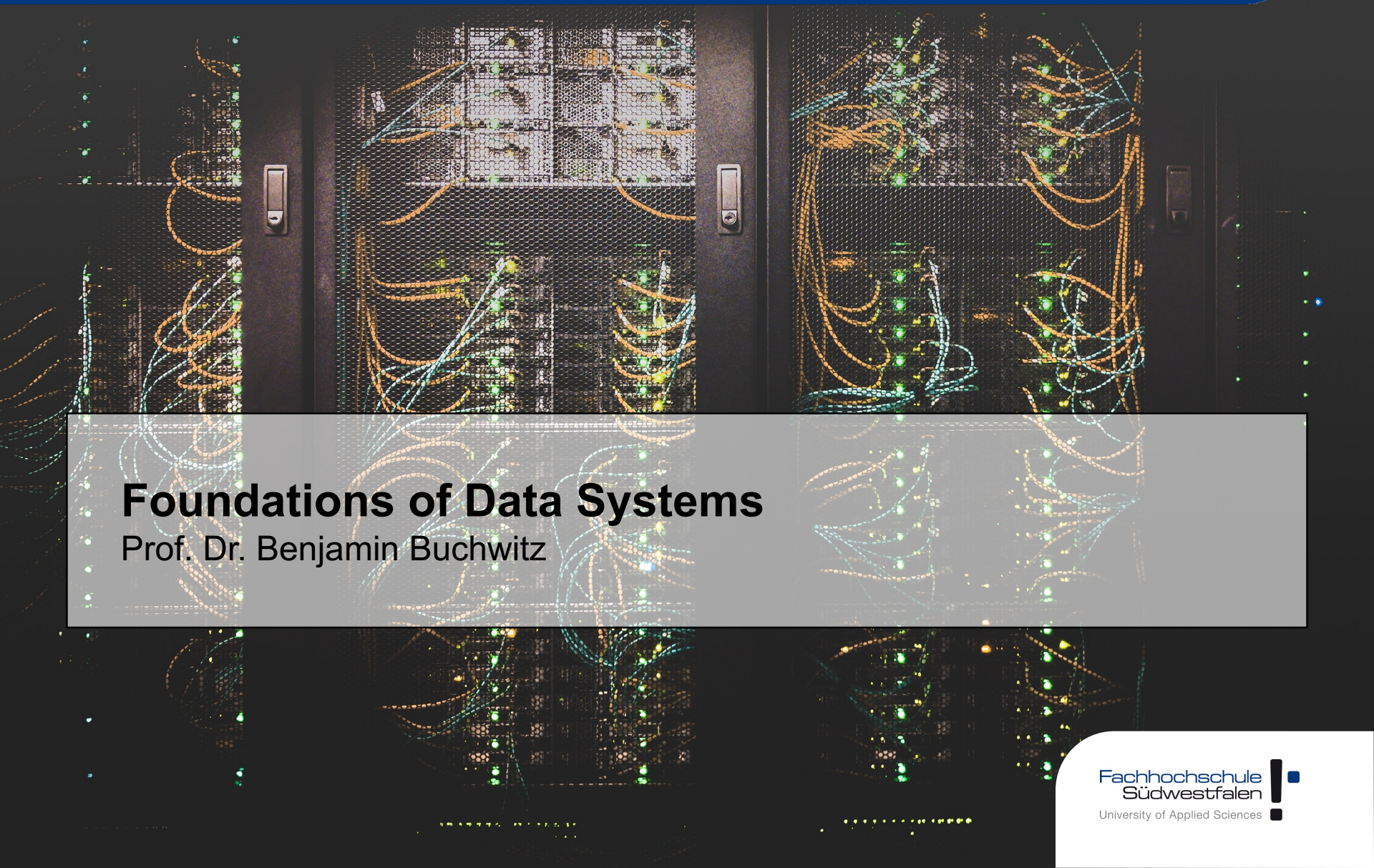
Big Data Data Analysis

Prof. Dr. Benjamin Buchwitz



Foundations of Data Systems

Prof. Dr. Benjamin Buchwitz

Fachhochschule Südwestfalen
University of Applied Sciences

Content and Learning Goals

Introduction

In this Introduction, we …

- learn what the driving principles behind Big Data Analytics are and establish those guidelines as cornerstones when designing data-intensive applications.
- understand that not only functional requirements are important, but that an essential part of the design process focusses on non-functional requirements.
- discuss the non-functional requirements reliability, scalability and maintainability in detail and link them to the context of data-intensive workloads.
- deep dive into application architectures that (partly) power popular large scale “cloud” applications in practice.
- see interesting examples why the principles discussed here matter and why it is worth spending conceptual time before actually starting to design a system that is intended for production purposes.

