System Architectures
Reactive Architecture Fundamentals

Jonathan Thaler

Department of Computer Science

FH Vorarlberg
University of Applied Sciences

Motivation

- In 2011, due to a security breach Sony decided to take their Playstation Network down for **23** days. Sony offered a number of **compensations** to the players.
- In 2015 HSBC (British Bank) had an **outtage** of their electronical payment system, which had the effect that people didn't get paid before a **holiday weekend**.
- In 2015, Bloomberg, a company active in High Frequency Trading, experienced software and hardware **failures**, which prevented critical trading for **2 hours**.

Unresponsive services can have serious consequences!

10-15 years ago...

- ... large systems comprised of a few nodes with the max in up to **tens** of nodes. **Nowadays** large systems go up into the 100s and **1000s** of nodes.
- ... a system could be down for maintenance for **quite a while** and it was **no big deal**. Such behaviour is not acceptable anymore **today**, and users (and other systems) expect an uptime of up to **100%**.
- ... **data was at rest**, which means it was stored and then consumed **later** in a batch process. Nowadays data is **constantly** processed and changing as it is being **produced**.

Towards Responsiveness

- This **dramatic shift** occured due to an interplay between shifting user experience and ever increasing bandwidth and computing power, which was able to satisfy these **expectations**.
- People increasingly became **dependent** on critical online services for their work, for example various Google applications, GitHub,... It is simply **unacceptable** that such a service is down for even a few hours.
- Nowadays users **expect** an **immediate** response from services - it is simply not acceptable to wait for a response for a few **seconds**, because users expect a reaction from the service **immediately**, better within the very second.

There is a tremendous expectation for **responsiveness**.

Responsive Architecture

- Increased bandwith and computing power alone is **not enough** to deliver services which are available 24/7 with immediate **responsiveness**.
- What is required is a **proper software architecture**, which exploits these technological advances to deliver the expected user experience.
- In the last years so called **Reactive Architecture** has turned out to be a very viable and powerful architecture to deliver these very requirements.

The primary goal of reactive architecture is to provide an **experience** that is **responsive** under **all** conditions.

Reactive Software System

Criterias for a reactive software system:

- **Scales** from 10 to 10,000,000 users, which happens in start-up scenarios.
- **Consume** only the resources necessary to support the **current load**. Although the system could handle 10 million users, it should not consume the resources required to handle 10 million users when curently only 1000 users are accessing the system.
- Handles **failures** with little to no effect on the user. Ideally, there is no effect, however this is not always possible but the effect should be as **small** as possible.
- Scalability and failure handling / tolerance is achieved by **distributing** the software across multiple machines. So the software must be able to be **distributed** across 10s, 100s or even 1000s of machines.
- When scaling across a large number of machines, **maintain** a consistent level of **quality** and **responsiveness** despite the complexity of the software. Therefore, even if the software is distributed across 10s or 100s of machines, the responsiveness must not increase 10 or 100 fold and should stay **roughly** the same.

Reactive Principles

Reactive Principles

Reactive Principles

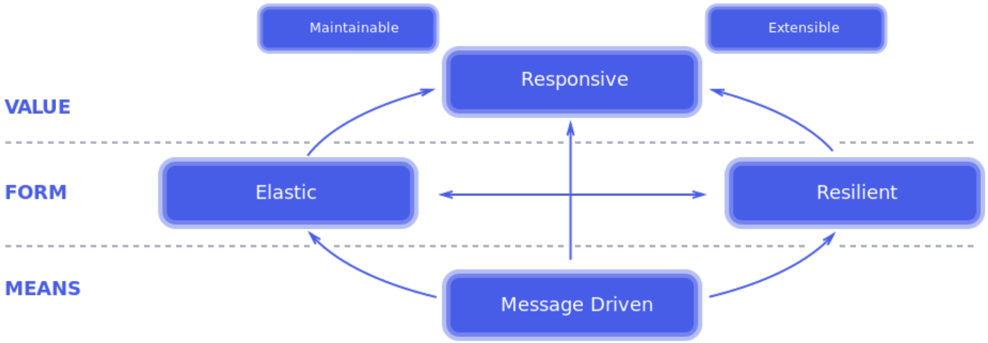
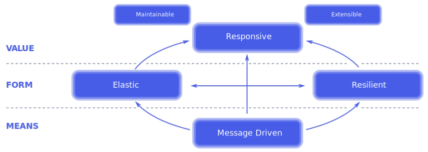


Figure: Reactive Principles

Reactive Principles

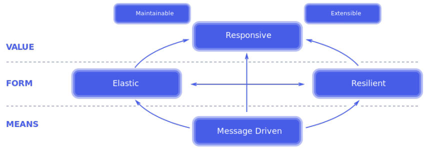
Responsive - A reactive system consistently responds in a **timely** fashion. It is the most **important** principle and all the other principles are there to ultimately **manifest** this principle.



