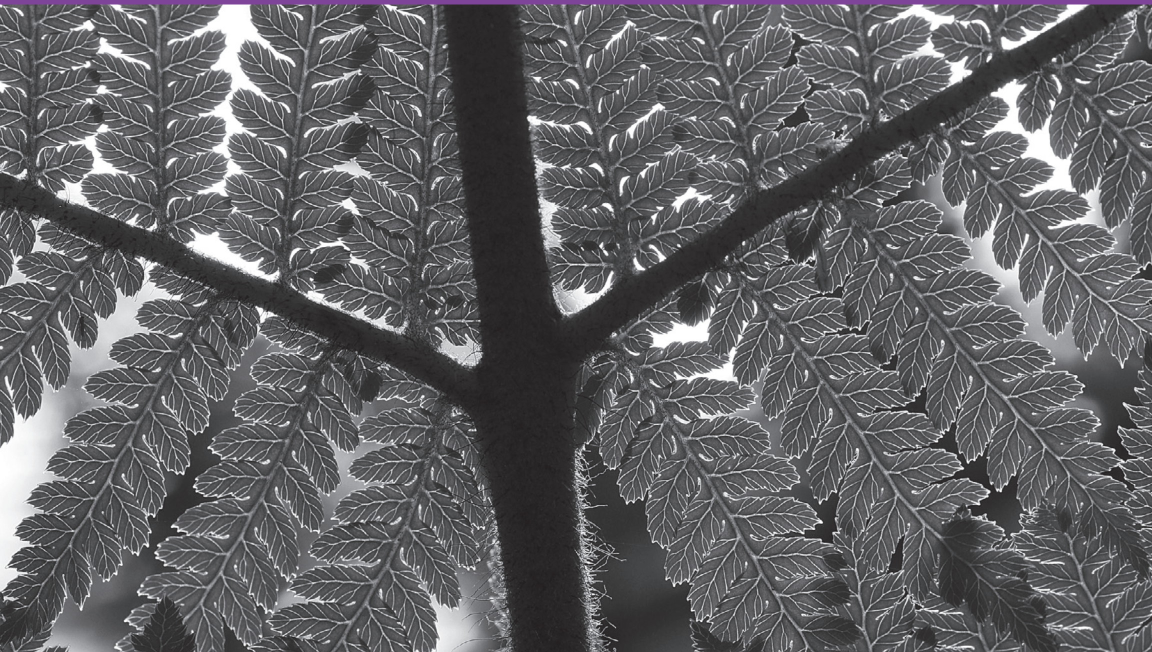
O'REILLY®

Compliments of Lightbend

Reactive Microsystems

The Evolution of Microservices at Scale



Jonas Bonér

REACTIVE MICROSERVICES WITH LAGOM

Deploy, scale, and manage effortlessly.

Try Lagom in production.
Get started today.
lightbend.com/lagom

Lightbend

Reactive Microsystems

The Evolution of Microservices at Scale

Jonas Bonér

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Reactive Microsystems

by Jonas Bonér

Copyright © 2017 Lightbend, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Brian Foster
Production Editor: Melanie Yarbrough
Copyeditor: Octal Publishing Services
Proofreader: Matthew Burgoyne
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

August 2017: First Edition

Revision History for the First Edition

2017-08-07: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Reactive Microsystems*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-99433-7

[LSI]

Table of Contents

Introduction v

1. Essential Traits of an Individual Microservice 1
Isolate All the Things 1
Single Responsibility 2
Own Your State, Exclusively 3
Stay Mobile, but Addressable 6

2. Slaying the Monolith 7
Don't Build Microliths 9

3. Microservices Come in Systems 11
Embrace Uncertainty 11
We Are Always Looking into the Past 12
The Cost of Maintaining the Illusion of a Single Now 13
Learn to Enjoy the Silence 13
Avoid Needless Consistency 14

4. Events-First Domain-Driven Design 17
Focus on What Happens: The Events 17
Think in Terms of Consistency Boundaries 21
Manage Protocol Evolution 25

5. Toward Reactive Microsystems 27
Embrace Reactive Programming 28
Embrace Reactive Systems 35
Microservices Come as Systems 44

6. **Toward Scalable Persistence. 49**
Moving Beyond CRUD 49
Event Logging—The Scalable Seamstress 50
Transactions—The Anti-Availability Protocol 59

7. **The World Is Going Streaming. 67**
Three Waves Toward Fast Data 68
Leverage Fast Data in Microservices 68

8. **Next Steps. 71**
Further Reading 71
Start Hacking 72

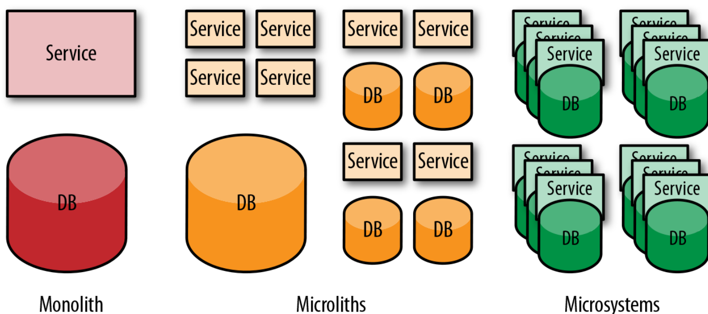
Introduction

The Evolution of Scalable Microservices

In this report, I will discuss strategies and techniques for building scalable and resilient microservices, working our way through the evolution of a microservices-based system.

Beginning with a monolithic application, we will refactor it, briefly land at the antipattern of single instance—not scalable or resilient—microliths (micro monoliths), before quickly moving on, and step by step work our way toward scalable and resilient microservices (microsystems).

Along the way, we will look at techniques from reactive systems, reactive programming, event-driven programming, events-first domain-driven design, event sourcing, command query responsibility segregation, and more.



We Can't Make the Horse Faster

> *If I had asked people what they wanted, they would have said faster horses.*
>
> —Henry Ford[1]

Today's applications are deployed to everything from mobile devices to cloud-based clusters running thousands of multicore processors. Users have come to expect millisecond response times (latency) and close to 100 percent uptime. And, by "user," I mean both humans and machines. Traditional architectures, tools, and products as such simply won't cut it anymore. We need new solutions that are as different from monolithic systems as cars are from horses.

Figure P-1 sums up some of the changes that we have been through over the past 10 to 15 years.

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

Figure P-1. Some fundamental changes over the past 10 to 15 years

To paraphrase Henry Ford's classic quote: we can't make the horse faster anymore; we need cars for where we are going.

So, it's time to wake up, time to retire the monolith, and to decompose the system into manageable, discrete services that can be scaled individually, that can fail, be rolled out, and upgraded in isolation.

1 It's been debated whether Henry Ford actually said this. He probably didn't. Regardless, it's a great quote.

They have had many names over the years (DCOM, CORBA, EJBs, WebServices, etc.). Today, we call them *microservices*. We, as an industry, have gone full circle again. Fortunately, it is more of an upward spiral as we are getting a little bit better at it every time around.

We Need to Learn to Exploit Reality

> *Imagination is the only weapon in the war against reality.*
> —Lewis Carroll, *Alice in Wonderland*

We have been spoiled by the once-believed-almighty monolith—with its single SQL database, in-process address space, and thread-per-request model—for far too long. It's a fairytale world in which we could assume strong consistency, one single globally consistent "now" where we could comfortably forget our university classes on distributed systems.

Knock. Knock. Who's There? Reality! We have been living in this illusion, far from reality.

We will look at microservices, not as tools to scale the organization and the development and release process (even though it's one of the main reasons for adopting microservices), but from an architecture and design perspective, and put it in its true architectural context: *distributed systems*.

One of the major benefits of microservices-based architecture is that it gives us a set of tools to exploit reality, to create systems that closely mimic how the world works.

Don't Just Drink the Kool-Aid

Everyone is talking about microservices in hype-cycle speak; they are reaching the peak of inflated expectations. It is very important to not just drink the Kool-Aid blindly. In computer science, it's all about trade-offs, and microservices come with a cost. Microservices can do wonders for the development speed, time-to-market, and Continuous Delivery for a large organization, and it can provide a great foundation for building elastic and resilient systems that can

take full advantage of the cloud.[2] That said, it also can introduce unnecessary complexity and simply slow you down. In other words, do not apply microservices blindly. Think for yourself.

2 If approached from the perspective of distributed systems, which is the topic of this report.

CHAPTER 1

Essential Traits of an Individual Microservice

In my previous book, *Reactive Microservices Architecture*, I discussed the essential traits of a microservice: isolation, autonomicity, single responsibility, exclusive state, and mobility. Let's take a few minutes to recap the essence of these traits.

Isolate All the Things

> *Without great solitude, no serious work is possible.*
> —Pablo Picasso

Isolation is the most important trait and the foundation for many of the high-level benefits in microservices.

Isolation also has the biggest impact on your design and architecture. It will, and should, slice up the entire architecture, and therefore it needs to be considered from day one.

It will even affect the way you break up and organize the teams and their responsibilities, as Melvyn Conway discovered in 1967 (later named Conway's Law):

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

Isolation between services makes it natural to adopt Continuous Delivery (CD). This makes it possible for you to safely deploy appli-

cations and roll out and revert changes incrementally, service by service.

Isolation makes it easier to scale each service, as well as allowing them to be monitored, debugged, and tested independently—something that is very difficult if the services are all tangled up in the big bulky mess of a monolith.

Act Autonomously

> *In a network of autonomous systems, an agent is only concerned with assertions about its own policy; no external agent can tell it what to do, without its consent. This is the crucial difference between autonomy and centralized management.*
>
> —Mark Burgess, *Promise Theory*

Isolation is a prerequisite for *autonomy*. Only when services are isolated can they be fully autonomous and make decisions independently, act independently, and cooperate and coordinate with others to solve problems.

Working with autonomous services opens up flexibility around service orchestration, workflow management, and collaborative behavior, as well as scalability, availability, and runtime management, at the cost of putting more thought into well-defined and composable APIs.

But autonomy cuts deeper, affecting more than the architecture and design of the system. A design with autonomous services allows the teams that build the services to stay autonomous relative to one another—rolling out new services and new features in existing services independently, and so on.

Autonomy is the foundation on which we can scale both the system and the development organization.

Single Responsibility

> *This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together.*
>
> —Doug McIlroy

The Unix philosophy[1] and design has been highly successful and still stands strong decades after its inception. One of its core principles is that developers should write programs that have a single purpose—a small, well-defined responsibility, and compose it well so it works well with other small programs.

This idea was later brought into the Object-Oriented Programming community by Robert C. Martin and named the Single Responsibility Principle[2] (SRP), which states that a class or component should *"have only one reason to change."*

There has been a lot of discussion around the true size of a microservice. What can be considered "micro"? How many lines of code can it be and still be a microservice? These are the wrong questions. Instead, "micro" should refer to scope of responsibility, and the guiding principle here is the Unix philosophy of SRP: let it do one thing, and do it well.

If a service has only one single reason to exist, providing a single composable piece of functionality, business domains and responsibilities are not tangled. Each service can be made more generally useful, and the system as a whole is easier to scale, make resilient, understand, extend, and maintain.

Own Your State, Exclusively

> *Without privacy, there was no point in being an individual.*
> —Jonathan Franzen, *The Corrections*

Up to this point, we have characterized microservices as a set of isolated services, each one with a single area of responsibility. This scheme forms the basis for being able to treat each service as a single unit that lives and dies in isolation—a prerequisite for resilience—and can be moved around in isolation—a prerequisite for elasticity (in which a system can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs).

1 The Unix philosophy is described really well in the classic book *The Art of Unix Programming* by Eric Steven Raymond (Pearson Education).

2 For an in-depth discussion on the Single Responsibility Principle, see Robert C. Martin's website The Principles of Object Oriented Design.

Although this all sounds good, we are forgetting the elephant in the room: *state*.

Microservices are most often stateful components: they encapsulate state and behavior. Additionally, isolation most certainly applies to state and requires that you treat state and behavior as a single unit.

They need to *own their state, exclusively*.

This simple fact has huge implications. It means that data can be strongly consistent only *within* each service but never *between* services, for which we need to rely on eventual consistency and abandon transactional semantics. You must give up on the idea of a single database for all your data, normalized data, and joins across services (see Figure 1-1). This is a different world, one that requires a different way of thinking and the use of different designs and tools—something that we will discuss in depth later on in this report.

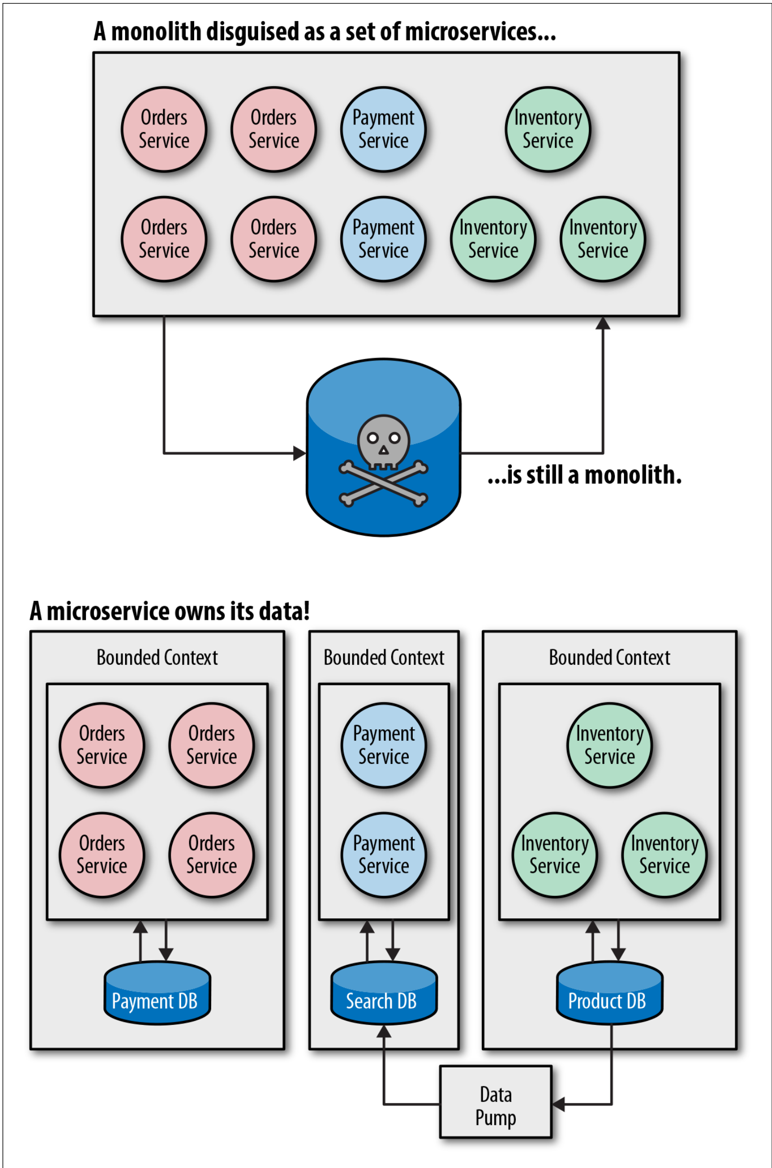


Figure 1-1. A monolith disguised as a set of microservices is still a monolith

Stay Mobile, but Addressable

> *To move, to breathe, to fly, to float, To gain all while you give, To roam the roads of lands remote, To travel is to live.*
>
> —H. C. Andersen

With the advent of cloud computing, virtualization, and Docker containers, we have a lot of power at our disposal to manage hardware resources efficiently. The problem is that none of these matter if our microservices and their underlying platform cannot make efficient use of them if they are statically locked into a specific topology or deployment scenario.