Data-Intensive Applications

Definition and Components

Data-Intensive Application

"We call an application *data-intensive* if data is its primary challenge – the quantity of data, the complexity of data, or the speed at which it is changing – as opposed to *compute-intensive*, where CPU cycles are the bottleneck."

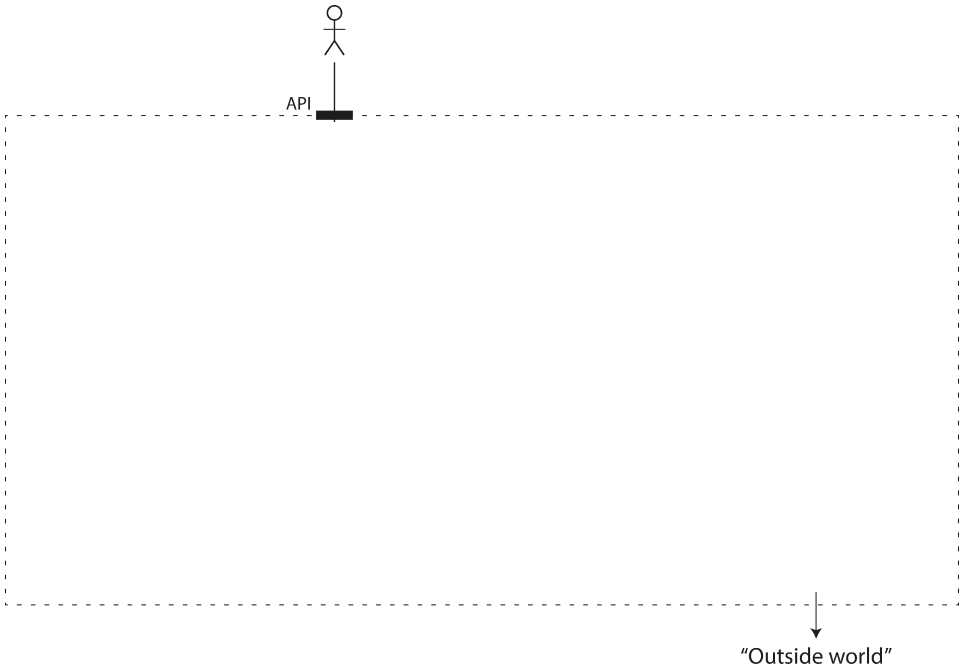
Common High-Level Components

- **Database**: Store structured or unstructured data (long time storage)
- **Caches**: Speed up reading data and storing intermediary results as well as intermediary (derived) data (short time storage)
- **Search Indexes**: Specialized data structure to quickly filter and find data sources in the system
- **Stream Processors**: Engine for asynchronous messaging and message (event) handling
- **Batch Processors**: Engine to process large amount of collected data (potentially periodically)

References: Kleppmann (2017)

Data-Intensive Applications

Example Application Architecture



- Application Programming Interface (API) hides technical details from the user, which constitutes the fact that the offered service can be seen as self-contained system.
- (Composite) Data Systems usually provide guarantees (e.g. speed, throughput), which imply that the author followed some principles and had some design goals.

References: Kleppmann (2017)

Design Goals for Data-Intensive Applications

Reliability, Scalability and Maintainability

Typical Questions when designing Data-Intensive Applications

- How to ensure that the dataset remains correct and complete (even in case of internal faults)?
- How du you scale when the user basis or load increases?
- What does a good Interface to the system look like and how to ensure that it stays valuable in the future?

Important Design goals in most software systems

Reliability Goal 1

The system works correctly (functionality and performance) in case of hardware, software or human errors.

Scalability Goal 2

Data, Traffic and complexity growth can be dealt with by the system without compromising functionality or performance.

Maintainability Goal 3

Maintaining current behaviour and adapting to future needs should be possible productively and with reasonable amount of resources.

Design Goal: Reliability

Goal 1

Reliability

The system works correctly (functionality and performance) in case of hardware, software or human errors.

Design Goal: Reliability

Goal 1

Hardware Faults

- E.g. HDD crashes, power outages or network problems occur.
- First countermeasure to avoid downtime: adding redundancy (RAID, Hot Swap CPUs, etc.) → not economically feasible.
- Additional shift in requirements: flexibility and elasticity over single machine reliability.
- Fault-tolerance in Software for multi machine applications (additional advantages, e.g. for rolling updates with no system downtime).

Software Faults

- While hardware faults often occur randomly (only weakly correlated, e.g. due to temperatures) software faults are systematic and usually highly correlated.
- Often occurring due to (undocumented) assumptions about environmental states.
- There is no typical countermeasure, but common practices include: carefully analysis about interactions and assumptions, thorough testing, process isolation, automatic monitoring.

