O'REILLY®

Reactive Microservices Architecture

Design Principles for Distributed Systems

Jonas Bonér

O'REILLY®

Additional Resources

4 Easy Ways to Learn More and Stay Current

Programming Newsletter
Get programming related news and content delivered weekly to your inbox.
oreilly.com/programming/newsletter

Free Webcast Series
Learn about popular programming topics from experts live, online.
webcasts.oreilly.com

O'Reilly Radar
Read more insight and analysis about emerging technologies.
radar.oreilly.com

Conferences
Immerse yourself in learning at an upcoming O'Reilly conference.
conferences.oreilly.com

Reactive Microservices Architecture

Design Principles for Distributed Systems

Jonas Bonér

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Reactive Microservices Architecture
by Jonas Bonér

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: Colleen Cole
Copyeditor: Colleen Toporek
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kevin Webber

March 2016: First Edition

Revision History for the First Edition

2016-03-15: First Release
2016-12-09: Second Release

978-1-491-95779-0

[LSI]

Table of Contents

CHAPTER 1
Introduction

We change a monolithic system only when we have no other choice. Rather than swiftly capture opportunity, we ponder if it's really worth upsetting the delicate balance of the house of cards we call our enterprise system. Often the opportunity quickly disappears, captured by a faster company, as in Figure 1-1.

> In the new world, it is not the big fish which eats the small fish, it's the fast fish which eats the slow fish.
>
> —Klaus Schwab

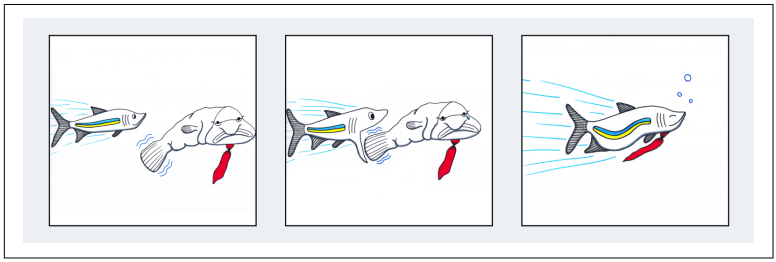

Figure 1-1. Slow fish versus fast fish

Microservices-Based Architecture is a simple concept: it advocates creating a system from a collection of small, isolated services, each of which owns their data, and is independently isolated, scalable and resilient to failure. Services integrate with other services in order to form a cohesive system that's far more flexible than the typical enterprise systems we build today.

Traditional enterprise systems are designed as *monoliths*—all-in-one, all-or-nothing, difficult to scale, difficult to understand and difficult to maintain. Monoliths can quickly turn into nightmares that stifle innovation, progress, and joy. The negative side effects caused by monoliths can be catastrophic for a company—everything from low morale to high employee turnover, from preventing a company from hiring top engineering talent to lost market opportunities, and in extreme cases, even the failure of a company.

The war stories often sound like this: "We finally made the decision to make changes to our Java EE application, after seeking approval from management. Then we went through months of big up-front design before we eventually started to build something. But most of our time during construction was spent trying to figure out what the monolith actually did. We became paralyzed by fear, worried that one small mistake would cause unintended and unknown side effects. Finally, after months of worry, fear, and hard work, the changes were implemented—and hell broke loose. The collateral damage kept us awake for nights on end while we were firefighting and trying to put the pieces back together."

Does this sound familiar?

Experiences like this enforce fear, which paralyzes us even further. This is how systems, and companies, stagnate. What if there was a better way?

> You've got to start with the customer experience and work back towards the technology.
>
> —Steve Jobs

The customers of Microservices are the organizations who invest in systems, so let's start with the customer: developers, architects, and key stakeholders.

Do you prefer to work on a large system and have a small impact, or work on a small, well-defined part of the system and have a large impact? Do you do your best work in a large bureaucratic group, or on a small team of people that you know and trust? Do you do your best work when delegated to, or when you're given room to think creatively and build useful things? Enter Microservices.

Services to the Rescue

> Although the world is full of suffering, it is also full of the overcoming of it.
>
> —Helen Keller

Microservices are the next design evolution in software not purely because of technical reasons. The ideas embodied within the term Microservices have been around well before our first venture into Service Oriented Architecture (SOA). Certain technical constraints held us back from taking the concepts embedded within the Microservices term to the next level: single machines running single core processors, slow networks, expensive disks, expensive RAM, and organizations structured as monoliths. Ideas such as organizing systems into well-defined components with a single responsibility are not new.

Fast forward to 2016. The technical limitations holding us back from Microservices are gone. Networks are fast, disks are cheap (and a lot faster), RAM is cheap, multi-core processors are cheap, and cloud architectures are revolutionizing how we design and deploy systems. Now we can finally structure our systems with the customer in mind.

Designing and programming software is fun, which is why most of us entered the software industry to begin with. Microservices are more than a series of principles and technologies. They're a way to approach the complex problem of systems design in a more empathetic way.

Microservices enable us to structure our systems the same way we structure our teams, dividing responsibilities among team members and ensuring they are free to own their work. As we detangle our systems, we shift the power from central governing bodies to smaller teams who can seize opportunities rapidly and stay nimble because they understand the software within well defined boundaries that they control.

Slicing the Monolith

Tackling a monolith means taking a hard look at your traditional Java EE systems. Written in a monolithic way, these systems tend to

have strong coupling between the components in the service[1] and between services. A system with the services tangled and interdependent is harder to write, understand, test, evolve, upgrade and operate independently. Worse still, strong coupling can also lead to cascading failures—where one failing service can take down the entire system, instead of allowing you to deal with the failure in isolation.

One problem has been that application servers (e.g., WebLogic, WebSphere, JBoss and Tomcat—even though Tomcat does not support EAR files) encourage this monolithic model. They assume that you are bundling your service JARs into an EAR 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, which manages the service "isolation" through class loader tricks. All in all, a very fragile model.

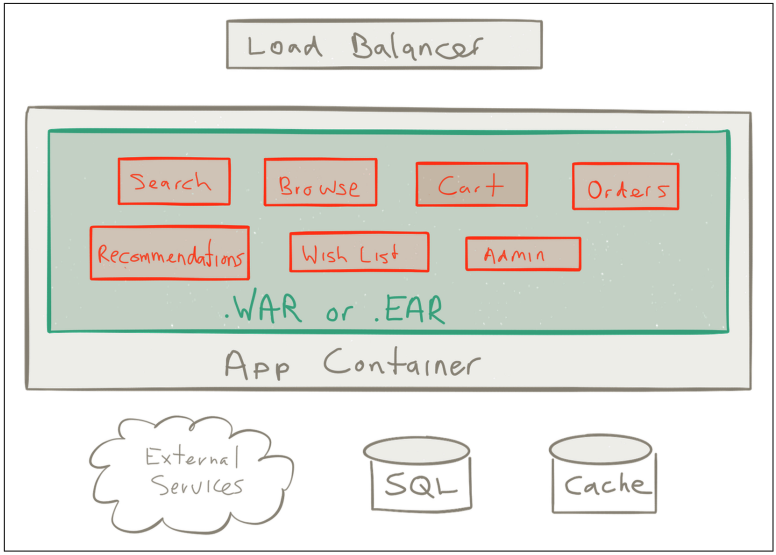

Figure 1-2. Classical Java EE architecture

Today we have a much more refined foundation for isolation of services, using virtualization, Linux Containers (LXC), Docker, and Unikernels. This has made it possible to treat isolation as a first-class

1 I am using the word *Microservice* and *service* interchangeably throughout this document. Both refer to the idea of a Reactive Microservice.

concern—a necessity for resilience, scalability, continuous delivery and operations efficiency. It has also paved the way for the rising interest in Microservices-Based Architectures, allowing you to slice up the monolith and develop, deploy, run, scale and manage the services independently of each other.

SOA Dressed in New Clothes?

> How splendid his Majesty looks in his new clothes, and how well they fit!" everyone cried out. "What a design! What colors! These are indeed royal robes!
>
> —"The Emperor's New Clothes" by H.C. Andersen

A valid question to ask is whether Microservices are actually just SOA dressed up in new clothes. The answer is both yes and no. Yes, because the initial goals—decoupling, isolation, composition, integration, discrete and autonomous services—are the same. And no, because the fundamental ideas of SOA were most often misunderstood and misused, resulting in complicated systems where an Enterprise Service Bus (ESB) was used to hook up multiple monoliths, communicating over complicated, inefficient and inflexible protocols.

Anne Thomas captures this very well in her article SOA is Dead; Long Live Services:[2]

> Although the word "SOA" is dead, the requirement for service-oriented architecture is stronger than ever. But perhaps that's the challenge: The acronym got in the way. People forgot what SOA stands for. They were too wrapped up in silly technology debates (e.g., "what's the best ESB?" or "WS-* vs. REST"), and they missed the important stuff: architecture and services.
>
> Successful SOA (i.e., application re-architecture) requires disruption to the status quo. SOA is not simply a matter of deploying new technology and building service interfaces to existing applications; it requires redesign of the application portfolio. And it requires a massive shift in the way IT operates.

The world of the software architect looks very different today than it did 10-15 years ago when SOA emerged. Today, multi-core processors, cloud computing, mobile devices and the Internet of Things

2 "SOA is Dead; Long Live Services" by Anne Thomas, VP and Distinguished Analyst at Gartner, Inc.

(IoT) are emerging rapidly, which means that all systems are distributed systems from day one—a vastly different and more challenging world to operate in.

