ソフトウェア・クラウド開発プロジェクト実践 I
Practices for Software and Cloud Development Project I

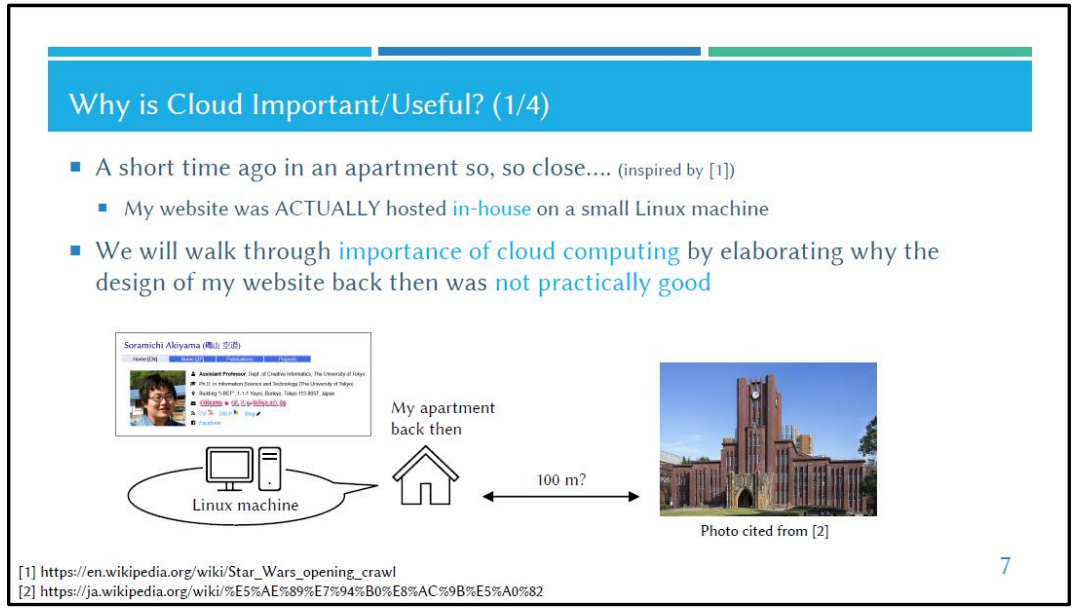
Soramichi Akiyama

穐山 空道

akiyama@ci.i.u-tokyo.ac.jp

Last Time in This Class…

- We learned why cloud is useful and how it is implemented



Recap:

- Cloud can reduce TCO (Total Cost of Owning) by managing computing resources on users' behalf
- Cloud models: SaaS, PaaS, IaaS
- Cloud deployment models: Public clouds, Private cloud (on-premise, off-premise)
- Cloud internal: Virtualization is key

- Today's topic
  - Reliability of cloud itself / services running on cloud

What is Reliability?

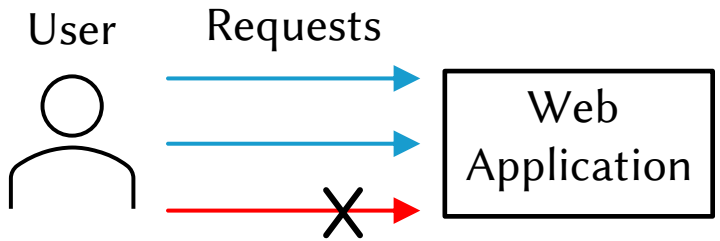
- A reliable system is...
  - Available with high confidence
  - Easily maintainable

- Availability (可用性) ← We focus on this today
  - An available system is accessible and usable as expected

- Maintainability (a.k.a. Serviceability, 保守性)
  - A maintainable system can be fixed when broken

Availability Metrics

- Availability does not just mean "up and running"
  - What does "running" mean? What if it is up but super slow?



User sent 3 requests,
but only 2 of them succeeded:
Availability == 67 % (2 / 3)

- Metric 1: Based on uptime
  - $$\frac{\text{Uptime}}{(\text{Uptime} + \text{Downtime})}$$
- Metric 2: Based on the number of successful queries
  - $$\frac{\text{The number of successfully handled queries}}{\text{The number of incoming queries}}$$
  - More useful than (1) when the system's states are not binary ("on" or "off")
  - Example scenario: A web app slows down when there are too many users, but still not down

Nines Notation

- Availability is often specified by nines notation
  - Example: 99.99 % uptime ("4 nines") means the system is up for 99.99% of the time

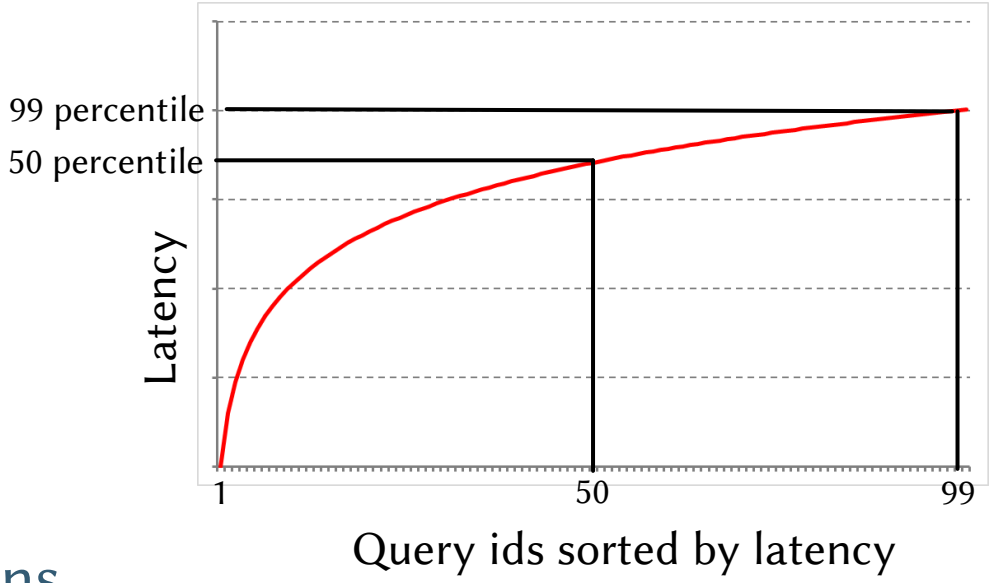
Nines	Two	Three	Four	Five	Six	Seven
%	99	99.9	99.99	99.999	99.9999	99.99999
Downtime in 1 year	4 days	9 hours	1 hour	5 mins	30 seconds	3 seconds

- Adding one nine requires to reduce downtime to 1/10
  - 99.99 % uptime == 0.001 % downtime
  - 99.999 % uptime == 0.0001 % downtime

1/10

Service Level Objectives (SLO)

- Objectives of reliability metrics
  - Defined internally to monitor the system's health
  - Example 1: Query success rate >= 99%
  - Example 2: 99 percentile of query latency < 100 ms



- SLOs guide development / maintenance decisions
  - Without SLOs: "The system looks somewhat slow these days… but should we do something?"
  - With SLOs: "99 percentile latency is reaching almost 100 ms. We should investigate the bottleneck and resolve it immediately."

Service Level Agreements (SLA)

- Agreements of reliability metrics
  - Contracted between users and service providers
  - SLA violations often come with penalties
  - Example: Guarantee 99 % uptime, and reimburse the money if violated

Reimburse

- to pay back money to somebody which they have spent or lost
- **reimburse something** *We will reimburse any expenses incurred.*
- **reimburse somebody (for something)** *You will be reimbursed for any loss or damage caused by our company.*

https://www.oxfordlearnersdictionaries.com/definition/english/reimburse

- SLA != SLO
  - SLO is internal, while SLA is external
  - A system may have no SLA even if it has SLOs (e.g., You have no SLA on using a search engine, while they still have internal SLOs)
  - Defining an SLA involves in business decisions as well as technical ones

Example of Real SLA (Amazon EC2)

General Service Commitment

AWS will use commercially reasonable efforts to make the Included Services each available for each AWS region with a Monthly Uptime Percentage of at least 99.99%, in each case during any monthly billing cycle (the "Service Commitment"). In the event any of the Included Services do not meet the Service Commitment, you will be eligible to receive a Service Credit as described below.

Service Credits

Service Credits are calculated as a percentage of the total charges paid by you (excluding one-time payments such as upfront payments made for Reserved Instances) for the individual Included Service in the affected AWS region for the monthly billing cycle in which the Unavailability occurred in accordance with the schedule below.

Monthly Uptime Percentage	Service Credit Percentage
Less than 99.99% but equal to or greater than 99.0%	10%
Less than 99.0% but equal to or greater than 95.0%	30%
Less than 95.0%	100%

4 nines uptime agreement

Reimbursement policy

https://aws.amazon.com/compute/sla/

Defining Good SLAs

- Well-defined SLAs are not perfect
  - Nine notations are averages
  - Taking averages blurs what is happening

Black Friday (shopping)

From Wikipedia, the free encyclopedia

For other uses, see List of Black Fridays and Black Friday (disambiguation).

Black Friday is a colloquial term for the Friday following Thanksgiving Day in the United States. Many stores offer highly promoted sales on Black Friday and open very early (sometimes as early as midnight[2]), or some time on Thanksgiving Day.

