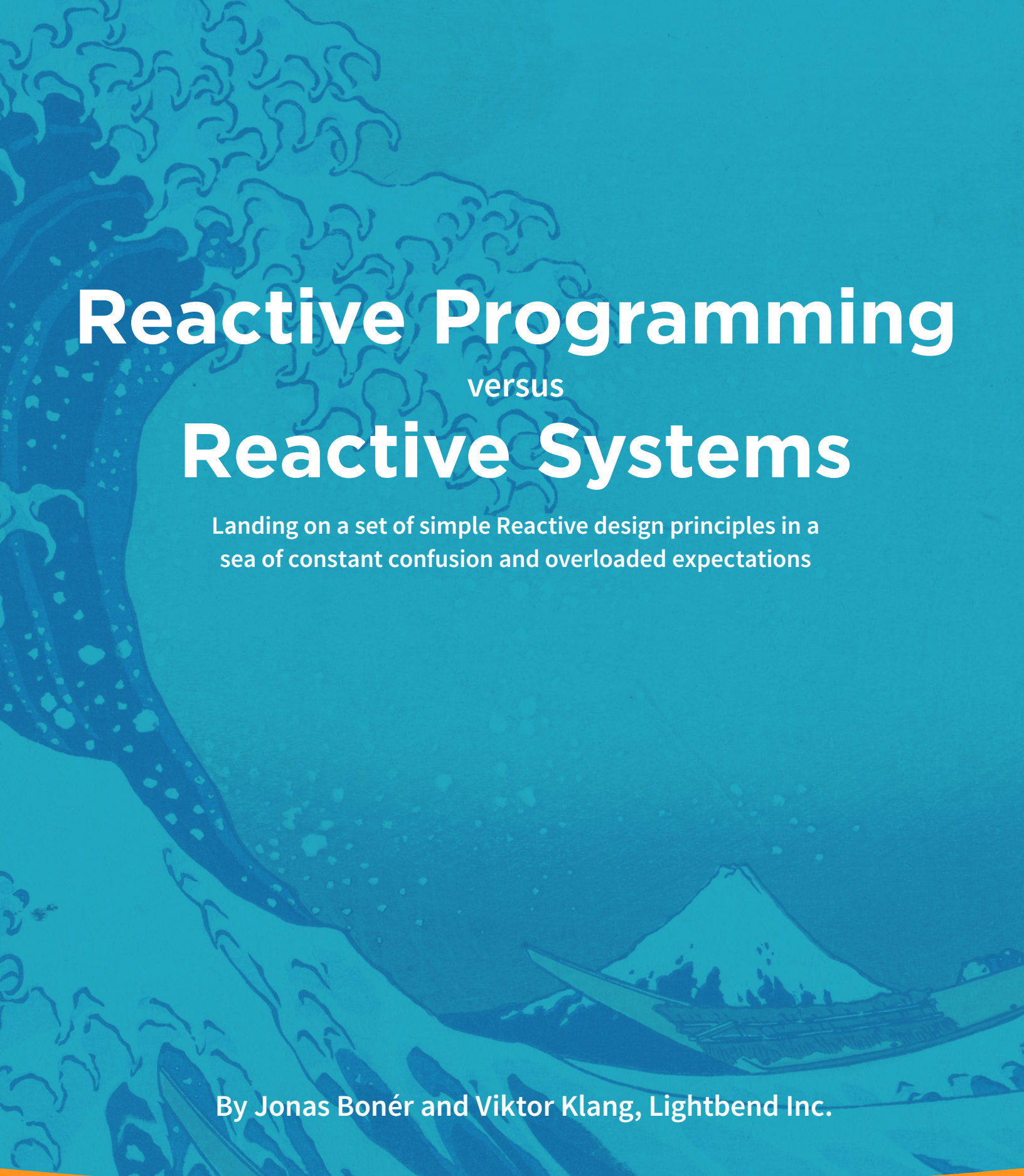


Reactive Programming versus Reactive Systems

Landing on a set of simple Reactive design principles in a sea of constant confusion and overloaded expectations

By Jonas Bonér and Viktor Klang, Lightbend Inc.

Lightbend

Table of Contents

Executive Summary

Since co-authoring the **Reactive Manifesto** in 2013, we've seen the topic of Reactive go from being a virtually unacknowledged technique for constructing applications—used by only fringe projects within a select few corporations—to become part of the overall platform strategy in numerous big players in the middleware field.

The goal of this white paper is to define and clarify the different aspects of "Reactive" by looking at the differences between writing code in a Reactive Programming style, and the design of Reactive Systems as a cohesive whole.

Key Takeaways (TL;DR)

- Since 2015, and particularly in 2016, there has been a huge growth in interest in Reactive—from both commercial middleware vendors and users.
- Reactive Programming is a distinct subset of Reactive Systems at the implementation level.
- Reactive Programming offers productivity for Developers—through performance and resource efficiency—at the component level for internal logic and dataflow management.
- Reactive Systems offers productivity for Architects and DevOps—through resilience and elasticity—at the system level, for building "Cloud Native"[1] or other large-scale distributed systems.
- It is highly beneficial to use Reactive Programming within the components of a Reactive System.
- It is highly beneficial to use Reactive Systems to create the system around the components written using Reactive Programming.

[1] A rather undefined term, generally referring to applications which do not depend on underlying OS features such as a file system, but instead use typically configurable endpoints, allowing them to run in a virtualized environment like the Cloud.

Reactive—A Set Of Design Principles

One recent indicator of success is that *Reactive* has become an overloaded term and is now being associated with several different things to different people—in good company with words like "streaming", "lightweight", and "real-time."