What we need are services that are *mobile*, allowing the system to be elastic and adapt dynamically to its usage patterns. Mobility is the possibility of moving services around at runtime while they are being used. This is needed for the services to stay oblivious to how the system is deployed and which topology it currently has—something that can (and often should) change dynamically.

Now that we have outlined the five essential traits of an individual microservice, we are ready to slay the monolith and put them to practice.

CHAPTER 2
Slaying the Monolith

> *Only with absolute fearlessness can we slay the dragons of mediocrity that invade our gardens.*
>
> —John Maynard Keynes

Before we take on the task of slaying the monolith, let's try to understand why its architecture is problematic, why we need to "slay the monolith" and move to a decoupled architecture using microservices.

Suppose that we have a monolithic Java Platform, Enterprise Edition (Java EE) application with a classic three-tier architecture that uses Servlets, Enterprise Java Beans (EJBs) or Spring, and Java Persistence API (JPA), and an Oracle SQL database. Figure 2-1 depicts this application.

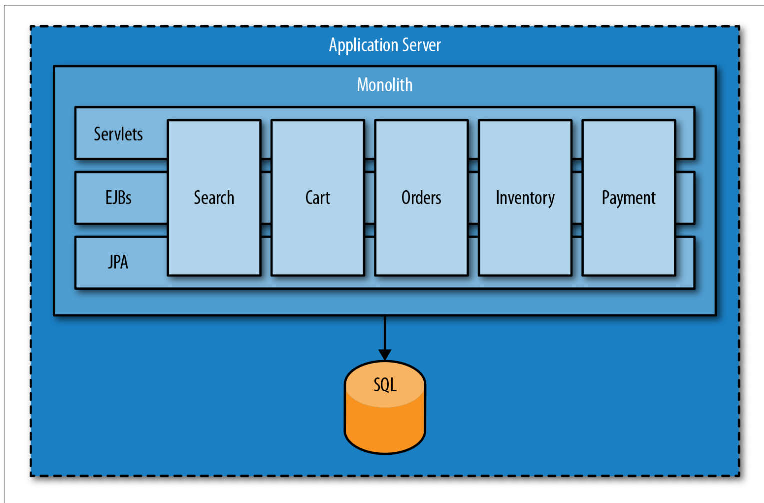


Figure 2-1. A monolithic application with a classic three-tier architecture

The problem with this design is that it introduces strong coupling between the components within each service and between services. Workflow logic based on deep nested call chains of synchronous method calls, following the thread of the request, leads to strong coupling and entanglement of the services, making it difficult to understand the system at large and to let services evolve independently. The caller is held hostage until the methods have executed all their logic.

Because all these services are tightly coupled, you need to upgrade all of them at once. Their strong coupling also makes it difficult to deal with failure in isolation. Exceptions—possibly blowing the entire call stack—paired with try/catch statements is a blunt tool for failure management. If one service fails, it can easily lead to cascading failures across all of the tiers, eventually taking down the entire application.

The lack of isolation between services also means that you can't scale each service individually. Even if you need to scale only one single service (due to high traffic or similar), you can't do that. Instead you must scale the whole monolith, including all of its other services. In the world of the monolith, it's always all or nothing, leading to lack of flexibility and inefficient use of resources.

Application servers (such as WebLogic, JBoss, Tomcat, etc.) encourage this monolithic model. They assume that you are bundling your service JARs into an EAR (or WAR) file as a way of grouping your services, which you then deploy—alongside all your other applications and services—into the single running instance of the application server. The application server then manages the service "isolation" through class loader magic. This is a fragile model, leaving services competing for resources like CPU time, main memory, and storage space, resulting in reduced fairness and stability as a result.

Don't Build Microliths

> ***microlith*** *(\ ˈmī-krə-ˌlith\):*
>
> —A very small stone tool made from a sharp blade-shaped piece of stone.

Suppose that we want to move away from the application server and the strongly coupled design and refactor this monolith into a microservices-based system. By just drinking the Kool-Aid, relying on a scaffolding tool, and following the path of least resistance, many people end up with an architecture similar to that shown in Figure 2-2.

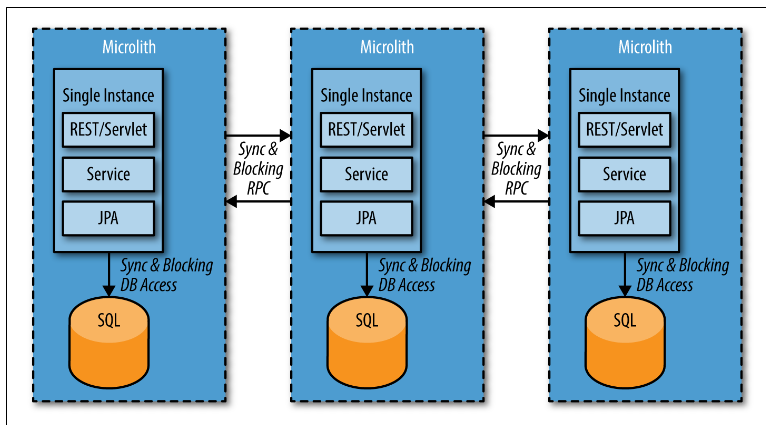


Figure 2-2. A system of microliths communicating over synchronous protocols

In this architecture, we have *single instance services* communicating over synchronous HTTP (often using RESTful[1] APIs), running in Docker containers, and using Create, Read, Update, and Delete (CRUD) through JPA talking to a—hopefully dedicated—SQL database (in the worst case, still using a single monolithic database, with a single, and highly normalized, schema for all services).

Well, what we have built ourselves is a set of *micro monoliths*—let's call them *microliths*.

A microlith is defined as a *single-instance service* in which synchronous method calls have been turned into synchronous REST calls and blocking database access remains blocking. This creates an architecture that is maintaining the strong coupling we wanted to move away from but with higher latency added by interprocess communication (IPC).

The problem with a single instance is that by definition it cannot be scalable or available. A single monolithic thing, whatever it might be (a human, or a software process), can't be scaled out and can't stay available if it fails or dies.

Some people might think, "Well, Docker will solve that for me." I'm sorry to say, but containers alone won't solve this problem. Merely putting your microservice instances in Docker or Linux (LXC) containers won't help you as much as you would like.

There's no question that containers and their orchestration managers, like Kubernetes or Docker Swarm, are great tools for managing and orchestrating hundreds of instances (with some level of isolation). But, when the dust settles, they have left you with the hard parts of distributed systems. Because microservices are not isolated islands and come in systems, the hardest parts are the space *in-between* the services, in things like communication, consensus, consistency, and coordination to state and resources. These are concerns that are a part of the application itself, not something that can be bolted on after the fact.

1 Nothing in the idea of REST itself requires synchronous communication, but it is almost exclusively used this way in the industry.

CHAPTER 3
Microservices Come in Systems

> *One actor is no actor. Actors come in systems.*
> —Carl Hewitt

In the spirit of famed computer scientist Carl Hewitt:[1] one microservice is no microservice. Microservices come in systems.

Like humans, microservices are autonomous and therefore need to communicate and collaborate with others to solve problems. And as with humans, it is in the collaboration with others that the most interesting opportunities and challenging problems arise.

What's difficult in microservices design is not creating the individual services themselves, but managing the space *between* the services. We need to dig deeper into the study of systems of services.

Embrace Uncertainty

> *What is not surrounded by uncertainty cannot be the truth.*
> —Richard Feynman

As soon as we exit the boundary of the single-service instance, we enter a wild ocean of nondeterminism—the world of *distributed systems*—in which systems fail in the most spectacular and intricate ways; where information becomes lost, reordered, and garbled; and where failure detection is a guessing game.

1 Carl Hewitt invented the Actor Model in 1973.

It sounds like a scary world.[2] But it is also the world that gives us solutions for resilience, elasticity, and isolation, among others. What we need is better tools to not just survive, but to thrive in the barren land of distributed systems.

We Are Always Looking into the Past

> *The contents of a message are always from the past! They are never "now."*
>
> —Pat Helland

When it comes to distributed systems, one constraint is that *communication has latency.*[3] It's a fact (quantum entanglement, wormholes, and other exotic phenomena aside) that information cannot travel faster than the speed of light, and most often travels considerably slower, which means that communication of information has latency.

In this case, exploiting reality means coming to terms with the fact that information is always from the past, and always represents another present, another view of the world (you are, for example, always seeing the sun as it was 8 minutes and 20 seconds ago). "Now" is in the eye of the beholder, and in a way, we are always looking into the past.

It's important to remember that reality is not *strongly consistent*,[4] but *eventually consistent*.[5] Everything is relative and there is no single "now."[6] Still, we are trying so hard to *maintain the illusion* of a single globally consistent present, a single global "now." This is no reason to be surprised. We humans are bad at thinking concurrently, and

2 It is—if you have not experienced this first-hand, I suggest that you spend some time thinking through the implications of L Peter Deutsch's Fallacies of Distributed Computing.

3 That fact that information has latency and that the speed of light represents a hard (and sometimes very frustrating) nonnegotiable limit on its maximum velocity is an obvious fact for anyone that is building internet systems, or who has been on a VOIP call across the Atlantic ocean.

4 Peter Bailis has a good explanation of the different flavors of strong consistency.

5 A good discussion on different client-side semantics of eventual consistency—including read-your-writes consistency and causal consistency—can be found in "Eventually Consistent—Revisited" by Werner Vogels.

6 Justin Sheehy's "There Is No Now" is a great read on the topic.

assuming full control over time, state, and causality makes it easier to understand complex behavior.

The Cost of Maintaining the Illusion of a Single Now

> *In a distributed system, you can know where the work is done or you can know when the work is done but you can't know both.*
>
> —Pat Helland

The cost of maintaining the illusion of a single global "now" is very high and can be defined in terms of *contention*—waiting for shared resources to become available—and *coherency*—the delay for data to become consistent.

Gene Amdahl's, now classic, Amdahl's Law explains the effect that contention has on a parallel system and shows that it puts a ceiling on scalability, yielding diminishing returns as more resources are added to the system.

However, it turns out that this is not the full picture. Neil Günter's Universal Scalability Law shows that when you add coherency to the picture, you can end up with negative results. And, adding more resources to the system makes things worse.

In addition, as latency becomes higher (as it does with distance), the illusion cracks. The difference between the local present and the remote past is even greater in a distributed system.

Learn to Enjoy the Silence

> *Words are very unnecessary. They can only do harm. Enjoy the silence.*
>
> —Martin Gore, *Enjoy the Silence*

Strong consistency requires coordination, which is very expensive in a distributed system and puts an upper bound on scalability, availability, low latency, and throughput. The need for coordination means that services can't make progress individually, because they must wait for consensus.

The cure is that we need to learn to enjoy the silence. When designing microservices, we should strive to minimize the service-to-service communication and coordination of state. We need to *learn to shut up*.

Avoid Needless Consistency

> *The first principle of successful scalability is to batter the consistency mechanisms down to a minimum.*
>
> —James Hamilton

To model reality, we need to rely on Eventual Consistency. But don't be surprised: it's how the world works. Again, we should not fight reality; we should embrace it! It makes life easier.

The term *ACID 2.0* was coined[7] by Pat Helland and is a summary of a set of principles for eventually consistent protocol design. The acronym is meant to somewhat challenge the traditional ACID from database systems:

- The "A" in the acronym stands for *Associative*, which means that grouping of messages does not matter and allows for batching.
- The "C" is for *Commutative*, which means that ordering of messages does not matter.
- The "I" stands for *Idempotent*, which means that duplication of messages does not matter.
- The "D" could stand for *Distributed*, but is probably included just to make the ACID acronym work.

There has been a lot of buzz about eventual consistency, and for good reason. It allows us to raise the ceiling on what can be done in terms of scalability, availability, and reduced coupling.

However, relying on eventual consistency is sometimes not permissible, because it can force us to give up too much of the high-level business semantics. If this is the case, using *causal consistency* can be a good trade-off. Semantics based on causality is what humans expect and find intuitive. The good news is that causal consistency can be made both scalable and available (and is even proven[8] to be the best we can do in an always available system).

7 Another excellent paper by Pat Helland, in which he introduced the idea of ACID 2.0, in "Building on Quicksand."

8 That causal consistency is the strongest consistency that we can achieve in an always available system was proved by Mahajan et al. in their influential paper "Consistency, Availability, and Convergence".

Causal consistency is usually implemented using logical time instead of synchronized clocks. The use of wall-clock time (timestamps) for state coordination is something that should most often be avoided in distributed system design due to the problems of coordinating clocks across nodes, clock skew, and so on. This is the reason why it is often better to rely on logical time, which gives you a stable notion of time that you can trust, even if nodes fail, messages drop, and so forth. There are several good options available, such as *vector clocks*,[9] or *Conflict-Free Replicated Data Types* (CRDTs).[10]

CRDTs is one of the most interesting ideas coming out of distributed systems research in recent years, giving us rich, eventually consistent, and composable data-structures—such as counters, maps, and sets—that are guaranteed to converge consistently without the need for explicit coordination. CRDTs don't fit all use cases, but is a very valuable tool[11] when building scalable and available systems of microservices.

Let's now look at three powerful tools for moving beyond microliths that can help you to manage the complexity of distributed systems while taking advantage of its opportunities for scalability and resilience:

- Events-First Domain-Driven Design
- Reactive Programming and Reactive Systems
- Event-Based Persistence

9 For good discussions on vector clocks, see the articles "Why Vector Clocks Are Easy" and "Why Vector Clocks Are Hard".

10 For more information, see Mark Shapiro's paper "A comprehensive study of Convergent and Commutative Replicated Data Types".

11 For a great production-grade library for CRDTs, see Akka Distributed Data.

CHAPTER 4

Events-First Domain-Driven Design

The term *Events-First Domain-Driven Design* was coined by Russ Miles, and is the name for set of design principles that has emerged in our industry over the last few years and has proven to be very useful in building distributed systems at scale. These principles help us to shift the focus from the nouns (the domain objects) to the verbs (the events) in the domain. A shift of focus gives us a greater starting point for understanding the essence of the domain from a data flow and communications perspective, and puts us on the path toward a scalable event-driven design.

Focus on What Happens: The Events

> *Here you go, Larry. You see what happens? You see what happens, Larry?!*
>
> —Walter Sobchak, Big Lebowski

Object-Oriented Programming (OOP) and later Domain-Driven Design (DDD) taught us that we should begin our design sessions focusing on the *things*—the *nouns*—in the domain, as a way of finding the Domain Objects, and then work from there. It turns out that this approach has a major flaw: it forces us to focus on *structure* too early.

