O'REILLY®

Why Reactive?

Foundational Principles for Enterprise Adoption

Konrad Malawski

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

Why Reactive?

Foundational Principles for Enterprise Adoption

Konrad Malawski

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

Why Reactive?
by Konrad Malawski

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

Editor: Brian Foster
Production Editor: Colleen Cole
Copyeditor: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

October 2016: First Edition

Revision History for the First Edition
2016-10-10: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491961575* for release details.

978-1-491-96157-5

[LSI]

Table of Contents

1. Introduction. 1
Why Build Reactive Systems? 2
And Why Now? 4

2. Reactive on the Application Level. 7
In Search of the Optimal Utilization Level 11
Using Back-Pressure to Maintain Optimal Utilization Levels 12
Streaming APIs and the Rise of Bounded-Memory Stream Processing 14
Reactive Is an Architectural and Design Principle, Not a Single Library 16

3. Reactive on the System Level. 17
There's More to Life Than Request-Response-JSON-over-HTTP 19
Surviving the Load…and Shaving the Bill! 24
Without Resilience, Nothing Else Matters 25

4. Building Blocks of Reactive Systems. 27
Introducing Reactive in Real-World Systems 28
Reactive, an Architectural Style for Present and Future 30

CHAPTER 1
Introduction

> *It's increasingly obvious that the old, linear, three-tier architecture model is obsolete.*[1]
>
> —A Gartner Summit track description

While the term *reactive* has been around for a long time, only recently has it been recognized by the industry as the de facto way forward in system design and hit mainstream adoption. In 2014 Gartner wrote that the three-tier architecture that used to be so popular was beginning to show its age. The goal of this report is to take a step back from the hype and analyze what reactive really is, when to adopt it, and how to go about doing so. The report aims to stay mostly technology agnostic, focusing on the underlying principles of reactive application and system design. Obviously, certain modern technologies, such as the Lightbend or Netflix stacks, are far better suited for development of Reactive Systems than others. However, instead of giving blank recommendations, this report will arm you with the necessary background and understanding so you can make the right decisions on your own.

This report is aimed at CTOs, architects, and team leaders or managers with technical backgrounds who are looking to see what reactive is all about. Some of the chapters will be a deep dive into the technical aspects. In Chapter 2, which covers reactive on the appli-

1 Gartner Summits, *Gartner Application Architecture, Development & Integration Summit 2014* (Sydney, 2014), *http://www.gartner.com/imagesrv/summits/docs/apac/application-development/AADI-APAC-2014-Brochure.pdf*.

cation level, we will need to understand the technical differences around this programming paradigm and its impact on resource utilization. The following chapter, about reactive on the system level, takes a step back a bit and looks at the architectural as well as organizational impact of distributed reactive applications. Finally, we wrap up the report with some closing thoughts and suggest a few building blocks, and how to spot really good fits for reactive architecture among all the marketing hype around the subject.

So, what does reactive really mean? Its core meaning has been somewhat formalized with the creation of the Reactive Manifesto[2] in 2013, when Jonas Bonér[3] collected some of the brightest minds in the distributed and high-performance computing industry—namely, in alphabetical order, Dave Farley, Roland Kuhn, and Martin Thompson—to collaborate and solidify what the core principles were for building reactive applications and systems. The goal was to clarify some of the confusion that around reactive, as well as to build a strong basis for what would become a viable development style. While we won't be diving very deep into the manifesto itself in this report, we strongly recommend giving it a read. Much of the vocabulary that is used in systems design nowadays (such as the difference between errors and failures) has been well defined in it.

Much like the Reactive Manifesto set out to clarify some of the confusion around terminology, our aim in this report is to solidify a common understanding of what it means to be reactive.

Why Build Reactive Systems?

> *It's no use going back to yesterday,*
> *because I was a different person then.*
>
> —Lewis Carroll

Before we plunge into the technical aspects of Reactive Systems and architecture, we should ask ourselves, "Why build Reactive Systems?"

2 Jonas Bonér et al., "The Reactive Manifesto," September 16, 2014, *http://www.reactivemanifesto.org*.

3 Jonas Bonér, Founder and CTO of Lightbend (previously known as Typesafe) in 2011, and Scalable Solutions in 2009, *http://jonasboner.com*.

Why would we be interested in changing the ways we've been building our applications for years? Or even better, we can start the debate by asking, "What benefit are we trying to provide *to the users* of our software?" Out of many possible answers, here are some that would typically lead someone to start looking into Reactive Systems design. Let's say that our system should:

- Be responsive to interactions with its users
- Handle failure and remain available during outages
- Strive under varying load conditions
- Be able to send, receive, and route messages in varying network conditions

These answers actually convey the core reactive traits as defined in the manifesto. Responsiveness is achieved by controlling our applications' hardware utilization, for which many reactive techniques are excellent tools. We look at a few in Chapter 2, when we start looking at reactive on the application level. Meanwhile, a good way to make a system easy to scale is to decouple parts of it, such that they can be scaled independently. If we combine these methods with avoiding synchronous communication between systems, we now also make the system more resilient. By using asynchronous communication when possible, we can avoid binding our lifecycle strictly to the request's target host lifecycle. For example, if the lifecycle is running slowly, we should not be affected by it. We'll examine this issue, along with others, in Chapter 3, when we zoom out and focus on reactive on the system level, comparing synchronous request-response communication patterns with asynchronous message passing.

Finally, in Chapter 4 we list the various tools in our toolbox and talk about how and when to use each of them. We also discuss how to introduce reactive in existing code bases as we acknowledge that the real world is full of existing, and valuable, systems that we want to integrate with.

And Why Now?

> *The Internet of Things (IoT) is expected to surpass mobile phones as the largest category of connected devices in 2018.*
>
> —Ericsson Mobility Report

Another interesting aspect of the "why" question is unveiled when we take it a bit further and ask, "Why now?"

As you'll soon see, many of the ideas behind reactive are not that new; plenty of them were described and implemented years ago. For example, Erlang's actor-based programming model has been around since the early 1980s, and has more recently been brought to the JVM with Akka. So the question is: why are the ideas that have been around so long now taking off in mainstream enterprise software development?

We're at an interesting point, where scalability and distributed systems have become the everyday bread and butter in many applications which previously could have survived on a single box or without too much scaling out or hardware utilization. A number of movements have contributed to the current rise of reactive programming, most notably:

IoT and mobile
: The mobile sector has seen a 60% traffic growth between Q1 2015 and Q1 2016; and according to the Ericsson Mobility Report,[4] that growth is showing no signs of slowing down any time soon. These sectors also by definition mean that the server side has to handle millions of connected devices concurrently, a task best handled by asynchronous processing, due to its lightweight ability to represent resources such as "the device," or whatever it might be.