As always, new challenges demand a new way of thinking and we have seen new systems emerge that are designed to deal with these new challenges—systems built on the *Reactive principles*, as defined by the Reactive Manifesto.[3]

The Reactive principles are in no way new. They have been proven and hardened for more than 40 years, going back to the seminal work by Carl Hewitt and his invention of the Actor Model, Jim Gray and Pat Helland at Tandem Systems, and Joe Armstrong and Robert Virding and their work on Erlang. These people were ahead of their time, but now the world has caught up with their innovative thinking and we depend on their discoveries and work more than ever.

What makes Microservices interesting is that this architecture has learned from the failures and successes of SOA, kept the good ideas, and re-architected them from the ground up using Reactive principles and modern infrastructure. In sum, Microservices are one of the most interesting applications of the Reactive principles in recent years.

3 "The Reactive Manifesto" can be found at *www.reactivemanifesto.org*. If you have not done so already, I recommend that you read it now because this book rests on the foundation of the Reactive principles.

CHAPTER 2
What Is a Reactive Microservice?

One of the key principles in employing a Microservices-based Architecture is Divide and Conquer: the decomposition of the system into discrete and isolated subsystems communicating over well-defined protocols.

Isolation is a prerequisite for resilience and elasticity and requires asynchronous communication boundaries between services to decouple them in:

Time
: Allowing concurrency

Space
: Allowing distribution and mobility—the ability to move services around

When adopting Microservices, it is also essential to eliminate shared mutable state[1] and thereby minimize coordination, contention and coherency cost, as defined in the Universal Scalability Law[2] by embracing a Share-Nothing Architecture.

1 For an insightful discussion on the problems caused by a mutable state, see John Backus' classic Turing Award Lecture "Can Programming Be Liberated from the von Neumann Style?"

2 Neil Gunter's Universal Scalability Law is an essential tool in understanding the effects of contention and coordination in concurrent and distributed systems.

At this point in our journey, it is high time to discuss the most important parts that define a Reactive Microservice.

Isolate All the Things

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

Isolation is the most important trait. It is the foundation for many of the high-level benefits in Microservices. But it is also the trait that has the biggest impact on your design and architecture. It will, and should, slice up the whole architecture, and therefore it needs to be considered from day one. It will even impact the way you break up and organize the teams and their responsibilities, as Melvyn Conway discovered and was later turned into Conway's Law in 1967:

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

Failure isolation—to contain and manage failure without having it cascade throughout the services participating in the workflow—is a pattern sometimes referred to as *Bulkheading*.

Bulkheading has been used in the ship construction for centuries as a way to "create watertight compartments that can contain water in the case of a hull breach or other leak."[3] The ship is divided into distinct and completely isolated watertight compartments, so that if compartments are filled up with water, the leak does not spread and the ship can continue to function and reach its destination.

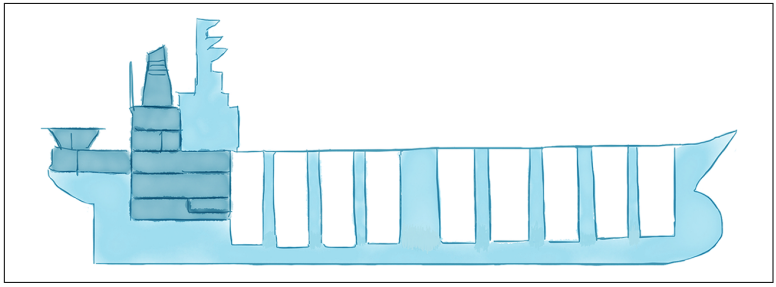

Figure 2-1. Using bulkheads in ship construction

3 For a discussion on the use of bulkheads in ship construction, see the Wikipedia page *https://en.wikipedia.org/wiki/Bulkhead_(partition)*.

Some people might come to think of the Titanic as a counter-example. It is actually an interesting study[4] 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 whole system. The Titanic did use bulkheads, but the walls that were suppose to isolate the compartments did not reach all the way up to the ceiling. So when 6 out of its 16 compartments were ripped open by the iceberg, the ship started to tilt and water spilled over from one compartment to the next, until all of the compartments were filled with water and the Titanic sank, killing 1500 people.

Resilience—the ability to heal from failure—depends on compartmentalization and containment of failure, and can only be achieved by breaking free from the strong coupling of synchronous communication. Microservices communicating over a process boundary using asynchronous message-passing enable the level of indirection and decoupling necessary to capture and manage failure, orthogonally to the regular workflow, using *service supervision*.[5]

Isolation between services makes it natural to adopt Continuous Delivery. This allows you to safely deploy applications and roll out and revert changes incrementally—service by service.

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

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

5 Process (service) supervision is a construct for managing failure used in Actor languages (like Erlang) and libraries (like Akka). Supervisor hierarchies is a pattern where the processes (or actors/services) are organized in a hierarchical fashion where the parent process is supervising its subordinates. For a detailed discussion on this pattern see "Supervision and Monitoring."

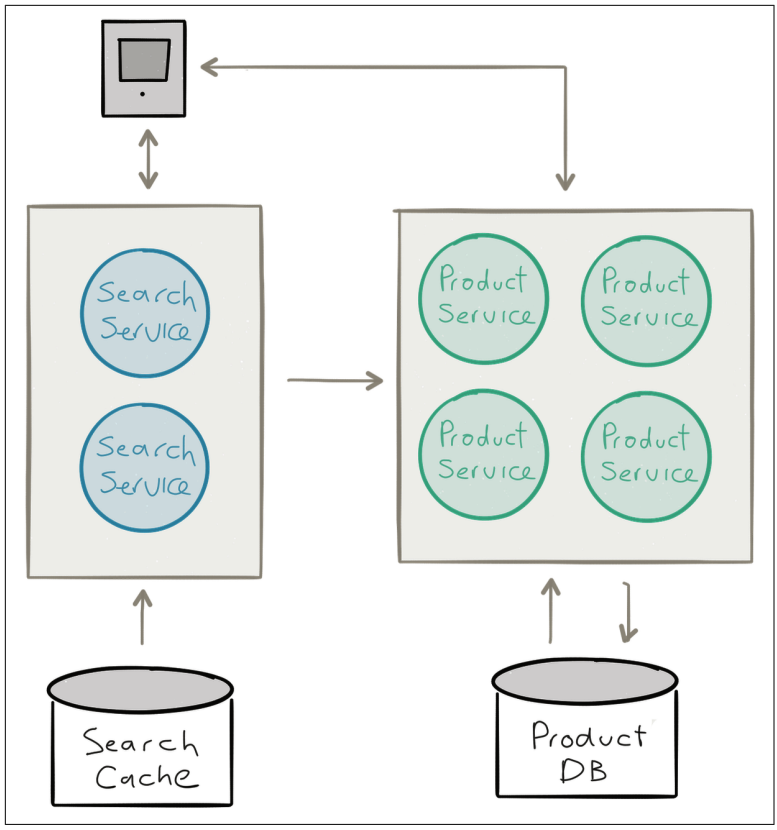

Figure 2-2. Bounded contexts of Microservices

Act Autonomously

> Insofar as any agent acts on reason alone, that agent adopts and acts only on self-consistent maxims that will not conflict with other maxims any such agent could adopt. Such maxims can also be adopted by and acted on by all other agents acting on reason alone.
>
> —Law of Autonomy by Immanuel Kant

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.

An *autonomous service* can only *promise*[6] its own behaviour by publishing its protocol/API. Embracing this simple yet fundamental fact has profound impact on how we can understand and model collaborative systems with autonomous services.

Another aspect of autonomy is that if a service only can make promises about its own behavior, then all information needed to resolve a conflict or to repair under failure scenarios are available within the service itself, removing the need for communication and coordination.

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 that can make communication—and consensus—a bit more challenging—something we will discuss shortly.

6 Our definition of a promise is taken from the chapter "Promise Theory" from *Thinking in Promises* by Mark Burgess (O'Reilly), which is a very helpful tool in modeling and understanding reality in decentralized and collaborative systems. It shows us that by letting go and embracing uncertainty we get on the path towards greater certainty.

Do One Thing, and Do It Well

> 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[7] 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 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 (SRP),[8] which states a class or component should "only have 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 only has one single reason to exist, providing a single composable piece of functionality, then 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.

7 The Unix philosophy is captured really well in the classic book *The Art of Unix Programming* by Eric Steven Raymond (Pearson Education, Inc.).

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

Own Your State, Exclusively

> Without privacy there was no point in being an individual.
>
> —Jonathan Franzen

Up to this point, we have characterized Microservices as a set of isolated services, each one with a single area of responsibility. This 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.

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

Microservices are often stateful entities: they encapsulate state and behavior, in similar fashion to an Object or an Actor, and isolation most certainly applies to state and requires that you treat state and behavior as a single unit.

Unfortunately, ignoring the problem by calling the architecture "stateless"—by having "stateless" controller-style services that are pushing their state down into a big shared database, like many web frameworks do—won't help as much as you would like and only delegate the problem to a third-party, making it harder to control—both in terms of data integrity guarantees as well as scalability and availability guarantees (see Figure 2-3).

