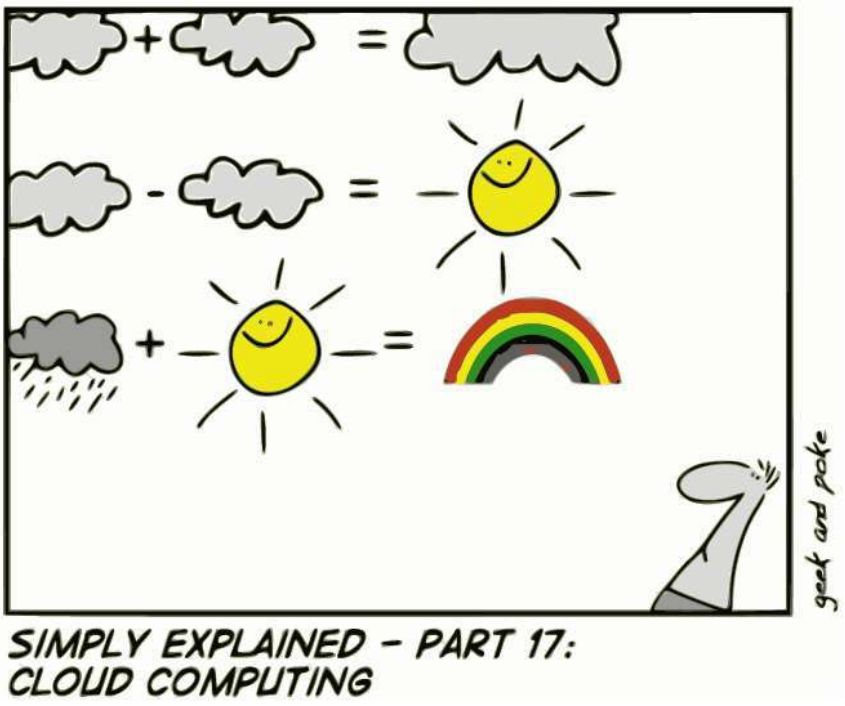


CLOUD SYSTEMS

Cloud Overview

Lecture #1

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

Overview

- What is cloud (computing)?
- How did today's cloud evolve?
- From enterprise computing to cloud computing
- A definition – summary

- https://www.youtube.com/watch?v=PPlgvSL00fA

What is cloud computing?

- Computing happens somewhere else, not on your PC or mobile device

- Several definitions exists
- Not all are helpful
- Good definitions are extensive

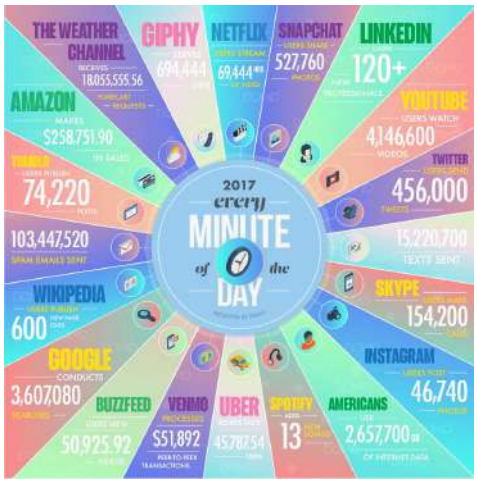
Cloud underpins modern computing

- Physical
  - The cloud is a global deployment of massive data centers connected by ultra-fast networking, designed for scalability and robustness
- Logical
  - A collection of tools and platforms that scale amazingly well
  - The platforms matter most; as a developer, they allow you to extend/customize them to create your application as a "personality" over their capabilities
- Conceptual
  - A set of scalable ideas, concepts and design strategies

Who invented cloud?

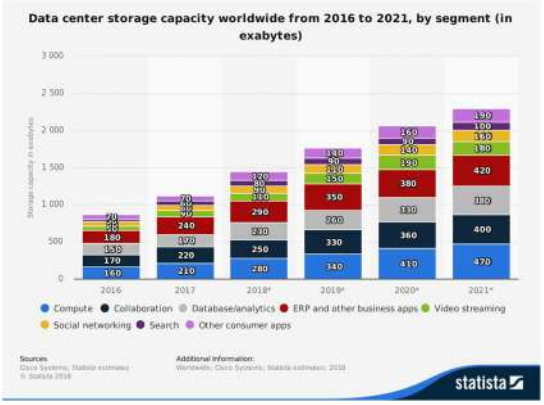
- If we really want to find one…
- Jeff Dean and Sanjay Ghemawat from Google
- Known for
  - Map Reduce
  - Bigtable
  - Google File System
  - …
- Famous for simple and robust ways to scale things up (and writing about them) – **This was the key to the modern cloud**

Everything is big in cloud



Data in cloud

- 1 Exabyte of data is 1,073,741,824 GB (Your hard disk probably holds 1 TB, but is way too slow by data-center standards)
- The Internet has about 2 B websites, and of these, 644 M have "active content"
- … and all of this is "pre Internet of Things"



IoT at the edge: The big new thing

- IoT is a huge opportunity and growth area
- How cloud options work and are used there?
- Focus on live interactions with the outside world
- This used to be dominated by web interfaces, but increasingly also includes 5G mobility and a wide variety of IoT sensors and actuators – the so-called **IoT cloud**

Concept for IoT cloud: Digital twin

- We often like to think of the cloud as having a software "image" that matches with IoT in the physical world
- The IoT devices are physical and live "outside" the cloud, but for each device we have a "digital twin" that lives inside the cloud, and is a kind of proxy
- Actions on the twin are translated to reading the sensor or telling the actuator to do something

How did today's cloud evolve?

- Prior to 2005, there were "data centers designed for high availability"
  - Amazon had especially large ones, to serve its web requests (This is all before the AWS cloud model)
  - The real goal was just to support online shopping
- Their system was not very reliable, and the core problem was scaling
  - Like a theoretical complexity growth issue
  - Amazon's computers were overloaded and often crashed

Wasn't Google first?

- Google was still building their first scalable infrastructure in this period
- Because Amazon ran into scaling issues first, Google (a bit later) managed to avoid them
  - In some sense, Amazon dealt with these issues "in real time"
  - Google had a chance to build a second system by learning from Amazon's mistakes and approaches

Yahoo experiment

- One really obvious symptom of overload is that everything got slow
- Meanwhile, people realized that speed is the absolute must-have requirement
- In the 2005 there was an experiment about ad-click-through
  - Customers who saw web page rendering faster than 100 ms clicked ads
  - For every 100 ms delay, **click-through rates noticeably dropped**
- Speed at scale determines revenue, and revenue shapes technology
  - Speed up the cloud!

More Yahoo findings and Amazon decision

- Rending the ads first did not help – in fact it hurts
- Customers wanted to see the "real content" first
- Rendering the ads after the content hurts too
- To get the best click-through rates, render your pages (ads included) fast

- Amazon company was told to focus on ensuring that every Amazon product page would render with minimal delay
  - Unfortunately, as more and more customers turned up, Amazon's web pages **slowed down**
  - This is called a **"crisis of the commons"** situation

Where are the commons in cloud?

- In the cloud we need to think about all the internal databases and services "shared" by lots and lots of microservice instances
- If we take the advice to "make everything as fast as possible", all those millions of first-tier microservices will be greedy
- But what works best for one instance, all by itself, might overload the shared services when the same code runs side by side with huge numbers of other instances ("when we run at scale")

Reinventing data center computing

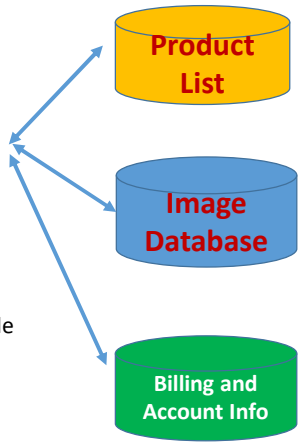
- In 2006 Amazon reorganized their whole approach
  - Requests arrived at a "first tier" of very lightweight servers
  - These dispatched work requests on a message bus or queue
  - The requests were selected by "microservices" running in elastic pools
  - One web request might involve tens or hundreds of microservices!
- They also began to guess at your next action and precompute what they would probably need to answer your next query or link click

Old approach (2005)

Computers were mostly desktops

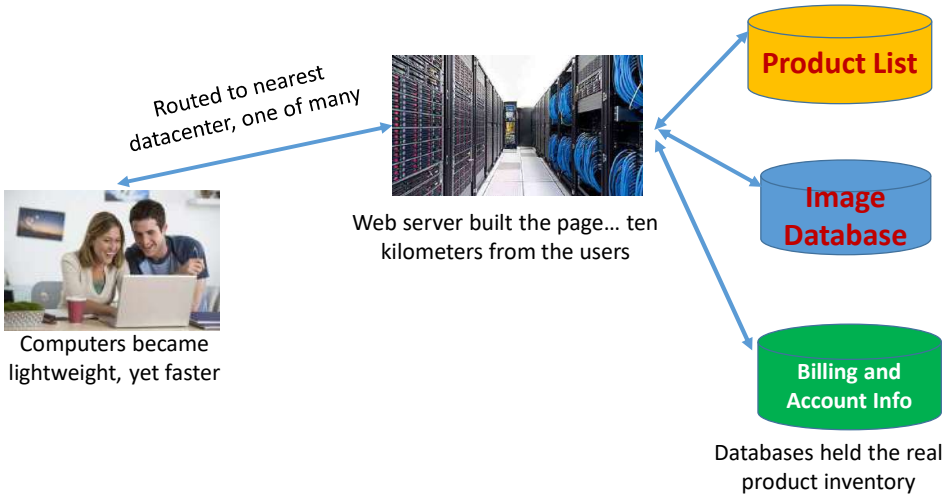
Internet routing was pretty static, except for load balancing

Web server built the page… in Seattle

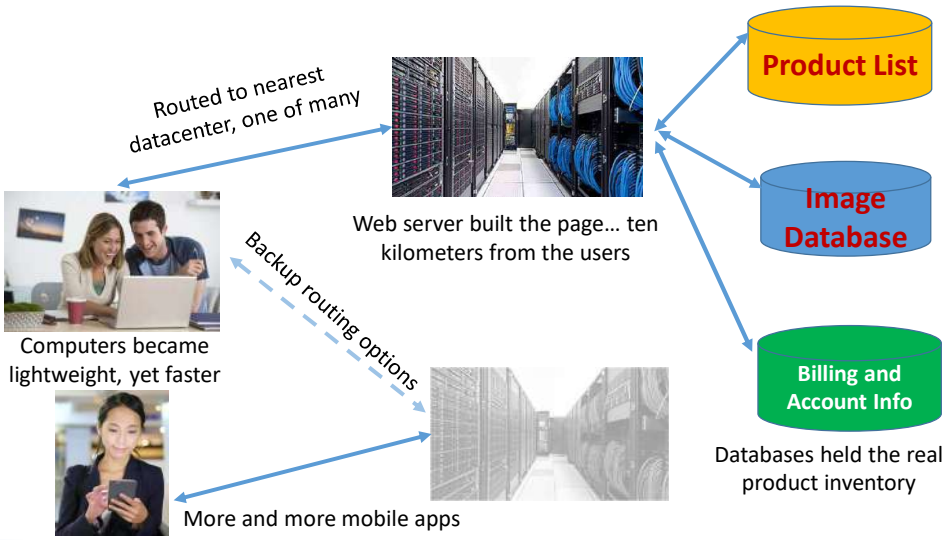


Databases held the real product inventory

New approach (2008)



New approach (2008)

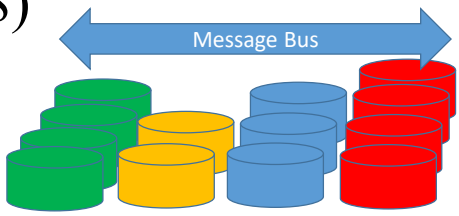


New approach (2008)

Desktops with snappier response

Routed to nearest datacenter, one of many

Web server becomes simpler and does less of the real work



Racks of highly parallel workers do much of the data fetching and processing, ideally ahead of need… The old databases are split into smaller and highly parallel services.

More and more mobile apps

Georeplication

Tier one / tier two

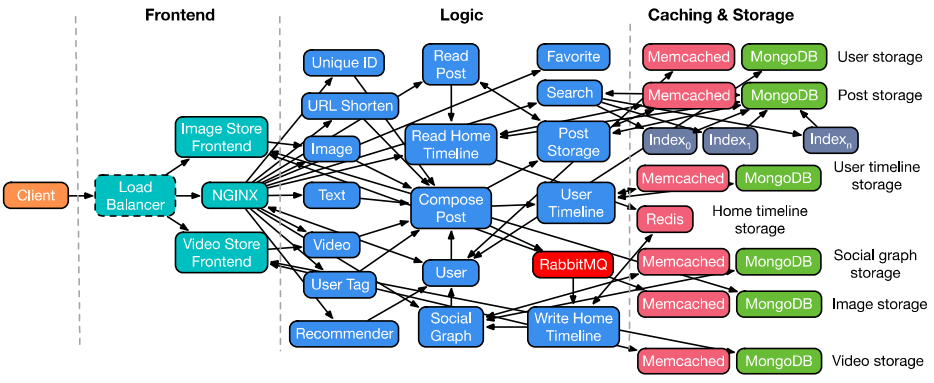
- We often talk about the cloud as a "multi-tier" environment
- Tier one: programs that generate the web page you see
- Tier two: services that support tier one (e.g. data processing, cache)

Today's cloud

- Tier one runs on very lightweight servers
  - They use very small amounts of computer memory
  - They do not need a lot of compute power either
  - They have limited needs for storage, or network I/O
- Tier two microservices specialize in various aspects of the content delivered to the end-user
  - They may run on somewhat "stronger" computers

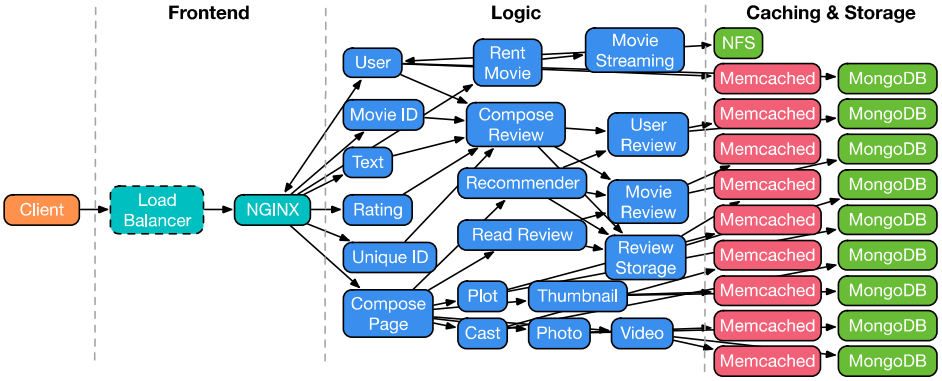
End-to-end microservices

- Social network



End-to-end microservices

- Media service



Each microservice is a parallel pool

- Every one of those little nodes is itself a small elastic pool of processes
- A microservice is a kind of program designed so that the data center can run one instance (or many instances), "elastically", to deal with dynamically varying demand
- The idea here is that any instance can handle any request equally well, so there is no need for very careful "routing" of specific requests to specific instances
- This lets the data center adapt to changing loads easily!

Pools are managed automatically

- In Azure, e.g., there is a tool called the "App Service"
- The App Service manages a big collection of compute resources in the cloud
  - Developers can install own services in it (as "containers")
  - Configuration files tell it when to launch them, automatically
- Among the features is a way for it to watch the queue of requests and automatically add instances or shut instances down to match loads

What does it mean "add instances"?

- For some applications (ones with NUMA threading for parallelism) we add instances by launching new threads on additional cores
- For others, we literally run two or more identical copies of the same program, on different computers
  - They use a "load balancer" to send requests to the least loaded instances
- And we can even combine these models…

- This is really just one of a few ways to get parallelism
  - **We need to understand why the cloud favors the approach**

How do cloud services scale?

- We have been acting as if each microservice is a set of "processes" but ignoring how those processes were built
- In fact they will use parallel programming of some form because modern computers have NUMA architectures
- How do cloud developers think about this form of parallelism?
- Not every way of scaling is equally effective!

Example: Cloud hosted music service

- Which parallelism is better?
- One multithreaded server, per node? (as we built for DFS in DS course)

- Basically, we have four options
  1. Keep the server busy by running one multithreaded application on it
  2. Keep it busy by running many unthreaded versions of the application as virtual machines, sharing the hardware
  3. Keep it busy by running many side by side processes, but do not virtualize
  4. Keep it busy by running many side by side processes using containers

Result…

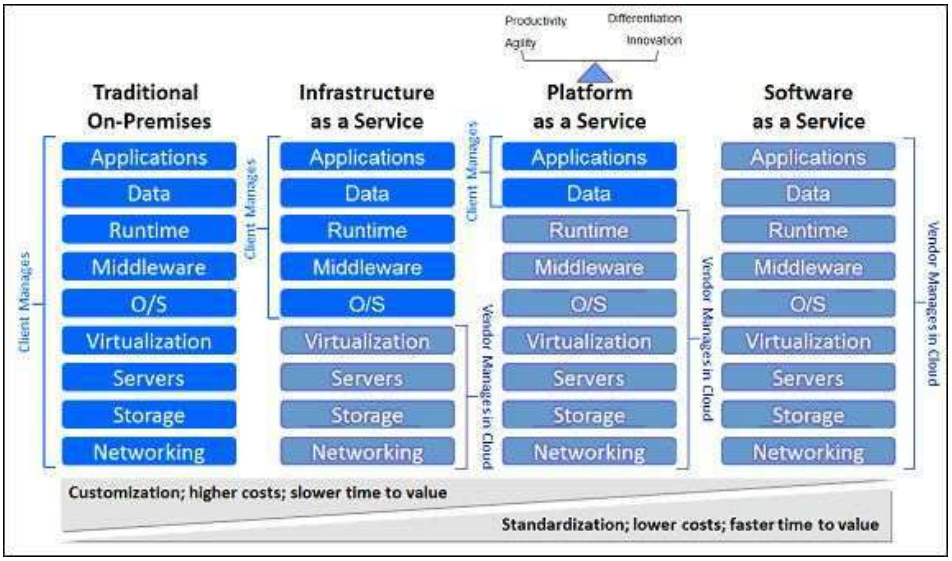
- Best is "container virtualization" with one server process dedicated to each distinct user
- A single cloud server might host hundreds of these servers
- They are easy to build: you create one music player, and then tell the cloud to run as many "instances" as required
- The solution today is a mix
  - Program in Linux but use "docker containers" in a framework called "Kubernetes"
  - Mostly code in C++ if your code is performance-intensive and Java/Python if not

Why favor container virtualization?

- Code is much easier to write
  - Most people can write a program to play music for a single client – this same insight applies to other programs, too
- Very easy for the cloud itself to manage
  - Containers are cheap to launch and also to halt, when the customer disconnects
- The approach also matches well with modern NUMA computer hardware
- In fact the way to approach this is to program in a way that matches best with the hardware
  - But it was hard to figure out what "best" should mean!
  - In 2006, when the cloud emerged, we did not know the best approach
  - Over time, everything had to evolve and be optimized for cloud uses

The cloud is evolving towards platforms

- Various XaaS acronyms
- IaaS – infrastructure as a service, basically it means "rent a machine, do what you like"
- PaaS – platform as a service (but really it means "customizable and extensible platform") is the most successful cloud model today
- SaaS – software as a platform, where provider runs web applications and customers only need a browser



PaaS concept

- The vendor offers all the standard code, pre-built
  - They create the platform in ways that perform and scale really well
- The developer will just plug in a small function (sometimes called a "lambda") to do the work, like playing back the requested video

Enterprise computing

- Use of computers for data processing in large organizations
  - Information Systems (IS) or Information Technology (IT) in general
- Cloud computing revolutionizes enterprise computing after 50 years of its existence
- Key elements of cloud computing
  - Computing resources packed as a commodity and made available over internet
  - The ability for end-users to rapidly provision the resources they need
  - A pricing model that charges consumers only for what they actually use

Evolution: Pre-internet era

- 1960s Mainframes
  - IBM System/360 “mainframe” computer and its successors (today e.g. IBM zSeries)
  - Many design features of mainframe era are now hallmarks of cloud computing world: virtual machines, fault tolerance, non-relational databases, centralized computing itself
- 1980s Client-server architecture
  - Microprocessor revolution of the 80s
    - PCs as business desktop workstations and home computers
    - Unix operating system and C programming language
  - Move data processing from expensive mainframes to desktop workstations
  - Client-server: forms-based vs. fat-client (SAP/R3)
  - Corporate data became available on same desktops used for word processing and spreadsheet applications (office tools)

Evolution: Internet era

- Early 1990s Three-tier architectures
  - High-volume transaction processing, client-server architecture did not scale
  - 3-tier applications with thin-client
- Middle of 1990s Web-enabled applications
  - Internet as a platform, raise of web (HTTP, CGI)
  - Web-enabled legacy application – browser based interface, CGI
- End of 1990s Web applications
  - Execute application functionality inside web-server process
    - Birth of web application server
  - Multi-threaded execution environments (containers) embedded within web server
    - Servlet container
    - EJB container

Evolution: Cloud era

- Early 2000s Internet as service
  - High-performance web applications
  - Horizontal scaling
  - Data centers become large "IT plants" akin to complex nuclear power plants
- Middle 2000s Cloud computing
  - The complexity of these environments has been a driving factor for the large-scale interest in cloud computing architecture
  - The attendant complexities are managed in a scalable and largely automated manner
- End of 2000s SOA and integration
  - Web services, W3C defines a web service as interoperable machine-to-machine interaction over HTTP
  - Internet of services
    - Applications can call the web services published by other applications over the internet
  - Automated and dynamic integration

Definition

"*By using virtualized computing and storage resources and modern web technologies, Cloud Computing provides scalable, network-centric, abstracted IT infrastructures, platforms, and applications as on-demand services. These services are billed on a usage basis*."