Cloud and containerization
: While we've had cloud-based infrastructure for a number of years now, the rise of lightweight virtualization and containers, together with container-focused schedulers and PaaS solutions,

4 "Ericsson Mobility Report," Ericsson, (June 2016), *https://www.ericsson.com/res/docs/2016/ericsson-mobility-report-2016.pdf*.

has given us the freedom and speed to deploy much faster and with a finer-grained scope.

In looking at these two movements, it's clear that we're at a point in time that both the need for concurrent and distributed applications is growing stronger. At the same time, the tooling needed to do so at scale and without much hassle is finally catching up. We're not in the same spot as we were a few years ago, when deploying distributed applications, while possible, required a dedicated team managing the deployment and infrastructure automation solutions.

It is also important to realize that many of the solutions that we're revisiting, under the umbrella movement called reactive, have been around since the 1970s. Why reactive is hitting the mainstream now and not then, even though the concepts were known, is related to a number of things. Firstly, the need for better resource utilization and scalability has grown strong enough that the majority of projects seek solutions. Tooling is also available for many of these solutions, both with cluster schedulers, message-based concurrency, and distribution toolkits such as Akka. The other interesting aspect is that with initiatives like Reactive Streams,[5] there is less risk of getting locked into a certain implementation, as all implementations aim to provide nice interoperability. We'll discuss the Reactive Streams standard a bit more in depth in the next chapter.

In other words, the continuous move toward more automation in deployment and infrastructure has led us to a position where having applications distributed across many specialized services spread out onto different nodes has become frictionless enough that adopting these tools is no longer an impediment for smaller teams. This trend seems to converge with the recent rise of the *serverless*, or ops-less, movement. This movement is the next logical step from each and every team automating their cloud by themselves. And here it is important to realize that reactive traits not only set you up for success right now, but also play very well with where the industry is headed, toward location-transparent,[6] ops-less distributed services.

5 Reactive Streams, a standard initiated by Lightbend and coauthored with developers from Netflix, Pivotal, RedHat, and others.

6 Location-transparency is the ability to communicate with a resource regardless of where it is located, be it local, remote, or networked. The term is used in networks as well as Reactive Systems.

CHAPTER 2
Reactive on the Application Level

> *The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer's bottleneck does.*[1]
>
> —John Backus

As the first step toward building Reactive Systems, let's look at how to apply these principles within a single application. Many of the principles already apply on the local (application) level of a system, and composing a system from reactive building blocks from the bottom up will make it simple to then expand the same ideas into a full-blown distributed system.

First we'll need to correct a common misunderstanding that arose when two distinct communities used the word "reactive," before they recently started to agree about its usage. On one hand, the industry, and especially the ops world, has for a long time been referring to systems which can heal in the face of failure or scale out in the face or increased/decreased traffic as "Reactive Systems." This is also the core concept of the Reactive Manifesto. On the other hand, in the academic world, the word "reactive" has been in use since the term "functional reactive programming" (FRP), or more

1 John Backus, "Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs," *Communications of the ACM* 21, no. 8 (Aug. 1978), doi:10.1145/359576.359579.

specifically "functional reactive activation,"[2] was created. The term was introduced in 1997 in Haskell and later Elm, .NET (where the term "Reactive Extensions" became known), and other languages. That technique indeed is very useful for Reactive Systems; however, it is nowadays also being misinterpreted even by the FRP frameworks themselves.

One of the key elements to reactive programming is being able to execute tasks asynchronously. With the recent rise in popularity of FRP-based libraries, many people come to reactive having only known FRP before, and assume that that's everything Reactive Systems have to offer. I'd argue that while event and stream processing is a large piece of it, it certainly is neither a requirement nor the entirety of reactive. For example, there are various other programming models such as the actor model (known from Akka or Erlang) that are very well suited toward reactive applications and programming.

A common theme in reactive libraries and implementations is that they often resort to using some kind of event loop, or shared dispatcher infrastructure based on a thread pool. Thanks to sharing the expensive resources (i.e., threads) among cheaper constructs, be it simple tasks, actors, or a sequence of callbacks to be invoked on the shared dispatcher, these techniques enable us to scale a single application across multiple cores. This multiplexing techniques allow such libraries to handle millions of entities on a single box. Thanks to this, we suddenly can afford to have one actor per user in our system, which makes the modelling of the domain using actors also more natural. With applications using plain threads directly, we would not be able to get such a clean separation, simply because it would become too heavyweight very fast. Also, operating on threads directly is not a simple matter, and quickly most of your program is dominated by code trying to synchronize data across the different threads—instead of focusing on getting actual business logic done.

The drawback, and what may become the new "*who broke the build?!*" of our days is encapsulated in the phrase "*who blocked the event-loop?!*" By blocking, we mean operations that take a long (pos-

2 Conal Elliott and Paul Hudak, "Functional Reactive Animation," *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming - ICFP '97*, 1997, doi:10.1145/258948.258973.

sibly unbounded) time to complete. Typical examples of problematic blocking include file I/O or database access using blocking drivers (which most current database drivers are). To illustrate the problem of blocking let's have a look at the diagram on Figure 2-1. Imagine you have two actual single-core processors (for the sake of simplicity, let's assume we're not using hyper-threading or other techniques similar to it), and we have three queues of work we want to process. All the queues are more or less equally imporant, so we want to process them as fair (and fast) as possible. The fairness requirement is one that we often don't think about when programming using blocking techniques. However, once you go asynchronous, it starts to matter more and more. To clarify, *fairness* in such a system is the property that the service time of any of the queues is roughly equal—there is no "faster" queue. The colors on each timeline on Figure 2-1 highlight which processor is handling that process at any given moment. According to our assumptions, we can only handle two processes in parallel.

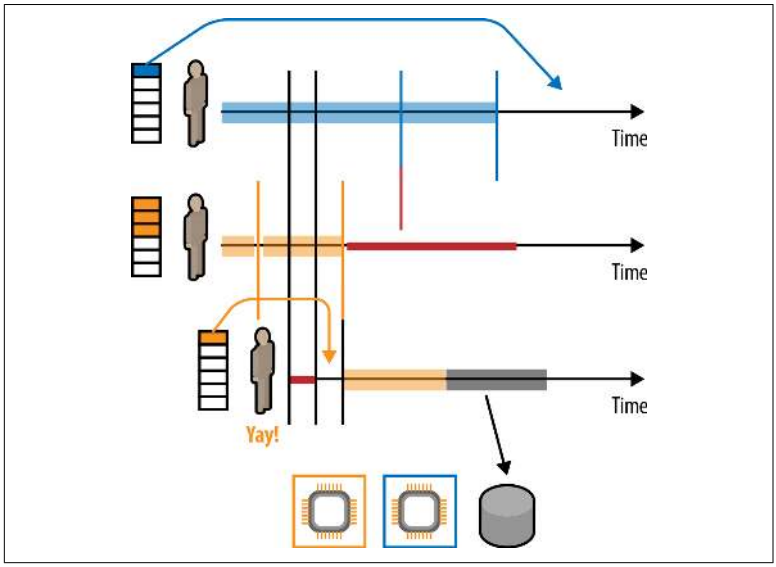

Figure 2-1. Blocking operations, shown here in Gray, waste resources often impacting overall system fairness and perceived response time for certain (unlucky) users