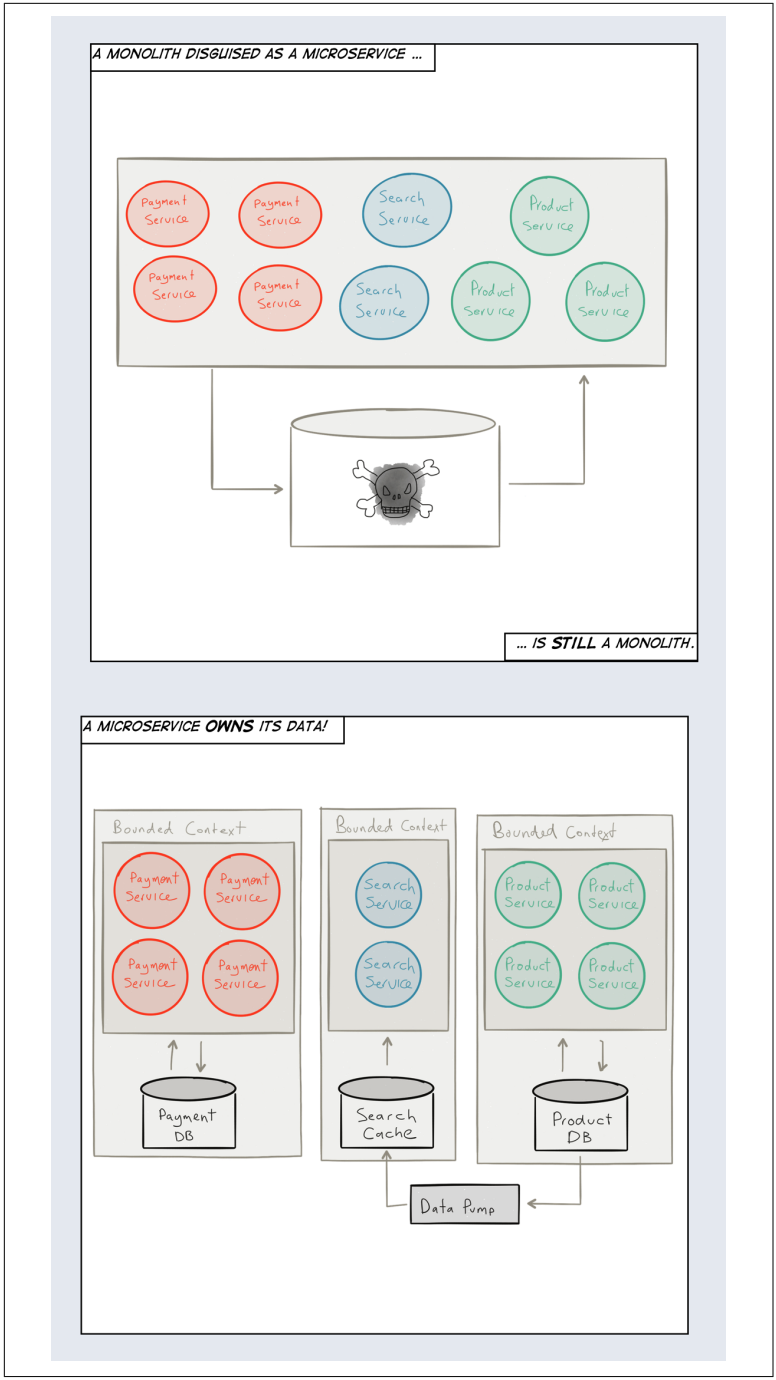

Figure 2-3. A disguised monolith is still a monolith

What is needed is that each Microservice take sole responsibility for their own state and the persistence thereof. Modeling each service as a Bounded Context[9] can be helpful since each service usually defines its own domain, each with its own Ubiquitous Language. Both these techniques are taken from the Domain-Driven Design (DDD)[10] toolkit of modeling tools. Of all the new concepts introduced here, consider DDD a good place to start learning. Microservices are heavily influenced by DDD and many of the terms you hear in context of Microservices come from DDD.

When communicating with another Microservice, across Bounded Contexts, you can only ask politely for its state—you can't force it to reveal it. Each service responds to a request at its own will, with immutable data (facts) derived from its current state, and never exposes its mutable state directly.

This gives each service the freedom to represent its state in any way it wants, and store it in the format and medium that is most suitable. Some services might choose a traditional Relational Database Management System (RDBMS) (examples include Oracle, MySQL and Postgres), some a NoSQL database (for example Cassandra and Riak), some a Time-Series database (for example InfluxDB and OpenTSDB) and some to use an Event Log[11] (good backends include Kafka, Amazon Kinesis and Cassandra) through techniques such as Event Sourcing[12] and Command Query Responsibility Segregation (CQRS).

There are benefits to reap from decentralized data management and persistence—sometimes called *Polyglot Persistence*. Conceptually, which storage medium is used does not really matter; what matters is that a service can be treated as a single unit—including its state and behavior—and in order to do that each service needs to own its state, exclusively. This includes not allowing one service to call directly into the persistent storage of another service, but only

9 Visit Martin Fowler's website For more information on how to use the Bounded Context and Ubiquitous Language modeling tools.

10 Domain-Driven Design (DDD) was introduced by Eric Evans in his book *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Addison-Wesley Professional).

11 See Jay Kreps' epic article "The Log: What every software engineer should know about real-time data's unifying abstraction."

12 Martin Fowler has done a couple of good write-ups on Event Sourcing and CQRS.

through its API—something that might be hard to enforce programmatically and therefore needs to be done using conventions, policies and code reviews.

An *Event Log* is a durable storage for the messages. We can either choose to store the messages as they enter the service from the outside, the *Commands* to the service, in what is commonly called called *Command Sourcing*. We can also choose to ignore the Command, let it perform its side-effect to the service, and if the side effect triggers a state change in the service then we can capture the state change as a new fact in an *Event* to be stored in the Event Log using *Event Sourcing*.

The messages are stored in order, providing the full history of all the interactions with the service and since messages 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 time) for resilience, debugging and replication—instead of having it abstracted away from the user as seen in RDBMSs. Pat Helland puts it very well:

> "Transaction logs record all the changes made to the database. High-speed appends are the only way to change the log. From this perspective, the contents of the database hold a caching of the latest record values in the logs. The truth is the log. The database is a cache of a subset of the log. That cached subset happens to be the latest value of each record and index value from the log."[13]

Command Sourcing and Event Sourcing have very different semantics. For example, replaying the Commands means that you are also replaying the side effects they represent; replaying the Events only performs the state-changing operations, bringing the service up to speed in terms of state. Deciding the most appropriate technique depends on the use case.

Using an Event Log also avoids the Object-Relational Impedance Mismatch, a problem that occurs when using Object-Relational Mapping (ORM) techniques and instead builds on the foundation of message-passing and the fact that it is already there as the primary communication mechanism. Using an Event Log is often the best

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

persistence model for Microservices due to its natural fit with Asynchronous Message-Passing (see Figure 2-4).

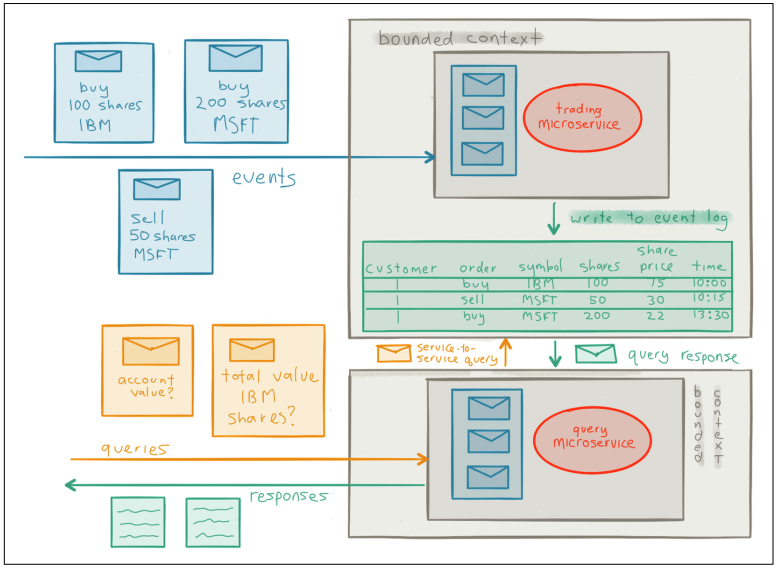

Figure 2-4. Event-based persistence through Event Logging and CQRS

Embrace Asynchronous Message-Passing

> 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

Communication *between* Microservices needs to be based on Asynchronous Message-Passing (while the logic *inside* each Microservice is performed in a synchronous fashion). As was mentioned earlier, an asynchronous boundary between services is necessary in order to decouple them, and their communication flow, in *time*—allowing concurrency—and in *space*—allowing distribution and mobility. Without this decoupling it is impossible to reach the level of compartmentalization and containment needed for isolation and resilience.

Asynchronous and non-blocking execution and IO is often more cost-efficient through more efficient use of resources. It helps minimizing 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. Let's say that each requests takes 100 milliseconds. If it needs to execute these in a synchronous sequential fashion the total processing time will be roughly 1 second (Figure 2-5).

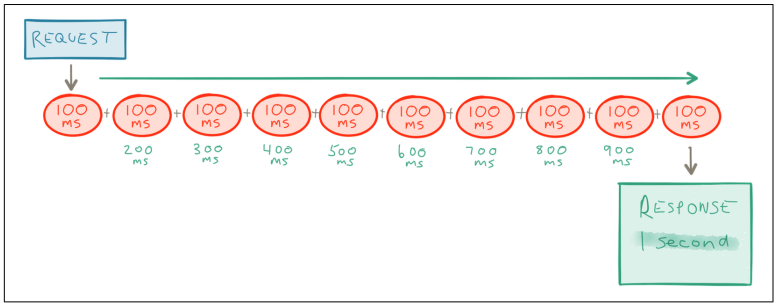

Figure 2-5. Synchronous requests increase latency

Whereas if it is able to execute them all asynchronously the processing time will just be 100 milliseconds—an order of magnitude difference for the client that made the initial request (Figure 2-6).

