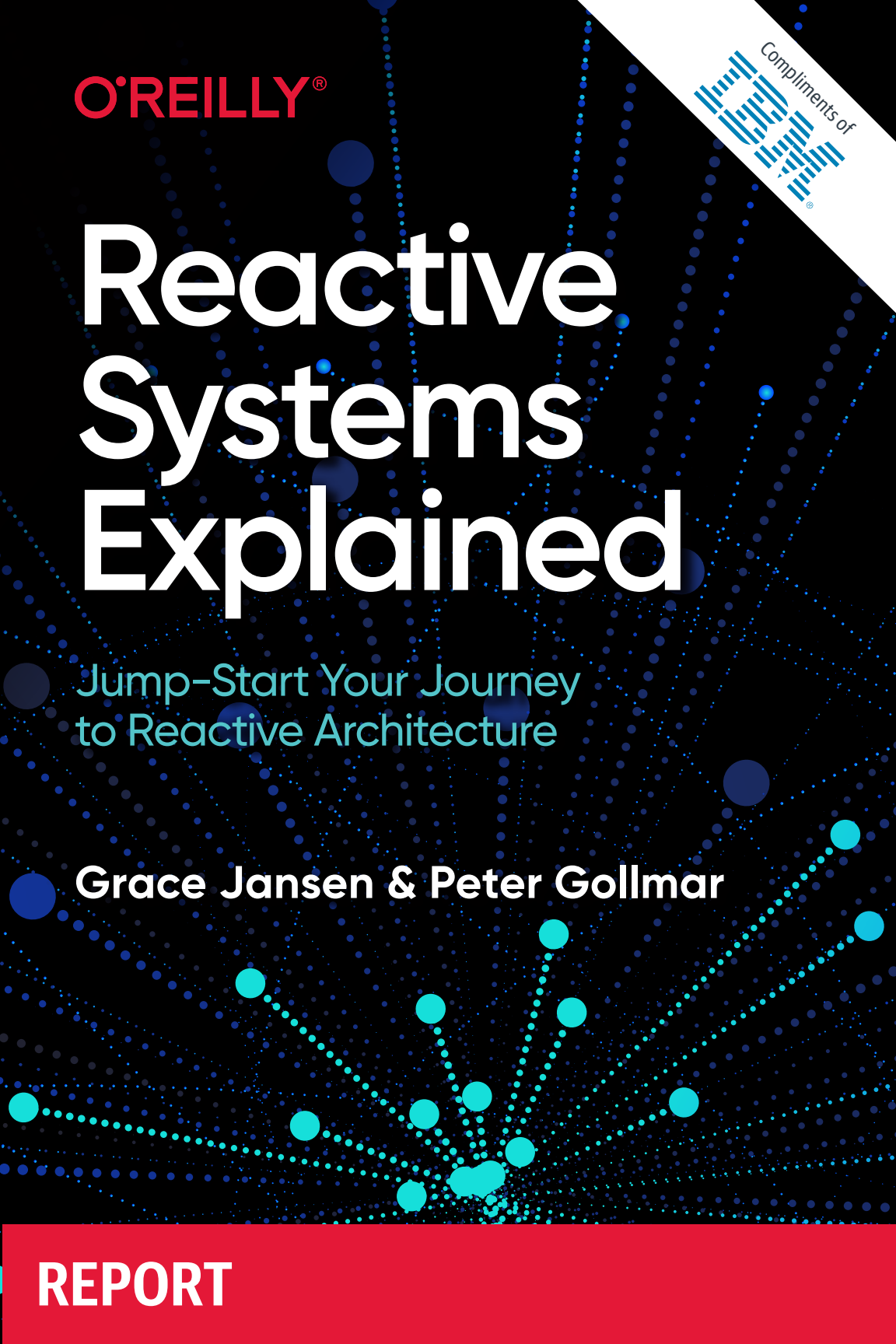
O'REILLY®
Compliments of
IBM
Reactive Systems Explained
Jump-Start Your Journey to Reactive Architecture
Grace Jansen & Peter Gollmar
REPORT

Reactive Systems Explained

Jump-Start Your Journey to Reactive Architecture

Grace Jansen and Peter Gollmar

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

Reactive Systems Explained
by Grace Jansen and Peter Gollmar

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*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Kathleen Carr
Development Editor: Sarah Grey
Production Editor: Beth Kelly
Copyeditor: Kim Wimpsett
Proofreader: Octal Publishing, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

December 2019: First Edition

Revision History for the First Edition
2019-12-04: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492077329* for release details.

978-1-492-07730-5

[LSI]

Table of Contents

Foreword

Dear reader,

In an ever-changing software landscape, user expectations grow ever more demanding. Developers need to be able to produce business value at a sustainable, rapid pace and for a user base whose size we do not necessarily know.

I started my own reactive journey back in 2009, when I asked myself whether developer productivity and software scalability were fundamentally incompatible. I realized that the most accurate system in the world would be pointless if it never provides value at the moment it is needed. That requires responsiveness. To address the challenge of responsiveness, we need to consider the evil twins that always play tricks on us: load and failure. What does the distribution of response times look like under varying load and failure conditions?

To handle failure conditions, we need replication and redundancy to ensure that functionality is still available; to cope with varying loads, we need elasticity to ensure that processing capacity can be adjusted dynamically according to demand. We need to break free of the single-machine assumption. We also need to consider infrastructural failures such as power or network failures. Different machines being spread out geographically also gives us the opportunity to serve consumers from the closest servers.