Black Friday has routinely been the busiest shopping day of the year in the United States since at least 2005.[3][4][5]

https://en.wikipedia.org/wiki/Black_Friday_(shopping)

- Failures that are the same in nine notations but different in reality
  1. Server outage in Black Friday vs. in a normal day
  2. Complete outage vs. graceful degradation (system kind of working but out of SLA)
  3. A second of separate downtime everyday vs. 30 seconds of continuous downtime once in a month

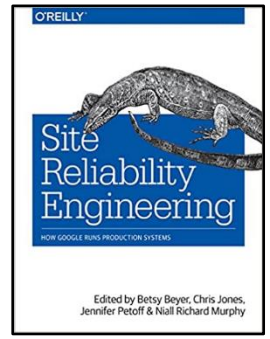
(*) Failures 1 and 2 are taken from:
Jeffrey C. Mogul and John Wilkes, “Nines are Not Enough: Meaning Metrics for Clouds”, *HotOS’20*

[Aside] Error-Budget

- The difference between an SLO and the actual metric value
  - Example: SLO of 99 percentile latency is 100 ms, while it is currently 60 ms

  → This system has 40 ms error-budget (or slack)

- Error-budget is useful as a common language of developers and operators
  - Developers: adding new features first, reliability second
  - Operators: the other way around
  - Quantitative error-budget can help balancing the two (e.g., 40 ms error-budget means that a new feature can probably be added without much concerning the latency)



*More on error-budget: *Site Reliability Engineering*, O'REILLY

Improving Reliability (1/2)

- Traditional way
  - Improve the reliability of underlying components to make the whole system reliable
  - Example: Use highly reliable capacitors (コンデンサ) inside an enterprise-class power supply

最上級の部品

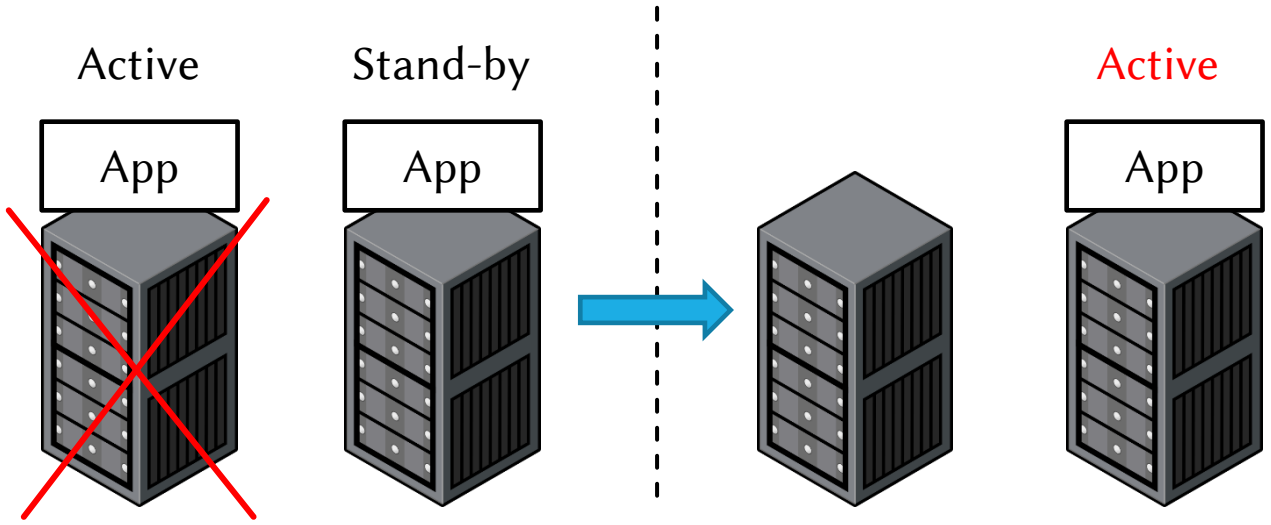
105°C 日本メーカー製 electrolyticコンデンサを使用し、最高仕様の内部部品とデジタル設計により、94% 以上の効率を実現します。

https://www.corsair.com/ja/ja/%E3%82%AB%E3%83%86%E3%82%B4%E3%83%AA%E3%83%BC/%E8%A3%BD%E5%93%81/%E9%9B%BB%E6%BA%90%E3%83%A6%E3%83%8B%E3%83%83%E3%83%88/axi-series-config/p/CP-9020087-JP

- It is the only way in many non-software systems
  - Physical components cannot be replaced or switched easily once deployed

Improving Reliability (2/2)

- Software engineers' way (modern? way)
  - "Embrace" failures by preparing for them
  - Example: Run two instances of your web application and switch between them when one of them fails, instead of running it on a super reliable server (i.e., fail over)



Embrace

2 ★ [uncountable] the act of accepting an idea, a proposal, a set of beliefs, etc, especially when it is done with enthusiasm

- *the country's eager embrace of modern technology*

https://www.oxfordlearnersdictionaries.com/definition/english/embrace_1

Why Software Engineers' Way is Preferrable?

- So many components exist in cloud datacenters
  - Fugaku (富岳) supercomputer has more than 7,500,000 cores! (*)
  - Note: cloud datacenters do not reveal how many components they have, so we use Fugaku as an alternative example

https://www.fujitsu.com/jp/about/businesspolicy/tech/fugaku/

- Things certainly break in such large scale
  - Even a very small failure rate does matter in large scale
  - Toy Example: Suppose 1% of HDDs fail in a year
  - Then 1,000 HDDs fail in a year if a cloud data center contains 100,000 of them!

(*) https://www.top500.org/lists/top500/list/2020/11/

Real Numbers of HDD Failures

- Backblaze (a cloud storage provider) observed 1,302 failing HDDs in 2020
  - Out of 162,299 of them (0.93 %)
  - Detailed observations and test conditions can be found at: https://www.backblaze.com/blog/backblaze-hard-drive-stats-for-2020/

Backblaze Hard Drive Failure Rates for 2020

Reporting period 1/1/2020 - 12/31/2020 inclusive

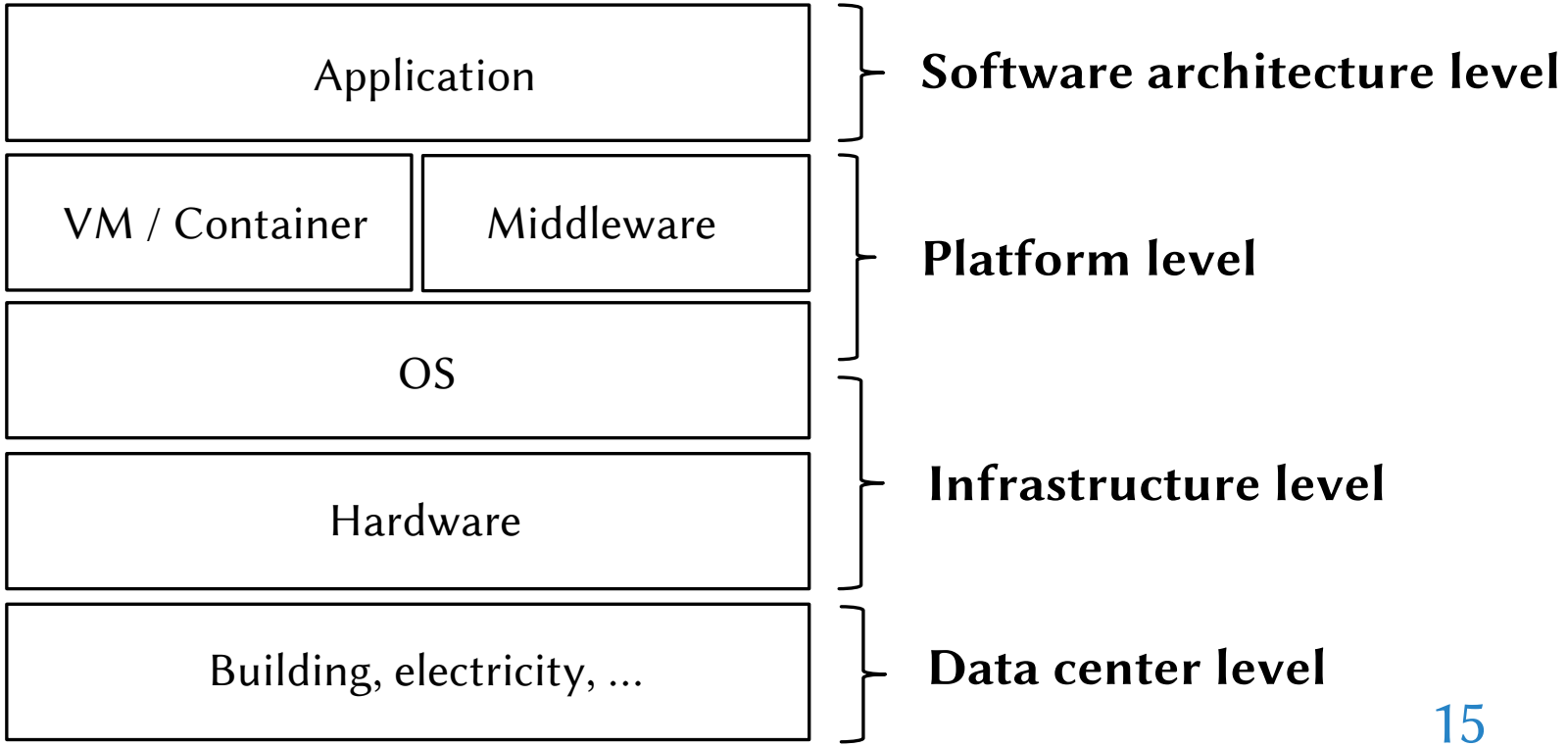
MFG	Model	Drive Size	Drive Count	Avg Age (months)	Drive Days	Drive Failures	AFR
HGST	HMS5C4040ALE640	4TB	3,100	56.65	1,083,774	8	0.27%
HGST	HMS5C4040BLE640	4TB	12,744	50.43	4,663,049	34	0.27%
HGST	HUH728080ALE600	8TB	1,075	34.85	372,000	3	0.29%
HGST	HUH721212ALE600	12TB	2,600	15.04	820,272	7	0.31%
HGST	HUH721212ALE604	12TB	2,506	3.78	275,779	9	1.19%
HGST	HUH721212ALN604	12TB	10,830	21.01	3,968,475	50	0.46%
Seagate	ST4000DM000	4TB	18,939	62.35	6,983,470	269	1.41%
Seagate	ST6000DX000	6TB	886	68.84	324,275	2	0.23%
Seagate	ST8000DM002	8TB	9,772	51.07	3,584,788	91	0.93%
Seagate	ST8000NM0055	8TB	14,406	41.34	5,286,790	177	1.22%
Seagate	ST10000NM0086	10TB	1,201	38.73	439,247	16	1.33%
Seagate	ST12000NM0007	12TB	23,036	29.78	11,947,303	339	1.04%
Seagate	ST12000NM0008	12TB	19,287	9.76	5,329,149	148	1.01%
Seagate	ST12000NM001G	12TB	7,130	6.08	1,296,149	30	0.84%
Seagate	ST14000NM001G	14TB	5,987	2.89	454,090	13	1.04%
Seagate	ST14000NM0138	14TB	360	1.56	5,784	0	0.00%
Seagate	ST16000NM001G	16TB	59	12.93	21,323	1	1.71%
Seagate	ST18000NM000J	18TB	60	3.27	5,820	2	12.54%
Toshiba	MD04ABA400V	4TB	99	67.29	36,234	0	0.00%
Toshiba	MG07ACA14TA	14TB	21,046	7.65	4,103,823	102	0.91%
Toshiba	MG07ACA14TEY	14TB	160	1.22	2,562	0	0.00%
Toshiba	MG08ACA16TEY	16TB	1,014	2.14	33,774	0	0.00%
WDC	WUH721414ALE6L4	14TB	6,002	1.68	229,861	1	0.16%
		Totals	**162,299**		**51,267,791**	**1,302**	**0.93%**