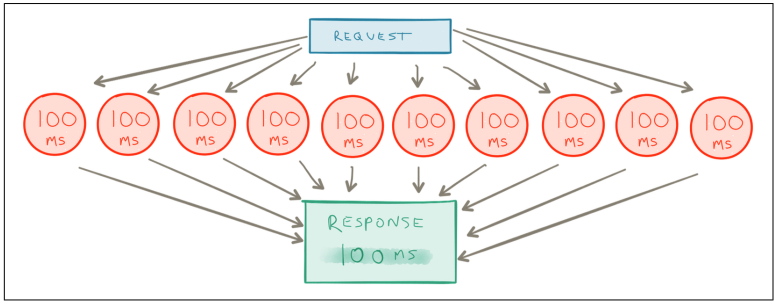

Figure 2-6. Asynchronous requests execute as fast as the slowest request

But why is blocking so bad?

It's best illustrated with an example. 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. If the service instead performs the call in an asynchronous and non-blocking fashion, it frees up the underlying thread to be used by someone else while waiting for the result to be returned. This leads to much more efficient usage—in terms of cost, energy and performance—of the underlying resources (Figure 2-7).

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. In order to reap the full benefits of non-blocking execution all parts in a request chain needs to participate—from the request dispatch, through the service implementation, down to the database and back.

Asynchronous message-passing helps making the constraints—in particular the failure scenarios—of network programming first-class, instead of hiding them behind a leaky abstraction[14] and pretending that they don't exist—as seen in the fallacies[15] of synchronous Remote Procedure Calls (RPC).

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

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

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

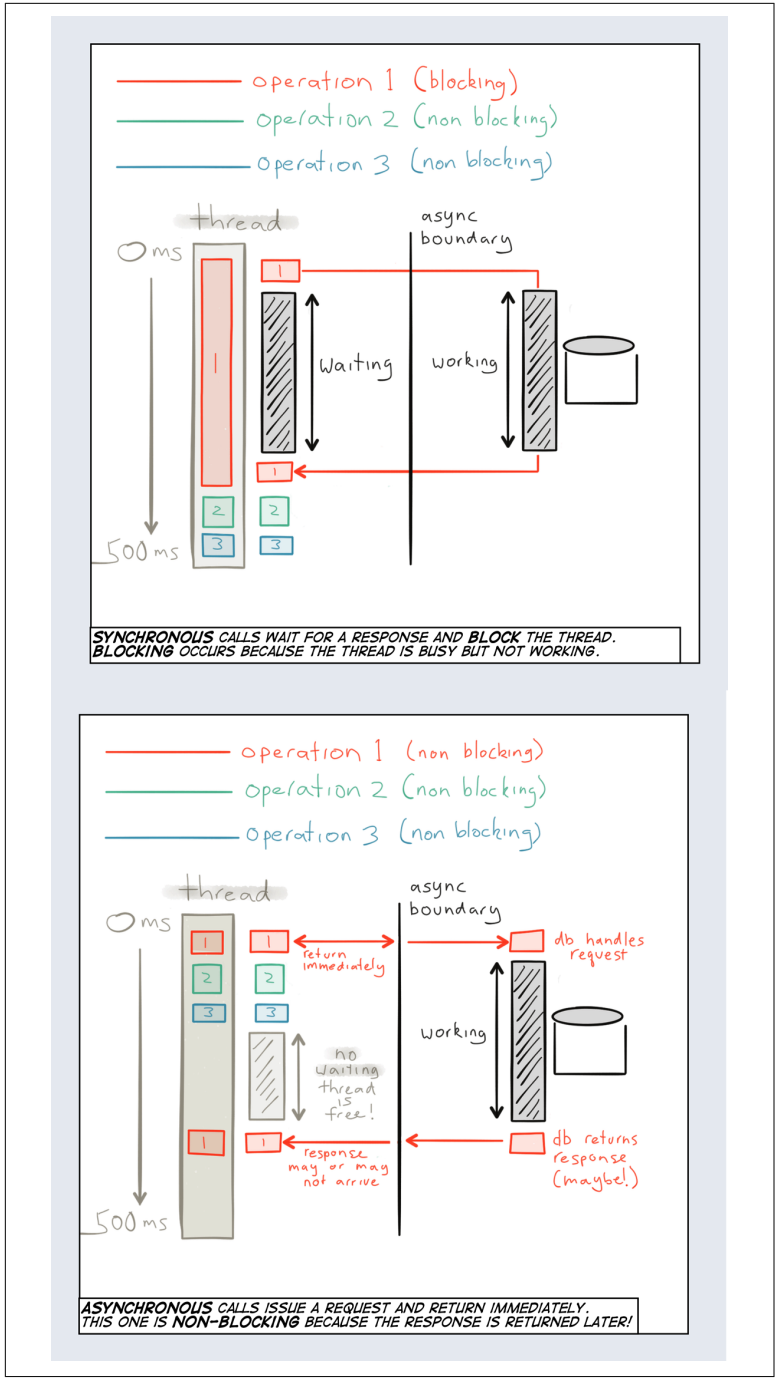

Figure 2-7. Why blocking is bad

It is unfortunate that REST is widely considered as the default Microservice communication protocol. It's important to understand that REST is most often *synchronous*[16] which makes it a *very unfitting* default protocol for inter-service communication. REST might be a reasonable option when there will only ever be a handful of services, or in situations between specific tightly coupled services. But use it sparingly, outside the regular request/response cycle, knowing that it is always at the expense of decoupling, system evolution, scale and availability.[17]

The need for asynchronous message-passing does not only include responding to individual messages or requests, but also to continuous streams of messages, 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.[18]

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 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.

The first wave of big data was "data at rest." We stored massive amounts in 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 subseconds 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

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

17 See the *Integration* section in Chapter 3 for a discussion on how to interface with clients that assumes a synchronous protocol.

18 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."

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 for most use cases and data sizes, move away from the traditional batch-oriented architecture altogether towards pure stream processing architecture.

This is the model that is most interesting to Microservices-based architectures because it gives us a way to bring the power of streaming and "data in motion" into the services themselves—both as a communication protocol as well as a persistence solution (through Event Logging, as discussed in the previous section)—including both client-to-service and service-to-service communication.

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 efficiently manage hardware resources. The problem is that none of this matters if our Microservices and its underlying platform cannot make efficient use of it. What we need are services that are mobile, allowing them to be elastic.

We have talked about asynchronous message-passing, and that it provides decoupling in time and space. The latter, decoupling in space, is what we call *Location Transparency*,[19] the ability to, at runtime, dynamically scale the Microservice—either on multiple cores

19 Location Transparency is an extremely important but very often ignored and underappreciated principle. The best definition of it can be found in the glossary of the Reactive Manifesto—which also puts it in context: *http://www.reactivemanifesto.org/glossary#Location-Transparency*.

or on multiple nodes—without changing the code. This is service distribution that enables elasticity and mobility; it is needed to take full advantage of cloud computing and its pay-as-you-go models.

For a service to become location transparent it needs to be addressable. But 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 if the service is running, has been stopped, is suspended, is being upgraded, has crashed, and so on. The address should always work (Figure 2-8). This means a client can always send messages to an address. In practice they might sometimes be queued up, resubmitted, delegated, logged, or sent to a dead letter queue.

Second, an address needs to be *virtual* in the sense that it can, and often does, represent not just one, but a whole set of runtime instances that together defines the service. Reasons this can be advantageous include:

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

Active-Passive[20] state replication between instances of a stateful service
: If a service is stateful then 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.

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

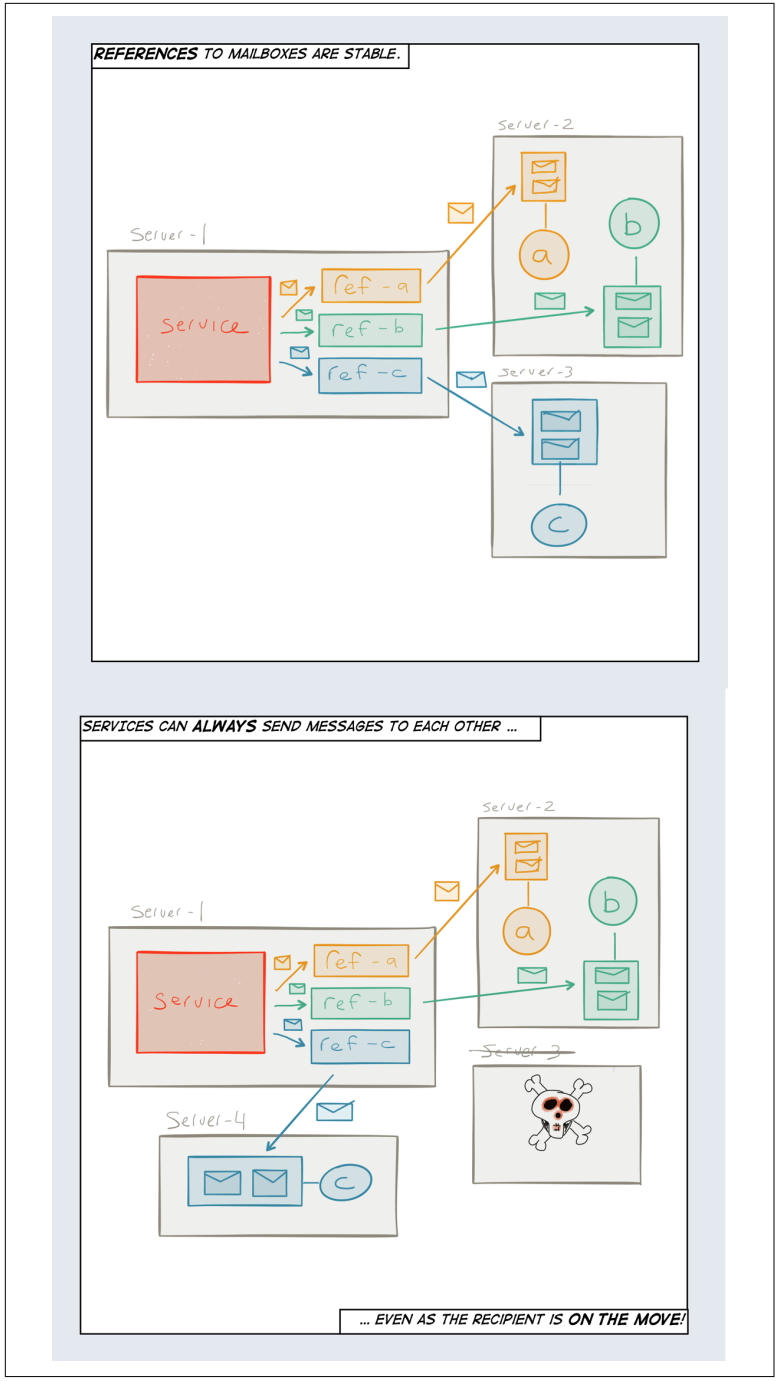

Figure 2-8. Virtual addressing allows seamless fail-over

Relocation of a stateful service

: It can be beneficial to move a service instance from one location to another in order to improve locality of reference[21] or resource efficiency.

