ISEngineering

Technische Universität Berlin

Enterprise Computing: Exercise 3 – CAP, Dynamo, GFS

Marco Peise

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Agenda

Lecture Microservices

Exercise 3

- CAP Theorem
- Dynamo
- GFS

Technische Universität Berlin

Lecture Microservices

Agenda

1. Web Engineering Fundamentals
    a. HTTP
    b. REST
    c. SOAP
    d. WSDL
    e. WS-*
2. Cloud Fundamentals
    a. Sample Cloud Services: AWS
    b. Design Principles
3. Microservices
    a. DevOps, CI/CD
    b. Microservices and Server-less Architectures

Enterprise Systems, 2016

Technische Universität Berlin

"The cloud" is the de-facto platform when building distributed systems

Containerization, *software-defined networking* and other trends advance cloud technology

Continuous delivery, *continuous integration* and *DevOps-based practices* radically change the way how tech-based businesses evolve

Microservices is a "cloud-native" architecture taking advantage of both

DevOps in a nutshell

Technische Universität Berlin

DevOps most generally is about software and business engineering

…emphasizing short, iterative planning and development cycles

- supporting changing requirements as projects evolve
- enabling fast and frequent delivery of high-quality functionality

…improving communication and collaboration between different teams

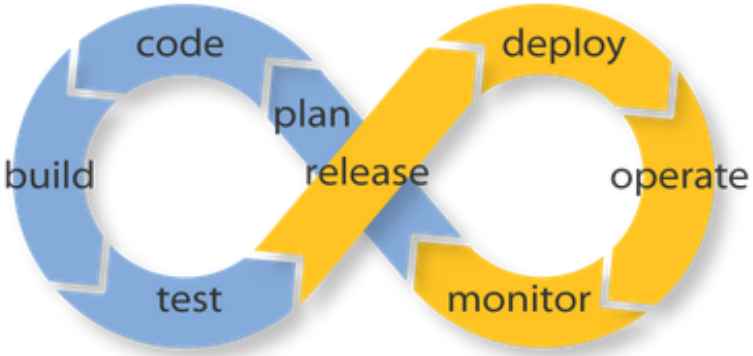
- focusing on small, highly decoupled tasks
- Communicating via language-agnostic APIs

overall establishing development practices that leverage frequent code commits, automated verification and builds, and early problem detection

Dev and Ops

Technische Universität Berlin

Dev: Agility, many versions and features



DevOps: Many stable releases

Ops: Stability, few releases

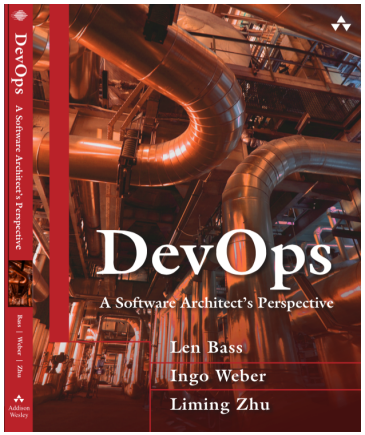
ISEngineering
Wirtschaftsinformatik – Information Systems Engineering

DevOps Definition

"DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality."

Source:

[BWZ 2015] *DevOps: A Software Architect's Perspective*.
Len Bass, Ingo Weber, Liming Zhu, Addison-Wesley Professional, 2015.
http://amzn.to/1Qlar8K



What does that mean?

Technische Universität Berlin

Quality of the code must be high

- Testing & test-driven development

Quality of the build & delivery mechanism must be high

- Automation & more testing
- A must when deploying to production 25x per day (etsy.com)

Time is split:

- From commit until deployment to production
- From deployment until acceptance into normal production
- Means testing in production

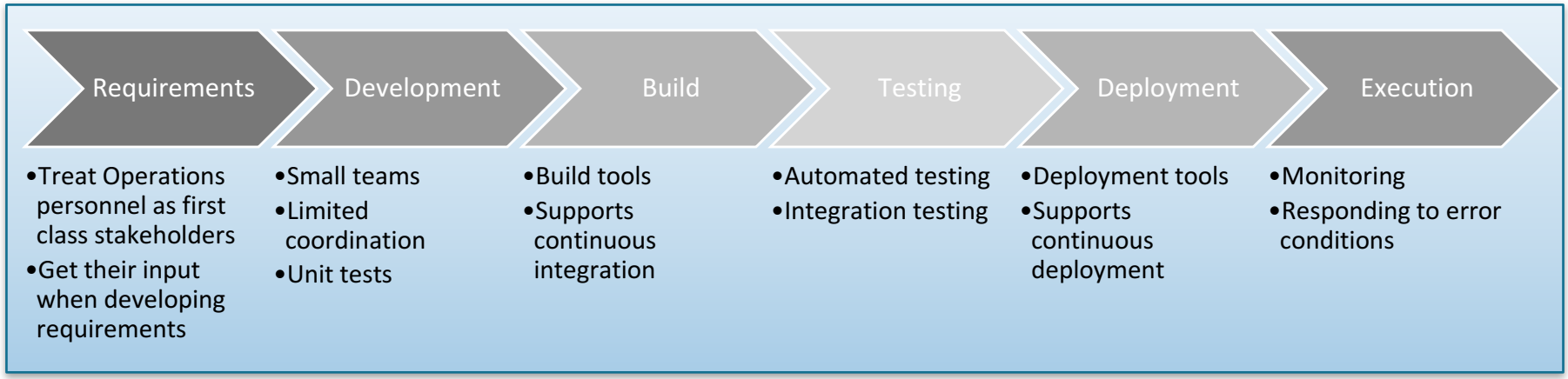
Goal-oriented definition

- May use agile methods, continuous integration, etc.
- Likely to use automation tools

Source: [BWZ 2015]

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

DevOps Practices (1/2)



Treat Ops as first-class citizens throughout the lifecycle – e.g., in requirements elicitation

- Many decisions can make operating a system harder or easier
- Logging and monitoring to suit Ops

Make Dev more responsible for relevant incident handling

- Shorten the time between finding and repairing errors

Source: [BWZ 2015]

DevOps Practices (2/2)

Use continuous deployment, automate everything

- Commits trigger automatic build, testing, deployment

Enforce deployment process is used by all

- No ad-hoc deployments
- Ensures changes are traceable

Develop infrastructure code with the same set of practices as application code

- “Infrastructure as Code”: using IaaS APIs, etc., to automate creation of environments
- Misconfiguration can derail your application
- Ops scripts are traditionally more ad-hoc

Source: [BWZ 2015]

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

DevOps consequences

Technische Universität Berlin

Architecturally significant requirement:

Speed up deployment through minimizing synchronous coordination among development teams.