Instead, we should turn our attention to the *things that happen*—the flow of *events*—in our domain. This forces us to understand how

change propagates in the system—things like communication patterns, workflow, figuring out who is talking to whom, who is responsible for what data, and so on. We need to model the business domain from a data dependency and communication perspective.

As Greg Young, who coined Command Query Responsibility Segregation (CQRS), says:

> *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.*
>
> *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.*

Modeling events and their causal relationships helps us to get a good grip on time itself, something that is extremely valuable when designing distributed systems.

Events Represent Facts

> *To condense fact from the vapor of nuance.*
>
> —Neal Stephenson, *Snow Crash*

Events represent facts about the domain and should be part of the Ubiquitous Language of the domain. They should be modelled as Domain Events and help us define the Bounded Contexts,[1] forming the boundaries for our service.

As Figure 4-1 illustrates, a bounded context is like a bulkhead: it prevents unnecessary complexity from leaking outside the contextual boundary, while allowing you to use a single and coherent domain model and domain language within.

1 For an in-depth discussion on how to design and use bounded contexts, read Vaughn Vernon's book *Implementing Domain-Driven Design* (Addison-Wesley).

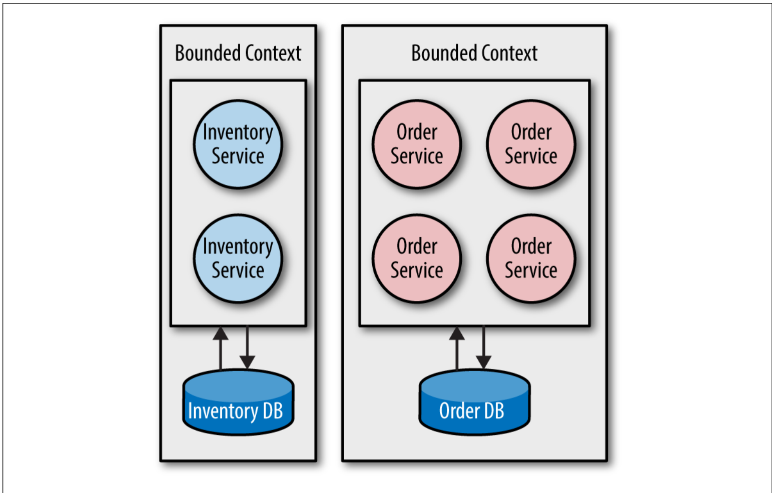


Figure 4-1. Let the bounded context define the service boundary

Commands represent an intent to perform some sort of action. These actions are often side-effecting, meaning they are meant to cause an effect on the receiving side, causing it to change its internal state, start processing a task, or send more commands.

A fact represents something that has happened in the past. It's defined by Merriam-Webster as follows:

> Something that truly exists or happens: something that has actual existence, a true piece of information.

Facts are immutable. They can't be changed or be retracted. We can't change the past, even if we sometimes wish that we could.

Knowledge is cumulative. This occurs either by receiving new facts, or by deriving new facts from existing facts. Invalidation of existing knowledge is done by adding new facts to the system that refute existing facts. Facts are not deleted, only made irrelevant for current knowledge.

Elementary, My Dear Watson

Just like Sherlock Holmes used to ask his assistant—Dr. Watson—when arriving to a new crime scene, ask yourself: "*What are the facts?*" Mine the facts.

Try to understand which facts are causally related and which are not. It's the path toward understanding the domain, and later the system itself.

A centralized approach to model causality of facts is *event logging* (discussed in detail shortly), whereas a decentralized approach is to rely on vector clocks or CRDTs.

Using Event Storming

> *When you come out of the storm, you won't be the same person who walked in.*
>
> —Haruki Murakami, *Kafka on the Shore*

A technique called *event storming*[2] can help us to mine the facts, understand how data flows, and its dependencies, all by distilling the essence of the domain through events and commands.

It's a design process in which you bring all of the stakeholders—the domain experts and the programmers—into a single room, where they brainstorm using Post-it notes, trying to find the domain language for the *events* and *commands*, exploring how they are causally related and the reactions they cause.

The process works something like this:

1. Explore the domain from the perspective of what happens in the system. This will help you find the events and understand how they are causally related.
2. Explore what triggers the events. They are often created as a consequence of executing the intent to perform a function, represented as a command. Here, among other attributes, we find user interactions, requests from other services, and external systems.
3. Explore where the commands end up. They are usually received by an aggregate (discussed below) that can choose to execute the side-effect and, if so, create an event representing the new fact introduced in the system.

---

2 An in-depth discussion on event storming is beyond the scope for this book, but a good starting point is Alberto Brandolini's upcoming book *Event Storming*.

Now we have solid process for distilling the domain, finding the commands and events, and understanding how data flows through the system. Let's now turn our attention to the aggregate, where the events end up—our source of truth.

Think in Terms of Consistency Boundaries

> *One of the biggest challenges in the transition to Service-Oriented Architectures is getting programmers to understand they have no choice but to understand both the "then" of data that has arrived from partner services, via the outside, and the "now" inside of the service itself.*
>
> —Pat Helland

I've found it useful to think and design in terms of *consistency boundaries*[3] for the services:

1. Resist the urge to begin with thinking about the *behavior* of a service.
2. Begin with the data—the facts—and think about how it is coupled and what dependencies it has.
3. Identify and model the integrity constraints and what needs to be guaranteed, from a domain- and business-specific view. Interviewing domain experts and stakeholders is essential in this process.
4. Begin with zero guarantees, for the smallest dataset possible. Then, add in the weakest level of guarantee that solves your problem while trying to keep the size of the dataset to a minimum.
5. Let the *Single Responsibility Principle* (discussed in "Single Responsibility" on page 2) be a guiding principle.

The goal is to try to minimize the dataset that needs to be *strongly consistent*. After you have defined the essential dataset for the service, *then* address the behavior and the protocols for exposing data through interacting with other services and systems—defining our *unit of consistency*.

3 Pat Helland's paper, "Data on the Outside versus Data on the Inside", talks about guidelines for designing consistency boundaries. It is essential reading for anyone building microservices-based systems.

Aggregates—Units of Consistency

> *Consistency is the true foundation of trust.*
> —Roy T. Bennett

The consistency boundary defines not only a *unit of consistency*, but a *unit of failure*. A unit that always fails atomically is upgraded atomically and relocated atomically.

If you are migrating from an existing monolith with a single database schema, you need to be prepared to apply *denormalization techniques* and break it up into multiple schemas.

Each unit of consistency should be designed as an aggregate.[4] An aggregate consists of one or many *entities*, with one of them serving as the *aggregate root*. The only way to reference the aggregate is through the aggregate root, which maintains the integrity and consistency of the aggregate as a whole.

It's important to always reference other aggregates by identity, using their primary key, and never through direct references to the instance itself. This maintains isolation and helps to minimize memory consumption by avoiding *eager loading*—allowing aggregates to be rehydrated on demand, as needed. Further, it allows for *location transparency*, something that we discuss in detail momentarily.

Aggregates that don't reference one another directly can be repartitioned and moved around in the cluster for *almost infinite scalability* —as outlined by Pat Helland in his influential paper "Life Beyond Distributed Transactions".[5]

Outside the aggregate's consistency boundary, we have no choice but to rely on *eventual consistency*. In his book *Implementing Domain-Driven Design* (Addison-Wesley), Vaughn Vernon suggests a rule of thumb in how to think about responsibility with respect to data consistency. You should ask yourself the question: "*Whose job is it to ensure data consistency?*" If the answer is that it's the service executing the business logic, confirm that it can be done within a single

4 For a good discussion on how to design with aggregates, see Vaughn Vernon's "Effective Aggregate Design".

5 You can find a good summary of the design principles for almost-infinite scalability here.

aggregate, to ensure strong consistency. If it is someone else's (user's, service's or system's) responsibility, make it eventually consistent.

Suppose that we need to understand how an order management system works. After a successful event storming session, we might end up with the following (drastically simplified) design:

- Commands: CreateOrder, SubmitPayment, ReserveProducts, ShipProducts
- Events: OrderCreated, ProductsReserved, PaymentApproved, PaymentDeclined, ProductsShipped
- Aggregates: Orders, Payments, Inventory

Figure 4-2 presents the flow of commands between a client and the services/aggregates (an open arrow indicates that the command or event was sent asynchronously).

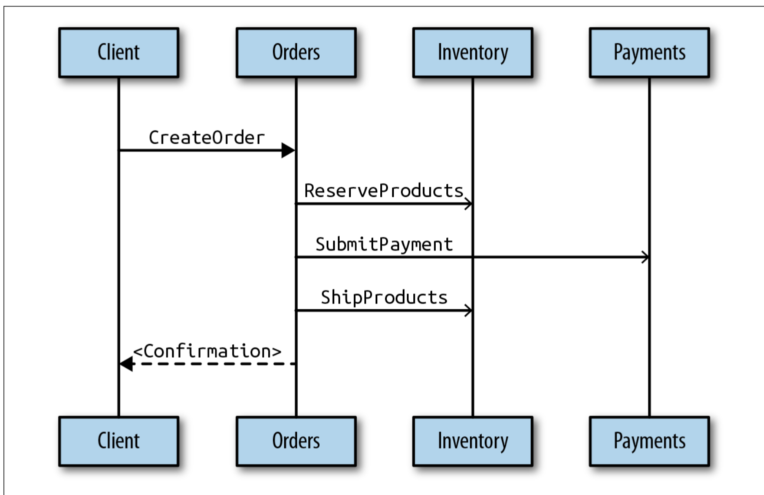


Figure 4-2. The flow of commands in the order management sample use case

If we add the events to the picture, it looks something like the flow of commands shown in Figure 4-3.

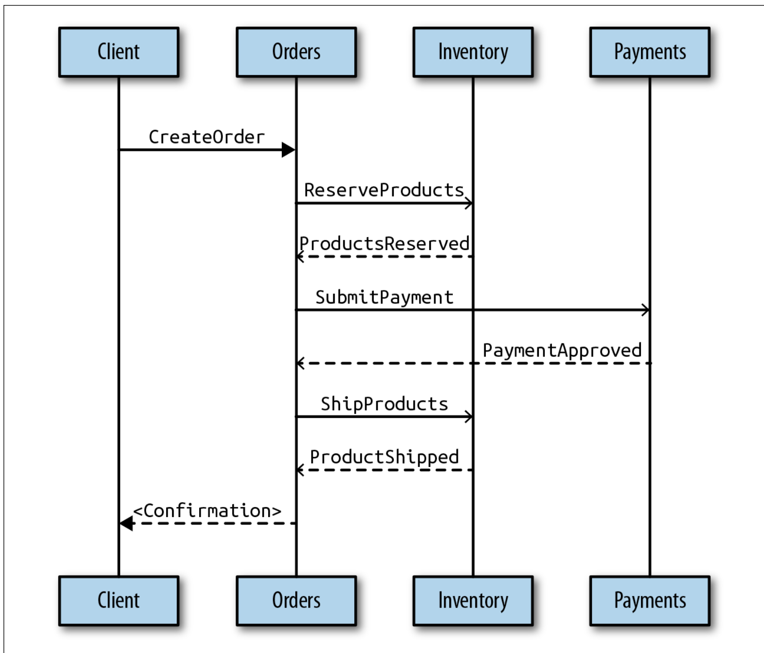


Figure 4-3. The flow of commands and events in the order management sample use case

Please note that this is only the conceptual flow of the events, how they flow between the services. An actual implementation will use subscriptions on the aggregate's event stream to coordinate workflow between multiple services (something we will discuss in depth later on in this report).

Contain Mutable State—Publish Facts

> *The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer's bottleneck does.*
>
> —John Backus (Turing Award lecture, 1977)

After this lengthy discussion about events and immutable facts you might be wondering if mutable state deserves a place at the table at all.

It's a fact that mutable state, often in the form of variables, can be problematic. One problem is that the assignment statement—as discussed by John Backus in his Turing Award lecture—is a destructive

operation, overwriting whatever data that was there before, and therefore resetting time, and resetting all history, over and over again.

The essence of the problem is that—as Rich Hickey, the inventor of the Clojure programming language, has discussed frequently—most object-oriented computer languages (like Java, C++, and C#) treat the concepts of *value* and *identity* as the same thing. This means that an identity can't be allowed to evolve without changing the value it currently represents.

Functional languages (such as Scala, Haskell, and OCaml), which rely on pure functions working with immutable data (values), address these problems and give us a solid foundation for reasoning about programs, a model in which we can rely on stable values that can't change while we are observing them.

So, is all mutable state evil? I don't think so. It's a convenience that has its place. But it *needs to be contained*, meaning mutable states should be used only for *local computations*, within the safe haven that the service instance represents, completely *unobservable* by the rest of the world. When you are done with the local processing and are ready to tell the world about your results, you then create an immutable *fact* representing the result and *publish* it to the world.

In this model, others can rely on stable values for their reasoning, whereas you can still benefit from the advantages of mutability (simplicity, algorithmic efficiency, etc.).

Manage Protocol Evolution

> *Be conservative in what you do, be liberal in what you accept from others.*
>
> —Jon Postel

Individual microservices are only independent and decoupled if they can evolve independently. This requires protocols to be resilient to and permissive of change—including events and commands, persistently stored data, as well as the exchange of ephemeral information.[6] The interoperability of different versions is crucial to enable the long-term management of complex service landscapes.

6 For example, session state, credentials for authentication, cached data, and so on.

Postel's Law,[7] also known as the *Robustness Principle*, states that you should "*be conservative in what you do, be liberal in what you accept from others*," and is a good guiding principle in API design and evolution for collaborative services.[8]

Challenges include versioning of the protocol and data—the events and commands—and how to handle upgrades and downgrades of the protocol and data. This is a nontrivial problem that includes the following:

- Picking extensible codecs for serialization
- Verifying that incoming commands are valid
- Maintaining a protocol and data translation layer that might need to upgrade or downgrade events or commands to the current version[9]
- Sometimes even versioning the service itself[10]

These functions are best performed by an Anti-Corruption Layer, and can be added to the service itself or done in an API Gateway. The Anti-Corruption Layer can help make the bounded context robust in the face of changes made to another bounded context, while allowing them and their protocols to evolve independently.

7 Originally stated by Jon Postel in "RFC 761" on TCP in 1980.

8 It has, among other things, influenced the Tolerant Reader Pattern.

9 For an in-depth discussion about the art of event versioning, I recommend Greg Young's book *Versioning in an Event Sourced System*.

10 There is a semantic difference between a service that is truly new, compared to a new version of an existing service.

CHAPTER 5
Toward Reactive Microsystems

Ever since I helped coauthor the Reactive Manifesto in 2013, Reactive has gone from being a virtually unknown technique for constructing systems—used by only fringe projects within a select few corporations—to become part of the overall platform strategy in numerous big players in the industry. During this time, Reactive has become an overloaded word, meaning different things to different people. More specifically there has been some confusion around the difference between "Reactive Programming" and "Reactive Systems" (a topic covered in depth in this O'Reilly article, "Reactive programming vs. Reactive systems").

