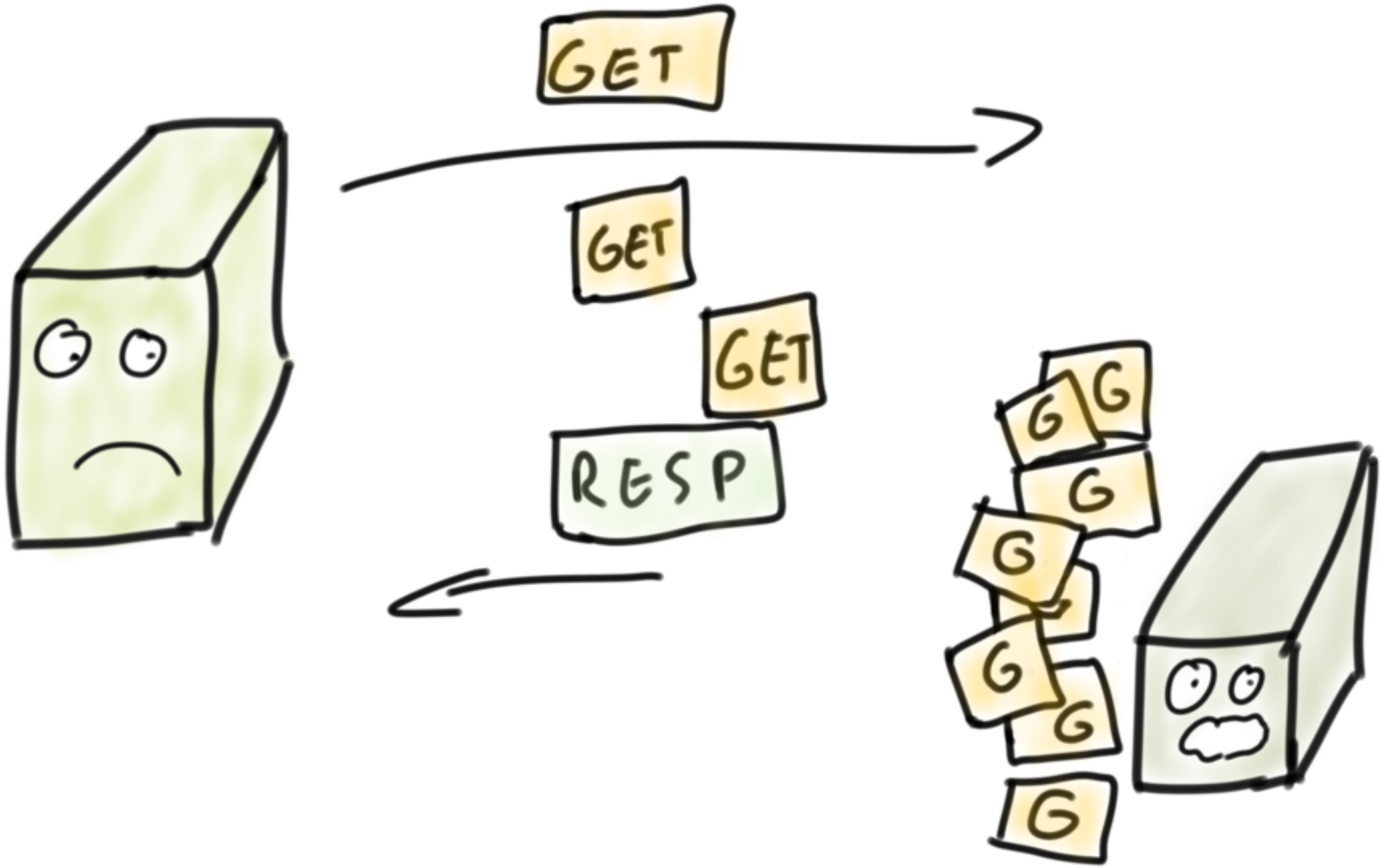
“Best practices are solutions to yesterdays problems.”

Circuit breaking as substitute of flow-control

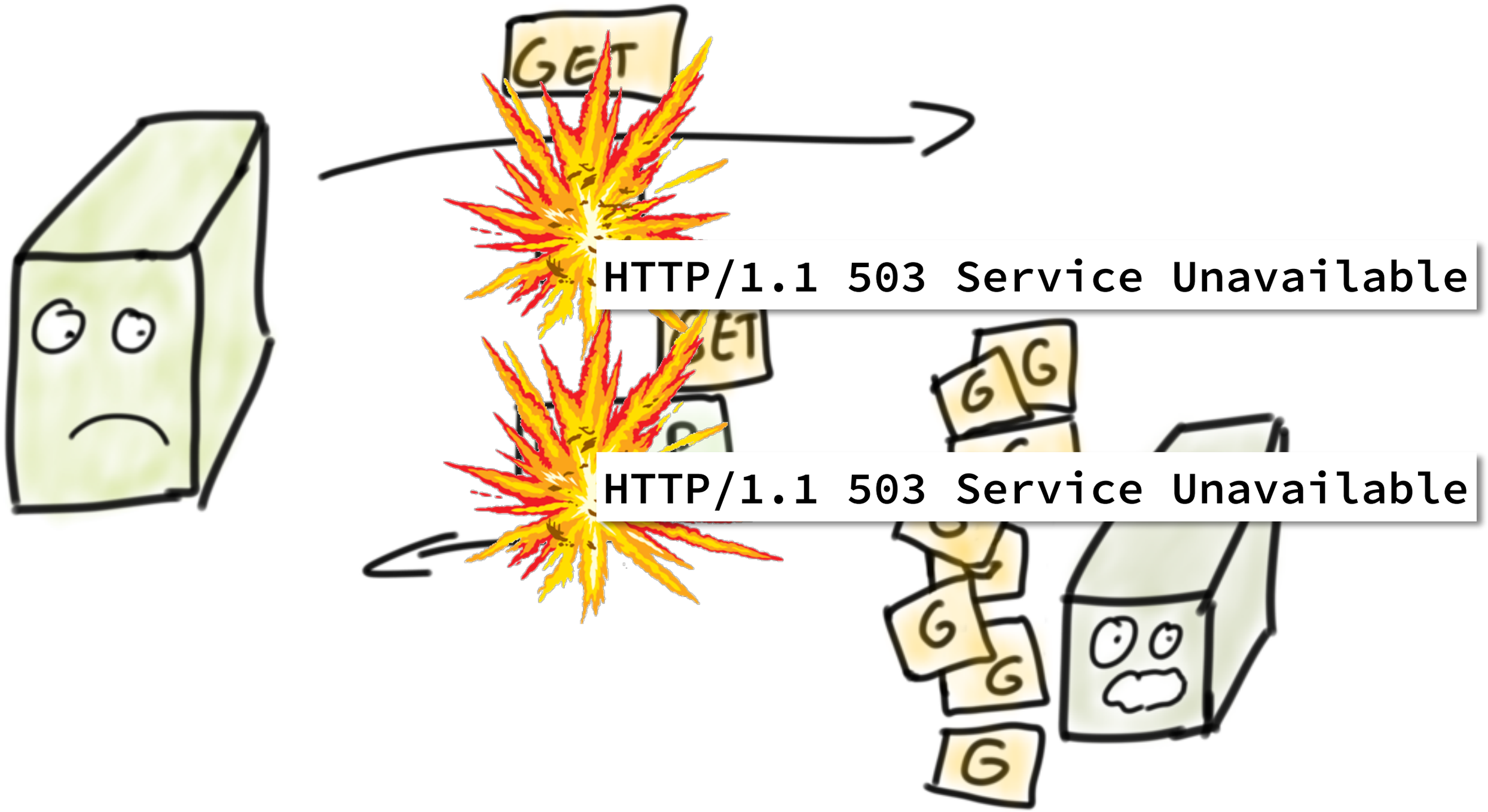
https://twitter.com/FrankBuytendijk/status/795555578592555008



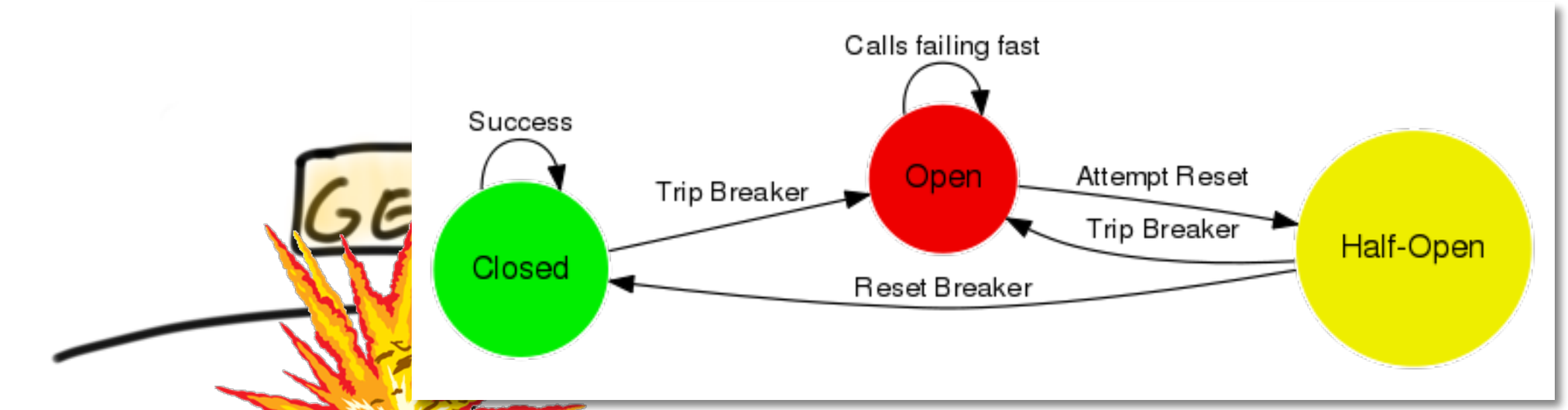
See also, Nitesh Kant, Netflix @ Reactive Summit
https://www.youtube.com/watch?v=5FE6xnH5Lak



See also, Nitesh Kant, Netflix @ Reactive Summit
https://www.youtube.com/watch?v=5FE6xnH5Lak



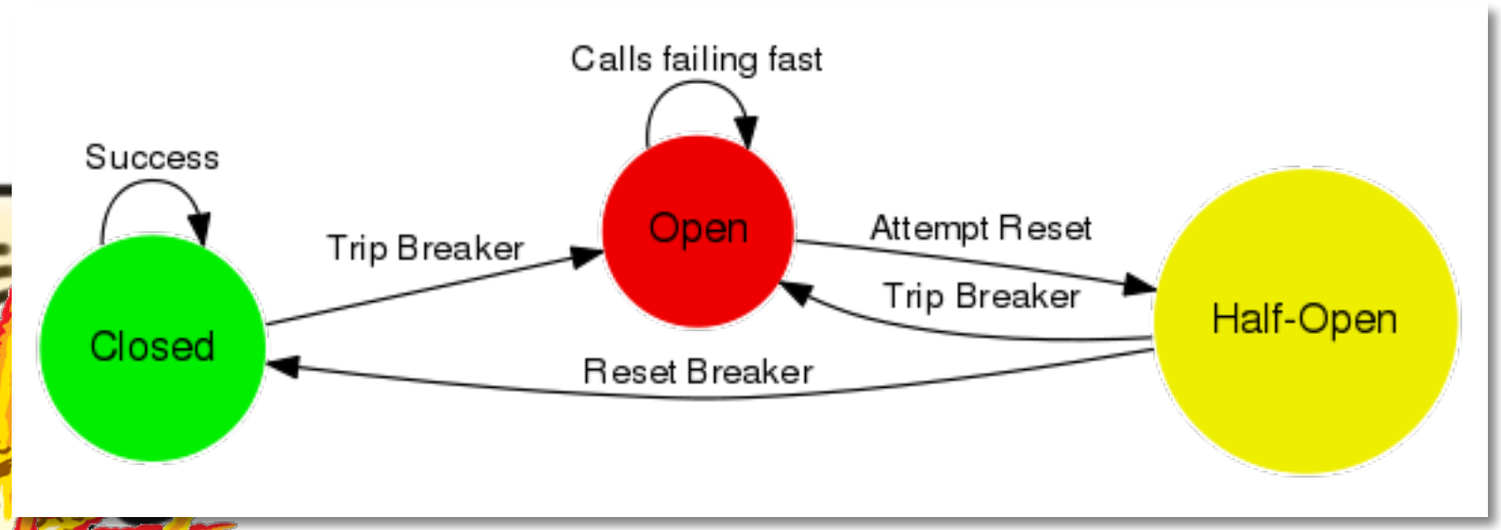
Throttling as represented by 503 responses. Client will back-off… but how?
What if most of the fleet is throttling?



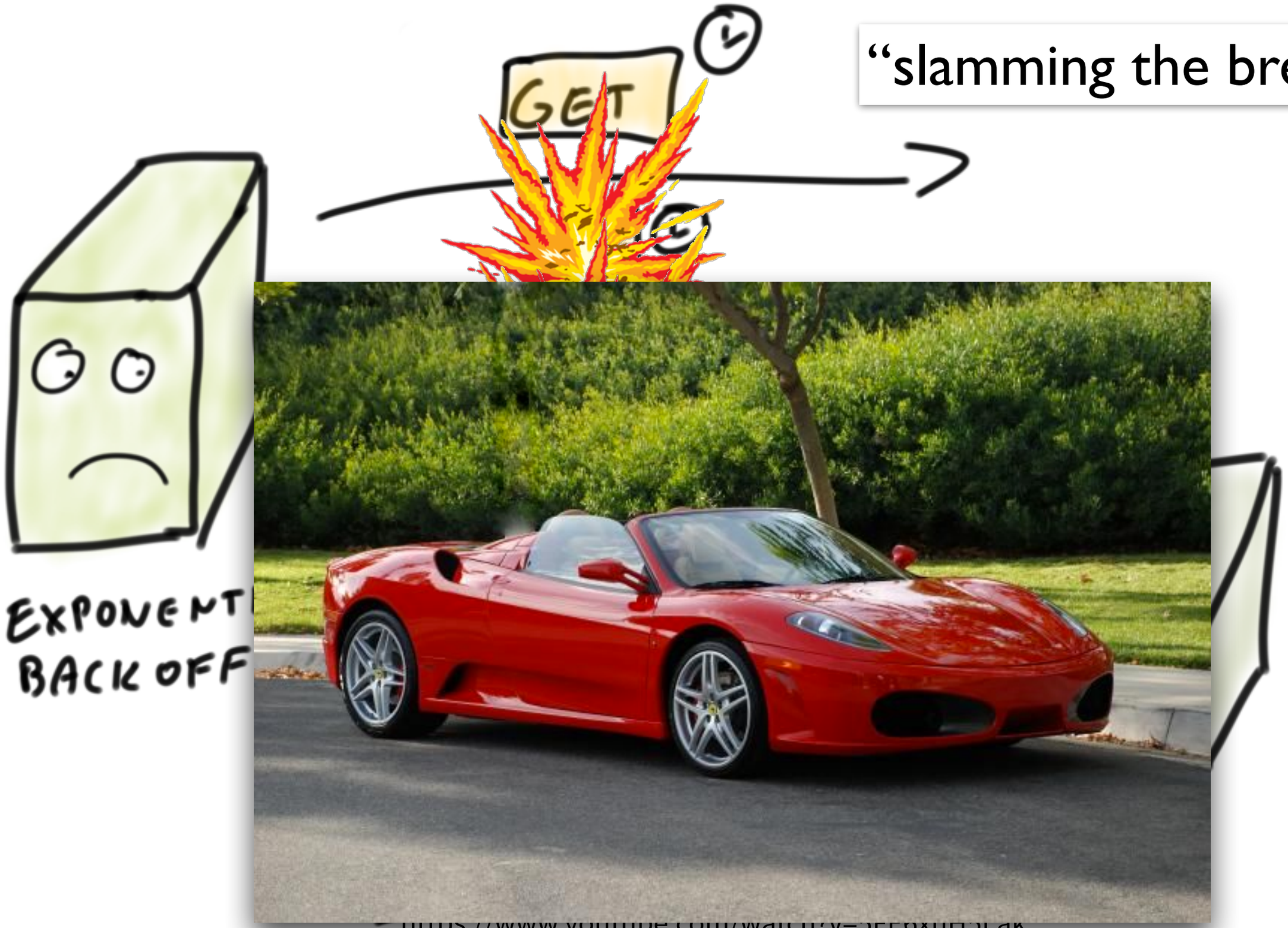
HTTP/1.1 503 Service Unavailable

HTTP/1.1 503 Service Unavailable

http://doc.akka.io/docs/akka/2.4/common/circuitbreaker.html

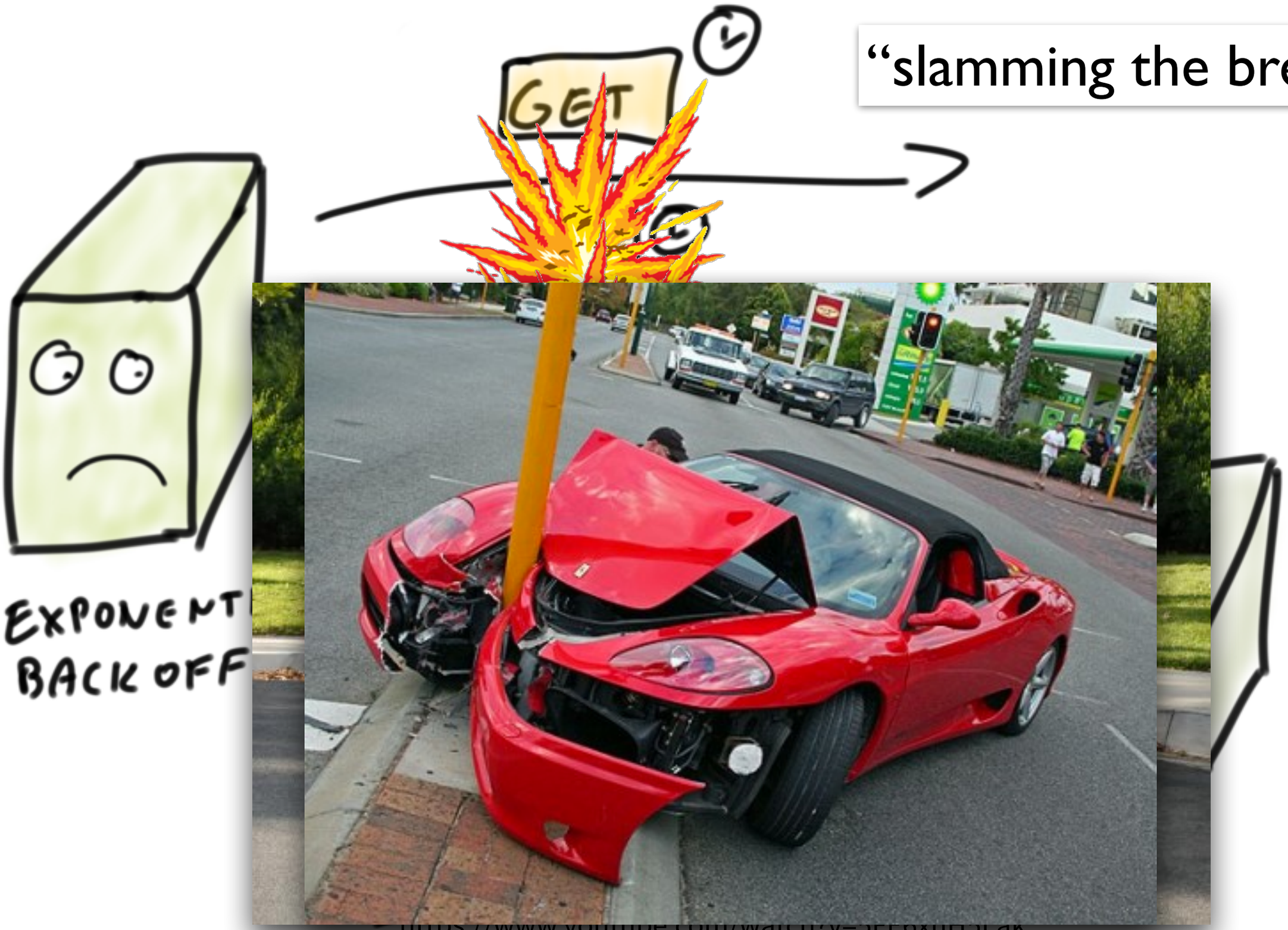


http://doc.akka.io/docs/akka/2.4/common/circuitbreaker.html



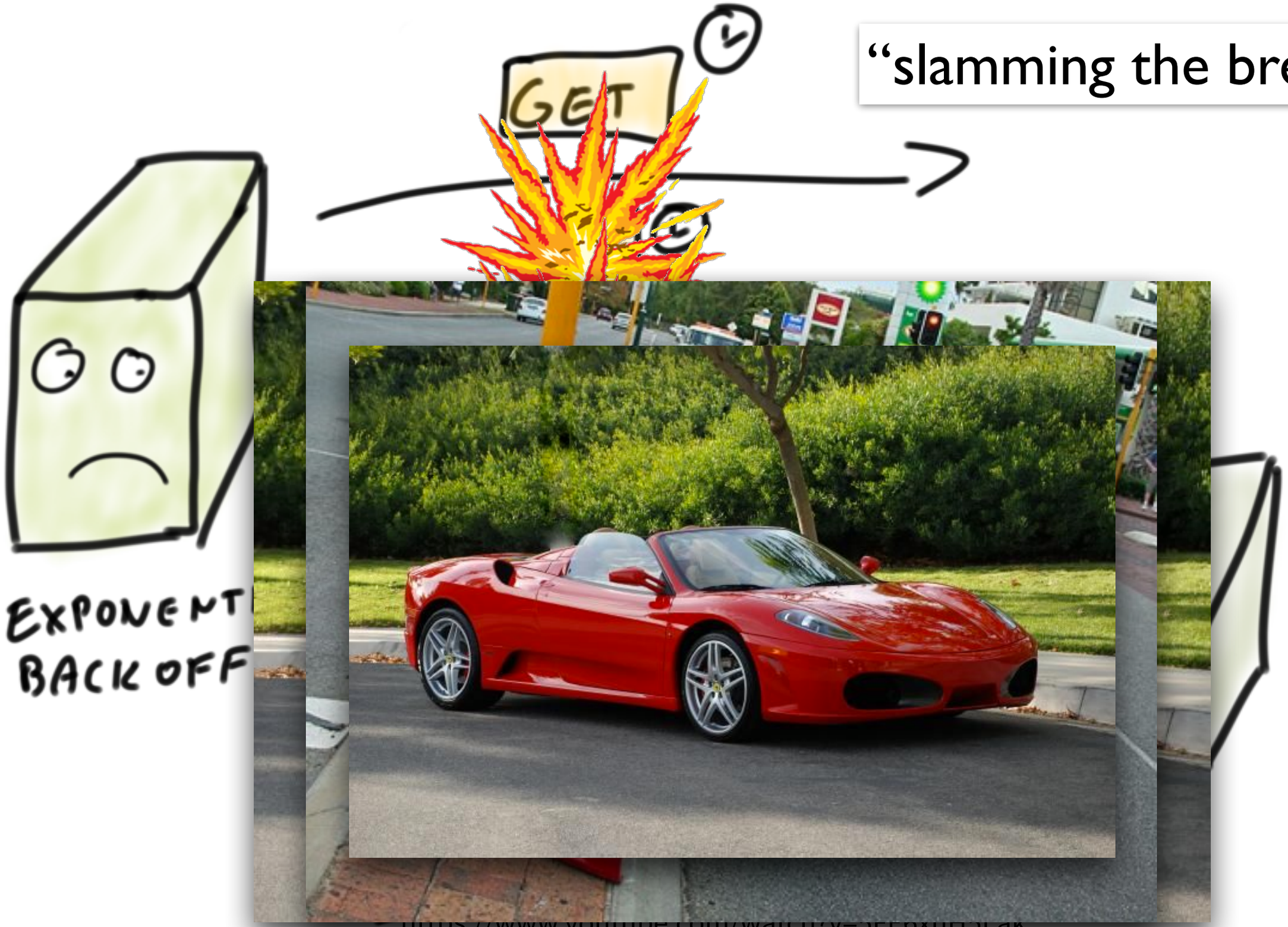
"slamming the breaks"