From the perspective of this white paper, "Reactive" is a set of design principles for creating cohesive systems. It's a way of thinking about systems architecture and design in a distributed environment where implementation techniques, tooling, and design patterns are components of a larger whole.

Consider the following analogy: an athletic team (e.g. football, basketball, etc.) is often composed of exceptional individuals. Yet, losing to an "inferior" team is nonetheless common when a team comes together and something doesn't click. This lack of synergy to operate effectively as a team is what we see.

This analogy illustrates the difference between a set of individual Reactive services cobbled together without thought—even though individually they're great—and a Reactive System.

In a Reactive System, it's the interaction between the individual parts that makes all the difference, which is the ability to operate individually yet act in concert to achieve their intended result.

A Reactive System is based on an architectural style that allows these multiple individual services to coalesce as a single unit and react to its surroundings while remaining aware of each other—this could manifest in being able to scale up/down, load balance and even take some of these steps proactively.

Thus, we see that it's possible to write a single application in a Reactive style (i.e. using Reactive Programming); however, that's merely one piece of the puzzle. Though each of the above aspects may seem to qualify as "Reactive," in and of themselves they do not make a *system* Reactive.

When people talk about Reactive in the context of software development and design, they generally mean one of three things:

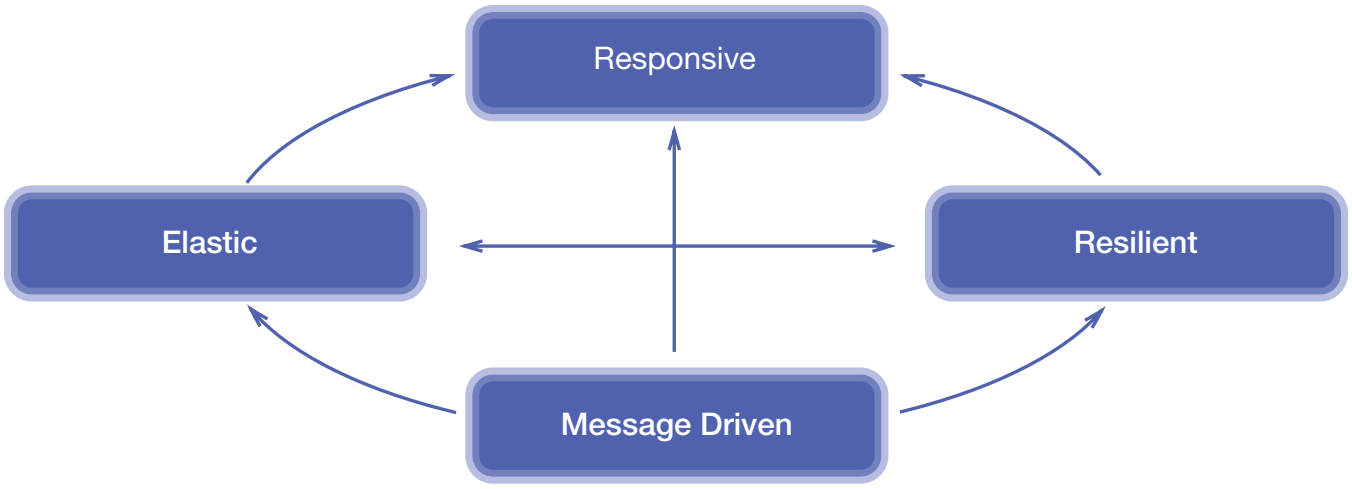
- Reactive Systems (architecture and design)
- Reactive Programming (declarative event-based)
- Functional Reactive Programming (FRP)

We'll examine what each of these practices and techniques mean, with emphasis on the first two. More specifically, we'll discuss when to use them, how they relate to each other, and what you can expect the benefits from each to be—particularly in the context of building systems for multicore, Cloud, and Mobile architectures.

In 2013, after having long experience of creating, maintaining and operationalizing Akka-based systems—and seeing the immense benefits compared to traditional approaches to solving concurrency and distribution—the idea of formalizing the experiences and lessons learned was sparked under the umbrella of the **Reactive Manifesto**.

The main driver behind modern systems is the notion of Responsiveness: the acknowledgement that if the client/customer does not get value in a timely fashion then they will go somewhere else. Fundamentally there is no difference between not getting value and not getting value when it is needed.

In order to facilitate Responsiveness, two challenges need to be faced: being Responsive under failure, defined as Resilience, and being Responsive under load, defined as Elasticity. The Reactive Manifesto prescribes that in order to achieve this, the system needs to be Message-driven.



The four tenets of the Reactive Manifesto

In 2016, several major vendors in the JVM space have announced core initiatives to embrace *Reactive Programming*—this is a tremendous validation of the problems faced by companies today.

Undertaking this change in direction from traditional programming techniques is a big and challenging task; having to maintain compatibility with pre-existing technologies as well as shepherding the user base to a different mindset—as well as building out internal developer and operational experience. As such, the investment by these companies is non-trivial and it needs no mention that this is a large engineering challenge.

While there seems to be much activity in the Reactive Programming space, at the systems architecture level it will take time to build up architectural and operational experience—something which is not automatically solved by adopting a different programming paradigm. It will be interesting to see what will

emerge on that front given the growing mindshare behind the Reactive Manifesto—the need to build Reactive Systems.

Let's start by talking about Functional Reactive Programming, and why we chose to exclude it from further discussions in this article.