Using virtual addresses 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 have to care about how and where the service is currently configured to operate.

21 Locality of Reference is an important technique in building highly performant systems. There are two types of reference locality: temporal, reuse of specific data; and spatial, keeping data relatively close in space. It is important to understand and optimize for both.

CHAPTER 3
Microservices Come in Systems

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

> No man is an island,
> Entire of itself,
> Every man is a piece of the continent,
> A part of the main.
>
> —John Donne

Now we have a pretty good understanding of what characterizes a Reactive Microservice. However, learning from Carl Hewitt: *one Microservice is no Microservice—they come in systems*. Like humans they act autonomously and therefore need to communicate and collaborate with others to solve problems—and as with humans, it is in collaboration that both the most interesting opportunities and challenging problems arise.

Individual Microservices are comparatively easy to design and implement—what is hard in a Microservices-based Architecture is all the things around them: discovery, coordination, security, replication, data consistency, failover, deployment, and integration with other systems, just to name a few.

Systems Need to Exploit Reality

> If you cannot solve a problem without programming.
> You cannot solve a problem with programming.
>
> —Klang's Conjecture by Viktor Klang

One of the major benefits of Microservices-based Architecture is that it gives you a set of tools to exploit reality, to create systems that closely mimic how the world works, including all its constraints and opportunities.

We have already discussed—highlighted by Conway's Law—how Microservices development is often a better fit to how your engineering organization and departments already work.

Another subtle, and more important fact to embrace, is that reality is not consistent—there is no single absolute present—everything is relative, including time and our experience of now.[1]

Information cannot travel faster than the speed of light, and most often travels considerably slower, which means that communication of information has latency.[2] Information is always from the past, and when you think of it, it holds true for everything we observe. When we observe or learn about an effect, it has already happened, not uncommonly quite some time ago—we are always looking into the past. "Now" is in the eye of the beholder.[3]

Here the Microservice can become an escape route from reality. Within each Microservice, we can live on a safe island of determinism and strong consistency (see ACID)—an island where we can

1 As Albert Einstein proved in his 1905 paper "On the Electrodynamics of Moving Bodies."

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

3 For a discussion of the relativity of time, the present and the past in distributed systems, and how we need change the way we model systems so that we can break free from the illusion that we live in a globally consistent now, see my talk "Life Beyond the Illusion of Present" YouTube video, 53:54, from a presentation at Voxxed Days, posted March 6, 2016. You should also take time to read Justin Sheehy's great article on the topic "There is No Now."

live happily under the illusion that time and the present is absolute (see Figure 3-1).

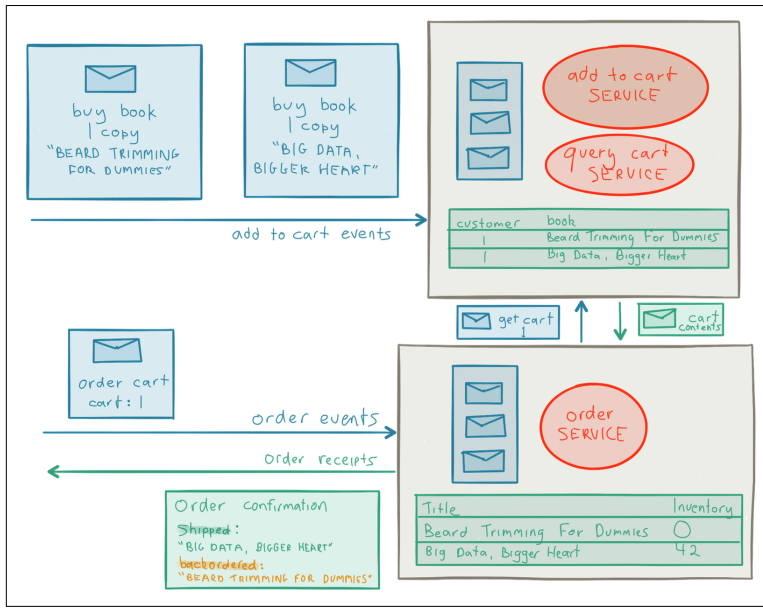

Figure 3-1. Employing Microservices means embracing eventual consistency

However, as soon as we exit the boundary of the Microservice we enter a wild ocean of non-determinism—the world of distributed systems, which is a very different world. You have probably heard that building distributed systems is hard.[4] It is. That being said it is also the world that gives us solutions for resilience, elasticity, isolation amongst others. At this point, what we need to do is not to run back to the monolith, but instead learn how to apply and use the right set of principles, abstractions and tools in order to manage it.

Pat Helland talks about this[5] as "data on the inside" versus "data on the outside:" in which inside data is "our current local present," out-

4 If you have not experienced this first-hand then I suggest that you spend some time thinking through the implications of L Peter Deutsch's Fallacies of distributed computing.

5 Pat Helland's paper is essential reading for anyone building Microservices-based systems "Data on the Outside versus Data on the Inside."

side data—the events—the "blast from the past," and commands between services "hope for the future."

> 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

Let's imagine that we have created a bunch of Microservices and now is the time to make them all work together as a system—what are some of the most important things we need to understand and do?

Service Discovery

> The greatest obstacle to discovery is not ignorance—it is the illusion of knowledge.
>
> —Daniel J. Boorstin

So now I have a set of Microservices that needs to collaborate. How can I help them to locate each other?

In order to communicate with another service, a service needs to know the other service's address. The simplest solution 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 this solution is that it forces a static deployment model which contradicts everything we are trying to accomplish with Microservices.

They need to stay decoupled and mobile, and the system needs to be elastic and dynamic. This can be addressed by adding a level of indirection using a pattern called *Inversion of Control* (IoC). 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 a Microservices-based Platform.

Once the information about each service has been stored it can be made available through a *Service Registry* that services can use to look the information up—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 (injected into the services using Dependency Injection)—using a pattern called Server-Side Service Discovery.

What do I need to consider when choosing a service discovery tool?

One way of storing service information is through a CP-based[6] (strongly consistent) configuration storage.[7] This is simple in one way, since you have all information in one place in an atomic fashion. But it often gives you much stronger consistency guarantees than needed,[8] at the expense of availability and an additional infrastructure cluster that needs to be understood and managed.

It is often better to rely on distributed AP-based[9] Peer-to-Peer technologies that use techniques like Epidemic Gossip sometimes together with Conflict-Free Replicated Data Types (CRDTs)[10] to disseminate the information in a simpler, eventually consistent and resilient way—without the need for additional infrastructure.[11]

6 CP refers to Consistency and Partition Tolerance in the CAP Theorem, and means that a system chooses Consistency over Availability in the face of network partitions.

7 Examples of CP-based service discovery systems include ZooKeeper, and etcd.

8 The nanosecond after you "know" a service's location, that location might have changed. So what was the gain in having "strong" consistency for that information?

9 AP refers to Availability and Partition Tolerance in the CAP Theorem, and means that a system chooses Availability over Consistency in the face of network partitions.

10 CRDTs is one of the most interesting ideas coming out of distributed systems research in recent years, giving us rich, eventually consistent, composable data-structures that are guaranteed to converge consistently without the need for coordination. For more information see "A comprehensive study of Convergent and Commutative Replicated Data Types."

11 Examples of AP-based service discovery systems include Lightbend Reactive Platform, Netflix Eureka, Serf, and regular DNS.

API Management

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

What are the challenges in managing service protocols and APIs when they evolve independently over time?

Individual Microservices are only independent and decoupled if they can evolve independently. This requires their data and protocols to be resilient to and permissive of change—for persistently stored data as well as for exchange of ephemeral information.[12] In particular, the interoperability of different versions is crucial to enable the long-term management of complex service landscapes.

Postel's Law,[13] 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.[14]

Challenges include versioning of the protocol and data and how to handle upgrades and downgrades of the protocol and data. This is a non-trivial problem that includes picking extensible codecs for serialization, maintaining a protocol and data translation layer and sometimes even versioning the service itself.[15] This is what is called an *Anti-Corruption Layer* in DDD, and can be added to the service itself or done in an API Gateway.

Let's say I have a client that, in order to perform a task, needs to talk to 10 different services, each with a different API. That sounds complicated. How can I simplify the API management?