Synchronous coordination, like a meeting, adds time since it requires

- Ensuring that all parties are available
- Ensuring that all parties have the background to make the coordination productive
- Following up to decisions made during the meeting

Source: [BWZ 2015]

DevOps consequences

Technische Universität Berlin

Keep teams relatively small

- "Two pizza rule": no team should be larger than can be fed with two pizzas
- Advantages: make decisions quickly, less coordination overhead, more coherent units

Business boundaries as service boundaries

- Small, autonomous teams focused on doing one thing well, building small services → Microservices
- Channel most interaction through service interfaces

Source: [BWZ 2015]

Microservices

…are small, autonomous services providing a small, focused amount of functionality

…to address the DevOps objective for fast and high-quality change management:

- Small teams develop small services
- Coordination overhead is minimized by channeling most interaction through service interfaces:
  - Team X provides service A, which is used by teams Y and Z
  - If changes are needed, they are communicated, implemented, and added to the interface

Team size becomes a major driver of the overall architecture!

Adapted from: [BWZ 2015]

Microservices [Fowler]

Technische Universität Berlin

"In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

These services are built around business capabilities and independently deployable by fully automated deployment machinery.

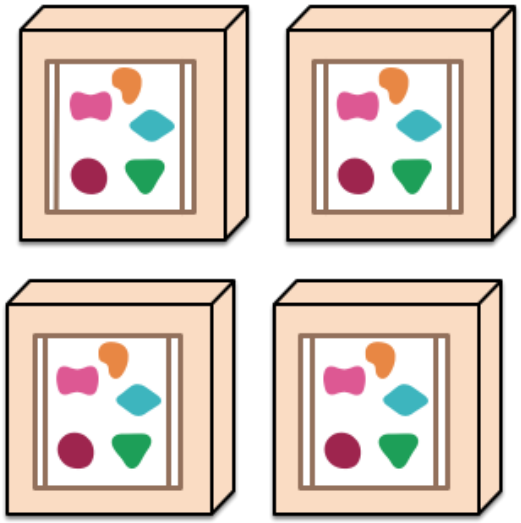
There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."

http://martinfowler.com/articles/microservices.html

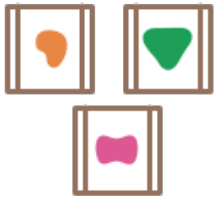
Monoliths versus Microservices

A monolithic application puts all its functionality into a single process...

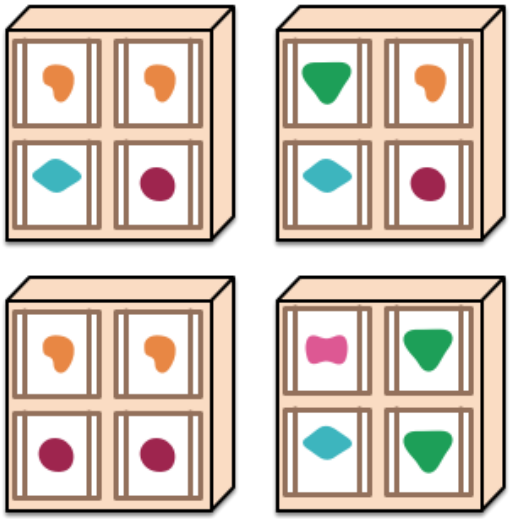
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...