Reactive *Programming* is a great technique for making individual components performant and efficient through asynchronous and nonblocking execution, most often together with a mechanism for backpressure. It has a *local focus* and is *event-driven*—publishing facts to 0–*N* anonymous subscribers. Popular libraries for Reactive Programming on the Java Virtual Machine (JVM) include Akka Streams, Reactor, Vert.x, and RxJava.

Reactive *Systems* takes a holistic view on system design, focusing on keeping *distributed systems* responsive by making them *resilient and elastic*. It is *message-driven*—based upon asynchronous message-passing, which makes distributed communication to addressable recipients first class—allowing for elasticity, location transparency, isolation, supervision, and self-healing.

Both are equally important to understand how, and when, to apply when designing microservices-based systems. Let's now dive into

both techniques to see how we can use them on different levels throughout our design.

Embrace Reactive Programming

Reactive Programming is essential to the design of microservices, allowing us to build highly efficient, responsive, and stable services. Techniques for Reactive Programming that we will discuss in depth include asynchronous execution and I/O, back-pressured streaming, and circuit breakers.

Go Asynchronous

Asynchronous and nonblocking I/O is about not blocking threads of execution—a process should not hold a thread hostage, hogging resources that it does not use. It can help eliminate the biggest threat to scalability: *contention*.[1]

Asynchronous and nonblocking execution and I/O is often more cost-efficient through more efficient use of resources. It helps minimize contention (congestion) on shared resources in the system, which is one of the biggest hurdles to scalability, low latency, and high throughput.

As an example, let's take a service that needs to make 10 requests to 10 other services and compose their responses. Suppose that each request takes 100 milliseconds. If it needs to execute these in a synchronous sequential fashion, the total processing time will be roughly 1 second, as demonstrated in Figure 5-1.

1 Like Gene Amdahl, who coined Amdahl's Law, has shown us.

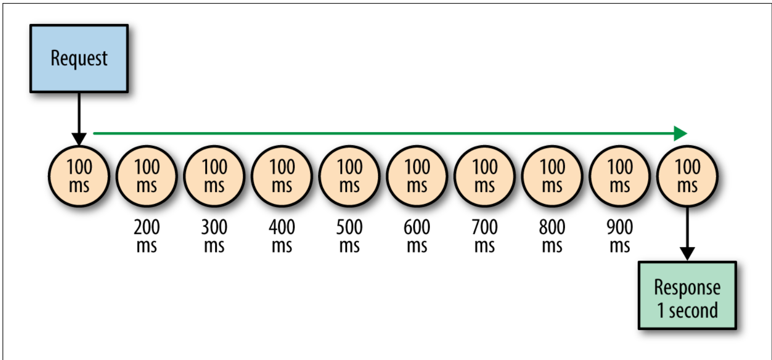


Figure 5-1. Sequential execution of tasks with each request taking 100 milliseconds

Whereas, if it is able to execute them all asynchronously, the total processing time will just be 100 milliseconds, as shown in Figure 5-2.

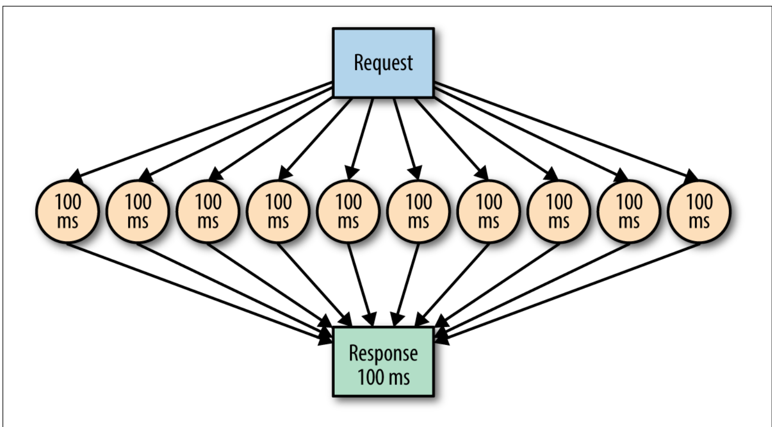


Figure 5-2. Parallel execution of tasks—an order of magnitude difference for the client that made the initial request

But why is blocking so bad?

If a service makes a blocking call to another service—waiting for the result to be returned—it holds the underlying thread hostage. This means no useful work can be done by the thread during this period. Threads are a scarce resource and need to be used as efficiently as

possible.[2] If the service instead performs the call in an asynchronous and nonblocking fashion, it frees up the underlying thread so that someone else can use it while the first service waits for the result to be returned. This leads to much more efficient usage in terms of cost, energy, and performance of the underlying resources, as Figure 5-3 depicts.

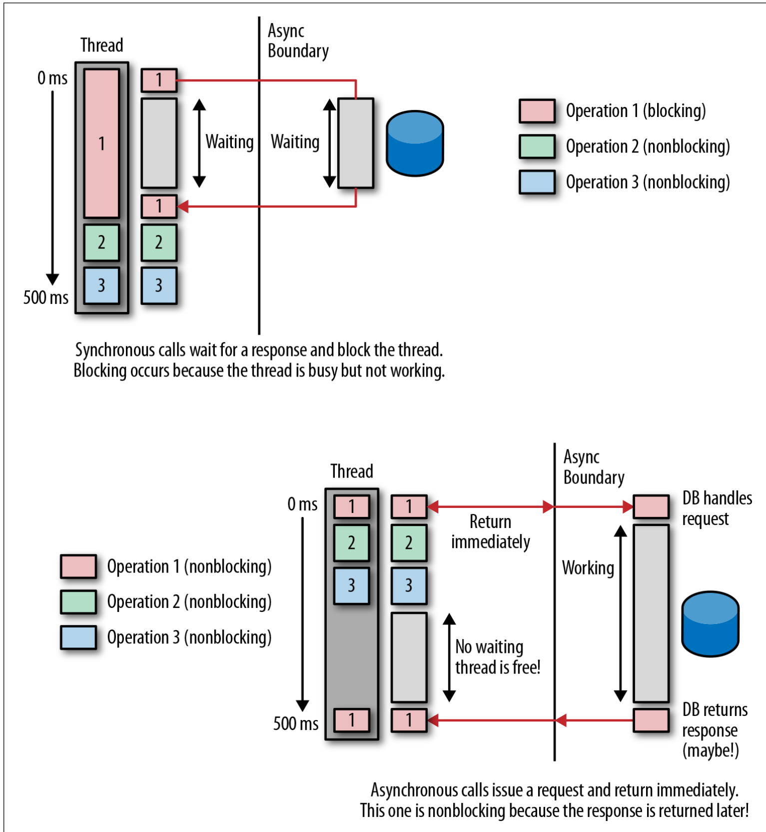


Figure 5-3. The difference between blocking and nonblocking execution

It is also worth pointing out that embracing asynchronicity is as important when communicating with different resources within a service boundary as it is between services. To reap the full benefits

2 With more threads comes more context switching, which is very costly. For more information on this, go to the "How long does it take to make a context switch?" blog post on Tsuna's blog.

of nonblocking execution, all parts in a request chain need to participate—from the request dispatch, through the service implementation, down to the database, and back.

Reconsider the Use of Synchronous HTTP

It is unfortunate that synchronous HTTP (most often using REST) is widely considered as the "state of the art" microservice communication protocol. Its synchronous nature introduces strong coupling between the services making it a *very bad default protocol* for interservice communication. Asynchronous messaging makes a much better default for communication between microservices (or any set of distributed components, for that matter).

If you need to use REST over HTTP, which could be a good option in certain cases, always use it outside the regular communication flow so as not to block the regular request-response cycle.

Always Apply Backpressure

Alongside going asynchronous, you should always apply backpressure. Backpressure is all about flow control, ensuring that a fast producer should not be able to overwhelm a slower consumer by being allowed to send it more data than it can handle, as shown in Figure 5-4.

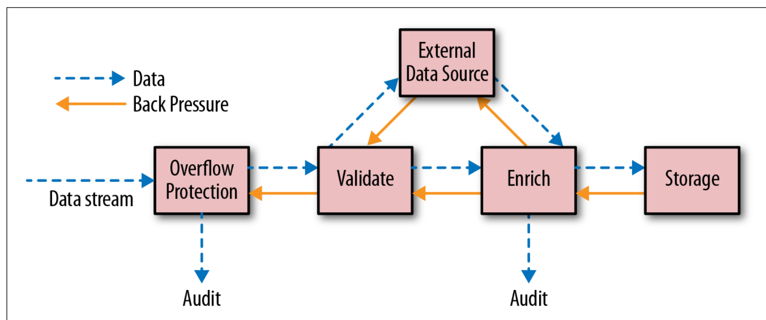


Figure 5-4. Backpressure includes having a backchannel where the consumer can manage control flow by communicating with the producer

Backpressure increases the reliability of not just the individual components, but also the data pipeline and system as a whole. It is usually achieved by having a backchannel going upstream in which downstream components can signal hints declaring whether the rate

of events should be slowed down or sped up. It is paramount that all of the parties in the workflow/data pipeline participate and speak the same protocol, which was the reason for the attempt to standardize a protocol for backpressure in the Reactive Streams specification.[3]

At Someone Else's Mercy

> *An escalator can never break: it can only become stairs. You should never see an "Escalator Temporarily Out of Order" sign, just "Escalator Temporarily Stairs. Sorry for the convenience."*
>
> —Mitch Hedberg

We can't bend the world to our will, so sometimes we find ourselves at the mercy of another system, such that, if it crashes or overload us with requests—and won't participate in backpressure protocols—it will take us down with it.

The problem is that not all third-party services, external systems, infrastructure tools, or databases will always play along and expose asynchronous and nonblocking APIs or protocols. This can put us in a dangerous situation in which we are forced to use blocking protocols or run the risk of being overloaded with more data than we can handle. If this happens, we need to protect ourselves, and a great way of doing that is to wrap the dangerous call in a "circuit breaker" in order to manage potential failures or usage spikes gracefully.

A circuit breaker is a finite-state machine (FSM), which means that it has a finite set of states: *Closed*, *Open*, and *Half-Open*. The default state is *Closed*, which allows all requests to go through.

When a failure (or a specific number of failures) have been detected, the circuit breaker "trips" and moves to an *Open* state. In this state, it does not let any requests through, and instead fails fast to shield the component from the failed service. Some implementations allow you to register a fallback implementation to be used when in the *Open* state to allow for graceful degradation.

After a timeout has occurred, there is a likelihood that the service is back up again, so it attempts to "reset" itself and move to a *Half-*

3 Started by Lightbend in 2014, which has, together with Netflix, Pivotal, Oracle, and Twitter, created a standard for backpressure on the JVM, now staged for inclusion in Java 9 as the Flow API.

Open state. Now, if the next request fails, it moves back to Open, and resets the timeout. But, if it succeeds, things are back in business, and it moves to the *Closed* state, as demonstrated in Figure 5-5.

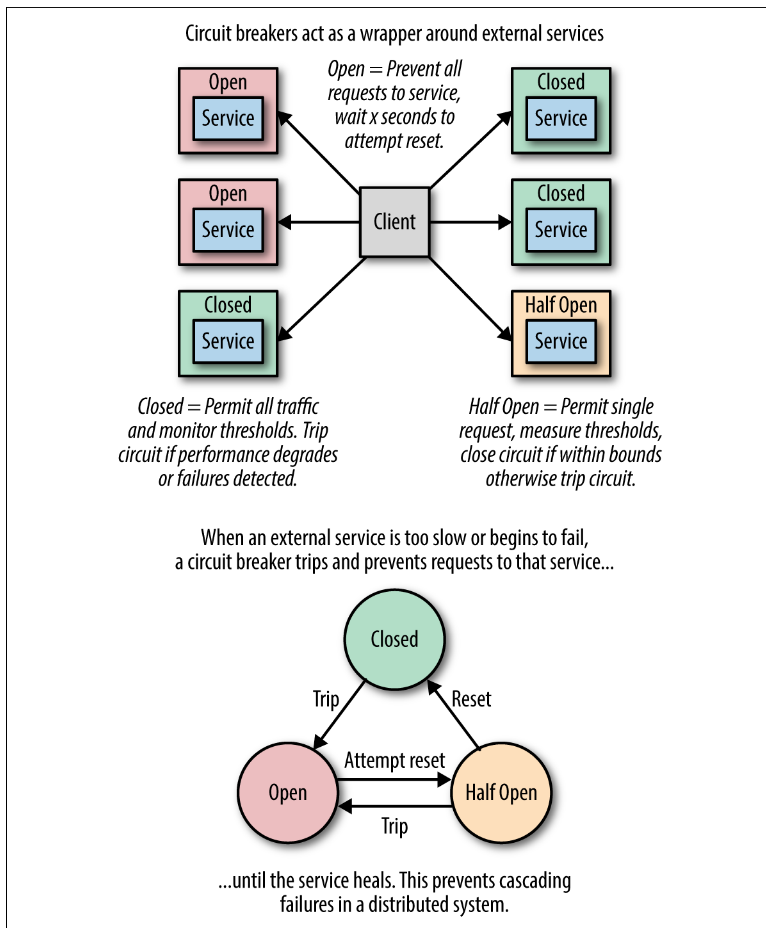


Figure 5-5. Circuit breakers can help improve the resilience of the service

Circuit breakers exist in most microservices frameworks and platforms; for example, in Akka, Lagom, and Hystrix.

Toward Reactive Microliths

Let's go back to our microliths to see if we can improve the design—making the services more efficient, performant, and stable—by applying the techniques of Reactive Programming.

The first thing we can do is to begin at the at the edge of the service and replace the synchronous and blocking communication protocol over REST with asynchronous alternatives.

One good alternative is to move to an event-driven model using Pub/Sub messaging over a message broker such as Apache Kafka or Amazon Web Services (AWS) Kinesis. This helps to decouple the services by introducing temporal decoupling—the services communicating do not need to be available at the same time—which increases resilience and makes it possible to scale them independently.

The world is going streaming, and whereas data used to be at rest, residing offline in SQL databases and queried on demand, nowadays we operate on data in motion. Applications today need to react to changes in data as it arrives, rather than batching it and delaying consumption until the batch is full. They need to perform continuous queries or aggregations of inbound data and feed it back into the application to affect the way it is operating. Microservices need to embrace streaming as a fundamental trait in order to be able to stream data between services, between user and application, and participate—as endpoints—in fast data/streaming pipelines.

Another problem with the microlith is the synchronous, and blocking, database access layer. A chain is not stronger than its weakest link, and blocking every request on the database communication introduces a potential Single Point Of Failure (SPOF) and Bottleneck—making the service vulnerable to failures and decreasing its scalability and efficiency due to the increased contention. We can address this by applying Reactive Programming techniques to the database access layer by wrapping it in a custom written asynchronous code[4] or, better, by relying on an asynchronous and nonblocking database driver.[5]

4 For an example of an asynchronous JDBC wrapper, see Peter Lawrey's post, "A JDBC Gateway Microservice".

5 Most NoSQL databases have asynchronous drivers. In the SQL world, you can turn to Postgres or MySQL.

Let's now apply Reactive Programming to our microliths (see Figure 5-6) by adding support for messaging and streaming, relying on asynchronous and nonblocking database access, and including circuit breakers for dangerous dependencies—turning them into a set of "Reactive Microliths."

