Web-scale systems

according to Netflix

Kjell Jørgen Hole
Simula@UiB

Bergen, Norway 10 March 2015

Web-scale (software) solutions grow without introducing bottlenecks that require periodic re-designs. This talk argues that web-scale solutions should be implemented in public clouds and outlines how to create solutions with very high availability, scalability, and performance. Netflix's cloud-based, video streaming system illustrates important insights.

Overview

- Introduction to web-scale solutions
- Web-scale solutions in the cloud
- Solutions with very high uptime

Web-scale solutions

This section introduces non-functional requirements for web-scale solutions and explains why "traditional" monolithic solutions have problems satisfying these requirements.

Non-functional requirements

- A customer-facing web-scale solution must have
  - high **availability** (99.99%), i.e. very little downtime
  - large **scalability** (tens of millions of customers)
  - high **performance** (low-latency responses)

Availability: 99.99% availability corresponds to about 53 minutes of downtime each year; often referred to as four nines of availability.
Scalability: refers to the number of users who have a positive experience.
Performance: the experience of an individual user. Often determined by measuring response time (latency).

Everything is broken

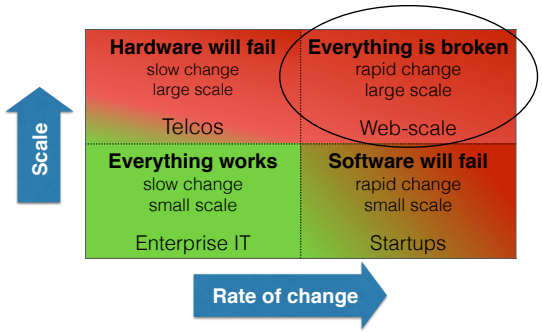


Fig. from Netflix

5

The rate of failure increases as a system scales and the number of changes increases. It is particularly challenging to create a high-availability web-scale system with a massive hardware platform, millions of users, and rapid innovation.

Monolithic solutions

- **Advantages**
  - good scalability using multiple servers and load balancers
  - low latency because modules can communicate efficiently
- **Disadvantages**
  - insufficient availability due to **cascading failures**
  - many dependencies make it hard to upgrade software quickly

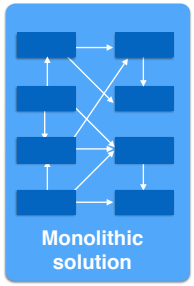


Fig. from Netflix

6

A **monolithic solution** is a large software application that is compiled into a single executable. Because a monolithic solution contains very many lines of source code, the solution takes a long time to compile, thus, reducing the productivity of the developers. Furthermore, a single executable takes a long time to start, reducing the availability of the solution. Finally, it is cumbersome to make changes to a solution because it is necessary to recompile the code and deploy a new executable. A **cascading failure** occurs when a local failure propagates over a solution due to tight coupling between its modules. The tight integration of the modules in a monolithic solution makes it fragile to cascading failures. An memory leakage is an example of a local failure that causes a systemic failure because the single executable will crash when there is no more memory available.

Problem: module dependencies

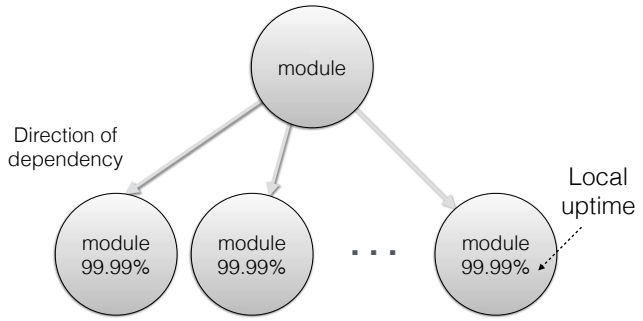


Fig. from Netflix

The figure illustrates module dependencies in a monolithic solution.

Overall availability

- Assume that a module failure leads to system failure
- 1000 modules with 99.99% availability each
- The overall system availability is

  0.9999^1000 ≈ 0.90

90% availability corresponds to about 36.5 days of downtime per year.

