REACTIVE MICROSYSTEMS

THE EVOLUTION OF MICROSERVICES AT SCALE

JONAS BONÉR

@JBONER

WE HAVE BEEN SPOILED BY

THE ALMIGHTY MONOLITH


KNOCK. KNOCK. WHO'S THERE?
REALITY

Yesterday	Today
Single machines	Clusters of machines
Single core processors	Multicore processors
Expensive RAM	Cheap RAM
Expensive disk	Cheap disk
Slow networks	Fast networks
Few concurrent users	Lots of concurrent users
Small data sets	Large data sets
Latency in seconds	Latency in milliseconds


WE CAN'T MAKE THE HORSE FASTER


WE NEED CARS FOR WHERE WE ARE GOING

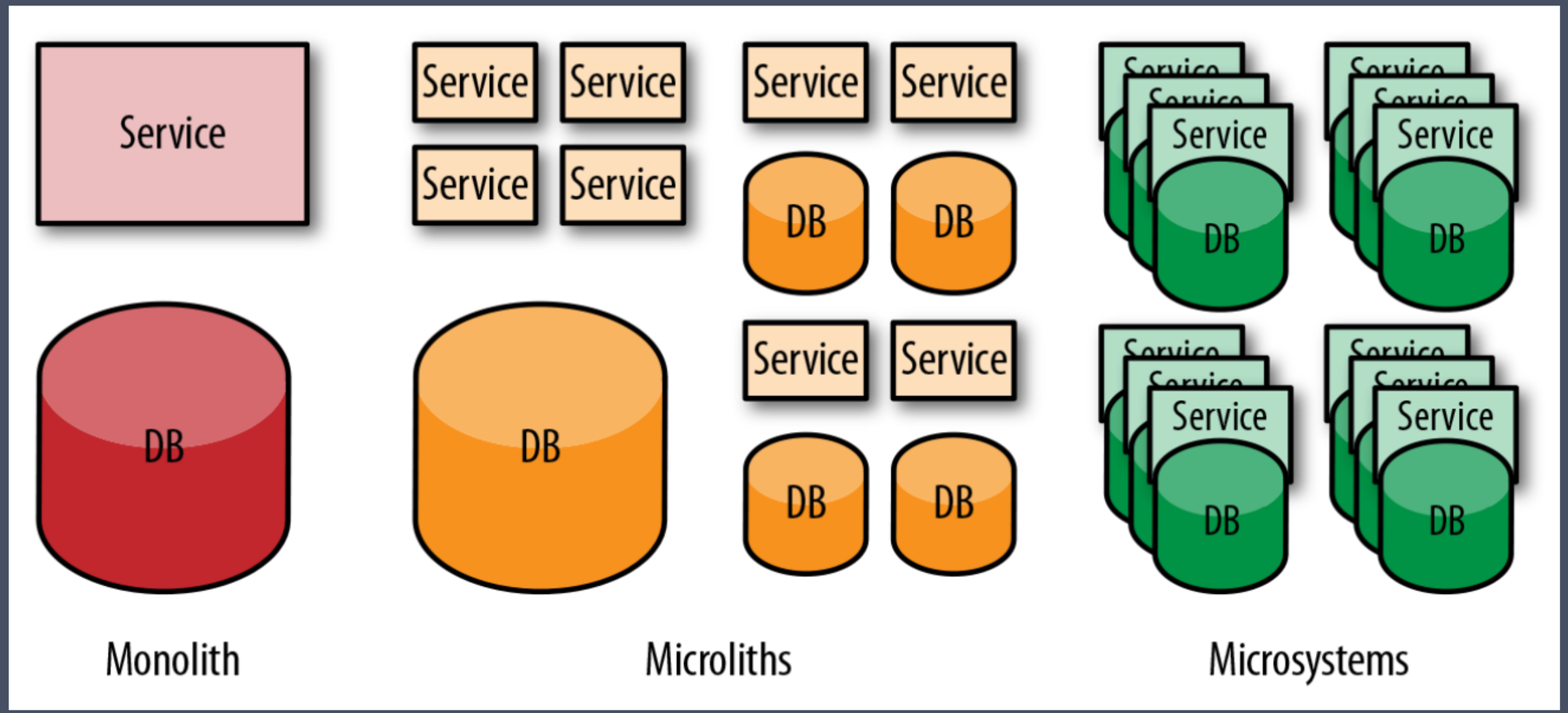
BUT DON'T JUST DRINK THE

THINK FOR YOURSELF

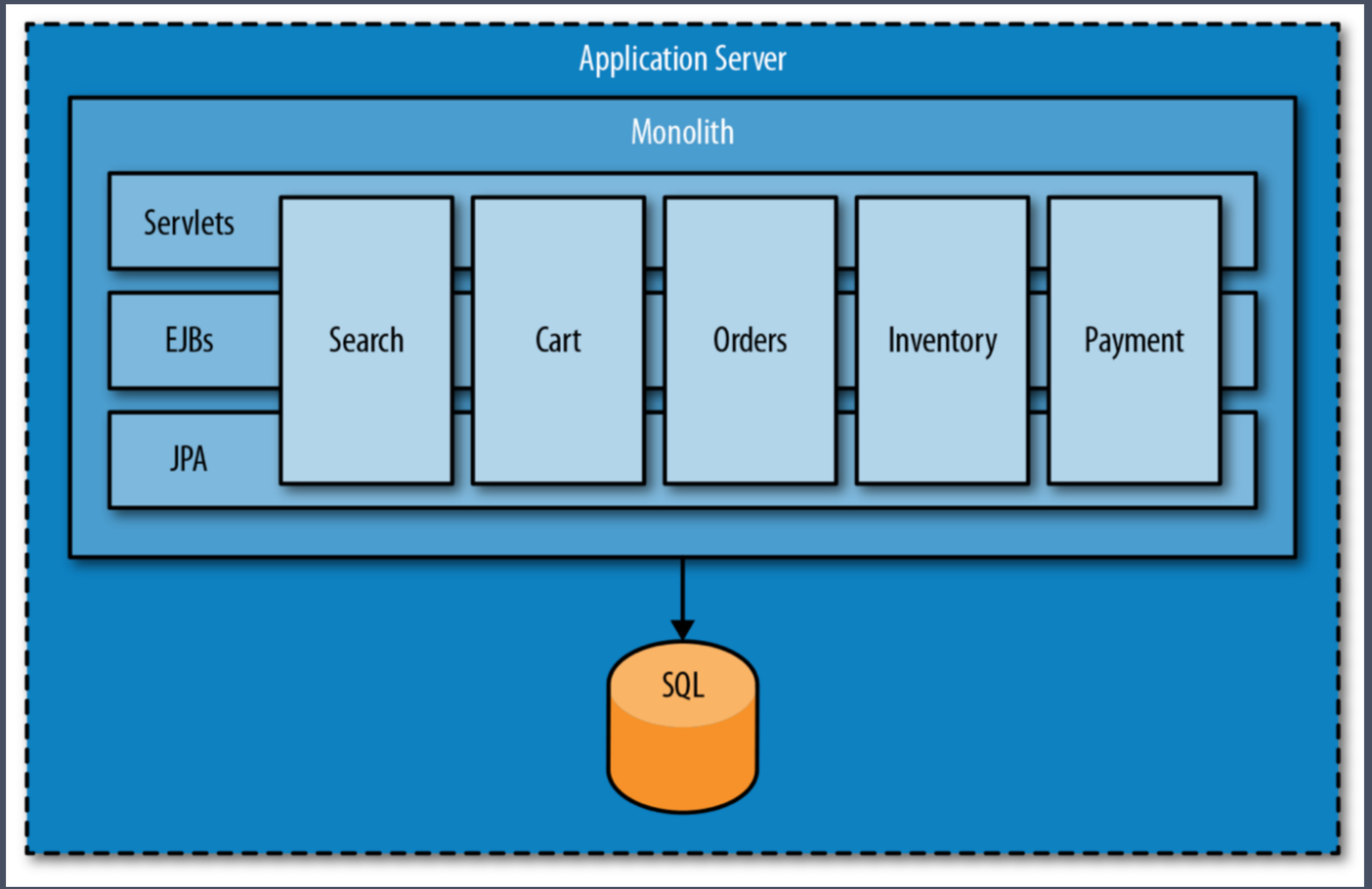
NO ONE WANTS
MICROSERVICES
IT'S A NECCESSARY EVIL

ARCHITECTURAL CONTEXT OF MICROSERVICES:
DISTRIBUTED SYSTEMS

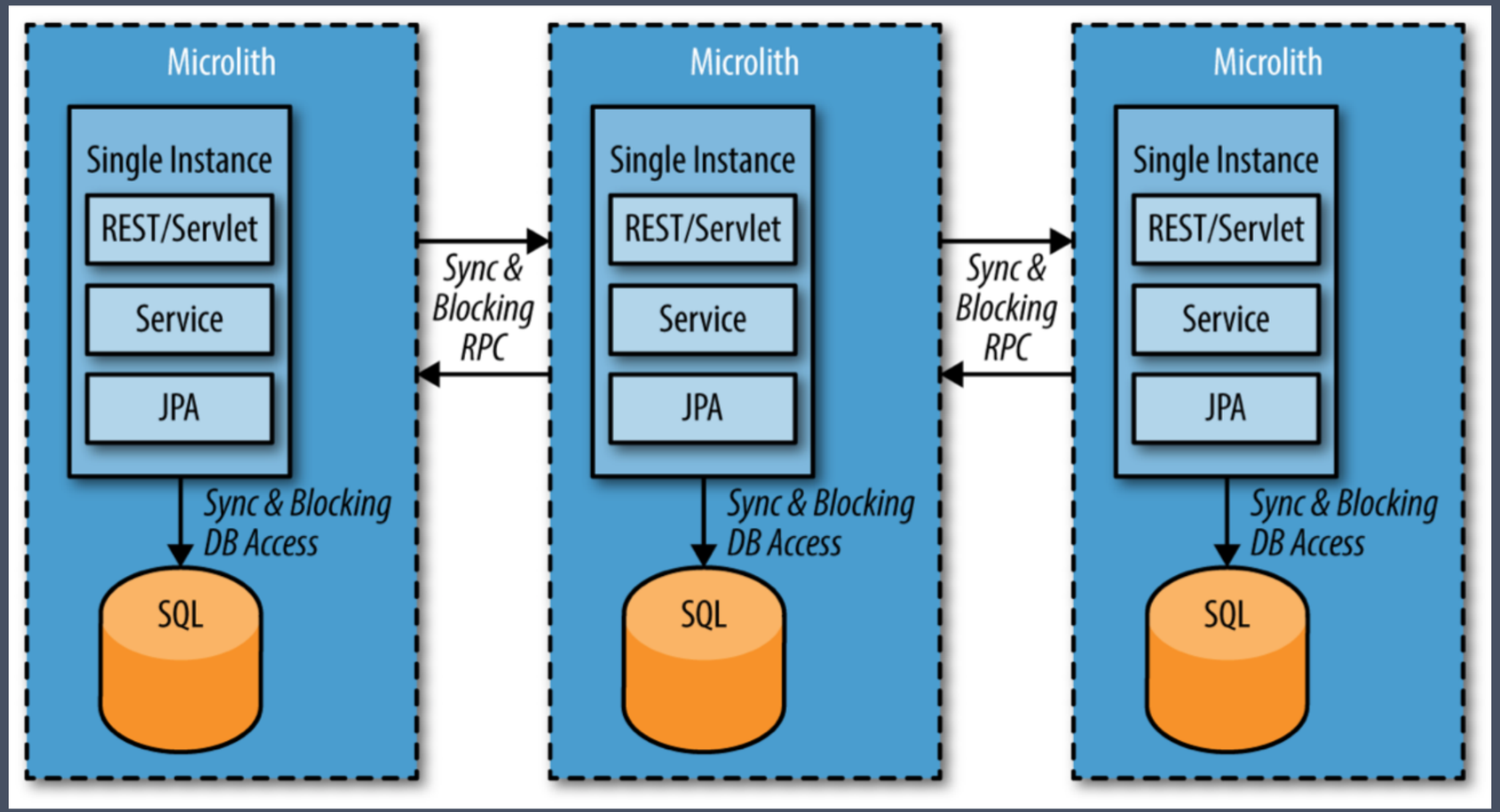
TODAY'S JOURNEY



LET'S SAY THAT WE WANT TO SLICE THIS MONOLITH UP



TOO MANY END UP WITH AN ARCHITECTURE LIKE THIS



MICROLITH:

SINGLE INSTANCE MICROSERVICE

- NOT RESILIENT
- NOT ELASTIC

ONE ACTOR IS NO ACTOR.
ACTORS COME IN SYSTEMS.

- CARL HEWITT

MICROSERVICES COME IN SYSTEMS

3 HELPFUL TOOLS

1. EVENTS-FIRST DDD
2. REACTIVE DESIGN
3. EVENT-BASED PERSISTENCE

PRACTICE

EVENTS-FIRST DOMAIN-DRIVEN DESIGN

DON'T FOCUS ON THE THINGS

THE NOUNS
THE DOMAIN OBJECTS

FOCUS ON WHAT HAPPENS

THE VERBS
THE EVENTS

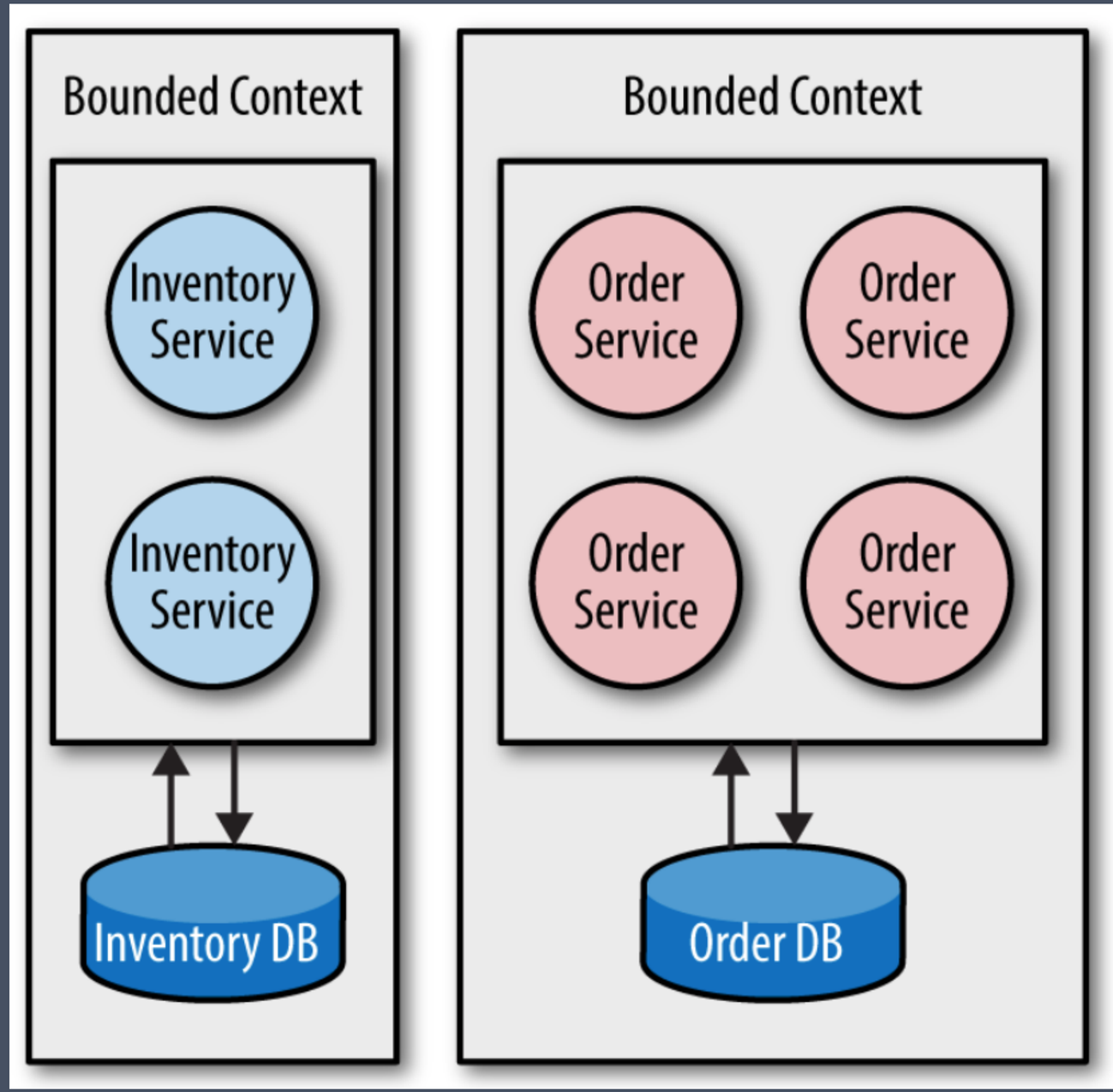
WHEN YOU START MODELING EVENTS.
IT FORCES YOU TO THINK ABOUT THE
BEHAVIOR OF THE SYSTEM.
AS OPPOSED TO THINKING ABOUT
STRUCTURE INSIDE THE SYSTEM.