- Baun et al.: *Cloud Computing: Web-basierte dynamische IT-Services*. Springer, 2011.

Definition

"***By using virtualized computing and storage resources and modern web technologies****, Cloud Computing provides scalable, network-centric, abstracted IT infrastructures, platforms, and applications as on-demand services. These services are billed on a usage basis.*"

- Fundamental technologies
  - **Virtualization** for shared an efficient resource utilization
  - **Web Services** (SOAP, REST) for communicating with services

Definition

"*By using virtualized computing and storage resources and modern web technologies, Cloud Computing provides* ***scalable, network-centric, abstracted IT infrastructures, platforms, and applications as on-demand services. These services are billed on a usage basis.***"

- Characteristics of cloud services
  - **Everything as a service** (XaaS)
  - **Scalable** – "elastic", i.e. automatic commission and decommission
  - **Network-centric** – services/resources are accessible over the internet
  - **Abstracted** – independent of the concrete hardware
  - **On-demand** – prompt request completion
  - **Pay as you go**

Cloud Computing – Services?

"*By using virtualized computing and storage resources and modern web technologies, Cloud Computing provides scalable, network-centric, abstracted* ***IT infrastructures, platforms, and applications*** *as on-demand* ***services****. These services are billed on a usage basis.*"

- Cloud computing is an umbrella term for different services
- What is a **service**?

IT Service

- Service in the information technology (IT) area
- Provided by a service provider for one or more customers
- Offered like a product
- Should be defined via a service level agreement (SLA)
- Provided by company's own department (**inhouse**) or by an external provider (**outsourcing**)
- How can cloud services be distinguished in an **organizational** way?

Organizational distinction of services

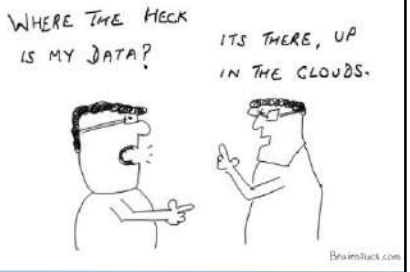
- **Public cloud**
  - Customer and provider do belong to different organizations – Outsourcing
  - No cost for purchasing, operate and maintain of own hardware
  - Resources ready for use immediately, and (almost) unlimited available
- **Private cloud**
  - Customer and provider do belong to the same organization
  - Costs are similar to a non-cloud-based architecture
- **Hybrid cloud**
  - Public and private clouds are used together
  - Application examples
    - Manage load peaks with public cloud services
    - Store backup data in public clouds

Functional distinction of services

- **Software as a Service (SaaS)**
  - Provider runs web applications
  - Customers only need a browser
- **Platform as a Service (PaaS)**
  - Provider run scalable runtime environment(s)
  - Customers run their own web applications in the infrastructure of the service provider
- **Infrastructure as a Service (IaaS)**
  - Provider runs physical servers
  - Customers run VMs with (almost) any operating systems and unmodified applications
    - Customers have administrator privileges in their VMs and define the firewall rules themselves

Aversion to cloud computing?

- Some reasons for aversion to cloud computing
  - Hardware is *sexy*
  - (As much as possible) own hardware is impressive on the open house day
  - Difficult to describe feeling of insecurity
    - Storing own data outside the home, generates an *uneasy* feeling
  - *On my hardware I am the boss*
  - Administrators love their hardware
    - Despite all the work and frustration
    - Stockholm syndrome?!
  - Loss of hardware = loss of power and influence?



What risks are real?

- What is the availability of the data and services?
- Can data loss happen?
- Is data in cloud services secure against unauthorized access?
- Can cloud services be used to cause damages?
- Are the cloud service providers trustworthy?

Risks of cloud computing

- Main differences between the cloud services
  - Functionality
  - Availability / quality
  - Price
  - Interface (often considered not important)
- Privacy and security
  - Solutions exist and not everything must be stored in the cloud
  - Parental computing in contrast to personal computing
- **Lock-in**
  - Services and tools of the service providers only implement their own API

Risk of lock-in

- If a customer decides to use a public cloud service, he also decides to use a specific interface
- Potential issue: **Lock-in**
  - A dependency between the user and the provider of the service exists
- Scenarios: Price increases, provider bankruptcy, change of service offering (functionality), ...
- A consequence of switching the provider is the **loss** of the infrastructure (**services**) and possibly even the **data**
  - Consequences for customers (especially companies) may be fatal
- If a customer uses a service for long term, he **invests** in this service
  - The own business model is focused on the service
  - Employees are trained
  - Services are refined

Impact of the lock-in for Dropbox

- Web service, started in 2007
- Provides a network file system for the synchronization of files between different computers and users
- Stores the users' files inside S3

Where does Dropbox store everyone's data?
Once a file is added to your Dropbox, the file is then synced to Dropbox's secure online servers. All files stored online by Dropbox are encrypted and kept securely on Amazon's Simple Storage Service (S3) in multiple data centers located across the United States.
Source: https://www.dropbox.com/help/7/en/

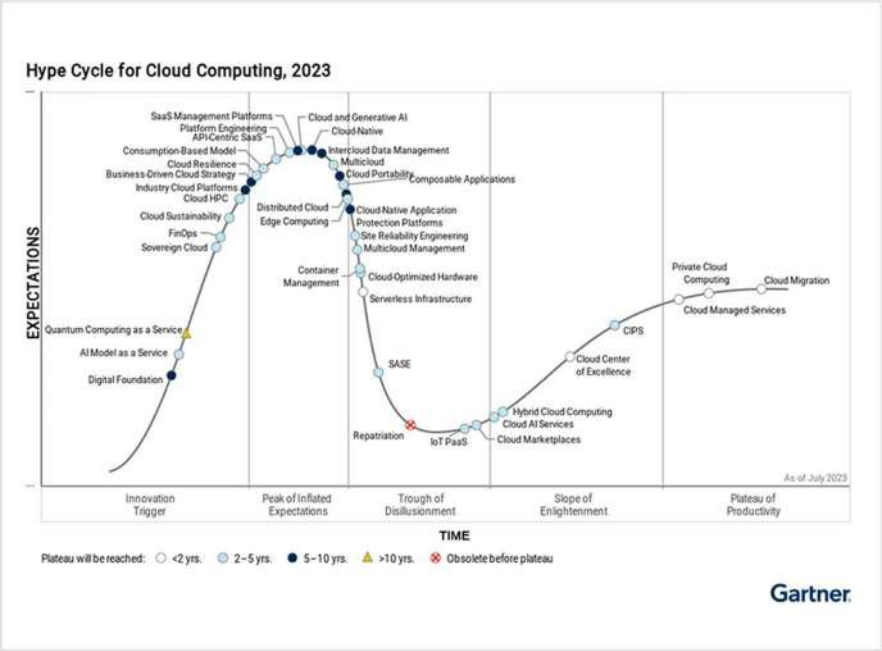
- Used business model: **Refine an existing cloud service**

- What would be the impact on Dropbox, if S3 would double the price or close down?
- What would be the impact on the customers of Dropbox and S3?
- Is there anything which can be done against the risk of a lock-in?

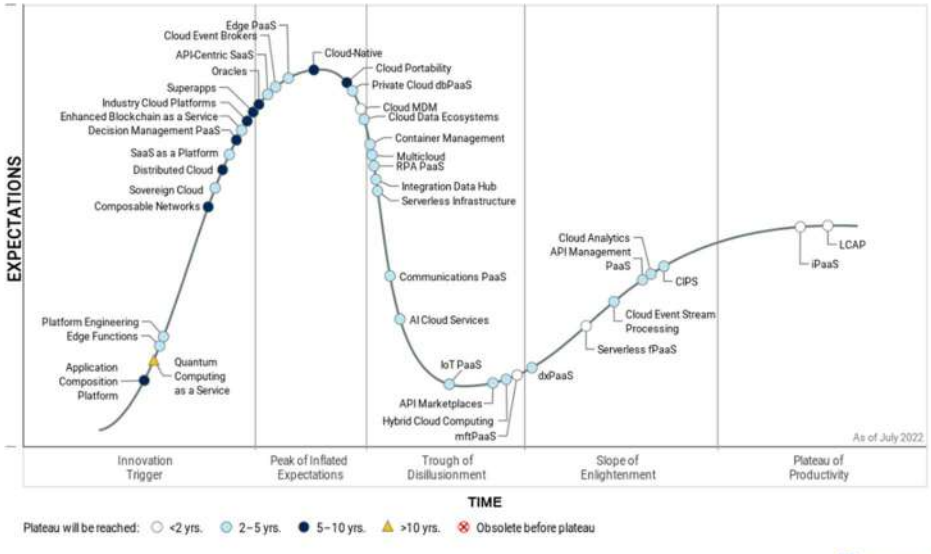
- **Big decision: In 2016 Dropbox decided to quit S3 and build own HW infrastructure**

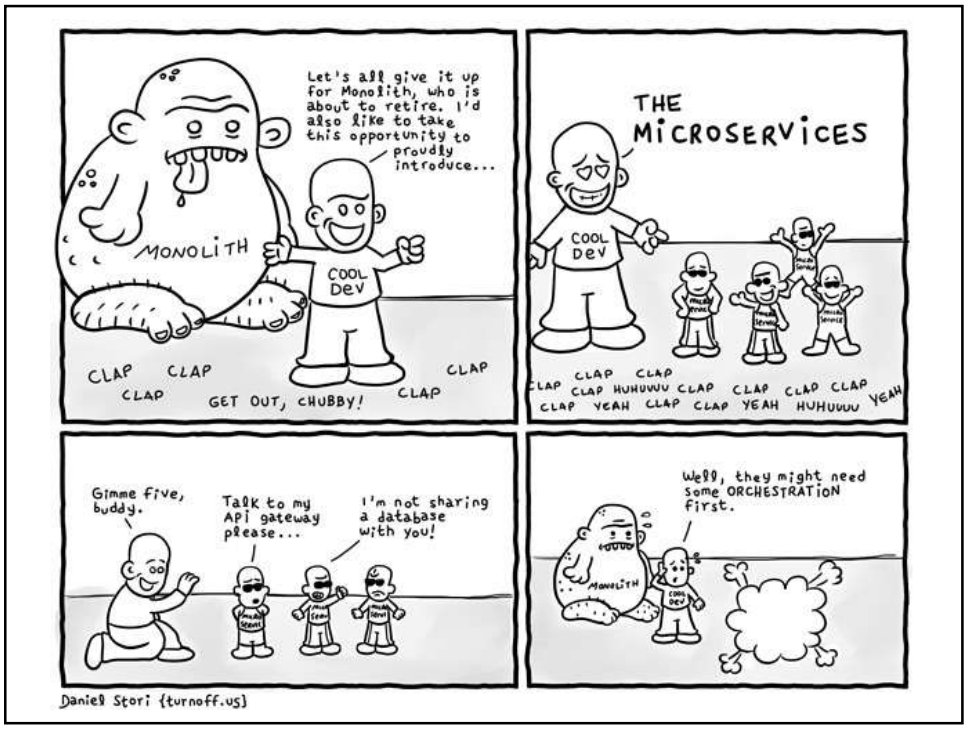
Ways to avoid the lock-in

- Competitors
  - Offer public services with the same functionality and API
- Free implementations
  - Running private cloud services with the same functionality and API
- Competitors and free solutions with compatible interface open up a lot of opportunities



Hype Cycle for Cloud Platform Services, 2022





CLOUD SYSTEMS

Microservices-Based Applications

Lecture #2

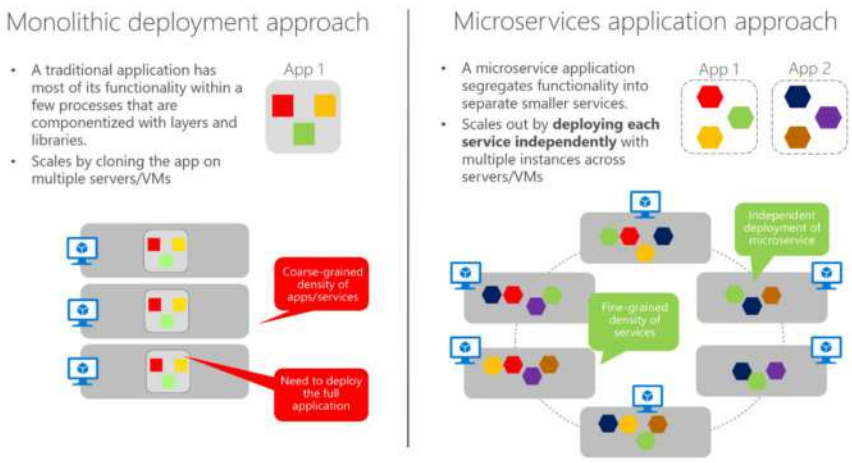
doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

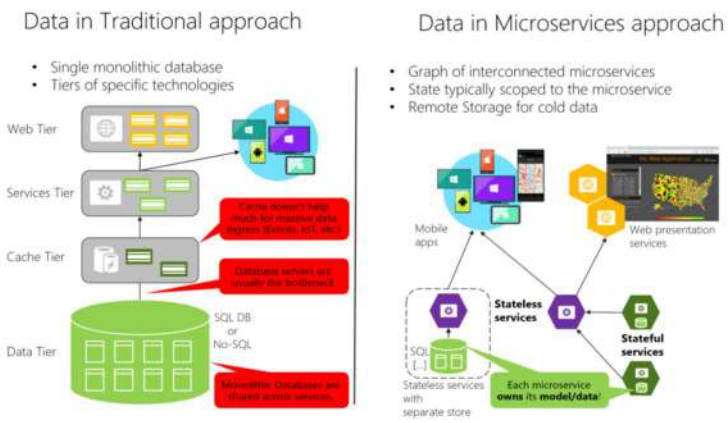
Service-oriented architecture

- Structure your application by decomposing it into multiple services that can be classified as different types like subsystems or tiers
  - Most commonly as HTTP services
  - Deployed as Docker containers
- Different from microservices architecture
  - SOA is based on large central brokers, central orchestrators at the organization level, the Enterprise Service Bus (ESB)

Microservices architecture



Data separation per microservices

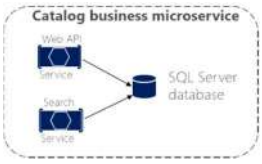


The relationship between microservices and the Bounded Context pattern

- The concept of microservice derives from the Bounded Context (BC) pattern in Domain-Driven Design (DDD)
- DDD deals with large models by dividing them into multiple BCs and being explicit about their boundaries
- Each BC must have its own model and database
  - Each microservice owns its related data
- Each BC usually has its own ubiquitous language to help communication between software developers and domain experts

Logical architecture vs physical architecture

- Building microservices does not require the use of any specific technology
  - Docker containers are not mandatory to create a microservice-based architecture
- Microservice could be physically implemented as a single service, process, or container
  - Parity between business microservice and physical service or container is not necessarily required in all cases when you build a large and complex application composed of many dozens or even hundreds of services
- The logical architecture and logical boundaries of a system do not necessarily map one-to-one to the physical or deployment architecture



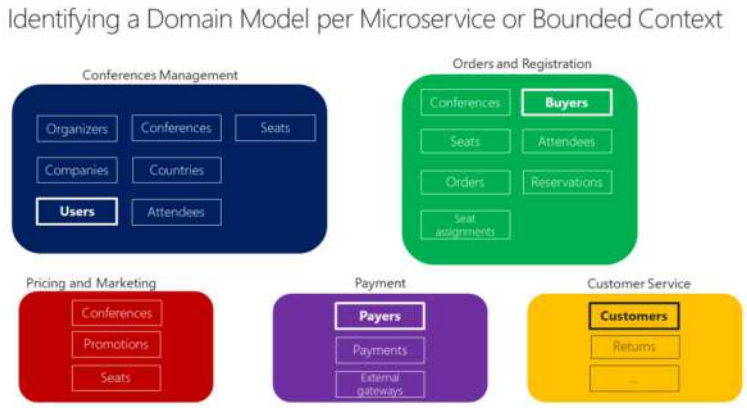
Challenges and solutions for distributed data management

- Challenge #1: How to define the boundaries of each microservice?
- Challenge #2: How to create queries that retrieve data from several microservices?
- Challenge #3: How to achieve consistency across multiple microservices?
- Challenge #4: How to design communication across microservice boundaries?

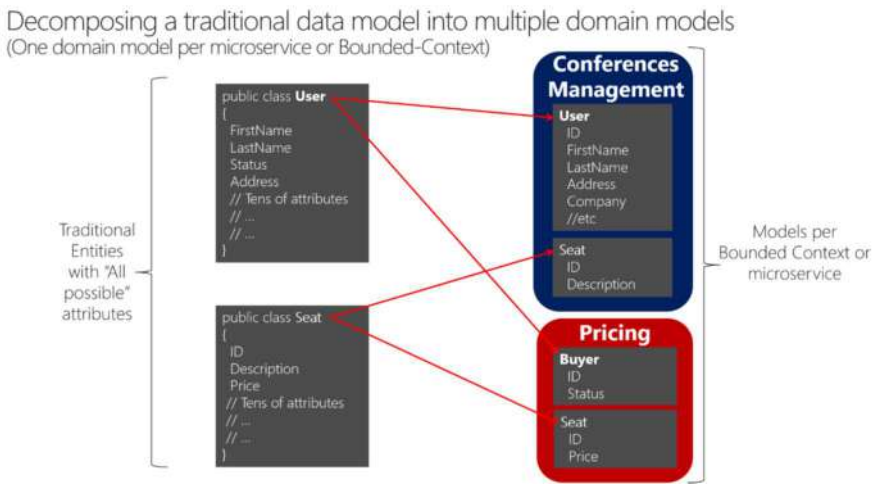
Identify domain-model boundaries for each microservice (1)

- Get to the most meaningful separation guided by your domain knowledge
  - The goal when identifying model boundaries and size for each microservice is not to get to the most granular separation possible
- Context Mapping (within DDD) identifies the various contexts in the application and their boundaries
- Similar to microservices
  - Autonomous
  - Implements certain domain capability
  - Provides interfaces

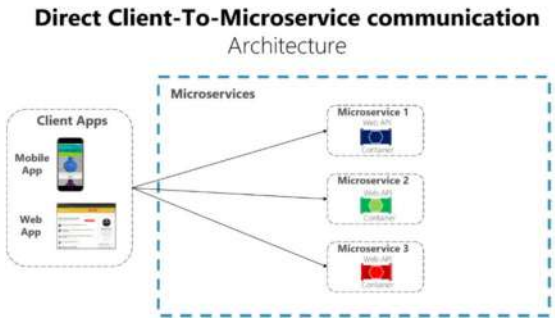
Identify domain-model boundaries for each microservice (2)



Identify domain-model boundaries for each microservice (3)



Direct client-to-microservice communication (1)



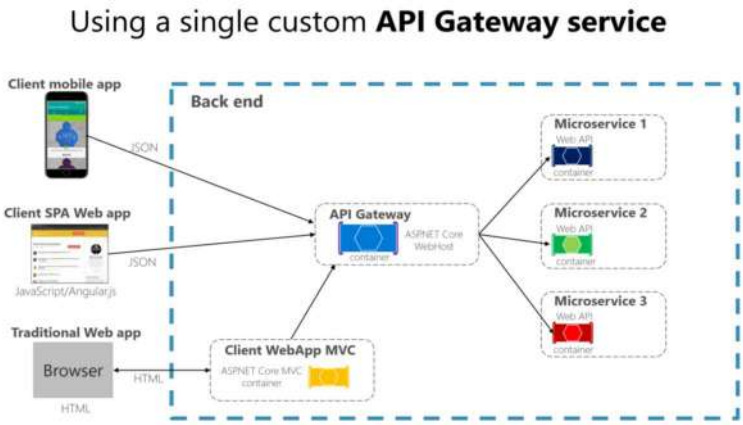
Direct client-to-microservice communication (2)

- Could be good enough for a small microservice-based application
  - Especially if the client app is a server-side web application
- How can client apps minimize the number of requests to the backend and reduce chatty communication to multiple microservices?
- How can you handle cross-cutting concerns such as authorization, data transformations, and dynamic request dispatching?
- How can client apps communicate with services that use non-Internet-friendly protocols?
- How can you shape a facade especially made for mobile apps?