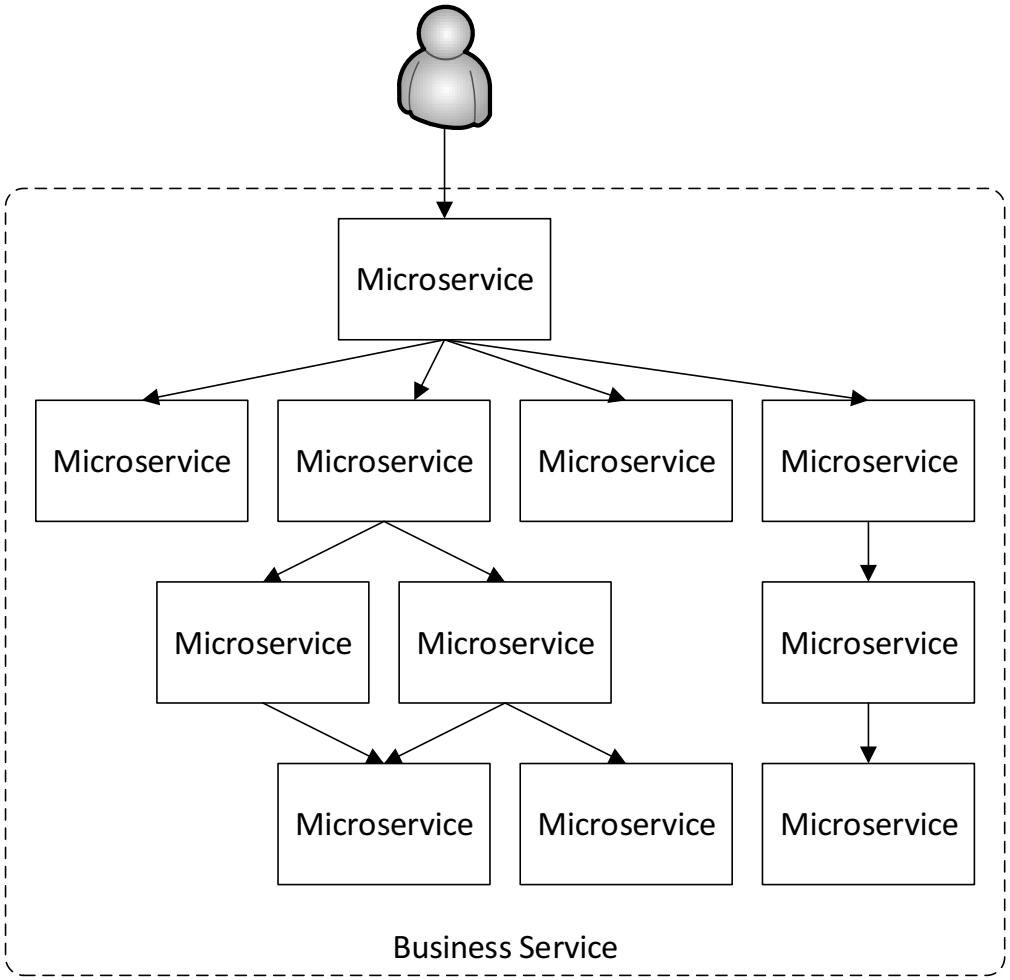


... and scales by distributing these services across servers, replicating as needed.



http://martinfowler.com/articles/microservices.html

Microservice Architecture



Each user request is satisfied by some sequence of services

Most services are not externally available

Each service communicates with other services through service interfaces

Service depth?

Source: [BWZ 2015]

Amazon design rules

All teams will henceforth expose their data and functionality through service interfaces.
Teams must communicate with each other through these interfaces.

There will be no other form of inter-process communication allowed:

- no direct linking, no direct reads of another team's data store,
- no shared-memory model, no back-doors whatsoever.
- The only communication allowed is via service interface calls over the network.

It doesn't matter what technology they[services] use.

All service interfaces, without exception, must be designed from the ground up to be externalizable.

Source: [BWZ 2015]

Characteristics of Microservices [Fowler]

1. Componentization via Services
2. Organized around Business Capabilities
3. Products not Projects
4. Smart endpoints and dumb pipes
5. Decentralized Governance
6. Decentralized Data Management
7. Infrastructure Automation
8. Design for failure
9. Evolutionary Design

http://martinfowler.com/articles/microservices.html

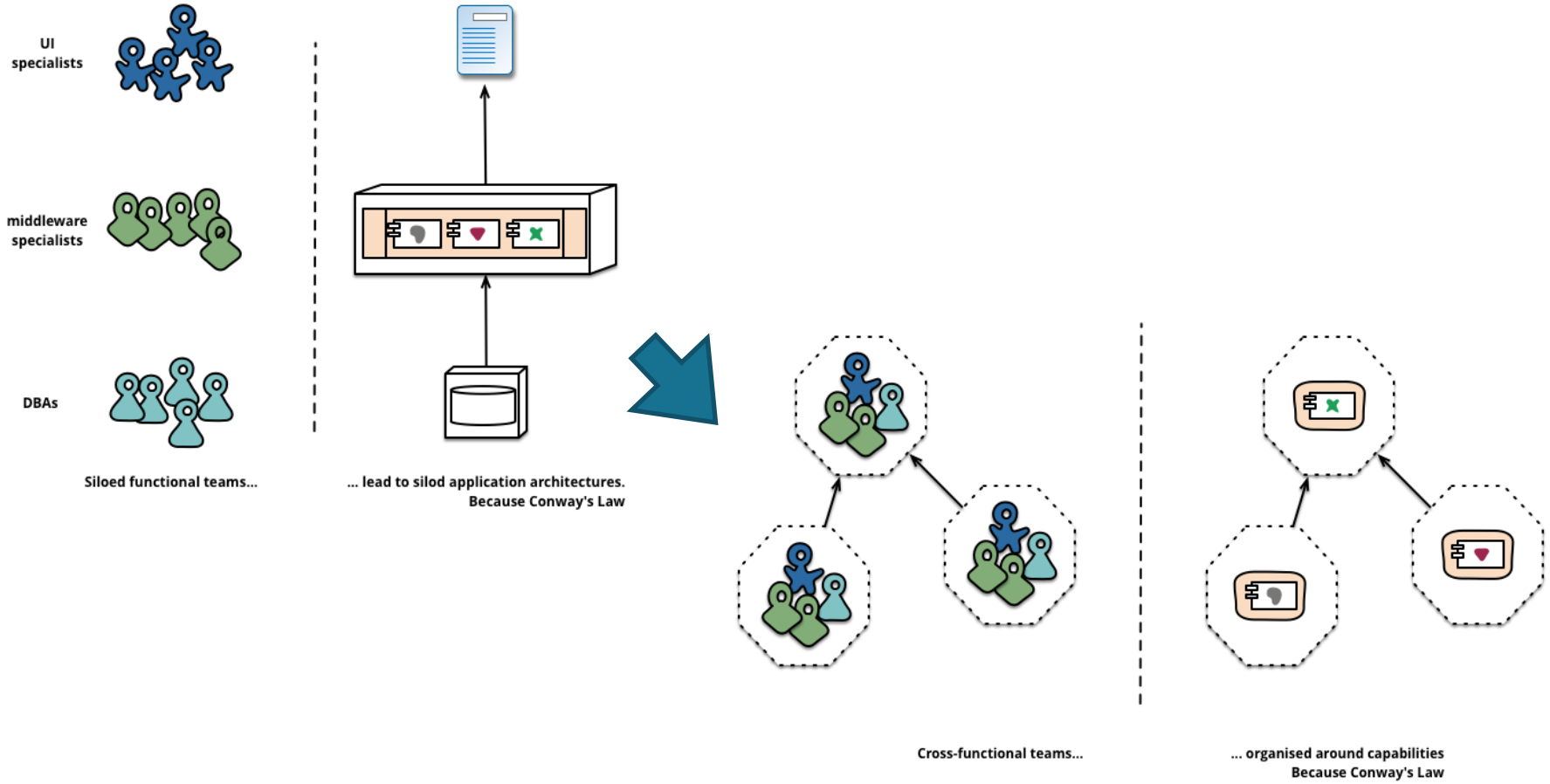
1. Componentization via Services

A component is a unit of software that is independently replaceable and upgradeable.
Microservice architectures will use libraries, but their primary way of componentizing their own software is by breaking down into services.