This is common scenario in large Microservices-based systems and can lead to unnecessary complexity on the client side. In these situations it is often better to, instead of having the client communicating

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

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

14 It has among other things influenced the Tolerant Reader Pattern.

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

directly with each Microservice, let it talk to an API Gateway service.[16] See Figure 3-2.

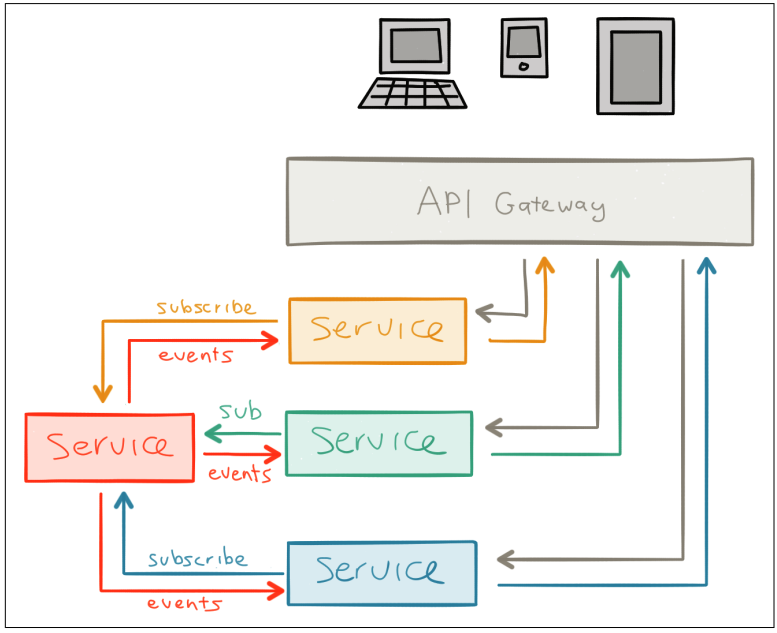

Figure 3-2. Simplifying client access with an API Gateway

The API Gateway is responsible for receiving the request from the client, routing it to the right set of services—doing protocol translations if necessary—composing the replies, and returning it to the client.

The benefits of this pattern include simplifying the client-to-service protocol by encapsulating the service's internal structure and their APIs. It is challenging to achieve this in a highly available and scalable way using a centralized solution. Instead, use a decentralized technology, as mentioned in Service Discovery.

But this is—as with all these core infrastructure services—not something that you should build yourself but ideally get as part of the underlying platform.

16 The API Gateway pattern has been used successfully by Netflix and Amazon API Gateway amongst others.

Managing Communication Patterns

> 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

How can I keep the complexity of communication patterns between Microservices in a large system under control?

The role of the ESB still has its place—now in the form of a modern scalable message queue.

In systems with a handful of Microservices, direct *Point-to-Point* communication gets the job done. However, once you go beyond that, allowing each one of them to just talk directly with everyone it pleases can quickly turn the architecture into an incomprehensible mess of chatter. Time to introduce some rules of engagement! What is needed is a logical decoupling of the sender and receiver and a way of routing data between the parties according to predefined rules.

One solution is to use a *Publish-Subscribe* mechanism, in which the publisher can publish information to a topic where the subscriber can listen in. This can be solved by using a scalable messaging system (for example Kafka or Amazon Kinesis) or a NoSQL database (preferably into an AP-style database like Cassandra or Riak).

In the SOA world, this role was usually played by the ESB. However, in this case we are not using it to bridge monoliths, but rather as a backbone publishing system for the services to use for broadcasting work or data, or as an integration and communication bus between systems (for example for ingesting data into Spark through Spark Streaming).

Sometimes using a Publish-Subscribe protocol is insufficient—for example, when you need more advanced routing capabilities that allow the programmer to define custom routing rules involving multiple parties, or when used in stages of data transformation, enrichment, splitting, and merging (for example, using Akka Streams or Apache Camel). See Figure 3-3.

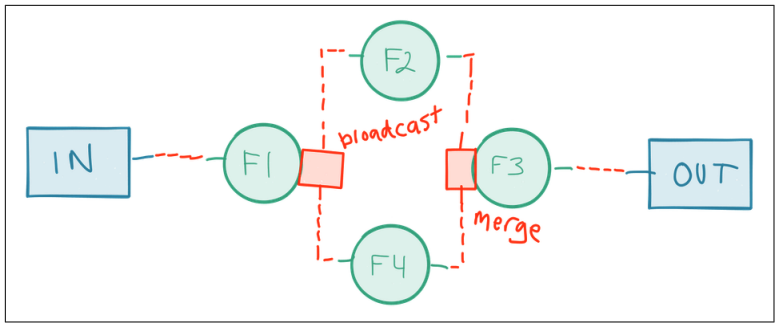

Figure 3-3. Routing and transformation of data streams

Integration

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

What about integrating multiple systems?

Most systems need a way of communicating with the outside world, either consuming and/or producing information from/to other systems.

When communicating with an external system, especially one that you have no control over, you are putting yourself at risk. You can never be sure how the other system will behave when the communication diverge from the "happy path"—when things start to fail, when the system is overloaded, and so on. You can't even trust that the other service will behave according to the established protocol. So you can see why it's important to take precautions to stay safe.

The first step is to agree on a protocol that minimizes the risk of having one system overloading another during unexpected load increase. If synchronous communication is used—even if it is only for a subset of the protocol—you are introducing strong coupling and are putting yourself in the hands and mercy of the other system.

Avoiding cascading failures requires services that are fully decoupled and isolated. This is best achieved using a fully asynchronous protocol of communication. It is equally important that the protocol includes a mechanism for agreeing on the velocity of the flow of data by applying what is called *back-pressure*, which ensures a fast system can't overload its slower counterpart. More and more tools and libraries are starting to embrace the Reactive Streams specifica-

tion (Reactive Streams-compatible products include Akka Streams, RxJava, Spark Streaming, and Cassandra drivers)—which will make it possible to bridge systems using fully asynchronous back-pressured real-time streaming—improving interoperability, reliability and performance of the composed system as a whole.

It is also crucial to have a way of managing faulty services; by capturing failures, you can retry tasks and, if the failure persists, quarantine the service for a specific period of time while waiting for the service to recover—which is abstracted away in the Circuit Breaker pattern[17] (production-grade Circuit Breaker implementations can be found in Netflix Hystrix and Akka). See Figure 3-4.

The role of system integration has historically fallen on passing around *flat files*, or relying on centralized services like an RDBMS or an ESB. But the increasing need for scale, throughput and availability has led many systems to adopt decentralized strategies for integration (for example HTTP-based REST services and ZeroMQ) or modern, centralized, scalable and resilient Pub-Sub systems (like Kafka and Amazon Kinesis).

Recent trends include using Event Streaming Platforms for system integration, bringing in ideas from Fast Data and real-time data management.

What about client to service communication—should that also be asynchronous?

Throughout this paper we have emphasized the need for asynchronous communication, execution, and IO. Relying on asynchronous message-passing is quite straightforward between services, where we have full control of the communication protocol and its implementation. However, when communicating with external clients we don't always have that luxury. Many clients—browsers, apps, and so on—assume synchronous communication, and in situations like this using REST is often a good choice.

17 The Circuit Breaker pattern is important in Microservices-based systems. Read about it more in Martin Fowler's "CircuitBreaker."

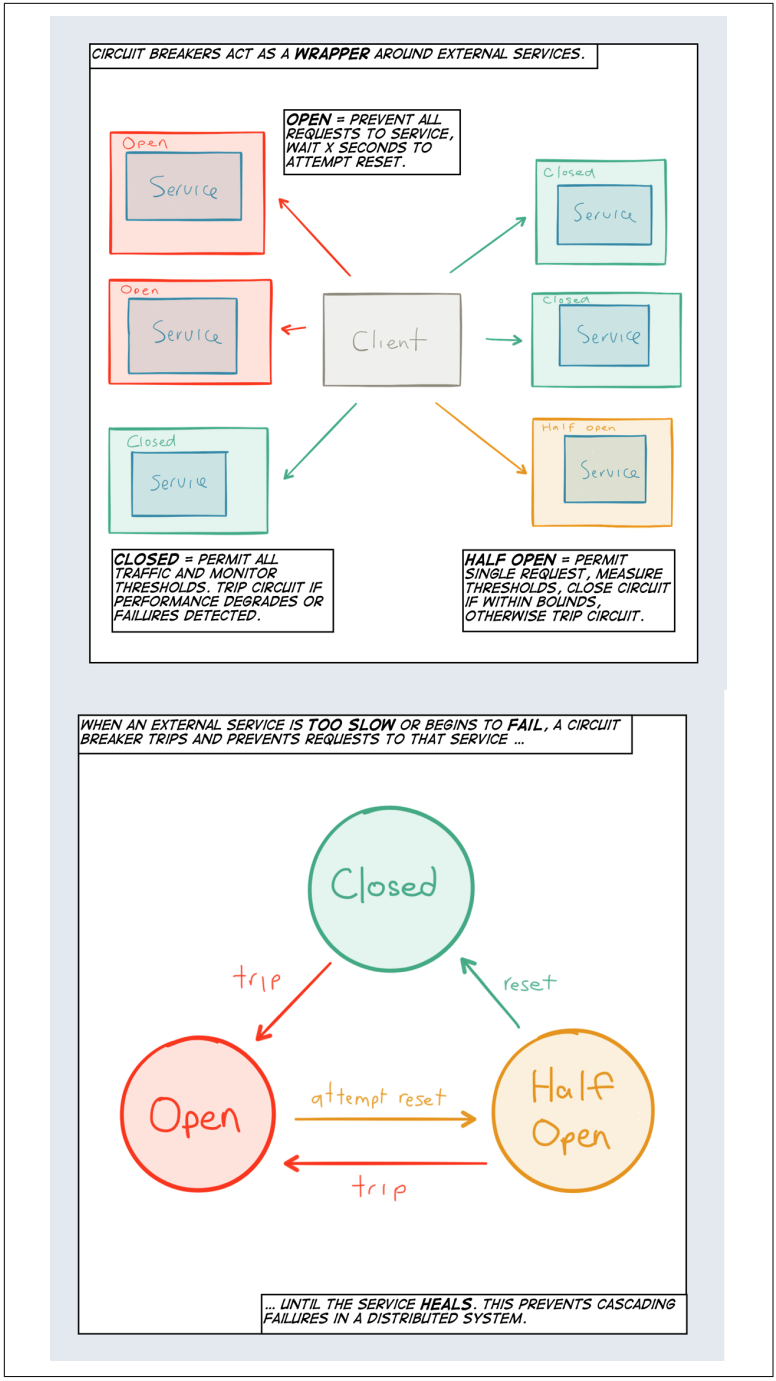

Figure 3-4. Managing faulty services with the Circuit Breaker pattern

What is important is to not go all in on synchronous client communication, but to think through and assess each client and use-case individually.[18] There are many situations where developers gravitate to a synchronous solution out of habit, instead of using where it really matters, where it can simplify things and improve interoperability.