Elasticity also implies distribution. To share the increased processing burden, we need to be able to add (and remove) machines and redirect traffic dynamically. This also has a secondary benefit in as much as we can now pay for actual infrastructure usage rather than paying for maximum capacity at all times.

The key to enabling all of this is to embrace messaging at the core, to break free from the assumption of locality and a shared now. With messaging it does not matter where the recipient is or whether the recipient is available when the message is sent.

Perhaps you, the reader, will achieve the joy of discovering Reactive Systems by virtue of reading this excellent and to-the-point report. I wish you a very enjoyable time, and I suspect that there is an idea or two in it that will blow your mind.

— Viktor Klang
Deputy CTO of Lightbend,
co-creator of Reactive Streams,
co-creator of Cloudstate.io,
Tech Lead Emeritus of the Akka project

Preface

First, we'd like to acknowledge those who created a vocabulary for discussing the principles of modern, distributed application architecture and authored the Reactive Manifesto. We are grateful to this community of innovators who continue to refine these concepts and realize them in the software they create. We extend our deep gratitude to Viktor Klang for taking time to provide the foreword for this report and for his leadership in the reactive systems community. We would also like to thank our reviewers, Jason Yong, Jeremy Hughes, Neil Patterson, and Grant Steinfeld, whose comments helped shape this report and make it of the highest quality. A big thanks to Anita Chung for urging us to take on this project and providing encouragement throughout the process. We are also indebted to many colleagues who have produced some of the material on reactive systems that served to inform this report, including Jonas Bonér, Kevin Webber, Hugh McKee, Iain Lewis, and Mark Sturdevant. Finally, we would like to thank Sarah Grey and the O'Reilly team for their editorial guidance and support as we created this report.

We dedicate this book to our friends and family for supporting us through its creation and for being a constant source of inspiration and encouragement.

Grace: I would like to personally dedicate this book to my parents, Colette and Patrick Jansen, and to my brother and sister, Matthew Jansen and Eleanor Jansen, for encouraging me to pursue my passion through a career in technology and encouraging me to coauthor this report. I would also like to take this opportunity to thank my friends, partner, and colleagues for their continuous support and patience throughout the writing of this book.

Peter: I dedicate this book to my wife, Becky, whose love and support give me the courage to pursue new ventures such as this. And to my children, Bobby, Oz, and Grace, who are a constant source of inspiration.

CHAPTER 1
The Journey to Reactive Systems

Enterprises are transforming themselves into cognitive businesses: companies that learn and adapt instantly to the changing conditions around them, using real-time data and AI to bring additional value to their customers. They are realizing this business agility by building applications capable of handling massive scale, massive amounts of data, or both, and running them in a hybrid, multicloud environment.

As enterprise developers and architect leaders within these organizations, you are key influencers. Your recommendations drive the buying decisions that lead to success or failure. The good news is that today you have an unprecedented array of open source technologies on which to base these next-generation applications. That unprecedented array of choices is also the bad news. This report discusses some of the factors driving organizations to employ a reactive-systems approach to cloud native development.

Within this report we define *reactive systems* as an architectural style that enables applications composed of multiple microservices working together as a single unit to better react to their surroundings and one another, manifesting in greater elasticity when dealing with ever-changing workload demands and resiliency when components fail. We also introduce some of the key patterns found within reactive systems and distinguish between various toolkits and frameworks. Our goal is to help enterprise developers and architects make better decisions about reactive systems.

Microservices: So Many Choices

For most enterprises, an essential part of becoming a cognitive business is migrating to a hybrid multicloud infrastructure and adopting microservices application architecture. A 2018 survey of developers[1] found that more than 90% were pursuing a microservices strategy. It's not hard to see why, given that everything is trending toward greater software development velocity. Microservices are intended to be small, self-contained, single-purpose units of computation, loosely coupled to other microservices through well-defined interfaces. These characteristics enable you to develop, test, and deploy them independently—different teams, different timelines. Combining this architectural approach with DevOps methodologies such as continuous integration, continuous delivery, and continuous deployment can lead to tremendous efficiencies. But for developers, two of the chief advantages of microservices are flexibility and choice. The relative independence they offer frees you to select a mixture of programming languages or frameworks based on the services you're developing and the skills available on your team.

However, applications are composed of *systems* of microservices. How an application's microservices behave internally, as well as how they interact with one another, determines the application's ability to scale dynamically, exhibit resiliency in the presence of failures, and maintain responsiveness as workload increases. In short, these behaviors establish the fundamentals of building a *reactive system*.

When and Where Are Reactive Systems Applicable?

You could, theoretically, engineer every microservices application as a reactive system. It's equally true that this approach will present quite a learning curve for many developers today. If the application does not require it, why do it? Here's what we think: you will need to climb that hill sooner than you might think, so you might as well start the journey now. It is not a matter of "if," but "when." Let's look at a few scenarios that would drive you to a reactive systems architecture.

1 Dimensional Research. "Global Microservices Trends: A Survey of Development Professionals," April 2018

Web/Mobile Commerce

Let's say it's late November, the US Thanksgiving holiday is over, and peak shopping season has begun. Millions of people are shopping online, and every one of them expects *your* website to be responsive no matter how many others are browsing and buying at the same time. A study of the impact website performance has on buying behaviors[2] shows that sluggish sites reduced both a customer's likelihood to make a purchase as well as the amount they would be willing to pay for a product. The researchers concluded that there was a "48% reduction in revenue between the high and poor performing website." The results of this study as well as numerous others highlights just how important it is for your website to remain responsive regardless of load.