The gray area signifies that the actor below has issued some blocking operation, such as attempting to write data into a file or to the network using blocking APIs. You'll notice that the third actor now is

not really doing anything with the CPU resource; it is being wasted waiting on the return of the blocking call. In Reactive Systems, we'd give the thread back to the pool when performing such operations, so that the middle actor can start processing messages. Notice that with the blocking operation, we're causing starvation on the middle queue, and we sacrifice both fairness of the overall system along with response latency of requests handled by the middle actor.

Some people misinterpret the observation and diagram as "Blocking is pure evil, and everything is doomed!" Sometimes opponents of reactive technology use this phrase to spread fear, uncertainty, and doubt (aka FUD, an aggressive marketing methodology) against more modern reactive tech stacks. What the message *actually* is (and always was) is that *blocking needs careful management!*

The solution many reactive toolkits (including Netty, Akka, Play, and RxJava) use to handle blocking operations is to isolate the blocking behavior onto a different thread pool that is dedicated for such blocking operations. We refer to this technique as *sandboxing* or *bulkheading*. In Figure 2-2, we see an updated diagram, the processors now represent actual cores, and we admit that we've been talking about thread pools from the beginning. We have two thread pools, the default one in yellow, and the newly created one in gray, which is for the blocking operations. Whenever we're about to issue a blocking call, we put it on that pool instead. The rest of the application can continue crunching messages on the default pool while the third process is awaiting a response from the blocking operation. The obvious benefit is that the blocking operation does not stall the main event loop or dispatcher.

However, there are more and perhaps less obvious benefits to this segregation. One of them might be hard to appreciate until one has worked more with asynchronous applications, but it turns out to be very useful in practice. Since we have now segregated different types of operations on different pools, if we notice a pool is becoming overloaded we can get an immediate hunch where the bottleneck in our application just appeared. It also allows us to set strict upper limits onto the pools, such that we never execute more than the allowed number of heavy operations. For example, if we configure a dispatcher for all the CPU intensive tasks, it would not make sense to launch 20 of those tasks concurrently, if we only have four cores.

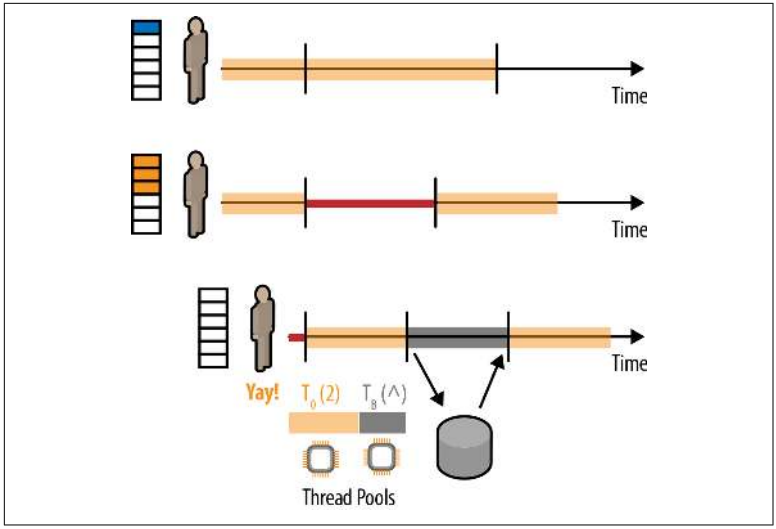

Figure 2-2. Blocking operations are scheduled on a dedicated dispatcher (gray). So that the normal reactive operations can continue unhindered on the default dispatcher (yellow)

In Search of the Optimal Utilization Level

In the previous section, we learned that using asynchronous APIs and programming techniques helps to increase utilization of your hardware. This sounds good, and indeed we do want to use the hardware that we're paying for to its fullest. However, the other side of the coin is that pushing utilization beyond a certain point will yield diminishing (or even negative if pushed further) returns. This observation has been formalized by Neil J. Gunther in 1993 and is called the Universal Scalability Law (USL).[3]

The relation between the USL, Amdahl's law, and queueing theory is material worth an entire paper by itself, so I'll only give some brief intuitions in this report. If after reading this section you feel intrigued and would like to learn more, please check out the white paper "Practical Scalability Analysis with the Universal Scalability Law" by Baron Schwartz (O'Reilly).

3 Neil J. Gunther, "A Simple Capacity Model of Massively Parallel Transaction Systems," proceedings of CMG National Conference (1993), *http://www.perfdynamics.com/Papers/njgCMG93.pdf*.

The USL can be seen as a more practical model than the more widely known Amdahl's law, first defined by Gene Amdahl in 1967, which only talks about the theoretical speedup of an algorithm depending on how much of it can be executed in parallel. USL on the other hand takes the analysis a step further by introducing the cost of communication, the cost of keeping data in sync—coherency—as variable in the quotation, and suggests that pushing a system beyond its utilization sweet spot will not only not yield any more speedup, but will actually have a *negative* impact on the system's overall throughput, since all kind of coordination is happening in the background. This coordination might be on the hardware level (e.g., memory bandwidth saturation, which clearly does not scale with the number of processors) or network level (e.g., bandwidth saturation or incast and retransmission problems).

One should note that we can compete for various resources and that the over-utilization problem applies not only to CPU, but—in a similar vein—to network resources. For example, with some of the high-throughput messaging libraries, it is possible to max out the 1 Gbps networks which are the most commonly found in various cloud provider setups (unless specific network/node configurations are available and provisioned, such as 10 Gbps network interfaces available for specific high-end instances on Amazon EC2). So while the USL applies both to local and distributed settings, for now let's focus on the application-level implications of it.

Using Back-Pressure to Maintain Optimal Utilization Levels

When using synchronous APIs, the system is "automatically" back-pressured by the blocking operations. Since we won't do anything else until the blocking operation has completed, we're wasting a lot of resources by waiting. But with asynchronous APIs, we're able to max out on performing our logic more intensely, although we run the risk of overwhelming some other (slower) downstream system or other part of the application. This is where *back-pressure* (or *flow-control*) mechanisms come into play.