Examples of use-cases that are inherently asynchronous but traditionally modeled as synchronous include: information on whether an item is in stock—if it's hot and stock is running out the user usually wants to be notified; the current specials at a restaurant—if they change, the user may want to know immediately; user comments on websites—often end up being a real time conversation; and ads—may respond and change depending on how a user is using the page.

We need to look at each use-case individually to understand what is the most natural way to express the communication between the client and the service. This often requires looking at the data integrity constraints to find opportunities to weaken the consistency (ordering) guarantees—relying on techniques like causality and read-your-writes[19]—with the goal of finding the minimal set of coordination constraints that gives the user intuitive semantics: the goal of finding best strategy to exploit reality.

Security Management

> The user's going to pick dancing pigs over security every time.
> Security is a not a product, but a process.
>
> —Bruce Schneier

If someone asks us to make sure that not every service can call the Billing service, what do we do then?

It is important to distinguish between authentication and authorization. *Authentication* is the process of making sure that a client (human or service) is who she says she is (typically using a username and a password), while *authorization* is the process of allowing or denying the user to access a specific resource.

---

18 Defining the procedure for assessing use-cases is outside of the scope of this paper.

19 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.

Both are important to get right and need to work in concert. There are many ways to make this work, each way with their own benefits and drawbacks.

TLS Client Certificates, also called Mutual Authentication or Two Way Authentication, can provide a very robust security solution for inter-service authentication in which each service is given a unique private key and certificate on deployment. In this strategy, it is not only the server that is verifying the identity of the client, but the client verifying the identity of the server. This means it's safe not only from eavesdropping, but from a completely hostile network where an attacker could potentially intercept and redirect requests—such as the Internet itself (see Figure 3-5). Communication over SSL is safe from eavesdropping and on an open, well understood standard. It is, however, complicated to manage, and benefits from support by the underlying platform.

If the services are HTTP-based, they can make use of HTTPS Basic Authentication. It is well understood and straightforward, but it can be complicated to manage SSL certificates on all the machines and the requests can no longer be cached by a reverse proxy.

One advantage is that it provides Two Way Authentication similar to the Client Certificate solution, where client verifies the identity of the server using the server's certificate before it sends the credentials, and the server verifies the identify of the client using the credentials it sends.

Another approach is to use *Asymmetric Request Signing*. In this solution, each service is given its own private key to sign requests with, while the public keys for each service are made known to the Service Discovery service. The drawback is that as a proprietary solution, it can be vulnerable to eavesdropping or request replay attacks if your network has been compromised.

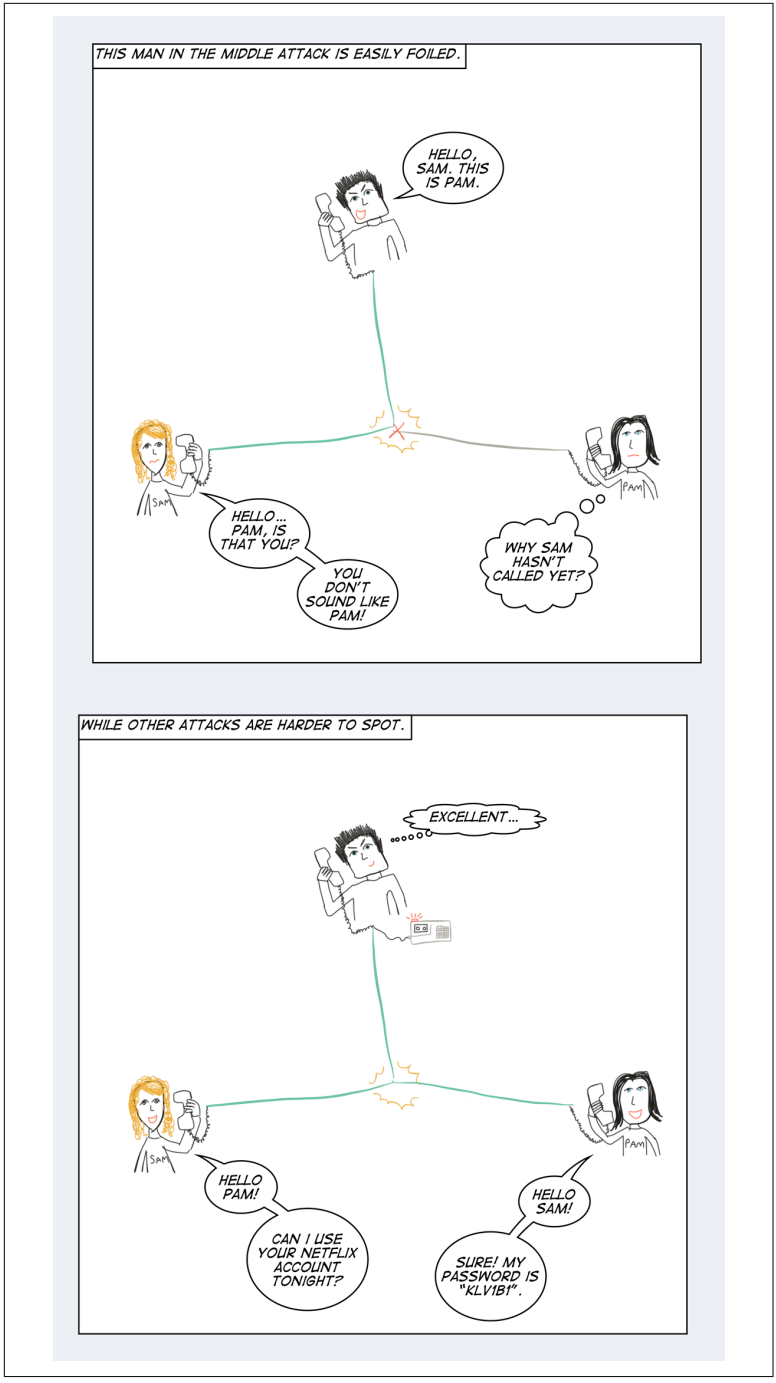

Figure 3-5. Man in the middle attack

Finally, basing the security on a *Shared Secret*, either using Hash Message Authentication Code (HMAC) signing of the request or a secret token that is shared at deployment time. This solution is conceptually simple but can be hard to implement since each service pair that talk to each other need a unique shared secret, making the number of shared secrets needed for the system the permutation of all services that talk to each other.

Minimizing Data Coupling

> Silence is not only golden, it is seldom misquoted.
>
> —Bob Monkhouse

We have been spoiled by the monolith talking to a centralized RDBMS for too long—assuming that the world can always be shoehorned into a strongly consistent (see ACID) model. But strong consistency requires coordination, which is extremely expensive in a distributed system, and puts an upper bound on scalability, throughput, low latency and availability.

The need for coordination—adding to the costs of contention and coherency, as defined in the Universal Scalability Law—means that individual services can't make progress individually but has to wait for consensus. When designing Microservices-based systems we should therefore strive to minimize the service-to-service coordination of state, to allow the Microservices to *comfortably share silence*.[20]

How can I design individual Microservices that ensure minimal coordination of state?

Traditionally, developers have been used to a monolithic architecture hooked up to a single SQL database—giving a single "global" unit of consistency. This model feels simple because it gives the illusion of one globally consistent "now," one absolute present—which is easy to reason about intuitively. But as we have discussed, breaking free from this illusion and splitting up the monolith into discrete isolated Microservices has a lot of benefits.

20 As the character Mia Wallace stated in Quentin Tarantino's movie *Pulp Fiction*, and Peter Bailis' later used in his excellent talk "Silence is Golden: Coordination-Avoiding System Design."

You have to start by looking at the data and work with a domain expert to understand its relationships, guarantees and integrity constraints from a business perspective, exploiting reality.

This often includes denormalizing the data. Continue by defining the consistency (transactional) boundaries in the system, within which you can rely on strong consistency. Then you should let these boundaries drive the design and scoping of the Microservices. If you design your services with data dependencies and relationships in mind it is possible to reduce, and sometimes completely eliminate, the coupling of data—which means that you do not have to coordinate the changes to it.

Minimizing the Cost of Coordination

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

What do I do if I have designed Microservices with minimal data coupling, but still have use cases where I need to coordinate data between them?

