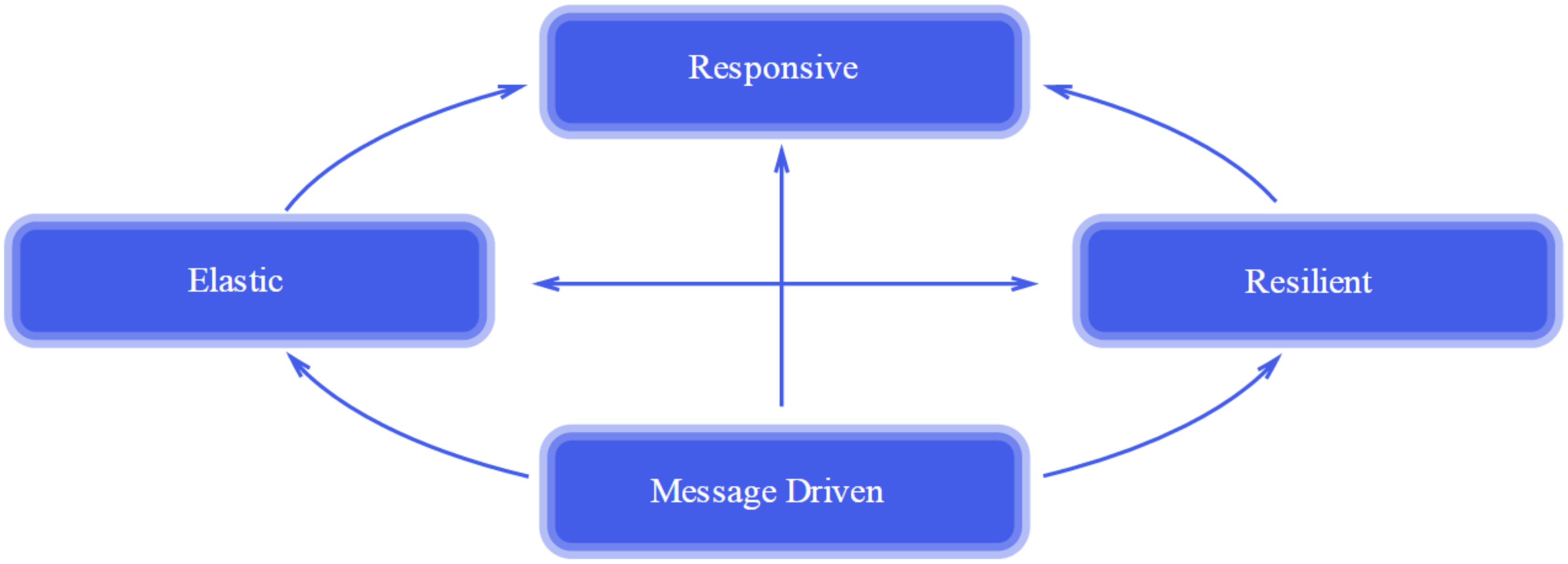
The Reactive Manifesto

Published on September 16 2014 (v2.0)

Reactive Systems are:

- **Responsive**
- **Resilient**
- **Elastic**
- **Message Driven**

Reactive Systems



Responsive

The system responds in a timely manner if at all possible

Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service.

Resilient

The system stays responsive in the face of failure.

Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole.

Elastic

The system stays responsive under varying workload.

Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs.

Message Driven

Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation and location transparency

Message Driven cont.

Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary.

Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host.

Reactive principles are not new

They have been proven and hardened for more than 40 years, going back to the seminal work by Carl Hewitt and his invention of the Actor Model, Jim Gray and Pat Helland at Tandem Systems, and Joe Armstrong and Robert Virding and their work on Erlang.