Human Errors

- Humans are known to be unreliable, even though often having good intentions in their actions.
- Avoiding human errors must be done by means of system design:
  - Decoupling places where failure occurs from failures (e.g. non-production sandboxes for development)
  - Testing, Testing, Testing on all levels (unit-testing, system integration tests, manual & automatic testing).
  - Allow quick recovery from human errors to minimize impact, e.g. by fast configuration roll backs, gradual go lives (staging).

Design Goal: Reliability

Goal 1

Failure > Fault

- **Fault**: A component of the systems is deviating from the designed / required specifications.
- **Failure**: The system (as a whole) stops providing its services to the user.

Faults can not be eliminated: Our goal is to anticipate faults (*certain types* only; tolerating every fault not possible) and create a fault-tolerant and therefore resilient system to prevent failures.

Implication: Need to build reliable systems from unreliable parts.

- Generally speaking it is a better idea to **tolerate faults** rather than avoiding them. However, this is not always possible as there are irreversible faults that can not be cured (e.g. security / data breaches).
- Counterintuitively it may make sense to explicitly **create and therefore increase the number of faults** (e.g. by randomly killing instances or processes). This ensures that fault-tolerance is working when faults occur naturally (Idea: Continuous Testing).

References: Kleppmann (2017)

Netflix Chaos Monkey

Reliability Engineering in the Netflix Ecosystem

- **Netflix runs** most of its user facing infrastructure on Amazon Web Services by deploying **virtual machines and containers** that deliver content to viewers.
- Chaos Monkey randomly terminates virtual machine instances and containers that run inside of a **production environment**.
- Exposing engineers to failures more frequently incentivizes them to build resilient services.

The Monkey Theory

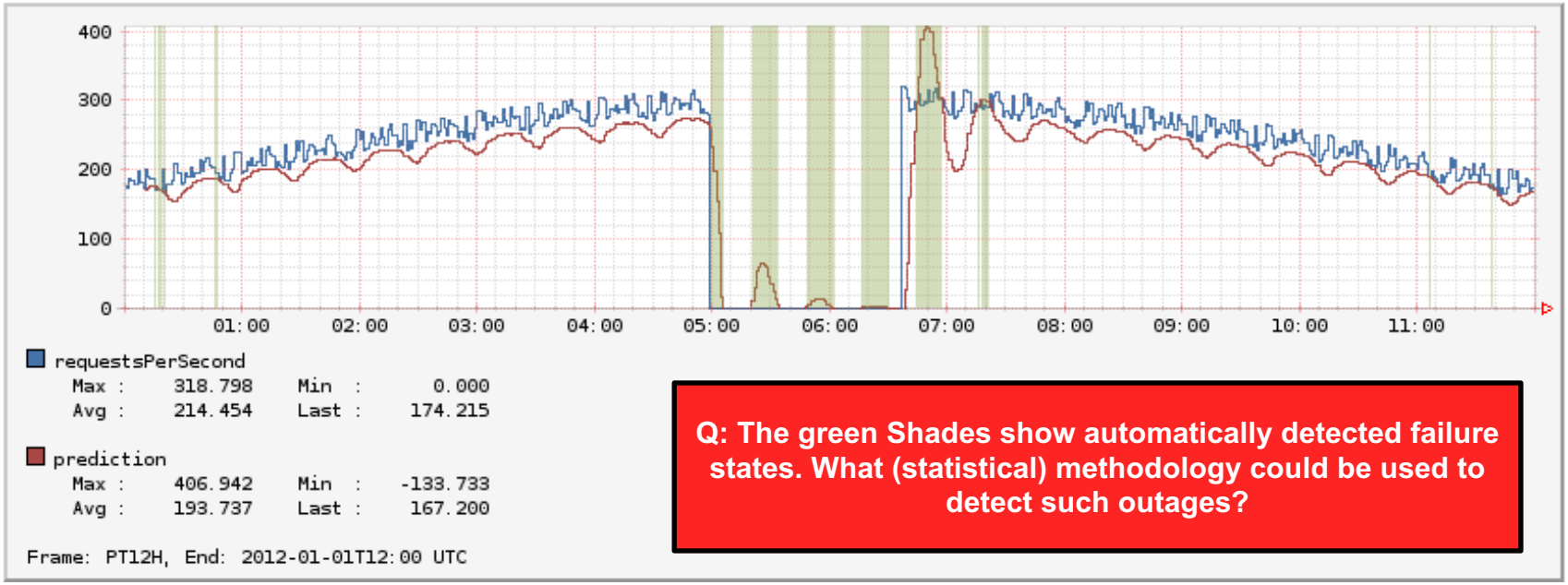
- Simulate things that go wrong.
- Find things that are different.