Functional Reactive Programming (FRP)

Functional Reactive Programming, commonly called "FRP," is frequently misunderstood. FRP was very **precisely defined** 20 years ago by Conal Elliott. The term has most recently been used incorrectly[2] to describe technologies like Elm, Bacon.js, and Reactive Extensions (RxJava, Rx.NET, RxJS) amongst others. Most libraries claiming to support FRP are almost exclusively talking about *Reactive Programming* and it will therefore not be discussed further.

Reactive Programming

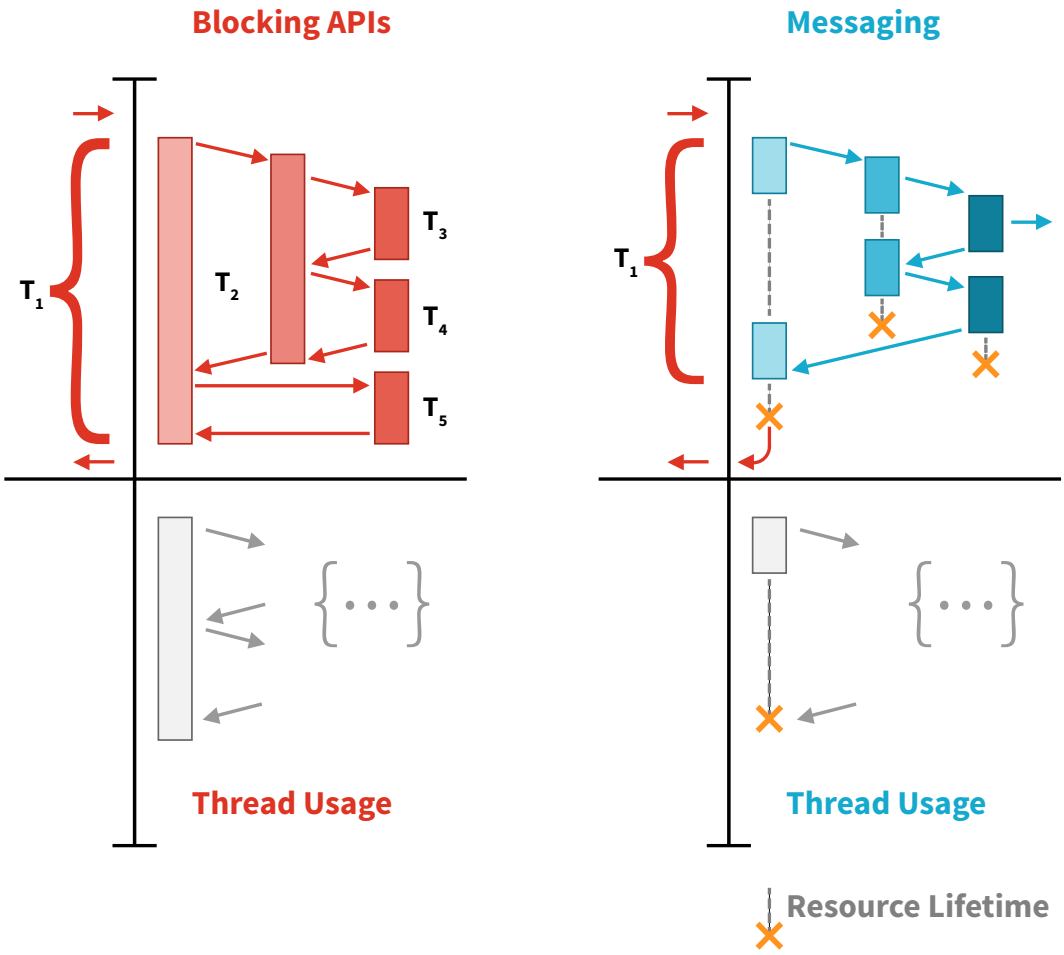
Reactive Programming, not to be confused with *Functional Reactive Programming*, is a subset of Asynchronous Programming and a paradigm where the availability of new information drives the logic forward rather than having control flow driven by a thread-of-execution.

It supports decomposing the problem into multiple discrete steps where each can be executed in an asynchronous and nonblocking fashion, and then be composed to produce a workflow—possibly unbounded in its inputs or outputs.

Asynchronous is defined by the Oxford Dictionary as "*not existing or occurring at the same time*", which in this context means that the processing of a message or event is happening at some arbitrary time, possibly in the future.

This is a very important technique in Reactive Programming since it allows for **non-blocking** execution—where threads of execution competing for a shared resource don't need to wait by blocking (preventing the thread of execution from performing other work until current work is done), and can as such perform other useful work while the resource is occupied. Amdahl's Law[3] tells us that contention is the biggest enemy of scalability, and therefore a Reactive program should rarely, if ever, have to block.

[2] According to Conal Elliott, the inventor of FRP, in this **presentation**.
[3] **Amdahl's Law** shows that the theoretical speedup of a system is limited by the serial parts, which means that the system can experience diminishing returns as new resources are added.



Synchronous, blocking communication (left) is resource inefficient and easily bottlenecked.
The Reactive approach (right) reduces risk, conserves valuable resources, and requires less hardware/infrastructure.

Reactive Programming is generally *Event-driven*, in contrast to Reactive Systems, which are *Message-driven*—the distinction between Event-driven and Message-driven is clarified in the next section.

The Application Program Interface (API) for Reactive Programming libraries are generally either:

- Callback-based—where anonymous, side-effecting callbacks are attached to event sources, and are being invoked when events pass through the dataflow chain.
- Declarative—through functional composition, usually using well established combinators like *map*, *filter*, *fold* etc.