So, does the reverse apply? If you improve the responsiveness of a solidly performing web commerce site, will that improve financial results? In a word, "yes." Verizon Wireless rearchitected its monolithic commerce platform to reactive microservices and achieved a threefold reduction in page response times, dramatically improved sales conversion rates, and realized a 235% increase in revenue over a comparable prior sales period.

Data-Driven Decisions

It's been said that data is now "the world's most valuable resource." Vast amounts of data exist, and even greater amounts are generated every second of the day by nature, people, and systems. Collecting, aggregating, and making use of this data is driving new businesses and improving the way we operate. The challenge, however, is dealing with its sheer volume and velocity.

PayPal presents an interesting study of its Product Performance Tracking system. This application processes information from across PayPal's global network of platforms, giving executives insight into the performance of products, and allowing the company to adjust and deliver a better experience for its customers. The data volume and velocity statistics are stunning: 40 terabytes of data and more than 10 billion messages every day, asynchronously and at

2 SinglesStone. "Poor Website Performance Undermines Customers' Purchase Intent and Brand Impression," 2014

widely varying rates. PayPal adopted a reactive architecture approach to build a system capable of responding quickly and elastically, without the need to provision new infrastructure to handle the load.

Data-driven decisions are good, but data-driven decisions in near real time are even better. This is where reactive microservices combine with artificial intelligence and machine learning in a reactive system. Consumer credit companies are using this approach to determine credit risk, shrinking processing time to mere minutes.

CHAPTER 2

Defining Reactive Systems

Reactive systems are already in use in a wide variety of industries and use cases. But, what makes them reactive systems? How is this defined, and what makes a reactive system truly reactive? What tools or implementations are needed to achieve this complete reactivity?

The Need for a Manifesto

Prior to 2013, reactive systems were virtually unknown. However, today "reactive" has risen in popularity and is being adopted by more and more Fortune 2000 companies. This is a direct response to enterprises' need to design and build applications capable of handling massively increased scale and quantities of data. However, this widespread adoption has led to the creation of a huge variety of implementations and various versions of "reactive." So, how do we define exactly what a standard reactive system is across the industry?

In 2013, the Reactive Manifesto was created to do exactly this. This manifesto was conceived with the aim of condensing all of the knowledge we had accumulated as an industry in designing and building highly reliable and scalable applications. It then distilled this knowledge into a set of required architectural characteristics that would make any application flexible, loosely coupled, and elastic. It also carved out a defined vocabulary to enable efficient and clear communication between all participants, including developers, project leaders, architects, and CTOs.

The Reactive Manifesto lays out four key high-level characteristics of reactive systems: they are responsive, elastic, resilient, and message driven (see Figure 2-1). Although there are dependencies between them, these characteristics are not like the hierarchical tiers in a standard layered application architecture; instead, they describe design properties that should be applied across the entire technology stack.

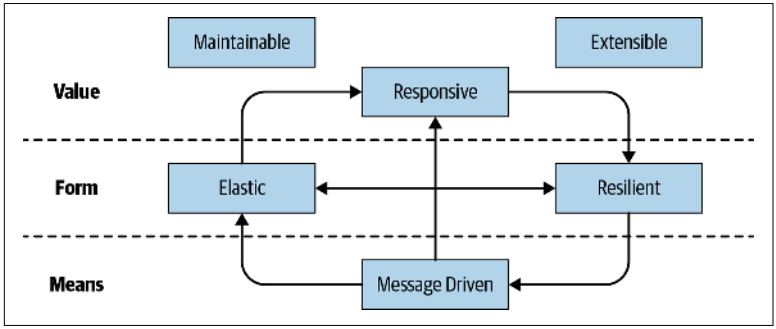

Figure 2-1. Traits of reactive systems

Reactive Systems Must Be…

To be truly reactive, a system must possess certain characteristics that allow it to be as follows:

Responsive
: This ensures that reactive systems provide rapid and consistent response times and a consistent quality of service. This simplifies error handling (allowing potential problems to be detected rapidly and dealt with effectively), builds end-user confidence, and encourages their further interaction with the system.

Resilient
: Resiliency ensures that reactive systems remain responsive in the face of failure. This is achieved through the combined use of replication, isolation, and delegation. Any failure must be contained within the originating component, isolating it from others to ensure that the failure doesn't bring down the entire system. Recovery of failing components is delegated to an external component so that the client of a component is not burdened with handling the failure. To ensure high availability, replication is used where necessary.

Elastic
: Reactive systems must remain responsive even under varying workloads and strains. Elasticity enables reactive systems to react elegantly to changes in input rate by dynamically increasing or decreasing the resources allocated. This helps relieve contention points and bottlenecks, giving the system the ability to shard or replicate components and distribute inputs among them. Elasticity makes reactive systems as resource efficient as possible.