- Responsiveness is the **cornerstone** of usability.
- It is basically not possible to provide a responsive user experience without resilience, elasticity and a system that is message driven.
- The goal is to make it fast and responsive **whenever** possible and as **often** as possible.
- **Unresponsive** systems cause users to walk away and look for alternatives, resulting in **loss** of **business** opportunity.

Reactive Principles

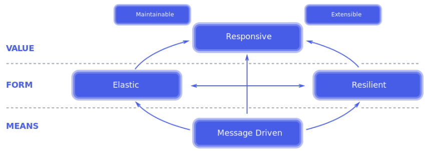
Resilient - A reactive system remains responsive, even if **failures** occur.



- **Replication**: there are multiple copies of services running.
- **Isolation**: services can function on their own.
- **Containment**: failure does not propagate to other services.
- **Delegation**: recovery is handled by an external component.
- The key is that any failures are **isolated** into a single component, They don't **propagate** and bring down the whole system.

Reactive Principles

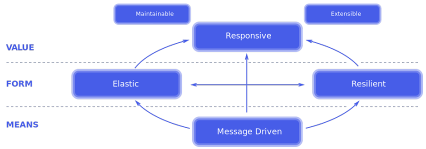
Elastic - A reactive system **remains** responsive, despite changes to system load.



- In older versions of the manifesto it was called **Scalability** but was subsequently renamed to also emphasise the need for a system to scale **down** after a spike in system load.
- It implies **zero contention** and **no central bottlenecks**.
- It is **not** possible to absolutely achieve this but the goal is to get **as close as possible**.
- Scaling **up** provides responsiveness during **peak**, while scaling **down** improves cost **effectiveness**.

Reactive Principles

Message Driven - A reactive system is built on a foundation of asynchronous, non-blocking **messages**.



- In older versions of the manifesto it was called **Event-Driven** but was subsequently renamed to avoid confusion with certain connotations of the term.
- **Enables** all the **other** principles and provides loose coupling, isolation and location transparency.

Reactive Principles

Reactive Systems vs. Reactive Programming

Reactive Systems vs. Reactive Programming

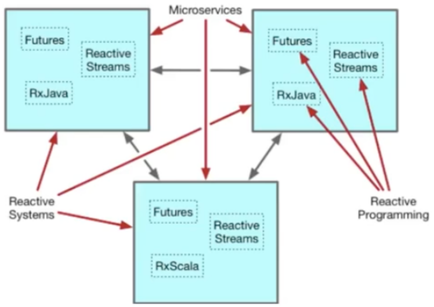


Figure: Reactive Systems vs. Reactive Programming

Reactive Systems vs. Reactive Programming

Reactive Systems

Apply the **reactive principles** on an **architectural level**.

Reactive systems are built using the principles from the reactive manifesto. In such systems all major architectural components interact in a **reactive** way, which are **separated** along **asynchronous** boundaries.

Reactive Systems vs. Reactive Programming

Reactive Programming

Can be used to **support** building reactive systems, but are not a **necessity** for building reactive systems.

Just because reactive programming is used, it does not mean you have a reactive system. It **supports** to break up the system into small discrete steps which are then executed in an asynchrounous non-blocking fashion such as *Futures/Promises*, *Streams*, *RxJava*.

Reactive Systems vs. Reactive Programming

Actor Model

The actor model provides facilities to **support** all **reactive principles**. It is **message driven** by default. The **location transparency** is there to support **elasticity** and **resilience** through distribution. The elasticity and resilience then provide **responsiveness** under a wide variety of circumstances.

Note that it is still **possible** to write a system with the Actor Model and **not** be reactive. But with the Actor Model, and the tools that are based on it, it is **easier** to write a reactive system.

Reactive Architecture

Building Scalable Systems

Building Scalable Systems

Scalable Systems

Building a scalable system is all about making choices between **scalability**, **consistency**, and **availability**. The **CAP** theorem (see later slides) shows that we can only have **two** of them at the same time.

The business side wants **both** but due to the **CAP** theorem this is not really an option.