https://www.youtube.com/watch?v=3PE6xnH3Eak



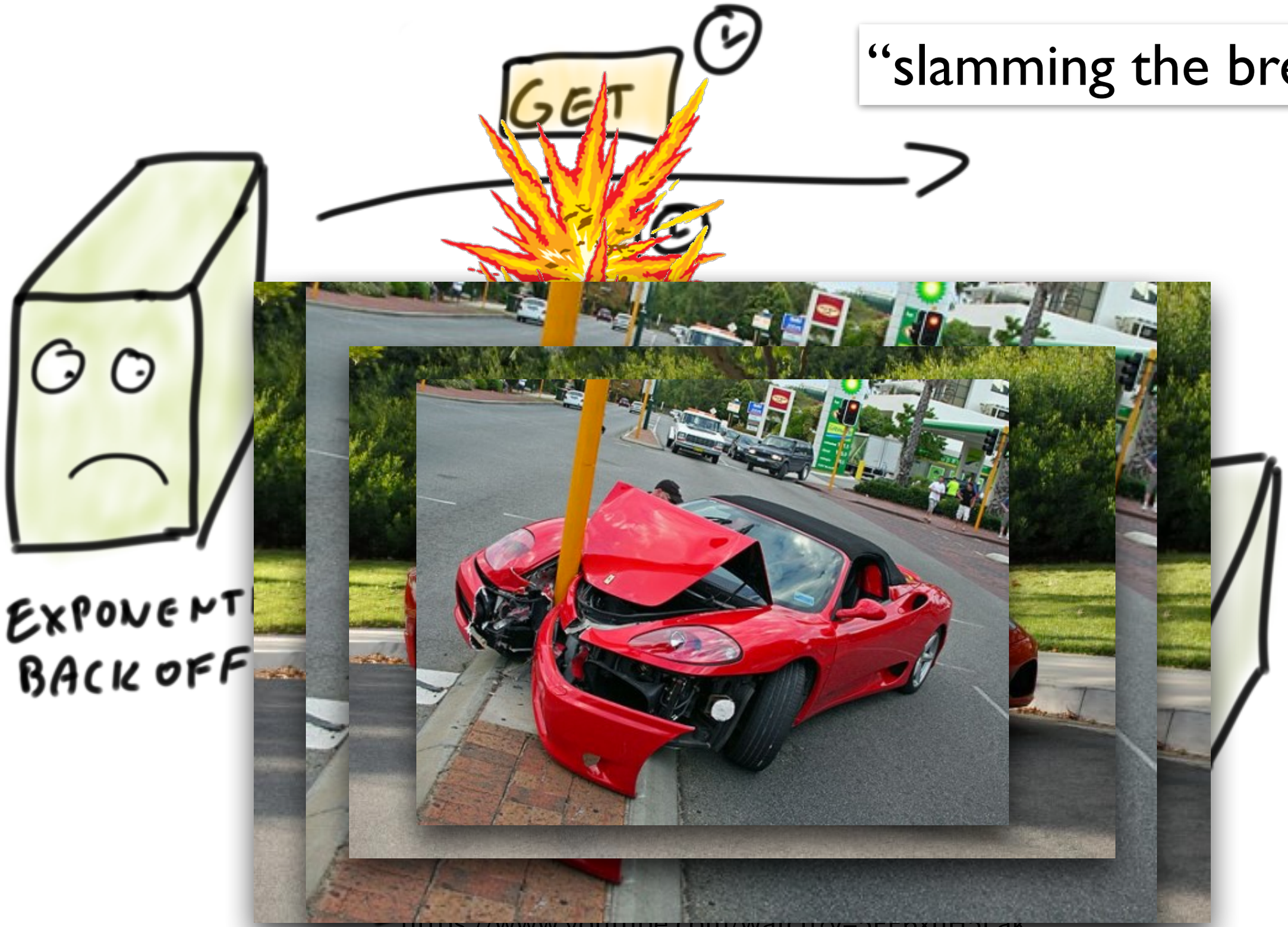
"slamming the breaks"

https://www.youtube.com/watch?v=5PE6xHH5Eak



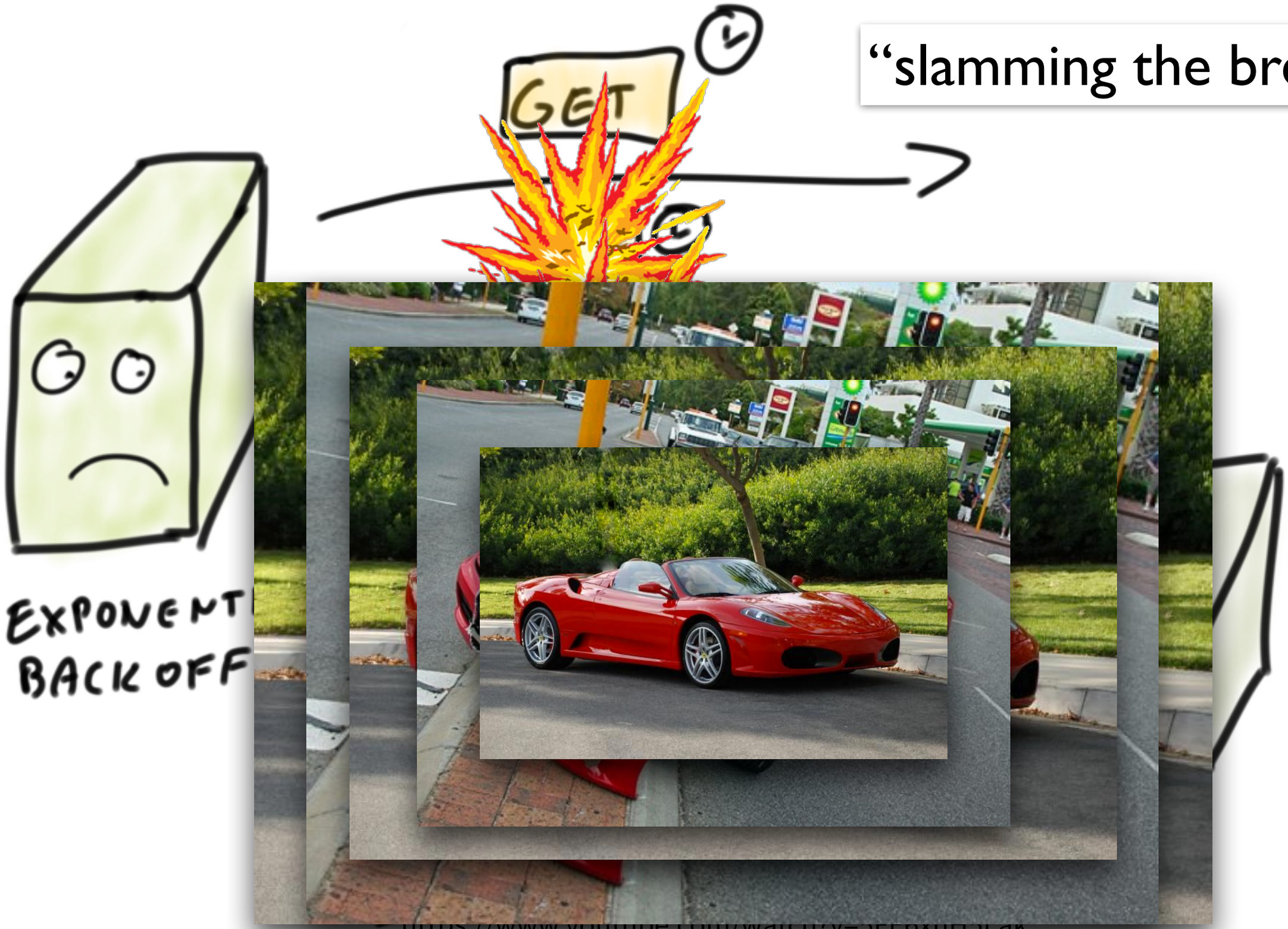
"slamming the breaks"

https://www.youtube.com/watch?v=5PE6xHH5Eak



"slamming the breaks"

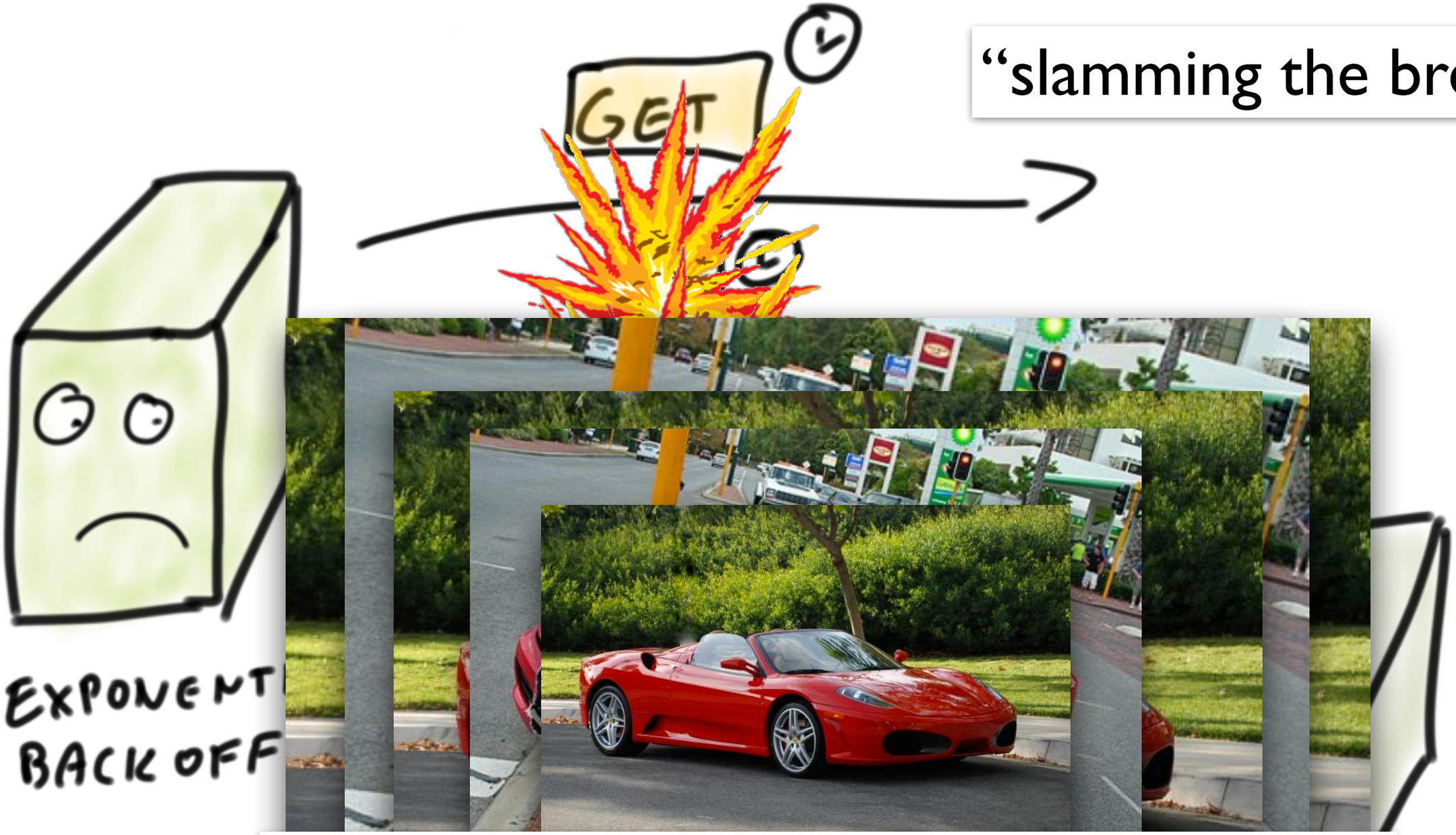
https://www.youtube.com/watch?v=5PE6xHH5Eak



"slamming the breaks"

https://www.youtube.com/watch?v=5PE6xnH5Eak

"slamming the breaks"

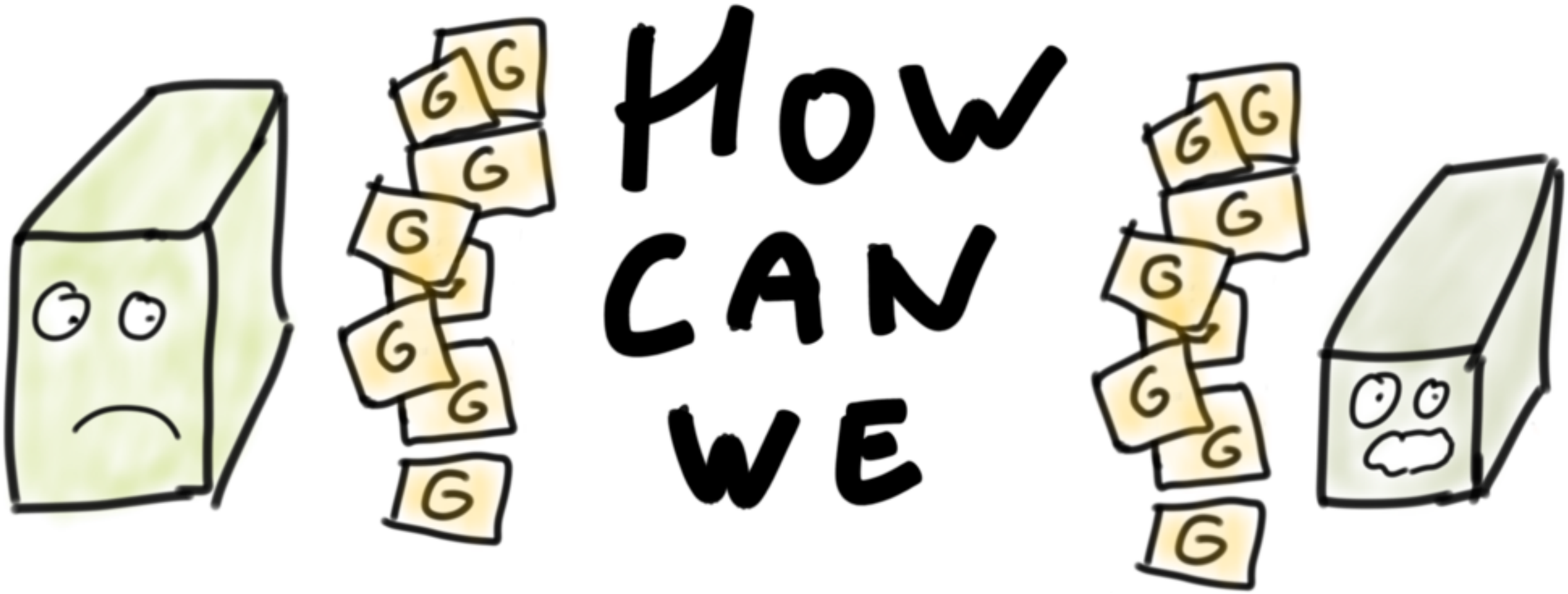


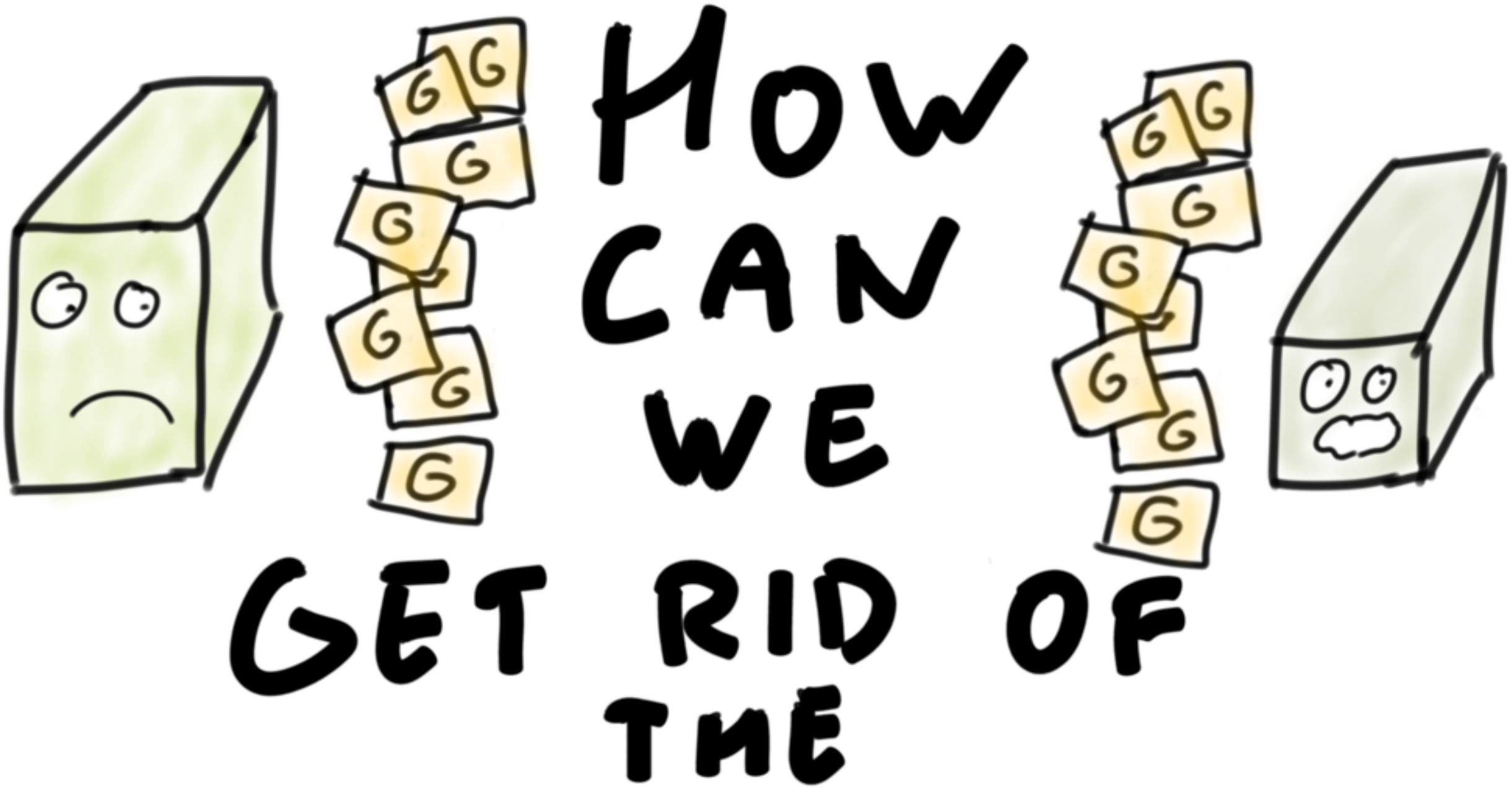
We'll re-visit this specific case in a bit :-)

Circuit Breakers

Are absolutely useful!

Still… “We can do better than that.”


HOW
CAN
WE


HOW CAN WE GET RID OF THE


HOW CAN WE GET RID OF THE GUESSWORK ??

We can do better.

But we'll need everyone to understand some shared semantics…

Reactive Streams

A **fundamental building block.**

Not end-user API by itself.

reactive-streams.org

Reactive Streams

More of an **SPI** (Service Provider Interface), than API.

reactive-streams.org

“Stream”

“Stream”
What does it mean?!

“Streams”

Suddenly everyone jumped on the word “Stream”.

Akka Streams / Reactive Streams started end-of-2013.

* when put in “” the word does not appear in project name, but is present in examples / style of APIs / wording.

“Streams”

Suddenly everyone jumped on the word “Stream”.

Akka Streams / Reactive Streams started end-of-2013.

Akka Streams
Reactive Streams
RxJava “streams”*
Spark Streaming
Apache Storm “streams”*
Java Steams (JDK8)
Reactor “streams”*
Kafka Streams
ztellman / Manifold (Clojure)
Apache GearPump “streams”
Apache [I] Streams (!)
Apache [I] Beam “streams”
Apache [I] Quarks “streams”
Apache [I] Airflow “streams” (dead?)
Apache [I] Samza
Scala Stream
Scalaz Streams, now known as FS2
Swave.io
Java InputStream / OutputStream / … :-)

* when put in “” the word does not appear in project name, but is present in examples / style of APIs / wording.