Similar to the Reactive Manifesto, the Reactive Streams initiative emerged from a collaboration between industry-leading companies building concurrent and distributed applications that wanted to standardize an interop protocol around bounded-memory stream

processing. This initial collaboration included Lightbend, Netflix, and Pivotal, but eventually grew to encompass developers from RedHat and Oracle.[4] The specification is aimed to be a low-level interop protocol between various streaming libraries, and it requires and enables applying back-pressure transparently to users of these streaming libraries. As the result of over a year of iterating on the specification, its TCK, and semantic details of Reactive Streams, they have been incorporated in the OpenJDK, as part of the JEP-266 "More Concurrency Updates" proposal.[5] With these interfaces and a few helper methods that have become part of the Java ecosystem directly inside the JDK, it is safe to bet on libraries that implement the Reactive Streams interfaces to be able to move on to the ones included in the JDK, and be compatible even in the future—with the release of JDK9.

It is important to keep in mind that back-pressure, Reactive Streams, or any other part of the puzzle is not quite enough to make a system resilient, scalable, and responsive. It is the combination of the techniques described here which yields a fully reactive system. With the use of asynchronous and back-pressured APIs, we're able to push our systems *to* their limits, but *not beyond* them. Answering the question of how much utilization is in fact optimal is tricky, as it's always a balance between being able to cope with a sudden spike in traffic, and wasting resources. It also is very dependent on the task that the system is performing. A simple rule of thumb to get started with (and from there on, optimize according to your requirements) is to keep system utilization below 80%. An interesting discussion about battles fought for the sake of optimizing utilization, among other things, can be read in the excellent Google Maglev paper.[6]

One might ask if this "limiting ourselves" could lower overall performance compared to the synchronous versions. It is a valid question to ask, and often a synchronous implementation will beat an asyn-

4 Viktor Klang, "Reactive Streams 1.0.0 Interview," Medium (June 01, 2015), *https://medium.com/@viktorklang/reactive-streams-1-0-0-interview-faaca2c00bec#.ckcwc9o10*.

5 Doug Lea, "JEP 266: More Concurrency Updates," OpenJDK (September 1, 2016), *http://openjdk.java.net/jeps/266*.

6 Eisenbud et al., "Maglev: A Fast and Reliable Software Network Load Balancer," *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, USENIX Association, Santa Clara, CA (2016), pp. 523-535, *http://research.google.com/pubs/pub44824.html*.

chronous one in a single-threaded, raw-throughput benchmark, for example. However, real-world workloads do not look like that. In an interesting performance analysis of RxNetty compared to Tomcat at Netflix, Brendan Gregg and Ben Christensen found that even with the asynchronous overhead and flow control, the asynchronous server implementation did yield much better response latency under high load than the synchronous (and highly tuned) Tomcat server.[7]

Streaming APIs and the Rise of Bounded-Memory Stream Processing

> *Ever-newer waters flow on those who step into the same rivers.*
>
> —Heraclitus

Streaming, much like reactive, is in a phase where the community is trying to actually define what it means when one uses the word "stream." Sadly, people are perhaps more confused about it than they are with "reactive." The reason the streaming landscape has become so confusing is that multiple libraries that address very different needs have come to use the word. For example, Spark Streaming and Flink address the large-scale data-transformation side but are not well suited for small- to medium-sized jobs, nor for embedding as source of data for an HTTP response APIs that respond by providing an infinite stream of data, like the well-known Twitter Streaming APIs.

In this chapter, we'll focus on what streaming actually means, why it matters, and what's to come. There are two sides of the same coin here: consuming and producing streaming APIs. There's a reason we discuss this topic in the chapter about reactive on the application level, and not system level, even though APIs serve as the integration layer between various systems. It has to do with the interesting capabilities that streaming APIs and bounded-in-memory processing give us. Most notably, using and/or building streaming libraries and APIs allows us to never load more of the data into memory than we actually need, which in turn allows us to build bounded-memory pipelines. This is a very interesting and useful property for capacity

7 Gregg, Kant, and Christensen, "rxNetty vs Tomcat notes," Netflix-Skunkworks/WSPerfLab on GitHub (Apr 10, 2015), *https://github.com/brendangregg/WSPerfLab/commit/master*.

planning, as now we have a guarantee of how much memory a given connection or stream will take, and can include these numbers in our capacity planning calculations.

Let's discuss this feature as it relates to the Twitter Firehose API, an API an application can subscribe to in order to collect and analyze *all* incoming tweets in the Twittersphere. Obviously, consuming such a high-traffic stream takes significant machine power on the receiving end as well. And this is where it gets interesting, what if the downstream (the customer, accessing the firehose API) is not able to consume it at the rate at which it is being emitted?

Let's look at this design challenge from the server's perspective. We have a live stream of data coming in from our backends, and we need to push those to downstream clients of the service. Some of them may be slow or may have trouble on their end which causes them to not consume the stream at all. What should we do? The usual answer is to buffer until the client comes back. That answer is both correct and scary at the same time. Of course, we want to buffer a little bit, to allow the client to recover from the slowness on their end and continue consuming the stream. However, we can't be expected to buffer these tweets indefinitely; that would take unbounded amounts of memory (which would be lovely to have, but we're not there yet). The solution is simple: we use *bounded buffers*. For example, if we see clients not being able to cope with the rate at which we'd like to emit events, we issue warnings to the clients (which in fact is an option available in all of Twitter's streaming APIs) that "we're queueing up messages for delivery to you. Your queue is now over 60% full." This is a very nice strategy because we can estimate based on our bounded-size queues, how many slow clients we're able to service, and balance our service levels and node utilization.

Monitoring the queue and buffer sized of clients of our APIs is a very interesting metric, and could even trigger some interaction with the customer. For example, we could offer them additional processing power, or suggest that they optimize their client in some way. Of course, once the buffer is full, we need to do something to save our application from any trouble. In the Twitter example, the answer is simple: we disconnect the client (the warning message always includes details about this). But this is not the only option. One could also drop the oldest or newest elements in the stream, especially if the oldest ones have become outdated. Needless to say,

such decisions should be business driven, although the key enabler of making them *trivial* are streaming-first libraries such as Akka Streams/HTTP, where it is as simple as picking an overflow strategy, which Akka documents in its quick start guide.

Reactive Is an Architectural and Design Principle, Not a Single Library

> *The whole is other than the sum of the parts.*
>
> —Kurt Koffka

You may have heard this quote phrased as the whole being "greater" than the sum of its parts. But it turns out that the actual quote was slightly different, as Koffka meant meant the whole being, in fact, something *different* than the sum of parts, not necessarily greater or larger (e.g., like the "invisible" triangle in Figure 2-3). The quote fits the situation we find ourselves in with the term "reactive." You can easily find numerous libraries with the "reactive" prefix added to them, as well as some trying to change the meaning of reactive to be the only task that library does (e.g., stream processing). It is important to realize these are excellent building blocks, but they rarely address the entire problem (most likely omitting to handle resilience or elasticity). In the next chapter, dedicated to Reactive Systems, we'll learn more about what else there is to reactive other than a nice programming model *inside* the applications we're building. We'll see how applications communicate and how we can scale them out and back down on demand.