Therefore, a choice is made between consistency and availability. Ultimately, making the **right** tradeoff between them is a **business** and not a technical issue.

Building Scalable Systems

1. **Scalability** *A system is scalable if it can* **meet** *increases in* **demand** *while remaining* **responsive**. A restaurant could be considered scalable if it can meet an increase in customers and still continue to respond to those customers needs in a fast and efficient way.
2. **Consistency** *A system is consistent if* **all** *members of the system have the* **same** *view or state.* In a restaurant if we ask multiple employees about the status of an order and we get the same answers then it is consistent.
3. **Availability** *A system is considered available if it* **remains** *responsive* **despite** *any failures.* In a restaurant if a cook accidentally burns his hand, and has to go to the hospital, that is a failure. If the restaurant can continue to serve the customers then the system is considered available.

Building Scalable Systems

Scalability

Building Scalable Systems: Scalability

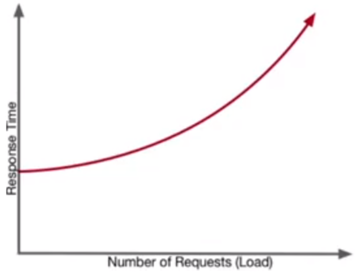


Figure: Performance vs. Scalability

- Performance optimises **response** time. Scalability optimises ability to handle **load**.
- **System**: takes 1 second to process one request and can handle one request at a time.
- **Optimisation 1**: improving the **performance** to take 0.5 seconds to process 1 request.
- **Optimisation 2**: improving the **scalability** to process 2 requests in parallel.
- Looking at *requests-per-second* does **not** say which improves as it combines **both** performance and scalability.

Building Scalable Systems: Scalability

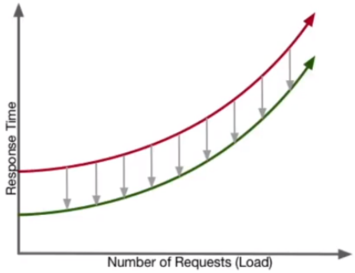


Figure: Performance

- When considering performance **isolated**: if performance is improved we improve our **response** time, but the number of requests (**load**) may have not changed.
- **Performace is theoretically limited**: it can never be smaller or equal 0 due to the laws of physics.

Building Scalable Systems: Scalability

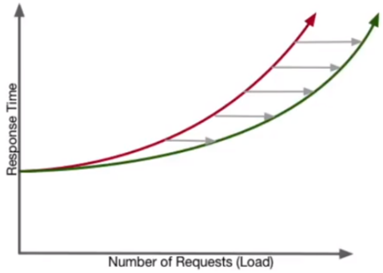


Figure: Performance

- Scalability improves the ability to handle **load**. That pushes the graph along the x-axis, but the **performance** of each request may **not** change.
- **Scalability is not theoretically limited**: it could be pushed along the x-axis forever.
- Therefore, when building Reactive Microservices the **focus** tends to be on improving **scalability** because this can in theory be **pushed forever**.

Building Scalable Systems: Consistency

Consistency

Building Scalable Systems: Consistency

- Distributed systems are systems that are **separated by space**.
- Due to the laws of phyiscs, which say that the speed by which information travels is finite (**speed of light**), because of the separation by space there is always **time required** to reach a **consensus**.
- If you want two pieces of your system to **agree** on the state of the world then they have to **communicate** with each other in order to come to some sort of **consensus**.
- In the time that it takes to **transfer** the information, the state of the original sender may have **changed**.
- The problem is that the receiver of information is always dealing with **stale data**.

In a **distributed** system we are always dealing with **stale data**. Reality is basically **eventually consistent**.

Eventual Consistency

Eventual Consistency

Eventual consistency guarantees that in the **absence** of new updates all accesses to a specific piece of data will **eventually** return the **most recent** value.

This implies that in order to reach a state of consistency you have to **stop all updates**, at least for some **period** of time, in order to reach that level of consistency.

- **Causal Consistency**: causally related items will be processed in a common order. For example if A causes B then A will always be processed before B.
- **Sequential Consistency**: processes all items in a sequential order regardless of whether they're causally related. This is a stronger form than causal consistency

Strong Consistency

Strong Consistency

Strong Consistency means that an update to a piece of data needs **agreement** from **all nodes** before it becomes **visible**. All accesses are **seen** by all parallel processes (or nodes, processors, etc.) in the **same order** (sequentially).