Origins of

Reactive Streams

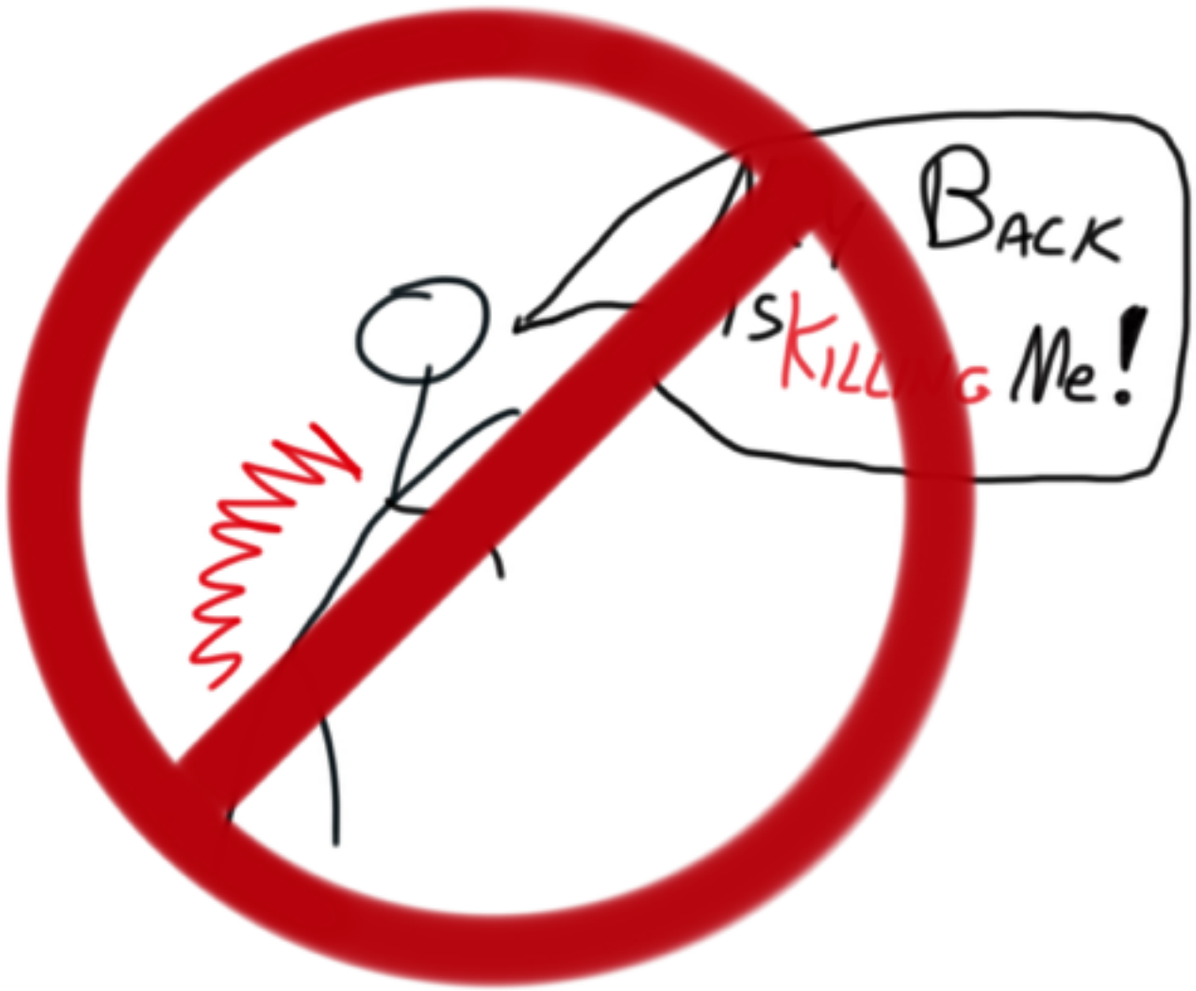
The specification.

What is back-pressure?

?

What is back-pressure?

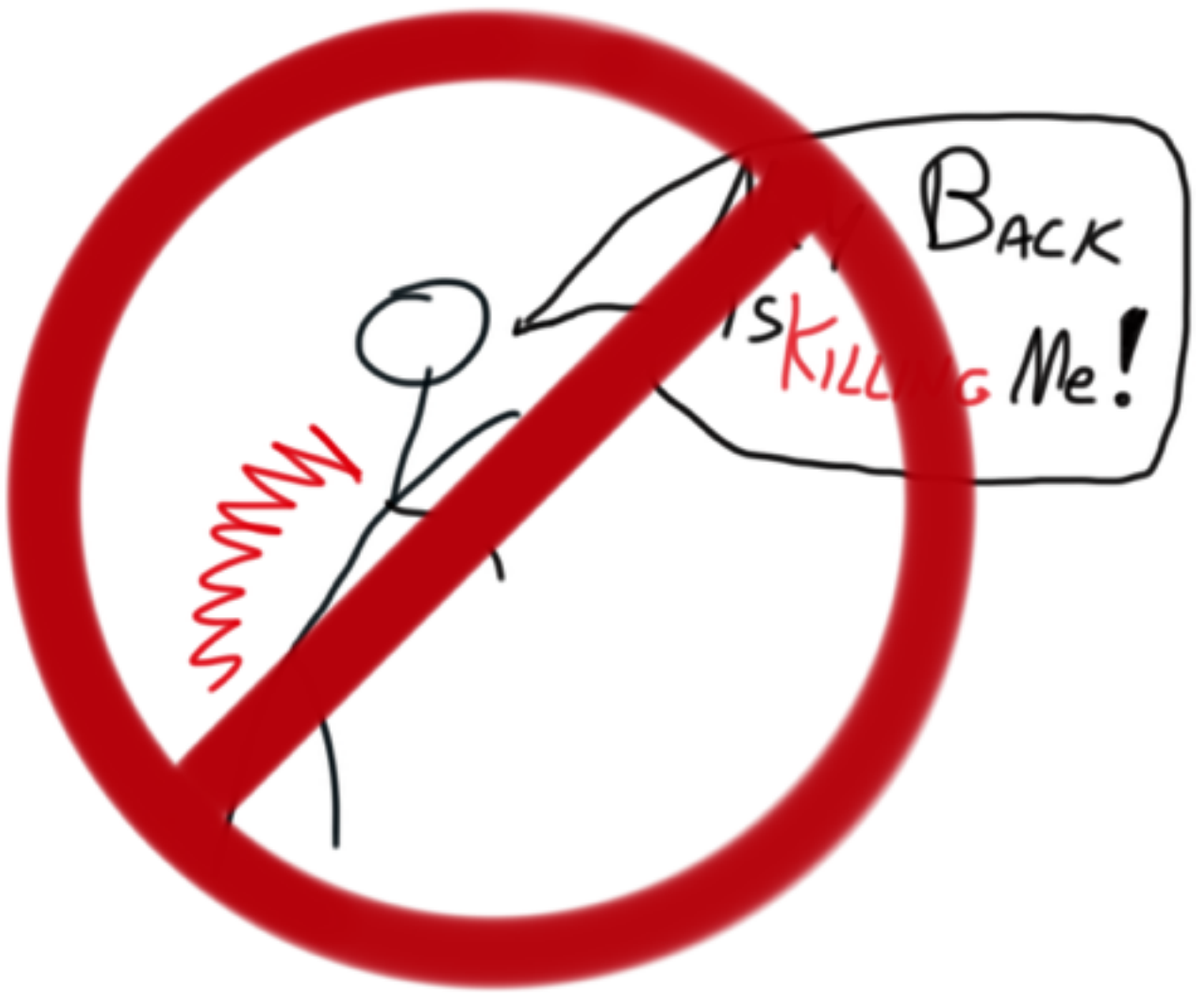
What is back-pressure?



No no no…!
Not THAT Back-pressure!

Also known as:
flow control.

What is back-pressure?



No no no…!
Not THAT Back-pressure!

Also known as:

application level **flow control.**

Reactive Streams - story: 2013's impls

~2013:

Reactive Programming
becoming widely adopted on JVM.

- **Play** introduced "**Iteratees**"
- **Akka** (2009) had **Akka-IO** (TCP etc.)
- **Ben** starts work on **RxJava**

} Teams discuss need for back-pressure in simple user API.
Play's Iteratee / Akka's NACK in IO.

http://blogs.msdn.com/b/rxteam/archive/2009/11/17/announcing-reactive-extensions-rx-for-net-silverlight.aspx
http://infoscience.epfl.ch/record/176887/files/DeprecatingObservers2012.pdf - Ingo Maier, Martin Odersky
https://github.com/ReactiveX/RxJava/graphs/contributors
https://github.com/reactor/reactor/graphs/contributors
https://medium.com/@viktorklang/reactive-streams-1-0-0-interview-faaca2c00bec#.69st3rndy

Reactive Streams - story: 2013's impls

Play Iteratees – pull back-pressure, difficult API

Akka-IO – NACK back-pressure; low-level IO (Bytes); messaging API

RxJava – no back-pressure, nice API

http://blogs.msdn.com/b/rxteam/archive/2009/11/17/announcing-reactive-extensions-rx-for-net-silverlight.aspx
http://infoscience.epfl.ch/record/176887/files/DeprecatingObservers2012.pdf - Ingo Maier, Martin Odersky
https://github.com/ReactiveX/RxJava/graphs/contributors
https://github.com/reactor/reactor/graphs/contributors
https://medium.com/@viktorklang/reactive-streams-1-0-0-interview-faaca2c00bec#.69st3rndy

Reactive Streams - expert group founded

October 2013

Roland Kuhn (Akka) and Erik Meijer (Rx .NET) meet in Lausanne,
while recording **“Principles of Reactive Programming” Coursera Course**.

Viktor Klang (Akka), Erik Meijer, Ben Christensen (RxJava)
and Marius Eriksen (Twitter) meet at Twitter HQ.

The term “**reactive non-blocking asynchronous back-pressure**” gets coined.

Afterwards more organisations are invited to join the effort, including Pivotal, RedHat etc.

Reactive Streams - expert group founded

October 2013

Roland Kuhn (Akka) and Erik Meijer (Rx .NET) meet in Lausanne,
while recording **“Principles of Reactive Programming” Coursera Course**.

Viktor Klang (Akka), Erik Meijer, Ben Christensen (RxJava)
and Marius Eriksen (Twitter) meet at Twitter HQ.

The term “**reactive non-blocking**

Goals:

- asynchronous
- never block (waste)
- safe (back-threads pressured)
- purely **local abstraction**
- allow synchronous impls.

Reactive Streams - expert group founded

October 2013

Roland Kuhn (Akka) and Erik Meijer (Rx .NET) meet in Lausanne,
while recording "Principles of Reactive Programming" Coursera Course.

Viktor Klang (Akka), Erik Meijer, Ben Christensen (RxJava)
and Marius Eriksen (Twitter) meet at Twitter HQ.

The term "reactive non-blocking asynchronous back-pressure" gets coined.

December 2013

Stephane Maldini & Jon Brisbin (Pivotal Reactor) contacted by **Viktor.**

Reactive Streams - expert group founded

October 2013

Roland Kuhn (Akka) and Erik Meijer (Rx .NET) meet in Lausanne,
while recording "Principles of Reactive Programming" Coursera Course.

Viktor Klang (Akka), Erik Meijer, Ben Christensen (RxJava)
and Marius Eriksen (Twitter) meet at Twitter HQ.

The term "**reactive non-blocking asynchronous back-pressure**" gets coined.

December 2013

Stephane Maldini & Jon Brisbin (Pivotal Reactor) contacted by **Viktor.**

Soon after, the "Reactive Streams" expert group is formed.

Also joining the efforts: Doug Lea (Oracle), Endre Varga (Akka), Johannes Rudolph &
Mathias Doenitz (Spray), and many others, including myself join the effort soon after.

Reactive Streams - expert group founded

October 2013

Roland Kuhn (Akka) and Erik M
while recording "Principles of Re

Viktor Klang (Akka), Erik Meije
and Marius Eriksen (Twitter) m

The term "reactive non-blocking

December 2013

Stephane Maldini & Jon Brisbin

Soon after, the "Reactive Streams" expert group is formed.

Also joining the efforts: Doug Lea (Oracle), Endre Varga (Akka), Johannes Rudolph & Mathias Doenitz (Spray), and many others, including myself join the effort soon after.

Reactive Streams - story: 2013's impls

2014–2015:

Reactive Streams Spec & TCK
development, and implementations.

1.0 released on April 28th 2015,
with 5+ accompanying implementations.

2015

Proposed to be included with **JDK9** by **Doug Lea**
via **JEP-266 "More Concurrency Updates"**

http://hg.openjdk.java.net/jdk9/jdk9/jdk/file/6e50b992bef4/src/java.base/share/classes/java/util/concurrent/Flow.java

Reactive Streams - story: 2013's impls

2014–2015:

Reactive Streams Spec & TCK
development, and implementations.

1.0 released on April 28th 2015,
with 5+ accompanying implementations.

2015

Proposed to be included with **JDK9** by **Doug Lea**
via **JEP-266 "More Concurrency Updates"**

http://hg.openjdk.java.net/jdk9/jdk9/jdk/file/6e50b992bef4/src/java.base/share/classes/java/util/concurrent/Flow.java

Reactive Streams

But what does it do!?

Back-pressure explained

Publisher[T]

Subscriber[T]

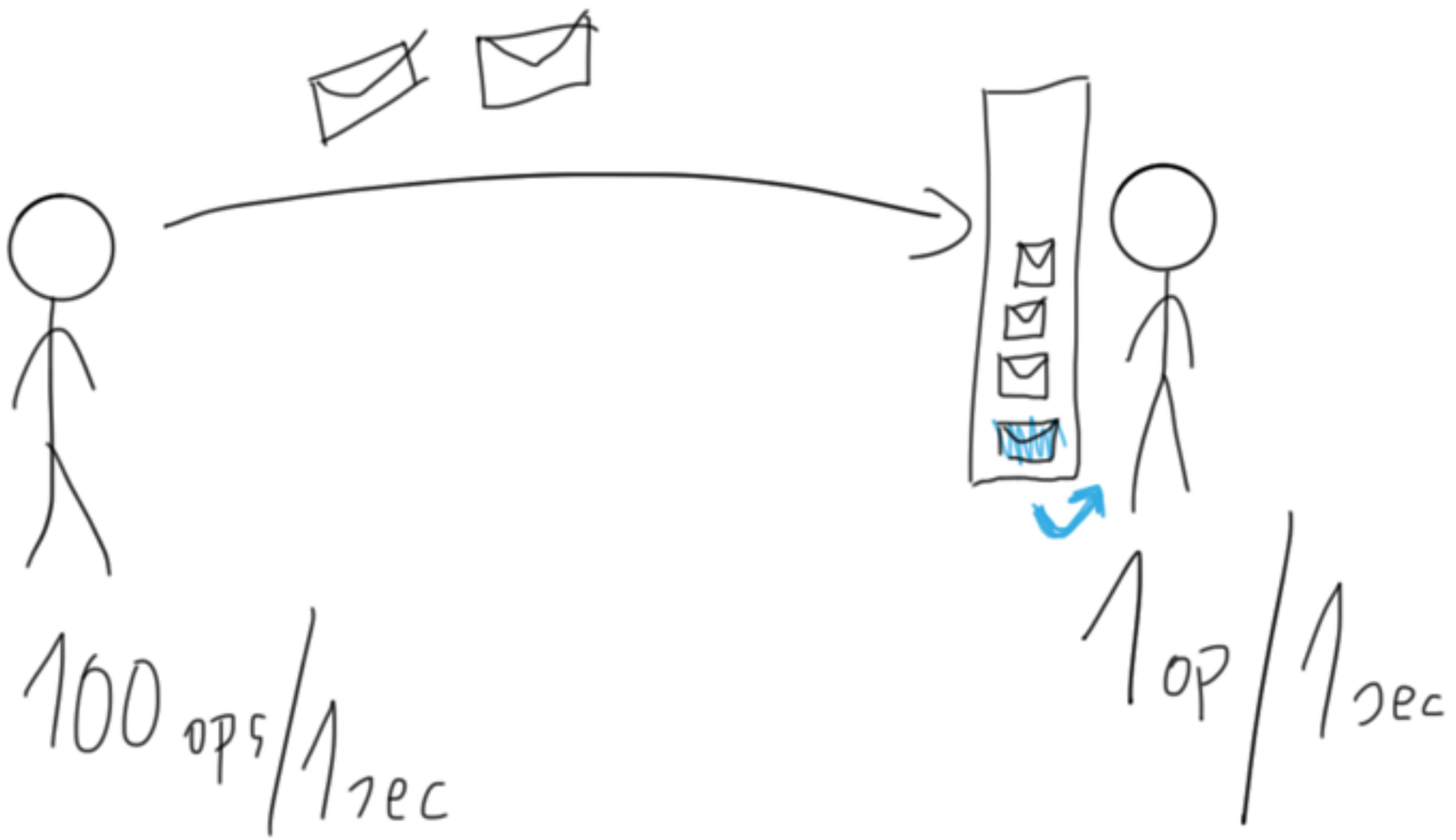
Push model

Fast Publisher

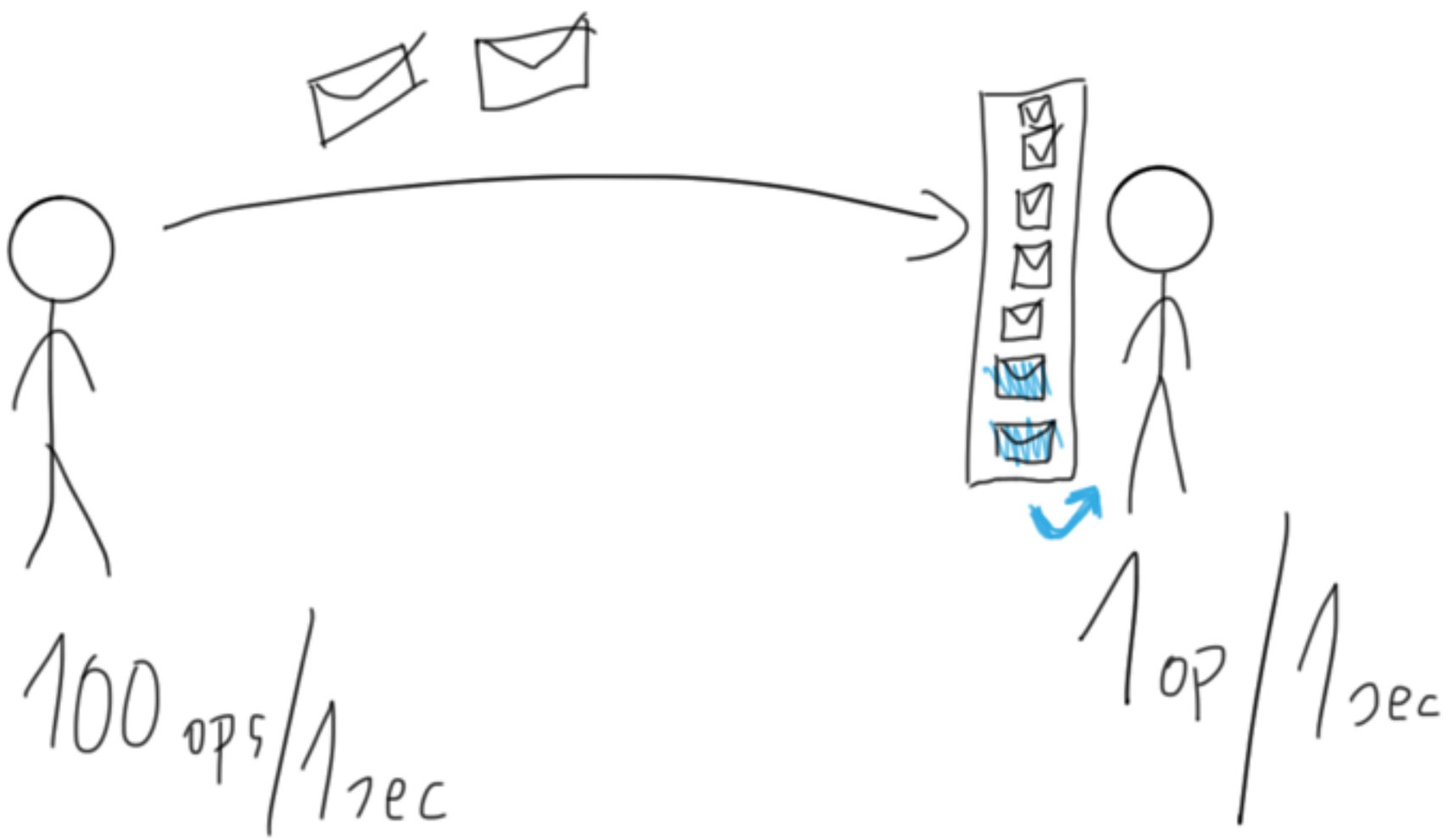
Slow Subscriber

Push model

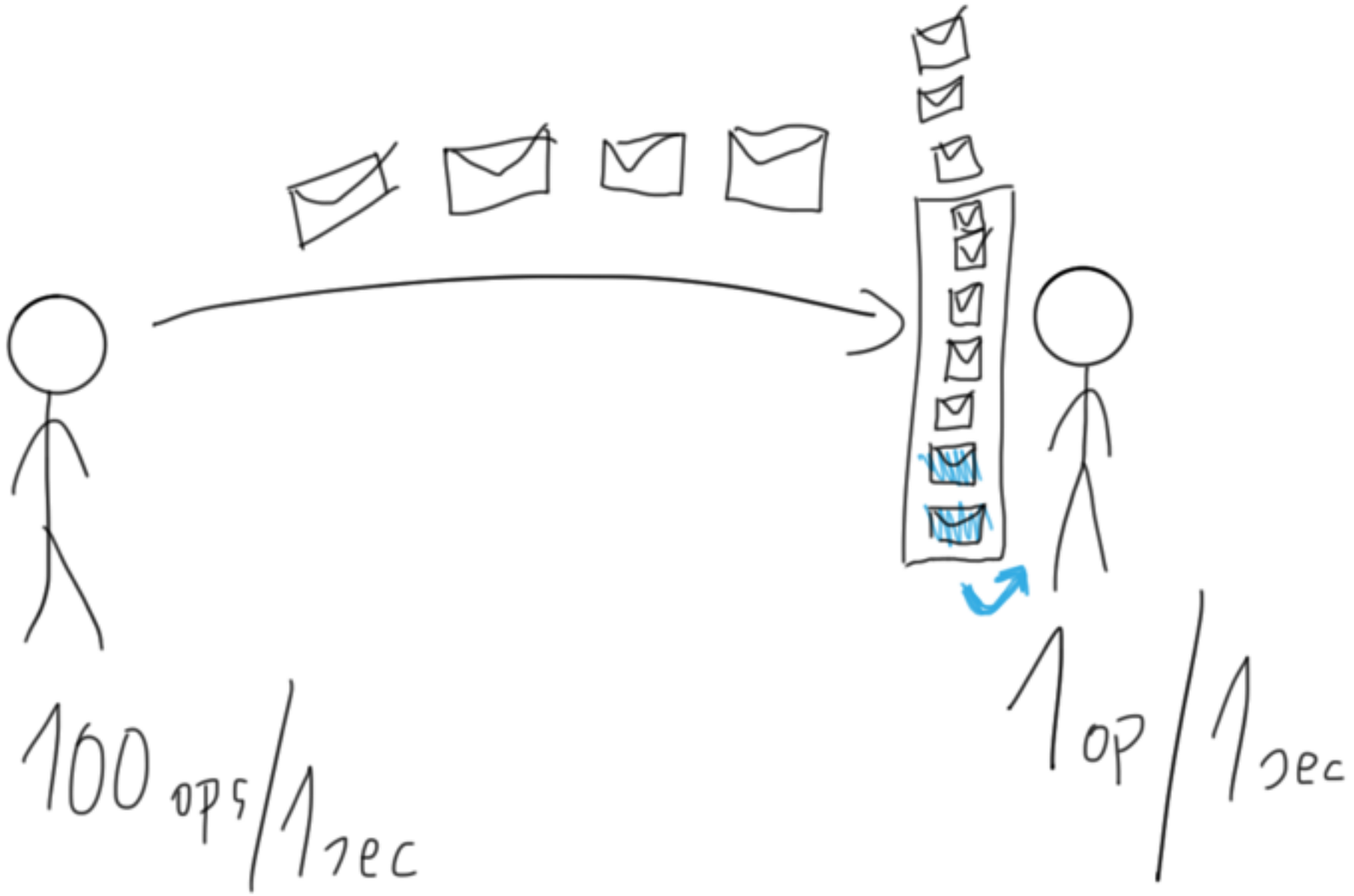
Subscriber usually has some kind of buffer.