– GREG YOUNG

LET THE

EVENTS DEFINE

THE BOUNDED CONTEXT



EVENTS REPRESENT FACTS

TO CONDENSE FACT FROM THE VAPOR OF NUANCE

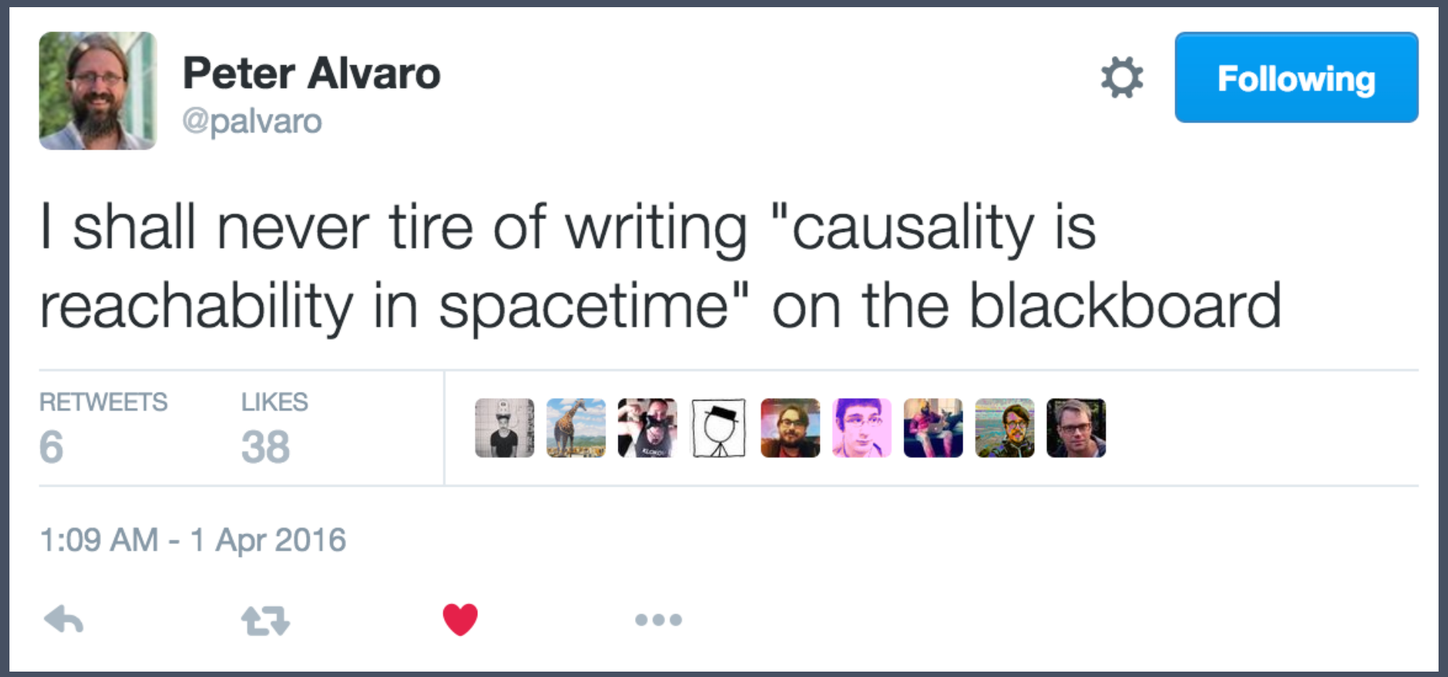
– NEAL STEPHENSON, SNOW CRASH

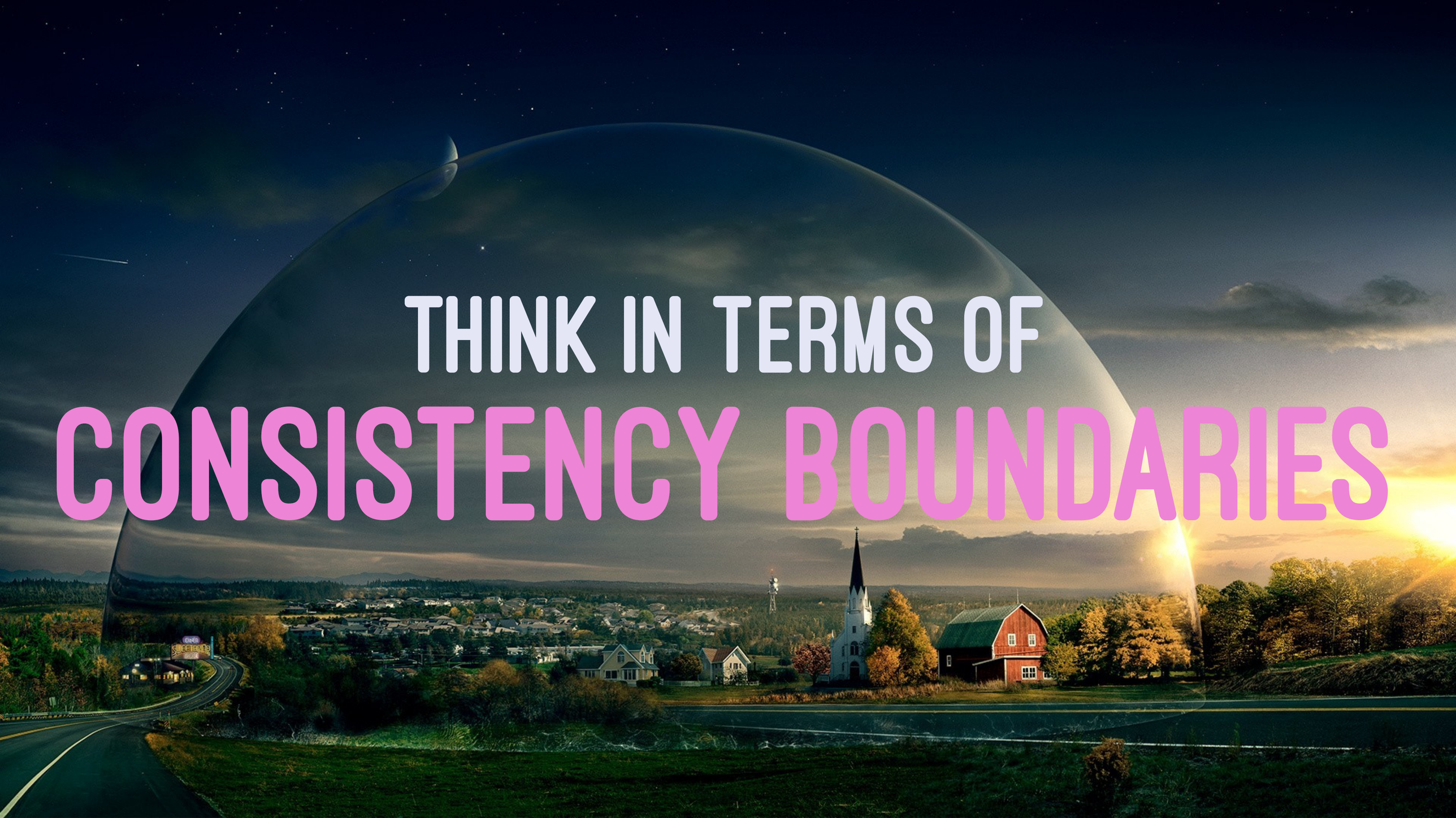
WHAT ARE THE FACTS?