BACKBLAZE

Reliability in Layers

- Reliability-improving measures are taken in different layers
  - Each measure embraces failures in the underlying layer(s)

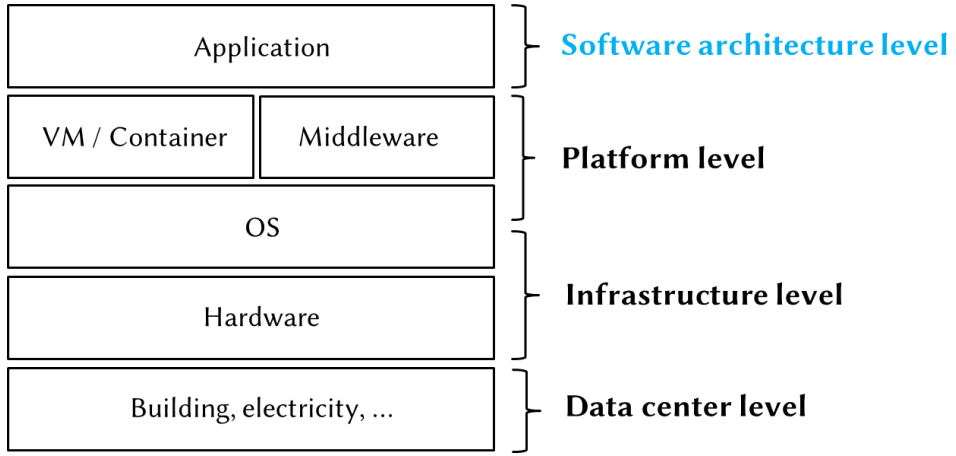
- We will walk through the layers to see how they embrace failures
- **Note**: Bodies of work exist for each layer, but we can only skim representative techniques due to time constraints



Reliability in Software Architecture Level

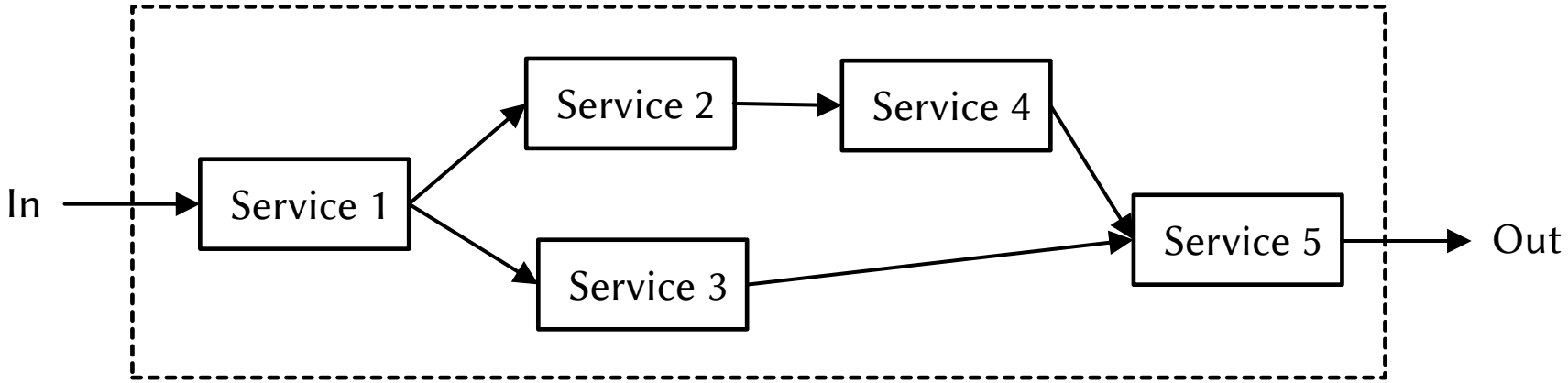
- Software architecture
  - Principles and best practices on how to structure complex software
  - Purposes: Easy maintainability, easy understandability, …
  - Examples: Object-oriented, MVC, …
  - Refer to Aoki-sensei's lecture about MVC

- Reliable software architecture
  - A software architecture that aims to be highly reliable
  - Prominent example: microservice architecture



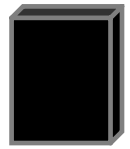
Microservice Architecture: Overview

- Constructing a (web) application as a swarm of micro services
  - Each service implements a single functionality
  - Each service runs in a separate process / container
- Communications among services through loosely coupling mechanisms
  - HTTP + REST APIs, gRPC, ...

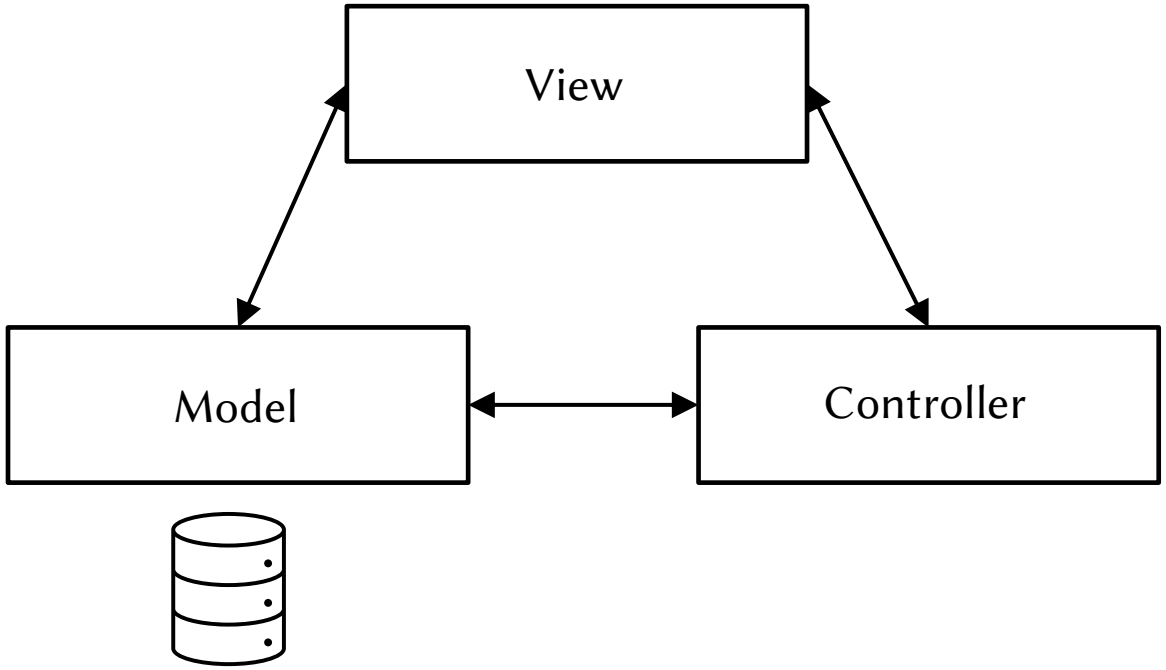


Straight-forward Implementation of MVC Architecture

- One “monolithic” process for each component
  - Runs in a single process, although modularized as functions and classes (of course)
  - Functionalities invoke each other using normal function calls



Monolith
(一枚岩)