The API gateway pattern (1)

- Coupling
  - Without a gateway, the client apps are coupled to the internal microservices
  - The client apps need to know how the multiple areas of the application are decomposed in microservices
- Too many round trips
  - A single page/screen in the client app might require several calls to multiple services
  - Aggregation handled in an intermediate level could improve the performance and user experience for the client application
- Security issues
  - Without a gateway, all the microservices must be exposed to the "external world", making the attack surface larger than if you hide internal microservices that are not directly used by the client applications
  - The smaller the attack surface is, the more secure your application can be
- Cross-cutting concerns
  - Each publicly published microservice must handle concerns such as authorization and SSL. In many situations, those concerns could be handled in a single tier so the internal microservices are simplified

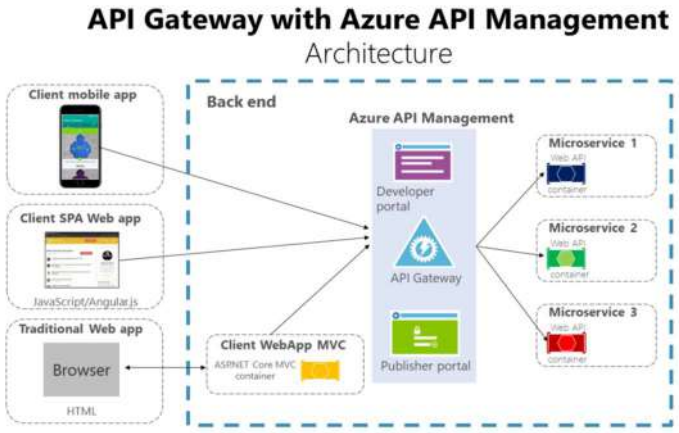
The API gateway pattern (2)



The API gateway pattern (3)

- Reverse proxy or gateway routing
  - The gateway provides a single endpoint or URL for the client apps and then internally maps the requests to a group of internal microservices
  - Offers a reverse proxy to redirect or route requests (layer 7 routing, usually HTTP requests) to the endpoints of the internal microservices
  - Helps to decouple the client apps from the microservices
- Requests aggregation
  - Aggregates multiple client requests (usually HTTP requests) targeting multiple internal microservices into a single client request (convenient when a client page/screen needs information from several microservices)
  - Reduce chattiness between the client apps and the backend API
  - It is more flexible to create aggregation microservices under the scope of the API Gateway, so you define the aggregation in code
- Cross-cutting concerns or gateway offloading
  - Offload functionality from individual microservices to the gateway, which simplifies the implementation of each microservice by consolidating cross-cutting concerns into one tier
  - Convenient for specialized features: Authentication and authorization, Service discovery integration, Response caching, Retry policies, circuit breaker, and QoS, Rate limiting and throttling, Load balancing, Logging, tracing, correlation, Headers, query strings, and claims transformation, IP allow listing

The API gateway pattern (4)



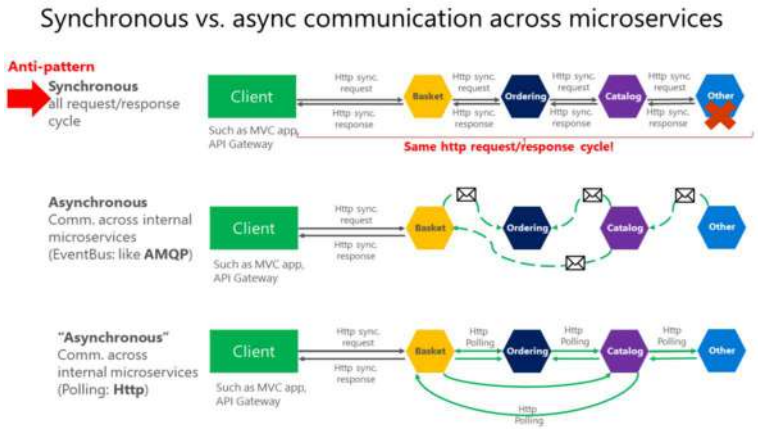
Communication in a microservice architecture (1)

- Synchronous protocol
  - HTTP is a synchronous protocol where client sends a request and waits for a response from the service
  - That is independent of the client code execution that could be synchronous (thread is blocked) or asynchronous (thread is not blocked, and the response will reach a callback eventually)
  - The important point here is that the protocol (HTTP/HTTPS) is synchronous and the client code can only continue its task when it receives the HTTP server response
- Asynchronous protocol
  - (HTTP methods POST, PUT, PATCH, DELETE can be designed in asynchronous mode)
  - Protocols like AMQP, XMPP, MQTT (protocols supported by many operating systems and cloud environments) use asynchronous messages
  - The client code or message sender usually doesn't wait for a response, it just sends the message as when sending a message to a RabbitMQ queue or any other message broker

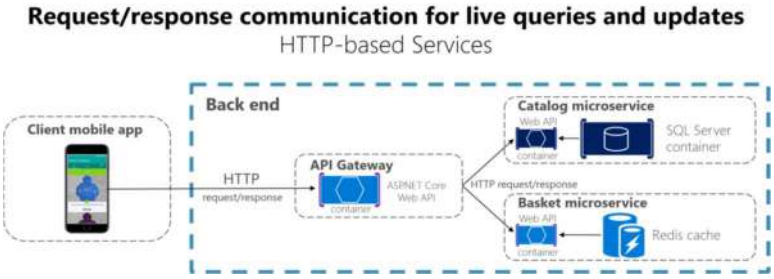
Communication in a microservice architecture (2)

- Single receiver
  - Each request must be processed by exactly one receiver or service
  - An example of this communication is the Command pattern
- Multiple receivers
  - Each request can be processed by zero to multiple receivers
  - This type of communication must be asynchronous
  - An example is the publish/subscribe mechanism used in patterns like Event-driven architecture based on an event-bus interface or message broker when propagating data updates between multiple microservices through events
  - Usually implemented through a service bus or similar artifact like Azure Service Bus by using topics and subscriptions

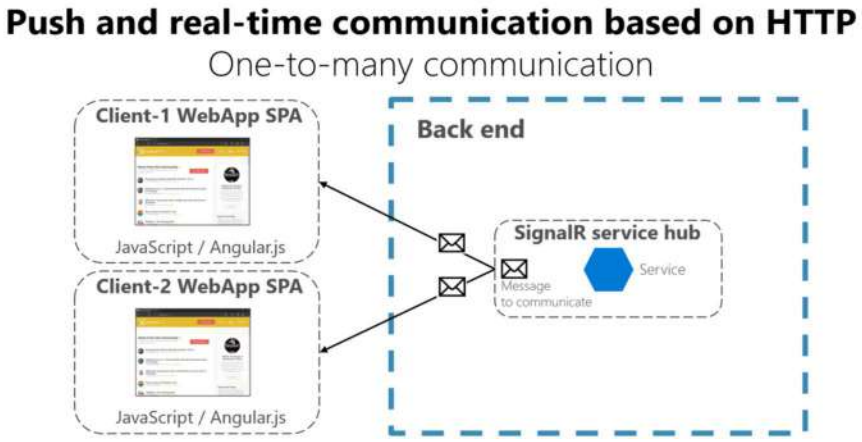
Communication in a microservice architecture (3)



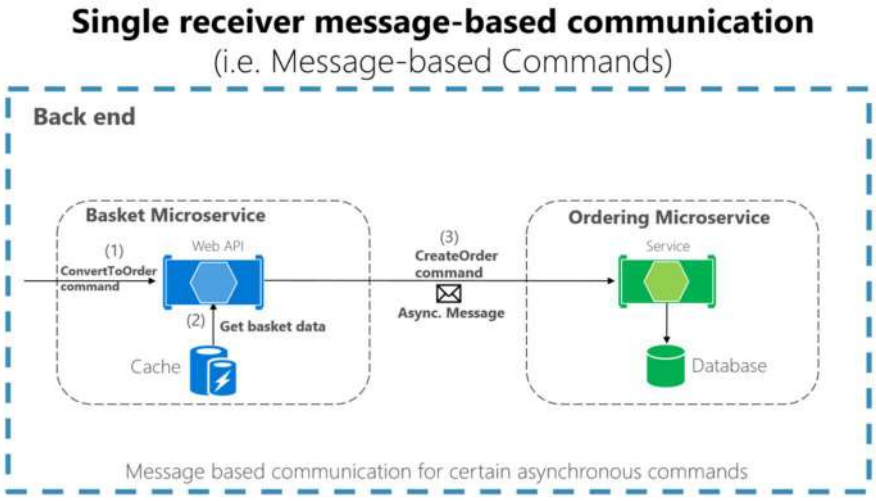
Communication styles (1)



Communication styles (2)



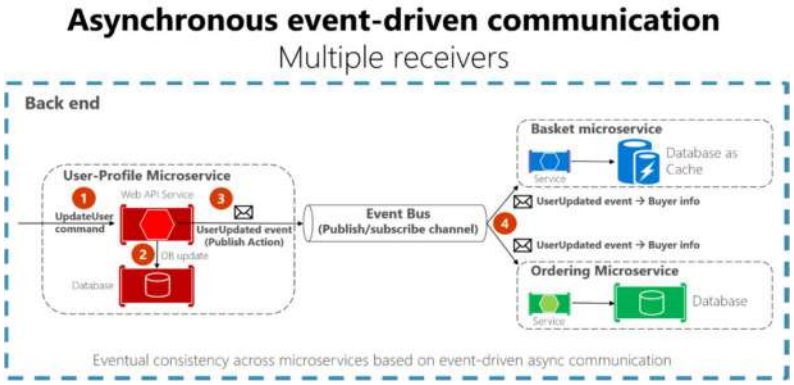
Asynchronous message-based communication (1)



Asynchronous message-based communication (2)

- Multiple-receivers message-based communication
  - Publish/subscribe mechanism so that communication from the sender will be available to additional subscriber microservices or to external applications
  - Helps you to follow the open/closed principle in the sending service
  - Subscribers can be added in the future without the need to modify the sender service
- When you use a publish/subscribe communication, you might be using an event bus interface to publish events to any subscriber

Asynchronous message-based communication (3)



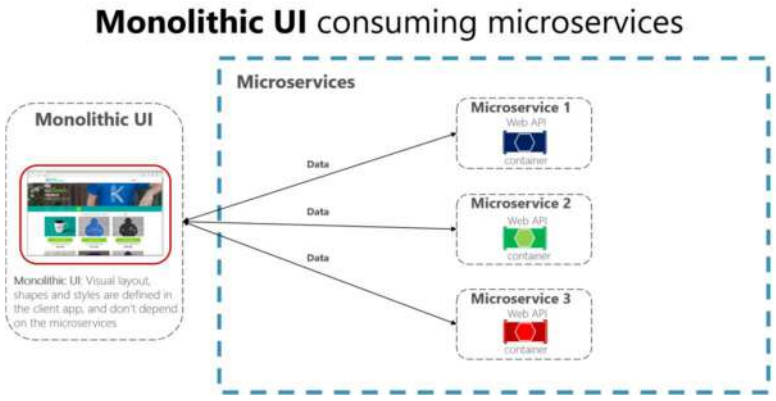
Creating, evolving, and versioning microservice APIs and contracts

- A microservice API is a contract between the service and its clients
  - If you change the contract, it will impact your client applications or your API Gateway
- The nature of the API definition depends on protocol
  - If you are using messaging (like AMQP), the API consists of the message types
  - If you are using HTTP and RESTful services, the API consists of the URLs and the request and response JSON formats
- It is important to have a strategy for your service versioning
  - Clients cannot follow your changes continuously
- Incrementally deploy new versions of a service in a way that both old and new versions of a service contract are running simultaneously
  - Embed the API version number in the URL or into an HTTP header
  - Use Mediator pattern (for example, MediatR library) to decouple the different implementation versions into independent handlers

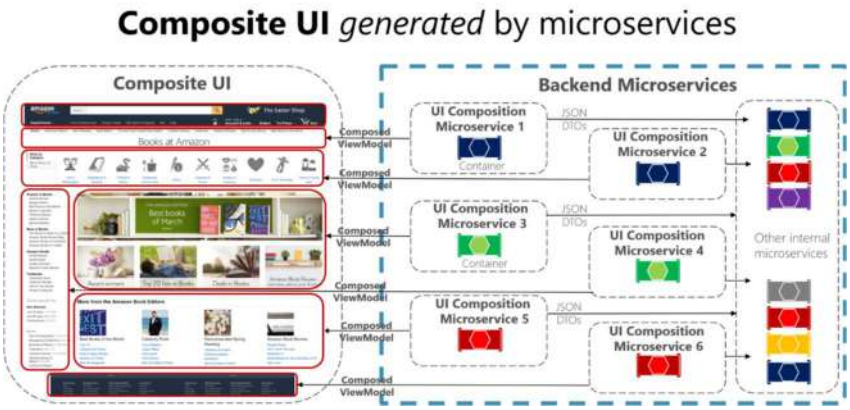
Microservices addressability and the service registry

- Microservice needs to be addressable wherever it is running
  - Microservice needs to have a unique name so that its current location is discoverable (similar as DNS for resolving URL)
- The service registry pattern is a key part of service discovery
  - The registry is a database containing the network locations of service instances
  - A service registry needs to be highly available and up-to-date
  - Clients could cache network locations obtained from the service registry
- In some microservice deployment environments (called clusters), service discovery is built in
  - For example, an Azure Container Service with Kubernetes (AKS)

Creating composite UI based on microservices (1)



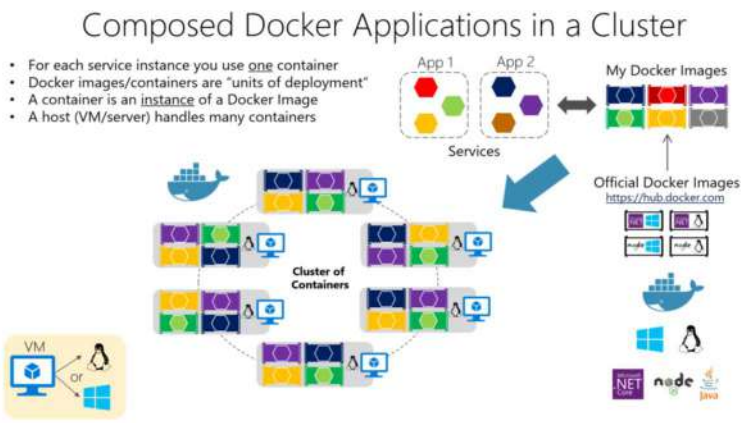
Creating composite UI based on microservices (2)



Resiliency and high availability in microservices

- Health management and diagnostics in microservices
  - Microservice must report its health and diagnostics, otherwise there is little insight from an operations perspective
  - **Correlating diagnostic events across a set of independent services and dealing with machine clock skews to make sense of the event order is challenging**
  - It is key that different teams agree on a single logging format. There needs to be a consistent approach to viewing diagnostic events in the application
- Health checks
  - Health events from a microservice help us make informed decisions and, help create self-healing services
  - Liveness: Checks if the microservice is alive, that is, if it is able to accept requests and respond
  - Readiness: Checks if the microservice's dependencies (database, queue services, etc.) are themselves ready, so the microservice can do what it is supposed to do
- Using diagnostics and logs event streams
  - Do not store events to central place
  - Event stream should write to standard output which is collected by execution environment (see Azure)
- Orchestrators managing health and diagnostics information
  - Development teams should focus on solving business problems and building custom applications, they should not focus on solving complex infrastructure problems
  - There are microservice-oriented platforms, referred to as orchestrators or microservice clusters

Orchestrate microservices and multi-container applications for high scalability and availability



Microservices Performance

Extra to Lecture #2

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

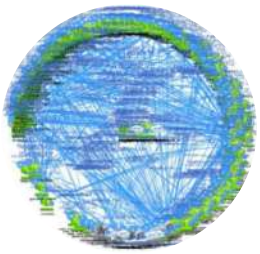
2025/2026

Benchmarking microservices

- Research of Prof. Christina Delimitrou from Cornell University
- Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices
  - https://doi.org/10.1145/3297858.3304004
- An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems
  - https://doi.org/10.1145/3297858.3304013

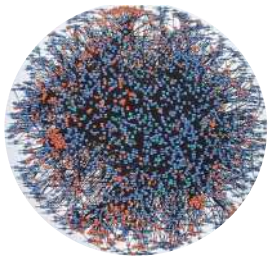
https://www.youtube.com/watch?v=Mf_C2xCpBdc

Performance management



Netflix

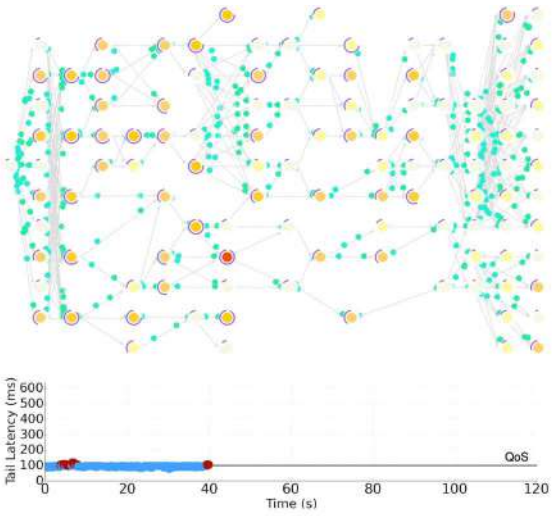
Twitter



Amazon

- Brings many benefits, but complicates cluster management and performance debugging
- Dependencies cause cascading QoS violations
- Difficult to isolate root cause of performance unpredictability

Performance visualization

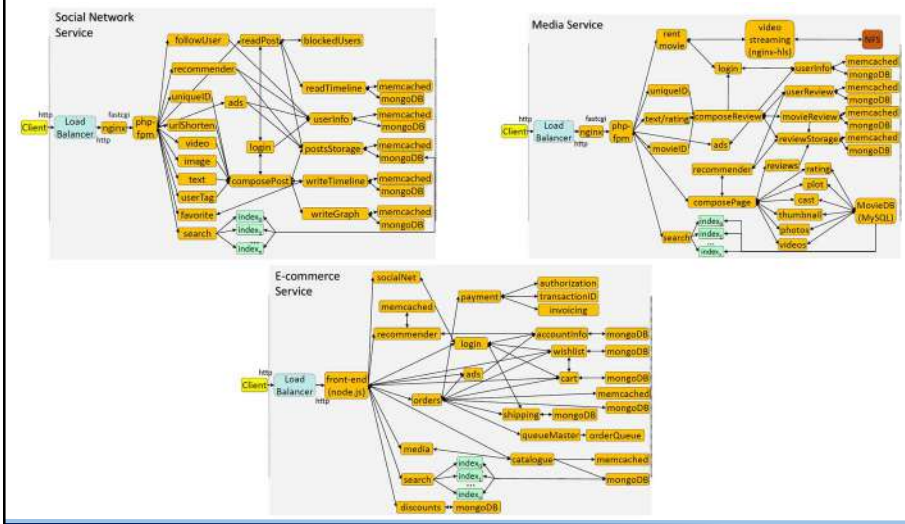


- Dependencies cause cascading QoS violations
- Empirical performance debugging
  - Too slow, bottlenecks propagate
- Long recovery times for performance

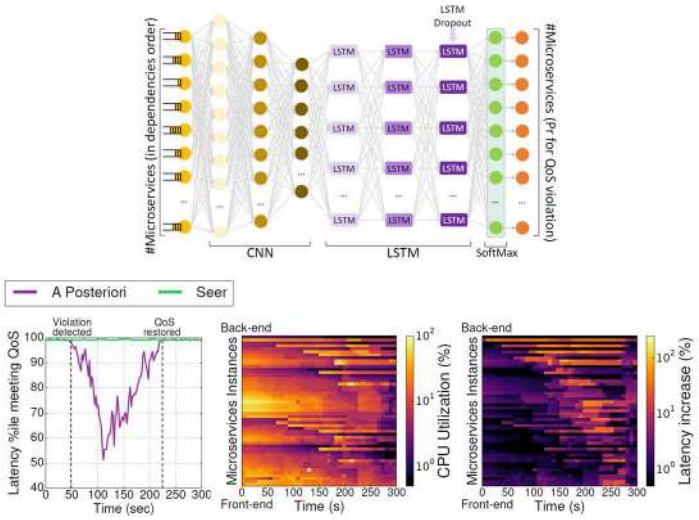
What did we see?