TRY OUT
EVENT STORMING

UNDERSTAND HOW FACTS ARE CAUSALLY RELATED

HOW FACTS FLOW IN THE SYSTEM




THINK IN TERMS OF
CONSISTENCY BOUNDARIES


WE NEED TO
CONTAIN MUTABLE STATE &
PUBLISH FACTS

AGGREGATE

⇒ UNIT OF CONSISTENCY

⇒ UNIT OF FAILURE

PRACTICE

REACTIVE DESIGN

REACTIVE PROGRAMMING

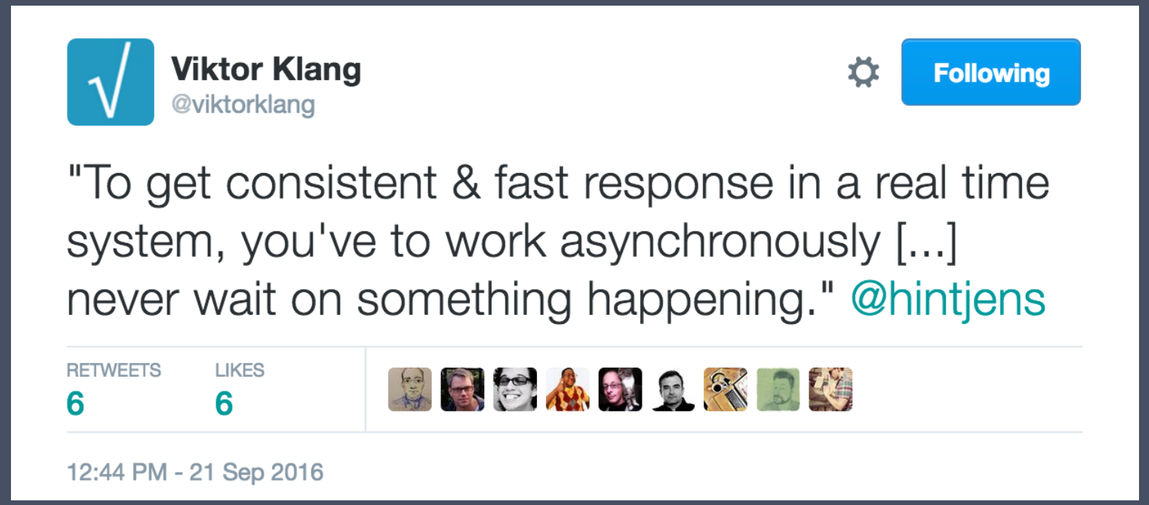
VS

REACTIVE SYSTEMS

REACTIVE PROGRAMMING CAN HELP US MAKE THE INDIVIDUAL INSTANCE HIGHLY PERFORMANT & EFFICIENT

GO

ASYNCHRONOUS



ASYNCHRONOUS & NON-BLOCKING

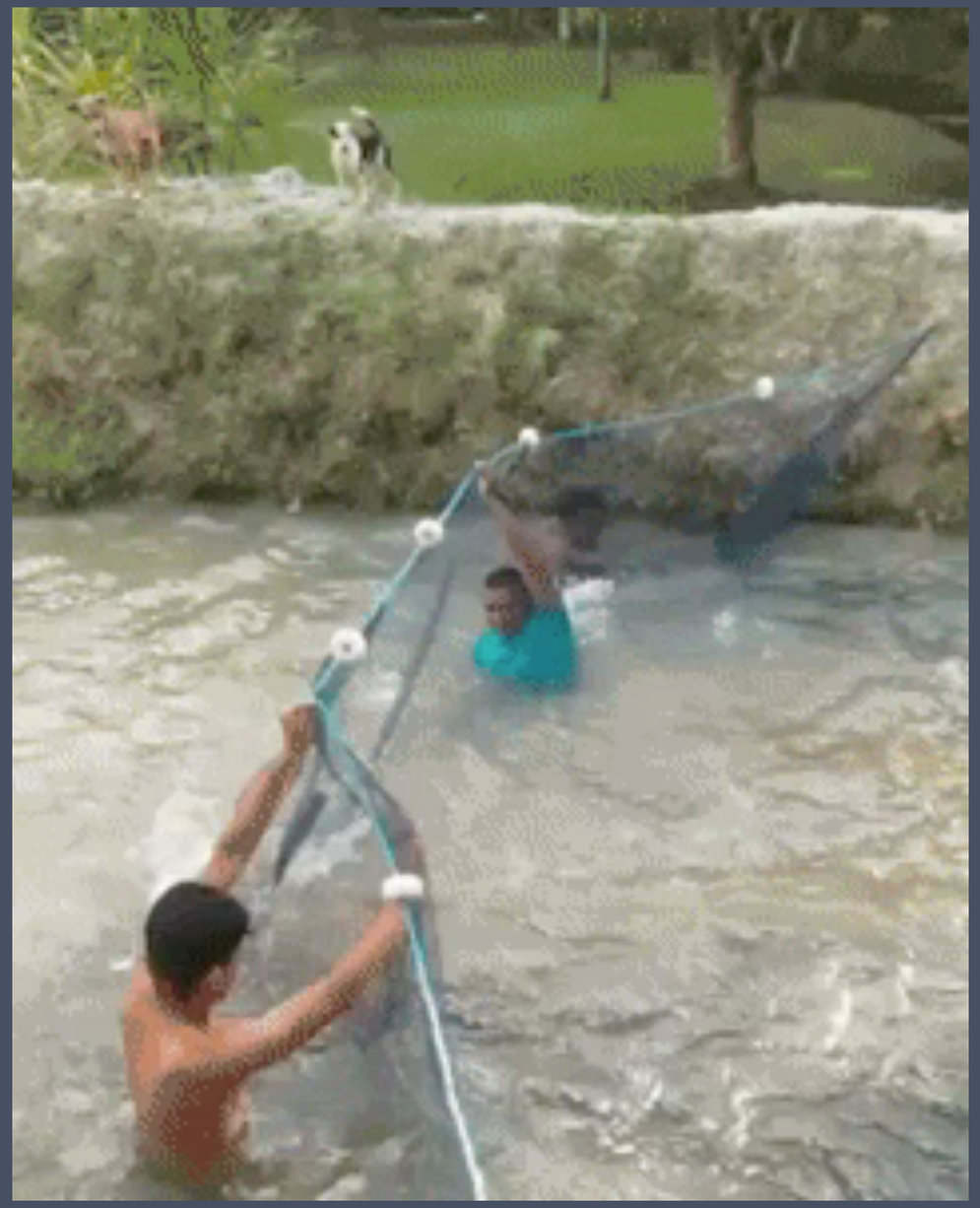
- MORE EFFICIENT USE OF RESOURCES
- MINIMIZES CONTENTION ON SHARED RESOURCES

ALWAYS APPLY BACKPRESSURE

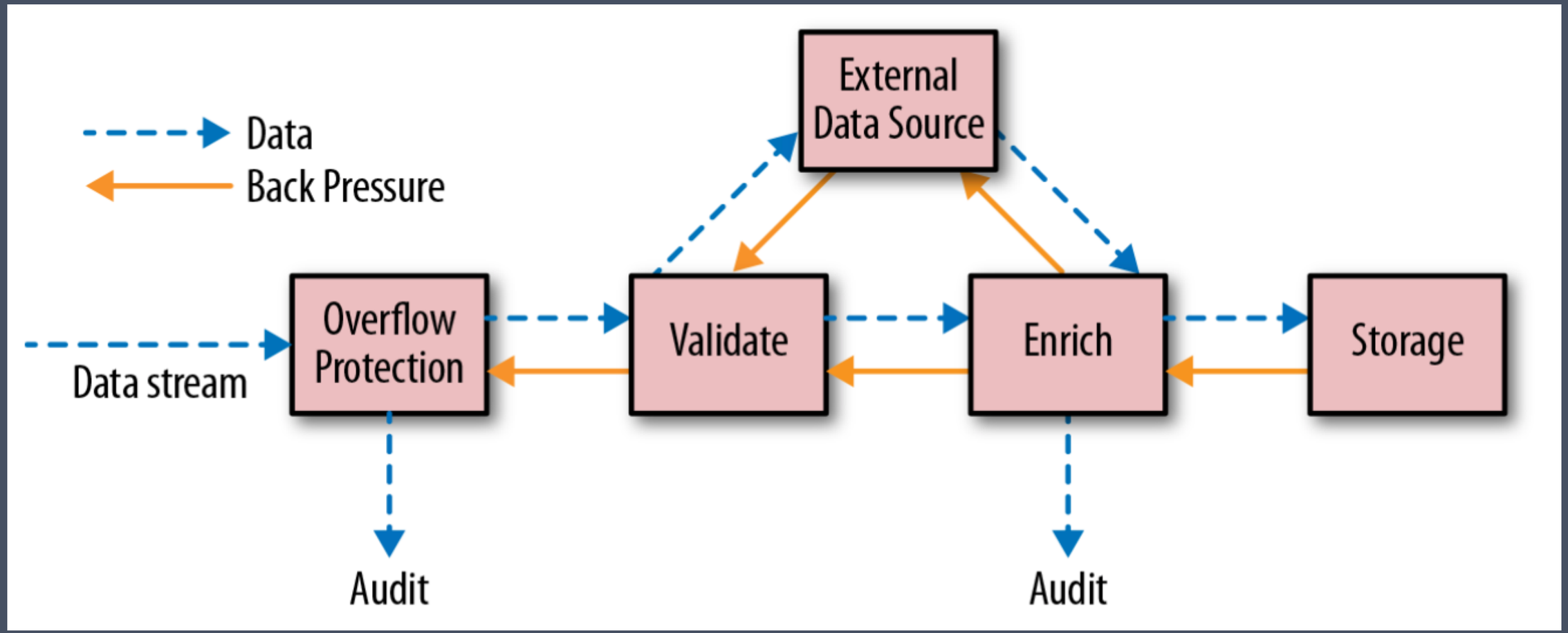
A FAST SYSTEM

SHOULD NOT OVERLOAD

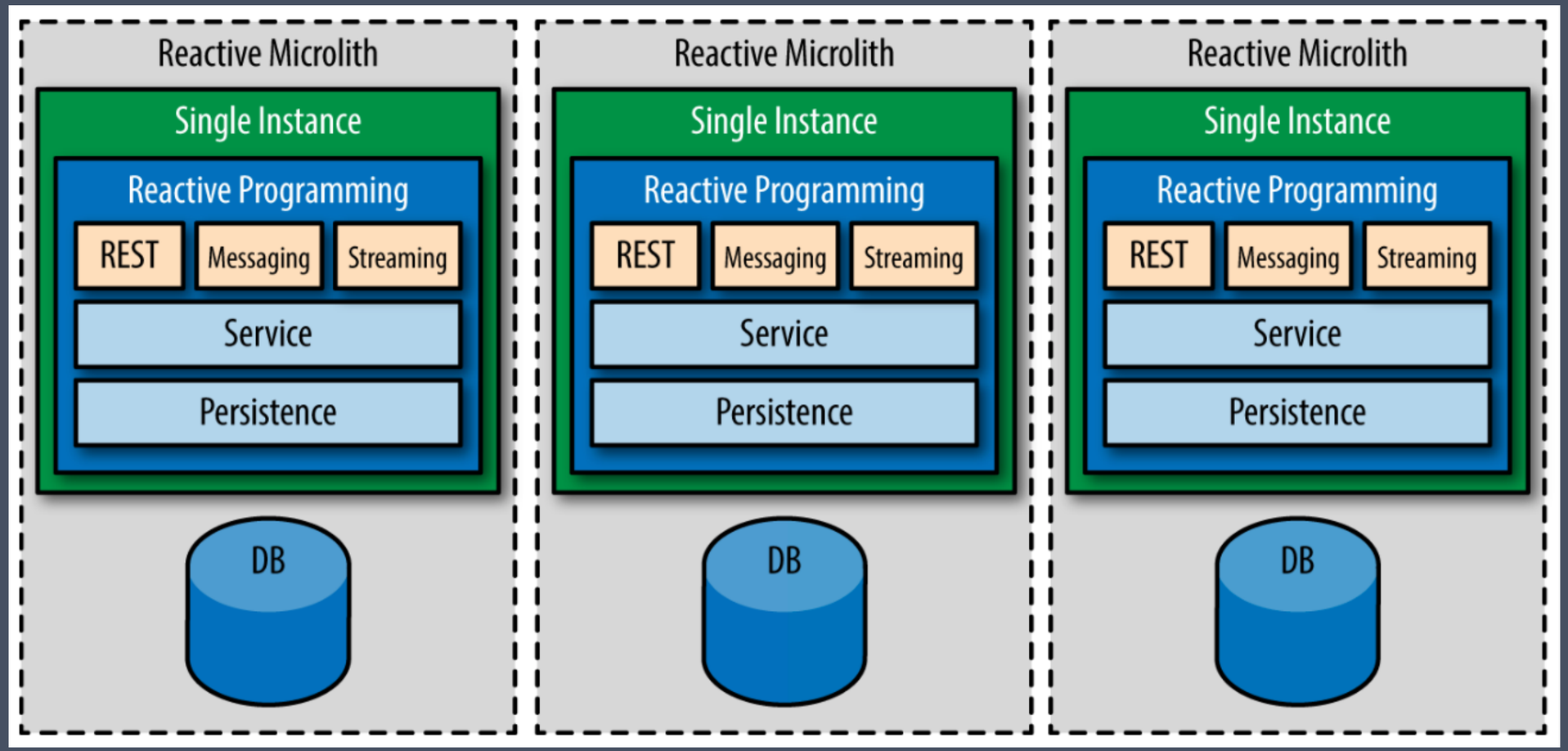
A SLOW SYSTEM



BACKPRESSURE

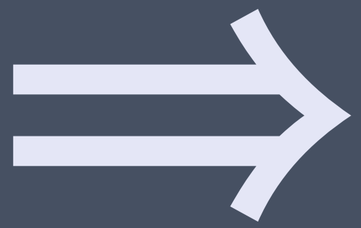


LET'S APPLY REACTIVE PROGRAMMING TO OUR MICROLITHS

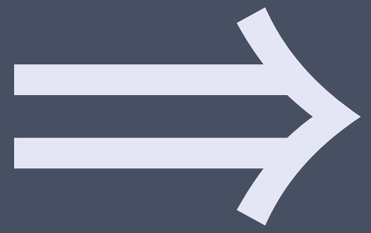


WE'RE GETTING THERE. BUT WE STILL HAVE A

SINGLE INSTANCE MICROSERVICE