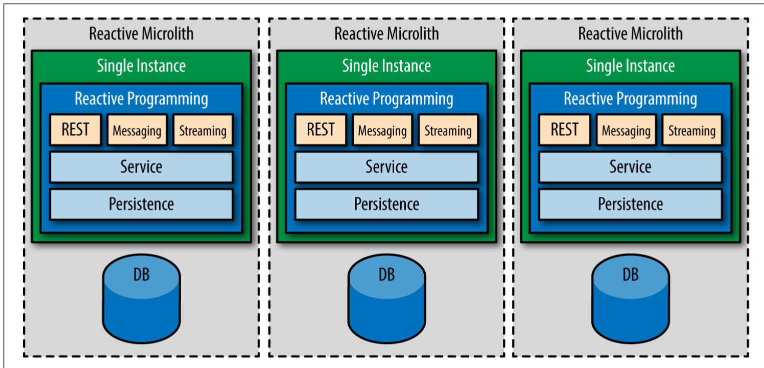


Figure 5-6. The system, evolved into a set of reactive microliths

We're getting there, but we still have a set of single-instance microservices—they are neither resilient nor elastic. This is something that we need to address.

Embrace Reactive Systems

> *Smalltalk is not only NOT its syntax or the class library, it is not even about classes. I'm sorry that I long ago coined the term "objects" for this topic because it gets many people to focus on the lesser idea. The big idea is 'messaging'.*
>
> —Alan Kay

Reactive Systems—as defined by the Reactive Manifesto—is a set of architectural design principles (see Figure 5-7) for building modern distributed systems that are well prepared to meet the demands of responsiveness under failure (resilience) and under load (elasticity) that applications need today.

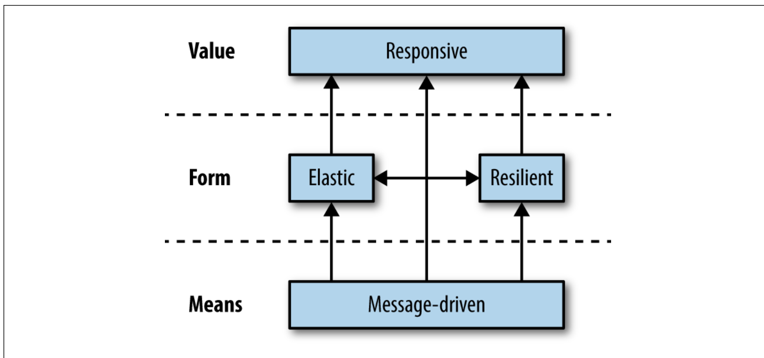


Figure 5-7. The four traits of a reactive system

Decoupling in Time and Space

> *Time and space are modes by which we think, and not conditions in which we live.*
>
> —Albert Einstein

The foundation for a Reactive System is asynchronous message-passing, which helps you to build loosely coupled systems with autonomous and collaborative components. Having an asynchronous boundary between components is necessary in order to decouple them and their communication flow in *time* (which allows for concurrency) and *space* (which allows for distribution and mobility).

Elasticity through Location Transparency

> *Scalability is possibly only if project elements do not form transformative relationships that might change the project.*
>
> —Anna Tsing, "On Nonscalability"

Traditionally, we are used to relying on different tools and techniques, with different semantics, across different levels of scale—for example, using callbacks and interleaving within a single core (as in Node.js), using threads, locks, and concurrent data structures (as in standard Java concurrency) when running on multiple cores, and relying on message queues or Remote Procedure Call (RPC) protocols (such as Java Message Service [JMS] or Remote Method Invocation [RMI]) for communication across nodes and datacenters. Making all of these paradigms work together in concert, as a single whole, is full of semantic mismatches.

Asynchronous message-passing allows for location transparency, which gives you one communication abstraction, across all dimensions of scale—up/down and in/out. One programming model, one API, with a single set of semantics, regardless of how the system is deployed or what topology it currently has is an example of this.[6]

This is where the beauty of asynchronous message-passing comes in, because it unifies them all, making communication explicit and first class in the programming model. Instead of hiding it behind a leaky abstraction[7], as is done in RPC,[8] EJBs, CORBA, distributed transactions, and so on.

Location transparency should not be confused with transparent remoting, or distributed objects (which Martin Fowler claimed *"sucks like an inverted hurricane"*). Transparent remoting hides the network, and tries to make all communication look like its local. The problem with this approach, even though it might sound compelling at first, is that local and distributed communication have vastly different semantics and failure modes.[9] Reaching for it only sets us up for failure. Location transparency does the opposite, by embracing the constraints of distributed systems.

Another benefit of asynchronous message-passing is that it tends to shift focus, from low-level plumbing and semantics, to the workflow and communication patterns in the system and forces you to think in terms of collaboration—how data flows between the different services, their protocols, and interaction patterns.

Location Transparency Enables Mobility

> *But I'll take my time anywhere. I'm free to speak my mind anywhere. And I'll redefine anywhere. Anywhere I roam. Where I lay my head is home.*
>
> —Lars Ulrich, James Hetfield, *Wherever I May Roam*

6 Elasticity presupposes dynamicity in terms of scaling.

7 As brilliantly explained by Joel Spolsky in his classic piece "The Law of Leaky Abstractions".

8 The fallacies of RPC have not been better explained than in Steve Vinoski's "Convenience over Correctness".

9 As explained by Jim Waldo et al., in their classic paper "A Note on Distributed Computing".

For a service to become location transparent, it needs to be addressable separately from "current" location. So, what does that really mean?

First, addresses need to be stable in the sense that they can be used to refer to the service indefinitely, regardless of where it is currently located. This should hold true whether the service is running, has been stopped, is suspended, is being upgraded, has crashed, and so on. The address should always work. A client should always be able to send messages to the address. In practice, the messages might sometimes be queued up, resubmitted, delegated, logged, or sent to a dead letter queue (Figure 5-8).

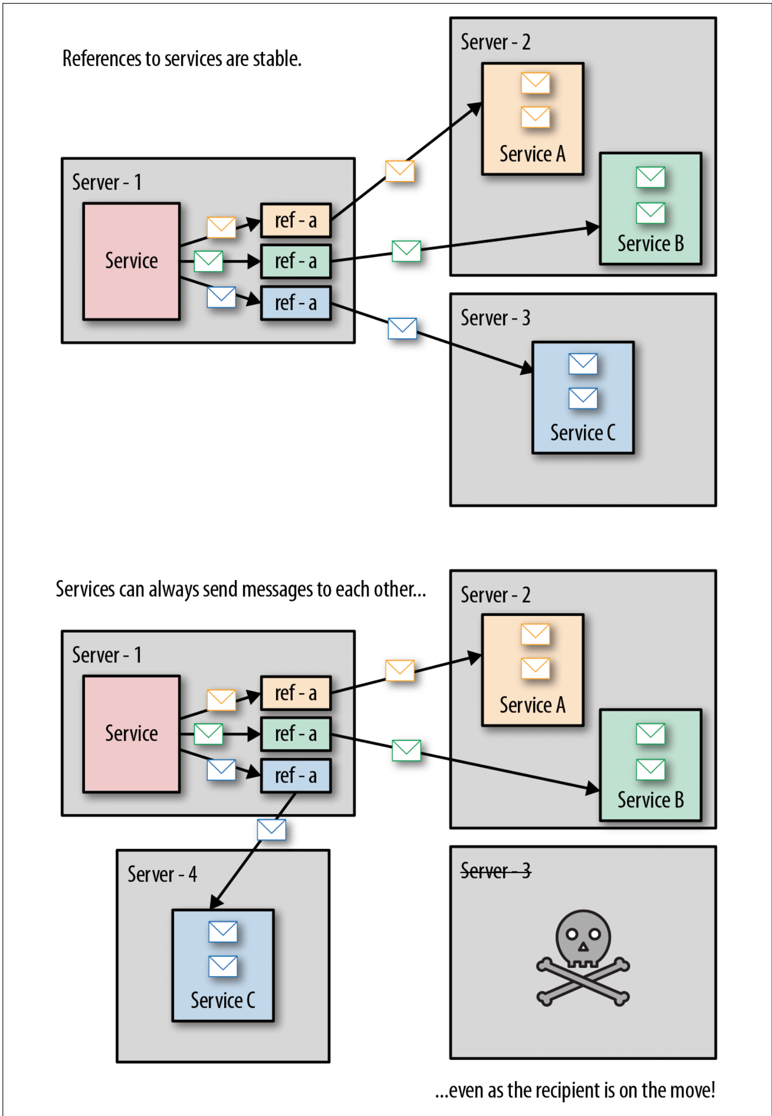


Figure 5-8. Services should be referenced through virtual stable references

Second, an address needs to be virtual in the sense that it can, and often does, represent not just one, but an entire set of runtime instances that together defines the service. Here are some of the reasons why this can be advantageous:

Load-balancing between instances of a stateless service
If a service is stateless, it does not matter to which instance a particular request is sent. Additionally, a wide variety of routing algorithms can be employed, such as round-robin, broadcast or metrics-based.

Active-Passive[10] state replication between instances of a stateful service
If a service is stateful, sticky routing needs to be used—sending every request to a particular instance. This scheme also requires each state change to be made available to the passive instances of the service (the replicas) each one ready to take over serving the requests in case of failover.

Relocation of a service
It can be beneficial to move a service instance from one location to another in order to improve *locality of reference*[11] or resource efficiency.

Using virtual addressing, through stable references, means that the client can stay oblivious to all of these low-level runtime concerns. It communicates with a service through an address and does not need to care about how and where the service is currently configured to operate.

These are all essential pieces on the path toward elasticity—a system that can react to load, react to current usage patterns, and scale its components adaptively in order to reach its Service-Level Objectives (SLOs).

Location Transparency Enables Dynamic Composition

> *Dynamics always trump semantics.*
> —Mark Burgess, *In Search of Certainty*

Another benefit of location transparency and communicating through stable references to services is that it allows for *dynamic composition*. The references themselves can be included in messages

10 Sometimes referred to as "Master-Slave," "Executor-Worker," or "Master-Minion" replication.

11 Locality of reference is an important technique in building highly performant systems. There are two types of reference locality: temporal (reuse specific data) and spatial (keep data relatively close in space). It is important to understand and optimize for both.

and passed around between services. This means that the topology of the system can change dynamically, at runtime, opening up for scaling the system in any dimension.

For one service to communicate with another service, it needs to know the other service's address. The simplest solutions would be to hardcode the physical address and port of all the services that a service needs to use, or have them externalized into a configuration file provided at startup time.

The problem with these solutions is that they force a static deployment model, which contradicts everything we are trying to accomplish with microservices. Services need to stay decoupled and mobile, and the system needs to be elastic and dynamic.

We can address this by adding a level of indirection[12] using a pattern called Inversion of Control (IoC), and taking advantage of location transparency.

What this means in practice is that each service should report information to the platform about where it is currently running and how it can be contacted. This is called Service Discovery and is an essential part of any microservices architecture.

After the information about each service has been stored, it can be made available through a Service Registry that services can use to look up the information using a pattern called Client-Side Service Discovery.

Another strategy is to have the information stored and maintained in a load balancer (as done in AWS Elastic Load Balancer) or directly in the address references that the services use (as done in Akka Actors) using a pattern called Server-Side Service Discovery.

Self-Healing Through Bulkheading and Supervision

> *Failure is simply the opportunity to begin again. This time more intelligently.*
>
> —Henry Ford

12 Inspired by David Wheeler, who once said, "All problems in computer science can be solved by another level of indirection."

The boundary, or bulkhead, between components/services that asynchronous messaging enables and facilitates is the foundation for resilient and self-healing systems, enabling the loose coupling needed for full isolation between them, as demonstrated in Figure 5-9.

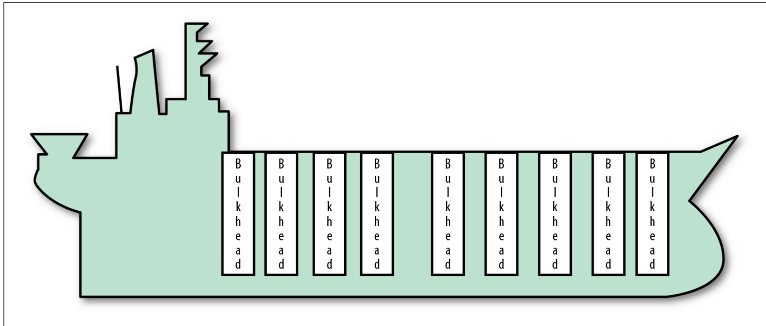


Figure 5-9. Ships use bulkheads for resilience

Bulkheading is most well known for being used in the ship construction industry as a way to divide the ship into isolated, watertight compartments. If a few compartments fill with water, the leak is contained and the ship can continue to function.

The same thinking and technique can be applied successfully to software. It can help us to arrive at a design that prevents failures from propagating outside the failed component, avoiding cascading failures taking down an entire system.

Some people might come to think of the Titanic as a counterexample. It is actually an interesting study[13] in what happens when you *don't* have proper isolation between the compartments, and how that can lead to cascading failures, eventually taking down the entire system. The Titanic did use bulkheads, but the walls that were supposed to isolate the compartments did not reach all the way up to the ceiling. So, when 6 of its 16 compartments were ripped open by the iceberg, the ship began to tilt and water spilled over the bulkheads from one compartment to the next, eventually filling up all compartments and sinking the Titanic, killing 1,500 people.

13 For an in-depth analysis of what made Titanic sink, see the article "Causes and Effects of the Rapid Sinking of the Titanic".

Bulkheading complements *Supervision*,[14] which is a general pattern for managing failure that has been used successfully in Actor languages (like Erlang—which invented it[15]) and libraries (like Akka) and is an important tool in distributed systems design. Components are organized into *Supervisor Hierarchies*, in which parent components supervise and manage the life cycle of its subordinate components. A subordinate never tries to manage its own failure; it simply crashes on failure and delegates the failure management to its parent by notifying it of the crash. This model is sometimes referred to as "Let it Crash" or "Crash-Only Software."[16] The supervisor then can choose the appropriate failure management strategy. For example, resume or restart the subordinate, delegate its work, or escalate the failure to its supervisor.

Figure 5-10 presents an example of a supervisor hierarchy. In this scenario, one child component fails—by raising an exception. The exception is captured by the component itself and reified as a failure message that is sent asynchronously to its supervisor. Upon reception of the message, the supervisor now can decide what to do with the failure. In this example, it decides that it is beyond its responsibilities to deal with this particular failure, and escalates it up the hierarchy, by sending it to its supervisor. The top-level supervisor now decides to restart the entire chain of failed components by sending it a restart command, recursively, bringing the whole component hierarchy back to a healthy state, ready to take on new tasks. This is self-healing in action.

14 For a detailed discussion on this pattern, see the Akka documentation on "Supervision and Monitoring".

15 Joe Armstrong's thesis "Making reliable distributed systems in the presence of software errors" is essential reading on the subject. According to Armstrong, Mike Williams at Ericsson Labs came up with the idea of "links" between processes, as a way of monitoring process health and lifecycle, forming the foundation for process supervision.

16 A couple of great, and highly influential, papers on this topic are "Crash Only Software" and "Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel", both by George Candea and Armando Fox.