Message driven
: An asynchronous message-passing between microservices in a reactive system establishes a boundary between components, helping to ensure loose coupling, isolation, and location transparency. Without this decoupling, it would be impossible to reach the level of compartmentalization and containment needed for isolation and resilience. Asynchronous message-passing also provides a means to delegate failures as messages. This method of explicit asynchronous message-passing allows for load management, greater elasticity, and load control by shaping and monitoring the message queues in the system and applying back pressure when necessary. This style of nonblocking communication also ensures that recipients of the messages consume resources only when active, leading to lower system overhead and more efficient use of resources. It also minimizes congestion on shared resources in the system, which is one of the biggest hurdles to scalability, low latency, and high throughput.

Events Versus Messages

In the original version of the Reactive Manifesto, "event driven" was one of the four characteristics specified instead of message driven. A *message* in this scenario is an item of data that is sent to a specific destination, whereas an *event* is defined here as a signal emitted by a component of a system upon reaching a specified state. So, in a message-driven system, addressable recipients lie dormant until they receive messages and react accordingly. But in an event-driven system, listeners are attached to the sources of events and invoked when the event is emitted. The primary difference is that an event-driven system focuses on the addressable event sources, whereas a message-driven system concentrates on addressable recipients. The

reason for this replacement in the Reactive Manifesto was that messages have a single clear destination, whereas events simply happen for others to observe them. Furthermore, messaging is preferably asynchronous, with the sending and the reception decoupled from the sender and the receiver, respectively. Event-driven messaging is still used as the communication implementation in reactive programming; we explain the difference between reactive systems and reactive programming in the next section.

Scale Matters

With the creation of the Reactive Manifesto came many implementations of reactive frameworks and toolkits, offering varying degrees of "reactive." Used in isolation, however, many of these common "reactive" frameworks do not form a fully reactive system as defined by the Reactive Manifesto.

Often when these common frameworks or tools are referred to as "reactive" or claim to implement "reactive," it's not reactive *systems* they're referring to. These applications often use the broad term of *reactive* to refer to reactive programming or reactive streams. But what do these terms really mean, and how do they differ from reactive systems?

Attempting to Program Your Way into the Reactive Trend

Reactive programming is a method for writing code based on reacting to changes.

In technical terms, this is a paradigm in which declarative code is used to construct asynchronous processing pipelines. Translated, this is essentially the same process our minds perform when we try to multitask. Rather than true parallel tasking, we actually switch tasks and split those tasks during their duration. This enables us to use our time efficiently instead of having to wait for the previous task to complete. This is exactly what reactive programming was created to do. It is an event-based model in which data is pushed to a consumer as it becomes available, turning it into an asynchronous sequence of events.

Reactive programming is a useful implementation technique for managing internal logic and dataflow transformation locally within components (intercomponents), through asynchronous and nonblocking execution. However, when there are multiple nodes, you need to start thinking hard about things like data consistency, cross-node communication, coordination, versioning, orchestration, failure management, separation of concerns and responsibilities—in short, system architecture. Reactive programming cannot address these issues or create resiliency and elasticity within a system. Instead, to maximize the value of reactive programming, we recommend it as a tool to build a reactive system.

Reactive Streaming

Reactive streaming is a specification designed to provide a standard for asynchronous stream processing with nonblocking back pressure (in other words, flow control).

The Reactive Streams specification was created to govern the exchange of stream data across an asynchronous boundary while ensuring that the receiving side is not forced to buffer arbitrary amounts of data. When handling "live" data, it is almost impossible to predict the volume of data passing through the system at any moment. This means that resource consumption has to be carefully controlled to ensure that a fast data source does not overwhelm the destination of the stream.

To tackle this, asynchrony is one of the essential components, enabling the parallel use of computing resources. It is equally important that the protocol includes a mechanism for agreeing on the velocity of the flow of data by applying back pressure. This prevents the stream destinations from being overwhelmed by ensuring a fast system cannot overload a slower counterpart. Back pressure is a term the software world borrowed from the world of fluid dynamics, where it refers to the "resistance or force opposing the desired flow of fluid through pipes." Translated to software, this term applies to the flow of data; thus the definition becomes "resistance or force opposing the desired flow of data through software."

If you embrace the Reactive Streams specification, it becomes possible to bridge systems using fully asynchronous back-pressured real-time streaming, improving the interoperability, reliability, and performance of the system as a whole. However, reactive systems do

not simply focus on reliability: they must also be elastic and responsive. Reactive streaming does not enable these characteristics; although is a useful tool, it does not, in itself, make a system reactive.

CHAPTER 3
Your Toolbox to Reactive

Now that you understand what a reactive system is and what it sets out to achieve, how do you go about implementing it? Within this chapter, we delve into tools and implementation patterns that you can use to transform your application into a reactive system.

Getting Responsive: Concurrency and Parallelism

If it's your goal to build responsive applications, the first order of business is making sure you're getting the most out of your hardware. We do this through concurrency and parallelism. Even though these concepts sound like the same thing, there's an important distinction in the context of computer systems. For the moment, let's just focus on these concepts in the context of a single CPU.

Two tasks are said to be running concurrently when they begin, run, and finish *within the same time window*. Parallelism is when two (or more) tasks are running *at the same time*. Figure 3-1 illustrates the difference between concurrency and parallelism.

Say you've been told to dig two holes and make progress on them both along the way. To accomplish this, you take a shovelful from hole A, then one from hole B, another from hole A, and so on. Every time the boss comes back to check on you, they'd see both holes getting deeper; you're working on them concurrently even though you obviously can scoop a shovelful from only one hole at a time.