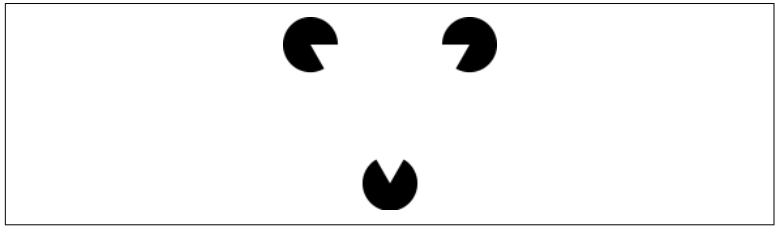

Figure 2-3. The lines of a triangle are not drawn yet you can see the edges of a white triangle

CHAPTER 3
Reactive on the System Level

One Actor is no Actor.
Actors come in Systems.

—Carl Hewitt, *creator of the actor model of concurrent computation*

More than 95 percent of your organization's problems derive from your systems, processes, and methods, not from your individual workers. […] Your people are doing their best, but their best efforts cannot compensate for your inadequate and dysfunctional systems.

Changing the system will change what people do. Changing what people do will not change the system.

—Peter R. Scholtes, *The Leaders Handbook*

In reality, we rarely talk about a single instance of a single application. Instead, we talk about systems, composed of multiple services, possibly implemented using various technologies and each of them having different latency and up-time requirements. As we'll discover in this chapter, many of the concepts and requirements we talked about in the previous chapter translate directly to the system level.

We often are led to think this is solely a technical aspect of systems. I would argue that that's only part of the story. Applications need to scale both in the technical meaning of the word, but also in the *organizational* meaning of it. Distributed systems (microservices

being a good example) allow you to divide responsibilities into multiple applications, and those have their own dedicated teams, with well-defined responsibilities. So, as it turns out, distributed systems allow and help you scale your organization. They allow you to decouple dependencies and get rid of strict dependencies between projects by allowing them to be developed and deployed independently. That independence allows for building the services by different teams, perhaps on the other side of the globe where such a service is being consumed. Once you're distributed, it does not matter how far away in time or space teams or applications are.

It is also important to realize that while Reactive Systems and the way of thinking in terms of messaging is actually pretty *simple*, it does not mean that it's *easy*. The difference—and the importance of not mixing up those two concepts—has been explained by Rich Hickey, the creator of the Clojure programming language, in his RailsConf 2012 keynote.[1] In this presentation, he argued that the "difference" was that "easy" is what is *familiar* to us. For example, even something really complicated becomes easy if you practice it for a long time, such as knowing deep internals and hidden dependencies between various modules of a very complex system you've been working on for the last 10 years. The fact that something is easy for you now does not make it any simpler than it really is—it still is complex. On the other hand, something can be "simple," meaning that the core concept and ideas behind it are pretty simple once you "get" them. But this does not mean that it won't be *hard* to learn it.

So we should be contrasting simple with complex, and easy with hard. Distributed systems, in any shape or form, are hard. The moment you have to deal with more than one computer, or more than one team within an organization, things become harder than if you just had to deal with a single one. In fact, microservices, *are* distributed systems, thus they should not be triviallised. However, the means you can use to communicate can be either simple (e.g., messaging) or complex (e.g., request/response with pooling, pipelining, circuit-breaking and timeouts aborting blocked dead connections).

1 Rich Hickey, "Simplicity Matters" (Speech presented at Rails Conf 2012, Austin, Texas, May 1, 2012), *https://www.youtube.com/watch?v=rI8tNMsozo0*.

In the following section, we'll have a look into communication styles within systems, review their scalability (both technical and organizational), and wrap it up by suggesting how to introduce Reactive Services into an existing code base.

There's More to Life Than Request-Response-JSON-over-HTTP

> *HTTP/2 was meant as a better HTTP/1.1, primarily for document retrieval in browsers for websites. We can do better than HTTP/2 for applications.*[2]
>
> —Ben Christensen, *ReactiveSocket*

Far too often are we seeing very synchronous request-response patterns dominate the communication patterns of our applications. This is not to say REST is a bad thing. In fact, it's possible to implement asynchronous communication patterns using RESTful principles; the only problem here is that, in practice, they rarely are. The last few years have shown that many organizations are solely focused not as much on the REST mantra and ideology as they are on the "request-response JSON over HTTP" way of thinking. Thankfully many teams are slowly realizing that some use cases can be served better by newer upcoming protocols or messaging patterns, as showcased by recent developments of Twitter's Finagle RPC, Google's new GRPC (Google RPC) libraries, and finally Facebook's and Netflix's research into ReactiveSocket. The term "REST" has over the last few years deteriorated to mean JSON-over-HTTP, but it does not have to be this way. The reason I draw this difference is that in the original publication[3] which coins the term REST, it does mention HTTP anywhere, yet somehow developers and architects came to understand REST as a strictly HTTP-bound architectural style. Part of the reason is that Fielding did participate in the URI, HTTP, and HTML IETF working groups, so obviously his work relates to them in some way.

---

2 Ben Christensen, "Reactivesocket/Motivations.md," ReactiveSocket on GitHub, *https://github.com/reactiveSocket/reactivesocket/blob/master/Motivations.md*.

3 Roy Thomas Fielding. "Architectural Styles and the Design of Network-based Software Architectures." (PhD diss., University of California, Irvine, 2000), *https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm*.

Before we talk about communication patterns *between services*, let's remind ourselves that a single *service* does not necessarily mean that it is handled by one *instance* of that service. For example, consider an image resizing service. The entry point is a single API endpoint, and users of the service would refer to it as "the image scaling service" (as in "is the image scaler down again?"). However, *inside* it may well be running multiple instances of the same service and balancing the incoming images across its *internal cluster*. It is important to realize that the tradeoffs we make for the *external* API and interactions of the service may be somewhat different than the ones we make for that service's *internal* communication. This maps directly to bounded contexts from DDD terminology. Within a single bounded context, we're communicating more freely; however, once we need to cross context boundaries, the required consistency guarantees change, depending on with whom and how we're communicating. Those differences, more than anything, would be driven by the fact that the service is usually *owned* by a single team. In other words, we've rediscovered encapsulation, but this time it's a result of organizational boundaries and the fact that internally each such team operates slightly differently. Instead of forcing all communication within a company into the same RESTful communication style, it often is beneficial to loosen up constraints for a service's internal communication, since a single team controlling it can move way faster and use other technologies than the ones that make up its public API. Figure 3-1 illustrates the difference between outside and inside requirements regarding communication styles.