… or more formally …

References: Edberg (2012), Principles of Chaos Engineering (2018), Chang et al (2015), Netflix (2020)

Netflix Chaos Monkey

Example Chaos Monkey Experiment



The graph above stems from Atlas – Netflix's Time Series Monitoring System – and shows automatic detected failure states (green shades) as result from outages generated by Chaos Monkey (and other reliability testing tools from the so called "Simian Army").

References: Edberg (2012)

Design Goal: Scalability

Goal 2

Scalability

Data, Traffic and complexity growth can be dealt with by the system without compromising functionality or performance.

Design Goal: Scalability

Goal 2

What is scalability?

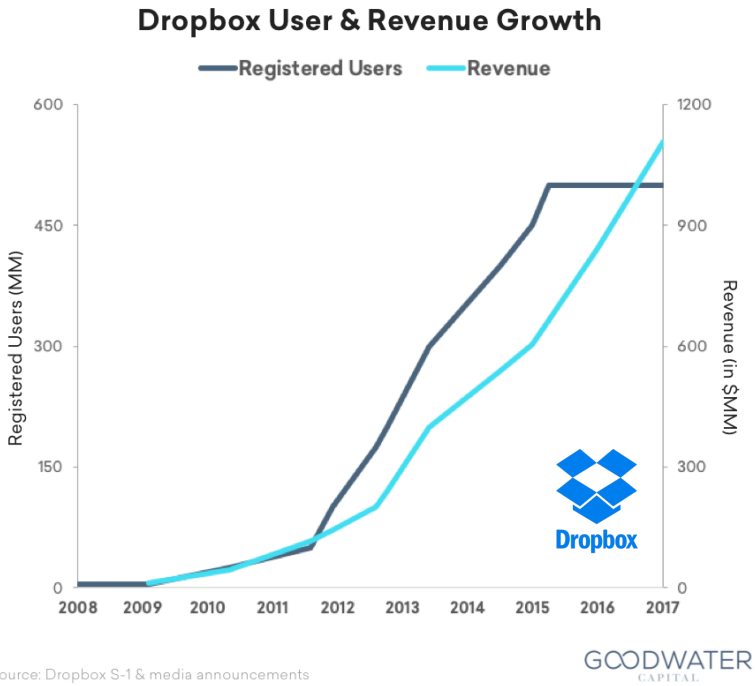
- Scalability is the ability to deal with fundamentally changing (practically mostly growing) system load.

How to achieve scalability?

- After understanding and describing the load on the system (Step 1), we can anticipate and asses implications when load volume or patterns change (step 2). This is the basis for (altering) the system design.

Example: Dropbox

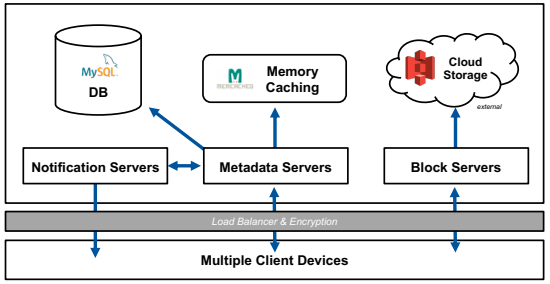
- Dropbox is a cloud file hosting and file synchronization service and was founded in 2007 by MIT Students.

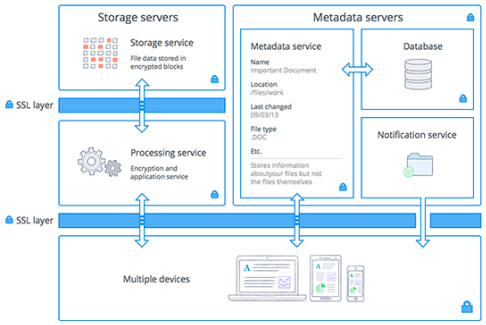


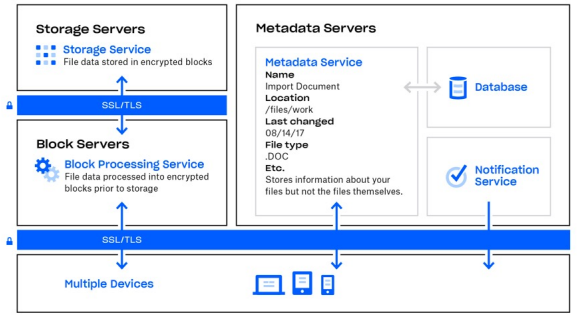
Example: **Do you think the Dropbox System Architecture changed between 2008 and 2015? If yes, how?**

References: Goodwater Capital (2018), Dropbox (2019)

Dropbox System Architecture Evolution







2008 / 2009 *2013* *2015*

- The Dropbox System Architecture stayed highly comparable over the years with very similar (high-level) building blocks from 2008 / 2009 to 2016.
- Dropbox leverages Cloud Services (AWS, etc.) as part of their Infrastructure and (as of today) comprises a real **Hybrid** (own Datacenter + Cloud) **Infrastructure.**
- However, Dropbox decided around 2013 / 2014 to move out of "Cloud Storage" and especially out of Amazon S3 and host all Block Content in their proprietary Exabyte-scale [illegible] (yes, Magic Pocket).