- The cloud scheduler watched each microservice pool
  - Each is shown as one dot, with color telling us how long the task queue was, and the purple circle showing how CPU loaded it is
- The picture did not show how many instances were active – that makes it too hard to render
  - Each pool had varying numbers of instances
  - The App Server was automatically creating and removing instances
- Each time the scheduler realized that it should add instances to a slow service, some of the "deadline violations" went away

Used applications for evaluation



Evaluation


LSTM Dropout
#Microservices (in dependencies order)
#Microservices (Pr for QoS violation)
CNN
LSTM
SoftMax
A Posteriori
Seer
Violation detected
QoS restored
Latency %ile meeting QoS
Time (sec)
Back-end
Front-end
Microservices Instances
Time (s)
CPU Utilization (%)
Latency increase (%)

CLOUD SYSTEMS

Elasticity and Scalability

Lecture #3

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

Concept of Elastic Scalability

- The pools of servers making up a microservice must support having instances added or removed **elastically** on demand
- We need to be sure that the performance of individual instances will remain good even if the number of instances suddenly doubles, or suddenly shrinks

Microservices overloads

- Microservices need to talk to other microservices, or from a storage server
  - Those shared servers can easily become hot spots
- **One big thing**: we could implement big services as monolithic products running on clusters (e.g. Oracle products)
  - But there is a limit to how big the clusters can grow
- **Many smaller things:** Split services into an array of "miniservers" – **sharding**
  - Very easy to scale, by adding shards, but imposes limitations

Why does single big service perform poorly?

- Remember Yahoo experiment (web page rendering under 100 ms) and Amazon experiences
  - Until 2005 "one server" was able to scale and keep up Amazon's shopping carts
    - A 2005 server often ran on a small cluster (e.g. 2 – 16 machines)
  - Worked well, but as the cloud grew, this form of scaling broke and companies threw unlimited money at the issue but critical services like databases still became hopelessly overloaded and crashed or fell far behind

Important research paper

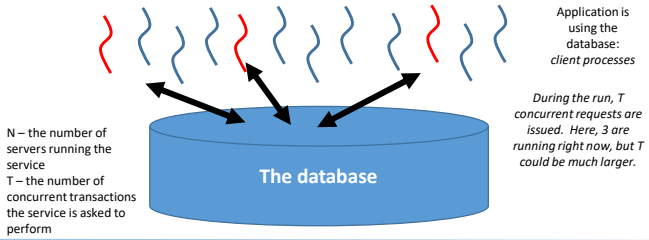
Jim Gray, Pat Helland, Patrick O'Neil, Dennis Shasha. **The dangers of replication and a solution.** In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996, pages 173 – 182. https://doi.org/10.1145/233269.233330

- An analysis to think about scaling for a system that behaves like a database
  - It could be an actual database like SQL Server or Oracle
  - But the model also covered any other storage layer where you want strong guarantees of data consistency
  - Single replicated storage instance, but it looks at structured versions too (e.g. a cluster)

- **Divide and conquer is the only option**

Core issue

- Lock-based consistency, which they model as database serializability (similar idea to "state machine replication" we know from previous project)
- Applications use read locks, and write locks, and because we want to accommodate more and more load by adding more servers, the work spreads over the pool of servers
- This is a very common way to think about servers of all kinds



Research findings

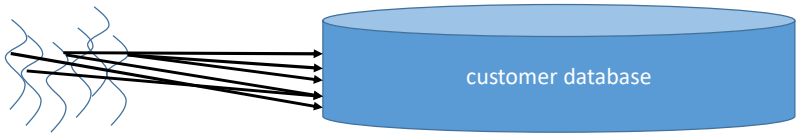
- Goal: For larger T, use more servers (enlarge N)
  - Typical business practice: "Our beta test was a success with 1000 shoppers. But we will see as many as 100000 at the same time. Can we just pay for extra cloud servers? Or must we redevelop the entire system from scratch?"
- Conclusion: It will not work
  - A scalable system needs to be able to handle more T's by adding to N
  - Paper shows that the work the servers must do will increase as T^5
  - Worse, with an even split of work, deadlocks occur as N^3, causing feedback
    - Because the reissued transactions get done more than once
  - Example: we anticipate doubled low (T becomes $2T$), so we deploy twice as many servers (N becomes $2N$). But overhead rises by $2^3 2^5 = 256$ times

Why do services slow down at scale?

- The paper pointed to several main issues
  - Lock contention – The more concurrent tasks, the more likely that they will try to access the same object (birthday paradox) and wait for locks
  - Abort – Many consistency mechanisms have some form of optimistic behavior built in. Now and then, they must back out and retry. Deadlock also causes abort/retry sequences
- The paper actually explores multiple options for structuring the data, but ends up with similar negative conclusions except in one specific situation

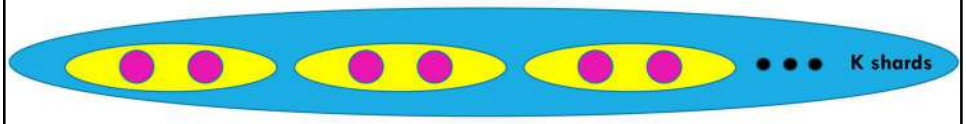
Key ideas of sharding

- There is one API for talking to the customer database

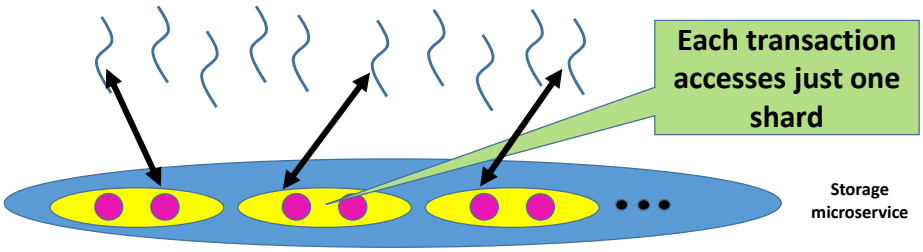


Remote procedure calls **are one way for code in one microservice to invoke APIs in some other microservice**

- Sharding breaks the single database into mini-databases "transparently"



Shards have 2 (or more) nodes



- Database (blue) is a set of shards (yellow)
- Each shard contain replicated servers on different computers (purple)
- State machine replication model is used to keep servers in synchrony

Benefits of replication?

- Gray's formula does not "kick in" for small N
  - We actually do get a speedup, if N is small enough
- Meanwhile, we gain fault-tolerance
  - If some server crashes, its replica can handle requests while it recovers

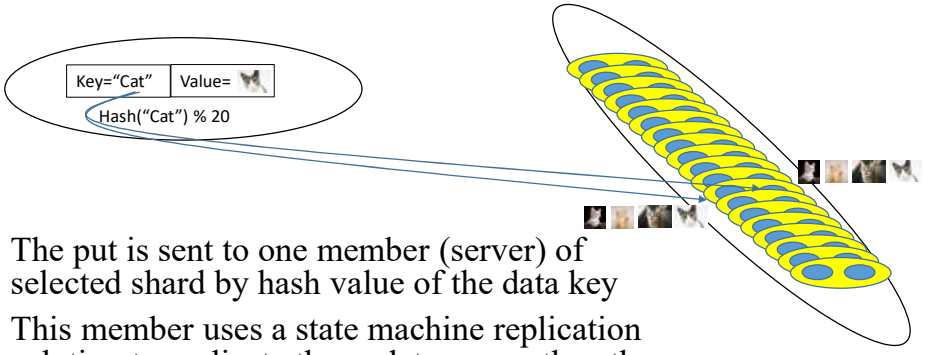
State machine replication

- We know the concept from DS course (see Lab #4)
- We have a set of servers {*P*, *Q*, *R*, …} and each server holds a replica of data and has the identical logic (code)
  - Any server can handle a read: the code just reads the local replica
  - Replicas see the same updates in the same order, hence stay in synchrony

Each instance accesses just one shard

- If updates and queries are done entirely on one shard, analysis presented in the paper does not apply
  - Because we can avoid locking (the update order is enough to ensure sequential progress of the replicas, with consistency)
- An application that needs to access multiple shards must treat the operations as concurrent, independent threads

PUT operation



- The put is sent to one member (server) of selected shard by hash value of the data key
- This member uses a state machine replication solution to replicate the update across the other shard members
- Notice that many keys map to the same shard, it uses an *O*(1) hash object for quick lookups

Some challenges for developers

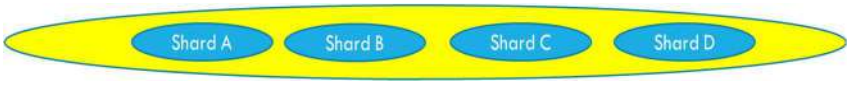
- Suppose your data does not have an obvious "key" to use. How would you create a unique name for each data object?
  - Key is like a "name"
- What if we wanted to store a data structure, such as a table or tree, in the DHT? Would we put the whole thing on one shard with one key, or spread each object (each row, or each node) around, using one key per object?
  - Is a data structure one object, or many objects?
- What performance impact would these choices have?
  - If the application using the data is on machine *P*, and the data is in a DHT on machines *Q*, *R*, *S*, …, what costs would it pay?

Elasticity adds a further dimension

- If we expect changing patterns of load, the cache may need a way to dynamically change the pattern of sharding
- Since a cache "works" even if empty, we could simply shut it down and restart with some other number of servers and some other sharding policy
  - But cold caches perform very badly
- Instead, we would ideally "shuffle" data

Elastic shuffle (1)

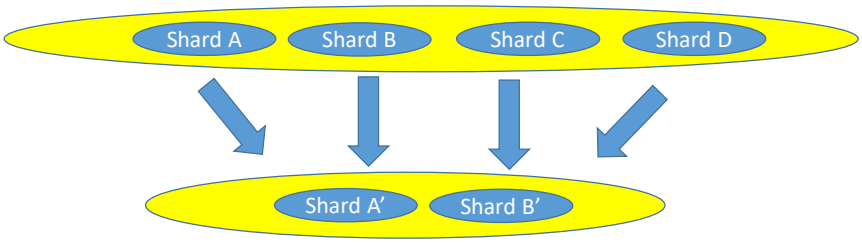
- Perhaps we initially had data spread over 4 shards



- We could drop down to 2 shards during low-load periods
  - Of course, half of items (hopefully, less popular ones) are dropped

Elastic shuffle (2)

- Here, we shuffle data to map from 4 shards down to 2



- Later, we could shuffle it again to elastically expand the cache

Cloud platforms automate this

- Elasticity can be hard to implement
- AWS, Amazon and Google all did the hard work for us
- When we use e.g. MongoDB, it automatically adjusts based on load, and the user is not even aware of it

Example sharded DHT API

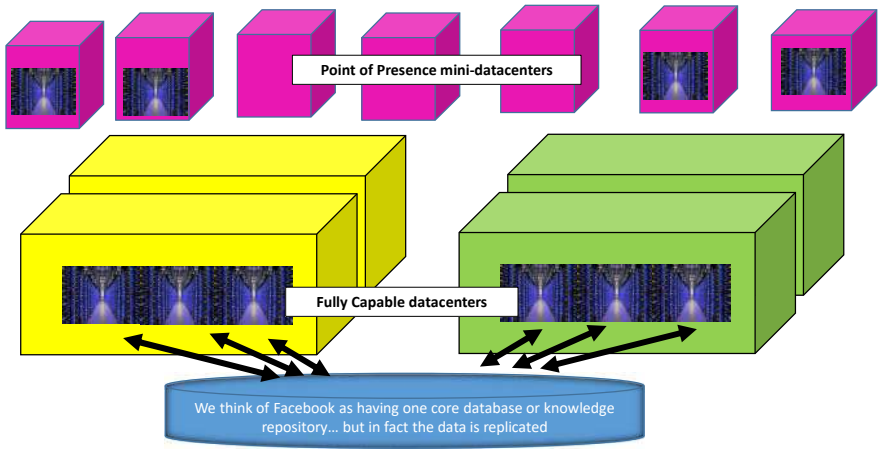
- MemCached API was the first widely popular example
- Today there are many important DHTs (MongoDB, Cassandra, DynamoDB, MemCached, DB Rocks) and the list just goes on and on
- All support some form of (key, value) **put**, **get**, and (most) offer **watch** operations
- Some hide these basic operations behind file system APIs, or "computer-to-computer email" APIs (publish-subscribe or queuing), or database APIs

Use case: Facebook's content-serving infrastructure

Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, Harry C Li. **An analysis of Facebook photo caching.** In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013, pp. 167 – 181. https://doi.org/10.1145/2517349.2522722

- Facebook's software for content serving is a heavily sharded service built with lots of microservices that talk to each other
  - The "transactions" are all very simple reads and writes and they can be done in a single operation on a single shard at a time
  - Cloud-scale infrastructure that runs on point of presence datacenters in which key-value caches are deployed
- Role is to serve videos and images (blobs) for end-users
  - **Weak consistency is fine because videos and images are immutable (each object is written once, then read many times)**
  - Requirements include speed, scaling, fault-tolerance, self-management

The Facebook blob cache is part of a hierarchy deployed at global scale



Content cache goals

- Support very fast lookup by any client from anywhere in the world
- Minimize the number of requests that reach the database backend
- Uses sharded key-vale storage at every level

Binary Large Objects (blobs)

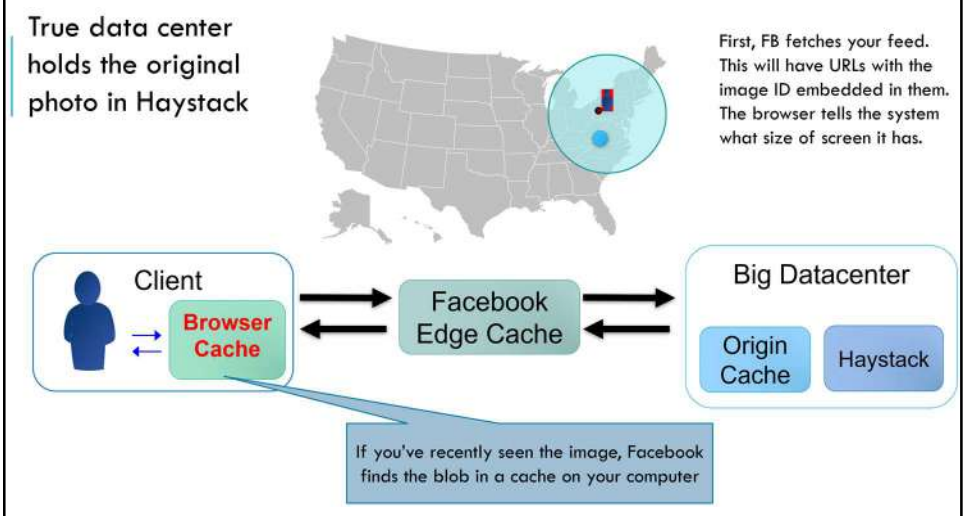
- Facebook image data is stored in blobs
- This includes
  - Original images, videos
  - Resized versions, and ones with different playback quality
  - Versions that have been processed to tag people, augmented reality

Haystack

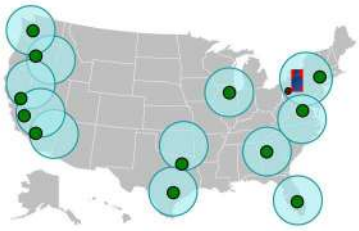
- Holds the real image and video data in huge "film strips", write-once
- Designed to retrieve any object with a single seek and a single read
  - Optimized for SSD (these have good transfer rates but are best for write-once, reread many loads, and have a long delay for starting a write)
- Facebook does not run a lot of copies
  - In USA one is on the West Coast, and one more on the East Coast
  - Each has a backup right next to it
- Main issue: Haystack would easily get overloaded without caching

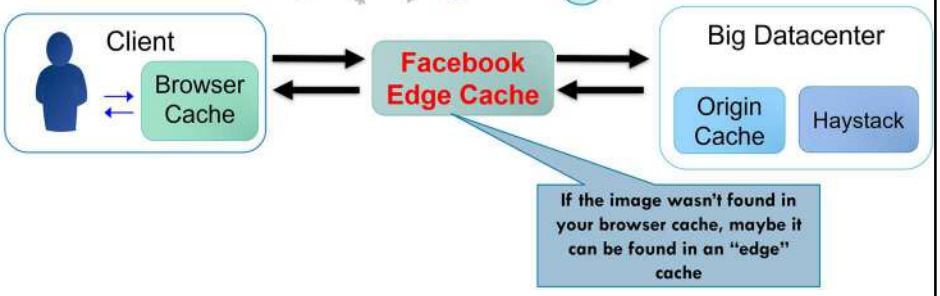
A cache for blobs?

- The keys would be photo or video id
- For each unique blob, Facebook has a special kind of tag telling us the resized dimensions of the particular instance we are looking at
  - Resizing takes time and requires a GPU and Facebook wants to avoid recomputing them on each fetch

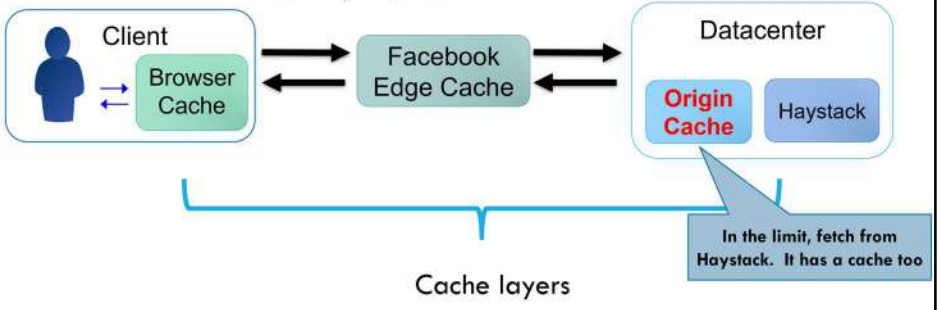


With a UID and a target size, we can search for the blob in the nearest point of presence cache.