Push model

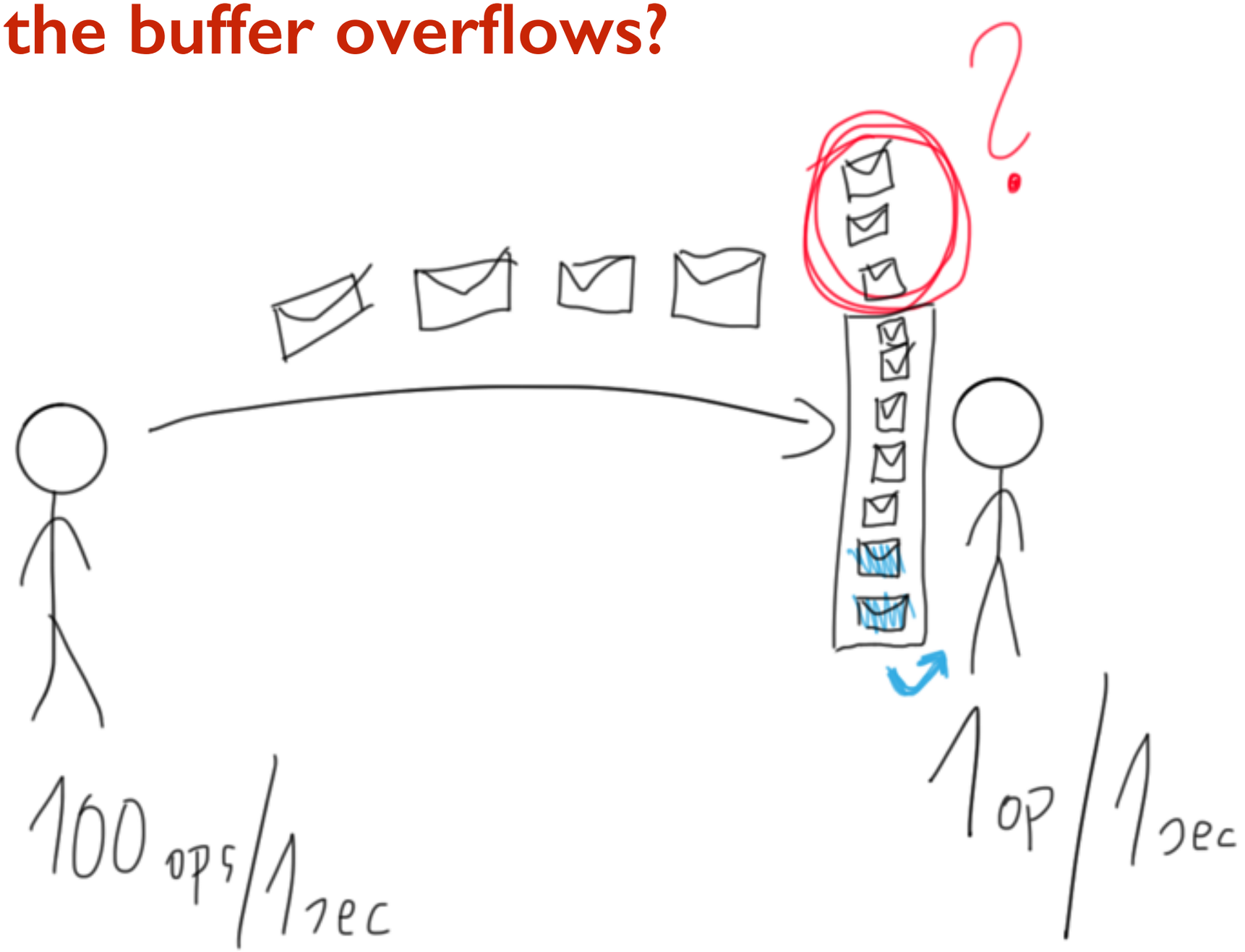


Push model



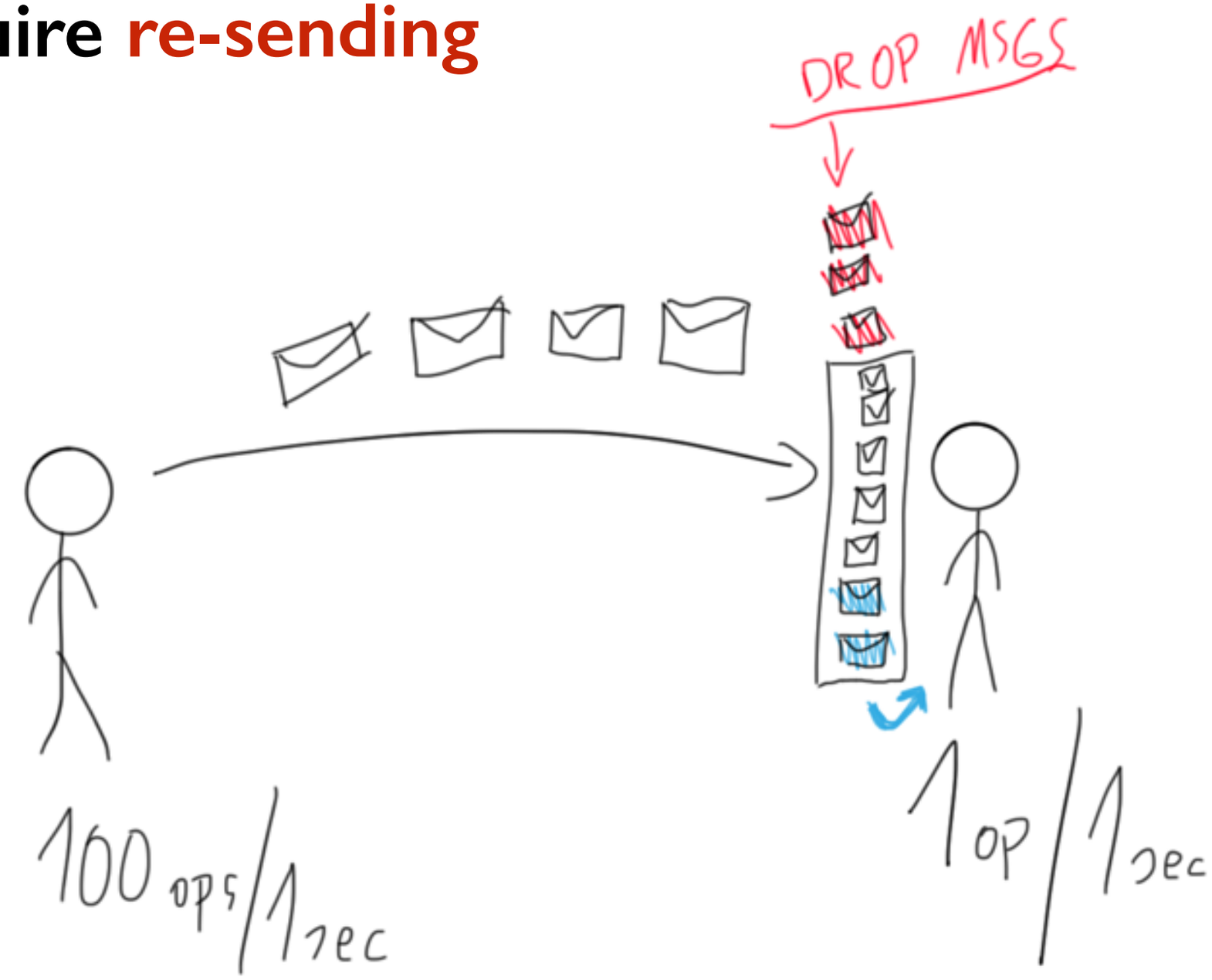
Push model

What if the buffer overflows?



Push model

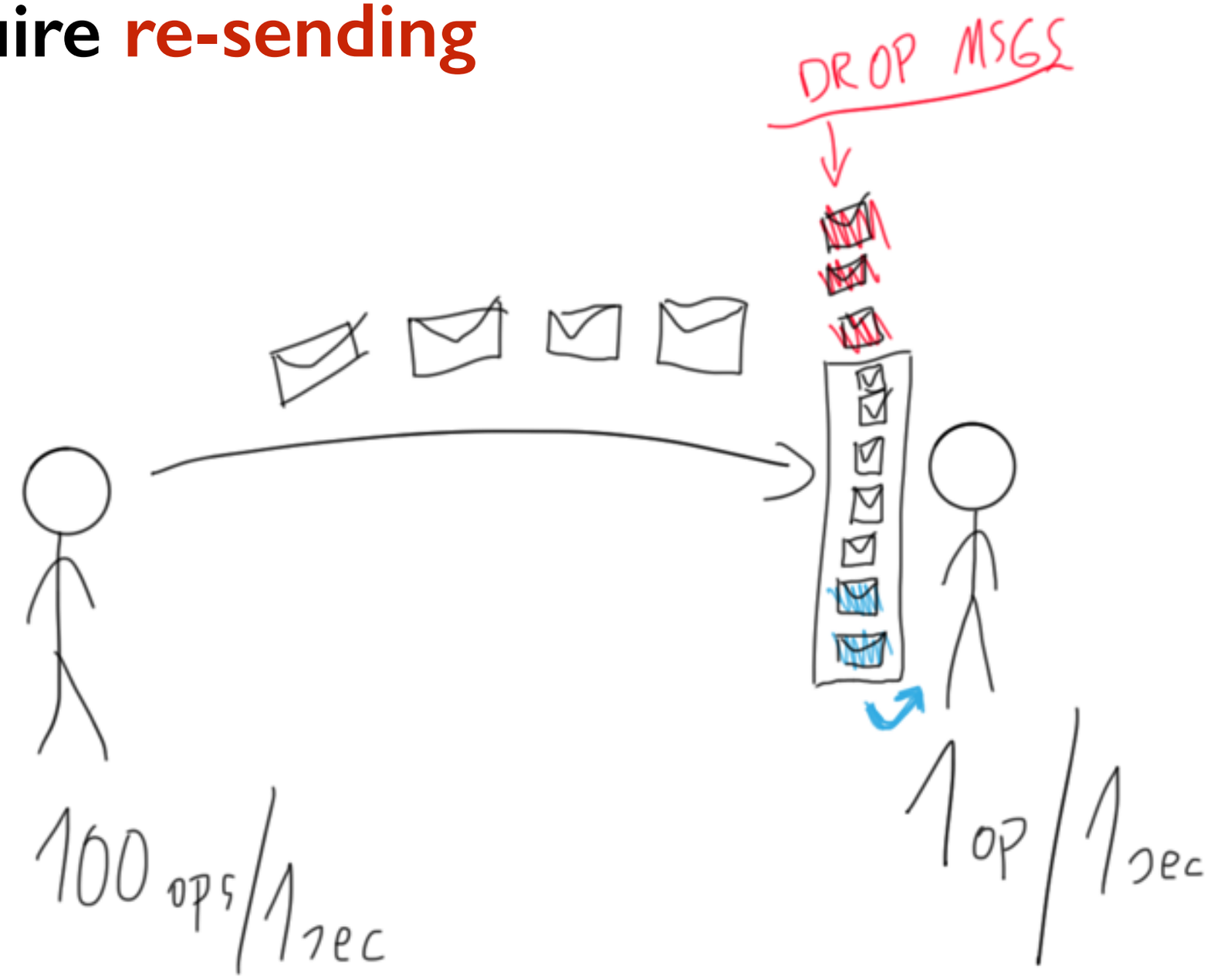
Use bounded buffer,
drop messages + require re-sending



Push model

Use bounded buffer,
drop messages + require re-sending

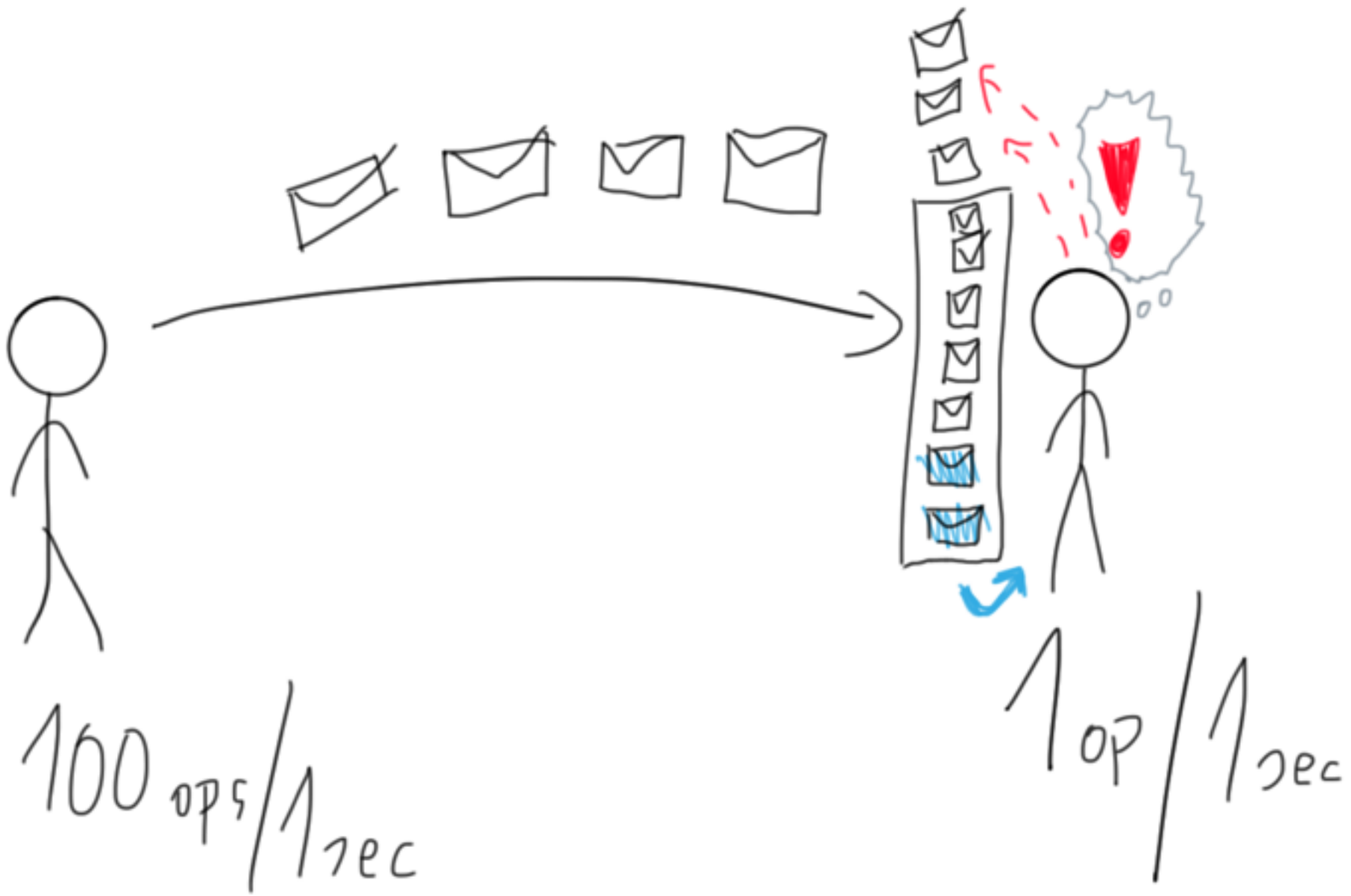
Kernel does this!
Routers do this!
(TCP)



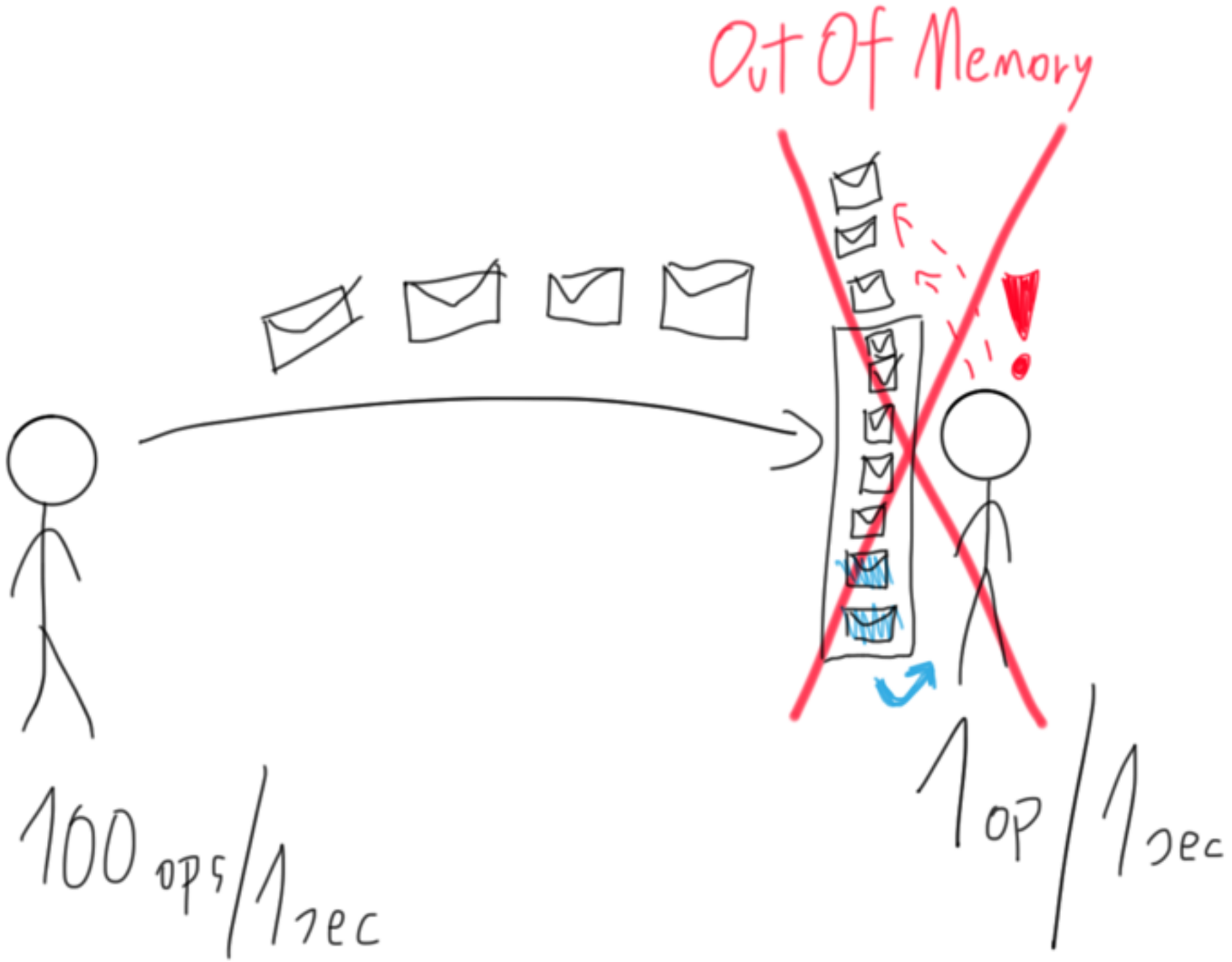
Push model

Increase buffer size…

Well, while you have memory available!



Push model

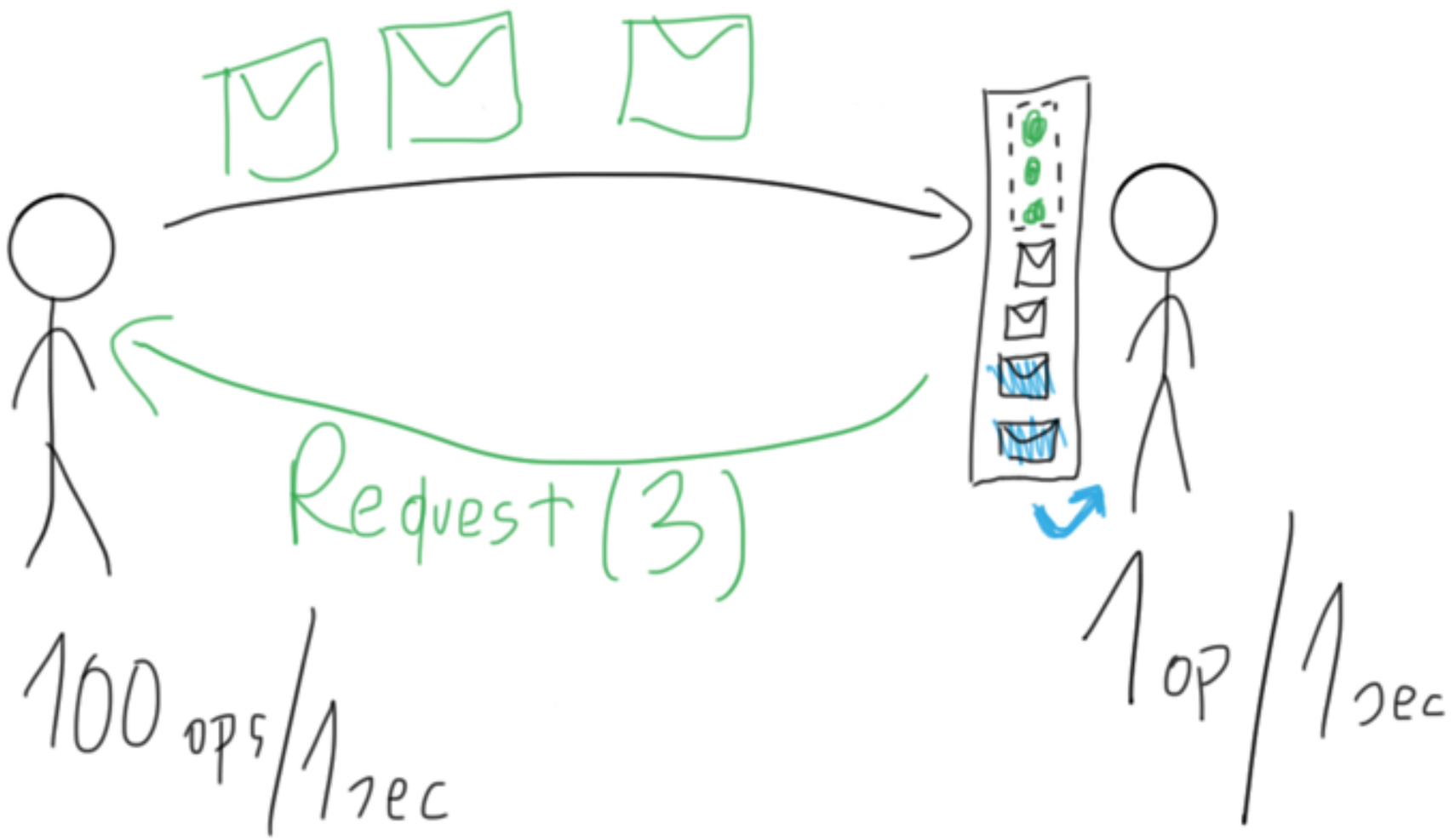


Reactive Streams explained in 1 slide

Reactive Streams: “dynamic push/pull”

Fast Publisher will send at-most 3 elements.