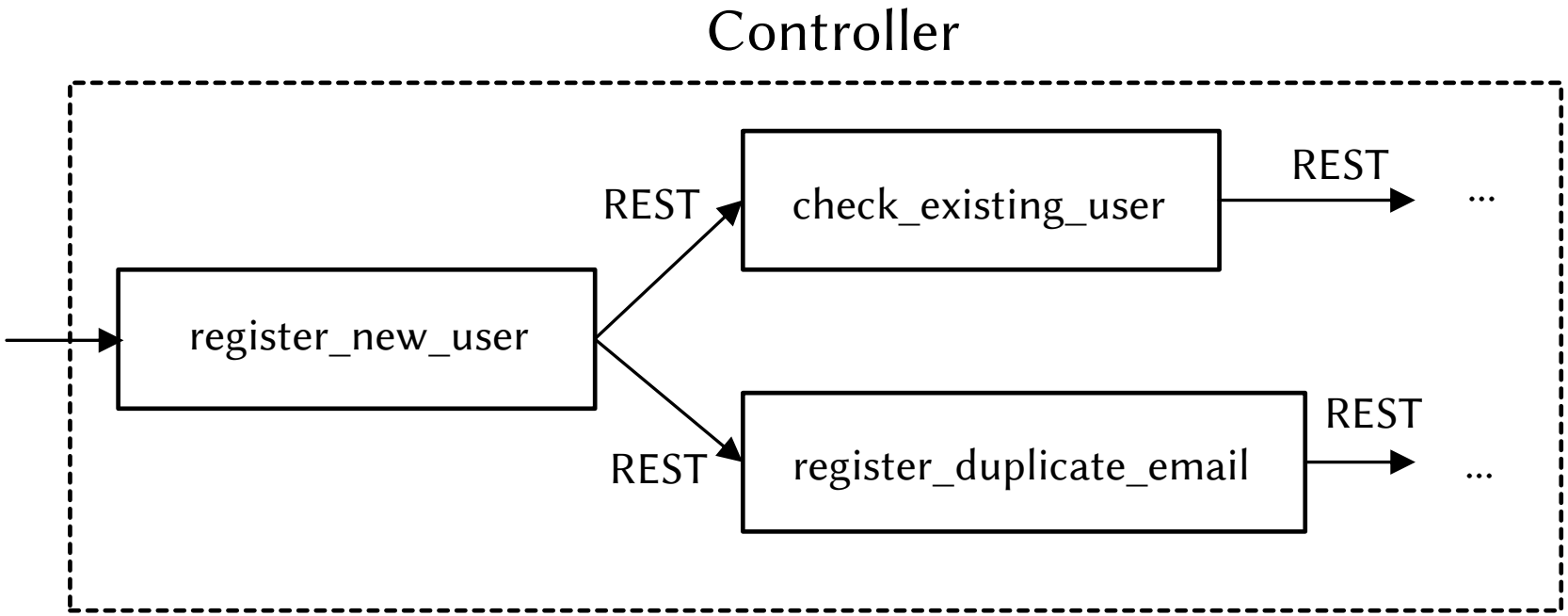

```
def check_existing_user(name):
  ....

def check_duplicate_email(email):
  ....

def register_new_user(name, email, password):
  ....
  if check_existing_user(name):
    ....

  if check_duplicate_email(email):
    ....
```

function calls

Controller.py

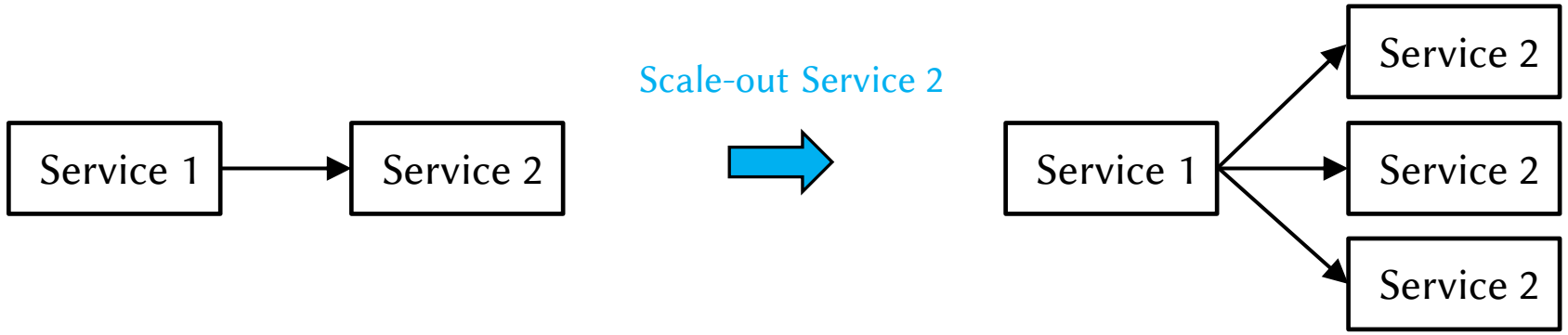
Microservice Implementation of MVC Architecture

- Each module is divided into micro services
  - Each module (service) runs in a separate process
  - Modules communicate with REST / http or alike even between the ones inside the same component (e.g., controller)



Microservices Offer Higher Reliability (1/2)

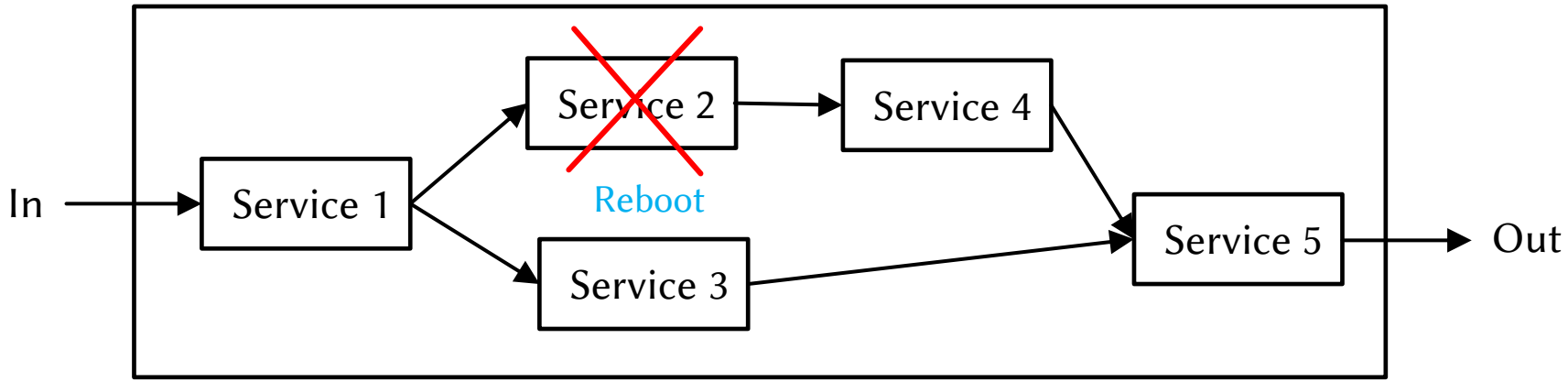
- Each service can be independently scaled
  - Ex: Service 1 relies on Service 2 and the latter has turned out to be more compute-intensive
  - Increase # of containers for Service 2 while that of Service 1 can remain the same



- This is trivial because…
  - Each service is already a separate process or a container
  - Services communicate with REST APIs and only loosely coupled

Microservices Offer Higher Reliability (2/2)

- Each service can be independently rebooted
  - Ex: Service 1 relies on Service 2 and the container running Service 2 fails somehow
  - Reboot only Service 2 while other parts are continuously working



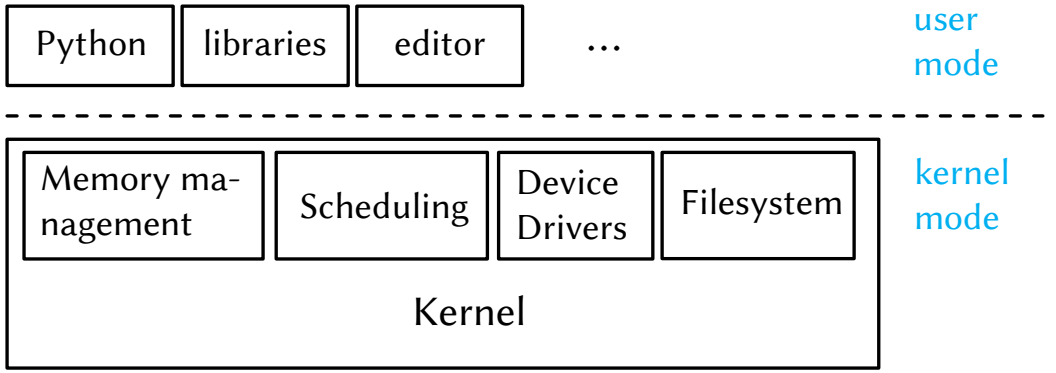
- This is also trivial because…
  - Rebooting Service 2 is just a lunch of a new process or a container
  - Other parts (e.g., Service 1 → Service 3 → Service 5) are independent from Service 2 by design

Adoptions of Microservice Architecture

- Netflix
  - https://www.youtube.com/watch?v=DvLvHnHNT2w English
- Uber
  - https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a English
- Mercari
  - https://www.publickey1.jp/blog/18/mercari_tech_conf_2018.html Japanese
- Cookpad
  - https://techlife.cookpad.com/entry/2016/03/16/100043 Japanese
- Notes
  - No enterprise reveals what kind of services they deploy in concrete (probably top-secret)
  - The articles focus mostly on deployment and (organization) management aspects