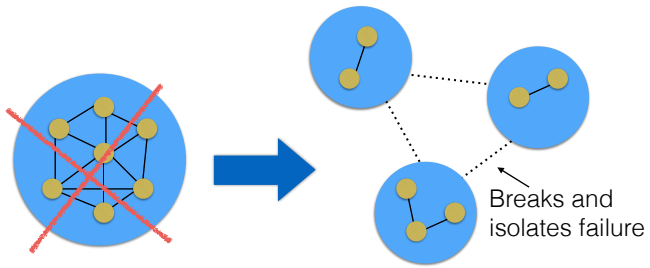
Break dependencies

Isolate local failures—failure in one module should never result in cascading failure taking down the whole system

9

If the dependencies are not removed, then the 1000 modules must each have 99.99999% availability to achieve an overall availability of 99.99%.

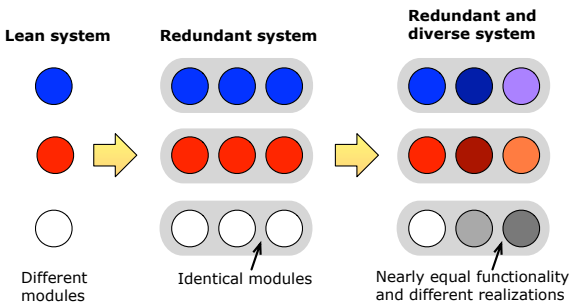
Need modules with weak links



10

While the concept of “modules” is both well understood and much used, the concept of “weak links” is much less understood and used. Later, we’ll discuss how the circuit breaker pattern realizes “weak links.”

Need redundancy and diversity



While "redundancy" and "diversity" are well-known concepts, they have only seen limited use in "traditional" computer centers because of the significant cost needed to develop and run multiple implementations of the same functionality.

Ideas to break dependencies

- Modules
- Weak links
- Redundancy
- Diversity

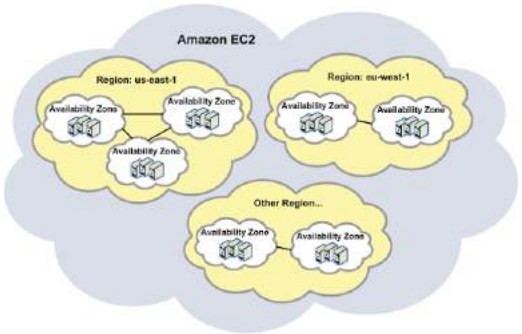
Note that these very general concepts (abstract ideas) can be applied to all kind of systems, not only web-scale systems.

Web-scale solutions in the cloud

13

This section explains why a system's modules should be implemented as micro-services in a public cloud.

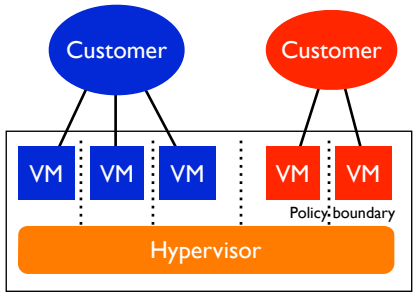
Cloud infrastructure



14

Amazon Web Services (AWS) consists of Regions in different parts of the world. Each Region contains multiple Zones, where a Zone corresponds to a huge datacenter. AWS is steadily expanding their global infrastructure to help customers achieve lower latency and higher throughput, and to ensure that customer data resides only in the Region they specify.

Virtualization



Single physical machine

A **hypervisor** is computer software that runs Virtual Machines (VMs). Each VM is guest machine. The hypervisor presents the guest Operating Systems (OSs) with a virtual operating platform and manages the execution of the guest OSs. Multiple instances of different OSs may share the virtualized hardware resources.

Why cloud?

- **Availability**: The cloud provides a cost-effective way to introduce the modules, weak links, redundancy, and diversity needed to break dependencies
- **Scalability**: Server virtualization supports the needed scalability
- **Performance**: The use of multiple cloud regions facilitate low-latency service all over the world

Why public cloud?

- Public cloud providers divide the infrastructure cost over very many customers making it far less expensive to use a public cloud than to build your own private cloud

Pay as you go is a billing method that is implemented in cloud computing and geared toward organizations, especially start-ups. An organization is only billed for the computing resources it provisions. Hence, the cost is small when the organization has few customers and only starts to grow as the organization attracts more users. This billing method lets start-ups compete with large established players.