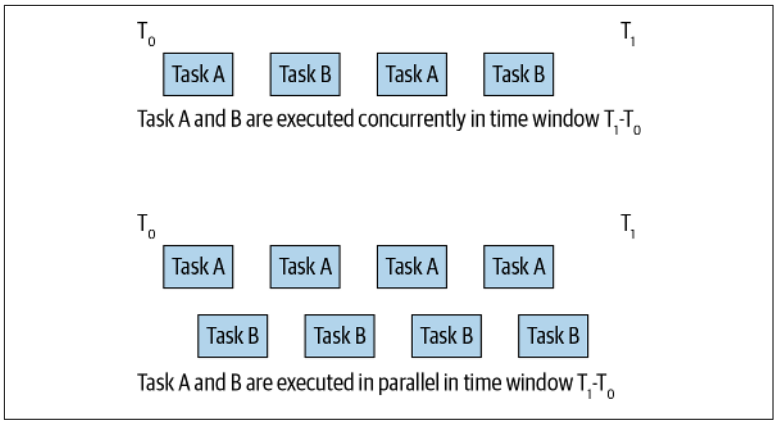

Figure 3-1. Concurrency and parallelism

Now, say there is another member of the work crew with no work to do at the moment. A smart boss would have them join in to dig with you—that is, to work in parallel—to get the job done faster.

In the remainder of this chapter, we discuss patterns for improving concurrency and parallelism in our applications, putting more "workers" on the job at once (parallelism), and keeping them all busy (concurrency) to get the job done faster and more efficiently.

Multithreading

Multithreading, an approach to programming in which two or more tasks within a process run on their own thread, has long been used as a way to achieve parallel execution within a compute platform. It is generally accepted that multithreaded programming is difficult to do well, but over the years good resources[1] have been developed and programming languages have evolved approaches to help make it a little easier. For example, Java 7 introduced the Fork/Join framework, which allows users to take large tasks, recursively break them down (or fork them) into smaller chunks, execute the subtasks in parallel, and recursively put the results together (join them) into a single result in the end. This is great for tasks that are generally recursive in nature.

1 See *Java Threads and Concurrency Utilities* by Jeff Friesen (Apress, 2015).

Even with these helpers, multithreaded programming requires great care to avoid race conditions and memory consistency problems. The combination of shared state and multithreading can produce wildly unexpected behavior in your applications unless you're careful. Also, the usual methods to protect yourself, like synchronization and locks, can be difficult to implement and can impact performance.

But there is a simpler, safer way to achieve parallelism.

The Reactor Pattern

The reactor pattern, illustrated in Figure 3-2 in its most basic form, approaches both concurrency and parallelism. In the typical implementation of the pattern, asynchronously received requests are demultiplexed (in a sense, serialized) for processing. The event loop, running on one thread, cycles through the incoming events and handles them. Callback functions are registered for requests that will result in a long-running task or blocking operation. The handle for the event gets added to a queue. The event loop iterates through the queue and will eventually observe the completion of the long-running task, trigger a callback, and return the result to the application.

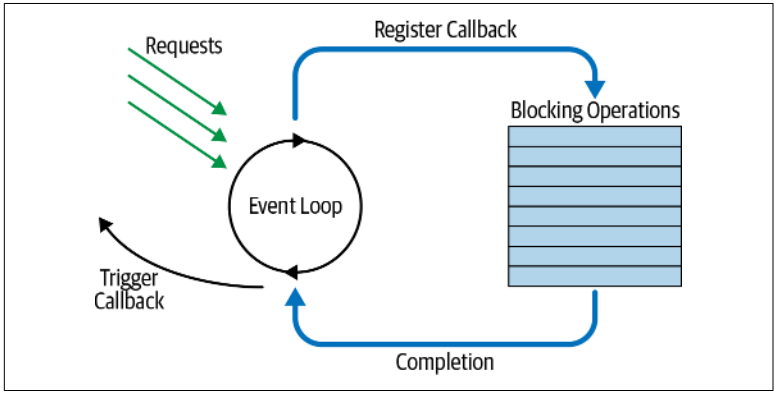

Figure 3-2. The reactor pattern

Node.js is one implementation of the reactor pattern, and its website does an excellent job of explaining how this implementation works.

You might ask how this works in a "systems" context. How do you scale a service built with the reactor pattern? Although it is possible to create multiple instances of this pattern (to scale a given service), it introduces external complexities: load balancing across instances (relatively trivial) and management of state across instances (not trivial at all).

The Multireactor Pattern

The multireactor pattern is an approach to taking fuller advantage of the available compute resources on multicore, multithreaded processors. In its basic form, instead of one event loop, you have many; the number usually depends on the number of cores on your machine. Vert.x, an open source toolkit for building reactive applications on the JVM, works in this way. For example, multiple event loops each run on their own thread, delivering events/tasks to handlers and servicing them upon completion. Code with blocking calls should be handled in the same manner as described earlier and run asynchronously on a separate thread (taken from a predefined thread pool).

Vert.x (Verticals)

Vert.x extends the multireactor pattern with a programming construct referred to as *verticals*. Although not "actors" in the strict sense (which we discuss in the next section), the Vert.x documentation describes a vertical as a "simple, scalable, actor-like deployment and concurrency model out of the box." Verticals are "chunks of code" that are deployed within a Vert.x instance. Standard verticals are assigned to an event loop when started and execute on the event loop thread. The warnings are clear: don't put blocking code here; use a worker vertical. Worker verticals are run on a different thread (never more than one thread concurrently). Verticals may be started and stopped asynchronously and communicate asynchronously over an event bus, giving you some of the basic building blocks to create a reactive system.