Most libraries provide a mix of these two styles, often with the addition of stream-based operators like windowing, counts, triggers, etc.

It would be reasonable to claim that Reactive Programming is related to **Dataflow Programming**, since the emphasis is on the flow of data rather than the flow of control.

Examples of programming abstractions that support this programming technique are:

- **Futures/Promises**—containers of a single value, many-read/single-write semantics where asynchronous transformations of the value can be added even if it is not yet available.
- Streams—as in **Reactive Streams**: unbounded flows of data processing, enabling asynchronous, non-blocking, back-pressured transformation pipelines between a multitude of sources and destinations.
- **Dataflow Variables**—single assignment variables (memory-cells) which can depend on input, procedures and other cells, so that changes are automatically updated. A practical example is spreadsheets—where the change of the value in a cell ripples through all dependent functions, producing new values "downstream."

Popular libraries supporting the Reactive Programming techniques on the JVM include, but are not limited to, Akka Streams, Ratpack, Reactor, RxJava and Vert.x. These libraries implement the Reactive Streams specification, which is a standard for interoperability between Reactive Programming libraries on the JVM, and according to its own description is *"...an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure."*

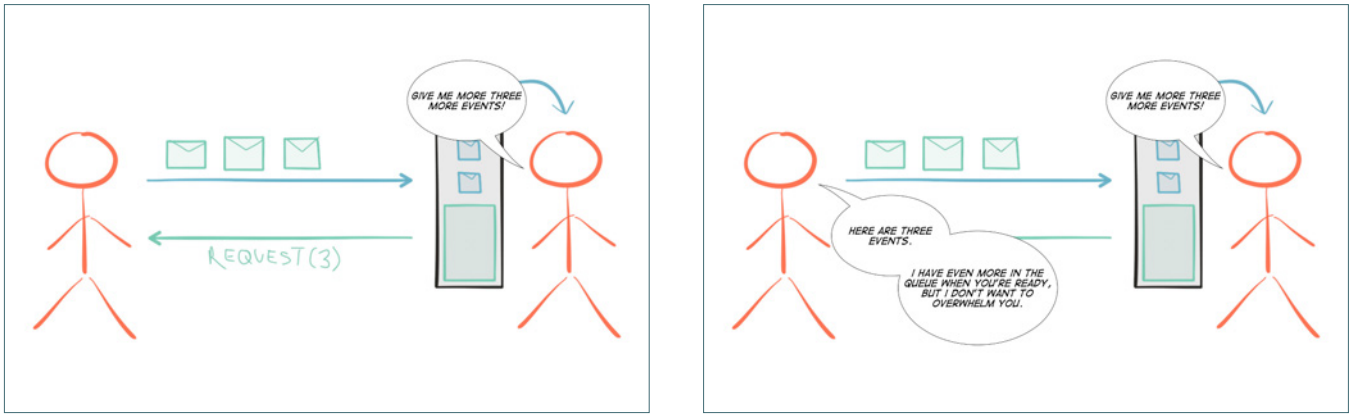
The Benefits (And Limitations) Of Reactive Programming

The primary benefits of Reactive Programming are: increased utilization of computing resources on multicore and multi-CPU hardware; and increased performance by reducing serialization points as per Amdahl's Law and, by extension, *Günther's Universal Scalability Law*[4].

A secondary benefit is one of developer productivity as traditional programming paradigms have all struggled to provide a straightforward and maintainable approach to dealing with asynchronous and nonblocking computation and IO. Reactive Programming solves most of the challenges here since it typically removes the need for explicit coordination between active components.

[4] Neil Günter's **Universal Scalability Law** is an essential tool in understanding the effects of contention and coordination in concurrent and distributed systems, and shows that the cost of coherency in a system can lead to negative results, as new resources are added to the system.

Where Reactive Programming shines is in the creation of components and composition of workflows. In order to take full advantage of asynchronous execution, the inclusion of **back-pressure** is crucial to avoid over-utilization, or rather unbounded consumption of resources.



To ensure steady state in terms of data flow, pull-based back-pressure sends demand flowing upstream and receives messages flowing downstream, which avoids the producer overwhelming the consumer(s). Images by Kevin Webber (@kvnwbbr).