This is pull-based-backpressure.



JEP-266 – soon...!

```
public final class Flow {
    private Flow() {} // uninstantiable

  @FunctionalInterface
    public static interface Publisher<T> {
       public void subscribe(Subscriber<? super T> subscriber);
    }


    public static interface Subscriber<T> {
      public void onSubscribe(Subscription subscription)
      public void onNext(T item);
      public void onError(Throwable throwable);
      public void onComplete();
    }


    public static interface Subscription {
       public void request(long n);
       public void cancel();
    }

    public static interface Processor<T,R> extends Subscriber<T>, Publisher<R> {
    }
}
```

JEP-266 – soon...!

```
public final class Flow {
    private Flow() {} // uninstantiable

  @FunctionalInterface
    public static interface Publisher<T> {
       public void subscribe(Subscriber<? super T> subscriber);
    }


    public static interface Subscriber<T> {
    }


    public static interface Subscription {
       public void request(long n);
       public void cancel();
    }

    public static interface Processor<T,R> extends Subscriber<T>, Publisher<R> {
    }
}
```

Single basic (helper) implementation available in JDK:
SubmissionPublisher

JEPs targeted to JDK 9, so far

102: Process API Updates
110: HTTP 2 Client
158: Unified JVM Logging
193: Variable Handles
199: Smart Java Compilation, Phase Two
200: The Modular JDK
201: Modular Source Code
211: Elide Deprecation Wa
212: Resolve Lint and Docl
213: Milling Project Coin
214: Remove GC Combina
215: Tiered Attribution for
216: Process Import Stater
217: Annotations Pipeline
219: Datagram Transport L
220: Modular Run-Time Im
222: jshell: The Java Shell
224: HTML5 Javadoc
225: Javadoc Search
226: UTF-8 Property Files
227: Unicode 7.0
228: Add More Diagnostic Commands
229: Create PKCS12 Keystores by Default
231: Remove Launch-Time JRE Version Selection
232: Improve Secure Application Performance
233: Generate Run-Time Compiler Tests Automat
235: Test Class-File Attributes Generated by Java
236: Parser API for Nashorn
237: Linux/AArch64 Port
238: Multi-Release JAR Files
40: Remove the JVM TI hprof Agent
41: Remove the jhat Tool
43: Java-Level JVM Compiler Interface
244: TLS Application-Layer Protocol Negotiation
245: Validate JVM Command-Line Flag Argument
246: Leverage CPU Instructions for GHASH a
247: Compile for Older Platform Versions
248: Make G1 the Default Garbage Collector
249: OCSP Stapling for TLS
250: Store Interned Strings in CDS Archives
251: Multi-Resolution Images
252: Use CLDR Locale Data by Default
253: Prepare JavaFX UI Controls & CSS APIs for Modularization
254: Compact Strings
255: Merge Selected Xerces 2.11.0 Updates into JAXP
256: BeanInfo Annotations
257: Update JavaFX/Media to Newer Version of GStreamer
258: HarfBuzz Font-Layout Engine
259: Stack-Walking API
260: Encapsulate Most Internal APIs
261: Module System
262: TIFF Image I/O
263: HiDPI Graphics on Windows and Linux
265: Marlin Graphics Renderer
266: More Concurrency Updates
269: Convenience Factory Methods for Collections
270: Reserved Stack Areas for Critical Sections
271: Unified GC Logging
272: Platform-Specific Desktop Features
273: DRBG-Based SecureRandom Implementations
274: Enhanced Method Handles
275: Modular Java Application Packaging
276: Dynamic Linking of Language-Defined Object Models
277: Enhanced Deprecation
278: Additional Tests for Humongous Objects in G1
279: Improve Test-Failure Troubleshooting
280: Indify String Concatenation
281: HotSpot C++ Unit-Test Framework
282: jlink: The Java Linker
283: Enable GTK 3 on Linux
285: Spin-Wait Hints
288: Disable SHA-1 Certificates
289: Deprecate the Applet API
290: Filter Incoming Serialization Data
292: Implement Selected ECMAScript 6 Features in Nashorn

AZUL SYSTEMS

Reactive Streams: goals

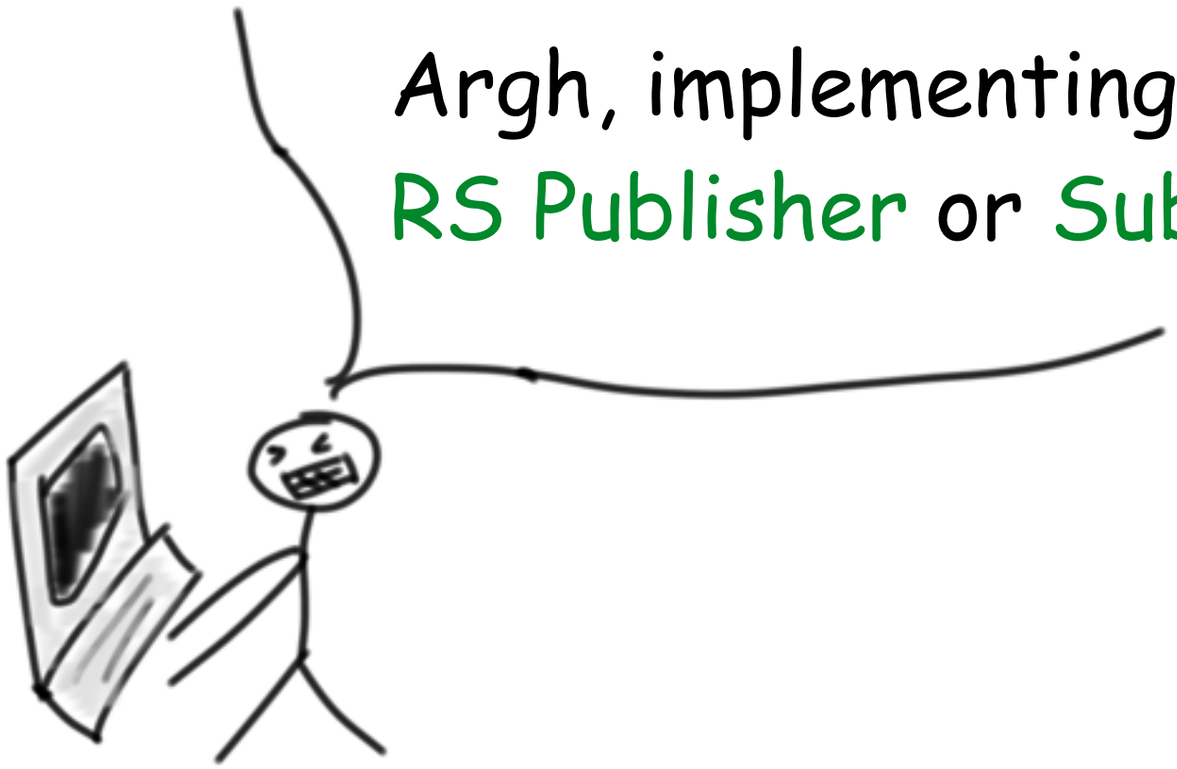
1) **Avoiding unbounded buffering** across **async** boundaries

2) **Inter-op interfaces** between various libraries

Reactive Streams: goals

1) Avoiding unbounded buffering across async boundaries

2) Inter-op interfaces between various libraries



Reactive Streams: goals

1) Avoiding unbounded buffering boundaries

2) Interoperability libraries

ID	Rule
1	The total number of `onNext` signals sent by a `Publisher` to a `Subscriber` MUST be less than or equal to the total number of elements requested by that `Subscriber`'s `Subscription` at all times.
2	A `Publisher` MAY signal less `onNext` than requested and terminate the `Subscription` by calling `onComplete` or `onError`.
3	`onSubscribe`, `onNext`, `onError` and `onComplete` signaled to a `Subscriber` MUST be signaled sequentially (no concurrent notifications).
4	If a `Publisher` fails it MUST signal an `onError`.
5	If a `Publisher` terminates successfully (finite stream) it MUST signal an `onComplete`.
6	If a `Publisher` signals either `onError` or `onComplete` on a `Subscriber`, that `Subscriber`'s `Subscription` MUST be considered cancelled.
7	Once a terminal state has been signaled (`onError`, `onComplete`) it is REQUIRED that no further signals occur.
8	If a `Subscription` is cancelled its `Subscriber` MUST eventually stop being signaled.
9	`Publisher.subscribe` MUST call `onSubscribe` on the provided `Subscriber` prior to any other signals to that `Subscriber` and MUST return normally, except when the provided `Subscriber` is `null` in which case it MUST throw a `java.lang.NullPointerException` to the caller, for all other situations [1] the only legal way to signal failure (or reject the `Subscriber`) is by calling `onError` (after calling `onSubscribe`).
10	`Publisher.subscribe` MAY be called as many times as wanted but MUST be with a different `Subscriber` each time [see 2.12].
11	A `Publisher` MAY support multiple `Subscriber`s and decides whether each `Subscription` is unicast or multicast.

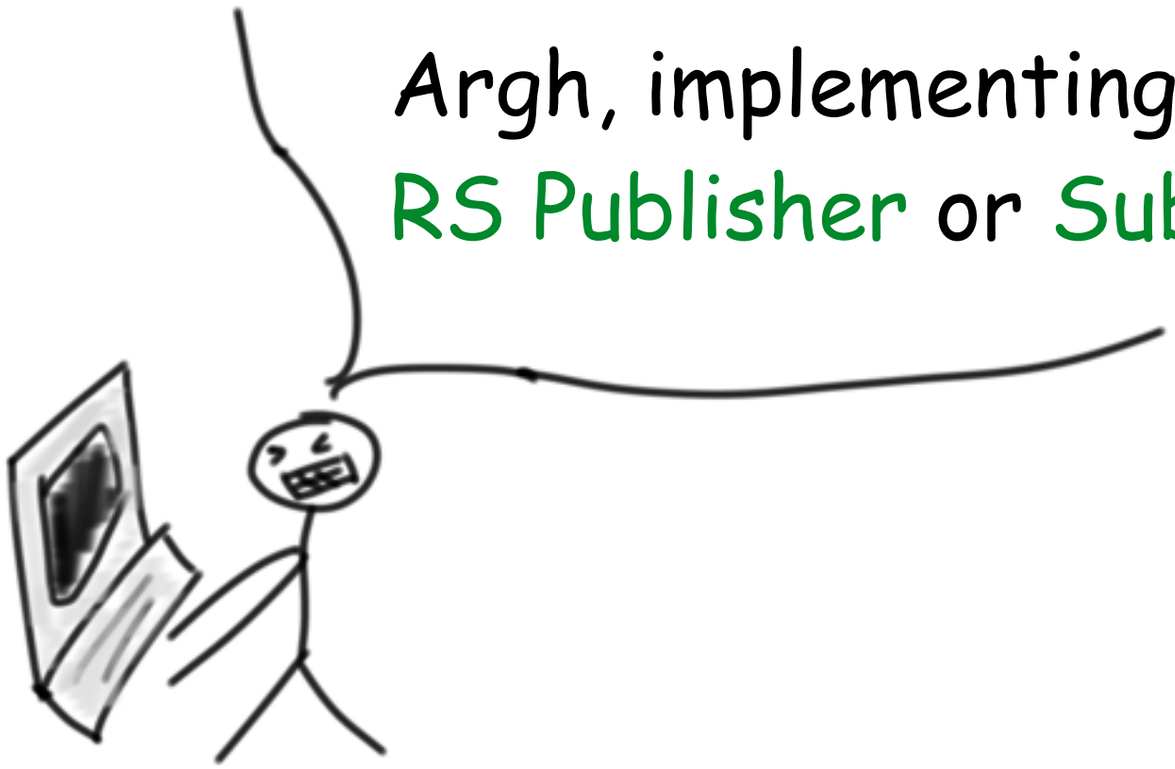
[1] : A stateful Publisher can be overwhelmed, bounded by a finite number of underlying resources, exhausted, shut-down or in a failed state.

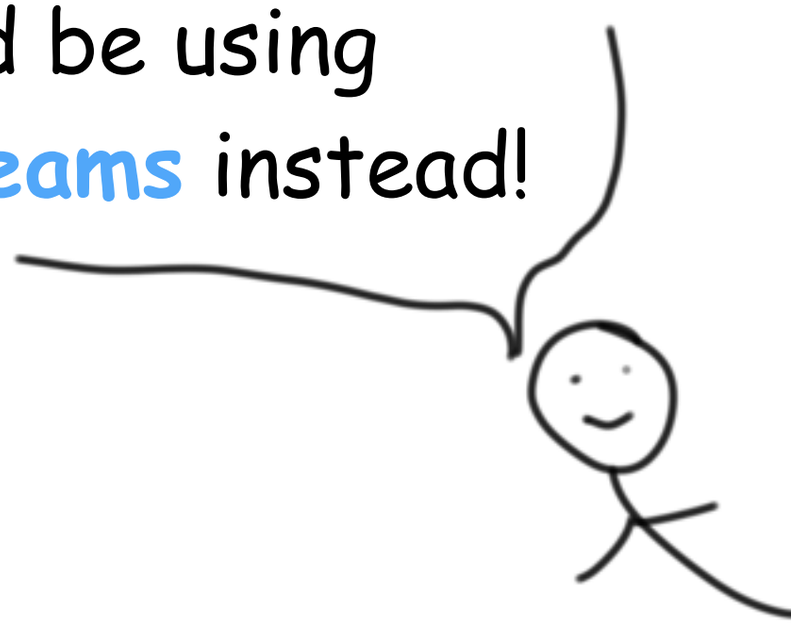
ID	Rule
1	A `Subscriber` MUST signal demand via `Subscription.request(long n)` to receive `onNext` signals.
2	If a `Subscriber` suspects that its processing of signals will negatively impact its `Publisher`'s responsivity, it is RECOMMENDED that it asynchronously dispatches its signals.
3	`Subscriber.onComplete()` and `Subscriber.onError(Throwable t)` MUST NOT call any methods on the `Subscription` or the `Publisher`.
4	`Subscriber.onComplete()` and `Subscriber.onError(Throwable t)` MUST consider the Subscription cancelled after having received the signal.
5	A `Subscriber` MUST call `Subscription.cancel()` on the given `Subscription` after an `onSubscribe` signal if it already has an active `Subscription`.
6	A `Subscriber` MUST call `Subscription.cancel()` if it is no longer valid to the `Publisher` without the `Publisher` having signaled `onError` or `onComplete`.
7	A `Subscriber` MUST ensure that all calls on its `Subscription` take place from the same thread or provide for respective external synchronization.
8	A `Subscriber` MUST be prepared to receive one or more `onNext` signals after having called `Subscription.cancel()` if there are still requested elements pending [see 3.12]. `Subscription.cancel()` does not guarantee to perform the underlying cleaning operations immediately.
9	A `Subscriber` MUST be prepared to receive an `onComplete` signal with or without a preceding `Subscription.request(long n)` call.
10	A `Subscriber` MUST be prepared to receive an `onError` signal with or without a preceding `Subscription.request(long n)` call.
11	A `Subscriber` MUST make sure that all calls on its `onXXX` methods happen-before [1] the processing of the respective signals. I.e. the Subscriber must take care of properly publishing the signal to its processing logic.
12	`Subscriber.onSubscribe` MUST be called at most once for a given `Subscriber` (based on object equality).
13	Calling `onSubscribe`, `onNext`, `onError` or `onComplete` MUST return normally except when any provided parameter is `null` in which case it MUST throw a `java.lang.NullPointerException` to the caller, for all other situations the only legal way for a `Subscriber` to signal failure is by cancelling its `Subscription`. In the case that this rule is violated, any associated `Subscription` to the `Subscriber` MUST be considered as cancelled, and the caller MUST raise this error condition in a fashion that is adequate for the runtime environment.

ID	Rule
1	`Subscription.request` and `Subscription.cancel` MUST only be called inside of its `Subscriber` context. A `Subscription` represents the unique relationship between a `Subscriber` and a `Publisher` [see 2.12].
2	The `Subscription` MUST allow the `Subscriber` to call `Subscription.request` synchronously from within `onNext` or `onSubscribe`.
3	`Subscription.request` MUST place an upper bound on possible synchronous recursion between `Publisher` and `Subscriber` [1].
4	`Subscription.request` SHOULD respect the responsivity of its caller by returning in a timely manner[2].
5	`Subscription.cancel` MUST respect the responsivity of its caller by returning in a timely manner[2], MUST be idempotent and MUST be thread-safe.
6	After the `Subscription` is cancelled, additional `Subscription.request(long n)` MUST be NOPs.
7	After the `Subscription` is cancelled, additional `Subscription.cancel()` MUST be NOPs.
8	While the `Subscription` is not cancelled, `Subscription.request(long n)` MUST register the given number of additional elements to be produced to the respective subscriber.
9	While the `Subscription` is not cancelled, `Subscription.request(long n)` MUST signal `onError` with a `java.lang.IllegalArgumentException` if the argument is <= 0. The cause message MUST include a reference to this rule and/or quote the full rule.
10	While the `Subscription` is not cancelled, `Subscription.request(long n)` MAY synchronously call `onNext` on this (or other) subscriber(s).
11	While the `Subscription` is not cancelled, `Subscription.request(long n)` MAY synchronously call `onComplete` or `onError` on this (or other) subscriber(s).
	While the `Subscription` is not cancelled,

Reactive Streams: goals

1) Avoiding unbounded buffering across async boundaries
2) Inter-op interfaces between various libraries





Reactive Streams

Already made a huge industry impact

Back-pressure as a feature

Spark / SPARK-7398

Add back-pressure to Spark Streaming (umbrella JIRA)

Agile Board

Sub-Tasks

1.	Implement a mechanism to send a new rate from the driver to the block generator		RESOLVED	Iulian Dragos
2.	Define the RateEstimator interface, and implement the ReceiverRateController		RESOLVED	Iulian Dragos
3.	Implement a PIDRateEstimator		RESOLVED	Iulian Dragos
4.	Implement the DirectKafkaRateController		RESOLVED	Iulian Dragos
5.	Make all BlockGenerators subscribe to rate limit updates		RESOLVED	Tathagata Das
6.	Handle a couple of corner cases in the PID rate estimator		RESOLVED	Tathagata Das
7.	BlockGenerator lock structure can cause lock starvation of the block updating thread		RESOLVED	Tathagata Das
8.	Rename the SparkConf property to spark.streaming.backpressure.{enable --> enabled}		RESOLVED	Tathagata Das
9.	Provide pluggable Congestion Strategies to deal with Streaming load		IN PROGRESS	*Unassigned*

Inspiring other technologies

elixir-lang / elixir-lang.github.com

Watch 18 | Star 129 | Fork

Code | Issues 3 | Pull requests 2 | Projects 0 | Pulse | Graphs

Give credit where credit is due

master

Browse fil

josevalim committed on GitHub on Aug 5 — 1 parent 62f6171 — commit 39c0eae818c9289d0e9f31902c5aa6cec445

Showing 1 changed file with 2 additions and 0 deletions.

Unified

2 _posts/2016-07-14-announcing-genstage.markdown

Vie

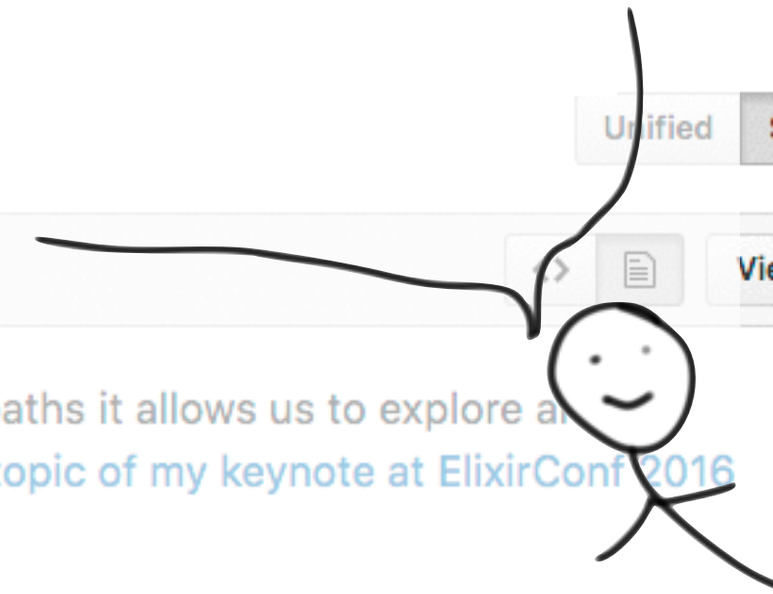
We are very excited with the possibilities GenStage brings to developers and all new paths it allows us to explore and research. So give it a try and let us know! GenStage, Flows, and more will also be the topic of my keynote at ElixirConf 2016 and we hope to see you there.

> Finally, we want to thank the akka-streams and reactive-streams projects which provided us guidance in implementing the demand-driven exchange between stages as well as the Apache Spark and Apache Beam initiatives that inspire the work behind `GenStage.Flow`.

Happy coding!

Inspiring other technologies

It's been a while since Java inspired other modern technologies, hasn't it?



Showing **1 changed file** with **2 additions** and **0 deletions**.

Unified

2 _posts/2016-07-14-announcing-genstage.markdown

We are very excited with the possibilities GenStage brings to developers and all new paths it allows us to explore a research. So give it a try and let us know! GenStage, Flows, and more will also be the topic of my keynote at ElixirConf 2016 and we hope to see you there.

> Finally, we want to thank the akka-streams and reactive-streams projects which provided us guidance in implementing the demand-driven exchange between stages as well as the Apache Spark and Apache Beam initiatives that inspire the work behind `GenStage.Flow`.