We define libraries as components that are linked into a program and called using in-memory function calls, while services are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call.

http://martinfowler.com/articles/microservices.html

2. Organized around Business Capabilities



http://martinfowler.com/articles/microservices.html

3. Products, not Projects

Technische Universität Berlin

Project model: the aim is to deliver some piece of software which is then considered to be completed. On completion the software is handed over to a maintenance organization and the project team that built it is disbanded.

Product model: A development team takes full responsibility for the software in production (“you build, you run it”).

http://martinfowler.com/articles/microservices.html

4. Smart endpoints and dumb pipes

Technische Universität Berlin

Applications built from microservices aim to be as decoupled and as cohesive as possible – they own their own domain logic and act more as filters in the classical Unix sense – receiving a request, applying logic as appropriate and producing a response.

These are choreographed using simple RESTish protocols rather than complex protocols such as WS-Choreography or BPEL or orchestration by a central tool.

http://martinfowler.com/articles/microservices.html

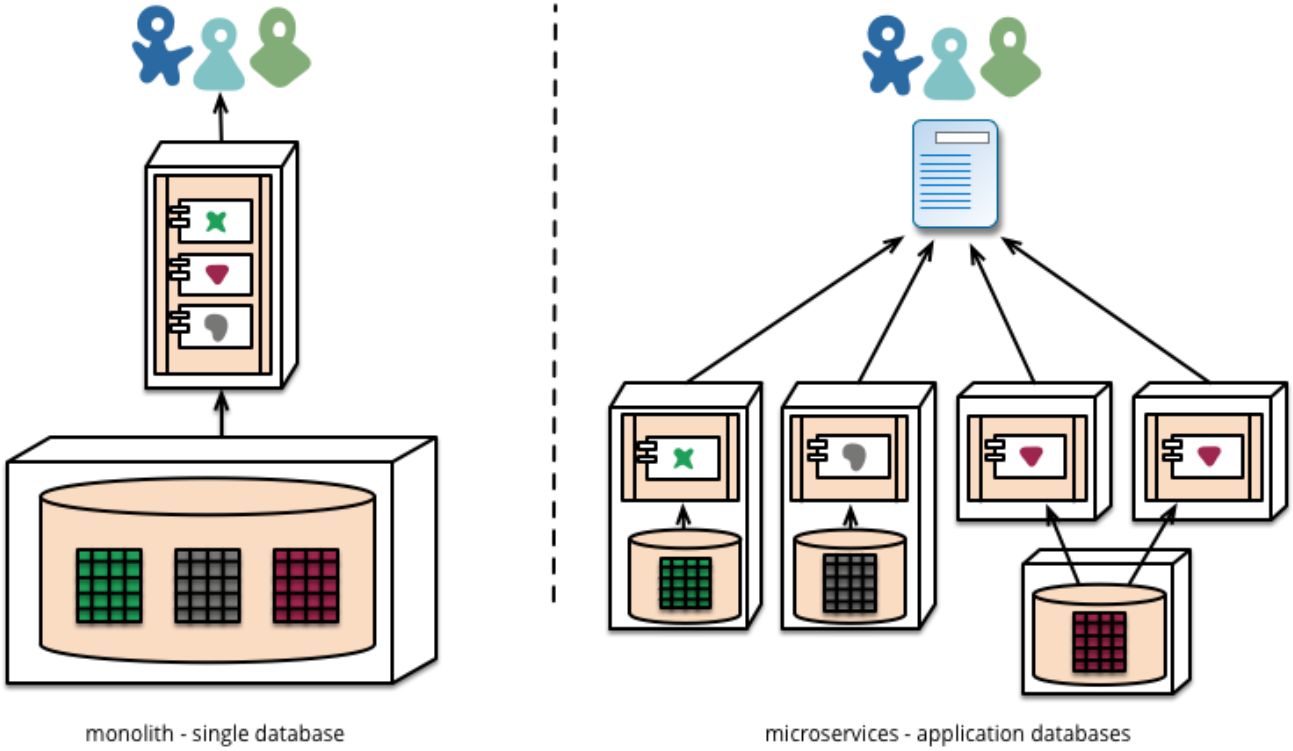
5. Decentralized Governance

One of the consequences of centralized governance is the tendency to standardize on single technology platforms. Experience shows that this approach is constricting - not every problem is a nail and not every solution a hammer.

Rather than use a set of defined standards written down somewhere on paper they prefer the idea of producing useful tools that other developers can use to solve similar problems to the ones they are facing.

http://martinfowler.com/articles/microservices.html

6. Decentralized Data Management

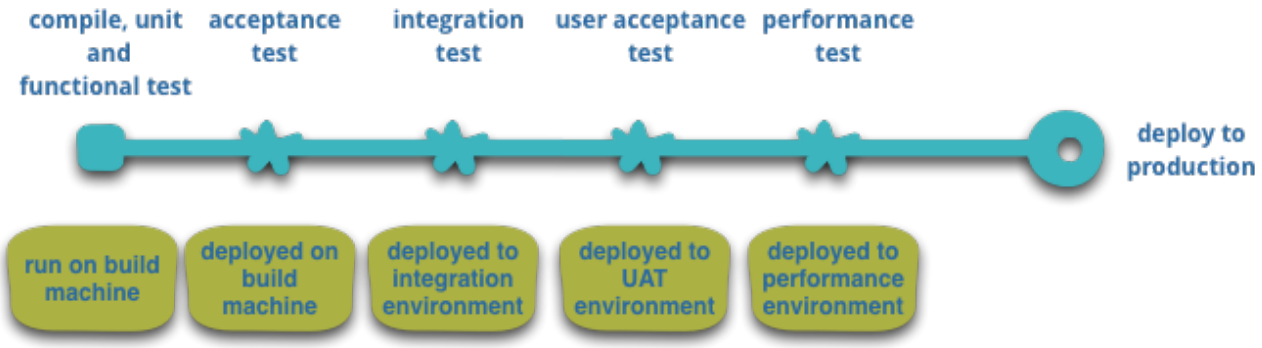


http://martinfowler.com/articles/microservices.html

7. Infrastructure Automation

Technische Universität Berlin

Automated, continuous deployment and testing processes



http://martinfowler.com/articles/microservices.html

ISEngineering
Wirtschaftsinformatik – Information Systems Engineering

8. Design for failure

A consequence of using services as components, is that applications need to be designed so that they can tolerate the failure of services.