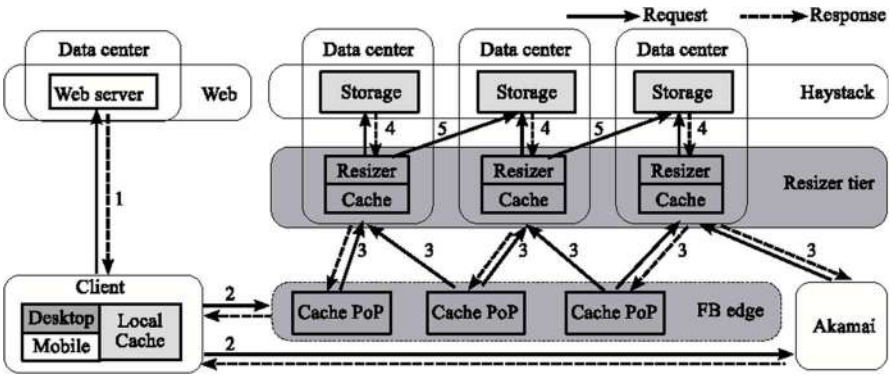




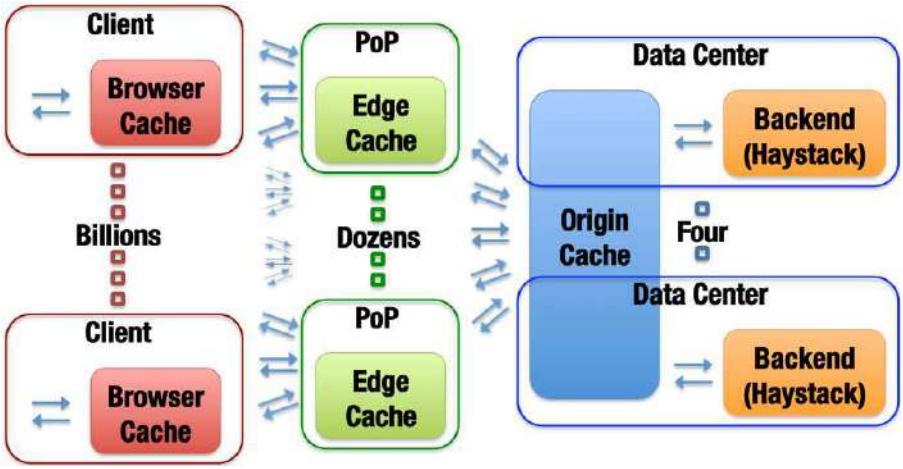
Origin:
Coordinated
FIFO
Main goal:
traffic sheltering



Instrumentalized architecture details for experiment

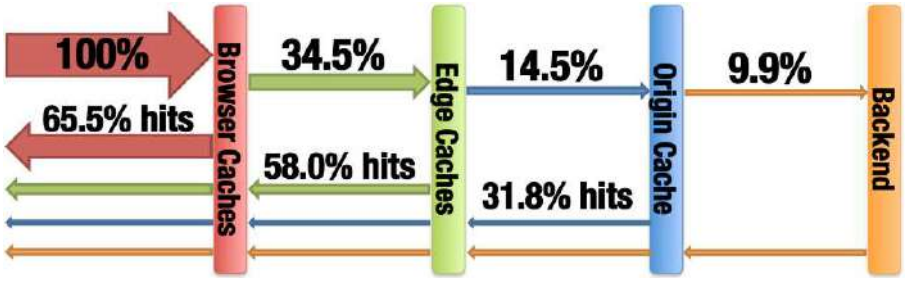


Blob-serving stack



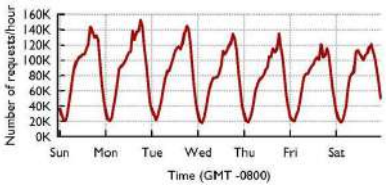
Observations

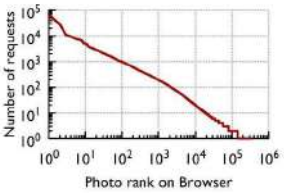
- Month long trace of photo accesses, which we sampled and anonymized
- Captures cache hits and misses at every level



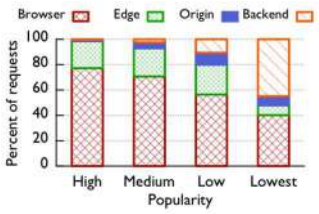
Observations

- Access vary by time of the day and by photo (some are more popular than others)



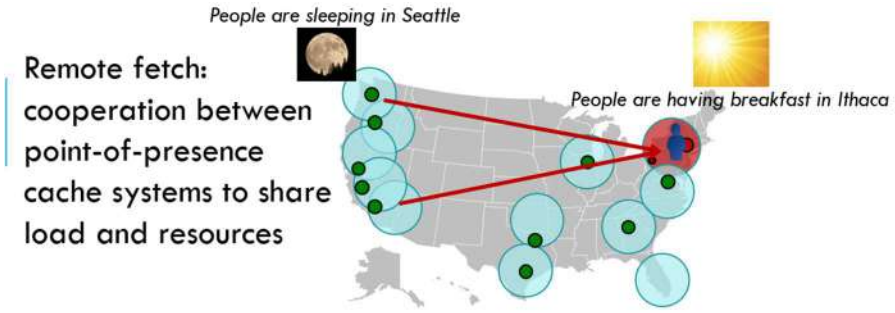


- The main job of each layer is different. This is further evidence that cache policy should vary to match details of the actual workload



Geo-scale caching

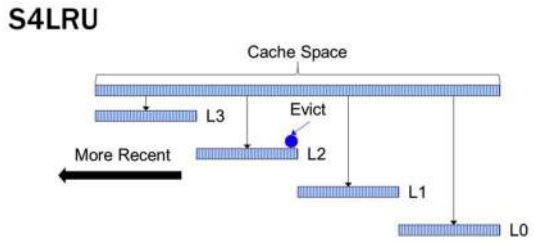
- One way to do far better turns out to be for caches to collaborate at a WAN layer
  - Some edge servers may encounter "suddenly popular" content earlier than others, and so those would do resizing operations first
- WAN collaboration between Edge caches is faster than asking for it from Haystack, and also reduces load on the Haystack platform
- Key insight: the Facebook internet is remarkably fast and stable, and this gives better than scaling because the full cache can be exploited



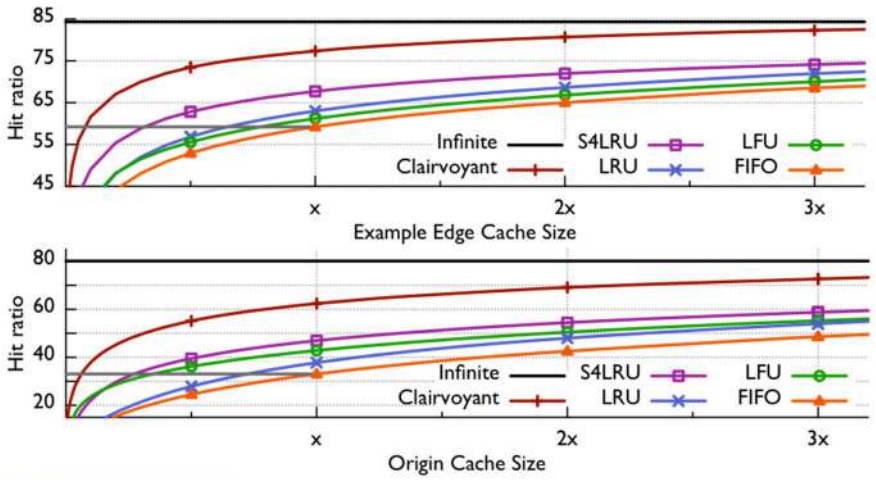
- In Ithaca, most of the content probably comes from a point of presence in the area (near Boston)
- But if Boston does not have it or is overloaded, they casually reach out to places like Spokane, or Arizona, especially during periods when those are lightly loaded, like early morning

Cache retention/eviction policy

- Facebook was using an LRU policy
- Experiment used trace to evaluate a segmented (4 segments) scheme (S4LRU)
- It outperforms all other algorithms => Facebook decision was to switch to S4LRU



S4LRU is far better than normal LRU



S4LRU did not really work well

- It turned out that the algorithm worked well in theory but created a pattern of reads and writes that were badly matched to flash memory
- Resulted in a whole two year project to redesign the "operating system" layer for big SSD disk arrays based on flash memory: **the RIPQ project**
  - Then S4LRU finally worked as proposed

RIPQ: Key innovations

Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, Kai Li. **RIPQ: Advanced Photo Caching on Flash for Facebook.** In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015, pages 373 – 386. https://dl.acm.org/doi/10.5555/2750482.2750510

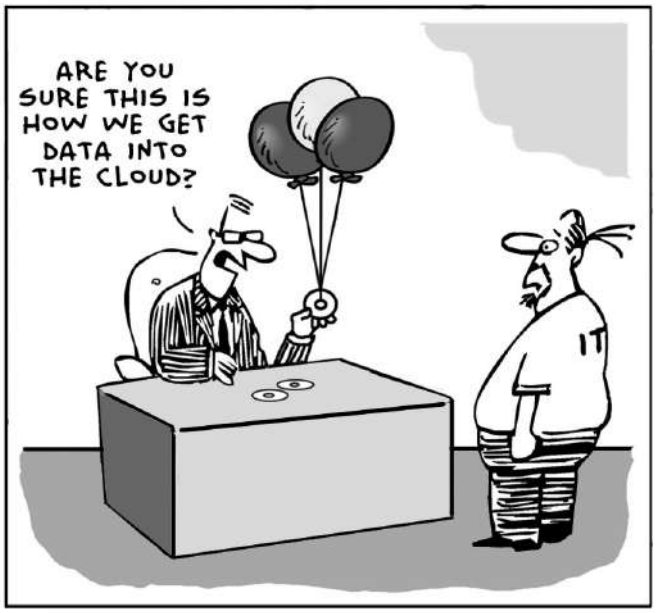
- Only write large objects to the SSD once, which treats SSD like an append-only "strip of images"
  - SSD is quite good at huge sequential writes. So this model is good
  - Same trick was needed in Haystack
- But how can it implement "priority"?
  - **The cache is an in-memory data structure with pointers onto the SSD**
  - It checkpoints the whole cache periodically, enabling warm restart

Summary

- A good example of a microservice is a key-value cache, sharded by key
- The shard layout will depend on how many servers are running, and whether they are replicated
  - These are examples of configuration data
- Many microservices are designed to vary the number of servers elastically
- A company like Facebook wants critical microservices to make smart use of knowledge about photo popularity, and about patterns of access
  - With this information it can be intelligent about what to retain in the cache, and could even prefetch data or precompute resized versions it will probably need
  - Fetching from another cache, even across wide-area links, is advantageous

Conclusions?

- In a cloud setting, we need massive scalability
  - This comes from hierarchy, replication and sharding
  - But we also need to match solutions to the setting
- Sharing in a (key, value) model is a great way to deal with scale
  - But that only gets you to the next set of questions
  - At cloud-scale nothing is trivial
- Facebook's caching "product" is great global system
  - Even if it was based on simple decisions, the global solution is not simple at all



CLOUD SYSTEMS

Stateless Programming and Function (Lambda) Computing Model

Lecture #4

doc. Ing. Martin Tomášek, PhD.

Department of Computers and Informatics

Faculty of Electrical Engineering and Informatics

Technical University of Košice

2025/2026

Overview

- Application walkthrough
- Stateless programming
- CAP, BASE
- Key value data model
- Concept of customization
- Code extensibility

Application walkthrough

- How does a typical interaction with the cloud work?
- First, the user does something on their browser (or app)

- The app binds to the cloud where its servers live by making a TCP connection
  - It will reuse the connection over long periods, so this step only imposes costs the first time
- On the other hand, some apps might need to talk to two or more sets of servers, and they would need one binding per server

Cookies

- To identify the user, a cloud system often requires that the application include a "cookie" with each request
- The cookie normally just holds a unique id, but could be a whole file with lots of data about you
- For security, the rule used is that when an app talks to some domain, it can only access the cookies created by/for that same domain

Transporting request/response messages

- The app can send a message holding the request, plus the cookie
- The request generally looks like a method call in the code in the app
  - But in fact a "stub" packages (serializes) the arguments into a message
  - The message is sent over TCP and the first-tier server deserializes (unpacks) the request to perform it
- The reply message will come back on the same TCP connection

Spread of microservices

- The first-tier server builds a web page (with URLs for any other media or advertising content)
  - But spreads any real work out over the pools of microservices
- This entails doing more requests, as messages
  - They might not be issued as RPC requests over TCP – some could be sent on a "message bus" or a "message queue"
- Some instance of the microservice becomes responsible for this request
  - E.g. the original request was "inline roller skates for a 12-year old boy", but that was the version seen by the first tier
  - Other particular microservice instance might be a specialist on "popular products matching the search" or "special deals to offer" or "customers like you also look at", etc.

Parallelism

- There was no real parallelism when the original message was sent from the app to the first tier
- But there could be hundreds of microservices running concurrently, generating portions of the data used to construct the web page response
- This gives a big speedup compared to doing everything in one server

Personalization

- Recall that all of these instances are stateless
- But also remember that they do have the original cookie giving the account ID information
  - This allows the microservice to create a personalized reply
- The real data resides in a key-value storage layer, or a file system, or a database system

Tools for relaying requests to microservices

- **Remote procedure call (gRPC)**
  - Client must know the server IP address
  - This is a limitation, often, a first-tier component will not know
- **A message bus** automatically tracks the members of the pool
  - You can ask for your request to go to any single member, or to all
  - Later the member that picked the request up can reply
  - If a timeout occurs (like because the member crashed), you reissue it

Message queue

- A message queue is more like an email system, request has a message group address (like an email address), and is stored under that folder
  - In fact it can even be used as a message bus, by changing configuration parameters
- A member of the microservice pool can ask for the next "unread" message, reply to the sender, etc.
  - Designed for scaling: a message topic will automatically be sharded into a set of message pools that are accessed separately
  - For better efficiency, many applications read a batch of messages, all at once, from the same group: "all pending messages", or "next 100"
  - Batched processing reduces overheads of talking to the bus again and again

App Service

- Each microservice is managed by the "App Service", which controls how many instances are launched, when to launch/kill members, and monitors overall load

1. Monitor the load on microservice nodes
2. Detect overload cases, such as backlog developing
3. Grow the pool of replicas for that microservice by launching new copies – new container that hold instances of the program implementing the microservice

Building code to run in the "pool of servers" model

- Recall from previous lectures that it can be much easier and more flexible to write a single program that is not heavily multi-threaded, and have each request processed by a single program instance
- Each instance access only one data key (a single shard)
- We can scale our service out by just running it many times

Stateless vs stateful programs

- A newly launched program has some initial state
  - Variables you initialized
  - Files on the local disk that it reads as input
- In the cloud, we call this local data and the local file system
- The cloud also has a global cloud file system, that you talk to it via messages
  - The cloud uses gRPC (Google), or the Azure/AWS equivalent

Stateless model (1)

- What does "state" mean? For a program it could mean
  - Values of variables inside the program as it runs
    - Constants as well as dynamically allocated variables/values
  - Data that the program needs to read, but will not change
    - Such as data extracted from the event that triggered the program to run
    - Or data it loads from some sort of file or database
    - Or data read from the OS like "today's date and time"
  - Data that the program might actually have to modify
    - It could live in the "environment", or the "Windows Active Directory"
    - It could live in a server, such as a file server or a key-value store or a database

Stateless model (2)

- A stateless program is a normal program, written the way you write any computer program
- But it follows one rule: do not hold "persistent" data (that would still be needed if we shut it down, then restart it) on the computer where it runs
  - It still has "state" in its variables and so forth, this is not an issue
  - But if it needs to save something, that update has to be sent to some other place, like into the cloud's global file system, or a database, etc.

The first tier of a cloud is stateless

- This design idea dates 2000 to Professor Eric Brewer, at Berkeley
- He observed very scalable pools of microservices are easier to build and extend if the data used is all read-only
  - Sometimes he called this "soft state", meaning the "hard version is elsewhere"
  - But most people just call this a stateless model
- A stateless server is easy to shut down (kill it, throw away any files it created)

Which can we do in a stateless way?

- Consider the Amazon shopping web pages
- We can build these by looking up data held in other services
  - A microservice that tracks your purchase history
  - A microservice that computes recommendations
  - A microservice that resizes images of products to fit your screen
  - A microservice that manages the shopping cart

Which can we do in a stateless way?

- Consider the Amazon shopping web pages
- We can build these by looking up data held in other services
  - A microservice that tracks your purchase history
  - **A microservice that computes recommendations**
  - **A microservice that resizes images of products to fit your screen**
  - A microservice that manages the shopping cart

Often we pair stateless frontend with stateful microservices

- Stateful services are harder to build, so usually vendors like Oracle or Amazon or Microsoft do that for you
  - The cloud file system is the most obvious example
  - Others include the image storage service (in addition it usually can resize photos, segment them, perhaps even recognize who is in them), databases, DHTs, etc.
- Then we might build a stateless service that sits in front and adds some of our own logic but relays anything that needs to be saved into the stateful tier

Some important stateful servers in the cloud

- Azure: The global file system, the Cosmos Database, the BLOB store, the message queue service, various DHT products like Cassandra
- AWS: Many of the same options, plus DynamoDB, Amazon's scalable DHT
- Even though these may use the term "database", and you access them with SQL, but e.g. Cosmos and Dynamo are not true databases (NoSQL)

Helpful ideas for stateless design

- Use a scalable DHT as a cache for any data the program reads
  - With luck we will get a good hit rate (like in Facebook) and this will shield the big stateful systems at the back-end from most of the load
- A DHT can hold much more stuff than any single computer could ever hold
- Be tolerant of staleness
  - This is a high cost to be sure our cached data
  - For example, if a site says "2 products left at this price", that could be a bit stale

Key ideas for stateless design

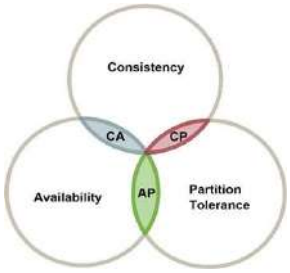
- Soft state can easily be regenerated, so do not worry much about fault tolerance
- In our DHT examples last week, we talked about state machine replication for fault-tolerance, but this involves using atomic multicast or Paxos for updates, to ensure that all replicas see the same update sequence
- If we do not care about losing data during a crash, we can skip that step
  - The real data would live in a back-end database or file system
  - Since we are only keeping cached data in the DHT, if a shard forgets, we can always fetch it from the back-end system a second time

CAP definition (informal)

- Consistency
  - The system responds using the most current values (updates)
  - **Conflicting updates are performed in some system-selected order by all replicas**
  - Queries thus will see a "single system" and will be up to date
- Availability
  - The system is **rapidly responsive**, and will **self-repair** if some single component fails, restoring normal functionality asap
  - If too many components fail all at once, availability is lost
- Partition Tolerant
  - A data center can have network issues, or entire services can be down
  - Yet **as seen from outside, the cloud should continue to respond even if its first-tier services are temporarily unable to talk to some inner services** they would normally depend upon

In the cloud, not every subsystems needs the strongest guarantees

- Basically, Eric Brewer argues that
  - The theoretically "best" solution often brings heavy costs
  - Consistency is one example: conflicting database updates can be forced into an agreed order, but this takes time and involves node-node dialog (hence only CA)
  - **Remember that to earn the most money, you need the fastest possible responses.** Brewer concluded that this means you might have to **relax consistency** (hence only AP)



Eric Brewer's CAP theorem

- **A system can only have 2 out of 3 from CAP**

- What to do?
  - Relax consistency (C)
  - Gain faster response (A)
  - Generate responses even when unable to talk to back-end servers (P)

BASE methodology (goes with CAP)

- Invented at eBay, adopted by Amazon and others
- BASE = Basic Availability, Soft State and Eventual Consistency
  - "Use CAP. It may cause inconsistency. Clean up later."
- How BASE works
  - Cache data but do not worry about cache entries getting stale
  - They were valid a little while ago

Today's cloud is a CAP + BASE world…

- All in all, cloud systems manage with stale data / weak consistency
- Most applications are read-only and are fine with slightly stale data
- In IoT, this will change, when we look at IoT we will need more

…and a key-value world

- The most common form of storage is a sharded key-value store
- Benefit: A key-value storage is just a file system, but one optimized for "whole objects" rather than an array of records represented by "byte-vectors"
- Databases are important too, but the cloud tries to shield them from load by putting big key-value storage caches in front of them

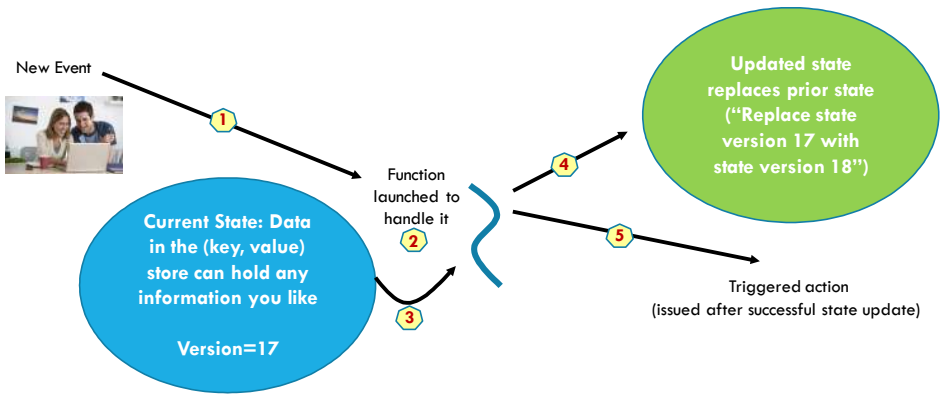
Key-value model

- Basically, like a file system but where the data is read or written all at once, as if version 5 "replaces" version 4
- The "key" is just the word we use for "file pathname"
  - As if the key-value store was really a file system
  - They even look like file system pathnames, starting from the root ("/")
- The "value" is just a vector of bytes
  - The key-value store does not care about the actual meaning of those bytes
- API: put, get, watch

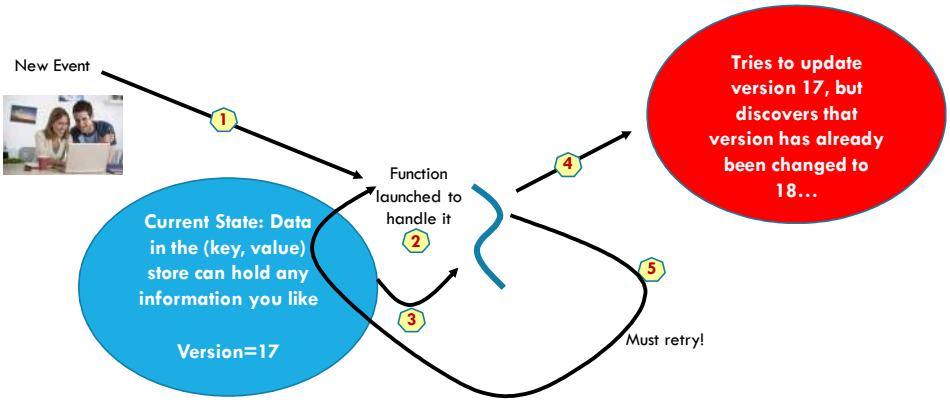
What makes all of this hard?