This is to be expected, and not a failure in the design. Many systems built with Microservices have use cases that need to coordinate data. Fortunately you are now in a position where you can add coordination as needed, instead of starting with coupling and trying to remove it—which is so much harder.

There are reasonable ways of coordinating data changes in an scalable and resilient fashion, but it requires that your operations on the data are composable.

Composability in this context means that changes to data can be made available to other services without stalling them (or yourself), without waiting on coordination to take place. Let's spend the next paragraphs discussing how this can be addressed using communication protocols that embrace techniques such as Apology-Oriented Programming, Event-Driven Architecture and ACID 2.0.

The idea of Apology-Oriented Programming[21] is built around the idea that it is easier to ask for forgiveness than permission. If you can't coordinate (and be sure about something), then take an educated guess, a bet that a condition will hold, and if you were wrong, apologize and perform a compensating action.

This approach matches reality very well. It's how humans collaborate all the time. Other examples include ATMs—allowing you to withdraw money in the case of network disconnect, and then later charging your account—and how airlines are overbooking flights—and then bribe themselves out of the problem through vouchers.

This model works very well with an Event-Driven Architecture that leverages asynchronous message-passing and Event Sourcing. In this model it is very important to distinguish between *Commands* and *Events*, where Commands represent the intent to perform a side-effecting operation—what Pat Helland calls "hope for the future"—and Events represent the fact that something has already happened—the history leading up to the current local present.

Queries are best performed using the CQRS pattern, where the write side—persisted as Events in the Event Log—is separated from the read side—stored in a rich schema format using a RDBMS or NoSQL database with great support for queries. Using an Event Log for state management and persistence has many other benefits, such as simplified auditing, debugging, replication and failover, allowing you to replay the event stream at any point, from any point in the past.

The term ACID 2.0 was coined[22] by Pat Helland and is a summary of a set of principles for scalable and resilient protocol and API 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

21 Pat Helland did not use the term Apology-Oriented Programming but introduced the general idea behind it in his blog post "Memories, Guesses, and Apologies."

22 Another excellent paper by Pat Helland, where he introduced the idea of ACID 2.0, is "Building on Quicksand."

duplication of messages does not matter. The "D" could stand for Distributed, but is probably included just to make the ACID acronym work.

One tool that embraces these ideas is CRDTs, as they are eventually consistent, rich data-structures (including counters, sets, maps and even graphs) that compose, and that converge without coordination. The ordering of the updates does not matter, and can always be automatically merged safely. CRDTs are fairly recent, but have been hardened in production for quite some years, and there are production-grade libraries that you can leverage directly (for example in Akka and Riak).

However, relying on eventual consistency is sometimes not permissible, since it can force us to give up too much of the high-level business semantics. If that is the case then 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[23] to be the best we can do in an always available system). Causal consistency is usually implemented using logical time[24] and is available in many NoSQL databases, Event Logging and Distributed Event Streaming products (products allowing use of logical time to implement causal consistency include Riak and Red Bull's Eventuate).

23 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."

24 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 etc. Instead, rely on logical time, which gives you a stable notion of time that you can trust, even if nodes fail, messages drop etc. There are several good options available, one is Vector Clock.

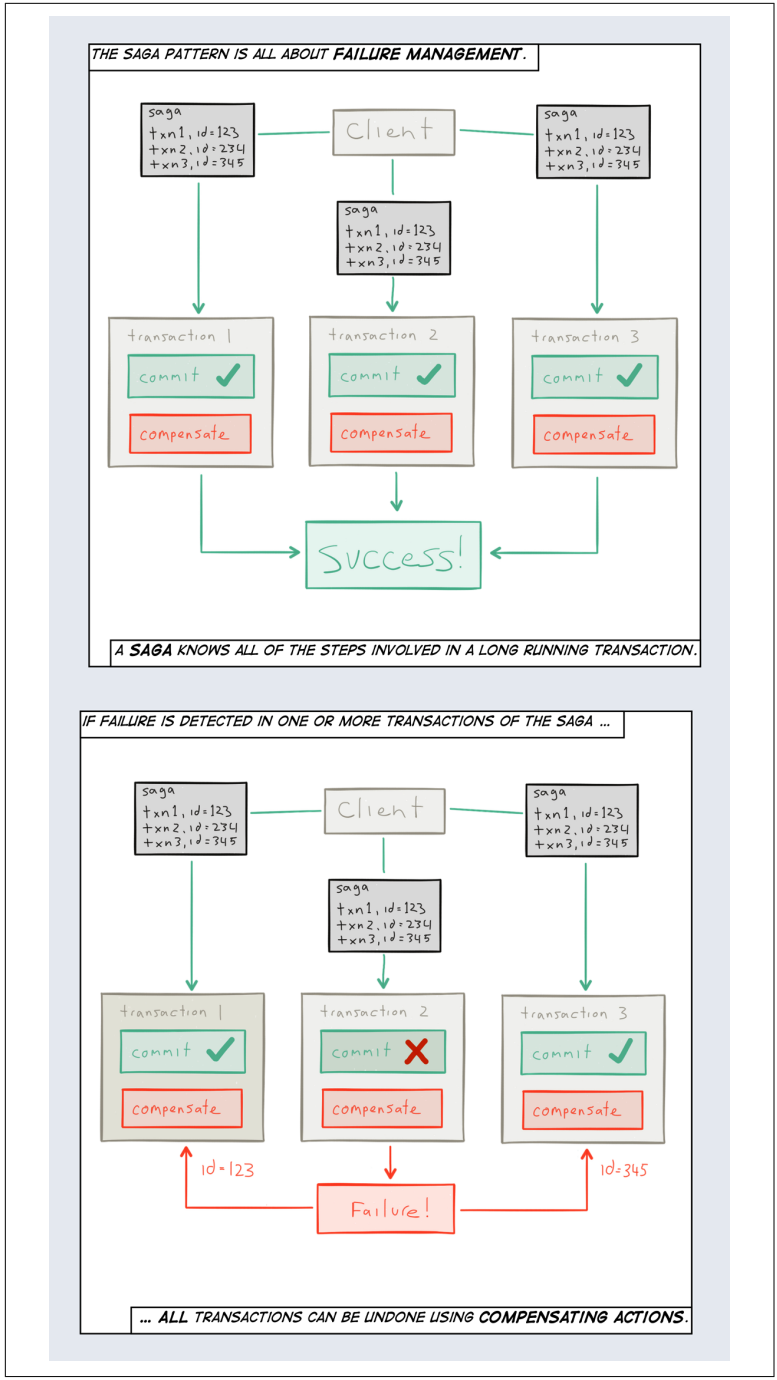

Figure 3-6. Resilient transaction management with the SAGA pattern

But what about RDBMSs? You can actually get pretty far using SQL as well. In one of his papers,[25] Peter Bailis talks about coordination-avoidance in RDBMSs and shows how many standard SQL operations can be made without coordinating the changes—i.e., without transactions. The list of operations includes: Equality, Unique ID Generation, Greater than Increment, Less than Decrement, Foreign Key Insert and Delete, Secondary Indexing and Materialized Views.

What about transactions? Don't I need transactions?

> In general, application developers simply do not implement large scalable applications assuming distributed transactions.[26]
>
> —Pat Helland

Historically, *distributed transactions*[27] have been used to coordinate changes across a distributed system. They do their job of simplifying the experience of concurrent execution well, by providing the illusion that you are the only person in the world using the data, or that everyone else is just sitting back and letting you perform your changes for as long as you wish. This is not true, and upholding this illusion is extremely costly, making systems slow, unscalable, and brittle.

The *Saga Pattern*[28] is a scalable and resilient alternative to distributed transactions (Figure 3-6). It is a way to manage long-running business transactions based on the discovery that long-running business transactions often comprise multiple transactional steps in which overall consistency of the whole transaction can be achieved by grouping these steps into an overall distributed transaction. The technique is to pair every stage's transaction with a compensating reversing transaction, so that the whole distributed transaction can be reversed (in reverse order) if one of the stage's transactions fails.

---

25 Peter Bailis is an assistant professor at Stanford and one of the leading experts on distributed and database systems in the world. The paper referenced is "Coordination Avoidance in Database Systems."

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

27 The golden standard is X/Open Distributed Transaction Processing, most often referred to as XA.

28 Originally defined in the 1987 paper "SAGAS" by Hector Garcia-Molina and Kenneth Salem.

It might come as a surprise to some people, but 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[29] that we could for example keep using Read Committed, Read Uncommitted, and Read Your Writes while we have to give up on Serializable, Snapshot Isolation, and Repeatable Read. This is recent research but something I believe more SQL and NoSQL databases to start taking advantage of in the near future.

Summary

When designing individual Reactive Microservices, it is important to adhere to the core traits of Isolation, Single Responsibility, Autonomy, Exclusive State, Asynchronous Message-Passing and Mobility. What's more, Microservices are collaborative in nature and only make sense as systems. It is *in between* the Microservices that the most interesting, rewarding, and challenging things take place, and learning from past failures[30] and successes[31] in distributed systems and collaborative services-based architectures is paramount. What we need is comprehensive Microservices platforms that provide the heavy lifting for distributed systems, and offer essential services and patterns built on a solid foundation of the Reactive principles.

29 For more information see the paper "Highly Available Transactions: Virtues and Limitations" by Peter Bailis et. al.

30 The failures of SOA, CORBA, EJB and synchronous RPC are well worth studying and understanding.

31 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.

About the Author

Jonas Bonér is co-Founder and CTO of Lightbend, inventor of the Akka project, co-author of the Reactive Manifesto and a Java Champion. Learn more about his work at jonasboner.com or follow him on Twitter at @jboner.