Any service call could fail due to unavailability of the supplier, the client has to respond to this as gracefully as possible. This is a disadvantage compared to a monolithic design as it introduces additional complexity to handle it.

The consequence is that microservice teams constantly reflect on how service failures affect the user experience. Netflix's Simian Army induces failures of services and even datacenters during the working day to test both the application's resilience and monitoring.

http://martinfowler.com/articles/microservices.html

9. Evolutionary Design

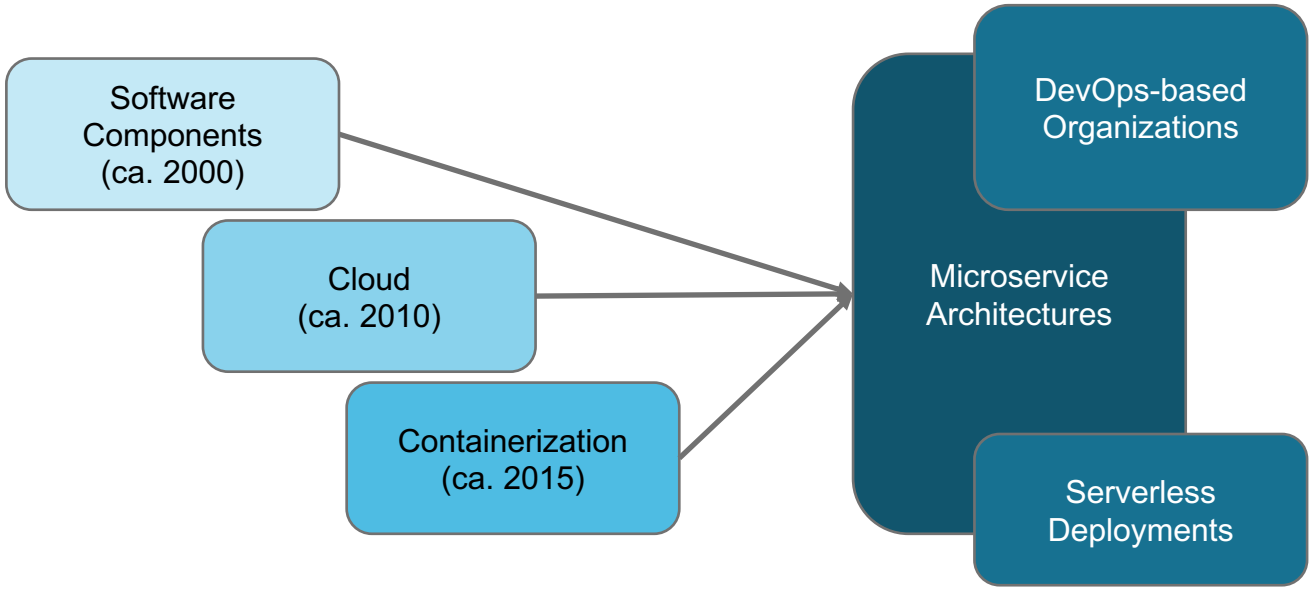
The key property of a component is the notion of independent replacement and upgradeability.

You want to keep things that change at the same time in the same module. Parts of a system that change rarely should be in different services to those that are currently undergoing lots of churn.

Putting components into services adds an opportunity for more granular release planning. With a monolith any changes require a full build and deployment of the entire application. With microservices, however, you only need to redeploy the service(s) you modified. This can simplify and speed up the release process.

http://martinfowler.com/articles/microservices.html

The evolution towards Microservices



toolchain

Code – Code development and review, continuous integration tools - Git
Build – Version control tools, code merging, build status - Jenkins
Test – Test and results determine performance - Bugzilla
Package – Artifact repository, application pre-deployment staging - Artifactory
Release – Change management, release approvals, release automation - Jenkins
Configure – Infrastructure configuration and management, Infrastructure as Code tools - Ansible
Monitor – Applications performance monitoring, end user experience - Nagios
Log Management – Dealing with log files – Elasticsearch, Logstash & Kibana (ELK)

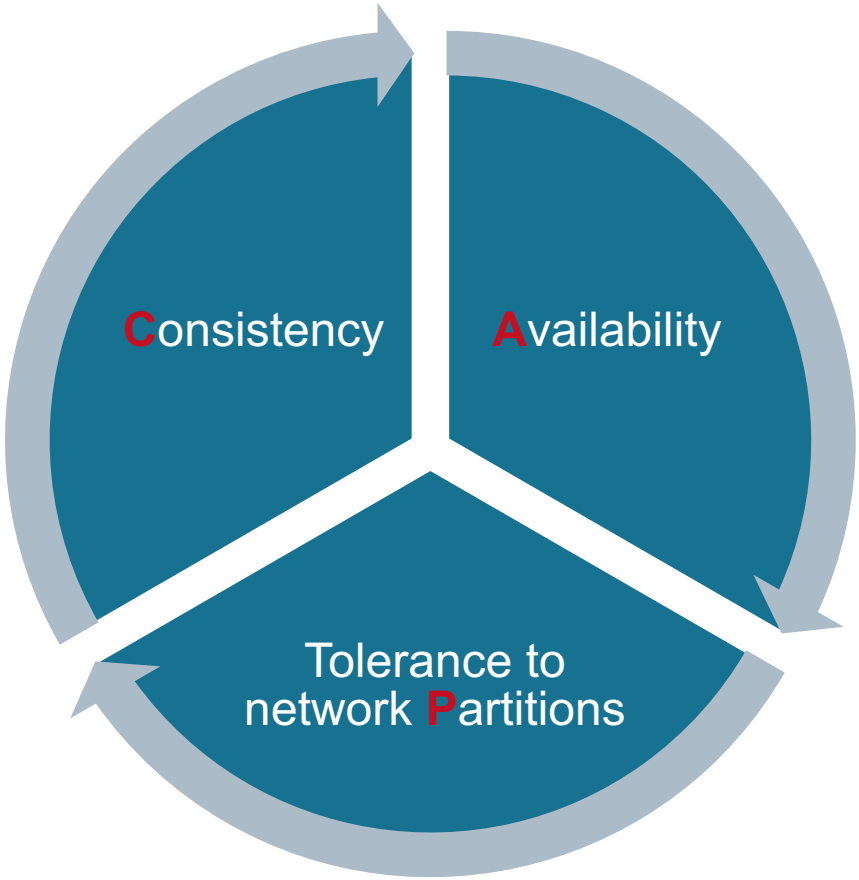
Containerization - Docker,
continuous Integration - Jenkins,
Infrastructure as Code – Puppet,
virtualization platform - Vagrant