- Weak consistency
  - Many layers of the cloud cache data, and many actions are taken using potentially stale information
- No locking
  - We do not have what database systems refer to as transactions
  - Each key-value operation is done all by itself, without locking
- But there is a kind of versioning that can be used as an alternative to locks

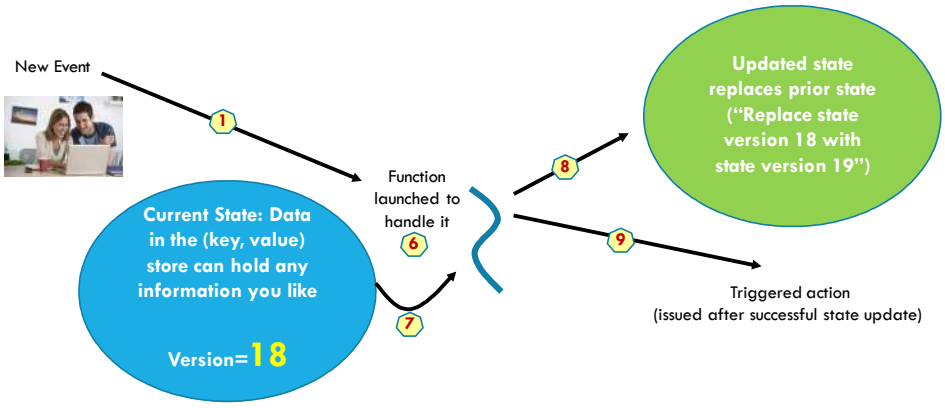
Atomic replace with a stateless function



Atomic replace with a stateless function



Atomic replace with a stateless function



Managing elastic microservices is hard

- It is very hard to build a completely new elastic application from the ground up
- In practice, these microservices need to be connected to all sorts of specialized vendor-created cloud infrastructure services and tools
- So, how do cloud customers create new customized solutions?

Concept of customization

- It centers on having some prebuilt component (for us, a very generic form of microservice instance), but one that does not really "do" anything
- For example, perhaps you can send it a very basic kind of request called ping, and it sends back a reply "ok", nothing more
  - This would not be a useful microservice but it would become the framework users can customize

Things that are NOT customization

- Telling a data analysis program which files to read
- Giving a parameter to a program to tell it to reverse the sort order
- Typing a command to a shell like bash to run some program

What makes it customization?

- When we talk about customization, you always **write actual code**
- Some customization frameworks require specific languages, while others allow customization with code in any language you like
  - .NET is very flexible, and has 44 languages built in
- But the central idea is that when certain "events" occur, your logic is executed and temporarily takes over – their platform is under your control

Why would we customize a database?

- Take the example of a database system
- Suppose we post some query that will trigger a particular action
  - If the price of Microsoft drops more than 1.5 % notify me by text message
  - If this patient's COVID-19 test comes back positive for infection, immediately start quarantine
  - If this car is likely to crash into the next car, notify them both instantly
    - You may also have to code the logic for "this car", "next car", and "likely to crash"
- Isn't it useful?

What is customization "about"?

- You are bringing arbitrary code into that existing server or platform component and changing what it can do
- In addition you are also "hitching a free ride" on the integration it already supports, with the app manager and with the tools it uses for elasticity
- What options exist for customizing a microservice?
- Let's think about customization as a more general concept

Customizable components

- You get to pick the component from the repository, and often there are add-on packages
- This is a bit like setting configuration parameters
- Many cloud services allow (or even require) customers to configure them before use
  - For example, you might set the minimum or maximum number of instances that will be running… this has cost implications for you
- Where do configuration live?
  - There are a variety of common options, but big cloud vendors try to standardize
  - On Azure, most configuration data is held in JSON files
  - Some programs want some form of configuration file on the local disk, in the local directory
    - It might have a name like something.rc or something.cfg

Other forms of customization (1)

- Run a web page
  - Install Linux server
  - Install Apache Web Server
  - Setup web page
  - Register domain name for the server
- Everything could be hosted on Amazon or Azure
  - The Apache Web Service is one possible option for the first tier – you can set up an account exactly this way, copy your web site to Azure, and then domain name via the Azure domain registry service
- The feature is called “buy a custom domain name” and you access it via the Azure app manager service

Other forms of customization (2)

- Think about coding in JavaScript or Python
- Both are interpreted programming languages, meaning they do not compile to machine code
  - A program called the JavaScript runtime or the Python runtime is launched first, and then it reads your code in, parses it, and this "runs your code"
- But in a sense, your code is not running – the runtime system is running

How do we leverage Python in the Azure cloud?

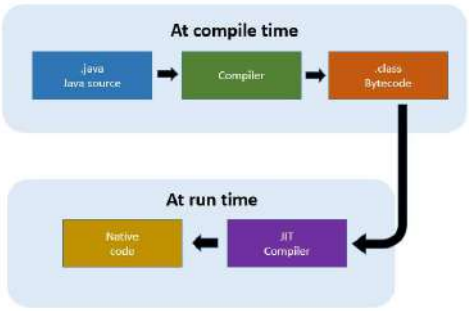
- Most people use a platform called Flask
  - Flask was originally created to let you use Python as a GUI solution (Flask automates behind a flexible app framework where you can drag and drop "widgets" like buttons and image windows)
- Flask has also a built-in framework for talking to the cloud using restful RPC called WSGI
  - WSGI encodes (serializes) messages into web pages
    - Then it uploads the whole web page
    - The reply comes back as a web page, too
  - This makes it easy to write programs that can send data to the cloud, although in fact it is kind of slow if you benchmark carefully
- Flask also supports a form of lightweight container virtualization
  - It allows you to package your entire Python program into a container that is pretty cheap to launch, or to shut down
  - Docker supports this, and Azure and AWS both support Docker
- With this many people create their own first-tier cloud services coded in Python, running in Flask

Customizing a running program

- One issue with Python and JavaScript is that they are slow compared to a compiled program, that maps directly to machine instructions
- But on the other hand, compiling a program is only possible if you know what machine architecture you plan to run on
  - AWS and Google and Azure all have different types of servers, there is no single standard
- It sounds as if you would need to compile one by one, on the specific servers
- Sometimes a preexisting running program provided by the vendor can call into code you supply, in Python, Java, C# or other languages

Customizing Java program

- Java compiles in two steps
  - Compile into a .class file in byte-code form
  - At runtime, a JIT operation does just-in-time compilation and generates machine code



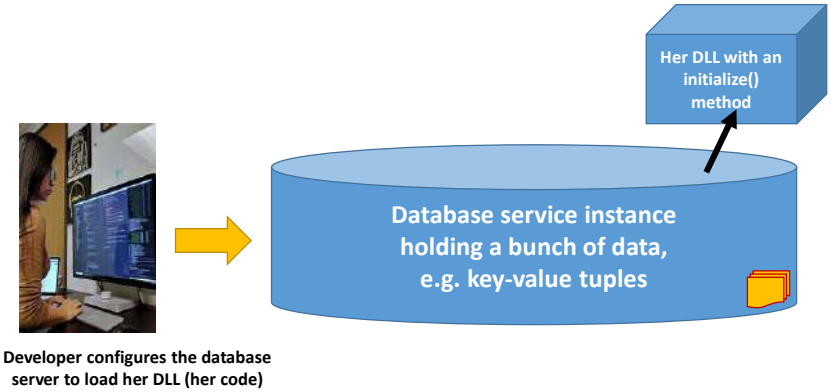
Steps to extensibility of C++

- C++ is considered as the fastest programming language
  - Many projects are using it these days, although there are other options too, but C++ (e.g. current version 23) is widely standard
  - (Of course there is also C, but let's consider, we MUST have a object-orientated design in the project)
- Suppose a program is written in C++ and already compiled, but we want to "extend it"
- First, the C++ program itself needs to be designed to be extensible
  - Many database services can support extensions
  - The extensible program will have some sort of option or command that tells is to "load and initialize" the user's extension code

Write extension code and load it

- Your code would be written as a dynamically loadable library (DLL)
  - This is quite easy and supported in most compiled languages
- Then your DLL can just have a method called initialize(), with void type, taking arguments that can be passed in
- You tell some service, to load the DLL
- The service instance finds the file mycode.dll, and "maps" it into memory (meaning, the code is accessible right in memory)
- Then it looks up the address where initialize() landed
  - And now it can call your initialize() method

Invoking extension code



What does initialize() do?

- The call of initialize() turns around and calls back into database service e.g. to register to **watch** some keys (recall that a key is filename or directory name)
- Now, if anything changes in that file (**put** to the same name), or in that directory (**put** with some extension of that path), the method given in the watch request will receive an upcall from the key-value server

Extensible things in Azure

- In fact, Azure is full of extensible microservices that can be customized this way, in many programming languages
  - All the .NET languages can be used: C#, F#, etc.
- First, you need to identify the specific kind of microservice you will use
- Then, from docs.microsoft.com, you learn about its extension options
  - These are often called "trigger points" or "event triggers"
- Then, from Visual Studio Code, you can write a "trigger function" or "lambda" for that kind of event
  - Each kind of event typically has its own associated metadata
  - The triggered lambda will be passed this metadata as arguments
  - For example, in some key-value storage service a lambda will learn the key that it was watching that caused the upcall

Other microservices has other customization options

- Each kind of microservice offered by a vendor like Microsoft is owned by a developer team, and these teams each make their own decisions
- As a result, there are quite a few places where Azure allows customization by user-supplied lambdas
  - They definitely prefer that these be in C#
- You need to read the documentation or do searches to figure out the options, Azure actually has two options
  - **Lambda**, or
  - **Functions**, that are heavier weight and really run entire programs in containers

Cloud services also support triggered functions

- Cloud database products support a kind of query called a "trigger" that will watch for some condition, then run a lambda or function you supply
  - They call these also "stored procedures"
- Cloud file systems and key-value stores can notify your program or run a function/lambda if the contents of a directory change, or a file is changed
  - A "change" includes modifying files, create, delete, rename
  - Your program will need to decide what changed and what to do, but often the answer is to just redisplay whatever data was in the file

Terminology summary

- Serverless computing
  - A style of computing in which we customize existing platforms rather than talking to big existing servers such as databases
- Function computing
  - Tends to refer to a whole container (docker) that will contain a program, with its own "main", that gets event information from its arguments or from the initial environment
- Lambda computing
  - You attach small lambdas in a language they support, and the platform does upcalls to your code (often via DLL loading)
- Also commonly called as FaaS (Function as a Service) computing

Future? …low code

- There is growing interest in a kind of "drag and drop" customization
- Suppose that some platform has a GUI for customization where the event upcall "points" are visible, and a widget box with pre-created data source widgets, action widgets for standard tasks
- …building a customized microservice could feel like making a PowerPoint presentation + vibe coding
  - Is it an interesting PhD research topic for you?

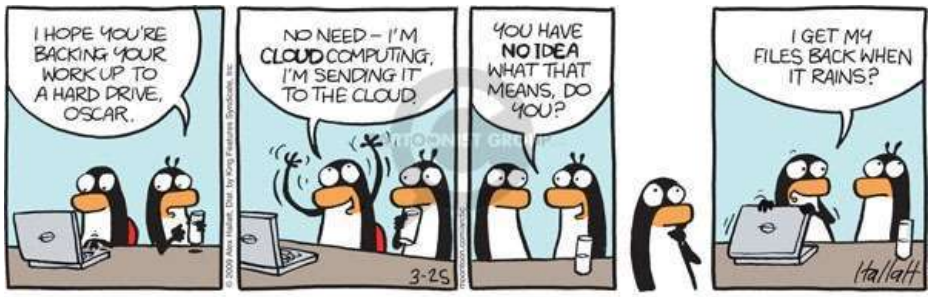
Summary

- Customizing a pre-existing service
- Differences between configuring it (by setting parameters) and adding our own functionality (attaching event handlers, functions, lambdas)
- Why customize?
  - Building an elastic cloud service from the ground up is feasible, but hard
  - Vendors offer standard but very basic microservices, with built-in elasticity, by customizing them, you transform the basic ones into "specialized" microservices

Summary

- Customizing a pre-existing service
- Differences between configuring it (by setting parameters) and adding our own functionality (attaching event handles, functions, lambdas)
- Why customize?
  - Building an elastic cloud service from the ground up is feasible, but hard
  - Vendors offer standard but very basic microservices, with built-in elasticity, by **customizing** them, you transform the basic ones into "specialized" microservices

In other courses, cut-and-paste gets you into trouble ☺ In CS, we require you do cut-and-paste! But document where things came from!



CLOUD SYSTEMS

Storing Complex Data in DHTs

Lecture #5

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

Motivation

- A DHT is a very simple microservice, just offering get/put/watch
- Why is this "enough" for the cloud?
  - Understanding how to put "anything at all" into a DHT for scalability and high performance: DHTs can hold data in memory
  - Limitations and factors the DHT app developer must keep in mind
- Central questions we will focus on
  - How spread out do I want my data to be?
  - How to deal with very large, complex, data structures (with pointers!)
  - What to do about the small but non-zero risk of key collisions

Think of data structures

- You all have worked with lists, trees, perhaps graphs
  - But your experience was with one program on one machine
  - Big cloud data sets can have a similar need – but with a lot of data, spread over a lot of machines
- Even a photo is actually a structure that has an image plus meta-data
- Your challenge is that to store data at huge scale, you want to use key-value products, but it is not so obvious how to "map" from these standard representations to the forms that can match with a put/get API

Simple data structure – a list

- Why lists?
- Usually because the application needs to associate multiple data fields with some item
- For example, a photo has an embedded list of meta-data such as location taken, date, camera settings, etc.
  - Access them via the menu option "properties" for the actual file
  - Initially, the camera itself takes the photo and generates the initial version of these fields
  - That list is baked right into the file holding the photo
  - In fact the file is a bit like an archive (think of JAR files, or ZIP)
    - The photo itself combines the pixel array with a type of JSON file listing attributes

Serialization/deserialization – Structure vs Byte array

- Any object class can optionally be defined to also have a method that will convert the data in the object to a byte vector, and a constructor to create a new instance of the object, given a byte vector
- These byte vectors have a length, which varies, and we can put them in messages, key-value stores, files, etc.
- The storage layer will not know what the serialization format is, so it just remembers that these 72,604 B "are" a "photo object" with some name
- To understand the bytes requires a program compiled using the class definition and its logic for serialization and deserialization

What if cloud wants to add more lists, or create big ones?

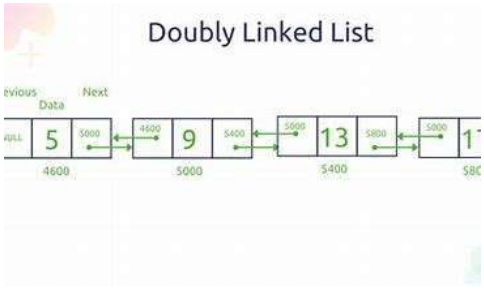
- For example, we might track all the friends of a person, or all the likes for a photo or video
- That list is generated by the cloud, not the camera (ideally, a photo or video should be immutable once we first store it)
  - Because an immutable object will not change, it is easy to cache
  - It can never become stale, but this also means it cannot hold the cloud-provided lists of additional properties, like "liked-by"

How can we store a list to DHT?

- For example, we could have a list that starts out initialized with the same data extracted from the photo meta-data, but then that can be extended with things like "liked by" or "other-photos-of", or "map location"
- This would allow us to leave the original photo object unmodified
- We learned that immutable objects are convenient because they are easy to cache and never become stale
- **So, by separating the list from the photo, we gain this form of scalable flexibility**

List in DHT

- Easy to just store the list "inside" the photo, give each node a unique id
- Save each node on the list using (node-id, obj) representation
  - The object is just the byte vector obtained by serializing the node
- Instead of memory pointers, just have the next and previous fields contain node-ids for the next node and the previous node



Think about costs

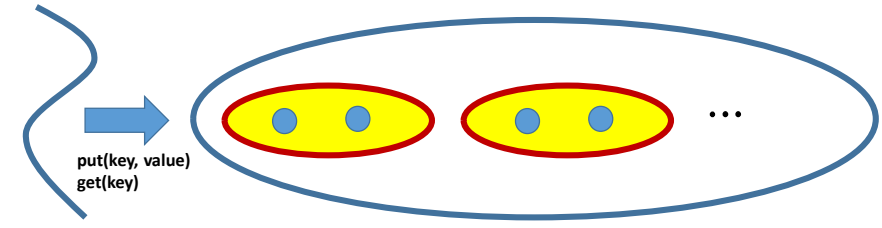
- For small numbers of objects, just calling get a few times (maybe in parallel) is a reasonable option
  - Datacenter networks are fast
- But if we form a really big list it gets very long and might be costly even to count the length of the list
  - Like if we had a list of "likes" for a famous video of an iconic song
- And other operations could be even more expensive

DHTs are shared, but it means the big list will be shared too

- What we already know about DHT?

- The DHT idea can be traced back to work by people at Google, and to papers like the Jim Gray paper on scalability
- We take some service and structure it into shards: sub-services with the identical API but handling disjoint subsets of the data
- Given a key and a value, the DHT will map the key to some shard and save the tuple at the member (or replicate it over the members)

DHT overview



A DHT assumes that the data-center network is very fast, 10 µs delays for a gRPC are typical – whereas the client application has 100 ms to send a reply back to the human end-user (customer)

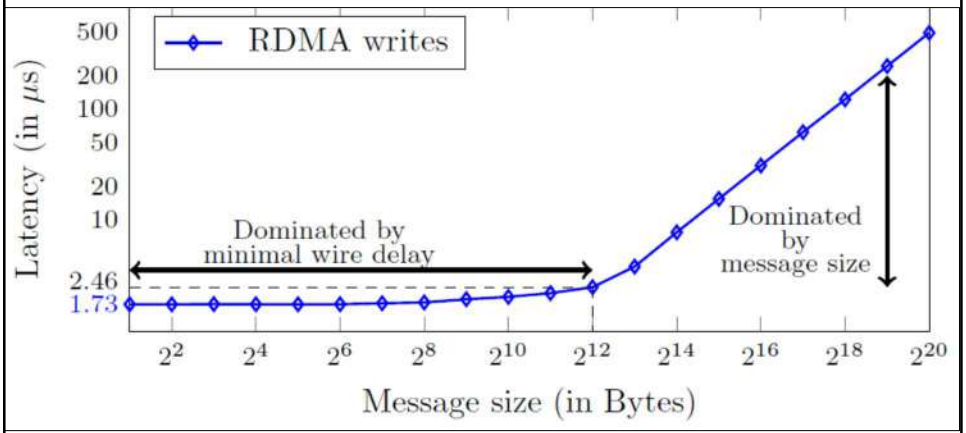
We can even treat them as databases, and this is important!

- The idea is called NoSQL
- Any list can be treated as a kind of database
  - As an example, I could "code" this in a few lines in C++, C#, Python, Java
    - From my list of friends, find friends who are currently within walking distance from me, not busy in a class now, and like pizza
  - We will look at the SQL notation used in language settings later

Concerns we have to think about

- When we create a very big list, how will it be accessed?
- Individual put/get operations should have $O(1)$ cost
- But that query would need to search for friends
  - Will it do a sequence of operations, one per node?
    - That could become very expensive
  - In fact for speed, we clearly need a way to do parallel search
  - And we probably want to look at many nodes per "operation"
- Networks operate in terms of "packets" with a maximum size (usually 4 kB, 8 kB, or 64 kB)
  - Smaller objects can be inefficient because the network packets are not full
- The first packet you request typically incurs one "delay" to send the request and one for the reply
  - But subsequent packets show up quickly in a chain, so we pay this first delay once, then additional cost as a function of total data size we transfer

Network speed / delay

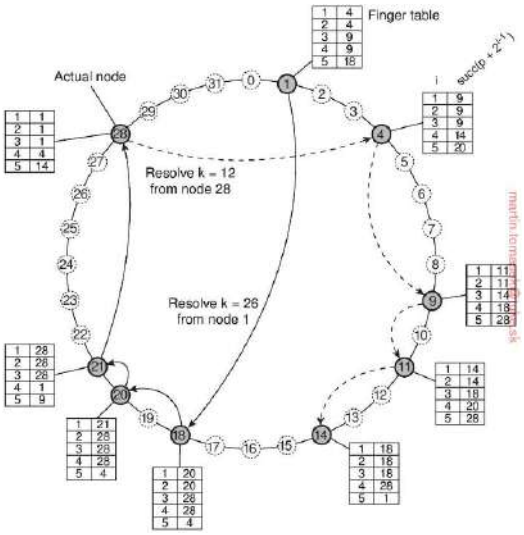


How can we use this information

- When storing individual objects like photos it might already be useful
- But cloud solution architects use concrete data even more when they think about complex operations, like that query searching for friends to get pizza
- Understanding how a NoSQL query might be implemented lets us visualize the data access patterns that would be used, which in turn can let us design the data stored in the DHT in a way that matches nicely

Example: Chord system from MIT for Akamai

- $FT_p[i] = succ(p + 2^{i-1})$
- $q = \begin{cases} FT_p[j] \le k < FT_p[j+1] \\ \\ FT_p[1] \text{ when } p < k < FT_p[1] \end{cases}$
- Lookup require $O(\log n)$ steps



Beehive uses Chord and range searches

- Beehive was one of a few DHTs that used unhashed keys instead of hashing them
  - It also varied the replication level, to make popular keys easier to find
- The benefit of not hashing the key is that data remains in sort order along the ring
- A disadvantage is that Beehive required very active placement management

Chord did not "take off"

- The problem turns out to be that in a modern cloud, SQL range searches are rare compared to fetching individual objects
- So, making a single lookup into an $O(\log n)$ operation to make range searching easier is not a popular concept
- Chord is used by the Akamai CDN, but it is not in wide use elsewhere

How do real cloud do it?