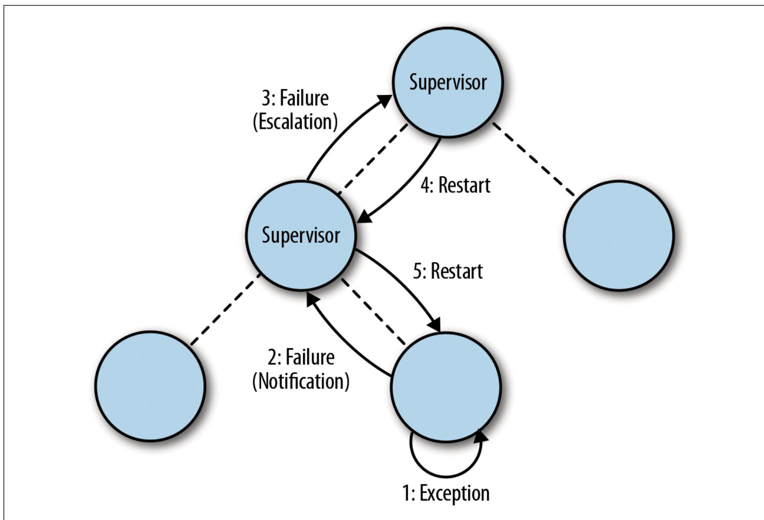


Figure 5-10. Supervisor hierarchies enable self-healing systems

With supervision hierarchies, we can design systems with autonomous components that watch out for one another and can recover failures by restarting the failed component(s).

Paired with location transparency—given that failures are nothing but regular messages flowing between the components—we can scale-out the failure management model across a cluster of nodes, while preserving the same semantics and simplicity of the programming model.

Microservices Come as Systems

> *The Japanese have a small word—ma—for "that which is in between"—perhaps the nearest English equivalent is "interstitial." The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.*
>
> —Alan Kay

We have discussed that microservices come *in systems*. However, what is less obvious is that microservices also come *as systems*. Each microservice needs to be designed as a system, a distributed one that needs to work together as a single whole.

We need to move from microliths to *microsystems*.

But, how? Let's begin by looking at the very different requirements for scaling stateful and stateless components and how we can take advantage of that in our first steps toward building scalable micro-systems.

Scaling State and Behavior Independently

It can be helpful to separate the *stateless processing* part of the service —the business logic—from the *stateful aggregate*—the system of record. This allows us to decouple them, run them on different nodes, or in-process on the same node if that is preferred, giving us the option to tune the availability through scaling.

Separating behavior and state into different processes, potentially running on different nodes, means that we cannot continue using local method dispatch for coordination. Instead, we need to coordinate using commands, sent as messages, in an asynchronous fashion.

Let's try that by taking the reactive microlith in Figure 5-6 and splitting off its stateless processing part from its stateful aggregate part (Figure 5-11).

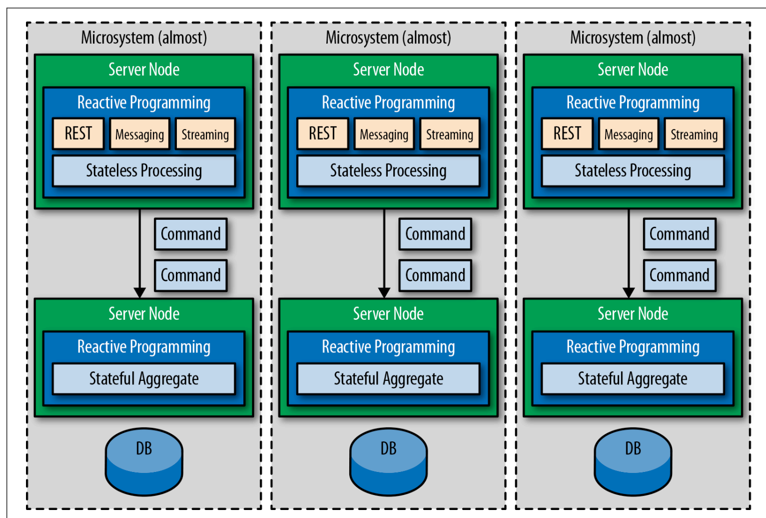


Figure 5-11. Separating the stateless processing from the stateful aggregate

Scaling stateless processing is trivial. It can be scaled linearly—assuming nonshared infrastructure—in a fully automatic fashion.

One example of this is AWS Lambda—kicking off the "serverless" trend—which scales stateless functions automatically within the boundaries of the SLOs that have been defined, without human intervention.

The processing instance is usually where we put translation logic, translating between different protocols or between different versions of the same protocol. It can function as the service's Anti-Corruption Layer, protecting the Bounded Context (the service itself) from the messiness of the world outside its boundary.

Scaling state, however, is very difficult. For services that do not have the luxury of being "forgetful" and need to read/write state, delegating to a database only moves the problem to the database. We still need to deal with the coordination of reading and writing. Ignoring the problem by calling the architecture "stateless" with "stateless" services, effectively pushing them down into a shared database, will only delegate the problem, push it down the stack, and make it more difficult to control in terms of data integrity, scalability, and availability guarantees.[17]

The aggregate instance is the single, strongly consistent, source of truth for the service and therefore can be allowed to run only a single active instance at a time.[18] It is usually deployed in an active-passive clustering scheme with a set of replicas ready to take over on failure.

Enter Reactive Microsystems

Now, we are in a position to scale-out the stateless processing part of our microservice to *N* number of nodes, and set up the stateful aggregate part in an active-passive manner for seamless failover.

We can do this in different ways. For example, in the Lagom microservices framework, each service is backed by an Akka Cluster,[19]

---

17 As always, it depends. A stateless architecture could work fine for sharded workloads, using consistent hashing algorithms, and a highly available database like Cassandra.

18 There are ways to support multiple concurrently active aggregates. As an example, we are currently building out a solution for Akka Persistence that is based on operation-based CRDTs and vector clocks.

19 Akka Cluster is a decentralized, node ring–based, cluster management library for Akka that is using epidemic gossip protocols, and failure detection, in a similar fashion as Amazon's Dynamo.

which will, among other things, ensure that all the service's aggregates are spread out on the available nodes and restarted on a healthy node upon node failure.

What we end up with is a system of (potentially) distributed parts that all need to work in concert, while being scaled alongside different axes—a *microsystem* (see Figure 5-12).

This design not only gives us increased scalability, but also resilience, by ensuring that there is always another node that can serve requests and take over processing or state management in the case of node failure.

To reap the full benefits of this design we need to apply the principles of Reactive Systems across the board. All components within the microsystem are part of a larger distributed system and need to communicate over asynchronous message-passing. The components need to be made mobile and location transparent, allowing the service to be elastic from within, and they need to be fully isolated and supervised by failure detection schemes to allow the microsystem as a whole to self-heal and stay resilient.

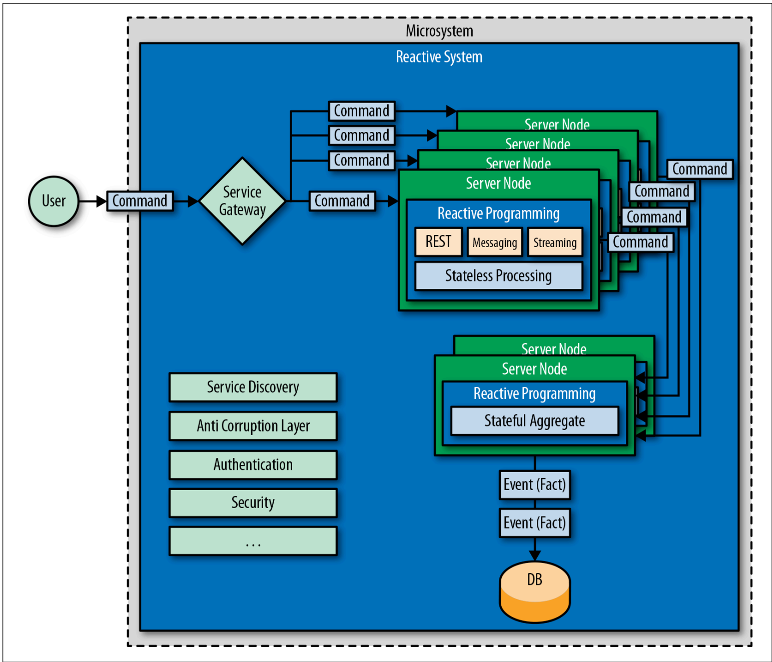


Figure 5-12. A scalable and resilient microservice consists of multiple distributed components—a microsystem

Figure 5-12 shows that we have added core infrastructure services like service discovery, anticorruption layer, authentication, security, and so on to the system. It is paramount to see them as part of the distributed system of the service—the microsystem—itself, to be able to take their constraints in terms of failure modes, availability, and scalability guarantees into account, instead of being surprised by them (most often at the most inappropriate time).

CHAPTER 6

Toward Scalable Persistence

> *Anything you build on a large scale or with intense passion invites chaos.*
>
> —Francis Ford Coppola

Up to this point, we have discussed the evolution of a monolith, through a system of microliths, to a design based on scalable and resilient microsystems. But, we have made it too easy for ourselves by ignoring the most difficult problem of them all: scaling state and, in particular, durable state.

Let's address this and look at how we can scale state (while staying available) in an event-based design, alongside the new opportunities and constraints such a design enables. First and foremost, it requires a new way of thinking about persistence and consistency, with the first step being to move beyond CRUD.

Moving Beyond CRUD

> *When bookkeeping was done with clay tablets or paper and ink, accountants developed some clear rules about good accounting practices.*
>
> *One never alters the books; if an error is made, it is annotated and a new compensating entry is made in the books. The books are thus a complete history of the transactions of the business.*
>
> *Update-in-place strikes many systems designers as a cardinal sin: it violates traditional accounting practices that have been observed for hundreds of years.*
>
> —Jim Gray, *The Transaction Concept*, 1981

Disk space used to be very expensive. This is one of the reasons why most SQL databases are using *update-in-place*—overwriting existing records with new data as it arrives.

As Jim Gray, Turing Award winner and legend in database and transaction processing research, once said, "*Update-in-place strikes many systems designers as a cardinal sin: it violates traditional accounting practices that have been observed for hundreds of years.*" Still, money talked, and CRUD was born.

The good news is that today disk space is incredibly cheap so there is little-to-no reason to use update-in-place for System of Record. We can afford to store all data that has ever been created in a system, giving us the entire history of everything that has ever happened in it.

We don't need *Update* and *Delete* anymore.[1] We just *Create* new facts —either by adding more knowledge, or drawing new conclusions from existing knowledge—and *Read* facts, from any point in the history of the system. CRUD is no longer necessary.

The question is this: how can we do this efficiently? Let's turn our attention to *Event Logging*.

Event Logging—The Scalable Seamstress

> *The truth is the log. The database is a cache of a subset of the log.*
>
> —Pat Helland[2]

One of the most scalable ways to store facts is in an Event Log. It allows us to store them in their natural causal order—the order in which they were created.

The event log is not just a database of the *current state* like traditional SQL databases, but a database of everything that has ever happened in the system, its *full history*. Here, time is a natural index,

1 This is general advice that is meant to guide the design and thought process; by no means is it a rule. There might be legal (data-retention laws), or moral (users requesting their account to be deleted) requirements to physically delete data after a particular period of time. Still, using traditional CRUD is most often the wrong way to think about the design of such a system.

2 The quote is taken from Pat Helland's insightful paper "Immutability Changes Everything".

making it possible for you to travel back and replay scenarios for debugging purposes, auditing, replication, failover, and so on. The ability to turn back time and debug the exact things that have happened, at an exact point in the history of the application, should not be underestimated.

Event Sourcing—A Cure for the Cardinal Sin

A popular pattern for event logging is *event sourcing*, in which we capture the state change—triggered by a command or request—as a new *event* to be stored in the event log. These events represent the fact that something has happened (i.e., *OrderCreated*, *PaymentAuthorized*, or *PaymentDeclined*).

One benefit of using event sourcing is that it allows the aggregate to cache the dataset[3]—the latest snapshot—in memory, instead of having to reconstitute it from durable storage every request (or periodically)—something that is often seen when using raw SQL, Java Database Connectivity (JDBC), Object-Relational Mapping (ORM), or NoSQL databases. This pattern is often referred to as a Memory Image, and it helps to avoid the infamous Object-Relational Impedance Mismatch. It allows us to store the data *in-memory* inside our services, in any format that we find convenient, whereas the master data resides on disk in an optimal format for *append-only* event logging, ensuring efficient write patterns—such as the Single Writer Principle—that are working in harmony with modern hardware, instead of at odds with it. If we now add *Command Query Responsibility Segregation* (CQRS, discussed in a moment) to the mix, to address the query and consistency problems, we have the best of both worlds without much of the drawbacks.

The events are stored in causal order, providing the full history of all the events representing state changes in the service (and in case of the commands, the interactions with the service). Because events most often represent service transactions, the event log essentially provides us with a transaction log that is explicitly available to us for querying, auditing, replaying messages from an arbitrary point in

3 Well-designed event sourcing implementations, such as Akka Persistence, will shard your entities across the cluster and route commands to the shards accordingly, so the aggregate is only cached in one place and doesn't need any replication or synchronization.

time for failover, debugging, and replication—instead of having it abstracted away from the user, as seen in Relational Database Management Systems (RDBMS's).

Each event-sourced aggregate usually has an *event stream* through which it is publishing its events to the rest of the world—for example, such as through Apache Kafka. The *event stream* can be subscribed to by many different parties, for different purposes. Examples include a database optimized for queries, microservices/aggregates that react to events as a way of coordinating workflow, and supporting infrastructure services like audit or replication.

Untangle Your Read and Write Models by using CQRS

CQRS is a technique, coined by Greg Young, to separate the *write* and *read model* from each other, opening up for using different techniques to address each side of the equation.

The read model is most often referred to as the *query side* in CQRS, but that somewhat misses the main point. The read model is better defined as anything that depends on the data that was written by the write model. This includes query models, but also other readers subscribing to the event stream—including other services performing a side effect, or any downstream system that acts in response to the write. As we will see in "Coordinating Work Across Aggregates" on page 54, this broader view of the read model, powered by the event stream, can be the foundation for reliably orchestrating workflow across services, including techniques for managing consistency guarantees and at-least-once delivery of commands.

The write and read models exhibit very different characteristics and requirements in terms of data consistency, availability, and scalability. The benefit of CQRS is that it allows both models to be stored in its optimal format.

There are two main advantages of using CQRS:

Resilience
: Separating the read and write models gives us *temporal decoupling* of the writes and the actions performed in response to the writes—whether it's updating an index in a query database, pushing the events out for stream processing, performing a side-effect, or coordinating work across other services. Temporal decoupling means that the service doing the write as well as

the service performing the action in response to the write don't need to be available at the same time. This avoids the need to handle things like retries, increases reliability, stability, and availability in the system as a whole, and allows for eventual consistency (as opposed to no consistency guarantees at all).