Technische Universität Berlin

Exercise 3

Brewer‘s CAP Theorem

You can have at most two of these properties for any shared-data system:



Task 1 CAP Theorem

a) Shortly explain the CAP theorem by example of the Domain Name System (DNS).

b) Shortly describe two dimensions of data consistency from both a data-centric and a client-centric perspective.

Techniques used in Dynamo

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

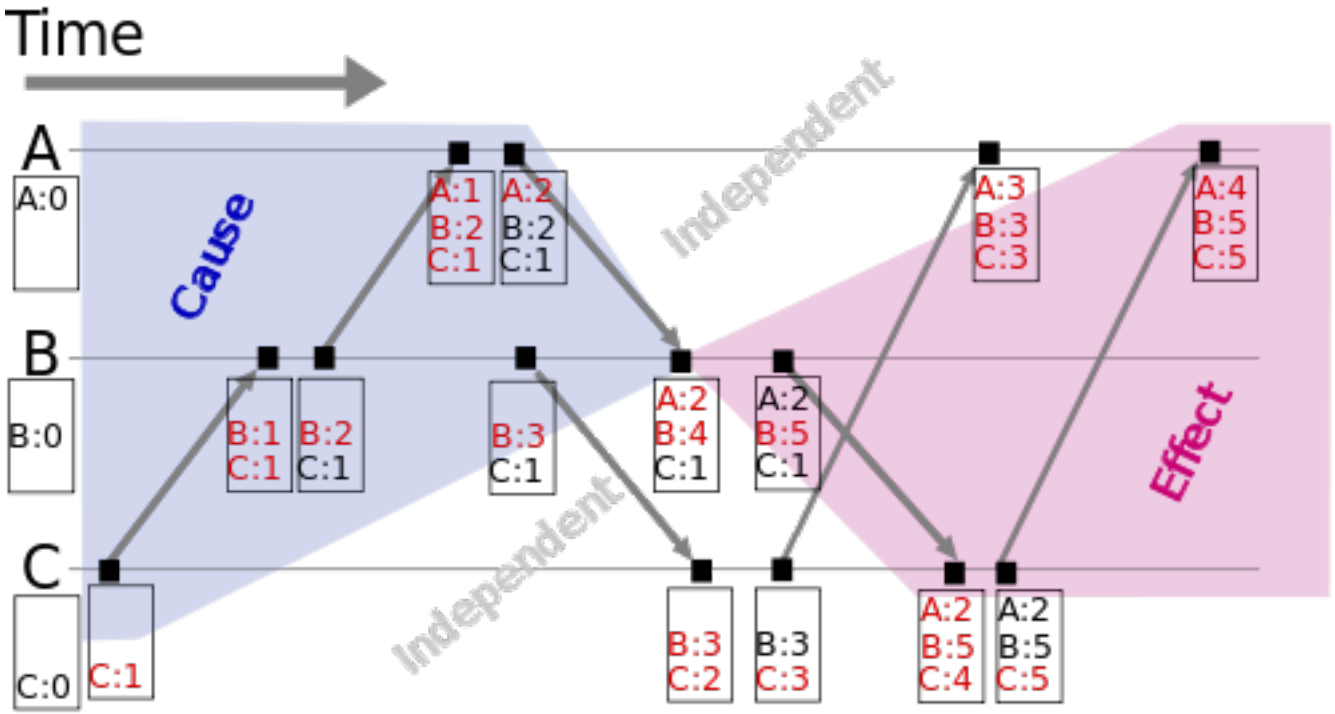
Task 2 Dynamo

Technische Universität Berlin

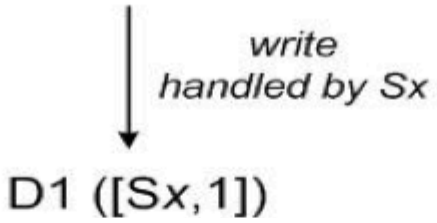
a) Which queries does Dynamo support and for which type of data storage is Dynamo optimized?

b) Pessimistic replication (as implemented in Dynamo, for example) is used to offer high availability and low latency. True or false?

Vector Clock Example


Time
A
A:0
B
B:0
C
C:0
C:1
B:1
C:1
B:2
C:1
A:1
B:2
C:1
A:2
B:2
C:1
B:3
C:1
A:2
B:4
C:1
A:2
B:5
C:1
B:3
C:2
B:3
C:3
A:2
B:5
C:4
A:2
B:5
C:5
A:3
B:3
C:3
A:4
B:5
C:5
Cause
Effect
Independent
Independent

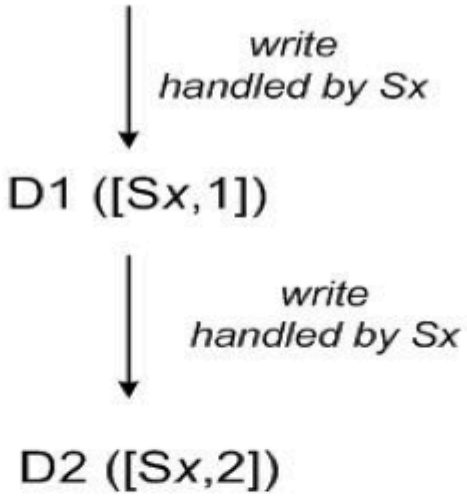
Vector clock example



- A client writes a new object. The node (say Sx) that handles the write for this key increases its sequence number and uses it to create the data's vector clock. The system now has the object D1 and its associated clock [(Sx, 1)].