[Aside] Microservice-ish Architecture in OS Design (1/2)

- Monolithic kernel: a straight-forward design
  - Everything runs as a single binary in kernel mode
  - Adopted both by Linux and Windows (*)

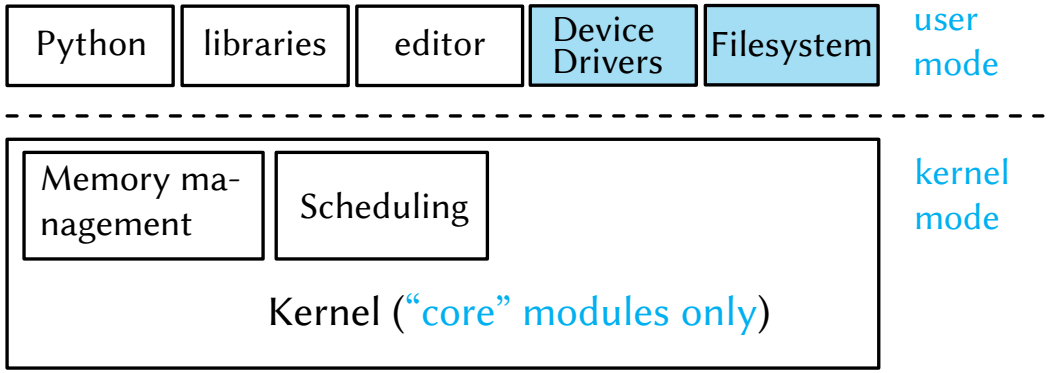


- Pros
  - Communication between kernel functionalities are easy and fast (just normal function calls)
- Cons: Low Reliability
  - Whole kernel hangs up if a device driver dies
  - Whole kernel could be compromised when a device driver is malicious

(*) Windows is also based on the monolithic kernel architecture, despite many people's belief (see *Windows Internals*, Microsoft Press)

[Aside] Microservice-ish Architecture in OS Design (2/2)

- Microkernel: a more reliable design
  - Only "core" functions run inside the kernel
  - Everything else (e.g., device drivers) run in userland

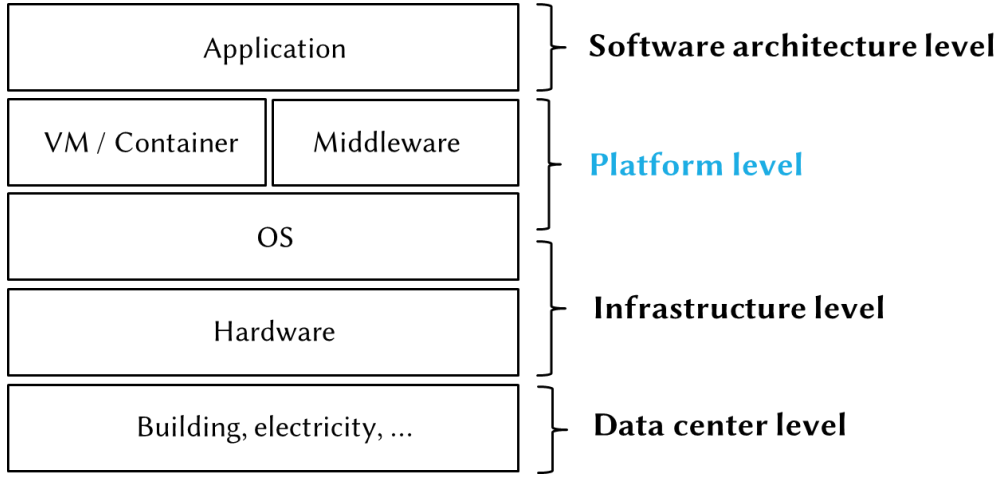


- Pros
  - OS functionalities can be independently rebooted when crash or compromised
- Cons
  - Communication between OS functionalities have larger overhead
  - Transitions between user and kernel modes requires using exceptions and are relatively slow

5 mins break

Reliability in Platform Level

- Platform
  - Software systems on which applications rely on
  - Examples: Middleware, DBMS, operating system



- Reliable platform
  - Keeps applications running even when underlying components (e.g., machines) fail

Example: Borg

- A reliable platform that runs (almost) everything in Google
  - Includes both production jobs and internal jobs
  - Includes both latency- and throughput-sensitive jobs

Kubernetes (or k8s), an OSS version of Borg

- Achieves high reliability by:
  - Utilizing the Paxos algorithm to avoid single point of failures
  - Automatically re-executing failed (or intentionally killed) processes
  - Spreading processes of the same job to multiple failure domains
    - More in later slides about failure domains

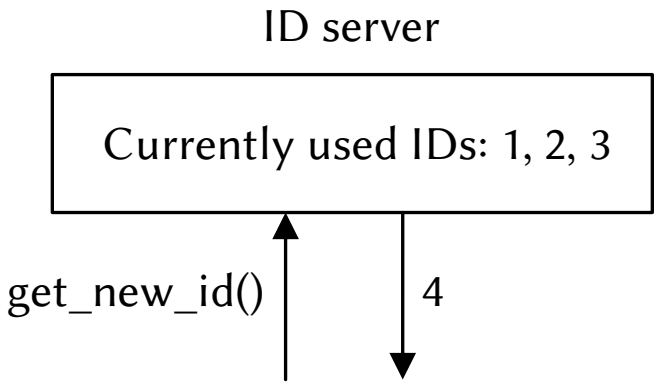
Single Point of Failure (SPOF, 単一障害点)

- A component that makes the whole system stop if fails
  - i.e., a component on which everything relies either directly or indirectly
  - Applicable both to software and hardware levels
  - Example: if the UTokyo account authentication system dies, we cannot login to any service (Zoom, UTAS, LMS, UTokyo Wifi, ..,) even if they are alive

- Getting rid of SPOFs is difficult
  - A component that knows and controls everything typically exists

Paxos Algorithm (1/2)

- A reliable consensus making algorithm
  - Decides a specific value among all participants with guaranteed extent of fault tolerance
  - Examples of failures: messages might not arrive, participants might not respond, two "leader" might suggest different values
- Example: assigning a new ID to a query in a web application

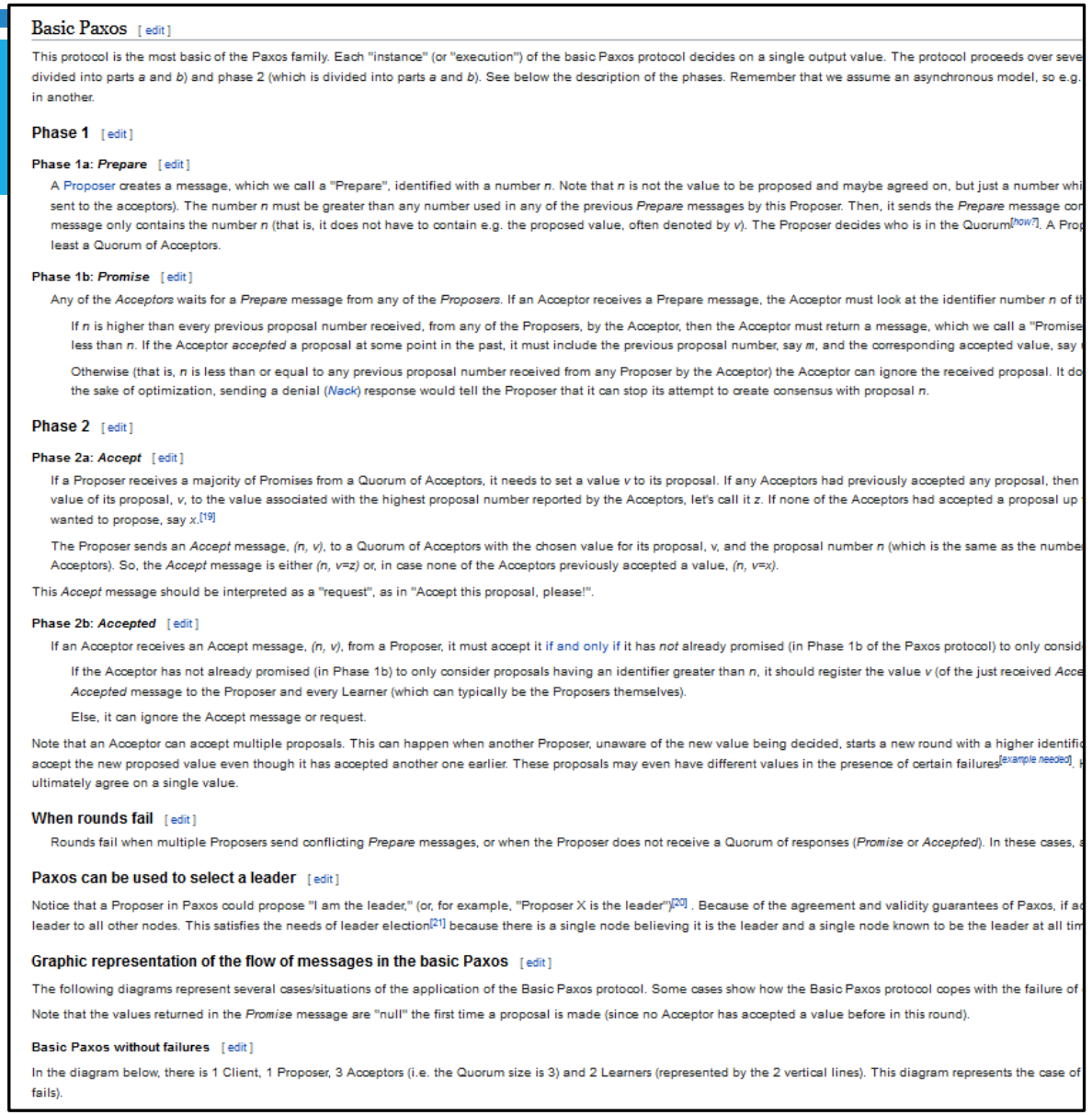


Traditional system (left):
- ID server knows IDs already assigned
- ID server gives a new ID based on this information
- ID server is a SPOF

System leveraging Paxos:
- Multiple servers make a consensus on the next ID to assign
- Can keep working under failures

Paxos Algorithm (2/2)

- How does Paxos work?
  - No time to cover, unfortunately…
- Famous to be moderately complex
  - The English Wikipedia article has almost 64 kilo bytes (consists only of plain texts)
  - A presentation (*) from a PFI (**) researcher has 64 pages (longer than this presentation!)