Scalability

The temporal decoupling of reads and writes gives us the ability to scale them independently of one another. For example, in a heavily loaded *read-mostly* system, we can choose to scale-out the query side to tens or even hundreds of nodes, while keeping the write side to three to five nodes for availability, or vice versa for a *write-mostly* system (probably using sharding techniques). A decoupled design like this makes for a lot of flexibility, gives us more headroom and knobs to turn, and allows us to better optimize the hardware efficiency, putting the money where it matters most.

It's worth mentioning that CQRS is a general design pattern that you can use successfully together with almost any persistence strategy and storage backends, but it happens to fit perfectly together with event sourcing in the context of event-driven and message-driven architectures, which is the reason why you will often hear them mentioned in tandem. The opposite is also true: you can use event sourcing successfully without CQRS, reaping the benefits of the event-driven model, log-based structure, historic traceability, and so on, with the option of adding in CQRS at a later point in time, if the need arises.

Figure 6-1 shows the flow of commands and events in an event-sourced service. It begins with a *Command* (*SubmitPayment*) sent to a service (*Payments*) from an external "user" of the service (*Orders*). The *Command* flows through the boundary of the *Bounded Context* for the service and is received by the *Processing Layer*, where it is validated, translated, and so on before the business logic is executed and a new *Command* (*ApprovePayment*)—representing the intent to change the service's state—is created and sent to the service's *Aggregate*. The *Aggregate* receives the *Command* and creates an *Event* (*PaymentApproved*) representing the state change, and stores it in its *Event Log*. After the *Event* is successfully saved, it is pushed to the *Event Stream* for public consumption, where it is relayed to its subscribers (in this case the *Order*s service and the *Payment Query Database*).

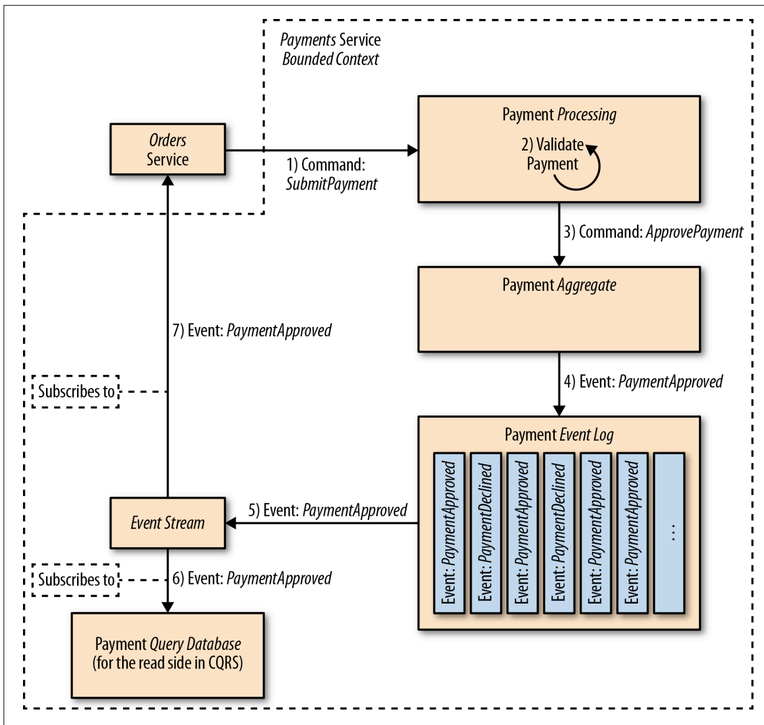


Figure 6-1. The flow of commands and events in the payments service

A design like this helps with scalability, resilience, and consistency, allowing a failed service to resume processing another service's event stream after restart (or even replay parts of the stream if the context is lost).

Coordinating Work Across Aggregates

> *Nature laughs at the difficulties of integration.*
>
> —Pierre-Simon Laplace

As just discussed, each event-sourced aggregate has an *Event Stream* that is available for anyone to subscribe to. These event streams can be used to coordinate work across aggregates by means of chaining events, with the completion (or intent) of each step in the workflow represented by an event; for example, *OrderCreated*, *PaymentApproved*, or *PaymentDeclined*, where the next step in the process is started upon the reception of the previous event.

A simple way to help you name events is to remember that they should be a past-tense description of what happened. Unlike commands, which would be phrased in the imperative (*CreateOrder*, *SubmitPayment*, and *ShipProducts*).

One of the benefits of this event-driven approach to aggregate coordination and workflow management is that it is very resilient—thanks to being decoupled, asynchronous, and nonblocking, with all events already made durable and replayable upon failure. It also is fairly easy and intuitive to understand and to wire-up in terms of event subscriptions.

Some people call this the Process Manager pattern.

Figure 6-2 presents an example of how we can use a *Process Manager* to coordinate the work between the *Orders*, *Inventory*, and *Payment* aggregates, by subscribing to events in the *Event Stream*. All communication here (except potentially step 1 and 11) are done over asynchronous message-passing. In this illustration, I have omitted the event log for simplicity.

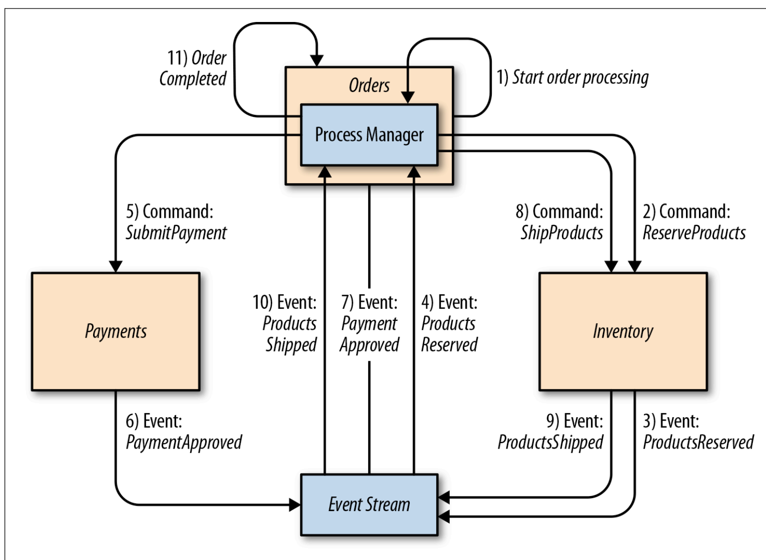


Figure 6-2. Using a Process Manager to coordinate work between multiple services

The order management process runs as follows:

1. The *Orders* service starts a workflow by initiating the processing of a new order.
2. The *Process Manager* sets up the workflow by subscribing to the interesting events in the *Event Stream*: *ProductsReserved*, *PaymentApproved*, *PaymentDeclined*, and *ProductsShipped* (this is, of course, drastically simplified compared to a real-world solution). It then tries to reserve the products by sending a *ReserveProducts* command to the *Inventory* service.
3. The *Inventory* service reserves the products, creates a *ProductsReserved* event, stores it in the *Event Log*, and emits it to the *Event Stream*.
4. The *Process Manager* receives the *ProductsReserved* event and starts the next phase in the workflow: the payment of the products.
5. The *Process Manager* initiates the payment process by sending a *SubmitPayment* command to the *Payments* service.
6. The *Payments* service tries to verify the payment, and, if successful, creates and emits a *PaymentApproved* event to the *Event Stream* (if the payment is not successful, it emits a *PaymentDeclined* event).
7. The *Process Manager* receives the *PaymentApproved* event and starts the final phase in the workflow: shipping the products.
8. The *Process Manager* initiates the shipping process by sending a *ShipProducts* command to the *Inventory* service.
9. The *Inventory* service completes the shipping, and emits a *ProductsShipped* event to the *Event Stream*.
10. The *Process Manager* receives the *ProductsShipped* event, marking the end of the ordering process.
11. The *Process Manager* notifies the *Orders* service of the successful completion of the order.

Leverage Polyglot Persistence

> *The limits of my language means the limits of my world.*
> —Ludwig Wittgenstein

As we have already discussed, CQRS makes it possible for us to build on the event-sourcing foundation by having the *query side*

subscribe to the aggregate's *Event Stream* (the write side), consuming successfully saved events and storing them in an optimal representation for queries.

This design allows us to have multiple read models (views), each one maintaining its own view of the data. For example, you could use Cassandra, a SQL database, or Spanner for regular queries, ElasticSearch for general search, and HDFS for batch-oriented data mining.

If you are using CQRS with event sourcing, you can even add additional views dynamically, at any point in time, because you can simply replay the event log—the history—to bring the new view up to speed.

When it comes to the read side there are many options:

- Traditional RDBMSs, like MySQL, Oracle, or Postgres
- Scalable distributed SQL, like Spanner or CockroachDB
- Key-value-based, like DynamoDB, or Riak
- Column-oriented, like HBase, or Vertica
- Hybrid key-value/column-oriented, like Cassandra
- Document-oriented, like MongoDB or Couchbase
- Graph-oriented, like Neo4j or AllegroGraph
- Time–series-based, like OpenTSDB or InfluxDB
- Streaming products, like Flink or Kafka Streams
- Distributed file systems, like HDFS or Alluxio
- Search products, like ElasticSearch

You'll notice that all those data stores have very different "sweet spots" and are optimized for different access patterns.

As an example, if you want to build a graph of friends, and run queries along the lines of *"who's my friend's best friend?"*, this query will be most efficiently answered by a graph-oriented database (such as Neo4j). A graph database is easily populated dynamically, as friends are adding each other on your social site, by subscribing to the *Friend* aggregate's *Event Stream*. In the same way, you can build other views, using read-specialized databases from your event streams, in real time.

In other words, by having the *event log* as the single *source of truth* in your system, you easily can produce any kind of view on the data that works best for the specific use-case—so-called *Polyglot Persistence*.

Know Your Trade-offs

> *There are no solutions; there are only trade-offs.*
> —Thomas Sowell

One trade-off is that CQRS with event sourcing forces you to tackle the *essential complexity*[4] of the problem head on. This is often a good thing, but if you are building a minimum viable product (MVP) or prototype, a throwaway that you need to get to market quickly in order to test an idea, you might be better off starting with CRUD (and a monolith), moving to a more sustainable design after it has proved its value in the market.

Another trade-off in moving to an architecture based on CQRS with event sourcing is that the write side and read side will be *eventually consistent*. It takes time for events to propagate between the two storage models, which often reside on separate nodes or clusters. The delay is often only a matter of milliseconds to a few seconds, but it can have a big impact on the design of your system.

Using techniques such as a reactive design and event-driven design, denormalization, minimized units of consistency are essential, and makes these trade-offs less of an issue.

In general, it is important to take a step back from years of preconceived knowledge and biases, and look at how the world actually works. The world is seldom strongly consistent, and embracing reality, and the actual semantics in the domain, often opens up opportunities for relaxing the consistency requirements.

One problem that all persistence approaches have—but for which event sourcing together with CQRS offers a straightforward solution—is the fact that saving the event, and executing the side-effect that the event represents, often can't be performed as a single atomic operation. The best strategy is to rely on acknowledgements (the

4 You can find a good definition of the difference between *essential complexity* and *accidental complexity* here.

events) pushed through the *event stream*, and think in terms of at-least-once execution of the side-effects, which means that you must consider whether to make the side-effect idempotent.

You have two options:

- Perform the side-effect before persisting the event, with the risk that the side-effect is performed but the event is never stored. This pattern works well when you can depend on the upstream to retry an operation—by resubmitting the command—until it is successful. It can be done by subscribing to the aggregate's event stream (as previously discussed), waiting for the acknowledgement (the event itself) that the event was persisted successfully, and if not received within a specific time window, resubmitting the command.
- Store an event representing the intent to perform the side-effect, perform the side-effect itself, and finally persist an event—the acknowledgment—that the side-effect was successfully performed. In this style, you take on the responsibility of executing the action, so upon replay you then can choose to reexecute the side-effect if the acknowledgement is missing.

Finally, one thing to take into account is that using CQRS with event sourcing makes it more difficult to delete data—which is something that is becoming increasingly important, for both legal and moral reasons. First, we need to manage deletion of the events in the event log. There are many different strategies for this, but discussing them is beyond the scope of this report. Second, the events also have been made available through the aggregate's event stream, where they could have been picked up by several other databases, systems, and services. Keeping track of where all events end up is a nontrivial exercise that needs to be thought through carefully when designing the system (and be maintained throughout long-term evolution of the system).

Transactions—The Anti-Availability Protocol

> *Two-phase commit is the anti-availability protocol.*
> —Pat Helland

At this point, you might be thinking, "But what about transactions? I really need transactions!"

Let's begin by making one thing clear: transactions are fine within individual services, where we can, and should, guarantee strong consistency. This means that it is fine to use transactional semantics within a single service (the bounded context)—which is something that can be achieved in many ways: using a traditional SQL database like Oracle, a modern distributed SQL database like CockroachDB, or using event sourcing through Akka Persistence. What is problematic is expanding them beyond the single service, as a way of trying to bridge data consistency across multiple services (i.e. bounded contexts).[5]

The problem with transactions is that their only purpose is to try to maintain the illusion that the world consists of a single globally strongly consistent present—a problem that is magnified exponentially in *distributed* transactions (XA, Two-phase Commit, and friends). We already have discussed this at length: it is simply not how the world works, and computer science is no different.

As Pat Helland says,[6] "*Developers simply do not implement large scalable applications assuming distributed transactions.*"

If the traits of scalability and availability are not important for the system you are building, go ahead and knock yourself out—XA and two-phase commit are waiting. But if it matters, we need to look elsewhere.

Don't Ask for Permission—Guess, Apologize, and Compensate

> *It's easier to ask for forgiveness than it is to get permission.*
>
> —Grace Hopper

So, what should we do? Let's take a step back and think about how we deal with partial and inconsistent information in real life.

For example, suppose that we are chatting with a friend in a noisy bar. If we can't catch everything that our friend is saying, what do we do? We usually (hopefully) have a little bit of patience and allow

5 The infamous, and far too common, anti-pattern "Integrating over Database" comes to mind.

6 This quote is from Pat Helland's excellent paper "Life Beyond Distributed Transactions".

ourselves to wait a while, hoping to get more information that can fill out the missing pieces. If that does not happen within our window of patience, we ask for clarification, and receive the same or additional information.

We do not aim for guaranteed delivery of information, or assume that we can always have a complete and fully consistent set of facts. Instead, we naturally use a protocol of *at-least-once message delivery* and *idempotent messages*.

At a very young age, we also learn how to take *educated guesses* based on *partial information*. We learn to react to missing information by trying to *fill in the blanks*. And if we are wrong, we take *compensating actions*.

We need to learn to apply the same principles in system design, and rely on a protocol of: *Guess*; *Apologize*; *Compensate.*[7] It's how the world works around us all the time.

One example is ATMs. They allow withdrawal of money even under a network outage, *taking a bet* that you have sufficient funds in your account. And if the bet proved to be wrong, it will correct the account balance—through a *compensating action*—by deducting the account to a negative balance (and in the worst case the bank will employ collection agencies to recuperate any incurred debt).

Another example is airlines. They deliberately overbook aircrafts, *taking a bet* that not all passengers will show up. And if they were wrong, and all people show up, they then try to bribe themselves out of the problem by issuing vouchers—performing *compensating actions*.