Reactive Programming

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 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.**

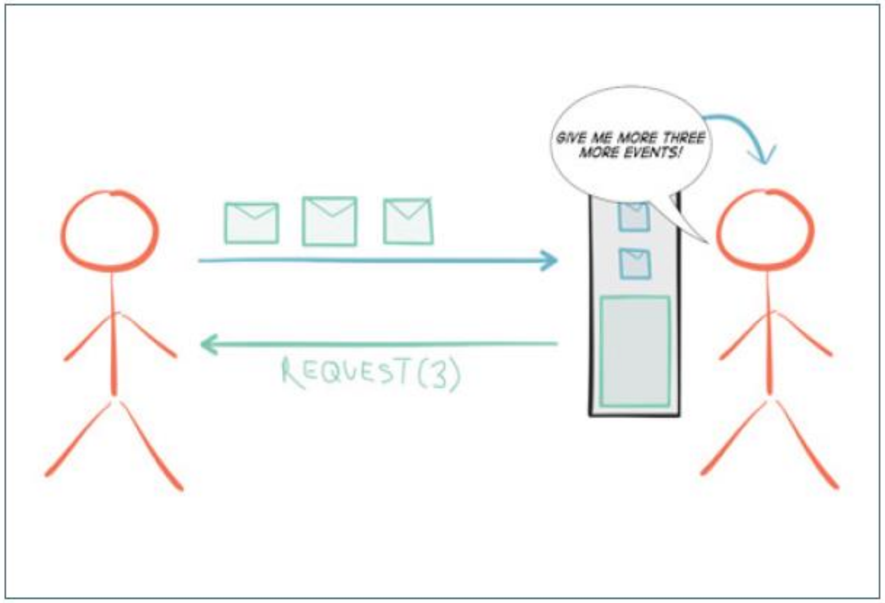
Programming technique

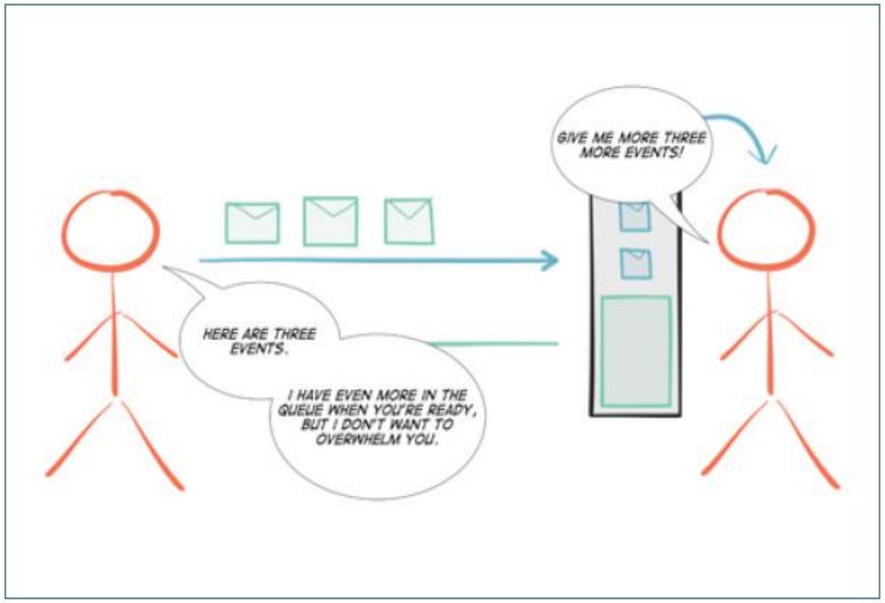
Futures/Promises—containers of a single value, many-read/single-write semantics where asynchronous transformations of the value can be added even if it is not yet available.

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.

Back-pressure




GIVE ME MORE THREE MORE EVENTS!
HERE ARE THREE EVENTS.
I HAVE EVEN MORE IN THE QUEUE WHEN YOU'RE READY, BUT I DON'T WANT TO OVERWHELM YOU.

Event-Driven vs Message-Driven

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.

Event-Driven vs Message-Driven

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.

Event-Driven vs Message-Driven

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.

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 Resilience of Reactive Systems

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.

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.

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.

Another contrast to the Reactive Systems approach is that pure Reactive Programming allows decoupling in *time, but not space*

How Does Reactive Programming & Systems Relate To Microservices?

- **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.

End