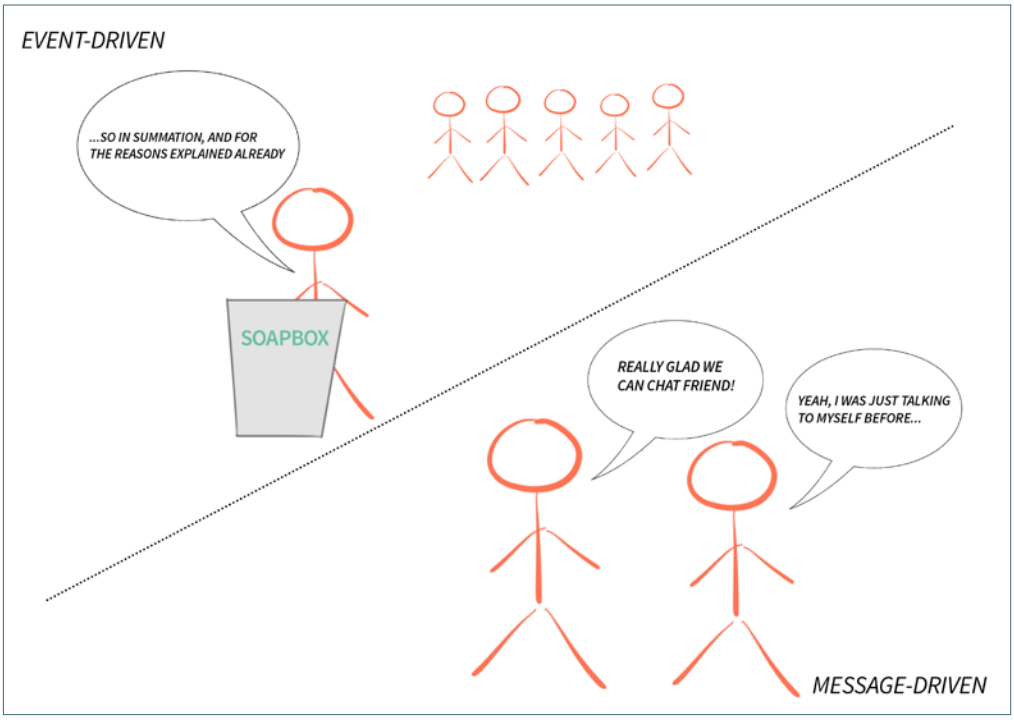
But even though Reactive Programming is a very useful piece when constructing modern software, in order to reason about a system at a higher level one has to use another tool: Reactive Architecture—the process of designing Reactive Systems. Furthermore, it is important to remember that there are many programming paradigms and Reactive Programming is but one of them, so just as with any tool, it is not intended for any and all use-cases.

Event-Driven VS Message-Driven

As mentioned previously, Reactive Programming—focusing on computation through ephemeral dataflow chains—tend to be *Event-driven*, while Reactive Systems—focusing on resilience and elasticity through the communication, and coordination, of distributed systems—is ***Message-driven***[5](also referred to as Messaging).

The main difference between a Message-driven system with long-lived addressable components, and an Event-driven dataflow-driven model, is that Messages are *inherently directed*, Events are not. **Messages have a clear, single, destination; while Events are facts for others to observe**. Furthermore, messaging is preferably asynchronous, with the sending and the reception decoupled from the sender and receiver respectively.

[5] Messaging can be either synchronous (requiring the sender and receiver to be available at the same time) or asynchronous (allowing them to be decoupled in time). Discussing the semantic differences is out scope for this white paper.



While event-driven communication takes a "soapbox" approach, broadcasting facts (events) for others to observe (if they are listening), message-driven communication has an addressable recipient and a single purpose.

The glossary in the Reactive Manifesto **defines the conceptual difference as**:

> *A message is an item of data that is sent to a specific destination. An event is a signal emitted by a component upon reaching a given state. In a message-driven system addressable recipients await the arrival of messages and react to them, otherwise lying dormant. In an event-driven system notification listeners are attached to the sources of events such that they are invoked when the event is emitted.* ***This means that an event-driven system focuses on addressable event sources while a message-driven system concentrates on addressable recipients.***

Messages are needed to communicate across the network and forms the basis for communication in distributed systems, while Events, on the other hand, are emitted locally. It is common to use Messaging under the hood to bridge an Event-driven system across the network by sending Events inside Messages. This allows maintaining the relative simplicity of the Event-driven programming model in a distributed context and can work very well for specialized and well scoped use-cases (e.g., AWS Lambda, Distributed Stream Processing products like Spark Streaming, Flink, Kafka and Akka Streams over Gearpump, and Distributed Publish Subscribe products like Kafka and Kinesis).

However, there is a trade-off: what one gains in abstraction and simplicity of the programming model, one loses in terms of control.

Messaging forces us to embrace the reality and constraints of distributed systems—things like partial failures, failure detection, dropped/duplicated/reordered messages, eventual consistency, managing multiple concurrent realities, etc.—and tackle them head on instead of hiding them behind a leaky abstraction—pretending that the network is not there—as has been done too many times in the past (e.g. EJB, **RPC**, **CORBA**, and **XA**).

These differences in semantics and applicability have profound implications in the application design, including things like *resilience, elasticity, mobility, location transparency* and *management* of the complexity of distributed systems, which will be explained further in this white paper.

In a Reactive System, especially one which uses Reactive Programming, both events and messages will be present—**as one is a great tool for communication (messages), and another is a great way of representing facts (events)**.

Reactive Systems And Architecture

Reactive Systems—as defined by the Reactive Manifesto—is a set of architectural design principles for building modern systems that are well prepared to meet the increasing demands that applications face today.

The principles of Reactive Systems are most definitely not new, and can be traced back to the 70s and 80s and the seminal work by Jim Gray and Pat Helland on the **Tandem System** and Joe Armstrong and Robert Virding on **Erlang**. However, these people were ahead of their time and it's been only in the last 5-10 years that the technology industry has been forced to rethink current "best practices" for enterprise system development. This means learning to apply the hard-won knowledge of the Reactive principles on today's world of multicore, Cloud Computing and the Internet of Things.

The foundation for a Reactive System is *Message-Passing*, which creates a temporal boundary between components which allows them to be decoupled in *time*—this allows for concurrency—and *space*—which allows for distribution and mobility. This decoupling is a requirement for full **isolation** between components, and forms the basis for both *Resilience* and *Elasticity*.

From Programs To Systems

We are no longer building programs—end-to-end logic to calculate something for a single operator—as much as we are building systems.

The world is becoming increasingly interconnected. Systems are complex by definition—each consisting of a multitude of components, who in and of themselves also can be systems—which mean software is increasingly dependant on other software to function properly.