The Actor Model

The actor model concept was introduced by Carl Hewitt, Peter Bishop, and Richard Steiger in 1973 as an architectural foundation

for artificial intelligence. The model has been refined over the decades, and many excellent resources on the topic are now available.

In the actor model, actors are the fundamental unit of computation, and they have some important qualities that make them especially suitable to a distributed systems environment. First, actors are lightweight, loosely coupled, and maintain their own state. Second, message-passing between actors is completely asynchronous and without restriction to message ordering, making computations done throughout a system of actors inherently concurrent. Also, because interaction between actors is limited to message passing, they can be distributed across nodes (servers, virtual machines, containers). The result is a computational model that achieves concurrency, parallelism, and an inherent ability to scale horizontally.

The actor model *does* require you to think differently in your approach to programming, but there are languages and frameworks to help out.

Akka

Akka is an open source "toolkit for building highly concurrent, distributed, and resilient message-driven applications" running on a JVM. In addition to providing a hierarchical actor implementation (essential for failure detection/recovery), Akka includes libraries for actor cluster management, sharding, and persistence (invaluable for distributing applications across compute resources).

Let's go back to our discussion of concurrency, parallelism, and threads. In Akka, a message *dispatcher* is central to how threads are managed within an actor system. The dispatcher defines the executor service to be used, the size of the thread pool, and how many messages an actor may process before it relinquishes a thread. The dispatcher assigns a thread to an actor only when it has a message in its queue. The actor processes the messages and then gives back the thread. The obvious advantage is that threads are consumed only when there is actual work to be done. A less obvious advantage is that idle actors remain in memory, meaning they are immediately available for execution at all times. This presents little impact to system resources given that actors are quite small (less than one kilobyte), even if there are hundreds of thousands of them on a given node. The result is a highly efficient use of processor resources.

Summary

Each of the approaches outlined in this chapter can help you create responsive and scalable applications; it is up to you to determine which approach is best for your own use case. Multithreaded programming utilities in Java, for example, help attain a degree of concurrency and parallelism, but you must take care to avoid blocking code, race conditions, and data consistency problems. The reactor model, such as that used in Node.js, uses an event-loop approach to keep the main thread of execution busy doing productive work while shunting I/O or long-running tasks to a separate thread pool to prevent blocking. The Vert.x multireactor pattern extends this approach to take fuller advantage of available threads on modern, multicore processor-based servers. Finally, in the actor model, as implemented in Akka, for example, computations can proceed asynchronously and in parallel across the actor systems as available threads are allocated to actors with work to do.

CHAPTER 4

Putting Your Reactive Toolbox to Work

You spent most of the previous chapter focused on how the techniques and tools in our reactive toolbox help us get the most work out of your compute platforms. These help you to create an efficient, responsive service, but that's really not sufficient when you're aiming to create a reactive system. To create a fully reactive system, you need to consider the messaging within and between your services, the infrastructure it runs on, and the integration of other capabilities.

Going from Services to Systems: Being Message Driven

Microservices in a reactive system work together to achieve responsiveness, resilience, and elasticity, and this is largely achieved through being message driven. There are two levels of "message driven-ness" that we look at now: *intra-service*, or how the components of a service communicate; and *inter-service*, or messaging between services, as illustrated in Figure 4-1.

Creating multiple instances is one way of achieving scale and resilience *within* a given service. Ideally, you could scale your microservice in a way that's transparent to other parts of the system. For example, load balancing and routing between instances would be handled *internally* by the microservice, and additional instances

would be spun up or down as needed in response to load and other external conditions. None of this is trivial, but it's made possible by messaging between the elements that make up the microservice collective.

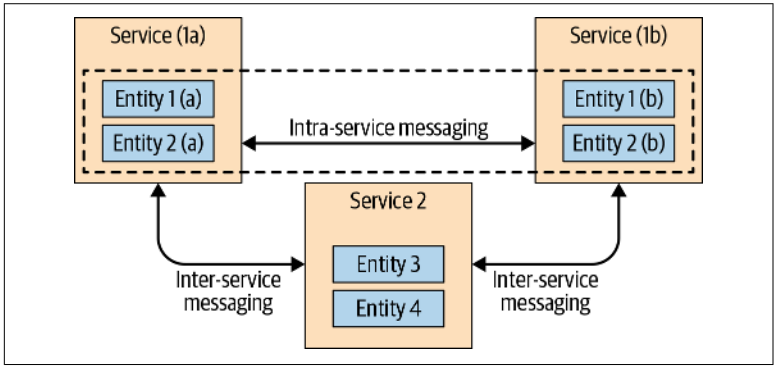

Figure 4-1. Messaging within and between services

There are open source libraries that can help you. One example is Akka, which provides libraries that allow you to manage the components of your microservice, actor systems in this case, as a "system of systems." Another open source library is Vert.x, which uses an event bus between components of your microservices (verticals) and supports both point-to-point and publish/subscribe (pub/sub) messaging.

By default, both Akka and Vert.x support "best effort" or "at most once" delivery of messages. This means acknowledging that "failures happen," so your application should be able to handle the potential for lost messages between components of your service.