In the excellent paper titled "Data on the outside versus Data on the inside," Pat Helland argues that there is an inherent difference in the requirements data has to face "inside" of a system (i.e., the internal datastore owned by your microservice) and "outside" (i.e., when emitting data representation to consumers of this service).[4] Specifically, he argues that internal data representation be mutable, since, we may want to find more efficient ways to represent the same data (e.g., by changing your SQL schemas.) But the data's external representation should be immutable. As long as users can get their list of contacts, it does not matter for them if internally we had to implement a JOIN or if it was fetched from a pre-calculated cache that we

4 Pat Helland, "Data on the Outside versus Data on the Inside," Proceedings of 2005 CIDR Conference, Asilomar, California, *http://cidrdb.org/cidr2005/papers/P12.pdf*.

mutably refresh every now and then. I'd argue that the same idea applies to communication patterns. Externally, we want to expose a stable, well-defined RESTful API that other teams within (and possibly outside) our company can use, or we can just publish the facts for others to use at will (resulting in a more event-based architecture). Internally, however, we need the communication protocols best suited for our needs. For example, internally we may opt for fire-and-forget[5] messaging using binary message formats.

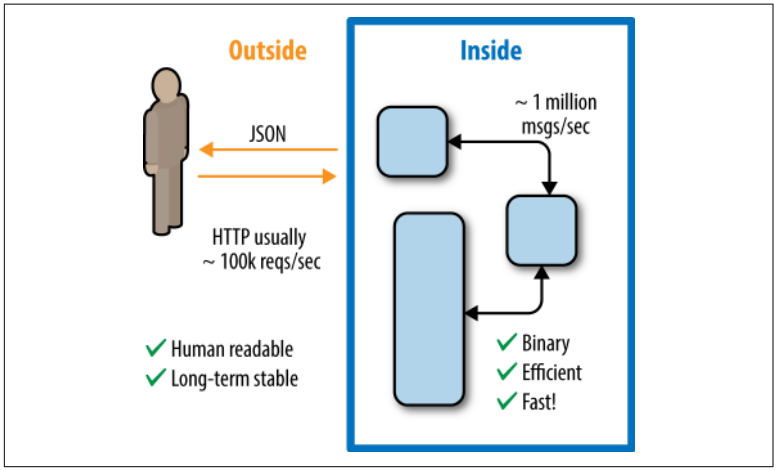

Figure 3-1. Data has different needs on the outside and inside—the same can be said about communication patterns and how we build and evolve APIs

The problem with strictly request-response APIs—RESTful APIs being just one of the examples; SOAP lures developers into a very similar approach as well—is that they are limiting in what developers can do. For example, in the HTTP world, subscriptions to event streams are somewhat of a hack, usually implemented by returning one "infinite" HTTP response that we then feed with line-by-line JSON data. A good example of such APIs are the Twitter Streaming APIs.

When discussing this topic, one may often hear the following counter-argument: "I can totally do async with REST." This state-

5 Fire-and-forget is a messaging pattern in which we do not expect a direct response to the message; as opposed to request-response protocols.

ment is, technically speaking, true. However, the *cost* of implementing it is really exponential when compared with a solution that supports messaging natively.

Let's examine what an asynchronous job submission API looks like when implemented in a RESTful style. First, we'll analyze the control flow, and then we'll look at how many resources need to be allocated throughout its entire lifecycle. REST in today's world usually is implemented on top of HTTP (though it does not have to; the original REST paper never mentioned any ties to HTTP). So to create a job we would perhaps PUT to a `/jobs` service, and it would reply with a `201 Created`, or rather `202 Accepted`, to signal that the request to run the job was accepted. It would also include a `Location` header to tell us where the result of the job will be located once it's done. A nonreactive client would have to then poll that location for the job's status. We also need to keep resources around to remember the mapping from our issued request to the process that handles the polling. Notice that even though the initial request/reply cycle does not tell us anything that we're *really* interested in (e.g., the job's result or progress), we still *must* wait for its response since we have no way to find the job's result later.

The same task can be achieved trivially in messaging-based systems, such as Akka or Erlang, by sending a fire-and-forget message that we want to run a job. Eventually we can include in that message a flag that we want to be informed about its progress, and we're *done* on the sending side. Once the job completes, we'll get a message back with its results. We don't need to care where or when it gets completed. Let's analyze the overhead of what data or state we need to keep around in this solution. We just need an addressable entity that will receive the message (in Erlang terms, it would be a process; in Akka, those are called actors). Those can be very lightweight—approximately 400 bytes each. Although this differs among the various implementations, the key takeaway is that it's very small. That's all the overhead we pay for the reactive interaction with the other system—no connection establishment, no back-and-forth round trips, and no polling.

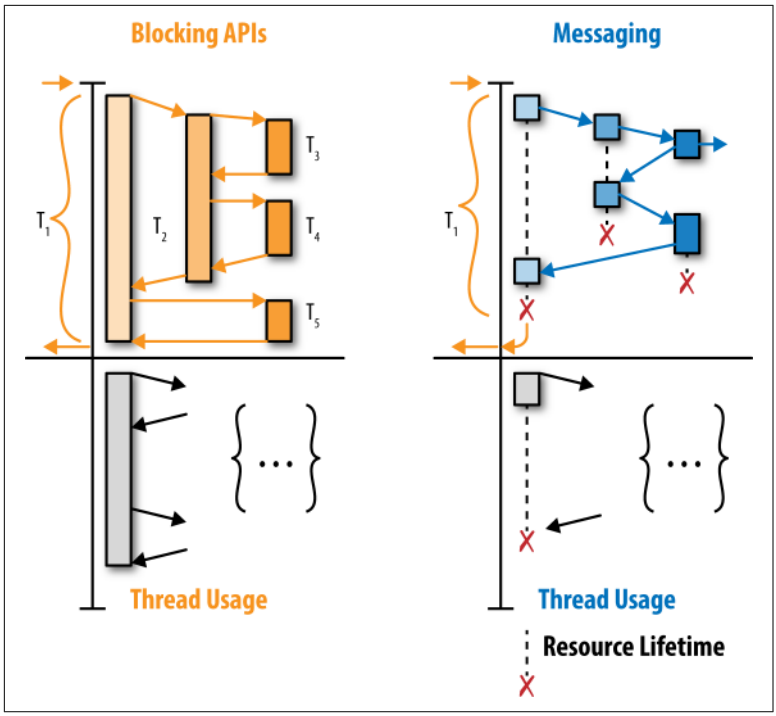

Figure 3-2. Comparing nonasynchronous HTTP call-style APIs with pure messaging protocols. Notice the amount of thread usage while the reactive version is much lower

Alternative patterns in HTTP to address these kinds of issues have been invented over the years, but they are very heavy and simply won't be able to survive the modern-day Internet, one where we have too many jobs, tasks, and clients happening at the same time. But let's address some of these patterns to show their weaknesses in scalability. Proponents of HTTP may suggest that we could have used a long-running HTTP PUT call in the first place, so we'd keep the request "open" until the job has completed.