Traditional **monolithic** architectures are usually based around **strong consistency**.

Often, **distributed** systems seem to exhibit characteristics of **strong consistency**. This is achieved by introducing mechanisms which **simulate** strong consistency, for example a **lock**.

Strong Consistency

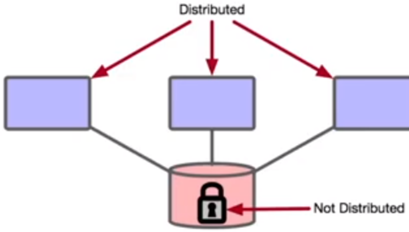


Figure: Locks reduce distributed problems to non-distributed ones.

- Any two things that content for a single limited resource are in **competition** which can have only **one** winner.
- Others are forced to **wait** for the winner to complete.
- As the number of things competing **increases**, the **time** until resources can be freed up **increases**.
- As **load** increases, we will eventually exceed acceptable time limits. This leads to **timeouts** and users leaving the system.

Strong Consistency

Contention reduces the ability to parallelise (Amdahls law).

The more parallel processors we add, the more **contention**, which eventually results in **diminishing** returns, making the performance actually **worse** on increased parallel processors than with less.

Strong Consistency

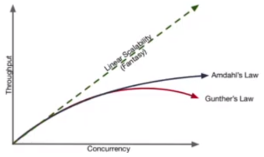


Figure: Laws of scalability.

Coherence Delay

- In a distributed system, **synchronising** the state of multiple nodes is done using **crosstalk**.
- Nodes in the system will send messages to **each other** informing of any state **changes**.
- The time it takes for this synchronisation to complete is called the **Coherence Delay**.
- Increasing the **number** of nodes, **increases** the coherence delay.
- If coherence delay is factored in, then increasing **contention** through increased **parallelism** can actually result in **negative** returns (Gunthers law).

Consistency

- Reactive systems understand the **limitations** that are imposed by these laws. They don't try to **avoid** these limitations but **accept** them and **minimize** their impact.
- **Linear scalability requires total isolation**: the system needs to be basically stateless.
- **Reducing contention**: isolating locks, eliminating transactions, avoiding blocking operations.
- **Mitigating coherence delays**: embracing eventual consistency, building on autonomy.

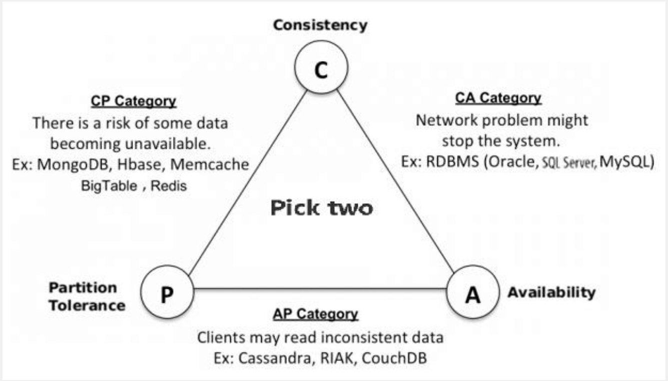
Building Scalable Systems

CAP Theorem

CAP Theorem

CAP Theorem

The CAP Theorem states that in a **distributed** system we **cannot** provide more than two of the following: consistency, availability, and partition tolerance.


Consistency
C
CP Category
There is a risk of some data becoming unavailable.
Ex: MongoDB, Hbase, Memcache BigTable , Redis
CA Category
Network problem might stop the system.
Ex: RDBMS (Oracle, SQL Server, MySQL)
Pick two
Partition Tolerance
P
A
Availability
AP Category
Clients may read inconsistent data
Ex: Cassandra, RIAK, CouchDB

CAP Theorem

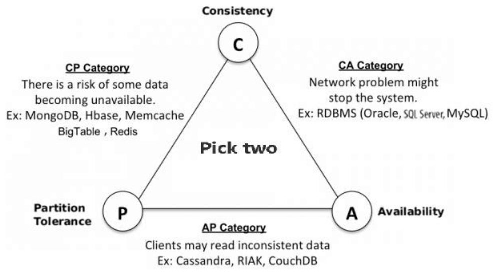
Partition Tolerance

Partition tolerance means that the system **continues to operate** despite an arbitrary number of messages being **dropped** or **delayed** by the network.

In reality no distributed system is safe from partitions. Networks can go down, nodes can go down; outages can be short or long lived.