Messaging *between* services is a different matter, and durable messaging is a prerequisite for a microservice-based application, reactive or otherwise. This is where a pub/sub system is really useful. In a pub/sub integration pattern, a microservice producing an event (data) publishes it to an event bus, and services subscribed to that event bus take note and consume the event or data. Communication between microservices is completely asynchronous and location independent. It is important that the event bus is "durable" so that events persist long enough for subscribers to pick them up; in case of a failure, the message stream can be reconstructed with the events

intact. Apache Kafka is one of the better-known platforms for implementing durable pub/sub messaging.

Reactive streaming takes inter-service messaging to the next level. As explained in Chapter 2, reactive streams implementations give you a graceful way to handle unbounded streams of data across asynchronous boundaries with back pressure. This makes it perfect to integrate your reactive microservices with external systems and to implement messaging between services within your reactive system. There are several frameworks and libraries from which to choose that implement the Reactive Streams specification, including Akka Streams and Vert.x Reactive Streams.

What type of infrastructure will allow us to reap the full benefits of this message-driven approach to application architecture?

Distributed Infrastructure

As the pure quantity of data and scale of connected devices, sessions, and transactions continue to rise at an exponential rate, businesses have turned to distributed, cloud-based infrastructures to build applications that can process these huge volumes of data and support vast numbers of online users concurrently. Cloud native systems have become the bread and butter of many applications. However, getting the most out of this compute platform entails writing highly concurrent, distributed software. That isn't easy! It requires properly handling threads, implementing synchronization, preventing race conditions, dealing with persistence and state, scaling the application, and responding to failures. Fortunately, this is exactly what reactive systems were designed for. Reactive systems embrace distributed infrastructure, creating a consistent and responsive experience that works as expected even in the face of failure and unpredictable loads.

Orchestrated Cloud Infrastructure

How can you go about deploying and managing your reactive system in a cloud environment? The answer is containers and Kubernetes. Containerization—packaging up your microservices and all of their dependencies into lightweight packages that can run anywhere—is a basic tenet of building cloud native software, and

Kubernetes has become the de facto open source standard for managing containerized applications in production.

Kubernetes is fundamentally a cluster orchestration system that brings "reactive systems" characteristics to container management. We touch on just a few basic Kubernetes concepts here, but there are plenty of good resources available.

Here are the basics in a nutshell:

Clusters and nodes
: A Kubernetes cluster consists of a set of nodes (VMs or physical machines) on which Kubernetes services are running.

Pods
: A pod is the fundamental unit of deployment in a Kubernetes cluster. It essentially wraps one or more app containers along with the necessary network and storage resources.

Controllers
: Lastly, controllers are Kubernetes services that watch over the cluster and take corrective action when the current state of the cluster strays from the desired state.

The cluster management capacity of Kubernetes gives you both resilience and elasticity at the infrastructure layer. Elasticity is achieved with Kubernetes autoscaling, which can dynamically adjust the number of nodes in your cluster as well as scale the number of pods running in your cluster based on workload. Automated pod recovery features enhance resilience by restarting failed pods or recreating pods that have been deleted. An application created with reactive architectural principles will expand, contract, and redistribute itself with changes in the underlying infrastructure.

Running your reactive microservices application on a Kubernetes orchestrated infrastructure can provide multiple levels of resilience and elasticity.

A blog series on IBM Developer illustrates the concepts described in this section. The series includes code for a simple Akka application deployed to Kubernetes as well as a simple tool to visualize the interaction of the Kubernetes pods, Akka clusters, and Akka actors.

Reactive Meets Machine Learning

Now that we've covered the application architecture and infrastructure approach for creating a responsive, elastic, and resilient application, the next step in your journey to become a cognitive business is to instrument your application to take advantage of data-driven insights in real time. This means combining vast amounts of data with machine learning to make your reactive system "smarter" in ways that create value for your customers.

First, we need to (briefly) introduce the concepts of machine learning (ML) and ML models, given that these are the means to infusing your applications with intelligence. ML is a subset of artificial intelligence that involves computers "learning" from exposure to data, being trained to find patterns in the data. The result is a computational model that can more or less correctly respond when new data is provided. Models can be developed for identification (such as image or voice recognition) or predictions (such as the weather).

Figure 4-2 illustrates the basic elements and flow of an ML-enabled reactive system. It's generalized to show devices as well as users as sources of data on which the models are based.

Let's say you are shopping at a brick-and-mortar store for new running shoes and find a pair you like, but you don't love the price. You pull out your mobile phone and open an app for an online retailer to see if it has the shoes at a better price. (Admit it. You've done this.) The online retailer will have modeled your buying behaviors and that of millions of others, so with the right supplemental data (for example, your location near a running shoe store), it can present you with an offer for a pair of great running shoes at an attractive price even before you begin your search in its app. You're delighted.

It's worth noting that ML-enriched applications can be created with virtually any application technology. However, beginning from within a reactive systems context gives you the ability to dynamically manage the entire ML model—serving life cycle: streaming data from a wide array of sources, retraining models, and automatically redeploying them in real time.