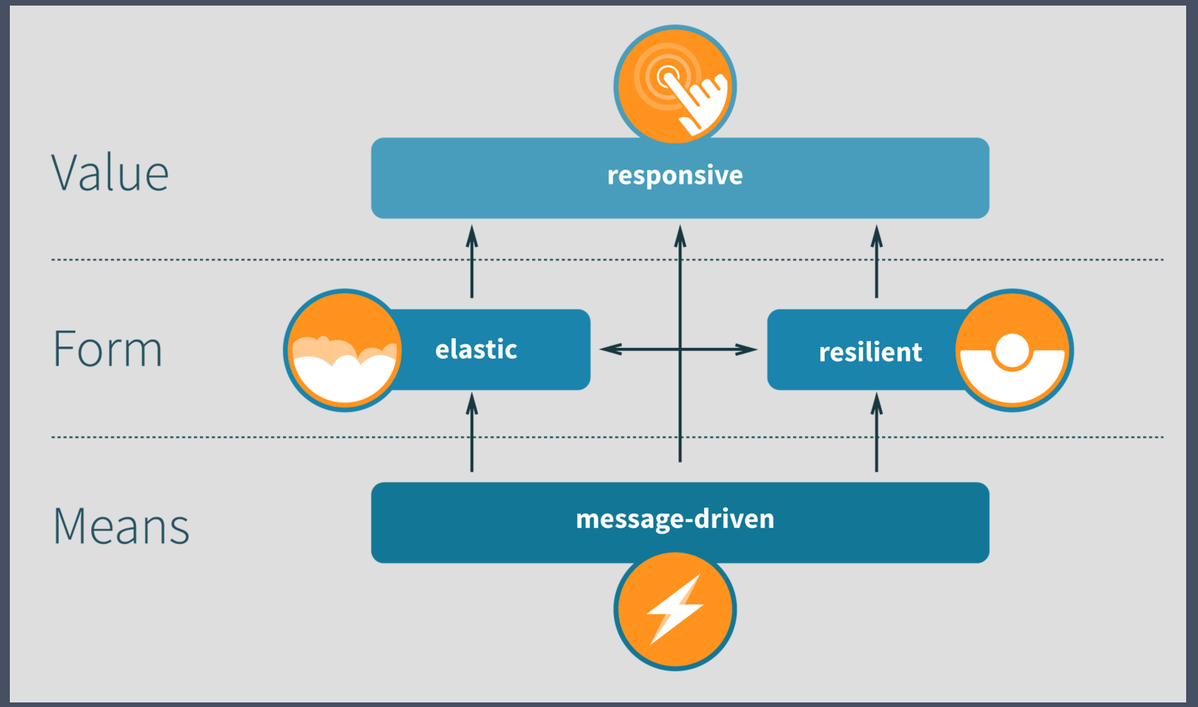


NOT SCALABLE



NOT RESILIENT

REACTIVE SYSTEMS CAN HELP US BUILD DISTRIBUTED SYSTEMS THAT ARE ELASTIC & RESILIENT



REACTIVE SYSTEMS ARE BASED ON

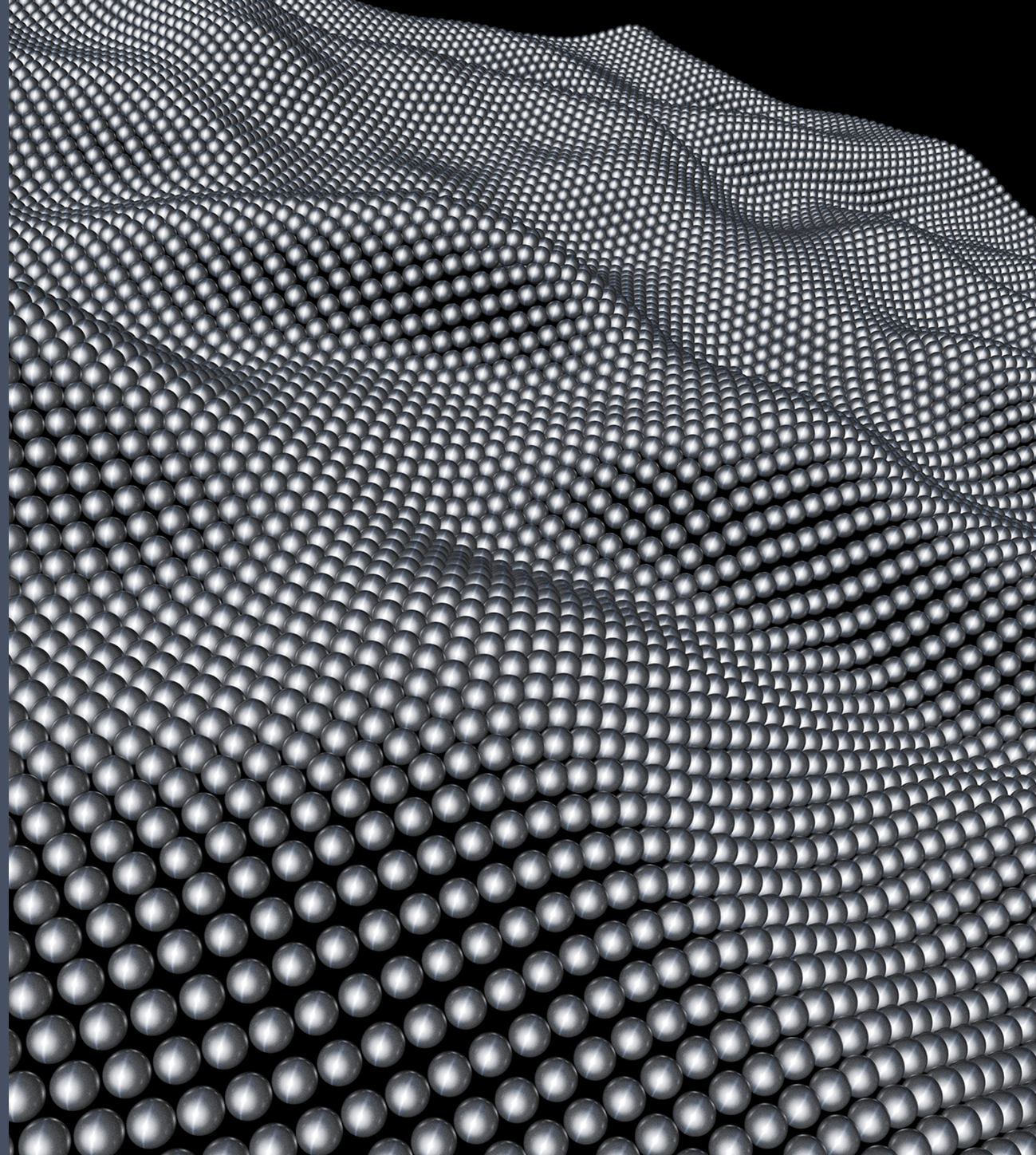
ASYNCHRONOUS MESSAGE-PASSING

ALLOWS DECOUPLING IN

SPACE

AND

TIME



ALLOWS FOR LOCATION TRANSPARENCY

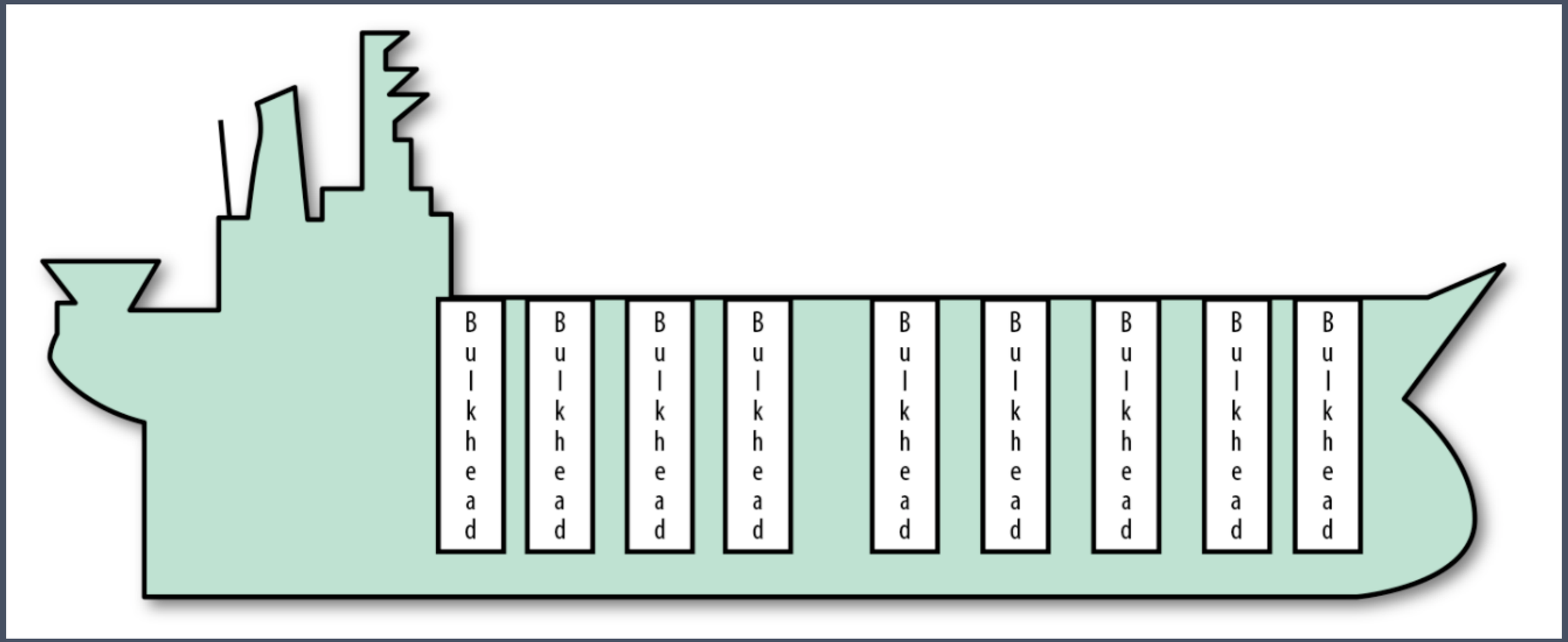
ONE COMMUNICATION ABSTRACTION ACROSS ALL DIMENSIONS OF SCALE

CORE ⇒ SOCKET ⇒ CPU ⇒

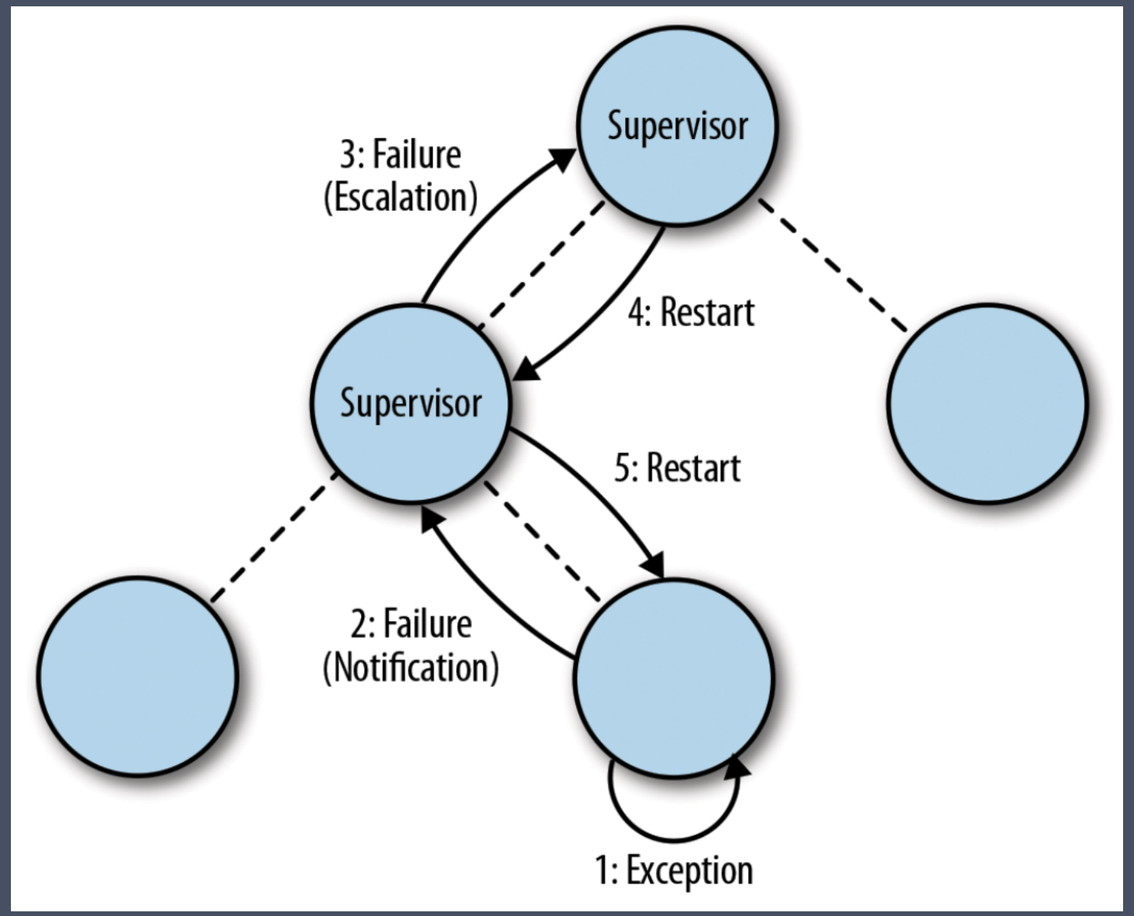
CONTAINER ⇒ SERVER ⇒ RACK ⇒

DATA CENTER ⇒ SYSTEM

SELF-HEALING THROUGH BULKHEADING



SELF-HEALING THROUGH SUPERVISION



MICROSERVICES COME AS SYSTEMS

EACH MICROSERVICE NEEDS BE DESIGNED AS A DISTRIBUTED SYSTEM

A MICROSYSTEM

WE NEED TO MOVE

FROM MICROLITHS TO MICROSYSTEMS

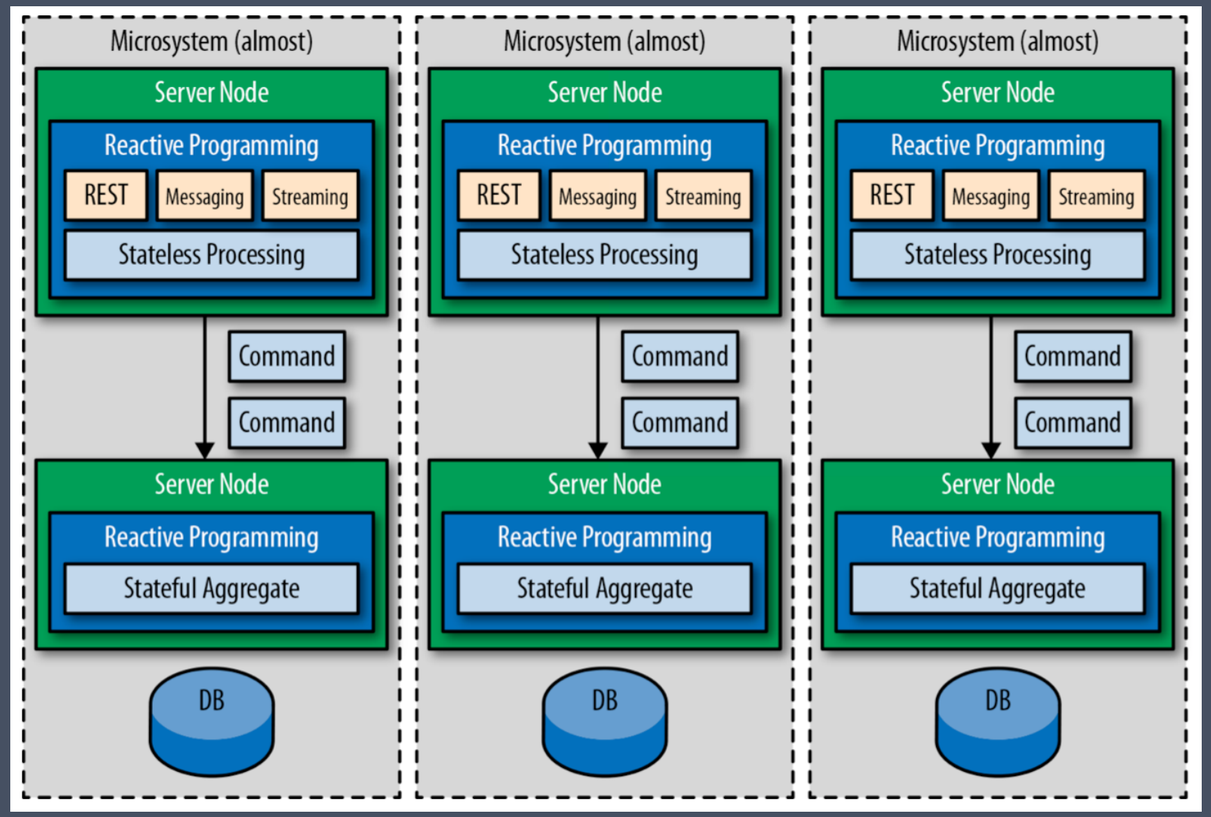
SEPARATE THE

STATELESS BEHAVIOR

FROM THE

STATEFUL ENTITY

TO SCALE THEM INDIVIDUALLY



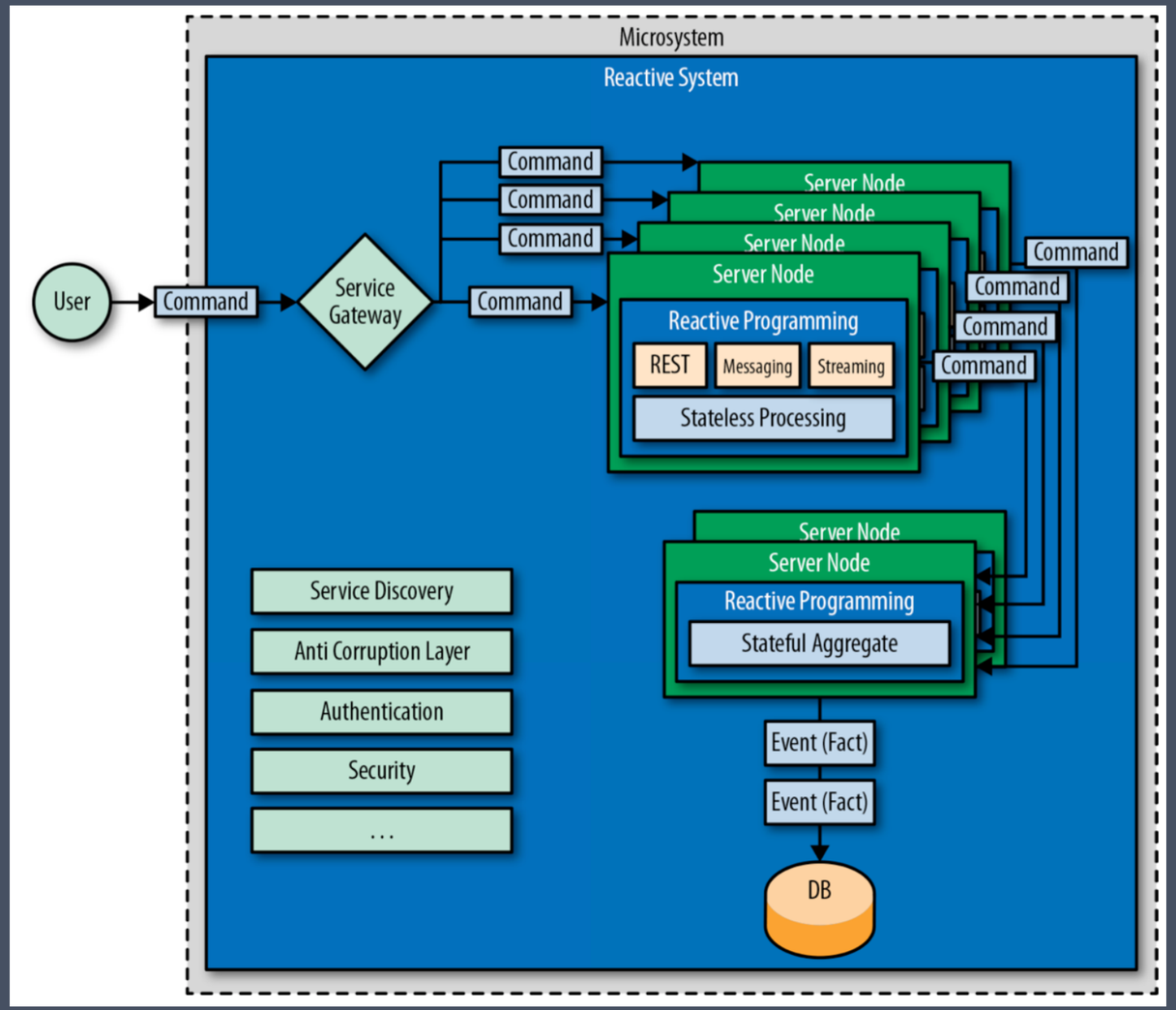
SCALING (STATELESS) BEHAVIOR