How was Dropbox able to successfully move out of the Amazon Simple Storage Service (S3)?

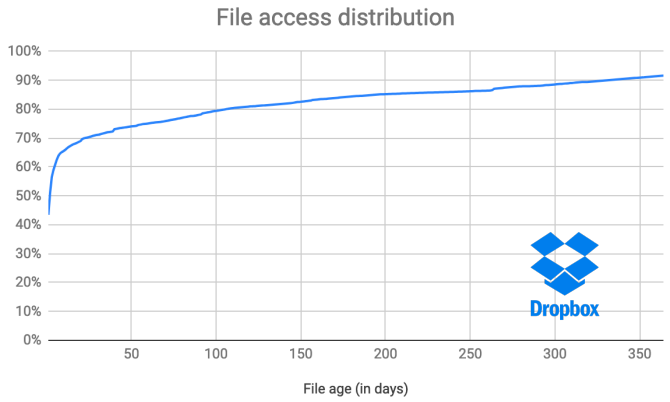
References: Barausse and Cilardo (2014), Dropbox (2015), Cowling (2016), Gupta (2016), Cowling (2018), Dropbox (2019), Le (2019)

Dropbox Magic Pocket Example

Step 1: Understanding and Describing Load

Requirements & Observations

- 500+ PB Data (1bn new files/day)
- Data Durability > 99.9999999999%
- Availability over 99.99% with at least 2 storage regions per file
- Low access latency for stored files
- Slightly more reads than writes
- Files quickly become cold (see cumulative distribution below)



References: Gupta (2016), Senapaty (2017), Kleppmann (2017)

Generalization: Step 1

Load is not a one-dimensional or simple construct, but a heterogenous concept that has multiple parameters and often time-wise variation on the individual entity level (requests, files, users, etc.).

Example load parameters:

- Requests per Second (web service)
- Ratio of reads to writes (database)
- Number of simultaneous users (chat room)

Dropbox Magic Pocket Example

Step 2: Asses implications of load volume or pattern changes

Dropbox **anticipated growth** from the beginning (back in 2007/2008) and has built a system that was flexible enough to be hosted on **hybrid infrastructure** environments.

Scope: Build Magic Pocket (MP) System for storing user data (not metadata) with predictable loads. Stay on Amazon S3 for EU customers as market growth and regulatory requirements are more unpredictable.

Performance is measured economically. Technical parameters should at least match Amazon S3, while outperforming it cost-wise.

Process: First build MP System on standard hardware, then build custom storage servers, simultaneously rewrite MP code in Rust (and abstain from Go) to reduce memory footprint.

Generalization: Step 2

Performance measurement is highly dependent on the utilized definition of performance (technically or economically), generally:

Batch Systems (e.g. Hadoop) are designed for number of records processes per time unit (throughput) or the time until a job is complete (in practice data is not evenly spread across workers and running time of a batch job is longer and throughput may be misleading).

Streaming or online systems are usually designed for response time, meaning the time between sending a request and receiving the response from the system.

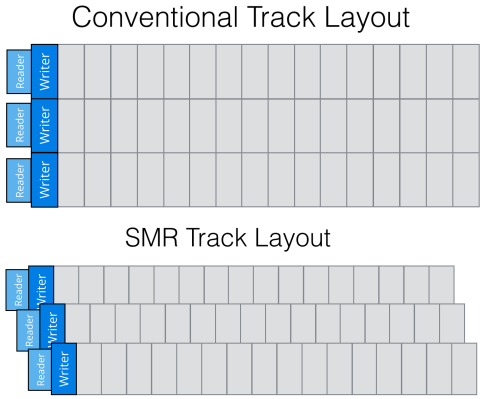
References: Gupta (2016), Cowling (2018), Metz (2016), Kleppmann (2017)