As a consequence, **sacrificing partition tolerance is not an option**

CAP Theorem


Consistency
C
CP Category
There is a risk of some data becoming unavailable.
Ex: MongoDB, Hbase, Memcache BigTable , Redis
CA Category
Network problem might stop the system.
Ex: RDBMS (Oracle, SQL Server, MySQL)
Pick two
Partition Tolerance
P
A
Availability
AP Category
Clients may read inconsistent data
Ex: Cassandra, RIAK, CouchDB

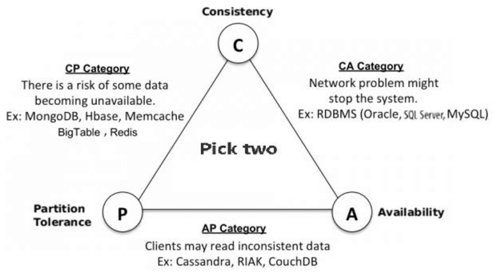
We are left with two options:

1. **AP** - **Sacrifice consistency**, allowing writes to both sides of the partition.

When the partition is resolved you will need a way to **merge** the data in order to **restore consistency**.

The system will always process the query and try to return the **most recent** available version of the information, even if it cannot guarantee it is up to date due to network partitioning.

CAP Theorem


Consistency
C
CP Category
There is a risk of some data becoming unavailable.
Ex: MongoDB, Hbase, Memcache BigTable , Redis
CA Category
Network problem might stop the system.
Ex: RDBMS (Oracle, SQL Server, MySQL)
Pick two
Partition Tolerance
P
A
Availability
AP Category
Clients may read inconsistent data
Ex: Cassandra, RIAK, CouchDB

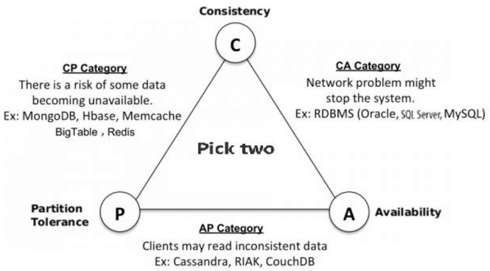
We are left with two options:

2. **CP - Sacrifice availability**, disabling or terminating one side of the partition.

During the partition, some or all of your system will be **unavailable**.

The system will return an **error** or a **timeout** if particular information cannot be guaranteed to be up to date due to network partitioning.

CAP Theorem


Consistency
C
CP Category
There is a risk of some data becoming unavailable.
Ex: MongoDB, Hbase, Memcache BigTable , Redis
CA Category
Network problem might stop the system.
Ex: RDBMS (Oracle, SQL Server, MySQL)
Pick two
Partition Tolerance
P
A
Availability
AP Category
Clients may read inconsistent data
Ex: Cassandra, RIAK, CouchDB

- **CA** systems **ignores** the network, partitioning and ultimately ignore the fact that they are a **distributed system**.
- When **another** system **communicates** with a CA system, there might be the chance of a **failure** in the CA system. Therefore, **availability** is **not** given anymore and we are back to a **CP** system, rather than a CA system.
- Using replicas would not solve this problem as then you would run into the consistency problem, resulting in an **AP** system.

CAP Theorem

In the **absence** of network failure (when the distributed system is running normally) **both** availability and consistency can be satisfied.

The choice between consistency and availability has to be made only when a network partition or **failure** happens.

CAP Theorem

Systems are often consistent and partition tolerant, but have areas where they're **not** fully consistent, such as in specific **edge cases**.

For example, when they **fail** over to a replica then they sacrifice consistency, so they're **not** necessarily **always** consistent and just aim to be consistent in **most cases**.

CAP Theorem

On the other hand a system might be available and partition tolerant but still have areas where its **not** actually available.

For example, it might be **generally** available but if a certain number of nodes fail then maybe the system become **unavailable** again.

CAP Theorem

Consistency or Availability?

CAP Theorem

Consistency or Availability?

The choice between consistency and availability isn't really a technical decision, it's actually a business decision.

- In reality, most systems **balance** the two concerns usually **favoring** one side or the other.
- The decision of when and where to sacrifice consistency or availability should be discussed with the **domain experts** and **product owners**.
- Software Developers (Architects) should **talk** to these people and make the tradeoffs clear: what are the **costs** if high availability or strong consistency have to be guaranteed in specific situations.

CAP Theorem

Ultimately, we need to factor in the **impact on revenue** if the system is unavailable vs. eventually consistent.