Need cloud-native solution

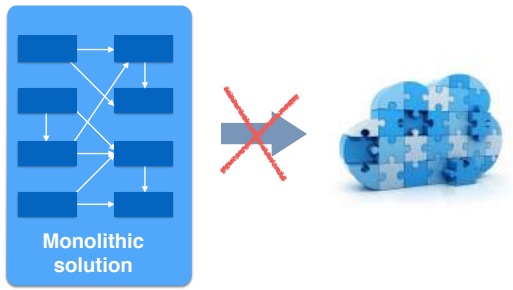


Fig. from Netflix

We cannot move a tightly-coupled solution into the cloud an expect the availability to increase. Instead, we need a **cloud-native solution** that leverages cloud-platform properties for scaling and performance; uses non-blocking communication in loosely coupled architecture; handles upgrades, scaling, and failure events without system downtime; and monitors solution as virtual machines come and go.

Modules:
micro-services in the cloud

- Netflix's streaming solution consists of hundreds of **micro-services** that run in Amazon's cloud
- The services communicate over network connections via a standardized, lightweight protocol

Micro-services are nothing new. They are actually a realization of the Unix philosophy of creating small, single-purpose programs that are "piped" together to achieve a desired result. Netflix has hundreds of micro-services running side-by-side in each cloud region. Some of the services are updated often, while others remain unchanged for long periods.

Properties of micro-services

- The functionality fulfills a single responsibility
- Easy to test, upgrade, and replace
- Fast startup and shutdown

A micro-service is a small standalone process that does one thing well. Together, micro-services decouple the functionality of a large application into highly independent chunks of code. Micro-services enhance fault tolerance, enable an application to scale, and make it possible for a solution to evolve.

Properties of …

- Services belonging to the same solution can be implemented in different programming languages
- Collections of micro-services can be updated independently of each other

Micro-service architecture

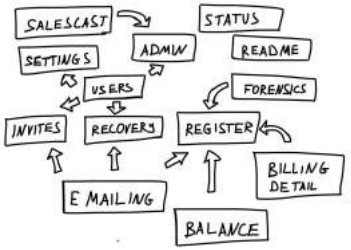


Fig. from abdullin.com

All communication between micro-services are via network calls, to enforce the separation between the services and avoid the problems of tight coupling. (Figure from http://abdullin.com/post/how-micro-services-approach-worked-out-in-production/)

Netflix architecture

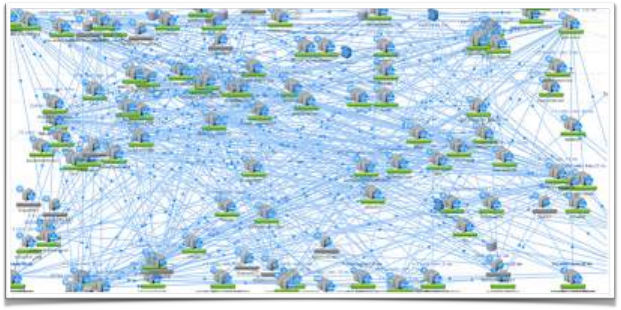


Fig. from Netflix

23

A micro-service solution mimics nature. The whole system is constantly evolving without the risk of downtime associated with monoliths. In particular, services come and go. Micro-service solutions are “living software” enabling us to get rid of the problem with legacy software because it is easy to remove old services and create new once.

Micro-service solutions in the cloud with very high uptime

24

In a rapidly changing world, it is not possible to plan everything in advance. Large ICT solutions need to have an evolutionary architecture that facilitates rapid change and supports high availability at the same time. Hence, we must abandon monoliths and split their functionality into more of less independent processes.

How to isolate failures

(according to Netflix)

1. Use micro-services and introduce weak links, redundancy, and diversity to **isolate** the impact of service failures
2. Induce failures to **learn** how to make a system increasingly robust to cascading failures

As we shall see, the company actually induces failures in their production system.

Weak links are circuit breakers

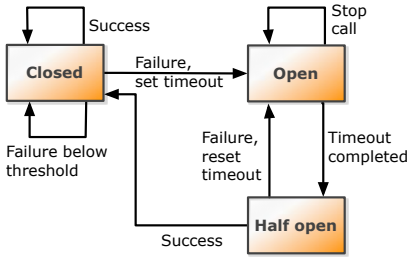
Weak links can be compared to circuit breakers.

Circuit breaker

- Netflix's **Hystrix** tool utilizes a "circuit-breaker" method to shut down requests to services when their latencies or number of failures become too large
- Fallbacks are provided wherever feasible to protect users from failure

https://github.com/Netflix/Hystrix/wiki
For more information, read about the CircuitBreaker pattern in Michael Nygard's book "Release It!"