Dropbox Magic Pocket Example

System Improvements and a glimpse on (technical) Performance

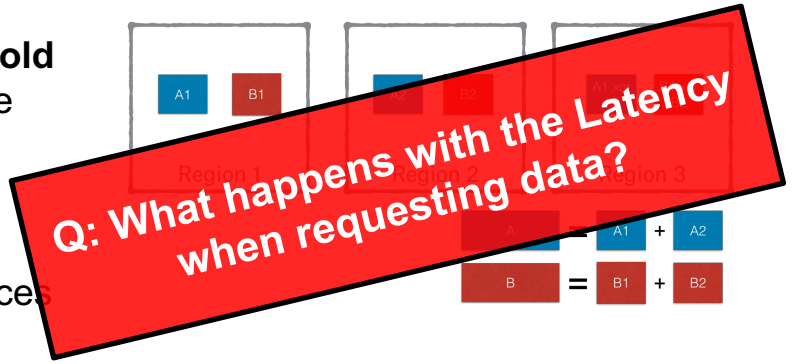
Shingled Magnetic Recording (SMR) HDDs

- Divide MP into warm and cold storage tiers while using Shingled Magnetic Recording (SMR) HDDs for the cold tier.
- SMR HDDs offer increased density by sacrificing random writes for forced sequential writes. Squeezing the tracks on SMR disks together causes the head to erase the next track (see right).



Three Region Setup

- The **warm tier** replication model is a 1+1 replication scheme (one data + parity fragment). The three region **cold tier** example is a 2+1 replication scheme (two data + one parity fragment).
- Warm and cold tiers feature replication within regions. Given that the intra-region replication follows the same mechanisms, using a three region setup effectively reduces disk space requirements.



Q: What happens with the Latency when requesting data?

References: Dropbox (2018), Le (2019)

Dropbox Magic Pocket Example

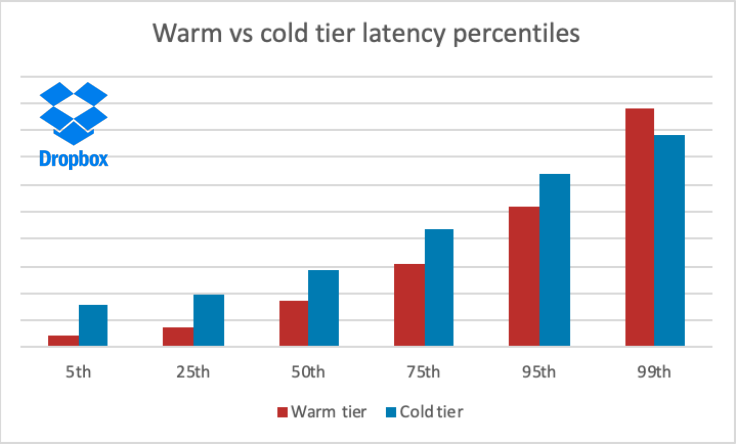
A: What happens to Latencies when requesting Data?

Analysis

- Even in the best-case a request for user data on the cold tier can't be satisfy without fetching fragments from multiple regions.
- This means (on average) latencies for the cold tier should be higher.
- The difference between the warm and cold tiers should stay roughly constant at the level of one network round-trip.
- In the warm tier data is fetched from the geographically nearest region. The other region is only polled when this region is not available or the request timed out.
- This naïve mode makes no sense when two out of three regions are required.

What mode should be used for the cold tier and what happens to large latencies?

Solution



- 5^{th} to 95^{th} latency percentiles grow as expected (time of one additional request).
- Large latencies (99^{th} percentile) are decreased as all three regions are polled and response from the fastest two are used to answer a request.

References: Le (2019)

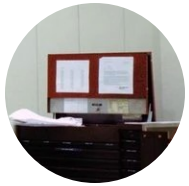
Design Goal: Maintainability

Goal 3

Maintainability

Maintaining current behaviour and adapting to future needs should be possible productively and with reasonable amount of resources.