It is also important to realize that it's not only about scalability. The number of decisions one has to make when designing such APIs with plain HTTP is growing exponentially. Will we block on the call? What method should we use? How do we tell the client to do polling? Should we use WebSockets instead? What about server-sent-events? With messaging, it's trivial—you send a message when you want something, and you get a message when something has

completed. So from that perspective it's also a huge simplification and less headache when designing a system, and we can focus on the actual job instead of debating implementation details.

Surviving the Load…and Shaving the Bill!

The most interesting aspect of Reactive Systems is their ability to elastically scale in the face of varying incoming traffic. Scaling usually serves one of two purposes: either we need to scale *out* (by adding more machines) and *up* (by adding beefier machines), or we need to scale *down*, reducing the number of resources occupied by our application.

The word "elastic" has become more and more prominent; and even though its meaning is very similar to our well-known and loved term "scalability," it is an important shift to notice. Especially during the early days of the cloud, you could easily go all in and end up being over-provisioned to the load you were actually handling. That is because scaling out in the cloud is rather trivial, such as adding new nodes, joining them up in a cluster, or hiding them behind a load balancer. Turns out what's not that trivial is regaining a somewhat reasonable utilization level on the spun-up nodes. This is why the recent trends of containers and cluster orchestrators (such as Mesos) are now on the rise: we've been over-provisioning our applications over the last few years. Over-provisioning by itself is not a bad thing. It gives us a "buffer" that can survive sudden spikes in traffic, surviving while we add more nodes to the cluster (perhaps that new marketing campaign was *really successful*). However, over-provisioning by *a lot* is a huge problem, like a large invoice for things that we didn't really use.

We found that when talking about scalability of systems, developers get fixated on scaling *out* or scaling *up*, and the scaling *down* part is somehow lost. I'd argue that scaling *down* is as important as scaling out/up: after all, once the intense traffic demands have laid off, there's no need to pay those high bills, and of course we want to conserve energy and be environment friendly as well. New tools like Mesos and Kubernetes, together with a container-focused approach to application deployment, are paving the way toward a more ops-less future for normal operation of apps.

An interesting scaling pattern popularized by the likes of Netflix and Gilt (a popular flash-sale site) is predictive scaling, in which we

know when spikes are going to hit so we can proactively provision servers for that period, and once traffic starts going down again, decrease the cluster size incrementally. One might get into a funny word-play here: is "proactive scaling" better than "reactive scaling"? No, not really, since Reactive Services are an enabeler for scaling of services. Proactive scaling, on the other hand, is a technique using which we can decide *when* to scale. For example, a flash-sale site, such as Gilt (which has been very open about its provisioning strategy), has a very high spike in traffic in the middle of the day, exactly when the flash-sale is active.[6] So the proactive part of scaling here means that the nodes can be spun up before the high load hits the servers, because it is predictable. This goes hand-in-hand with these services being reactive—after all, you can't proactively scale an app if you can't scale it at all.

Without Resilience, Nothing Else Matters

> *Resilience is the ability of a substance or object to spring back into shape. The capacity to recover quickly from difficulties.*
>
> —Merriam Webster

We discussed a lot around performance, scalability, messaging patterns, and various other aspects of reactive. Last, but not least, let's now loop back to what, perhaps, is the most important trait of them all: resilience. After all, it does not help to have super fast and scalable system that becomes completely unavailable if anything goes wrong.

After all, it does not matter how fast, shiny and great your system is if it is unavailable. Reactive Systems most typically deal with failure by scaling out through replacing failed nodes with new healthy ones, or simply adding more nodes in the face of increased traffic which would have otherwise overloaded the system. In that sense, those systems can be thought of as *antifragile*, the property of not only being able to survive under stress, but actually *improving* under it.[7]

6 Daniel Bryant, "Scaling Microservices at Gilt with Scala, Docker and AWS," InfoQ, April 26, 2015, *https://www.infoq.com/news/2015/04/scaling-microservices-gilt*.

7 Nassim Nicholas Taleb, *Antifragile: Things That Gain from Disorder* (New York: Random House, 2014).

Another anti-pattern that we implicitly discussed in the chapter about elasticity is that legacy systems sometimes tend to share access to a common expensive resource. On the application level, this would manifest as shared, concurrent access to a mutable state. On the system level, it is much easier to spot, as it often is simply a shared database or similar resource. A common theme of such resources is that all services need to access its shared state in order to function. In fact, the Reddit outage in 2016 was caused precisely by such shared services misbehaving.[8] In other words, it was a potential single point of failure, and took down the entire site down.

Reactive Systems follow the "own your own data" pattern, which we already discussed when talking about actors. It is the same pattern, but on a system level now. If the systems had internally kept more state about their surroundings, instead of relying on ZooKeeper, they perhaps would have been able to survive the outage (for example, an "uncertainty" timeout could have bbn triggered if the shared resource failed, giving the ops team more time to react before these servers had shut down). Systems which are strictly peer-to-peer and masterless are even better as they inherently structure themselves to avoid the single, special-resource problem.

8 Gooeyblob, "Why Reddit Was down on Aug 11," Reddit, August 11, 2016.

CHAPTER 4
Building Blocks of Reactive Systems

We build too many walls and not enough bridges.[1]

—Joseph Fort Newton

Systems never live in a vacuum, nor are they composed of identical parts. The real world is much more diverse, and attempting to ignore this fact inevitably leads to disappointment—and in the worst cases, failure.

Instead, we regognize this diversity and turn it into a strength, where various parts of the system can be specialized in the areas they are built for. As explained in Chapter 3, these components can be added one by one to gradually move the architecture of your system toward reactive principles. The journey to a Reactive System may be a long one, but it's one worth taking since it'll improve your architecture in ways we discussed throughout this report. It also is important to not become discouraged and to carefully judge where in your system it's worth making the move and where it's not. For example, if tasked to build a new functionality, think about whether it would be possible to build it as a reactive microservice and integrate it with the existing system, instead of extending the legacy codebase. For cases where you're not able to build a greenfield sys-

1 While often misattributed to Isaac Newton, the origin of this quote goes back to Joseph Fort Newton's "The One Great Church: Adventures of Faith" (1948), according to Wikipedia.

tem, apply the *Strangler pattern* as discussed previously, and pick the building blocks you need for that specific part of the system.