Circuit breaker



The circuit is closed and service A calls service B. If there is no failure, then the circuit remains closed and A is allowed to connect to B. Should a failure occur, the fraction of failures is updated. When the fraction becomes larger than a threshold, the circuit is opened and a timer is started. The circuit remains open until a timeout period is completed. Then the circuit is closed on a trial basis and A is again allowed to connect to B. If there is still a failure, then the circuit is re-opened, else it remains closed.

Redundancy via replacement

Redundant services with timeout and failover

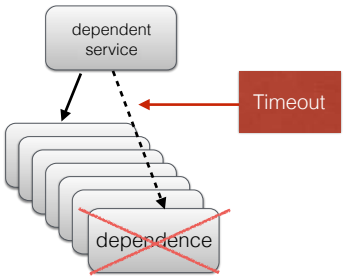


Fig. from Netflix

29

Micro-services are designed in to allows multiple instances to run behind a load balancer. If one instance goes down, then the calling instance can simply connect to another instance. An instance failure is often due to hardware error or network failure.

Default fallback response

Timeout with fallback default response used when *all* instances are affected

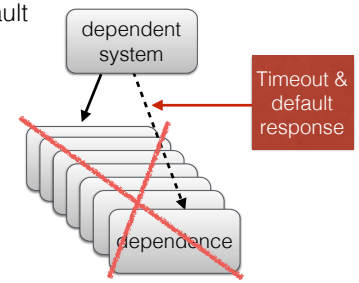


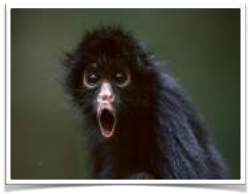
Fig. from Netflix

30

When there is a software error, all instances are affected and it is necessary to use a default response to contain the error. A careful analysis is needed to determine the appropriate response. The default response reduces the dependency between the instances.

Chaos Monkey

The tool **Chaos Monkey** disables random production instances to make sure the Netflix solution survives this common type of failure without any customer impact



31

The default instance groupings that Chaos Monkey uses for selection is Amazon's Auto Scaling Group (ASG). Within an ASG, Chaos Monkey will select an instance at random and terminate it. The ASG should detect the instance termination and automatically bring up a new instance.

Latency monkey

Latency Monkey tests what happens when the delay becomes too long

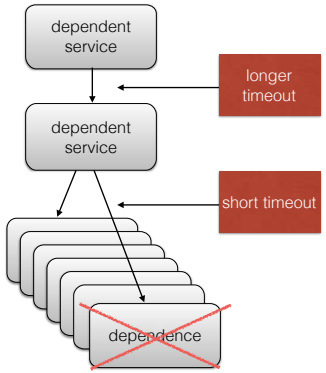


Fig. from Netflix

32

The shutdown of a low-level dependency can lead to a longer timeout at a higher layer, causing a cascading failure. There is no simple answer to this multi-level dependency problem, each case must be carefully studied.

Learning from failures (1)

- Netflix uses Chaos Monkey and Latency Monkey to test that the solution isolates local failures

Netflix induces failures in the production system to detect fragilities and learn how to improve the system's robustness to cascading failures. The company works to understanding what went wrong rather than who to blame for a failure.

Zone isolation

Chaos Gorilla generates zone failures

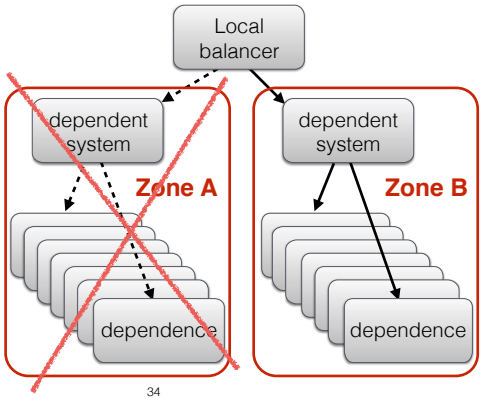
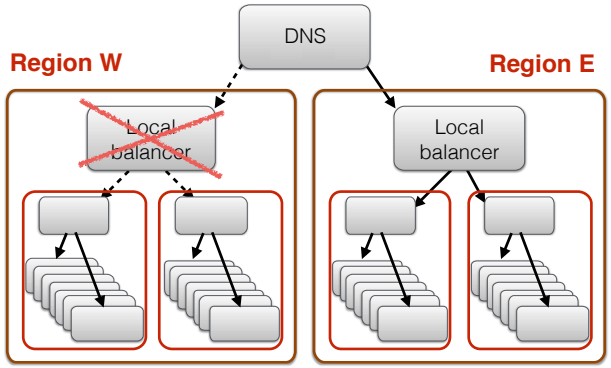