We need to learn to *exploit reality* to our advantage.

Use Distributed Sagas, Not Distributed Transactions

The Saga pattern[8] is a failure-management pattern that is a commonly used alternative to distributed transactions. It helps you to

7 It's worth reading Pat Helland's insightful article "Memories, Guesses, and Apologies".

8 See Clemens Vasters' post "Sagas" for a short but good introduction to the idea. For a more in-depth discussion, putting it in context, see Roland Kuhn's excellent book *Reactive Design Patterns* (Manning).

manage long-running business transactions that make use of compensating actions to manage inconsistencies (transaction failures).

The pattern was defined by Hector Garcia-Molina in 1987[9] as a way to shorten the time period during which a database needs to take locks. It was not created with distributed systems in mind, but it turns out to work very well in a distributed context.[10]

The essence of the idea is that one long-running *distributed* transaction can be seen as the composition of multiple quick *local* transactional steps. Every transactional step is paired with a *compensating reversing action* (reversing in terms of business semantics, not necessarily resetting the state of the component), so that the entire distributed transaction can be reversed upon failure by running each step's compensating action. Ideally, these steps should be *commutative* so that they can be run in parallel.

The Saga is usually conducted by a coordinator, a single centralized Finite State Machine, that needs to be made durable—preferably event logged, to allow replay on failure.

One of the benefits of this technique (see Figure 6-3) is that it is *eventually consistent* and works well with decoupled and asynchronously communicating components, making it a great fit for event-driven and message-driven architectures.

9 Originally defined in the paper "Sagas" by Hector Garcia-Molina and Kenneth Salem.

10 For an in-depth discussion, see Catie McAffery's great talk on Distributed Sagas.

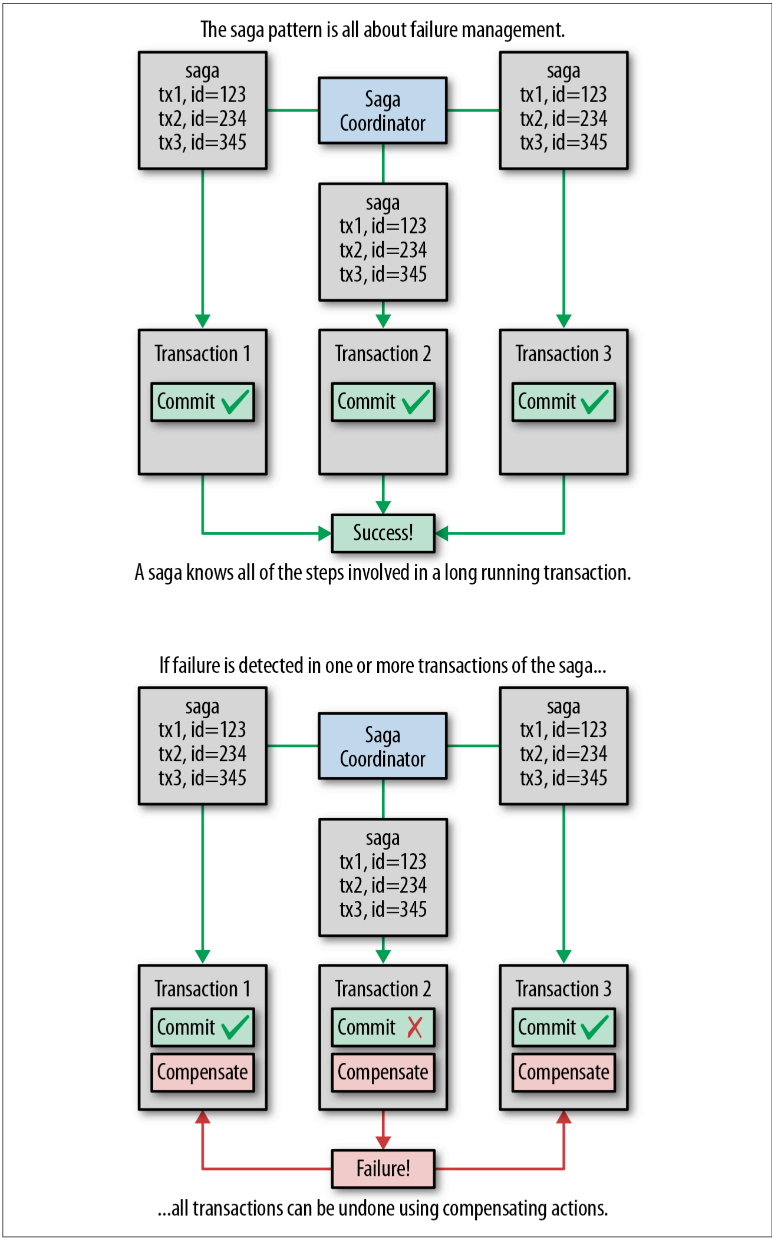


Figure 6-3. Using Sagas for failure management of long-running distributed workflows across multiple services

As we have seen, the Saga pattern is a great tool for ensuring *atomicity* in long-running transactions. But, it's important to understand that it does *not* provide a solution for *isolation*. Concurrently executed Sagas could potentially affect one another and cause errors. If this is acceptable, it is use-case dependent. If it's not acceptable, you need to use a different strategy, such as ensuring that the Saga does not span multiple consistency boundaries or simply using a different pattern or tool for the job.

Distributed Transactions Strikes Back

> *No. Try not. Do... or do not. There is no try.*
> —Yoda, *The Empire Strikes Back*

After this lengthy discussion outlining the problems with transactions in a distributed context and the benefits of using event logging, it might come as a surprise to learn that SQL and transactions are on the rise again.

Yes, SQL and SQL-style query languages are becoming popular again. We can see it used in the big data community as a way of querying large datasets. For example, Hive and Presto, as well as the NoSQL community, allow for richer queries than key/value lookups, such as Cassandra (with its CQL) and Google's Cloud Spanner.

Spanner[11] is particularly interesting because it is not only supporting SQL, but has managed to implement large-scale distributed transactions in a both scalable and highly-available manner. It is not for the faint of heart, considering that Google runs it on a private and highly optimized global network, using Paxos groups, coordinated using atomic clocks, and so on.[12]

It's worth mentioning that there is an open source implementation of Spanner called CockroachDB that can be worth looking into if you have use-cases that fit this model. However, they will not be a good fit if you are expecting low-latency writes from your datastore. These datastores choose to give up on latency—by design—in order to achieve high consistency guarantees.

11 For an understanding about how Spanner works, see the original paper, "Spanner: Google's Globally-Distributed Database".

12 If you are interested in this, be sure to read Eric Brewer's "Spanner, TrueTime, and the CAP Theorem".

Another recent discovery is that many of the traditional RDBMS guarantees that we have learned to use and love are actually possible to implement in a scalable and highly available manner. Peter Bailis et al. have shown[13] that we could, for example, keep using *Read Committed*, *Read Uncommitted*, and *Read Your Writes*, whereas we must give up on *Serializable*, *Snapshot Isolation*, and *Repeatable Read*. This is recent research but something I believe more SQL and NoSQL databases should start taking advantage of in the near future.

So, SQL, distributed transactions, and more refined models on how to manage data consistency at scale,[14] are on the rise. Though new, these models are backed by very active and promising research, and worth keeping an eye on. This is great news, since managing data consistency in the application layer has never been something that we developers are either good at, or enjoy.[15] It's been a necessary evil to get the job done.

---

13 For more information, see "Highly Available Transactions: Virtues and Limitations", by Peter Bailis et al.

14 One fascinating paper on this topic is "Coordination Avoidance in Database Systems" by Peter Bailis et al.

15 A must-see talk, explaining the essence of problem, and painting a vision for where we need to go as an industry, is Peter Alvaro's excellent RICON 2014 keynote "Outwards from the Middle of the Maze".

CHAPTER 7

The World Is Going Streaming

> *You could not step twice into the same river. Everything flows and nothing stays.*
>
> —Heraclitus

The need for asynchronous message-passing not only includes responding to individual messages or requests, but also to continuous streams of data, potentially unbounded streams. Over the past few years, the streaming landscape has exploded in terms of both products and definitions of what streaming really means.[1]

There's no clear boundary between processing of messages that are handled individually and data records that are processed *en masse*. Messages have an individual identity and each one requires custom processing, whereas we can think of records as anonymous by the infrastructure and processed as a group. However, at very large volumes, it's possible to process messages using streaming techniques, whereas at low volumes, records can be processed individually. Hence, the characteristics of data records versus messages is an orthogonal concern to how they are processed.

The fundamental shift is that we've moved from "data at rest" to "data in motion." The data used to be offline and now it's online. Applications today need to react to changes in data in close to real

1 We are using Tyler Akidau's definition of streaming: "A type of data processing engine that is designed with infinite data sets in mind", from his article "The world beyond batch: Streaming 101".

time—when it happens—to perform continuous queries or aggregations of inbound data and feed it—in real time—back into the application to affect the way it is operating.

Three Waves Toward Fast Data

The first wave of big data was "data at rest." We stored massive amounts in Hadoop Distributed File System (HDFS) or similar, and then had offline batch processes crunching the data over night, often with hours of latency.

In the second wave, we saw that the need to react in real time to the "data in motion"—to capture the live data, process it, and feed the result back into the running system within seconds and sometimes even subsecond response time—had become increasingly important.

This need instigated hybrid architectures such as the Lambda Architecture, which had two layers: the "speed layer" for real-time online processing and the "batch layer" for more comprehensive offline processing. This is where the result from the real-time processing in the "speed layer" was later merged with the "batch layer." This model solved some of the immediate need for reacting quickly to (at least a subset of) the data. But, it added needless complexity with the maintenance of two independent models and data processing pipelines, as well as a data merge in the end.

The third wave—that we have already started to see happening—is to fully embrace "data in motion" and, where possible, move away from the traditional batch-oriented architecture altogether toward a pure stream-processing architecture.

Leverage Fast Data in Microservices

The third wave—distributed streaming—is the one that is most interesting to microservices-based architectures.

Distributed streaming can be defined as partitioned and distributed streams, for maximum scalability, working with infinite streams of data—as done in Flink, Spark Streaming, and Google Cloud Dataflow. It is different from application-specific streaming, performed locally within the service, or between services and client/service in a point-to-point fashion—which we covered earlier, and includes pro-

tocols such as Reactive Streams, Reactive Socket, WebSockets, HTTP 2, gRPC, and so on.

If we look at microservices from a distributed streaming perspective, microservices make great stream pipeline endpoints, bridging the application side with the streaming side. Here, they can either ingest data into the pipeline—data coming from a user, generated by the application itself, or from other systems—or query it, passing the results on to other applications or systems. Using an integration library that understands streaming, and back-pressure, natively like Alpakka (a Reactive Streams-compatible integration library for Enterprise Integration Patterns based on Akka Streams)—can be very helpful.

From a microservices perspective, distributed streaming has emerged as a powerful tool alongside the application, where it can be used to crunch application data and provide analytics functionality to the application itself, in close to real time. It can help with analyzing both user provided business data as well as metadata and metrics data generated by the application itself—something that can be used to influence how the application behaves under load or failure, by employing predictive actions.

Lately, we also have begun to see distributed streaming being used as the data distribution fabric for microservices, where it serves as the main communication backbone in the application. The growing use of Kafka in microservices architecture is a good example of this pattern.

Another important change is that although traditional (overnight) batch processing platforms like Hadoop could get away with high latency and unavailability at times, modern distributed streaming platforms like Spark, Flink, and Google Cloud Dataflow need to be Reactive. That is, they need to *scale elastically*, reacting adaptively to usage patterns and data volumes; be *resilient*, always available, and never lose data; and be *responsive*, always deliver results in a timely fashion.

We also are beginning to see more microservices-based systems grow to be dominated by data, making their architectures look more like big pipelines of streaming data.

To sum things up: from an operations and architecture perspective, distributed streaming and microservices are slowly unifying, both relying on Reactive architectures and techniques to get the job done.

CHAPTER 8

Next Steps

We have covered a lot of ground in this report, yet for some of the topics we have just scratched the surface. I hope it has inspired you to learn more and to roll up your sleeves and try these ideas out in practice.

Further Reading

Learning from past failures[1] and successes[3] in distributed systems and collaborative services-based architectures is paramount. Thanks to books and papers, we don't need to live through it all ourselves but have a chance to learn from other people's successes, failures, mistakes, and experiences.

There are a lot of references throughout this report, I very much encourage you to read them.

When it comes to books, there are so many to recommend. If I had to pick two that take this story further and provide practical real-world advice, they would be Roland Kuhn's excellent *Reactive Design*

1 The failures of SOA, CORBA, EJB,[2] and synchronous RPC are well worth studying and understanding.

2 Check out Bruce Tate, Mike Clark, Bob Lee, Patrick Linskey's book, *Bitter EJB* (Manning).

3 Successful platforms with tons of great design ideas and architectural patterns have so much to teach us—for example, Tandem Computer's NonStop platform, the Erlang platform, and the BitTorrent protocol.

Patterns (Manning) and Vaughn Vernon's thorough and practical *Implementing Domain-Driven Design* (Addison-Wesley).

Start Hacking

The good news is that you do not need to build all of the necessary infrastructure and implement all the patterns from scratch yourself. The important thing is understanding the design principles and philosophy. When it comes to implementations and tools, there are many off-the-shelf products that can help you with the implementation of most of the things we have discussed.

One of them is the Lagom[4] microservices framework, an open source, Apache 2–licensed framework with Java and Scala APIs. Lagom pulls together most of the practices and design patterns discussed in this report into a single, unified framework. It is a formalization of all the knowledge and design principles learned over the past eight years of building microservices and general distributed systems in Akka and Play Framework.

Lagom is a thin layer on top of Akka and Play, which ensures that it works for massively scalable and always available distributed systems, hardened by thousands of companies for close to a decade. It also is highly opinionated, making it easy to do the right thing in terms of design and implementation strategies, giving the developer more time to focus on building business value.

Here are just some of the things that Lagom provides out of the box:

- Asynchronous by default:
  - Async IO
  - Async Streaming—over WebSockets and Reactive Streams
  - Async Pub/Sub messaging—over Kafka
  - Intuitive DSL for REST over HTTP, when you need it
- Event-based persistence:
  - CQRS and Event Sourcing—over Akka Persistence and Cassandra

4 Lagom means "just right," or "just the right size," in Swedish and is a humorous answer to the common but nonsensical question, "What is the right size for a microservice?"

  - Great JDBC and JPA support
- Resilience and elasticity of each microsystem through:
  - Decentralized peer-to-peer cluster membership
  - Consistency through CRDTs over epidemic gossip protocols
  - Failure detection, supervision, replication, and automatic failover/restart
  - Circuit breakers, service discovery, service gateway, and so on
- Highly productive (Rails/JRebel-like) iterative development environment:
  - Hot code reload on save and so on.
  - Automatic management of all infrastructure
  - IDE integrations

Let Lagom do the heavy lifting. Have fun.

About the Author

Jonas Bonér is Founder and CTO of Lightbend, inventor of the Akka project, coauthor of the *Reactive Manifesto*, a Java Champion, and author of *Reactive Microservices Architecture* (O'Reilly). Learn more about Jonas at his website.