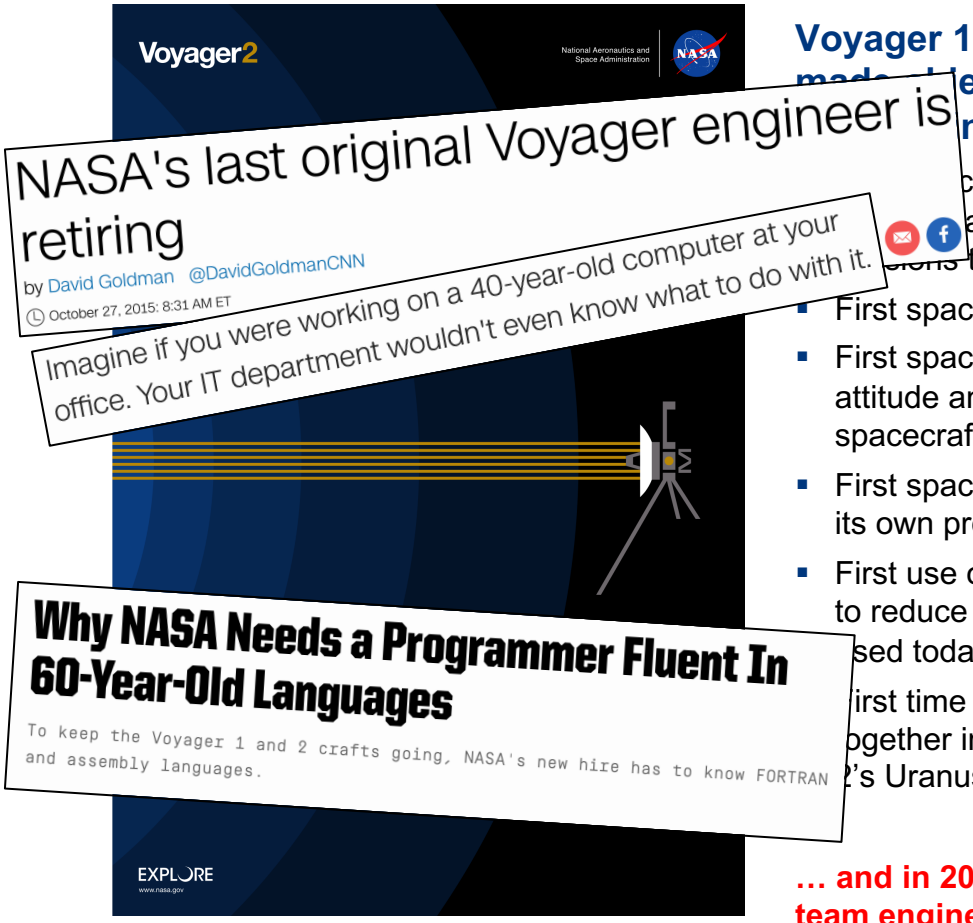
The Man



The System

System Details

NASA Voyager Spacecrafts



NASA's last original Voyager engineer is retiring

by David Goldman @DavidGoldmanCNN

October 27, 2015: 8:31 AM ET

Imagine if you were working on a 40-year-old computer at your office. Your IT department wouldn't even know what to do with it.

Why NASA Needs a Programmer Fluent In 60-Year-Old Languages

To keep the Voyager 1 and 2 crafts going, NASA's new hire has to know FORTRAN and assembly languages.

Voyager 1 and Voyager 2 are the most distant human-made objects in space and hold the following engineering and computing records:

- First spacecraft extensively protected against radiation, which also set the standard for radiation design margin still in use for space missions today
- First spacecraft protected against external electrostatic discharges
- First spacecraft **with programmable computer-controlled** attitude and articulation (which means the pointing of the spacecraft)
- First spacecraft with **autonomous fault protection**, able to detect its own problems and take corrective action
- First use of Reed-Solomon code for spacecraft data - an algorithm to reduce errors in data transmission and storage, which is widely used today
- First time engineers linked ground communications antennas together in an array to be able to receive more data (for Voyager 2's Uranus encounter)

… and in 2015 NASA announced that the last original project team engineer is going to retire.

References: Landau (2017)

Maintainability

The Job and why nobody likes it

Software Maintenance usually means the following:

- Fixing Bugs
- Keeping the System operational
- Investigating Failures
- Adapting the System to new Platforms and modifying it for alternative use cases
- Adding new features

Software Maintenance is hard, because…

- it involves fixing other peoples mistakes.
- working with outdated platforms.
- it often requires forcing systems to do things that they were not intended to do.
- every legacy system is unpleasant in its own individual way.

… in practice this means …

References: Kleppmann (2017)

Main Areas of Software Maintenance

Operability, Simplicity, Evolvability

Software Maintenance can be divided into three main areas, which can all be interpreted as non-functional requirements for data-intensive applications:

Operability

- Make it easy for the operations team (“Ops”) to keep the system running smoothly.
- “Good operations can often work around the limitations of bad software, but good software cannot run reliably with bad operations”

Simplicity

- Make it easy for engineers to understand the systems by removing as much complexity as possible from the system (conceptual, not in the sense of usability).

Evolvability (a.k.a. Extensibility, Modifiability, Plasticity)

- Make it easy for engineers to make changes to the system and adapting it for unanticipated use cases as requirements change.

References: Kleppmann (2017), Kreps (2012)

Operability, Simplicity, Evolvability

Area Deep Dive

Operability

Operability can be achieved by making routine tasks of the ops team easy, e.g. by:

- Provide support for automation using standard tools.
- Avoid dependence on individual machines and system components.
- Provide good documentation with an easy to understand operational model (If X, then Y).
- Be predictable and minimize surprises.

Simplicity

- Large projects often become complex and difficult to understand (called “Big Ball of Mud”).
- Abstractions is a tool against complexity and means reducing complexity esp. accidental complexity that arises through the nature of the implementation, but not from the problem (example: SQL).
- Finding good new Abstractions is hard, but there are many algorithms that help keep complexity low (e.g. MapReduce).