Fig. from Netflix

Netflix uses three zones in each region to limit the consequences of firmware failures, certain serious software bugs, power failures, and severe network failures. The zones correspond to different data centres. Note that the load balancer is a single point of failure.

Region isolation



Chaos Kong is used to test region failures

Fig. from Netflix

DNS (Domain Name Server) splits traffic load i two halves. To handle infrastructure failures, Netflix uses multiple regions and switch users to a new region when needed.

Learning from failures (2)

- Netflix uses Chaos Gorilla and Chaos Kong to verify that their solution can handle major problems with the cloud infrastructure

Diversity

- Two software programs are **diverse** if they have (nearly) the same functionality, but different implementations

Diversity via “canary push”

- Since a web-scale solution supports users all over the world, there is no good time to take down the system and upgrade its software
- An alternative is to introduce new code by keeping both old and new code running and switch user requests to new code

This process is possible in the cloud because an application owner can easily double the use of resources for a limited period, e.g. a 24 hour cycle.

Simple canary push

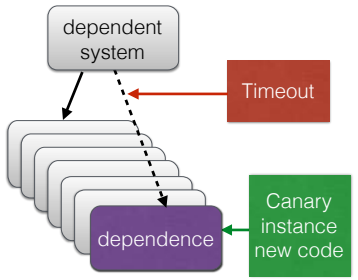


Fig. from Netflix

39

The stability of a “canary” cannot be fully evaluated before it gets a heavy traffic load in the production system.

Red/black deployment

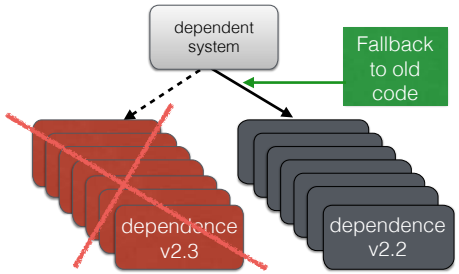


Fig. from Netflix

40

Use enough copies of new code in an auto-scaling group to carry the load. Keep the old code to ensure that we can handle peak load if there is a problem with the new code.

Standby blue system

- Software error in both red and black deployment
- Blue system is an indecently authored system delivering a minimal solution
- Used when all resent versions of the code fail

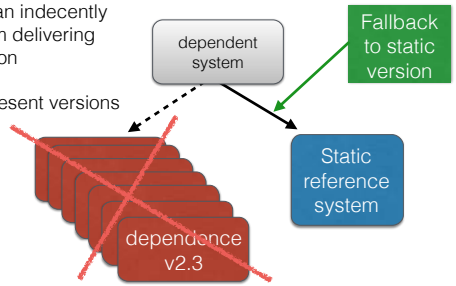


Fig. from Netflix

Several versions of the code may contain a "time bomb" that only goes off after a long period. Since the blue system is non-adaptive or static, it is easier to scale than the regular code.

Summary

- Cloud-based solutions with micro-service architectures can provide higher availability than today's monolithic applications

Observation: Norwegian banks believe they will save billions when Norway gets rid of cash. However, the banks will have to pay for new payment systems based on cloud technology to achieve the availability, scalability, and performance required by a "cash free society."

General principles

- Modularization
- Weak links
- Redundancy
- Diversity
- Learn from induced failures

Design principles

Operational principle

43

While we have only discussed the above principles in a cloud context, they are also valid outside the cloud.

References

- techblog.netflix.com
- martinfowler.com/articles/microservices.html
- highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html
- S. Newman, *Building Microservices*, O'Reilly Media, 2015
- M.T. Nygard, *Release It!* Pragmatic Bookshelf, 2007
- M.J. Kavis, *Architecting the Cloud*, Wiley, 2014
- N. Taleb, *Antifragile: Things That Gain from Disorder,* Random House, 2012

44

Thank you!