Basic Paxos [edit]

This protocol is the most basic of the Paxos family. Each "instance" (or "execution") of the basic Paxos protocol decides on a single output value. The protocol proceeds over seve
divided into parts *a* and *b*) and phase 2 (which is divided into parts *a* and *b*). See below the description of the phases. Remember that we assume an asynchronous model, so e.g.
in another.

Phase 1 [edit]

Phase 1a: *Prepare* [edit]

A Proposer creates a message, which we call a "Prepare", identified with a number *n*. Note that *n* is not the value to be proposed and maybe agreed on, but just a number whi
sent to the acceptors). The number *n* must be greater than any number used in any of the previous *Prepare* messages by this Proposer. Then, it sends the *Prepare* message cor
message only contains the number *n* (that is, it does not have to contain e.g. the proposed value, often denoted by *v*). The Proposer decides who is in the Quorum[how?]. A Prop
least a Quorum of Acceptors.

Phase 1b: *Promise* [edit]

Any of the *Acceptors* waits for a *Prepare* message from any of the *Proposers*. If an Acceptor receives a Prepare message, the Acceptor must look at the identifier number *n* of t

If *n* is higher than every previous proposal number received, from any of the Proposers, by the Acceptor, then the Acceptor must return a message, which we call a "Promise
less than *n*. If the Acceptor *accepted* a proposal at some point in the past, it must include the previous proposal number, say *m*, and the corresponding accepted value, say

Otherwise (that is, *n* is less than or equal to any previous proposal number received from any Proposer by the Acceptor) the Acceptor can ignore the received proposal. It do
the sake of optimization, sending a denial (*Nack*) response would tell the Proposer that it can stop its attempt to create consensus with proposal *n*.

Phase 2 [edit]

Phase 2a: *Accept* [edit]

If a Proposer receives a majority of Promises from a Quorum of Acceptors, it needs to set a value *v* to its proposal. If any Acceptors had previously accepted any proposal, then
value of its proposal, *v*, to the value associated with the highest proposal number reported by the Acceptors, let's call it *z*. If none of the Acceptors had accepted a proposal up
wanted to propose, say *x*.[19]

The Proposer sends an *Accept* message, *(n, v)*, to a Quorum of Acceptors with the chosen value for its proposal, *v*, and the proposal number *n* (which is the same as the numbe
Acceptors). So, the *Accept* message is either *(n, v=z)* or, in case none of the Acceptors previously accepted a value, *(n, v=x)*.

This *Accept* message should be interpreted as a "request", as in "Accept this proposal, please!".

Phase 2b: *Accepted* [edit]

If an Acceptor receives an Accept message, *(n, v)*, from a Proposer, it must accept it if and only if it has *not* already promised (in Phase 1b of the Paxos protocol) to only consid

If the Acceptor has not already promised (in Phase 1b) to only consider proposals having an identifier greater than *n*, it should register the value *v* (of the just received *Acce
Accepted message to the Proposer and every Learner (which can typically be the Proposers themselves).

Else, it can ignore the Accept message or request.

Note that an Acceptor can accept multiple proposals. This can happen when another Proposer, unaware of the new value being decided, starts a new round with a higher identifi
accept the new proposed value even though it has accepted another one earlier. These proposals may even have different values in the presence of certain failures[example needed]. H
ultimately agree on a single value.

When rounds fail [edit]

Rounds fail when multiple Proposers send conflicting *Prepare* messages, or when the Proposer does not receive a Quorum of responses (*Promise* or *Accepted*). In these cases,

Paxos can be used to select a leader [edit]

Notice that a Proposer in Paxos could propose "I am the leader," (or, for example, "Proposer X is the leader")[20] . Because of the agreement and validity guarantees of Paxos, if a
leader to all other nodes. This satisfies the needs of leader election[21] because there is a single node believing it is the leader and a single node known to be the leader at all tim

Graphic representation of the flow of messages in the basic Paxos [edit]

The following diagrams represent several cases/situations of the application of the Basic Paxos protocol. Some cases show how the Basic Paxos protocol copes with the failure of

Note that the values returned in the *Promise* message are "null" the first time a proposal is made (since no Acceptor has accepted a value before in this round).

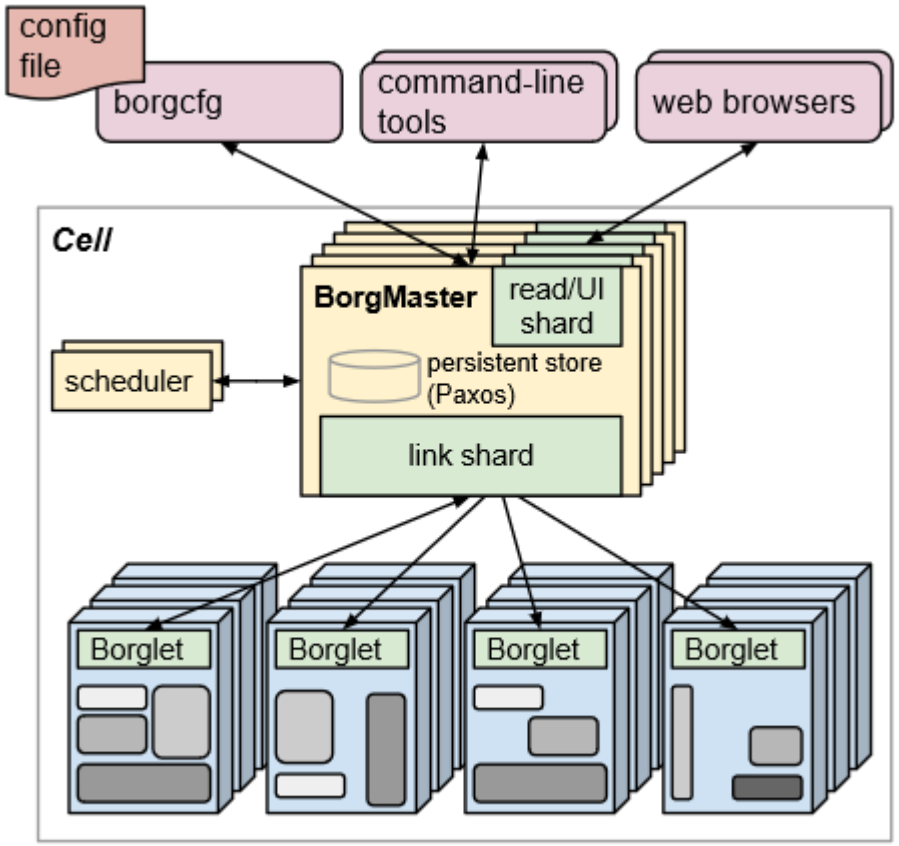
Basic Paxos without failures [edit]

In the diagram below, there is 1 Client, 1 Proposer, 3 Acceptors (i.e. the Quorum size is 3) and 2 Learners (represented by the 2 vertical lines). This diagram represents the case of
fails).

https://en.wikipedia.org/wiki/Paxos_(computer_science)

(*) https://www.slideshare.net/pfi/paxos-13615514
(**) Currently known as PFN

Borg Architecture

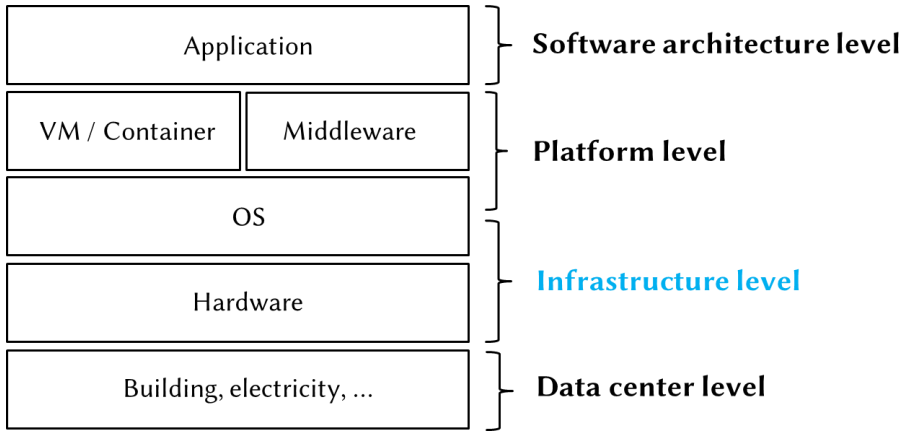


Cited from A. Verma *et al.*, "Large-scale cluster management at Google with Borg", *EuroSys'15*