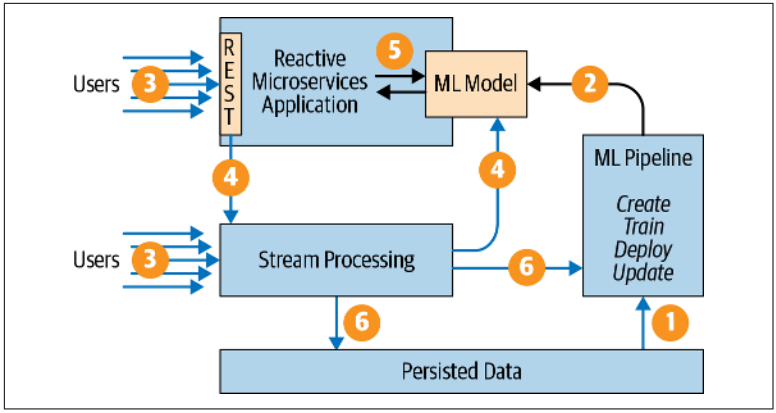

Figure 4-2. The ML-infused application flow begins in the lower right with the model development. The model is created and trained (1) using historical data, then deployed (2), and then made available to an application. Data arrives into the system from users (3a) and other sources (3b) and becomes the source data used by the model (4) to produce a prediction, or recommendation (5). This additional data (6) can be used to retrain the machine learning model to improve its accuracy.

Resources for Further Learning

We've only just scratched the surface on this topic, so, here we include a couple of good references by experts in this area: *Fast Data Architectures for Streaming Applications* by Dean Wampler (O'Reilly) and *Serving Machine Learning Models* by Boris Lublinsky (O'Reilly). You can also find some additional resources on IBM Developer to help you get to coding right away. First is an example of streaming online retail store data into an analytics pipeline. Next is a code pattern to help you use machine learning to develop a product recommender. Taken together, these form two of the important building blocks for our cognitive application example.

Conclusions

Enterprises have been striving to create ultra-fast, ultra-responsive applications since the dawn of the internet. But, with the vast number of connected devices, huge quantities of data, and ever-growing number of consumers of our applications, traditional methods of trying to achieve this just don't cut it. Reactive systems enable you to achieve this responsiveness through elasticity and resiliency in an autonomous, cost-effective manner—no longer are specialist teams needed to redesign and redeploy applications when they need to scale due to load changes; no longer do applications go down every time a new feature is introduced. Throughout this report we've listed the many benefits that reactive systems give your enterprise applications—in summary, the ability to design and build truly responsive, cognitive applications that manage themselves.

When Is Reactive Systems the Right Choice?

The right time to consider using reactive application architecture to transform your enterprise applications into reactive systems is when you begin caring about any of the cornerstones of the Reactive Manifesto—resiliency, elasticity, or responsiveness. If your application is dealing with vast volumes of data and you want your system to remain responsive and provide the same quality of service to every user regardless of changes in load, reactive architecture is definitely worth considering. Our old architecture patterns just weren't designed to cope with the changing world of data we live in and the huge fluctuations in load on our systems.

Reactive isn't for everyone, though, just as microservices are not the answer to modernizing every application. Before jumping in, consider whether your enterprise cares about maintaining responsiveness by having a more resilient and elastic system. If you do, maybe it's time you had a go at creating your own reactive system.

How to Get Started

You've seen that there are some important decisions to make before diving into the design of your own reactive system. The first step toward redesigning a traditional application into a reactive system is

deciding how best to split up your application. Event storming[1] combined with domain-driven design are great techniques for breaking down an application into its distinct business domains.

The next important decision is which programming framework to use. There are many different ways to implement a reactive system. We hope this overview has given you enough information and resources to decide which implementation is best for you. Each programming framework achieves concurrency, parallelism, resiliency, and messaging in a variety of ways and interacts with the underlying infrastructure differently. So, it's important to fully understand the capabilities of the implementation you choose.

After you break down your application into its respective microservices and select your programming framework, it's time to decide which reactive patterns are best for your system. There's a whole laundry list of patterns that you can combine to build a reactive system. For example, Event Sourcing and Command Query Responsibility Separation (CQRS) are often used together to optimize performance and scalability.

Additional Resources

IBM's Reactive in Practice tutorial blog series goes into many of these reactive patterns in depth, along with an example of migrating a traditional application to a reactive system. It is a great place to understand more about these patterns and help choose which you may need. It can be used as a template to lead you through your own transformation: simply Git clone the source code, rewrite it to your favorite implementation, and edit it for your own application. Finally, no list of references on reactive systems would be complete without mentioning *Reactive Microsystems* by Jonas Bonér.

We hope this book has given you a solid understanding of what reactive systems are and why they may just be the next step forward in the evolution of your own enterprise systems. We wish you the

1 Event storming brings together the IT, business, and service delivery teams in a collaborative workshop to model business processes. For an example, see *https://oreil.ly/mr83f*.

best as you improve the responsiveness, resiliency, and elasticity of your applications, providing better customer interactions.

About the Authors

Grace Jansen is a developer advocate at IBM, having switched from biology to software engineering since leaving university. She specializes in advocating reactive application architecture and reactive systems and is a seasoned speaker and regular presenter at international conferences. Her talks have gained popularity through their use of biological analogies to link modern enterprise application architecture to the natural world.

Peter Gollmar is an offering manager at IBM responsible for introducing new products and developer offerings with strategic partners. He works with a talented team of developer advocates to engage the community on reactive systems technologies. In previous roles with IBM, Peter led business transformation initiatives using IT to create new ways to deliver solutions to IBM's clients.