Happy coding!

Akka Streams

The **implementation**.

Complete and awesome Java and Scala APIs.
As *everything* since day 1 in Akka.

Akka Streams in 20 seconds:

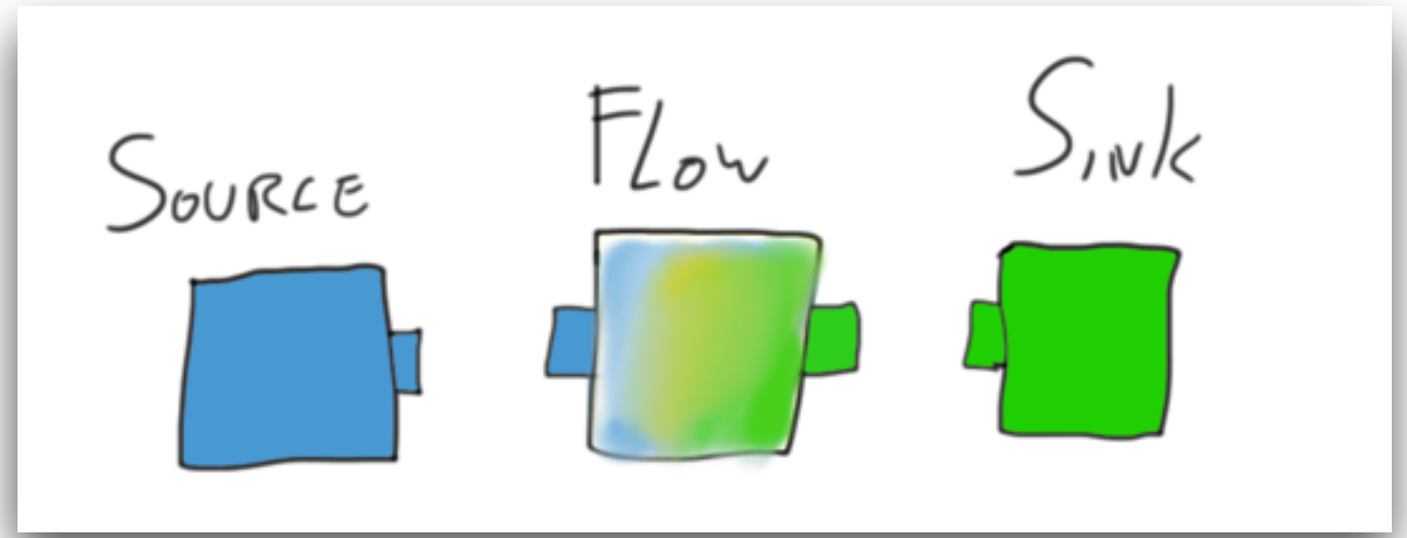
```
// types:
Source<Out, Mat>
Flow<In, Out, Mat>
Sink<In, Mat>

// generally speaking, it's always:
val ready =
  Source.from…(???).via(flow).map(i -> i * 2).to(sink)

val mat: Mat = ready.run()

// the usual example:
val f: Future<String> =
  Source.single(1).map(i -> i.toString).runWith(Sink.head)
```

Proper static typing!

Akka Streams in 20 seconds:

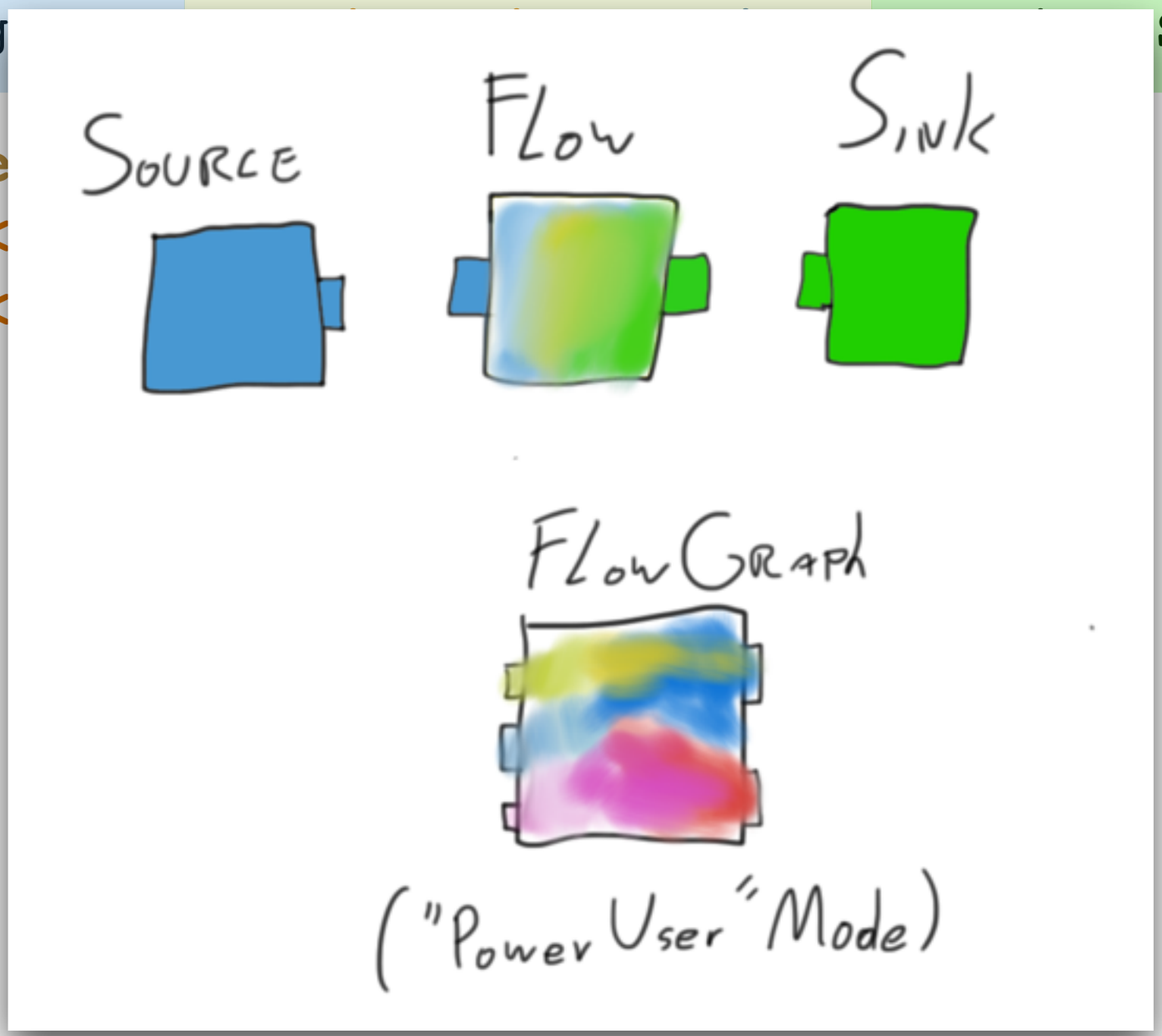
```
Source.single(1).map(i -> i.toString).runWith(Sink.head())
```

```
// types:
Source<Int, NotUsed>
  Flow<Int, String, NotUsed>
       Sink<String, Future<String>>
```



Akka Streams in 20 seconds:

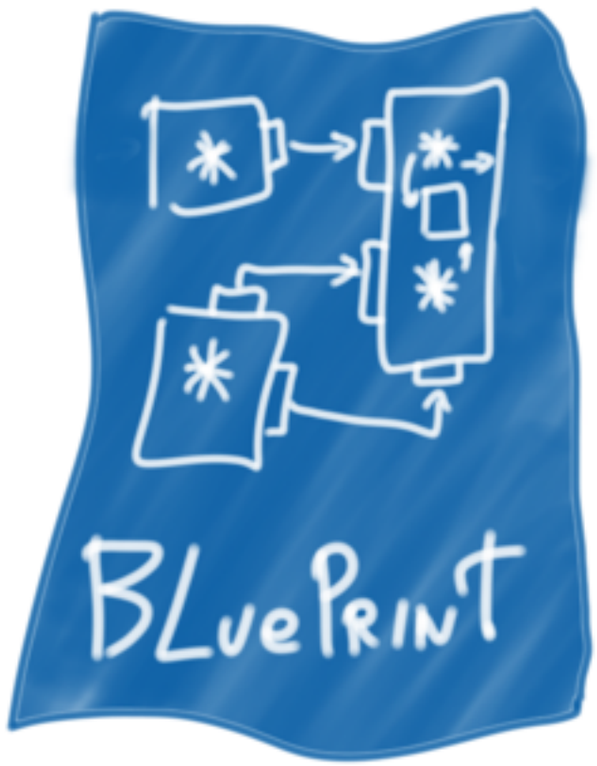
```
Source.sing                                    Sink.head())

    // type
    Source<
      Flow<
```



Materialization

Gears from GeeCON.org, did I mention it's an awesome conf?

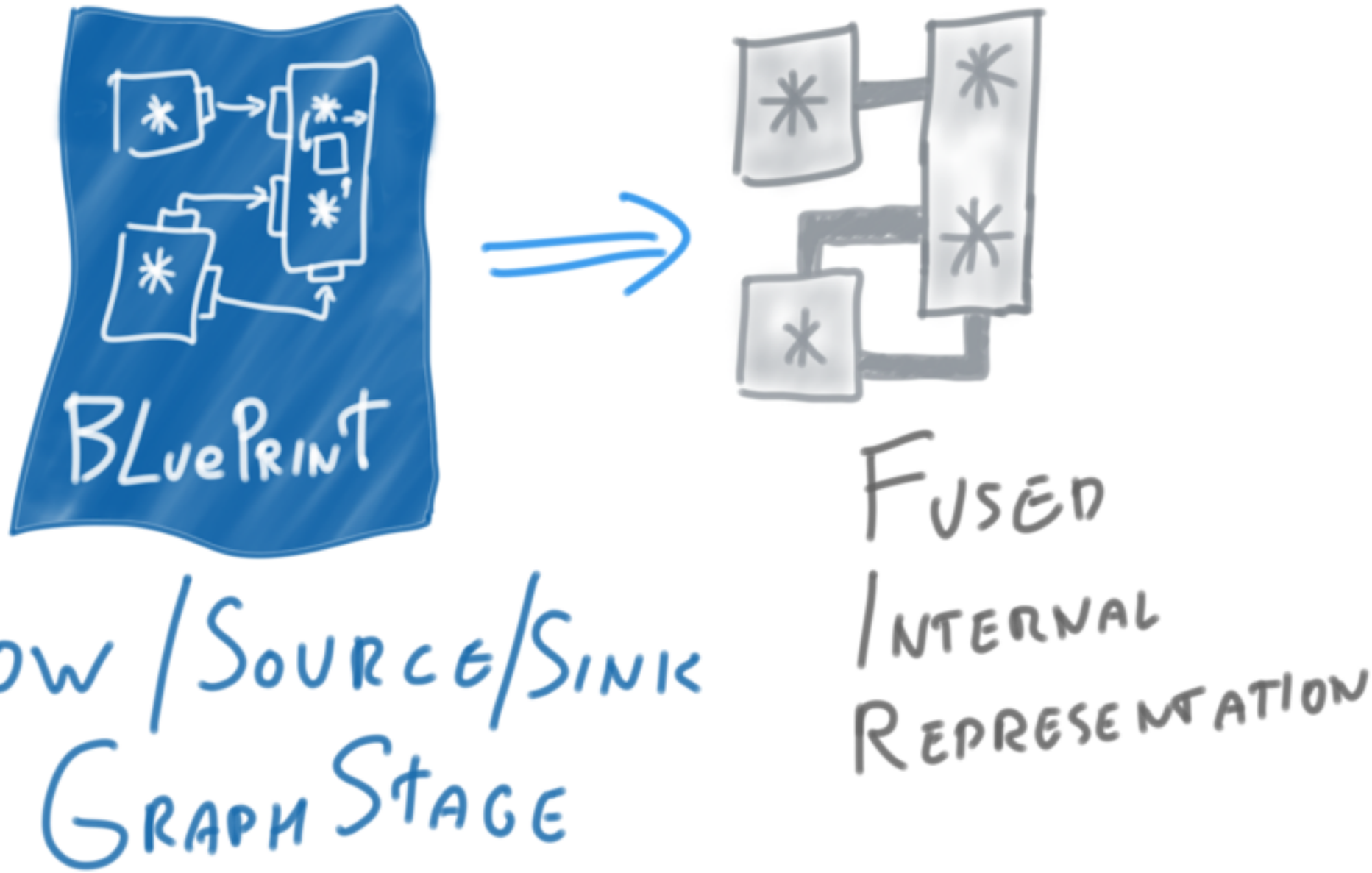
What is “materialization” really?



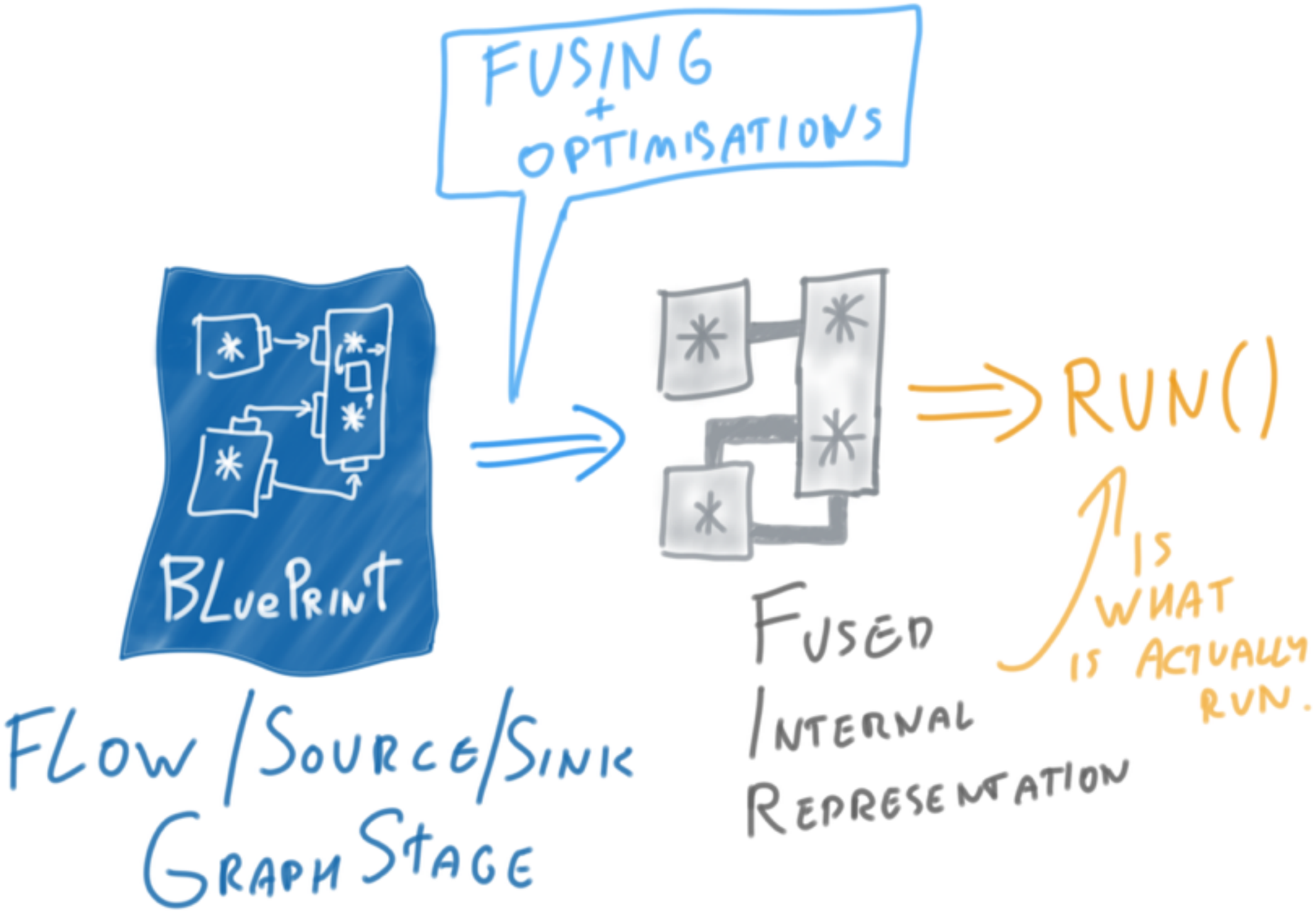
FLOW / SOURCE/SINK

GRAPH STAGE

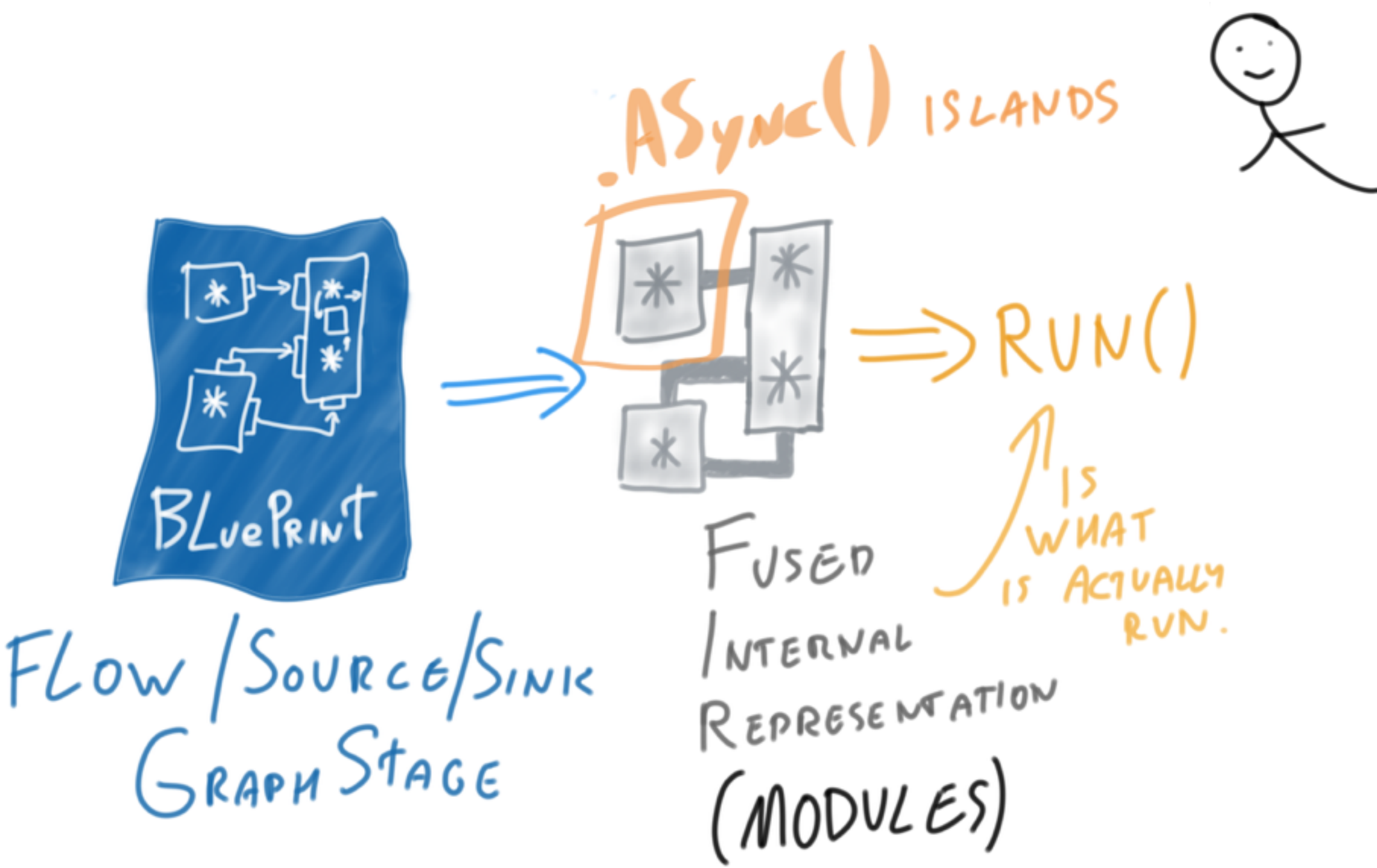
What is “materialization” really?



What is “materialization” really?



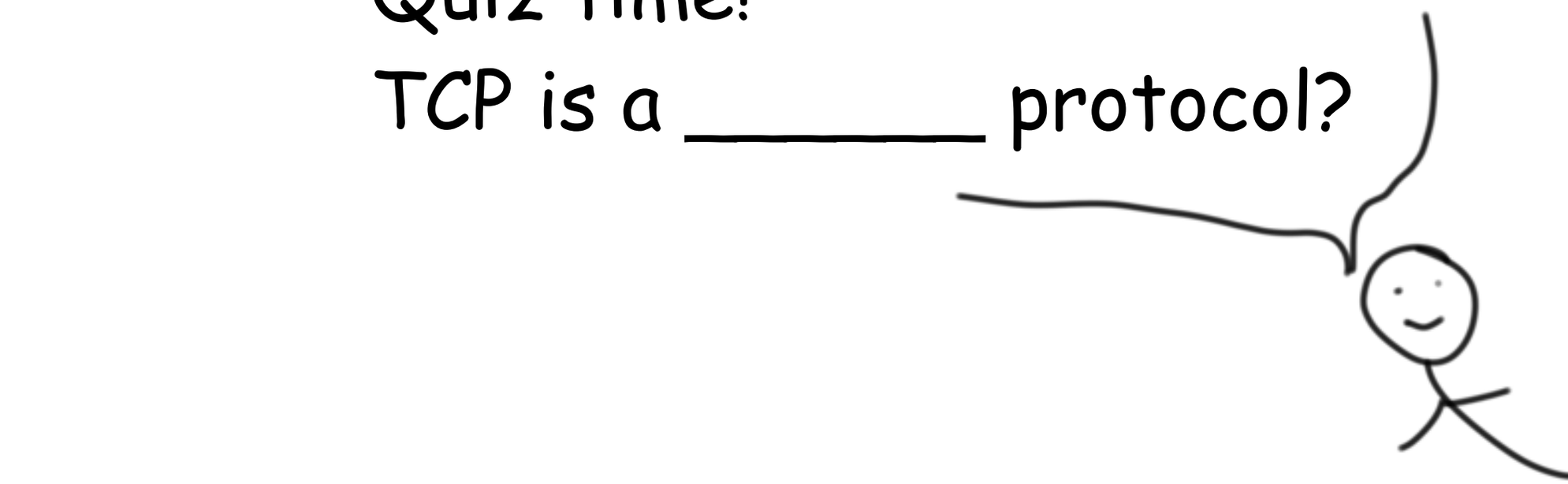
What is “materialization” really?




akka streams
& HTTP

Akka Streams & HTTP

A **core feature** not obvious to the **untrained eye**…!