Evolvability

- Requirements will change due to new insights, user requests, regulatory requirements, etc.
- Agile patters help adapting to change, e.g. test-driven development and refactoring.
- Agile techniques often focus on small parts of a project and need to be adapted when designing entire systems.
- Evolvability is tightly linked to Simplicity: Simple systems are easier to modify.

References: Kreps (2012), Hamilton (2007), Cook (1998), Moseley and Marks (2006), Brooks (1995), Hickey (2011), Breivold et al (2008)

Summary and Take Aways

Introduction: Foundations of Data Systems

In this Introductory chapter, we …

- understood that **Reliability** means making systems work correctly in the presence of faults. While hardware faults (usually) occur randomly, software bugs occur systematically. Good fault-tolerant systems hide faults from the end user and that faults should be actively managed instead of trying to avoid them.
- learned that **Scalability** means having plans for keeping performance stable when load patterns change (growth/shrinkage). Scalability requires first describing load and performance with measurable and for the business needs relevant quantitative indicators before implementing pathways to deal with increasing or changing load.
- saw that **Maintainability** is about making life better for the engineering and operations teams, who need to work with our system in the future. Good systems use abstractions that reduce complexity and make it easy to adapt the system to (unintended) use cases. Operability is about exposing system health and implementing effective ways of managing it.
- borrowed experience and looked at examples from two amazing companies and a federal authority (Netflix, Dropbox, NASA).

Thank you for your Attention

I hope you learned something!

Prof. Dr. Benjamin Buchwitz
Phone: 0291 9910 4520
E-Mail: buchwitz.benjamin@fh-swf.de
Office: 2.3.14 (Meschede, Lindenstr. 43)
Consultation by Appointment per E-Mail

References

Chapter: Introduction

- Barausse, Azurra Maria; Cilardo, Chiara (2014): Dropbox – Architecture and Business Perspective, Presto Team Presentation.
- Breivold, Hongyu Pei; Crnkovic, Ivica; Eriksson, Peter J. (2008): Analyzing Software Evolvability, 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC).
- Brooks, Frederick P. (1995): No Silver Bullet – Essence and Accident in Software Engineering, The Mythical Man-Month, Anniversary edition, Addison-Wesley.
- Chang, Alan; Tschaen, Brendan; Benson, Theophilus; Vanbever, Laurent (2015): Chaos Monkey: Increasing SDN Reliability through Systematic Network Destruction, SIGCOMM 2015, London, United Kingdom.
- Cook, Richard I. (1998): “How Complex Systems Fail,” Cognitive Technologies Laboratory.
- Cowling, James (2016): Inside the Magic Pocket, Dropbox Tech Blog.
- Cowling, James (2018): Dropbox – Background on Magic Pocket, Presentation at Techfield Days.
- Dropbox (2015): Dropbox Business Security, Dropbox Whitepaper.
- Dropbox (2018): Extending Magic Pocket Innovation with the first petabyte scale SMR drive deployment, Magic Pocket Hardware Engineering Team, Dropbox Tech Blog.
- Dropbox (2019): Dropbox Business Security, Dropbox Whitepaper.
- Edberg, Jeremy (2012): Rainmakers - How Netflix Operates Clouds for Maximum Freedom and Agility, AWS re:invent 2012 Presentation.
- Goodwater Capital (2018): Understanding Dropbox: Consumerizing the Cloud, Report.
- Gupta, Akhil (2016): Scaling to exabytes and beyond, Dropbox Tech Blog.

References

Chapter: Introduction

- James Hamilton (2007): On Designing and Deploying Internet-Scale Services, 21st Large Installation System Administration Conference (LISA).
- Hickey, Rich (2011): Simple Made Easy, Strange Loop.
- Jay Kreps (2012) “Getting Real About Distributed System Reliability,” blog.empathybox.com.
- Kleppmann, Martin (2017): Designing Data-Intensive Applications - The Big Ideas Behind Reliable, Scalable, and Maintainable Systems, O‘Reilly.
- Landau, Elizabeth (2017): First and Farthest: How the Voyagers Blazed Trails, Jet Propulsion Laboratory, California Institute of Technology, National Aeronautics and Space Administration, Pasadena California, USA.
- Le, Preslav (2019): How we optimized Magic Pocket for cold storage, Dropbox Tech Blog.
- Metz, Casde (2016): The Epic Story of Dropbox's Exodus From the Amazon Cloud Empire, Wired.com Article.
- Moseley, Ben; Marks, Peter (2006): Out of the Tar Pit, BCS Software Practice Advancement (SPA).
- Principles of Chaos Engineering (2018): Principles of Chaos Engineering, principlesofchaos.org.
- Senapaty, Shashank (2017): Introducing Cape, Dropbox Tech Blog.