Source: http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

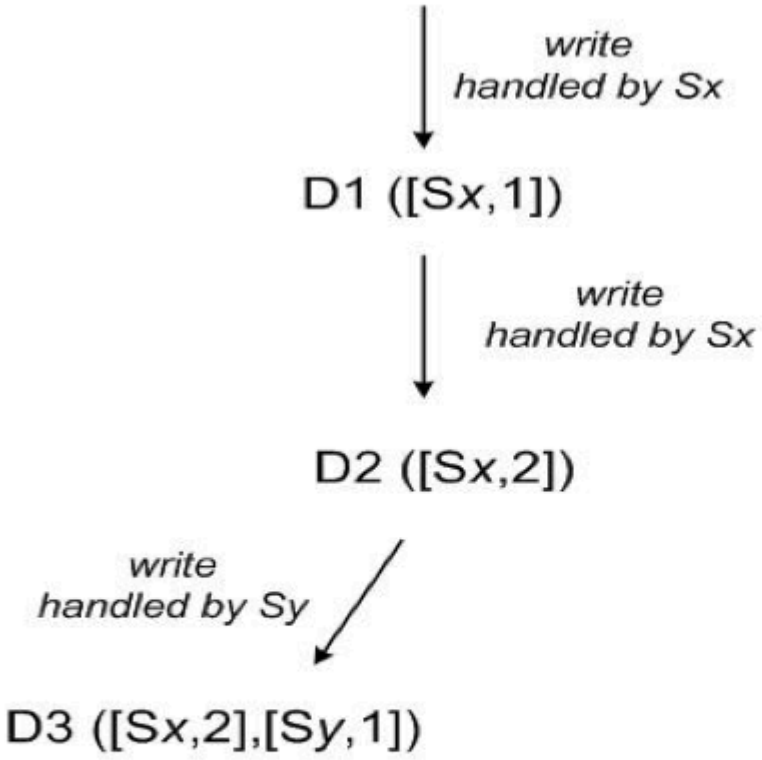
Vector clock example



- The client updates the object. Assume the same node handles this request as well. The system now also has object D2 and its associated clock [(Sx, 2)]. D2 descends from D1 and therefore over-writes D1.

Source: http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

Vector clock example



However there may be replicas of D1 lingering at nodes that have not yet seen D2.

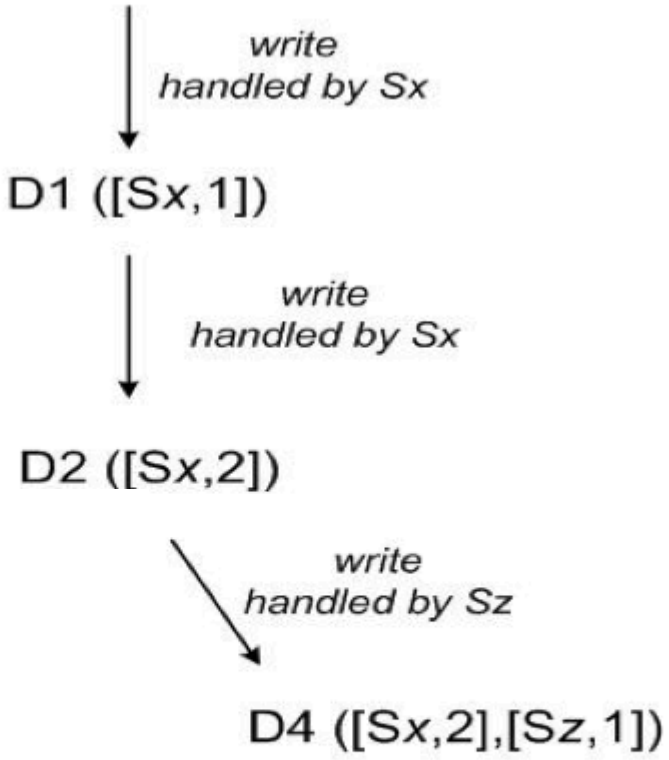
Let us assume that the same client updates the object again and a different server (say Sy) handles the request. The system now has data D3 and its associated clock [(Sx, 2), (Sy, 1)].

Source: http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

Vector clock example

Next assume a different client reads D2 and then tries to update it, and another node (say Sz) does the write.

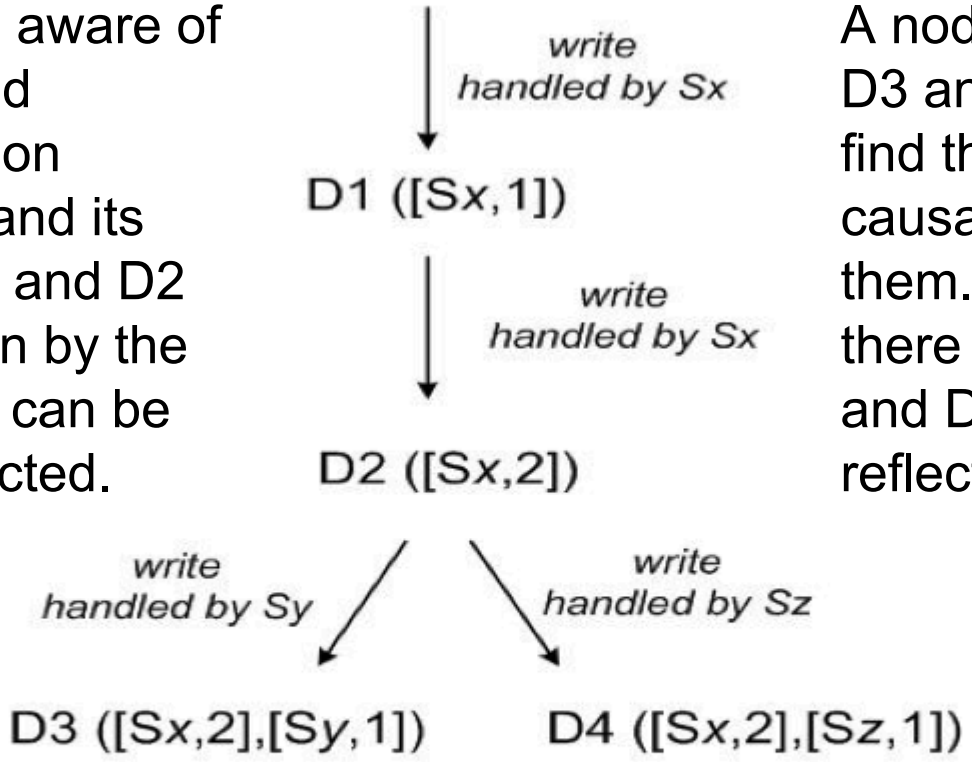
The system now has D4 (descendant of D2) whose version clock is [(Sx, 2), (Sz, 1)].