The systems we create today are to be operated on computers small and large, few and many, near each other or half a world away. And at the same time, users' expectations have become harder and harder to meet as everyday human life is increasingly dependant on the availability of systems to function smoothly.

In order to deliver systems that users—and businesses—can depend on, they have to be *Responsive*, since it doesn't matter if something provides the correct response if the response is not available when it is needed. In order to achieve this, we need to make sure that Responsiveness can be maintained under failure (*Resilience*) and under dynamically-changing load (*Elasticity*). To make that happen, we make these systems *Message-Driven*, and we call them *Reactive Systems*.

The Resilience of Reactive Systems

Resilience is about *Responsiveness under failure* and is an inherent functional property of the system, something that needs to be designed for, and not something that can be added in retroactively.

Resilience is beyond Fault-tolerance—it's not about graceful degradation—even though that is a very useful trait for systems—but about being able to fully recover from failure: to self-heal.

This requires component isolation and containment of failures in order to avoid failures spreading to neighbouring components—resulting in, often catastrophic, cascading failure scenarios.

So the key to building Resilient, self-healing systems is to allow failures to be: contained, reified as messages, sent to other components (that act as supervisors), and managed from a safe context outside the failed component. Here, being Message-driven is the enabler: moving away from strongly coupled, brittle, deeply nested synchronous call chains that everyone learned to suffer through…or ignore. The idea is to decouple the management of failures from the call chain, freeing the client from the responsibility of handling the failures of the server.

The Elasticity of Reactive Systems

Elasticity is about *Responsiveness under load*—meaning that the throughput of a system scales up or down (i.e. adding or removing cores on a single machine) as well as in or out (i.e. adding or removing nodes/machines in a data center) automatically to meet varying demand as resources are proportionally added or removed. It is the essential element needed to take advantage of the promises of Cloud Computing: allowing systems to be resource efficient, cost-efficient, environmentally-friendly and pay-per-use.

Systems need to be adaptive—allowing for intervention-less auto-scaling, replication of state and behavior, load-balancing of communication, failover and upgrades, all without rewriting or even reconfiguring the system. The enabler for this is *Location Transparency*: the ability to scale the system in the same way, using the same programming abstractions, with the same semantics, *across all dimensions of scale*—from CPU cores to data centers.

As the Reactive Manifesto **puts it**:

> *One key insight that simplifies this problem immensely is to realize that we are all doing distributed computing. This is true whether we are running our systems on a single node (with multiple independent CPUs communicating over the QPI link) or on a cluster of nodes (with independent machines communicating over the network). Embracing this fact means that there is no conceptual difference between scaling vertically on multicore or horizontally on the cluster.*
>
> *This decoupling in space [..], enabled through asynchronous message-passing, and decoupling of the runtime instances from their references is what we call Location Transparency.*

So no matter where the recipient resides, we communicate with it in the same way. The only way that can be done semantically equivalent is via Messaging.

The Productivity of Reactive Systems

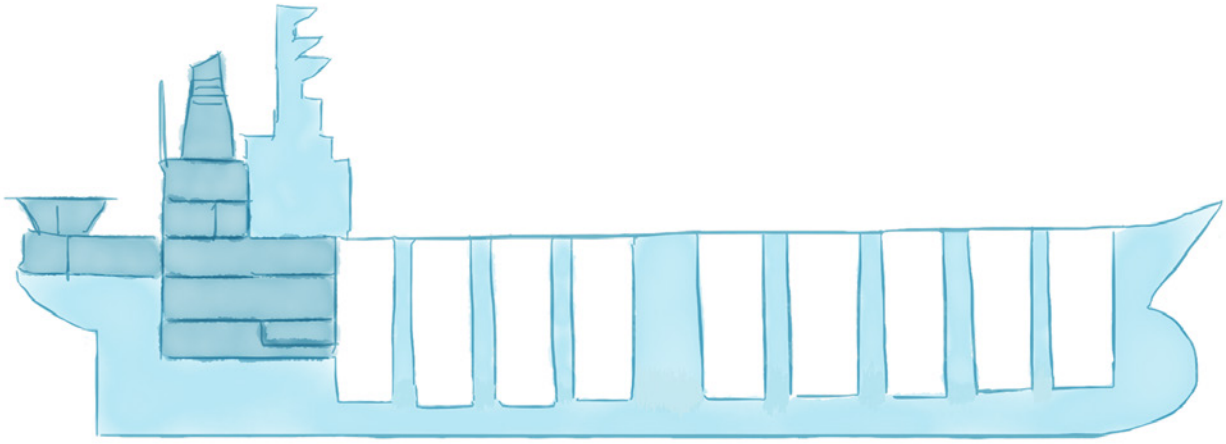
As most systems are inherently complex by nature, one of the most important aspects is to make sure that a system architecture will impose a minimal reduction of productivity, both in the development and maintenance of components, while at the same time reducing *accidental complexity* to a minimum.

This is important, since during the lifecycle of a system—if not properly designed—it will become harder and harder to maintain, and require an ever increasing amount of time and effort to understand in order to localize and to rectify problems.

Reactive Systems represent the most *productive* system architecture that we know of (in the context of multicore, Cloud and Mobile architectures):

- Isolation of failures offer **bulkheads** between components, preventing failures from cascading and limiting the scope and severity of failures.
- Supervisor hierarchies offer multiple levels of defences paired with self-healing capabilities, which removes a lot of transient failures from ever incurring any operational cost to investigate.
- Message-passing and location transparency allow for components to be taken offline and replaced or rerouted without affecting the end-user experience. This reduces the cost of disruptions, their relative urgency, and also the resources required to diagnose and rectify.
- Replication reduces the risk of data loss, and lessens the impact of failure on the availability of retrieval and storage of information.