IS EASY

SCALING (STATEFUL) ENTITIES IS HARD


THERE IS NO SUCH THING AS A
"STATELESS" ARCHITECTURE
IT'S JUST SOMEONE ELSE'S PROBLEM

REACTIVE MICROSYSTEM



PRACTICE

EVENT-BASED PERSISTENCE

UPDATE-IN-PLACE STRIKES SYSTEMS DESIGNERS AS A CARDINAL SIN: IT VIOLATES TRADITIONAL ACCOUNTING PRACTICES THAT HAVE BEEN OBSERVED FOR HUNDREDS OF YEARS.

– JIM GRAY

THE TRUTH IS THE LOG.
THE DATABASE IS A CACHE OF A SUBSET OF THE LOG.

– PAT HELLAND

EVENT LOGGING

FOR SCALABLE PERSISTENCE

CRUD IS DEAD


In loving memory of two operations that changed the world.
UPDATE and DELETE
1983 - 2013

EVENT SOURCING

A CURE FOR THE CARDINAL SIN

MODELING EVENTS FORCES YOU TO HAVE A TEMPORAL FOCUS ON WHAT'S GOING ON IN THE SYSTEM.
TIME BECOMES A CRUCIAL FACTOR OF THE SYSTEM.

– GREG YOUNG

THE LOG

IS A DATABASE OF THE PAST

NOT JUST A DATABASE OF THE PRESENT

EVENT LOGGING AVOIDS THE INFAMOUS

OBJECT-RELATIONAL IMPEDENCE MISMATCH

UNTANGLE THE READ & WRITE MODELS WITH CQRS

ADVANTAGES OF USING CQRS:

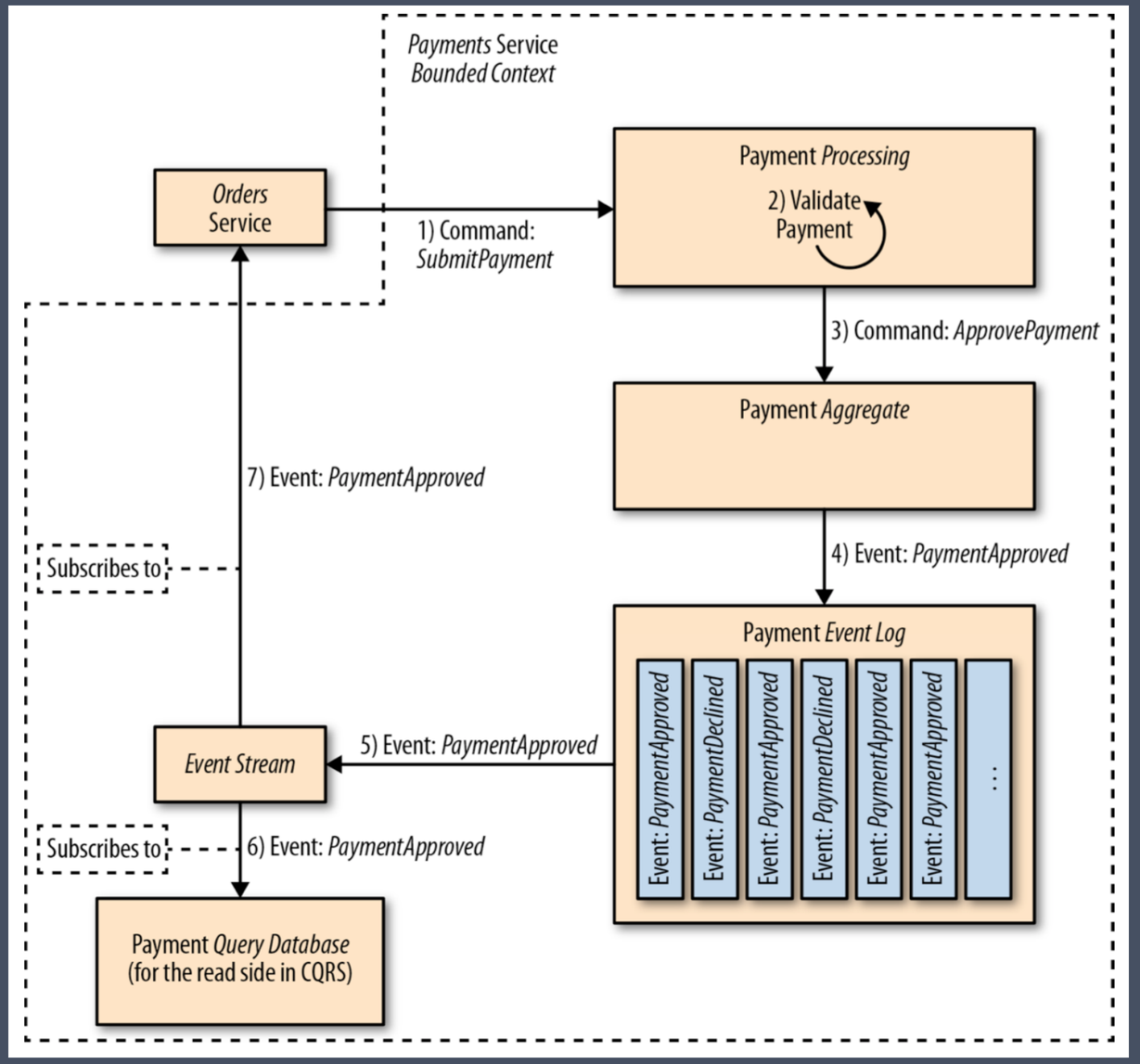
1. RESILIENCE
2. SCALABILITY
3. POLYGLOT PERSISTENCE

USE

EVENT SOURCING

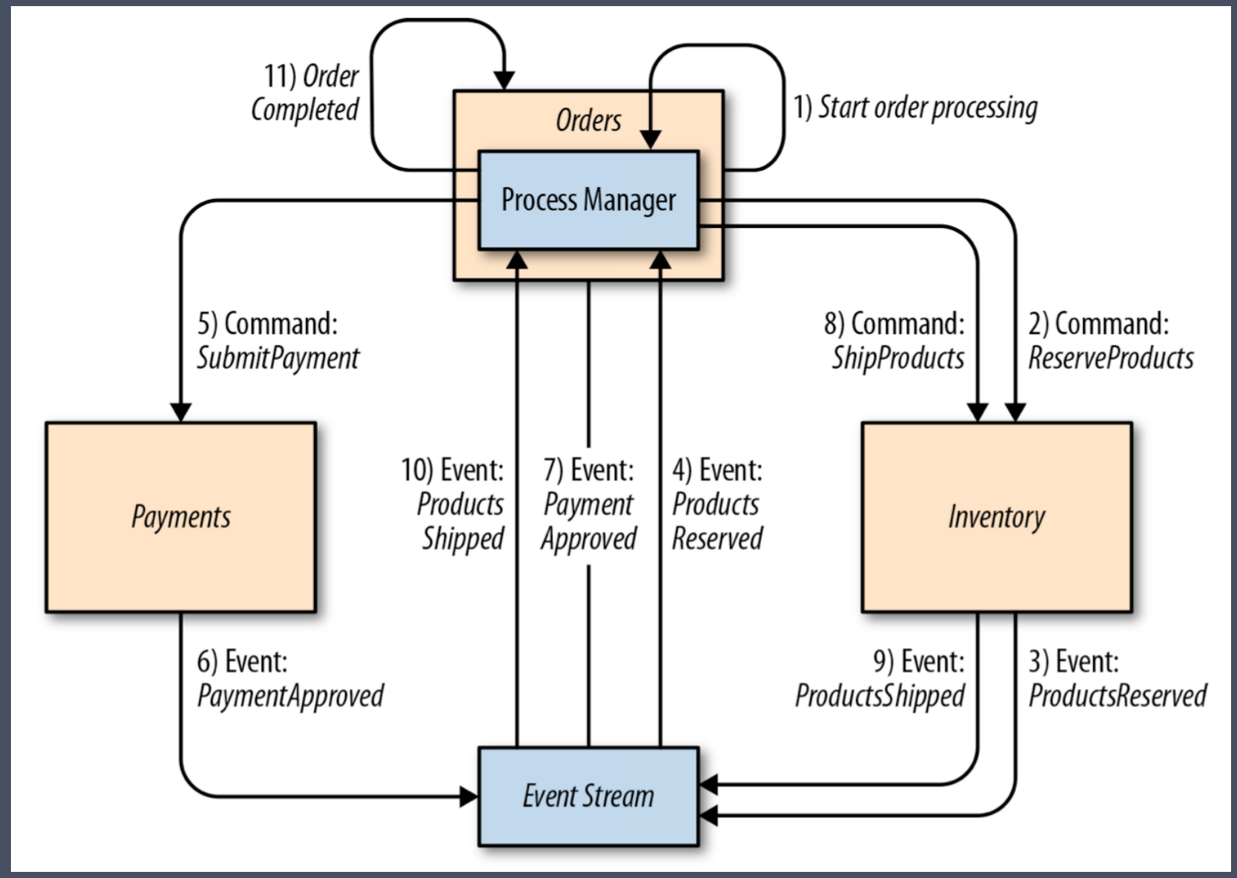
AND

EVENT STREAMING



HOW CAN WE
COORDINATE WORK ACROSS AGGREGATES?

EVENT-DRIVEN WORKFLOW



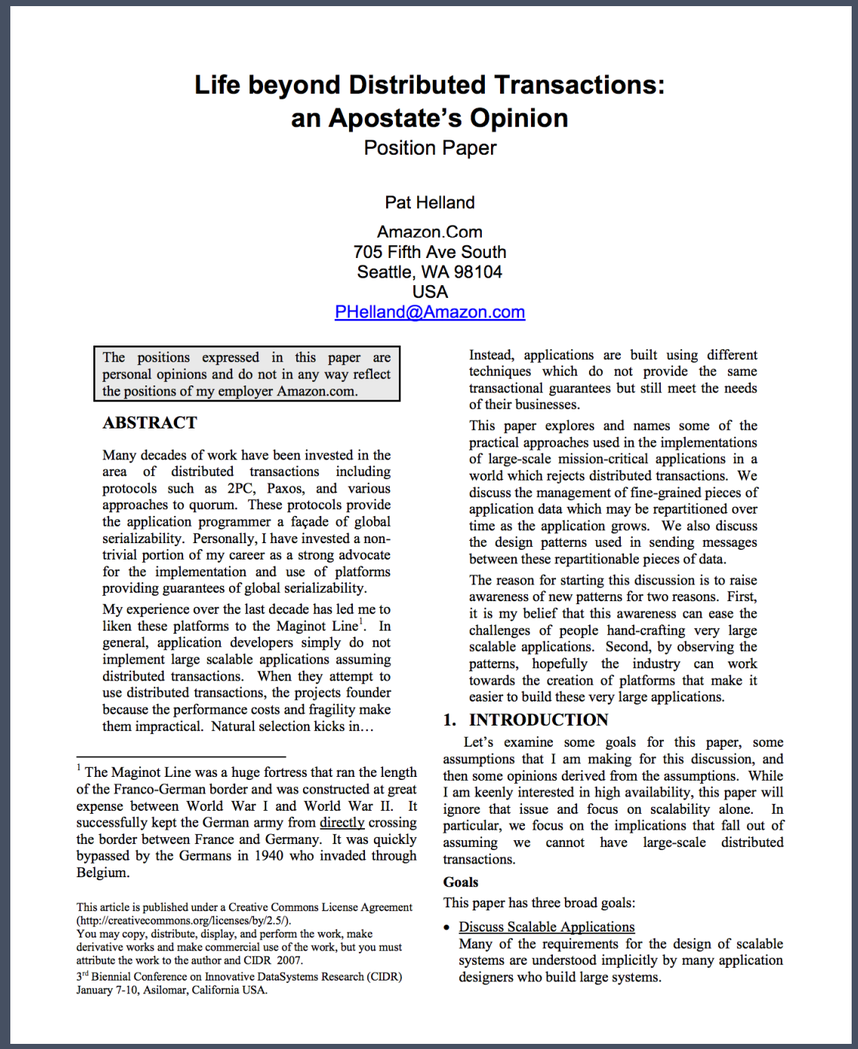
BUT WHAT ABOUT TRANSACTIONS?

TWO-PHASE COMMIT IS THE ANTI-AVAILABILITY PROTOCOL.

- PAT HELLAND

IN GENERAL,
APPLICATION DEVELOPERS
SIMPLY DO NOT IMPLEMENT
LARGE SCALABLE
APPLICATIONS
ASSUMING DISTRIBUTED
TRANSACTIONS.

– PAT HELLAND



Life beyond Distributed Transactions: an Apostate's Opinion

Position Paper

Pat Helland

Amazon.Com
705 Fifth Ave South
Seattle, WA 98104
USA
PHelland@Amazon.com

The positions expressed in this paper are personal opinions and do not in any way reflect the positions of my employer Amazon.com.

ABSTRACT

Many decades of work have been invested in the area of distributed transactions including protocols such as 2PC, Paxos, and various approaches to quorum. These protocols provide the application programmer a façade of global serializability. Personally, I have invested a non-trivial portion of my career as a strong advocate for the implementation and use of platforms providing guarantees of global serializability.

My experience over the last decade has led me to liken these platforms to the Maginot Line[1]. In general, application developers simply do not implement large scalable applications assuming distributed transactions. When they attempt to use distributed transactions, the projects founder because the performance costs and fragility make them impractical. Natural selection kicks in… Instead, applications are built using different techniques which do not provide the same transactional guarantees but still meet the needs of their businesses.

This paper explores and names some of the practical approaches used in the implementations of large-scale mission-critical applications in a world which rejects distributed transactions. We discuss the management of fine-grained pieces of application data which may be repartitioned over time as the application grows. We also discuss the design patterns used in sending messages between these repartitionable pieces of data.