- A *cell* consists of a bunch of machines
  - A cell has one "logical" BorgMaster
  - User submits jobs to BorgMaster
- BorgMaster is duplicated to avoid SPOF
  - Uses Paxos to elect the main one among the duplicated ones
- Each machine has a Borglet
  - Borglet is controlled by BorgMaster
- BorgMaster recreates processes running in a machine when a Borglet does not respond

Reliability in Infrastructure Level

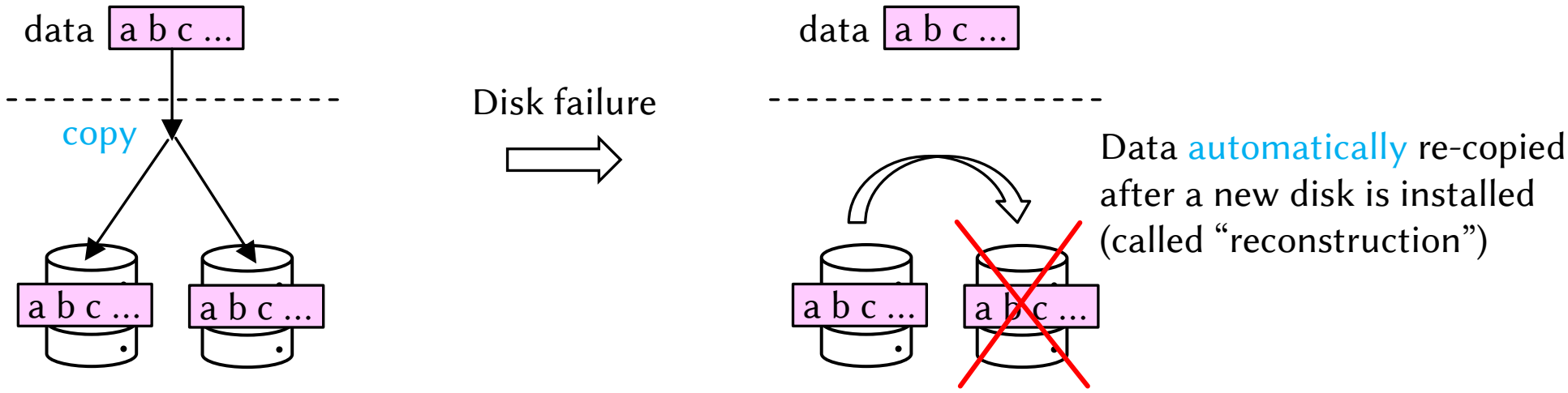
- Infrastructure
  - Fundamental computing resources that applications and platforms rely on
  - CPU, storage, network, memory, (operating system)



- Reliable infrastructure
  - Keeps platforms and applications running even when components (e.g., HDDs) of the underlying hardware fail

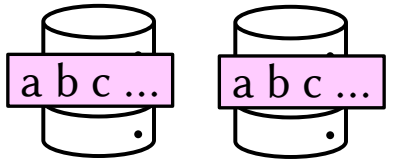
RAID (Redundant Arrays of Inexpensive Disks)

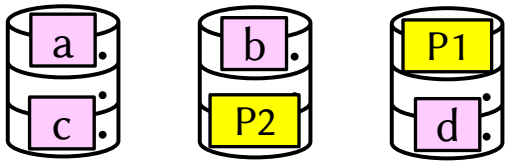
- Improve reliability of storage systems by combining multiple disks
  - Especially, multiple inexpensive disks
  - Software engineer's way of improving storage reliability
- Store multiple data copies on different disks
  - Exposed as a single volume from user' perspective
  - Automatically "reconstructed" when one (or a few) of the disks fail



RAID Types

- RAID 1 (a.k.a. Mirroring)
  - Copy an entire disk to a spare one
  - Tolerate one disk failure out of two (or N failures out of 2N)
- RAID 5
  - Parallel and faster read than RAID1
  - Chop data into and store chunks to different disks
  - Add parities to tolerate disk failures
- RAID 6
  - Add another parity (cf. double parity) to improve reliability
  - (a, b, P1, P2) are paired: Tolerate two disk failures

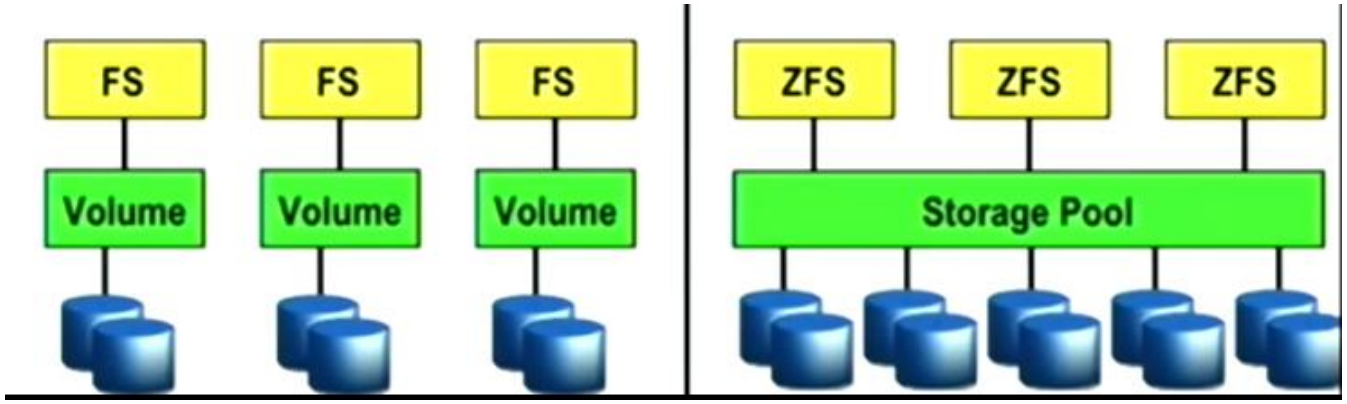




(a, b, P1) are paired: Any one of them can be recovered from the other two
→ Single disk redundancy in this case

ZFS

- RAID capacity is limited by the smallest disk of the array
  - RAID is disk-based by design
  - Example: RAID1 with a 1TB and a 2TB disk → Usable capacity is 1 TB
- ZFS: Building a reliable filesystem atop a storage pool



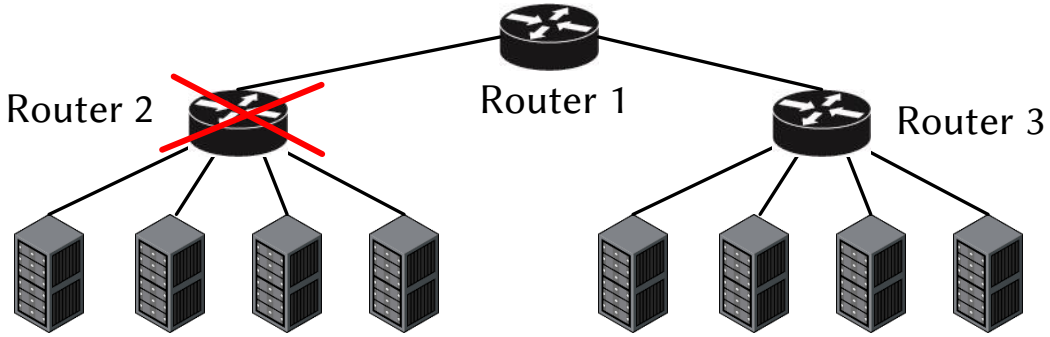
Cited from a newbie guide of ZFS: https://www.youtube.com/watch?v=MsY-BafQgj4

Left: Traditional filesystems
- Filesystem is created in a volume (e.g., a disk, a physical partition)

Right: ZFS
- Filesystem is created in a storage pool
- Storage pool consists of multiple disks of different capacity
- It is exposed as a one virtual volume
- It is made reliable by a RAID-like mechanism

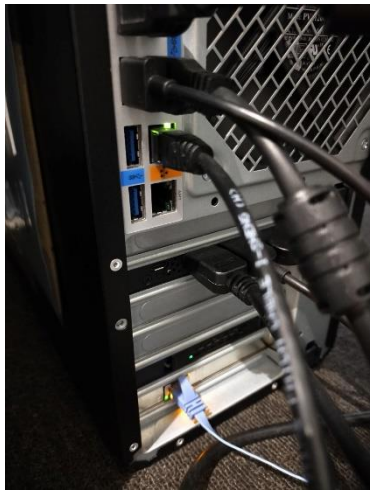
Reliable Networking

- Network failures
  - A single machine becomes unreachable (e.g., a network interface of a machine fails)
  - A single route becomes unusable (e.g., a network router fails)