- What other options do we have?
- Parallel search on chunks of data is a popular tradeoff
- Group key-value objects so that code can access many of them in one get operation
- Then design the language-layer NoSQL API to efficiently perform parallel operations on chunks

- **A concept called MapReduce**

Collocation: Risk of growing chunks

- For example, in a list we could group neighboring nodes so that when we access one, we also get its neighbors
- Design a chunk size efficient for the network
- This leads to a two-stage list design
  - Each *C* successive nodes will be held in one chunk
  - The chunk gets its own key and has the successor and previous pointers, but the *C* values are in an array
- More tradeoffs
  - A single get will tend to fetch the whole chunk
  - Putting a new value into the middle of the list would be costly, and probably should not be supported (the chunk might get too big)
    - At best we would let the chunk grow, then split it
  - But now we could potentially scan multiple chunks in parallel

Concurrency: Risk of conflicting updates

- For a shared list that collocates lots of data in larger objects, we might have many programs simultaneously doing get operations, which is great, because a key-value DHT is made for that
- But we could also have a few concurrent inserts or delete operations (put)
- We would use versioned updates, since locks risk (Jim Gray's issue)

Notice all the tradoffs

- How long will our lists be?
- How big will the objects (and the network transfers) be?
- How often will updates conflict, and need versioned update operations?
- How many shards should we maintain?
- Would we be better off using a Chord-like approach?

Is there a singe best solution? No!

- Different applications need their own different mappings to the DHT model, for optimal efficiency
- You are expected to just use a white board, estimate the percentage of read-only queries versus updates, and design a solution that spreads the loads and should not have a lot of collisions between get operations
- Most likely this analysis will be part of your decision process for figuring out how many nodes to collocate in one object on one shard

CAP + BASE again?

- Eric Brewer's rule (CAP) – **relax consistency, use AP**
- Yes, sometimes web pages might be wrong
  - This is fine if it is rare
- The goal is to convince yourself that it will be rare, have the underlying system be basically consistent, and have it converged (self-repair) if an inconsistency does arise (BASE methodology)

What could happen in the application?

- **404 Not Found**

- The error message could be caused by not finding data in a DHT
  - They probably stored their content in the DHT, but somehow got an error when trying to fetch this content to build the web page
  - It could be an example of CAP: When a DHT resizes elastically, sometimes it makes errors for a few seconds afterwards
  - The data must have been changing when I accessed it
- So, my web page builder in the first tier of the cloud saw inconsistent data
- And it trusts CAP, so it still built "something" but not a very useful page
- I refresh, and then it works
  - This is BASE => CAP in action

Why would it have been changing?

- Perhaps the shard was still in the process of updating but I already tried to query it
- Perhaps this DHT happened to have a crash and needed a moment to repair itself and recover
- Perhaps elastic resizing was occurring, and my data was temporarily on the wrong shard

Suitable genuine unique DHT keys

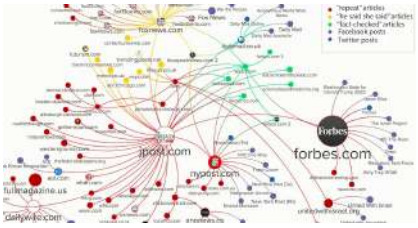
- You need a unique name for the objects you are storing
- For example: Peter's friend is named Susan. But "Susan" is not a unique name. The DHT could have some other object with that name too
- On the other hand, "/users/Peter/friends/Susan" is a unique key, and we can hash it with SHA64 or SHA128 into a unique integer
- Clouds provide a genuinely unique key
  - Microsoft and AWS both have "registry" services
  - The resulting keys will have no obvious meaning to a human user, but is unique
  - These are for objects we will fetch "algorithmically", not for external programs to use directly

Could keys collide? No!

- Most cloud systems favor large keys, like 64 bits
- Key generators use a variety of tricks to make sure that they will not give out the same key twice (UUID, GUID, etc.)
- For example, the key could include the name of the server that generated the key, which ensures there cannot be a tie

Summary

- Cloud data forms a huge, complicated graph
  - We will see in some future lecture



- Almost anything can be stored into a DHT
  - The cloud does this
  - But it is not free: you need to be clever to encode your application into a DHT
- Think about keys, object sizes, access patterns, costs
  - Be wary of "leakage" (neglecting to delete temporary data) or you will get a BIG monthly bill

CLOUD SYSTEMS

Accessing Collections from Modern Programming Languages

Lecture #6

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

Databases

- Database systems offer the abstraction of "one big repository"
- Data lives in "relational tables"
  - Nodes in an entity-relations graph tied together by key-foreign key edges
- A database uses SQL to express read-only queries and updates
- Data is often too big to fit in memory – disk I/O is a big consideration
- ACID concept – transactional model
  - Transactions run concurrently, using locking to avoid conflicts and 2-phase commit at the end

Can database be sharded?

- Yes, and every major SQL product automatically does this sort of thing
  - But often, not into totally independent mini-databases
- Sharding of relations (tables) is common, and enables modern SQL databases to hold really huge amounts of data in a single relation
  - Big clusters
- The scalability of concurrent SQL computing is limited by locking conflicts, timeouts, abort/rollback and retry
  - Jim Gray's issue

Key-value storage

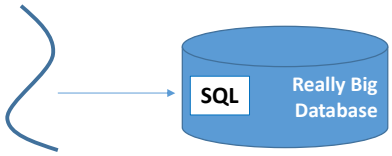
- Key-value DHTs offer the abstraction of "many pools of memory"
  - A pool of memory is called a shard, owned by one or more servers
- By design, we have enough shards so that data would normally fit in memory, allowing $O(1)$ access
  - Supports get/put/watch API
- Clients issue requests concurrently, but a server normally handles them one by one, enabling it to offer atomicity without needing locking
- Shards implement state machine replication for put
  - For get you just send an RPC to any random member of the shard

With big data, key-value store is like a collection of mini-databases

- We might have gigabytes of data in each shard
  - An Intel server can host 64 GB of memory, and we could fill this completely
- A new generation of "persistent" memory will expand this limit to 256 GB soon, and even more down the road, still with DRAM access speed
- Are they really databases?
  - These mini-databases offer $O(1)$ in-memory speed
    - We hash to figure out which shard, then use an $O(1)$ lookup structure to find the actual object in this massive memory region
  - Does it make sense to think of a get as doing something more elaborate than just finding the (key, value) tuple and sending it back to the caller?

Where does a program run?

- With SQL, you give your program (transaction) to the SQL database
- It compiles and executes the SQL code (think of the database system as a form of interpreter)
  - Even if the SQL code was submitted from a client program, the SQL execution occurs inside the database
- So, the program is run by the database on your behalf, and it can even send tasks to "workers" if the database is spread over multiple machines

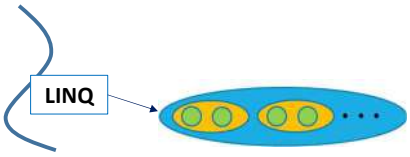


LINQ: SQL embedding into code

- LINQ is the Language Integrated Query package from Microsoft
- Allows a program to establish a binding (connection) to services like CosmosDB, BLOB store, Databricks, etc.
  - These connectors need to support the concept of "iteration" over a collection (list) of tuples
  - If your objects have a regular structure, this is exactly like scanning a database – and we can do SQL queries
    - LINQ offers this model
  - But those queries will run in just one server

Where does program run?

- LINQ is part of some client program, and run entirely inside the client
- The key-value store is "accessed" purely via put and get
  - Any data the LINQ logic "looks at" must be fetched tuple by tuple
- To scan a lot of data items, must fetch them first, one by one



- For example, experimental DHTs like Cascade can run LINQ on a server directly
  - Similar to MapReduce concept we will see later

LINQ is like a software library

- It is used by a program as it executes, and lets the program treat lists and other collections as in-memory "relational tables"
- Most programmers do know SQL and find it powerful and intuitive
  - LINQ enables you to use SQL on this internal data structures, too
- Same remark applies to Pandas, PyTorch, Tensor Flow, Spark
  - All of them are drawn to the SQL model because it is convenient

Can we really think of a shard as a mini-database?

- Sometimes, for a key-value store, this might be a sensible thing to do
  - If the key-value store holds multiple kinds of data, you could first filter to focus on "one table"
- With LINQ we can run any kind of database-like logic on the local shard, and it can even use get to fetch some data from remote shards
  - Best to cache that remote data, if you do this

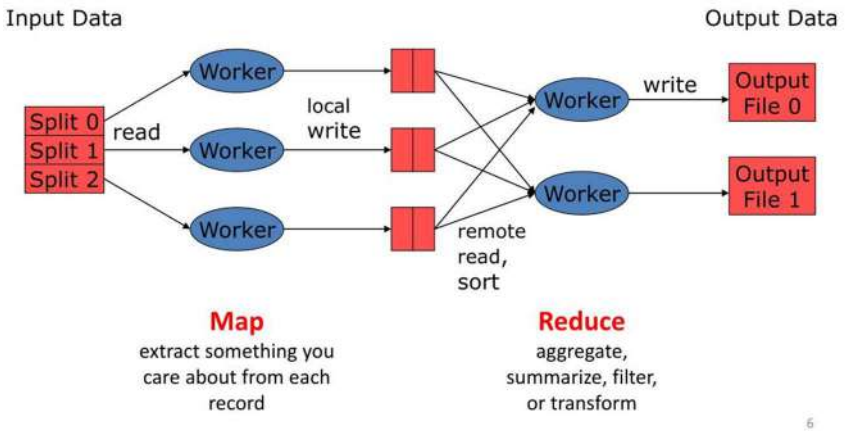
Staying in the spirit of key-value model

- Cloud builders do not view a key-value sharded store as a true database
  - They keep the sharding policy in mind when they mentally visualize their system, and think of NoSQL as a convenient way to express data transformations and filtering
  - Fit is great for big collections, social network graphs stored as key-value data
  - But not for a genuine relational database split into lots of shards
- With LINQ triggered on multiple shards, the developer is leveraging parallelism
  - Concurrency rules the cloud: "embarrassing parallelism" when searching or transforming the entire big-data collection
- Our goal is really fast individual get operations, parallel search/transformation

The MapReduce pattern

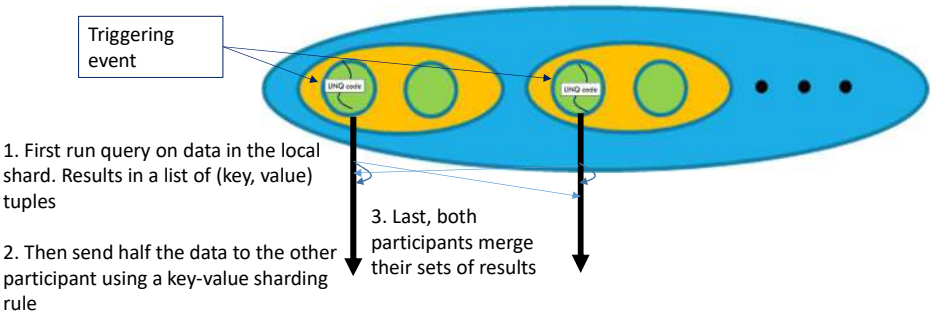
- Google was first to explore this idea of running the same code but on sharded data, doing purely local data access in parallel
- They had huge collections of web pages and needed to run algorithms to decide which web sites to return when a user issues a search
  - Their developers quickly ran into a problem: parallel computing on shards is easy, but sometimes we need a way to merge the subresults
  - All at one place? But this could overwhelm that server
  - Keep results sharded? They favored this, but it needs a special coding style

MapReduce workflow



MapReduce on key-value data

- MapReduce with a simple shuffle, group-by, and reduce step
  - We will revisit this in full detail in some future lecture



MapReduce summary

- Map – Request triggers computation by one member of each shard
- Compute – Each runs code locally, processes their local "share" of the data
- Shuffle – Output is in the form of (key, value) tuples. Hash the keys to figure out which to keep and which to send to some other shard
- Reduce – Collect all the incoming data, group it by key, then combine ("reduce") the values so that for each key, you keep just one result

When to use MapReduce

- Easiest if the computation is embarrassingly parallel
  - Apply some function to every data item, group results by a shared key, combine the subresults, and leave all of it sharded
- Much harder for tasks that are inherently entangled
  - For example, a MapReduce version of binsort/mergesort must estimate bin sizes, and that turns out to be tricky – yet sorting individual bins and merging is easy

Concept: The NoSQL model

- NoSQL is a term for any system that supports SQL but really operates on sharded data
  - Examples: AWS DynamoDB, Azure CosmosDB, Cassandra, DB Rocks, MemCacheDB
- NoSQL inherently has limitations
  - These SQL queries are not true SQL on a single database, they do not have full transactional semantics (no ACID), they cannot combine data from multiple shards, no locking occurs
- With put, we normally just send the update to both replicas via a state machine replication protocol
  - They can then perform the identical changes

The LINQ operates data in memory

- **In your own address space**

- With an SQL model, the program describes a transaction on the full database, which is "out there"
  - The database itself plans an execution strategy
- With NoSQL from LINQ or similar packages, you fetch data into your address space using get (or perhaps some form of multi-get to pull a batch of content all at once), then process it in-memory inside your program
  - Your LINQ expression determines the execution strategy

LINQ do not need locking

- Normal SQL needs locks to deal with concurrency, like if we had multiple threads each running SQL code at the same time
- In the LINQ-style NoSQL model, our code is generally not concurrent (not multithreaded) and does the entire operation in one shot
- No locks are needed!
  - Jim Gray's concerns would not apply in this situation

LINQ requires structured data

- It is common to say that cloud data is structured or unstructured
- Unstructured data means web pages, photos, or other kinds of content that is not organized into some kind of table
- Structured data means "a table" with a regular structure, or a list of items in a similar format such as (key, value) tuples in a collection
- When we first introduced key-value storage we talked about storing pretty much anything into a DHT
  - We would consider most data to be unstructured
- But there are many tools to convert unstructured data to structured data
  - For example, given a web page, we could extract all the *n*-grams, or all the URLs we find in it
  - Given a product documents, we could extract one-row summaries using an OCR and then a parser

A collection concept

- A collection is any kind of list of data that has some form of key for each item
  - The value could be a simple value like a number, or a table
- Unlike in cloud storage, collections are a programming concept used inside your code
  - So, the value can also be any form of object, or even another collection
- Now you can think about code that iterates over the (key, value) pairs and even does database-style operations on them

Iterators

- Well-known design pattern for you
- Modern object-oriented languages allow for loops to scan collections, or subsets of them
  - The scan will be in the order of the collection itself
- This is done using an "iterator" object
  - Often the syntax hides the object from you if you just plan to scan the entire collection
- An iterator object represents some portion of the collection
  - It has a begin point, a next operator, and an end point: first(), next(), hasNext()

Many cloud storage layers provide iterators as connectors

- Suppose you have data in a cloud DHT, database, file system, etc.
- You can generally access your data by
  - Opening the storage system using a library method that returns an iterator
    - By default, it will iterate over all of your content in the service
  - Then you can apply a filter to "focus" the iterator on just certain items
- A filter will select certain items, but skip others
- For example, Java Streams which are well-known to you

LINQ examples

- Code is very "succinct"
- Lots of use of lambdas
- Looks just like SQL
- Data could be entirely local, or we could use get to fetch data from a mix of the local shard and remote shards