The reason for starting this discussion is to raise awareness of new patterns for two reasons. First, it is my belief that this awareness can ease the challenges of people hand-crafting very large scalable applications. Second, by observing the patterns, hopefully the industry can work towards the creation of platforms that make it easier to build these very large applications.

[1] The Maginot Line was a huge fortress that ran the length of the Franco-German border and was constructed at great expense between World War I and World War II. It successfully kept the German army from directly crossing the border between France and Germany. It was quickly bypassed by the Germans in 1940 who invaded through Belgium.

This article is published under a Creative Commons License Agreement (http://creativecommons.org/licenses/by/2.5/).
You may copy, distribute, display, and perform the work, make derivative works and make commercial use of the work, but you must attribute the work to the author and CIDR 2007.

3rd Biennial Conference on Innovative DataSystems Research (CIDR) January 7-10, Asilomar, California USA.

1. INTRODUCTION

Let's examine some goals for this paper, some assumptions that I am making for this discussion, and then some opinions derived from the assumptions. While I am keenly interested in high availability, this paper will ignore that issue and focus on scalability alone. In particular, we focus on the implications that fall out of assuming we cannot have large-scale distributed transactions.

Goals

This paper has three broad goals:

- Discuss Scalable Applications
  Many of the requirements for the design of scalable systems are understood implicitly by many application designers who build large systems.

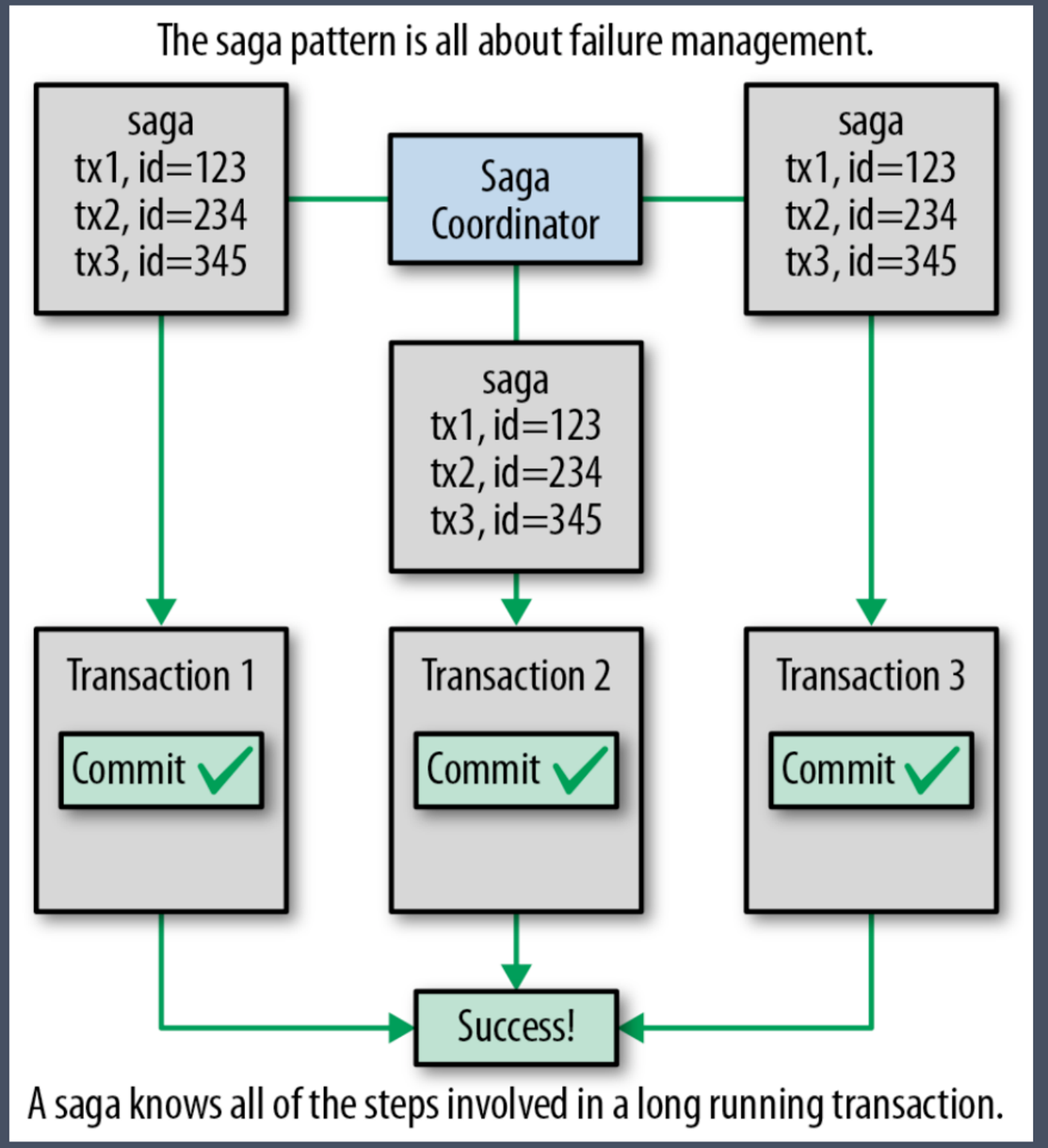
USE A PROTOCOL OF

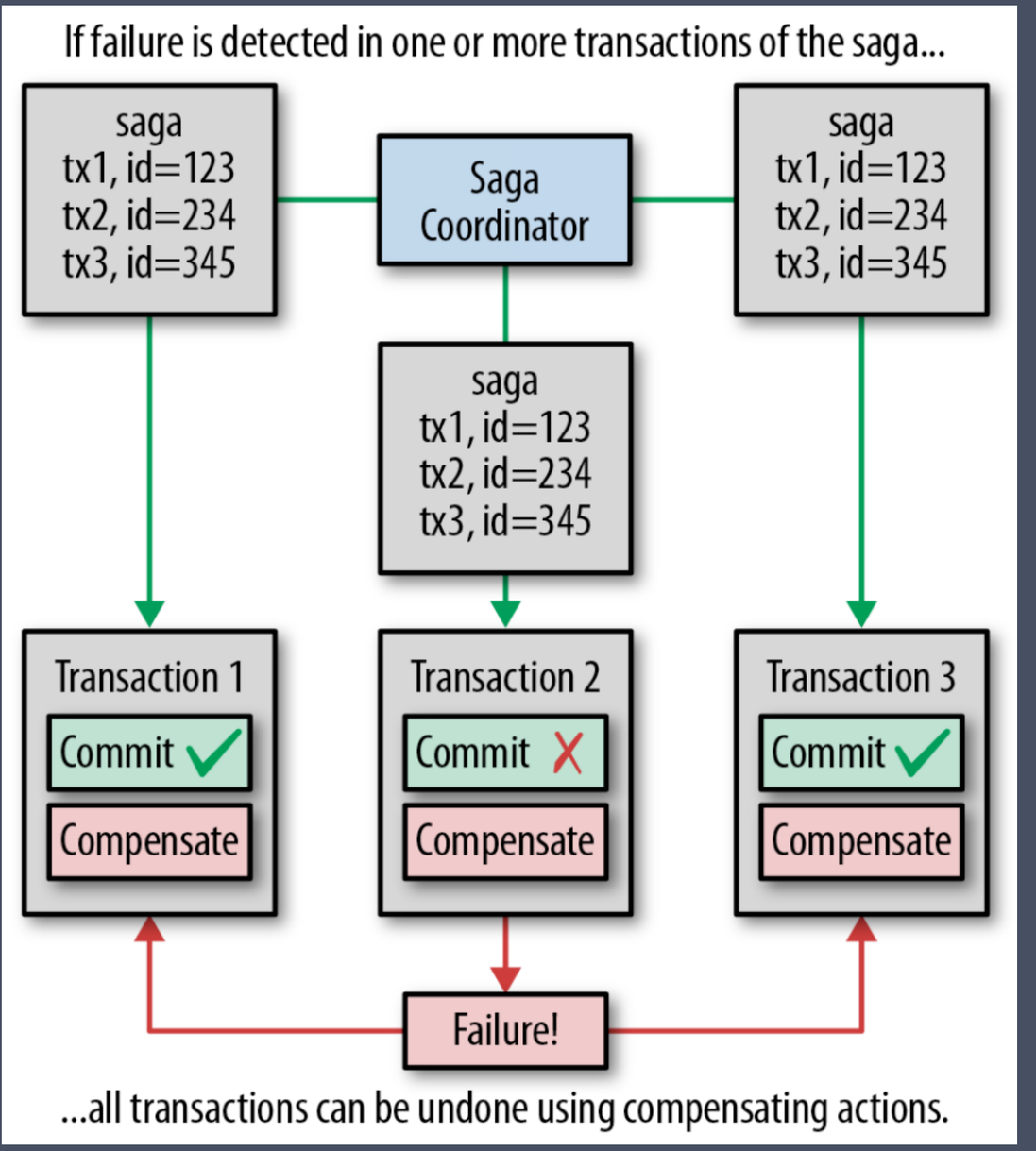
GUESS.
APOLOGIZE.
COMPENSATE.

IT'S HOW THE WORLD WORKS

USE SAGAS

NOT DISTRIBUTED TRANSACTIONS


The saga pattern is all about failure management.
saga
tx1, id=123
tx2, id=234
tx3, id=345
Saga
Coordinator
saga
tx1, id=123
tx2, id=234
tx3, id=345
saga
tx1, id=123
tx2, id=234
tx3, id=345
Transaction 1
Commit
Transaction 2
Commit
Transaction 3
Commit
Success!
A saga knows all of the steps involved in a long running transaction.


If failure is detected in one or more transactions of the saga...
saga
tx1, id=123
tx2, id=234
tx3, id=345
Saga
Coordinator
saga
tx1, id=123
tx2, id=234
tx3, id=345
saga
tx1, id=123
tx2, id=234
tx3, id=345
Transaction 1
Commit
Compensate
Transaction 2
Commit
Compensate
Transaction 3
Commit
Compensate
Failure!
...all transactions can be undone using compensating actions.

IN SUMMARY

1. DON'T BUILD MICROLITHS
2. MICROSERVICES COME IN (DISTRIBUTED) SYSTEMS
3. MICROSERVICES COME AS (MICRO)SYSTEMS
4. EMBRACE THE REACTIVE PRINCIPLES
5. EMBRACE EVENT-FIRST DDD & PERSISTENCE
6. PROFIT!

TRY THE

LAGOM

MICROSERVICES FRAMEWORK

POWERED BY AKKA & PLAY

LAGOMFRAMEWORK.COM

lagom

LEARN MORE

DOWNLOAD MY BOOK FOR FREE AT:

BIT.LY/REACTIVE-MICROSYSTEMS