So, what are the other building blocks one can use to build Reactive Systems? We discussed a few of them implicitly already, but we did not have the chance to touch upon all of the various techniques.

For example, as with the case of streaming, we did discuss building and consuming streaming APIs with Akka Streams and Akka HTTP or RxJava. But we did not talk about moving away from nightly batch jobs toward more reactive pipelines of streaming data analysis. These techniques, such as Apache Spark, Flink, or Gear Pump, allow you to improve the responsiveness of your applications.[2] Instead of waiting for the nightly batch to complete to send out reports to customers, their data can be processed ad hoc or in a streaming fashion, giving quicker feedback to customers.

Other topics that we barely had the space to mention are cluster schedulers and platforms such as Mesos and Mesosphere DC/OS. Once we build Reactive Services, it's now time to deploy them somewhere. In order to take advantage of the possibilities given to us by Reactive Services, we need a fabric that supports these capabilities. A good example here is Mesos, which allows us to deploy either on-premise or in the cloud and standardize the way we allocate instances of our applications to resources in our cloud. Thanks to our reactive applications being built with scalability and resilience in mind, schedulers such as Mesos can kill and start instances of our apps as the pressure on the system changes.

Introducing Reactive in Real-World Systems

> *Fully async architecture has [measurable] benefits. However I don't expect to see a software system like that. Instead, we deal with mixed-codebases.*[3]
>
> —Ben Christensen

The reality of our daily work is that we rarely have the opportunity to start completely fresh. Even if we do have greenfield projects, to

2 Apache Gearpump, mostly backed and driven by Intel, the Google MillWheel inspired large scale real-time stream processing engine.

3 Ben Christensen, "Applying reactive Programming with Rx" (Presented at GOTO Chicago 2015, Jul 15, 2015), *https://www.youtube.com/watch?v=8OcCSQS0tug*.

be relevant, they have to integrate with existing systems, and those may not have been built with the level of resilience and semantics we would have hoped for in this day and age. We don't believe that looking down on legacy systems is a healthy approach. Legacy systems are in place because they succeeded at delivering something in their time—something important enough that kept them alive and running up until this day. Once we come to accept this simple truth, one can look for a productive way to move forward and adopt new techniques, such as reactive architecture. In this section, we aim to highlight a successful and proven approach for introducing new techniques into an existing ecosystem.

One of the ways to introduce change into existing code bases, especially when moving to different paradigms or languages, is to use the "Ivy Pattern" (sometimes referred to as the "Strangler Pattern"[4]), which has been used in multiple projects in the field, however not often knowingly. The concept is rather simple, and is illustrated in Figure 4-1.

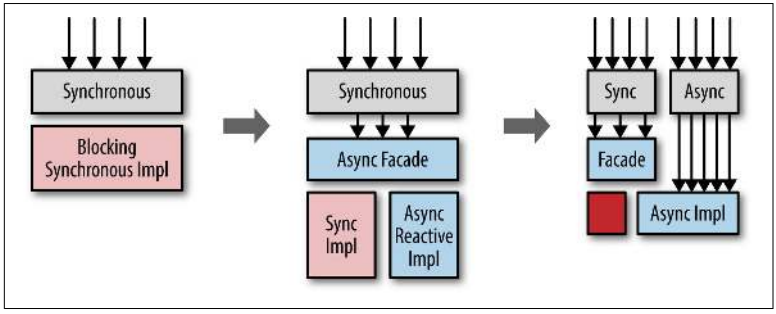

Figure 4-1. Applying the strangler pattern to hide old implementations behind a new Reactive API, and introducing new features in the Reactive part of the system, migrating old functionality to the new core only on an as-needed basis

The idea here is to avoid rewriting the legacy code that works until you hit an actual problem with it, at which point you can consider rewriting it in the new style (but you're not forced to.) This pattern received its name from how ivy plants strangle trees on which they grow. The idea here is the same, the new system grows around the

4 Martin Fowler, "StranglerApplication," Martinfowler.com, June 29, 2004, *http://martinfowler.com/bliki/StranglerApplication.html*

old one, and without the need to rewrite anything in the old system; meanwhile, the rest of the system can move on to the more reactive style of development, rewriting the internal if (and when) needed, and not sooner. This helps to avoid high-risk "big bang" rewrites, which often end up in disastrous failures—mostly due to underestimating or not understanding the previous codebase.

Reactive, an Architectural Style for Present and Future

> *I know; you* did *send me back to the future.*
> *But I'm back. I'm back* from *the future.*
>
> —Marty McFly, *Back to the Future*

It is important to realize early on that reactive is not about a single specific technology or library. Instead of spending this book on explaining a specific library, we spent the space and time to dive into the core concepts behind many of them. With this knowledge you should be able to continue your journey and decide on your own which tools suit you best and will help you move toward this new programming paradigm. It is also very interesting to see that we've learned from past mistakes[5] when developers tried to hide distributed systems, as if they're just a special case of local execution. A mistake made with synchronous RPC systems such as CORBA and heavyweight SOA processes. We think that we've learned some good lessons from these experiments and now with embracing the network more than ever before. And instead of avoiding failure at all costs, we embrace it in our systems, letting them scale and accommodate it.

Of course, there are some great tools available; but used incorrectly, even the best tool or library won't solve the problem by itself. You may recall the same situation when Agile was taking over the IT industry, and for good reason. But many teams back then applied "the daily standup" without thinking too much about how it should help their team. Instead, they applied it rigidly "by the book," and

5 Steve Vinoski, "Convenience Over Correctness", IEEE Internet Computing IEEE Internet Comput. 12, no. 4 (2008), doi:10.1109/mic.2008.75; Waldo et al.; "A Note on Distributed Computing", IEEE Micro, 1994; and Anne Thomas Manes, "SOA Is Dead; Long Live Services", Application Platform Strategies Blog, January 5, 2009.

when it didn't solve the rest of the broken process they had in place, they blamed Agile as a whole. Nowadays it is hard to imagine software development without some of the Agile and Lean practices; however, as with any methodology, it still remains open to misinterpretation.

Thus, it is important when adopting reactive to give it some thought and to understand the principles before moving forward. I hope this report will prove to be useful in doing exactly that and in guiding you and your teams toward making the best decisions for the systems you build.

About the Author

Konrad Malawski is senior developer in the Akka team at Lightbend, a distributed systems toolkit for the JVM. Before joining Lightbend (previously Typesafe), he worked at eBay on a content delivery platform in London. He is a leading contributor to the current Reactive Streams TCK, and other open source projects. He has founded and helps run multiple user groups, including the GeeCON conference in Poland. When he's not coding, Konrad spreads the joy of computer science by organizing a white paper reading club. He was named a JavaOne RockStar in 2015.