Akka Streams & HTTP

A **core feature** not obvious to the **untrained eye**…!

Streaming in Akka HTTP

`HttpServer` as a:

`Flow[HttpRequest, HttpResponse]`

http://doc.akka.io/docs/akka/2.4/scala/stream/stream-customize.html#graphstage-scala

"Framed entity streaming" https://github.com/akka/akka/pull/20778

Streaming in Akka HTTP

`HttpServer` as a:

`Flow[HttpRequest, HttpResponse]`

`HTTP Entity` as a:

`Source[ByteString, _]`

http://doc.akka.io/docs/akka/2.4/scala/stream/stream-customize.html#graphstage-scala

“Framed entity streaming” https://github.com/akka/akka/pull/20778

Streaming in Akka HTTP

`HttpServer` as a:

`Flow[HttpRequest, HttpResponse]`

`HTTP Entity` as a:

`Source[ByteString, _]`

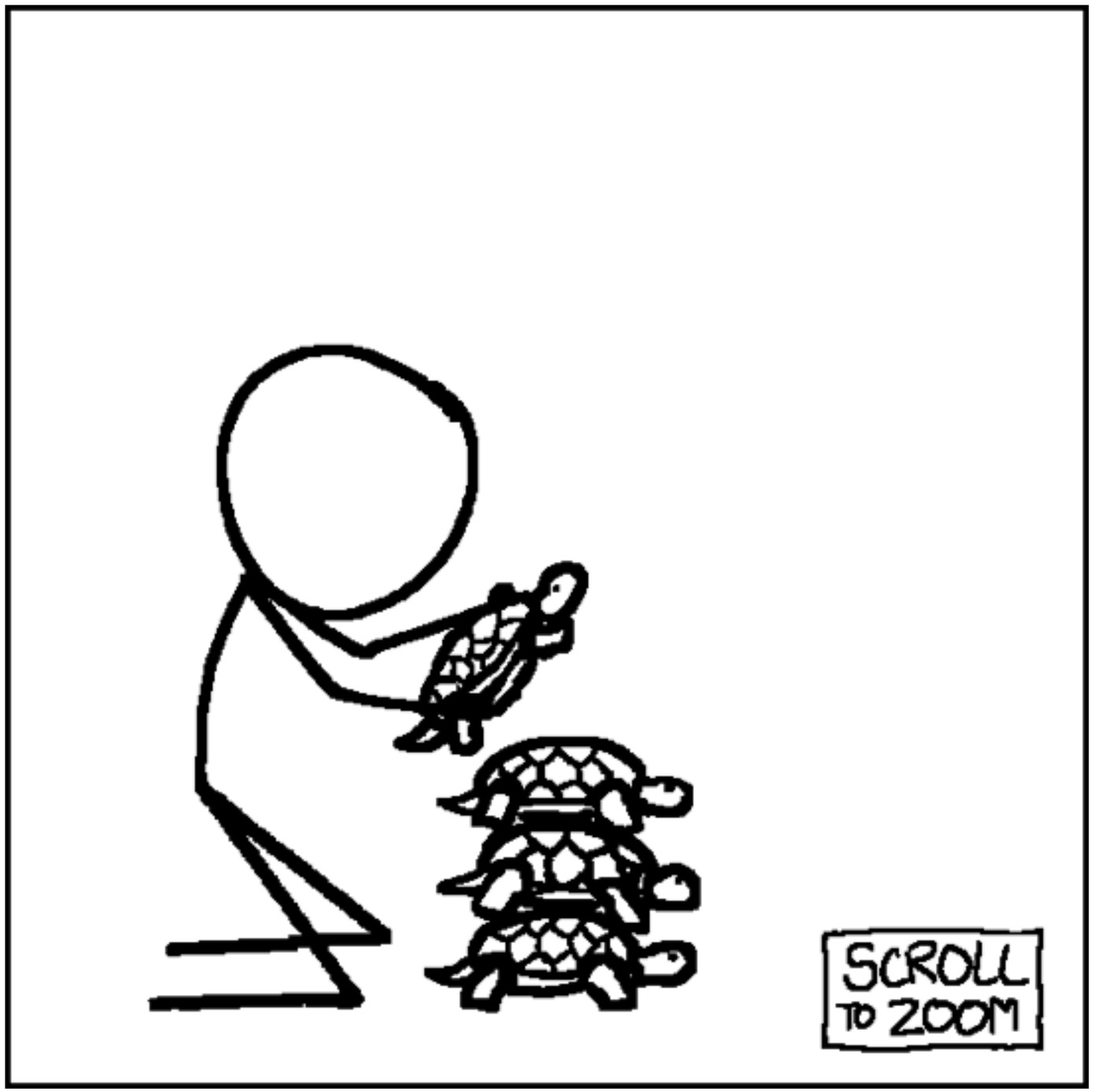
`Websocket connection` as a:

`Flow[ws.Message, ws.Message]`

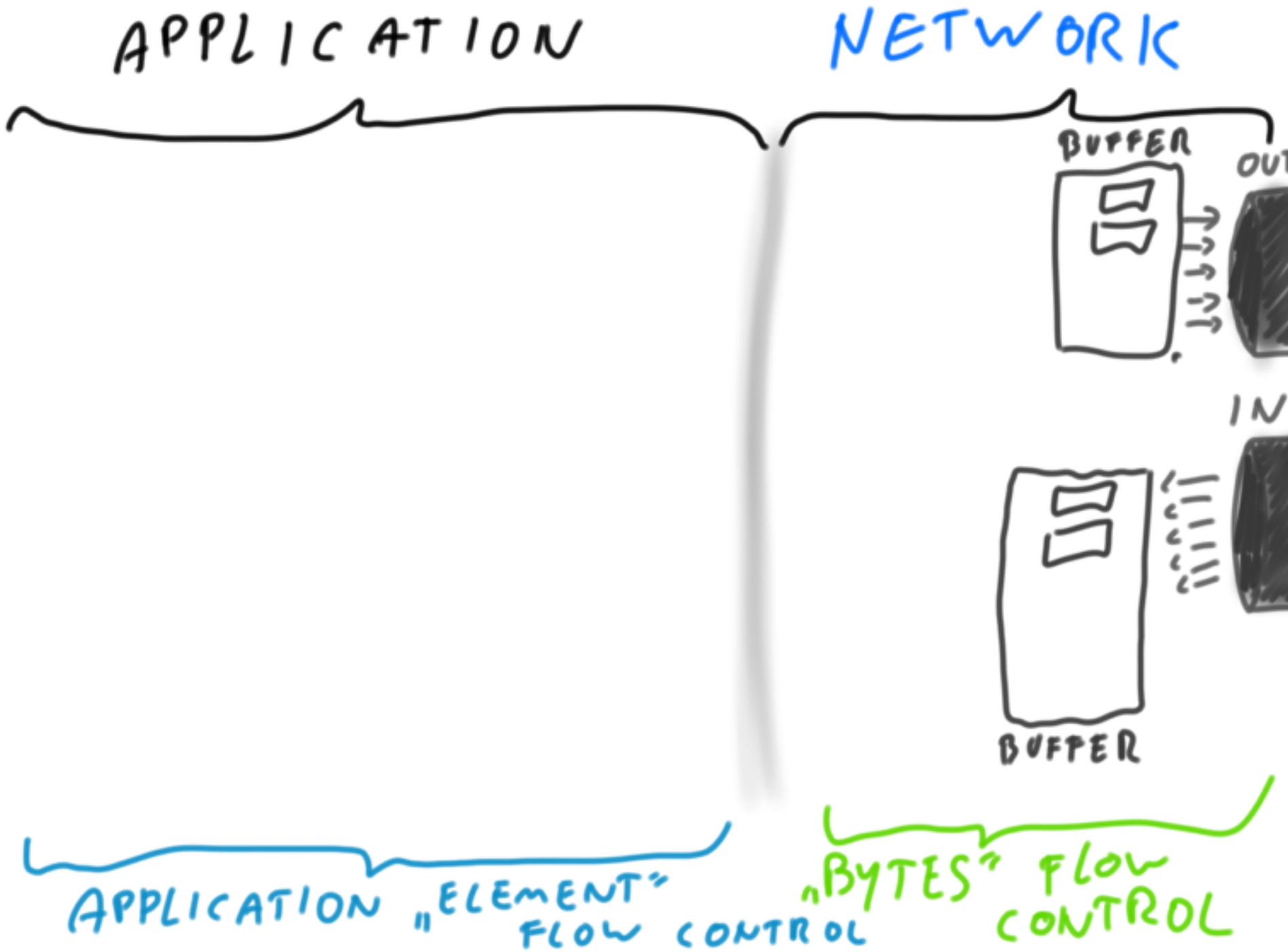
http://doc.akka.io/docs/akka/2.4/scala/stream/stream-customize.html#graphstage-scala

“Framed entity streaming” https://github.com/akka/akka/pull/20778

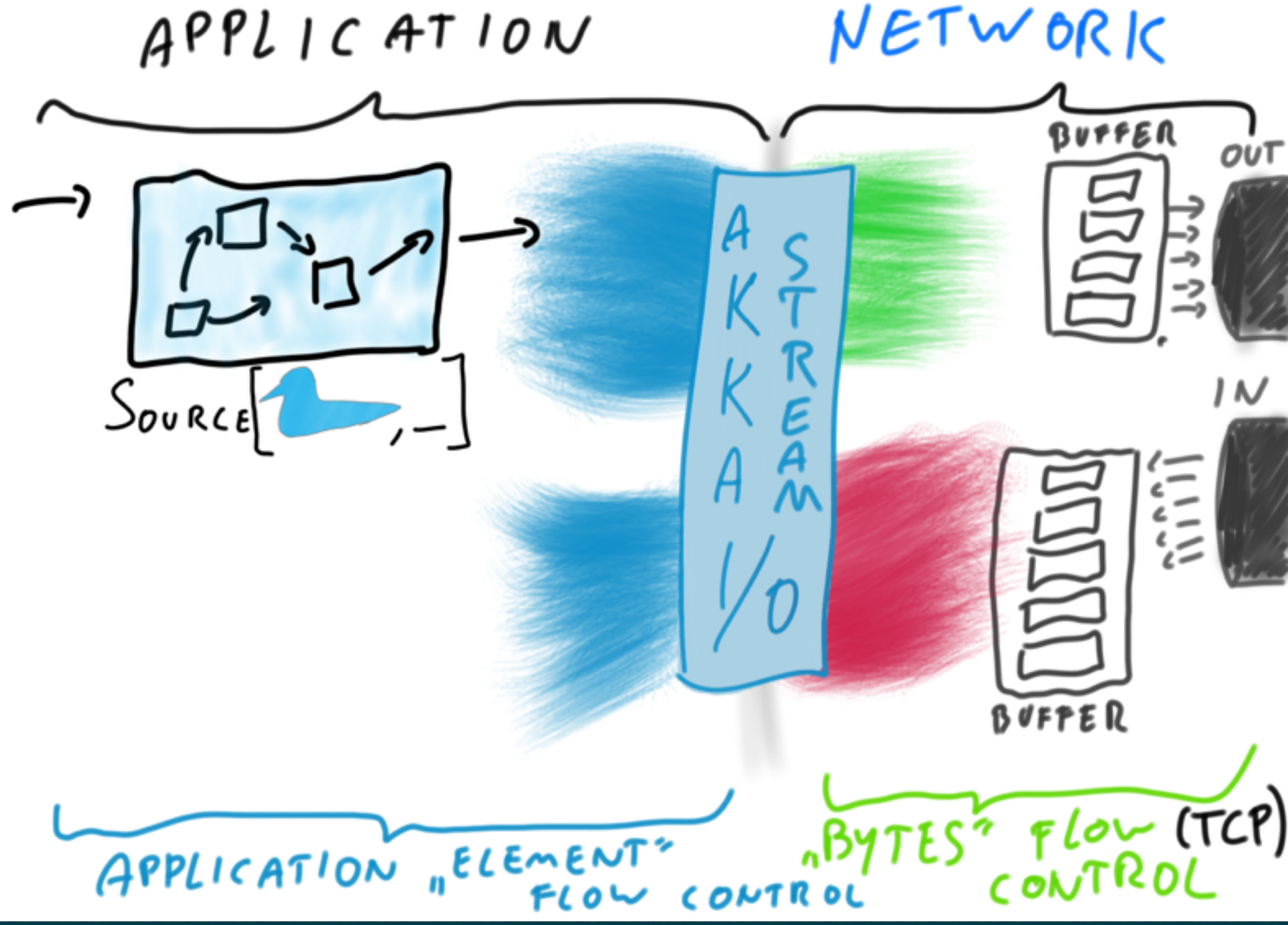
It’s ~~turtles~~ buffers all the way down!



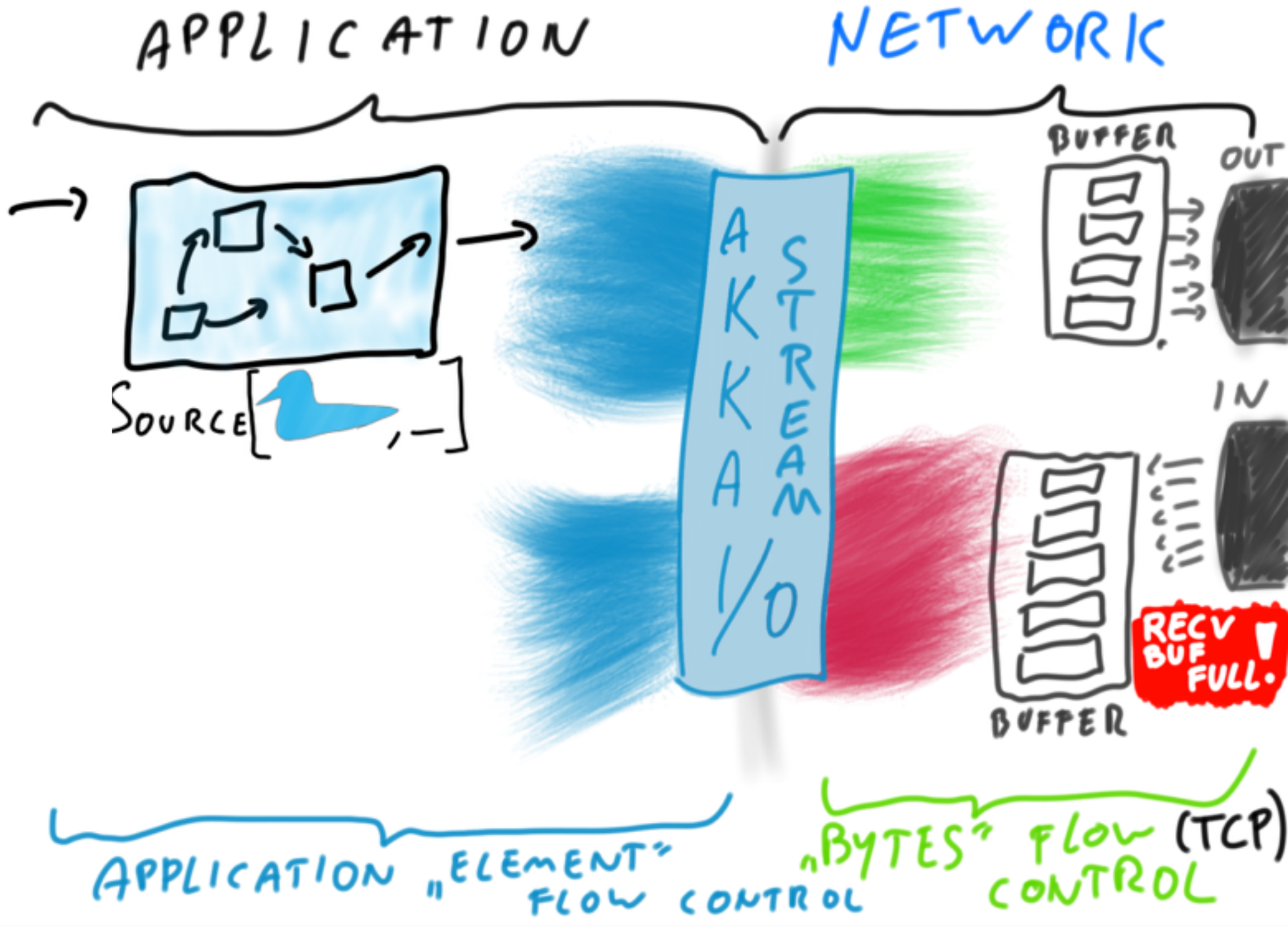
Streaming from Akka HTTP



Streaming from Akka HTTP



Streaming from Akka HTTP



Streaming from Akka HTTP (Java)

```
public static void main(String[] args) {
  final ActorSystem system = ActorSystem.create();
  final Materializer materializer = ActorMaterializer.create(system);
  final Http http = Http.get(system);


  final Source<Tweet, NotUsed> tweets = Source.repeat(new Tweet("Hello world"));

  final Route tweetsRoute =
    path("tweets", () ->
      completeWithSource(tweets, Jackson.marshaller(), EntityStreamingSupport.json())
    );

  final Flow<HttpRequest, HttpResponse, NotUsed> handler =
    tweetsRoute.flow(system, materializer);

  http.bindAndHandle(handler,
    ConnectHttp.toHost("localhost", 8080),
    materializer
  );
  System.out.println("Running at http://localhost:8080");

}
```

Streaming from Akka HTTP (Java)

```java
public static void main(String[] args) {
  final ActorSystem system = ActorSystem.create();
  final Materializer materializer = ActorMaterializer.create(system);
  final Http http = Http.get(system);


  final Source<Tweet, NotUsed> tweets = Source.repeat(new Tweet("Hello world"));

  final Route tweetsRoute =
    path("tweets", () ->
      completeWithSource(tweets, Jackson.marshaller(), EntityStreamingSupport.json())
    );

  final Flow<HttpRequest, HttpResponse, NotUsed> handler =
    tweetsRoute.flow(system, materializer);

  http.bindAndHandle(handler,
    ConnectHttp.toHost("localhost", 8080),
    materializer
  );
  System.out.println("Running at http://localhost:8080");

}
```

Streaming from Akka HTTP (Scala)

```scala
object Example extends App
  with SprayJsonSupport with DefaultJsonProtocol {
  import akka.http.scaladsl.server.Directives._

  implicit val system = ActorSystem()
  implicit val mat = ActorMaterializer()

  implicit val jsonRenderingMode = EntityStreamingSupport.json()
  implicit val TweetFormat = jsonFormat1(Tweet)

  def tweetsStreamRoutes =
    path("tweets") {
      complete {
       Source.repeat(Tweet(""))
      }
    }

  Http().bindAndHandle(tweetsStreamRoutes, "127.0.0.1", 8080)
  System.out.println("Running at http://localhost:8080");
}
```

Ecosystem that solves problems

> (is greater than)

solving all the problems ourselves


Codename:
Alpakka

Alpakka – a community for Stream connectors

Alpakka

A community for Streams connectors

Alpakka – a community for Stream connectors

Threading & Concurrency in Akka Streams Explained (part I)

Mastering GraphStages (part I, Introduction)

Akka Streams Integration, codename Alpakka

A gentle introduction to building Sinks and Sources using GraphStage APIs (Mastering GraphStages, Part II)

Writing Akka Streams Connectors for existing APIs

Flow control at the boundary of Akka Streams and a data provider

Akka Streams Kafka 0.11

Alpakka – a community for Stream connectors

Existing examples:

MQTT

AMQP

Streaming HTTP

Streaming TCP

Streaming FileIO

Cassandra Queries

“Reactive Kafka” (akka-stream-kafka)

S3, SQS & other Amazon APIs

Streaming JSON

Streaming XML

Alpakka – a community for Stream connectors

Existing examples:

MQTT

AMQP

Streaming HTTP

Streaming TCP

Streaming FileIO

Cassandra Queries

“Reactive Kafka” (akka-stream-kafka)

S3, SQS & other Amazon APIs

Streaming JSON Parsing

Streaming XML Parsing

Is now the time to adopt?

Reactive Streams / Akka Streams


Totally, go for it.

Taking it to the next level:

ReactiveSocket.io

Taking it to the next level:

ReactiveSocket.io

A collaboration similar in spirit, and continuing from where Reactive Streams brought us today.

Reactive Streams over network boundaries

Reactive Streams = **async** boundaries

Reactive Socket = **RS** + **network** boundaries

http://reactivesocket.io/

Reactive Streams over network boundaries

Reactive Streams = **async** boundaries

Reactive Socket = **RS** + **network** boundaries

Primarily led by:

- Ben Christensen, Todd Montgomery (**Facebook**) & team
- Nitesh Kant (**Netflix**) & team

Lightbend on board as well – right now we're prototyping with it.

http://reactivesocket.io/

Reactive Streams over network boundaries

"ReactiveSocket is an application protocol providing Reactive Streams semantics over an asynchronous, binary boundary."

http://reactivesocket.io/

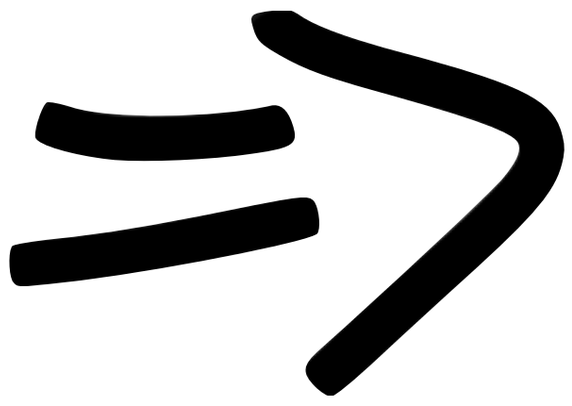
Reactive Streams over network boundaries

Binary ⇒

Support various platforms:

- java
- c++
- js
- ...

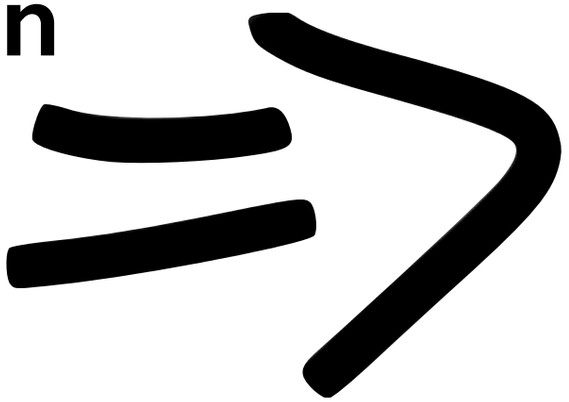
Reactive Streams over network boundaries

Async 

Obviously we want it to be async and properly bi-directional.