Source: http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

Vector clock example

A node that is aware of D1 or D2 could determine, upon receiving D4 and its clock, that D1 and D2 are overwritten by the new data and can be garbage collected.



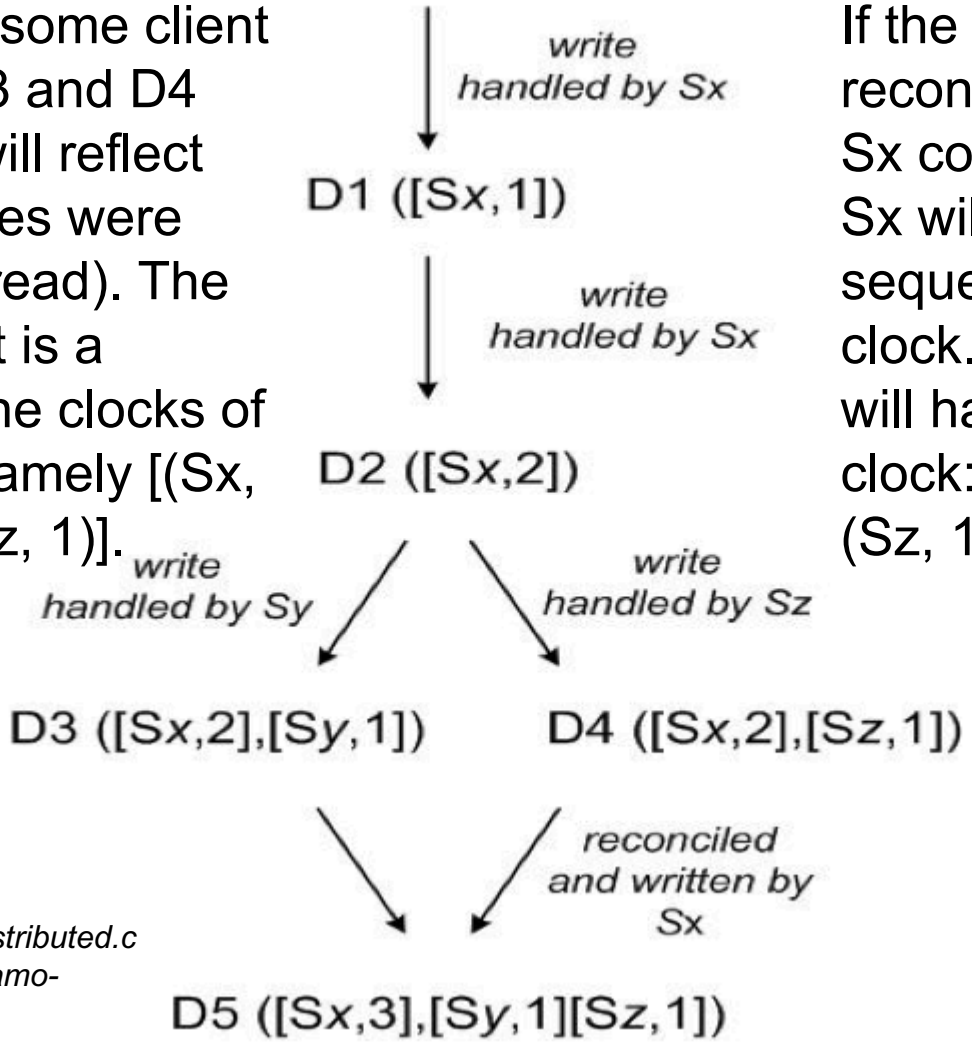
A node that is aware of D3 and receives D4 will find that there is no causal relation between them. In other words, there are changes in D3 and D4 that are not reflected in each other.

Both versions of the data must be kept and presented to a client (upon a read) for semantic reconciliation.

Source: http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

Vector clock example

Now assume some client reads both D3 and D4 (the context will reflect that both values were found by the read). The read's context is a summary of the clocks of D3 and D4, namely [(Sx, 2), (Sy, 1), (Sz, 1)].



If the client performs the reconciliation and node Sx coordinates the write, Sx will update its sequence number in the clock. The new data D5 will have the following clock: [(Sx,3), (Sy, 1), (Sz, 1)].

Source: http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

Task 2 Dynamo c)

Dynamo uses vector clocks to determine the total order of write operations. Given the vector clocks in the table below with conflicting versions on servers A, B, and C. Please state whether or not the conflict can be reconciled automatically (yes/no) and how the vector clock must look like after a conflict resolution.

Task 2 Dynamo c)

Vector clocks before conflict resolution	Can be reconciled automatically (yes/no)	Vector clocks after conflict resolution
D1 ([A,1]) D2 ([B,1]) D3 ([C,1])		
D1 ([A,1] [B,2]) D2 ([A,2] [B,2])		
D1 ([A,1] [B,4] [C,13]) D2 ([A,2] [B,3] [C,15]) D3 ([A,2] [B,4] [C,15]) D4 [A,1] [B,3] [C,14])		
D1 ([A,1] [B,2] [C,1]) D2 ([A,2] [B,2])		

Task 2 - Dynamo

d) Shortly explain the trade-off between consistency, read latency, and write latency in Dynamo and how a Dynamo-based application could be tuned either towards fast reads or towards fast writes using the (N,R,W) configuration.

e) What is the minimum cluster size, i.e., number of servers, of a Dynamo configuration (N=9, R=1, W=9) and why? For this minimum cluster size: how many data records are stored on each node after 1 million data records have been inserted into the Dynamo cluster by a client program?