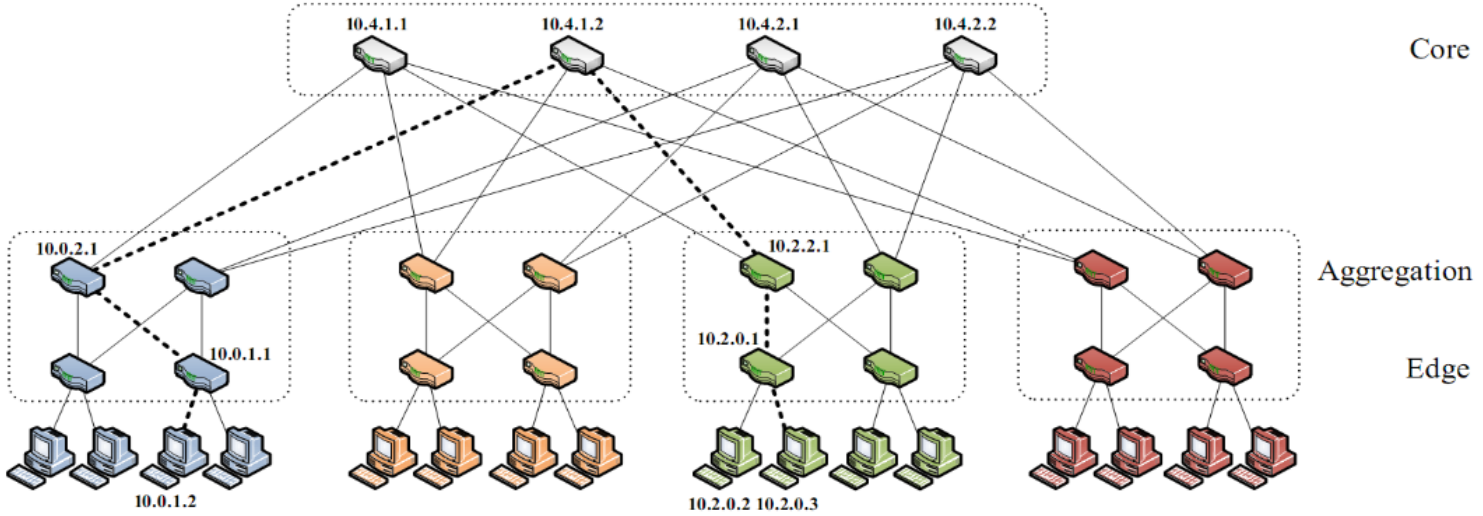
Every machine under router 2 becomes unreachable if router 2 fails

- Common measures against network failures
  - Having multiple network interfaces in a machine
  - Having multiple routes between any two machines



Fat Tree Topology

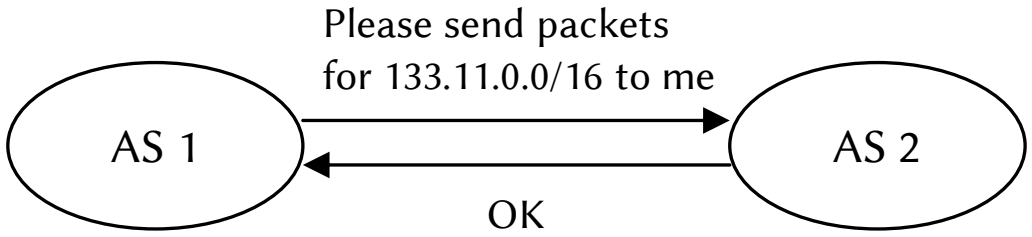
- Network topology
  - Connectivity between network components (servers, routers, switches)
  - Goals: fault tolerance, effective bandwidth, efficiency
- Fat tree
  - Multiple paths exist between any two components
  - Achieved by having multiple switches to aggregate lower layers



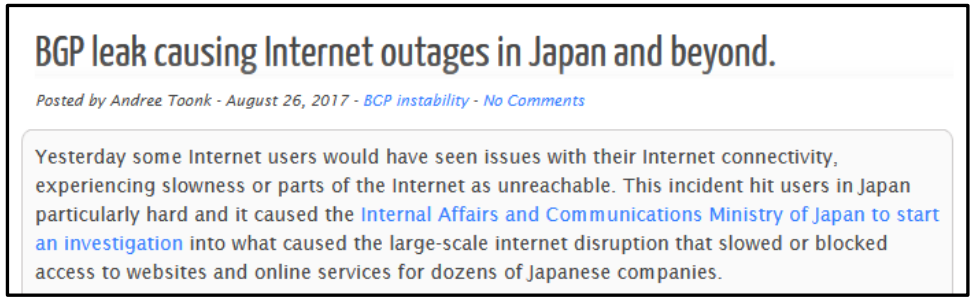
https://www.cs.cornell.edu/courses/cs5413/2014fa/lectures/08-fattree.pdf

[Aside] Is the Internet Reliable?

- BGP (Border Gateway Protocol)
  - Exchange route info between ASes (Autonomous Systems; internet service providers, cloud providers, etc.)



- BGP misconfiguration makes (a part of) the Internet go down
  - Packets "black-holed" to a wrong AS
  - Little adoption of authentication mechanisms (ongoing area)
- Actual outage due to this occurs time to time
  - https://bgpmon.net/bgp-leak-causing-internet-outages-in-japan-and-beyond/



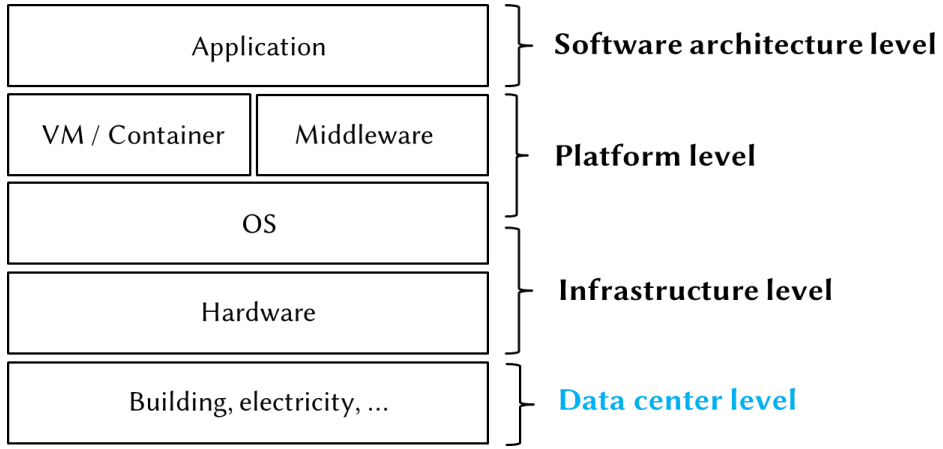
BGP leak causing Internet outages in Japan and beyond.

Posted by Andree Toonk - August 26, 2017 - BGP instability - No Comments

Yesterday some Internet users would have seen issues with their Internet connectivity, experiencing slowness or parts of the Internet as unreachable. This incident hit users in Japan particularly hard and it caused the Internal Affairs and Communications Ministry of Japan to start an investigation into what caused the large-scale internet disruption that slowed or blocked access to websites and online services for dozens of Japanese companies.

Datacenter Level Reliability

- Even infrastructure-level reliability is not enough
  - What if a whole building runs out of power supply?
  - What if a natural disaster hits a datacenter?

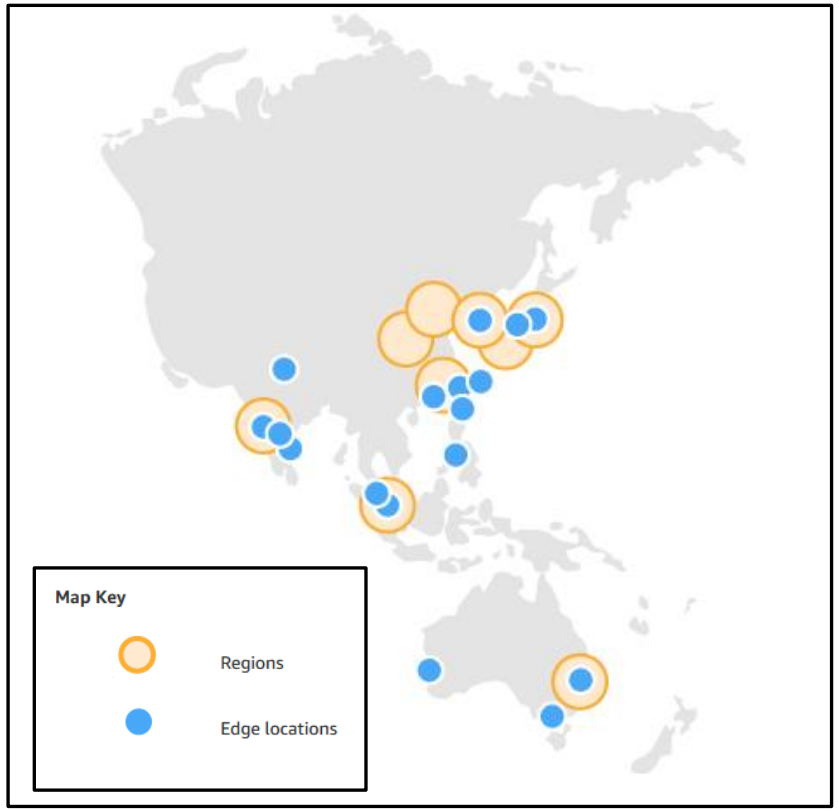


- Datacenter level reliability
  - Base idea: Preparing multiple failure domains
  - Smaller level: Multiple cluster of machines that rely on independent sets of physical components (e.g., power supply)
  - Larger level: Multiple datacenters that are geographically apart from each other

AWS Example: Regions and Availability Zones

- **Regions** (larger level)
  - Clusters of datacenters that are physically apart
  - Examples: Tokyo, Osaka (from 2021), Beijing, Hongkong, …
  - Even if a big earthquake hits Tokyo, the Osaka one would keep working

- **Availability zones** (smaller level, kind of)
  - A region includes multiple availability zones
  - An availability zone consists of multiple datacenters within 100 km (middle-sized level?)



https://aws.amazon.com/about-aws/global-infrastructure/regions_az/