Elasticity allows for conservation of resources as usage fluctuates, allowing for minimizing operational costs when load is low, and minimizing the risk of outages or urgent investment into scalability as load increases.



Though done poorly in the Titanic, bulkheading has long been used in the ship construction industry to avoid cascading failures from affecting other operations.

Thus, Reactive Systems allows for the creation systems that cope well under failure, varying load and change over time—all while offering a low cost of ownership over time.

How Does Reactive Programming Relate To Reactive Systems?

Reactive Programming is a great technique for managing internal logic and dataflow transformation, locally within the components, as a way of optimizing code clarity, performance and resource efficiency. Reactive Systems, being a set of architectural principles, puts the emphasis on distributed communication and gives us tools to tackle resilience and elasticity in distributed systems.

One common problem with only leveraging Reactive Programming is that its tight coupling between computation stages in an Event-driven callback-based or declarative program makes *Resilience* harder to achieve because its transformation chains are often ephemeral and its stages—the callbacks or combinators—are anonymous, i.e. not addressable.

This means that they usually handle success or failure directly *without signalling it to the outside world*. This lack of addressability makes the recovery of individual stages harder to achieve as it is typically unclear where exceptions should, or even could, be propagated. As a result, failures are tied to ephemeral client requests instead of to the overall health of the component—if one of the stages in the dataflow chain fails, then the whole chain needs to be restarted, and the client notified. This is in contrast to a Message-driven Reactive System, which has the ability to self-heal without necessitating notifying the client.

Another contrast to the Reactive Systems approach is that pure Reactive Programming allows decoupling in *time*, but not *space* (unless leveraging Message-passing to distribute the dataflow graph under the hood, across the network, as discussed previously).

Decoupling in time allows for concurrency, but it is decoupling in space that allows for distribution, and mobility—allowing for not only static but also dynamic topologies—which is essential for Elasticity.

A lack of location transparency makes it hard to scale out a program purely based on Reactive Programming techniques adaptively in an elastic fashion and therefore requires layering additional tools on top, such as a Message Bus, Data Grid or bespoke network protocols. This is where the Message-driven approach of Reactive Systems shines, since it is a communication abstraction that maintains its programming model and semantics across all dimensions of scale, and therefore reduces system complexity and cognitive overhead.

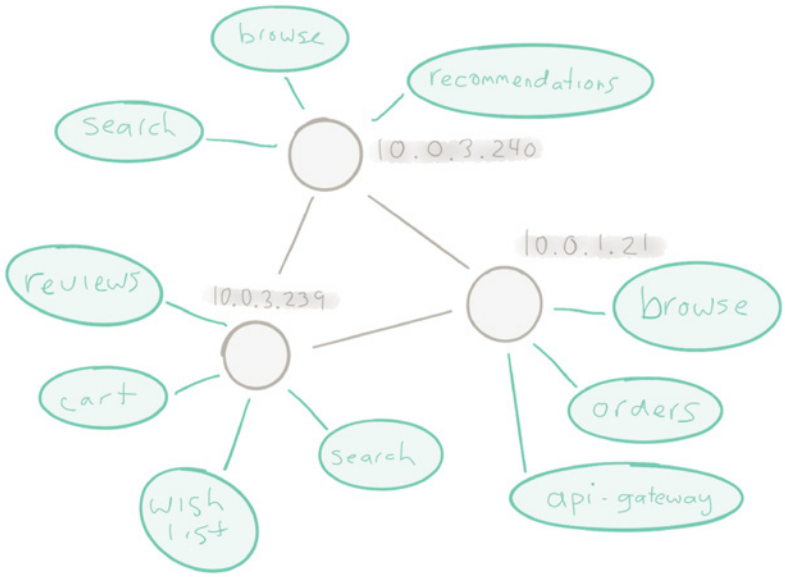
A commonly cited problem of callback-based programming is that while writing such programs may be comparatively easy, it can have real consequences in the long run.

For example, systems based on anonymous callbacks provide very little insight when you need to reason about them, maintain them, or most importantly figure out what, where and why production outages and misbehavior occur.

Libraries and platforms designed for Reactive Systems (such as the **Akka** project and the **Erlang** platform) learned this lesson long ago and are relying on long-lived addressable components that are easier to reason about for the future. When failures occur, the component is uniquely identifiable along with the message that caused the failure. With the concept of addressability at the core of the component model, monitoring solutions have a *meaningful* way to present data that is gathered—leveraging the identities that are propagated.

The choice of a good programming paradigm, one that enforces things like addressability and failure management, has proven to be invaluable in production, as it is designed with the harshness of reality in mind, to expect and embrace failure rather than the lost cause of trying to prevent it.

All in all, Reactive Programming is a very useful implementation technique, which can be used in a Reactive Architecture. Remember that it will only help manage one part of the story: dataflow management through asynchronous and nonblocking execution—usually only within a single node or service. Once there are multiple nodes, there is a need to start thinking hard about things like data consistency, cross-node communication, coordination, versioning, orchestration, failure management, separation of concerns and responsibilities etc.—i.e. system architecture.



While Reactive Programming focuses on asynchronous, nonblocking dataflow management between a single node or service, complex Reactive System architectures need far more to successfully deploy multiple services across nodes and clusters. Image by Kevin Webber (@kvnwbbr).