Reactive Streams over network boundaries

Application protocol



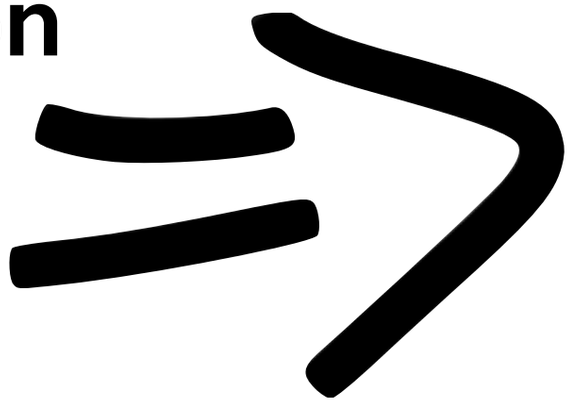
Again, bridging app-level semantics to wire semantics.

Reactive Streams semantics:

- “you can do 10 requests”

Reactive Streams over network boundaries

Application protocol



Again, bridging app semantics to wire semantics.

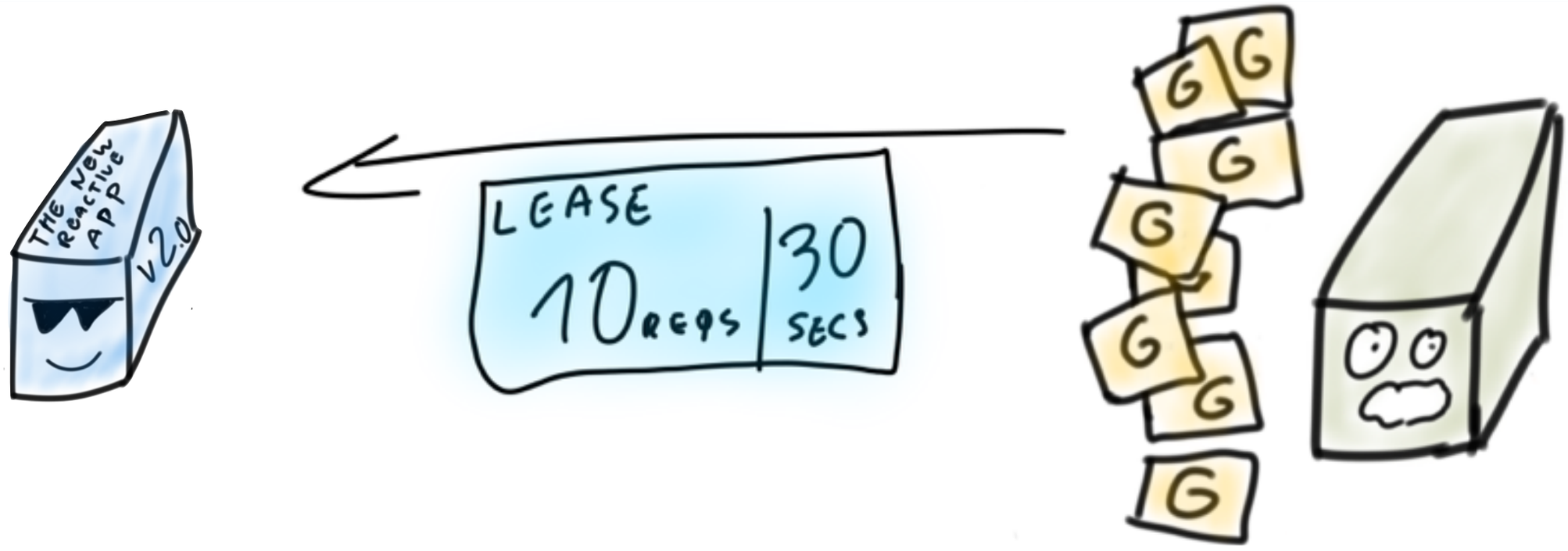
Reactive Streams semantics:

- “you can do 10 requests”

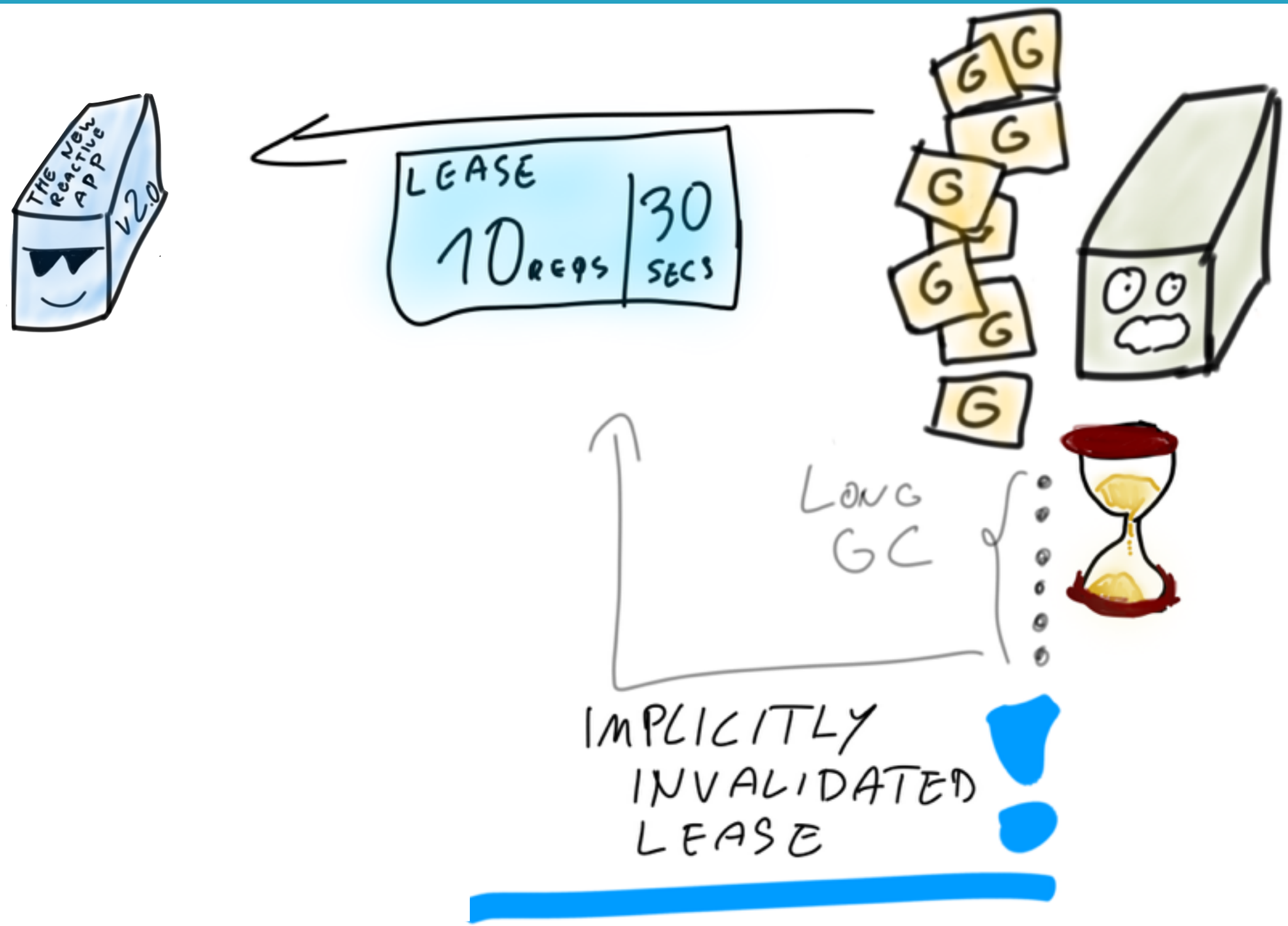
Extra Lease semantics:

- “you can do 10 reqs in 30secs”

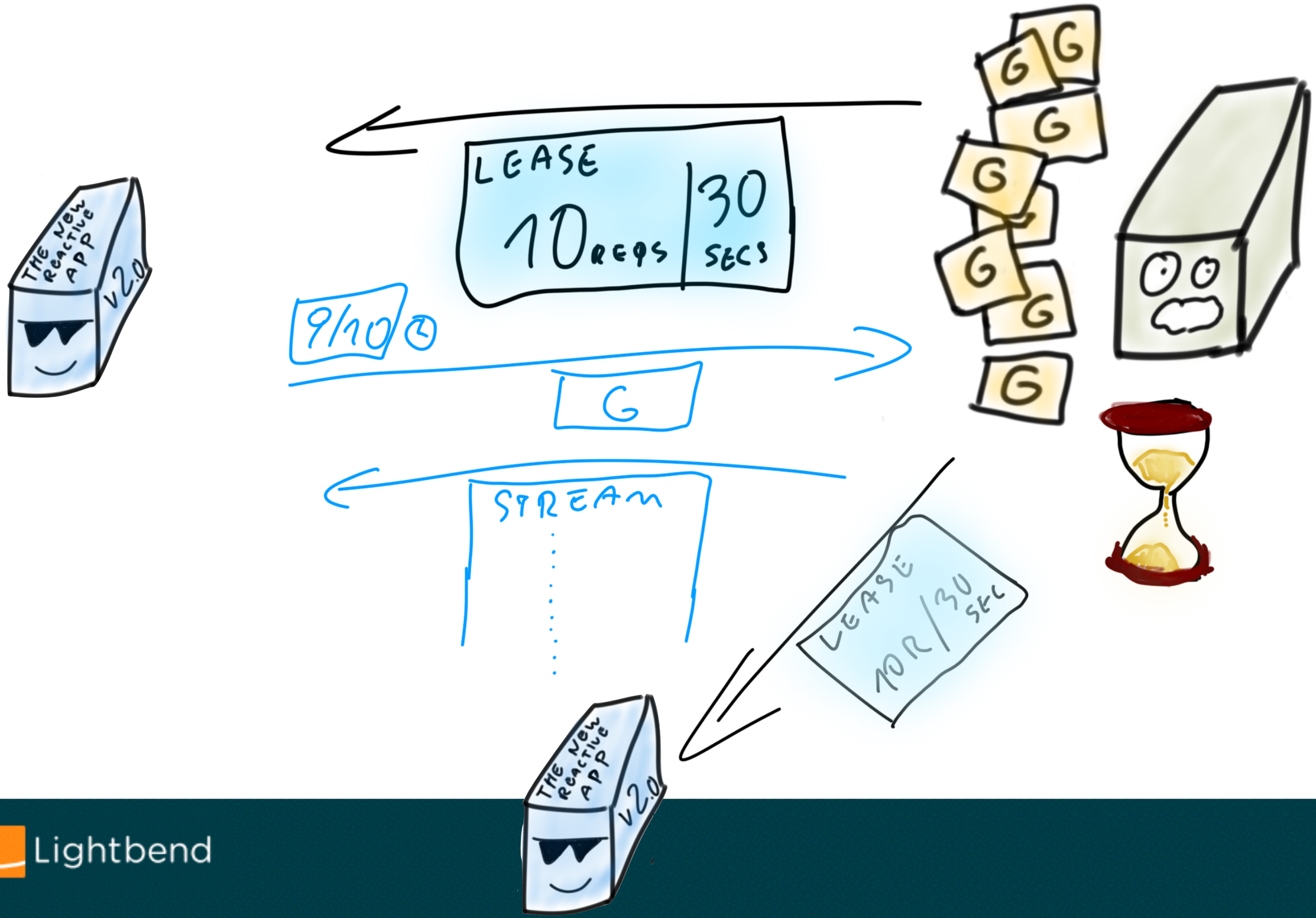
Lease semantics, “flipping the problem”



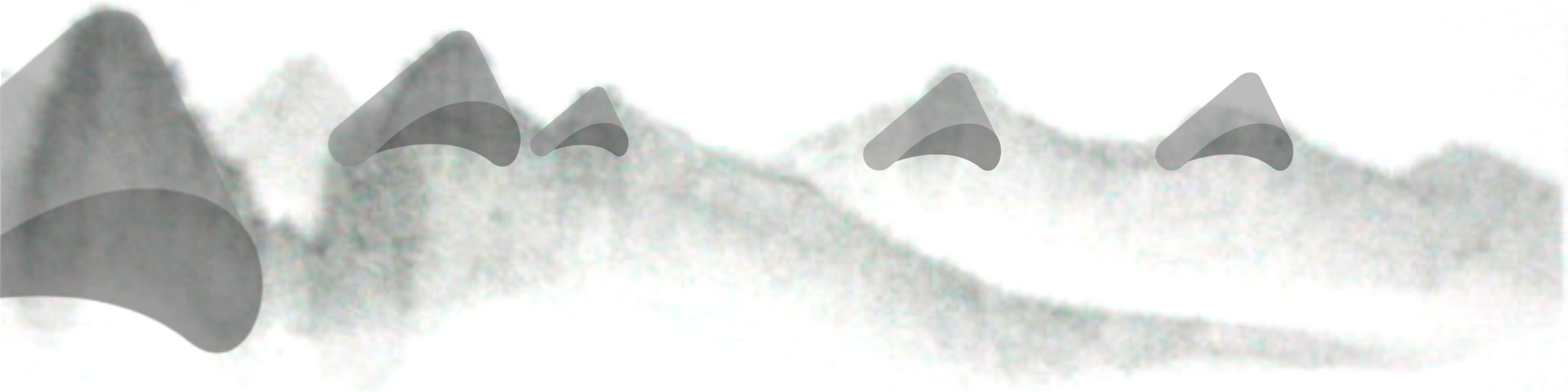
Lease semantics, “flipping the problem”



Lease semantics, “flipping the problem”



Exciting times ahead!



State and Future[_] of Reactive

Reactive Systems – well established "goal" architecture
…excellent building blocks available, and getting even better with:

Reactive-Streams eco-system blooming!
… as very important building block of the puzzle.

Akka Streams driving implementation of **Reactive Streams**
(first passing TCK, prime contributor to spec, strong ecosystem)

Reactive Socket continuing to improve app-level flow-control semantics. More control than "just use HTTP/2".
… considering resumability for streams as well.

The best is yet to come:
combining all these components into resilient, scalable systems!

Happy hAkking!

We <3 contributions

- **Easy to contribute:**
  - https://github.com/akka/akka/issues?q=is%3Aissue+is%3Aopen+label%3Aeasy-to-contribute
  - https://github.com/akka/akka/issues?q=is%3Aissue+is%3Aopen+label%3A%22nice-to-have+%28low-prio%29%22

- **Akka:** akka.io && github.com/akka
- **Reactive Streams:** reactive-streams.org
- **Reactive Socket:** reactivesocket.io

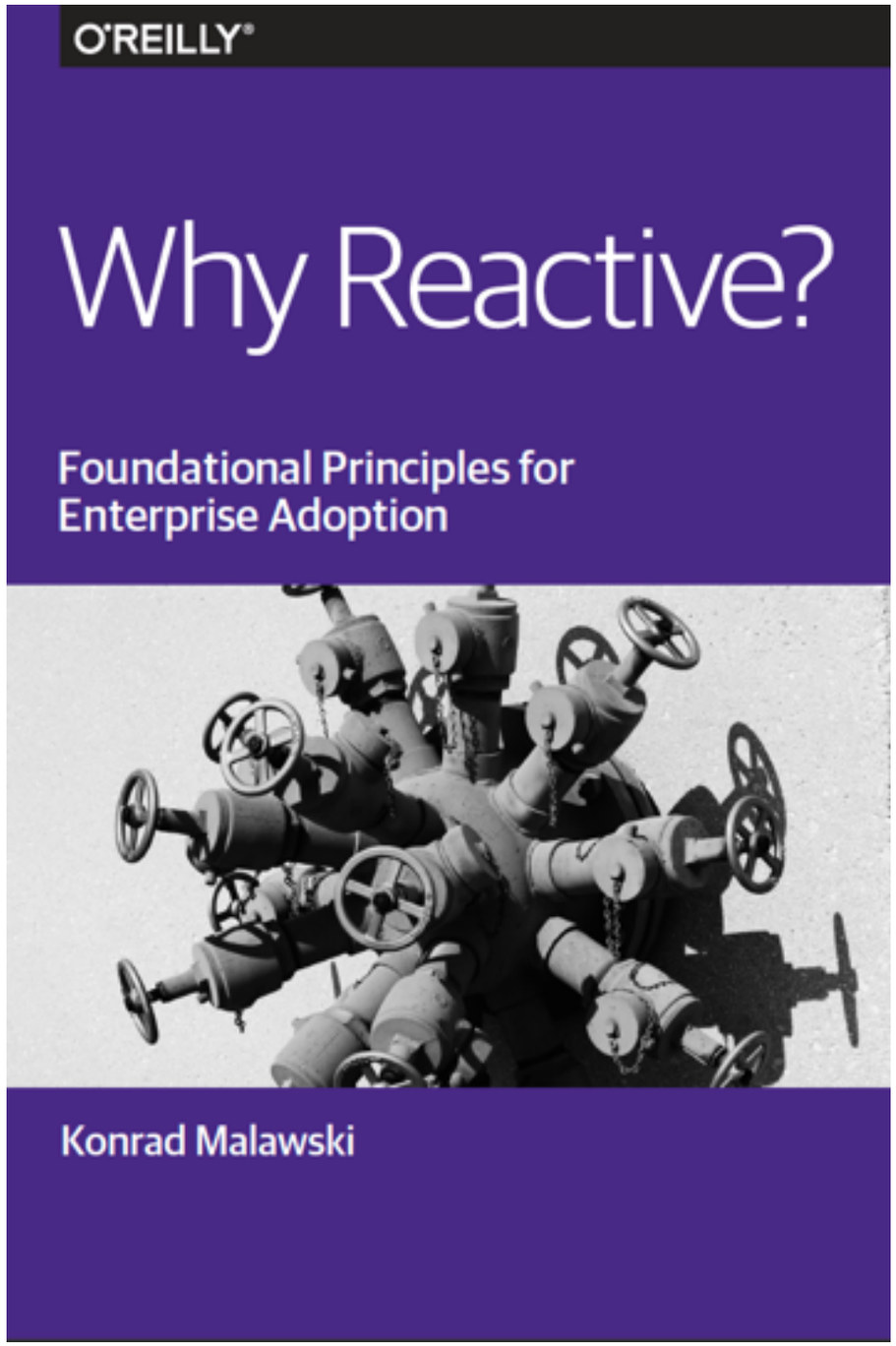
- **Mailing list:**
  - https://groups.google.com/group/akka-user

- **Public chat rooms:**
  - http://gitter.im/akka/dev developing Akka
  - http://gitter.im/akka/akka using Akka

Pics

Gundam pictures from: http://www.wallpaperup.com/tag/gundam/3

Obligatory “read my book!” slide :-)



Free e-book and printed report.

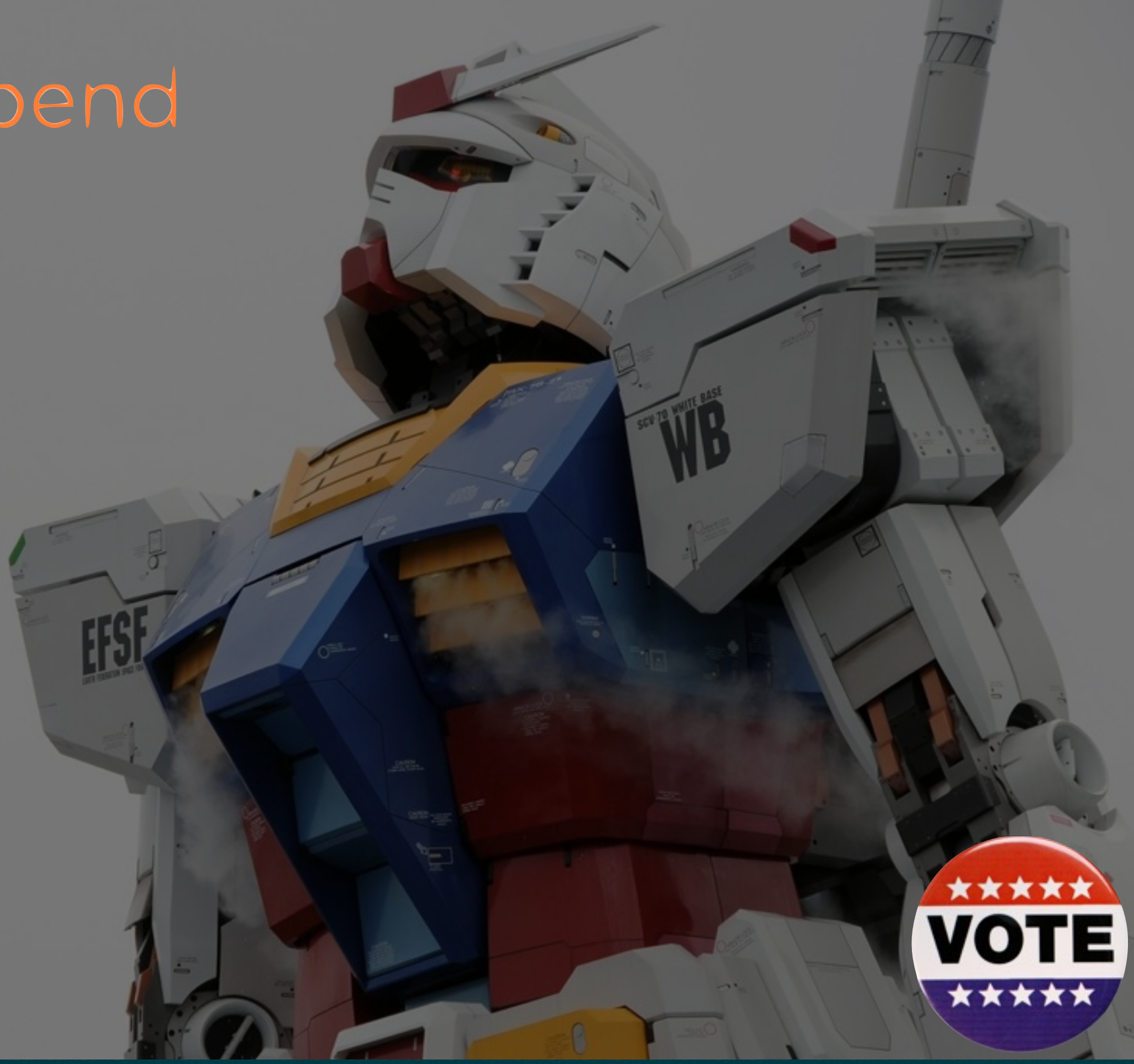
bit.ly/why-reactive

Covers what reactive actually is.
Implementing in existing architectures.

Thoughts from the team that’s building reactive apps since more than 6 years.

Lightbend

Q/A



Lightbend

ktoso @ lightbend.com
twitter: ktosopl
github: ktoso
team blog: blog.akka.io
home: akka.io
myself: kto.so