Therefore, to maximize the value of Reactive Programming, use it as one of the tools to construct a Reactive System. Building a Reactive System requires more than abstracting away OS-specific resources and sprinkling asynchronous APIs and **Circuit Breakers** on top of an existing, legacy, software stack. It should be about embracing the fact that you are building a distributed system comprising multiple services—that all need to work together, providing a consistent and responsive experience, not just when things work as expected but also in the face of failure and under unpredictable load.

How Does Reactive Programming & Systems Relate To Fast Data Streaming?

Fast Data Streaming (Distributed Stream Processing)[6] is, from a user's perspective, generally Event-driven using local and stream-based, abstractions exposing end-user APIs that rely on *Reactive Programming* constructs like functional combinators and callbacks.

Then, underneath the end-user API, it typically uses Message-passing and the principles of *Reactive Systems* in-between nodes supporting a distributed system of stream processing stages, durable event logs, replication protocols—although these parts are typically not exposed to the developer. It is a good example of using *Reactive Programming* at the user level and *Reactive Systems* at the system level.

How Does Reactive Programming & Systems Relate To Microservices?

Microservices-based Architecture—designing a system of autonomous[7] distributed services, often with the Cloud as the designated deployment platform—benefits a lot from the value delivered by embracing Reactive.

As we have seen, both *Reactive Programming* and *Reactive Systems* design are important—in different contexts and for different reasons:

- Reactive Programming is used within a *single* Microservice to implement the service-internal logic and dataflow management.
- Reactive Systems design is used *in between* the Microservices, allowing the creation of systems of Microservices that play by the rules of distributed systems—Responsiveness through Resilience and Elasticity made possible by being Message-Driven.

[6] For example using Spark Streaming, Flink, Kafka Streams, Beam or Gearpump.
[7] The word autonomous comes from the greek words auto which means self and nomos which means law. I.e. an agent that lives by its own laws: self-governance and independence.

How Does Reactive Programming & Systems relate to Mobile Applications And The Internet Of Things (IoT)?

The Internet of Things (IoT)—with its explosion of connected sensors, gadgets and appliances—creates challenges in how to deal with all of these simultaneously connected devices that produce lots of data that has to be retrieved, aggregated, analyzed and pushed back out the the devices, all while maintaining overall system responsiveness. These challenges include managing significant bursts of traffic in receiving sensor data, processing of large amounts of data—in batch processes or real-time—and doing expensive simulations of real-world usage patterns. Some IoT deployments furthermore require the back-end services to manage the devices, not just consume the data sent from the devices.

When building services to be used by potentially millions of connected devices, there's a need for a model which copes with information flow at scale. There's a need for strategies for handling device failures, for when information is lost, and when services fail—because they will. The back-end systems managing all this needs to be able to scale on demand and be fully resilient, in other words, there's a need for *Reactive Systems*.

Having lots of sensors generating data, and being unable to deal with the rate with which this data arrives—a common problem set seen for the back-end of IoT—indicates a need to implement back-pressure for devices and sensors. Looking at the end-to-end data flow of an IoT system—with tons of devices: the need to store data, cleanse it, process it, run analytics, without any service interruption—the necessity of asynchronous, non-blocking, fully back-pressured streams becomes critical, this is where *Reactive Programming* really shines.

How Does Reactive Programming & Systems Relate To Traditional Web Applications?

Web Applications can greatly benefit from a *Reactive Programming* style of development, making it possible to compose request-response workflows involving branching out to service-calls, fetching resources asynchronously, and composing responses and subsequent marshalling to the client. Most recently, server push á la Server-Sent Events and WebSockets have become increasingly used, and performing this at scale requires an efficient way of keeping many open connections, and where IO doesn't block—*Reactive Programming* has tools for this, more specifically Streams and Futures, which makes it straightforward to do non-blocking and asynchronous transformations and push those to the

clients. *Reactive Programming* can also be valuable in the data access layer—updating and querying data in resource efficient manner—preferably using SQL or NoSQL databases with asynchronous drivers.

Web applications also benefit from *Reactive System* design for things like: distributed caching, data consistency, and cross-node notifications. Traditional web applications normally use stateless nodes. But as soon as you start using Server-Sent-Events (SSE) and WebSockets, your nodes become stateful, since at a minimum, they are holding the state of a client connection, and push notifications need to be routed to them accordingly. Doing this effectively requires a *Reactive System* design, since it is an area where directly addressing the recipients through messaging is important.

Summary

Enterprises and middleware vendors alike are beginning to embrace Reactive, with 2016 witnessing a huge growth in corporate interest in adopting Reactive. In this white paper, we have described Reactive Systems as being the end goal—assuming the context of multicore, Cloud and Mobile architectures—for enterprises, with Reactive Programming serving as one of the important tools.

Reactive Programming offers productivity for Developers—through performance and resource efficiency—at the component level for internal logic and dataflow transformation while Reactive Systems offers productivity for Architects and DevOps—through resilience and elasticity—at the system level, for building “Cloud Native” and other large-scale distributed systems. We recommend combining the techniques of Reactive Programming within the design principles of Reactive Systems.

Build modern systems for the modern world.

lightbend.com

Lightbend (Twitter: **@Lightbend**) provides the leading Reactive application development platform for building distributed applications and modernizing aging infrastructures. Using microservices and fast data on a message-driven runtime, enterprise applications scale effortlessly on multi-core and cloud computing architectures. Many of the most admired brands around the globe are transforming their businesses with our platform, engaging billions of users every day through software that is changing the world.

Lightbend, Inc. 625 Market Street, 10th Floor, San Francisco, CA 94105 | **www.lightbend.com**