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words
    .GroupBy(w => w.Length, w => w.ToUpper())
    .Select(g => new { Length = g.Key, Words = g })
    .OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine("Words of length {0}:", obj.Length);
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}
```


A sampling of LINQ operators

Filters and reorders:
where(predicate), where_i(predicate)
take(count), takeWhile(predicate), takeWhile_i(predicate)
skip(count), skipWhile(predicate), skipWhile_i(predicate)
orderBy(), orderBy(transform)
distinct(), distinct(transform)
append(items), prepend(items)
concat(linq)
reverse()
cast()

Transformers:
select(transform), select_i(transform)
groupBy(transform)
selectMany(transfom)

Bits and Bytes:
bytes(ByteDirection?)
unbytes(ByteDirection?)
bits(BitsDirection?, BytesDirection?)
unbits(BitsDirection?, BytesDirection?)

Aggregators:
all(), all(predicate)
any(), any(lambda)
sum(), sum(), sum(lambda)
avg(), avg(), avg(lambda)
min(), min(lambda)
max(), max(lambda)
count(), count(value), count(predicate)
contains(value)
elementAt(index)
first(), first(filter), firstOrDefault(), firstOrDefault(filter)
last(), last(filter), lastOrDefault(), lastOrDefault(filter)
toStdSet(), toStdList(), toStdDeque(), toStdVector()

Fancy stuff:
gz(), ungz(), leftJoin, rightJoin, crossJoin, fullJoin

Joins exist too

- The confusing thing to think about is how a JOIN will work if we have sharded data, run your LINQ queries concurrently in each shard, and each query is accessing just the local data
- We would be treating each shard as a distinct mini-database
- Suppose that we have one table and we fully replicate it across all the key-value servers
- Every shard now has a copy of the table
- In this case a local LINQ query could still compute the entire JOIN, in pieces
  - Each shard would generate one piece of the full JOIN result

More missing things

- An aggregation query like SUM or AVG will compute the local answer but not the global one
- Here we need to send our partial results to one node, to combine them
- Easy to do, but only if you can visualize the pattern you are requesting

Required steps

- Application must be expressed in a way where NoSQL is sufficient
  - There is no automatic transformation from SQL on a full database to a sharded set of NoSQL actions on local portions of the data
  - If you cannot shard the task and write it with local SQL logic, you will not be able to use this method
- Application also requires a binding to the data source, such as the file system or perhaps the key-value store
  - This is similar to saying "to read a file, first the application must know the file name and be able to open it"
    - The difference is that a binding connects to a service that could be sharded, whereas a file is a single thing in a file system or key-value store

Each cloud has its own way of binding

- In some cloud systems bindings are very static, but clouds, like Azure and AWS, allow you to express a binding to a service that might not be running
- The "app service" would then launch your required service on demand, but startup could take 30 s or more for a complex service
  - Pre-binding is important if you care about performance
- Once launched, you would want the service to remain active for a while

Missing values

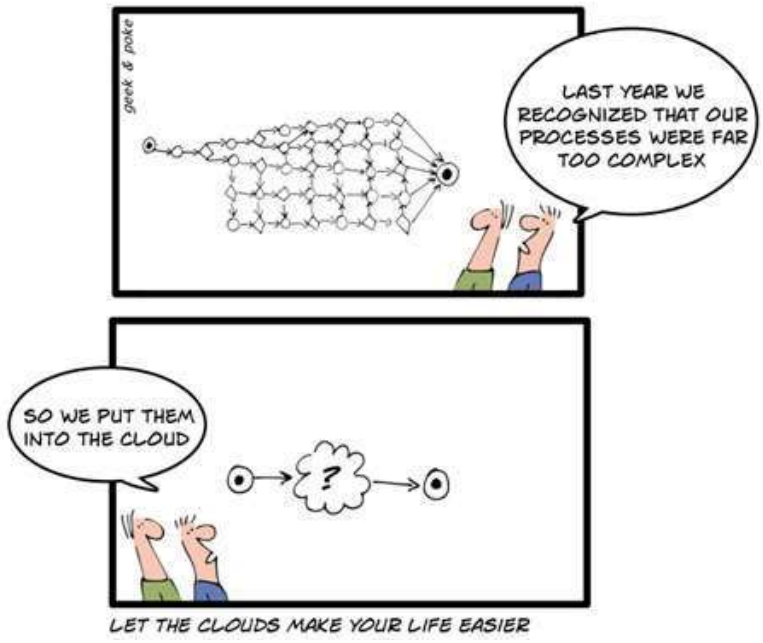
- Unstructured data converts to structured data but with gaps
  - Most kinds of objects are nullable – a null represents a gap
- This means that null is a legal value, and can be used for missing data
  - Assumes that you understand the semantics of LINQ with null values
- Some databases instead have a default value for missing data, like –99

Recommended links

- LINQ overview – .NET | Microsoft Docs
  - https://docs.microsoft.com/en-us/dotnet/standard/linq
- Complete list of all LINQ operators
  - https://dotnettutorials.net/lesson/linq-operators
- Write LINQ queries in C# | Microsoft Docs
  - https://docs.microsoft.com/en-us/dotnet/csharp/linq/write-linq-queries
- Pandas for Python
  - https://pandas.pydata.org/pandas-docs/stable/getting_started

Summary

- In today's cloud platforms, data is big and often sharded all the time
- Tools like Pandas, PyTorch and LINQ make it easy to compute on this data, especially if it has a regular structure
- Requires new thinking about what it means for data to be a database
  - LINQ and other NoSQL (perhaps + MapReduce) are very useful



CLOUD SYSTEMS

Design Steps for Cloud Applications Using Patterns

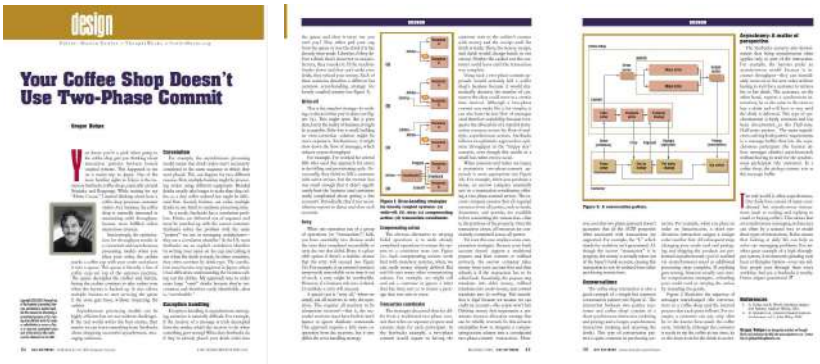
Extra to lecture #7

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

Inspiration

- HOHPE, G.: *Your Coffee Shop Doesn't Use Two-Phase Commit*. IEEE Software, Vol. 22, No. 2, 2005
  - https://doi.org/10.1109/MS.2005.52

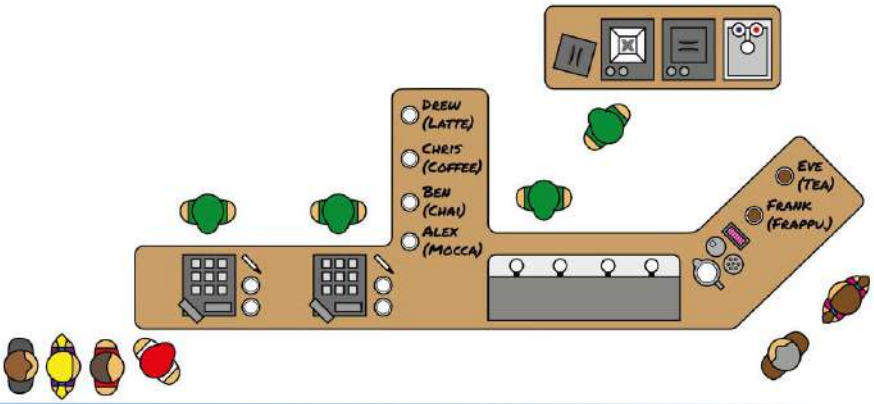


Methodology

- FEHLING, C. – LEYMANN, F. – RETTER, R. – SCHUPECK, W. – ARBITER, P.: *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer-Verlag Wien, 2014
  - https://link.springer.com/book/10.1007/978-3-7091-1568-8
  - https://www.cloudcomputingpatterns.org

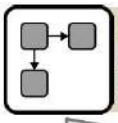
Coffee shop

Decomposition → Workload → Data (state) → Refinement of components → Elasticity and resiliency



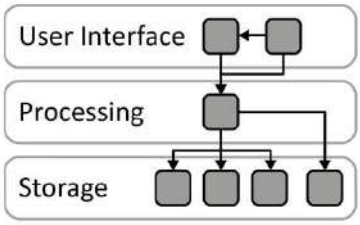
Decomposition → Workload → Data (state) → Refinement of components → Elasticity and resiliency

How to distribute application functionality?

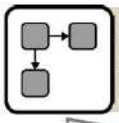


Distributed Application
A cloud application divides provided functionality among multiple application components that can be scaled out independently.

Layer-based Decomposition

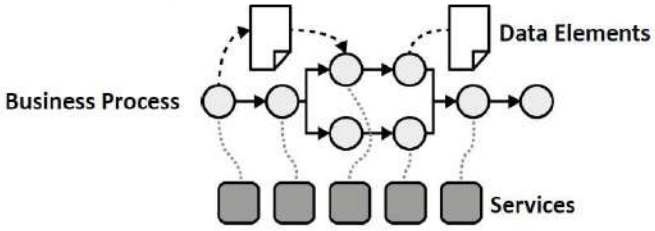


Components reside on separate functional layers
- Often: user interface, processing, storage
- Access is only allowed to **same layer and the layer below**
- →Dependencies between layers and interfaces are controlled

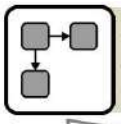


Distributed Application
A cloud application divides provided functionality among multiple application components that can be scaled out independently.

Process-based Decomposition

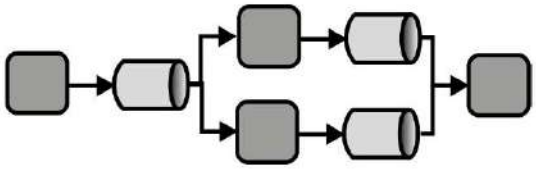


Business process model determines decomposition
- **Activities**: tasks executed in a specific order (**control flow**)
- **Data elements**: information handled by activities (**data flow**)
- Functional application components (**services**) are accessed by process



Distributed Application
A cloud application divides provided functionality among multiple application components that can be scaled out independently.

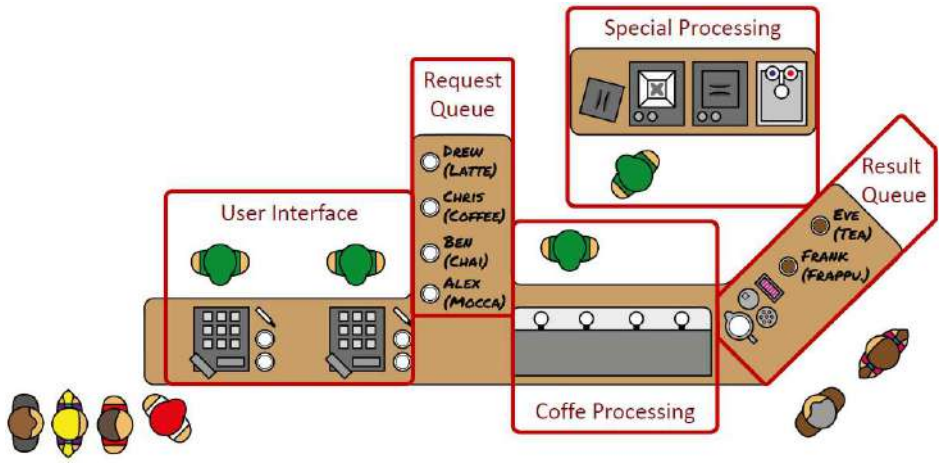
Pipes-and-Filters-based Decomposition



Decomposition based on the data processing function
- **Filter**: application component processing data
- **Pipe**: connection between filters (commonly messaging)

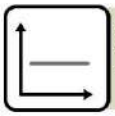
Coffee shop – Decomposition of functions

- Identify functional components



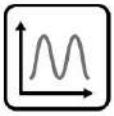
Decomposition → **Workload** → Data (state) → Refinement of components → Elasticity and resiliency

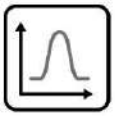
What workload do components experience?

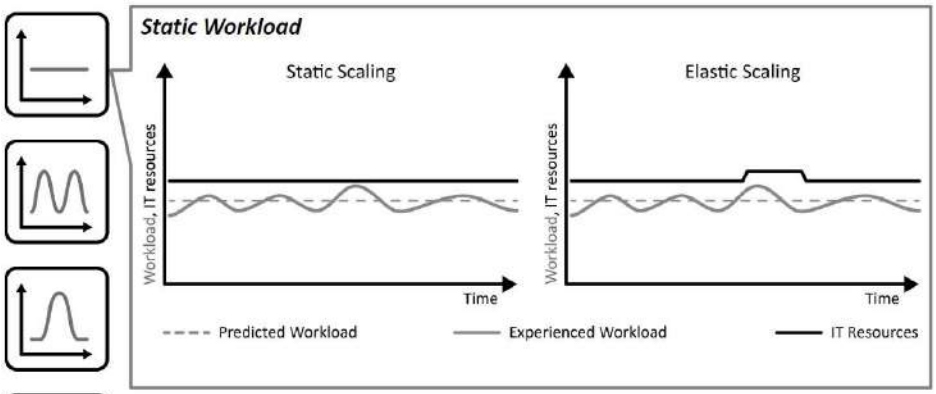


Static Workload

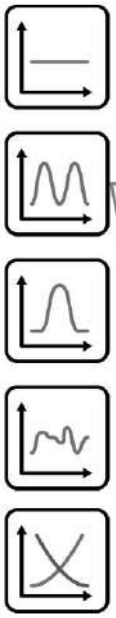
IT resources with an equal utilization over time experience static workload.



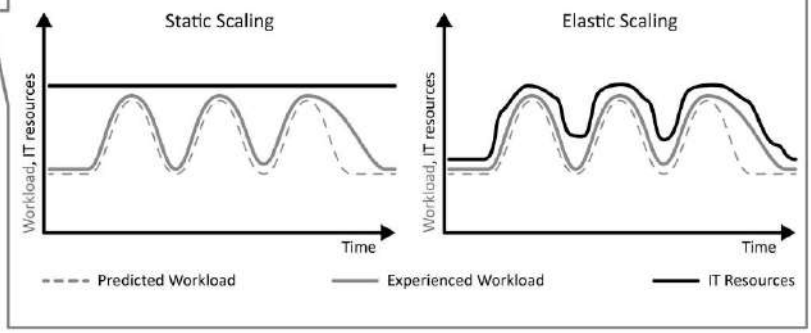


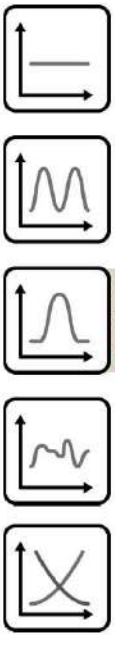


Periodic Workload
IT resources with a peaking utilization at reoccurring time intervals experience periodic workload.



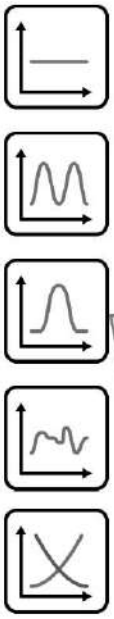
Periodic Workload

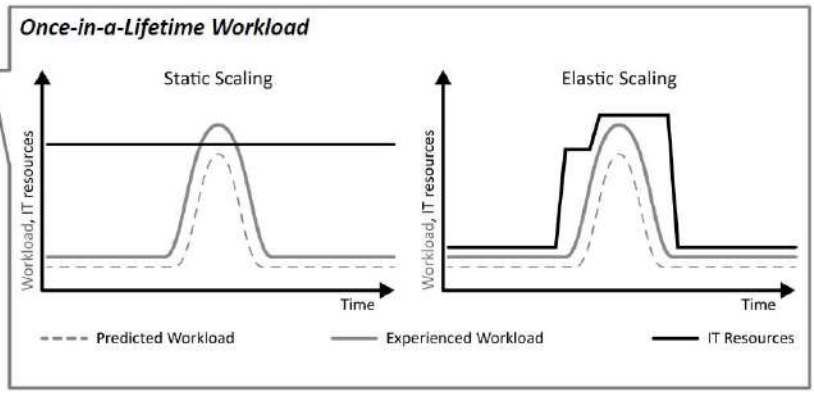


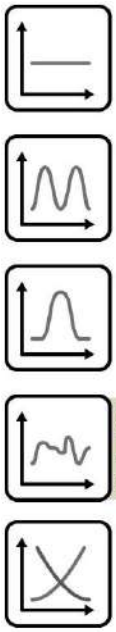


Once-in-a-Lifetime Workload

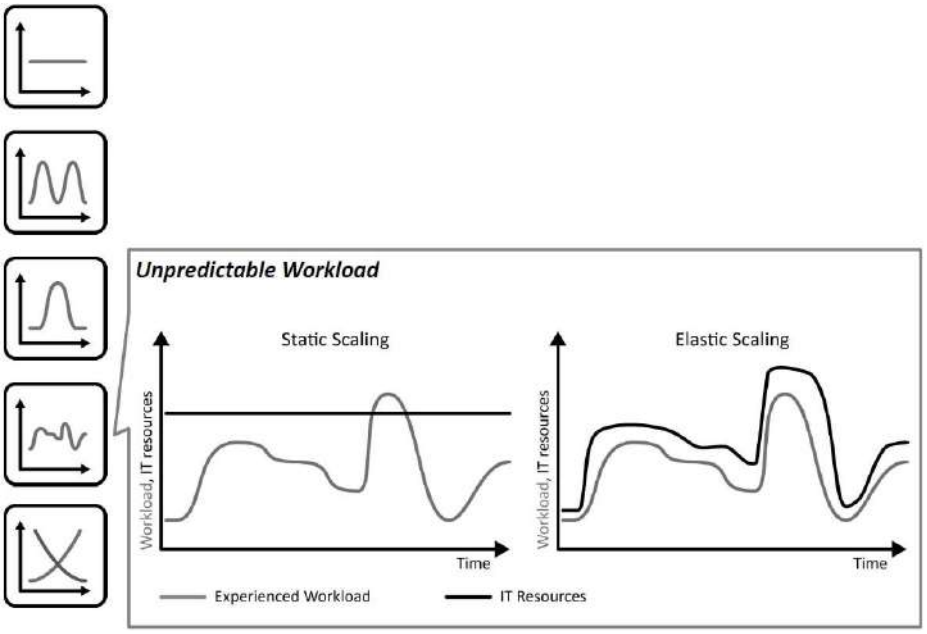
IT resources with an equal utilization over time disturbed by a strong peak occurring only once experience once-in-a-lifetime workload.

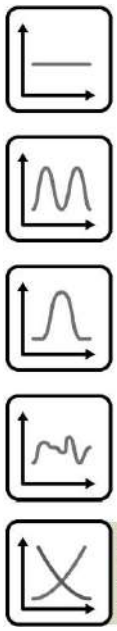






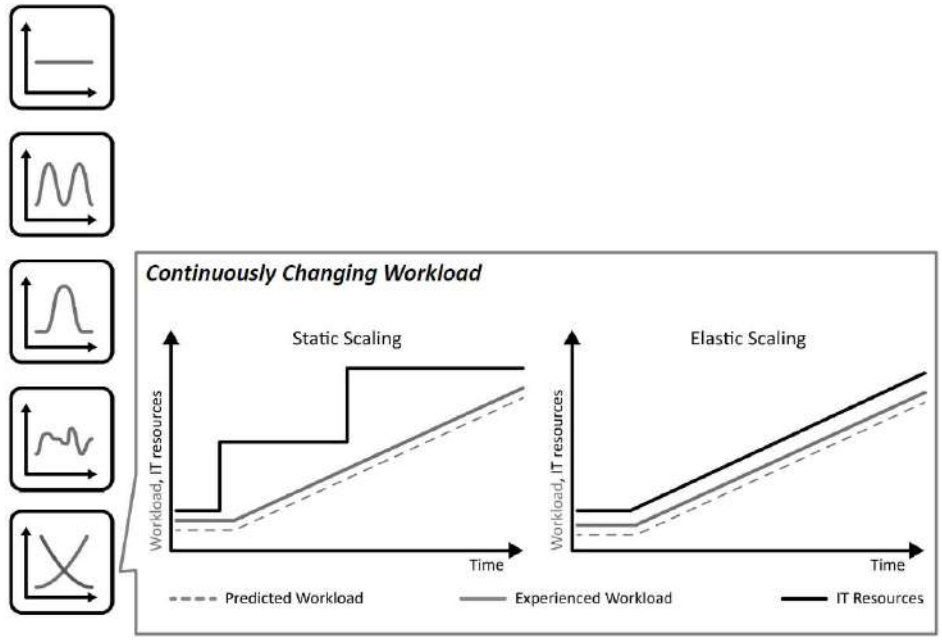
Unpredictable Workload
IT resources with a random and unforeseeable utilization over time experience unpredictable workload.





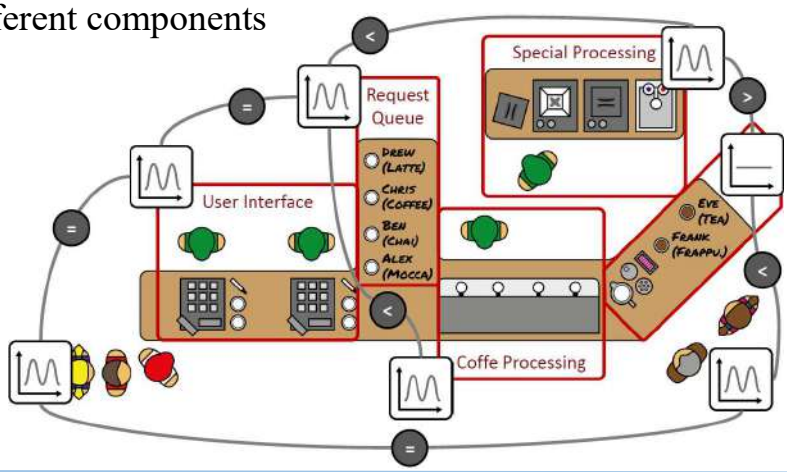
Continuously Changing Workload

IT resources with a utilization that grows or shrinks constantly over time experience continuously changing workload.



Coffee shop – Workloads

- Identify and compare workload generated by user groups at different components



Workload – Lessons learned

- Workload can differ **significantly** between components
- Scaling them as a holistic unit can be very inefficient

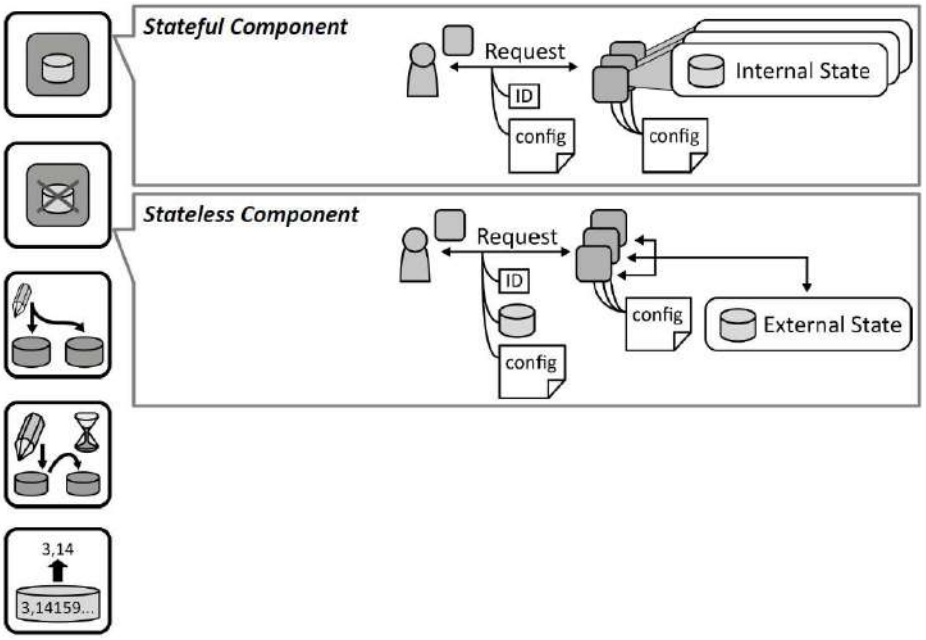
Where does the application handle state?

Notion of state

- We differentiate between
- **Session state**
  - State of a client's interaction with an application
  - Commonly referred to when discussing *statelessness*
  - Example: customer's shopping card of an online store
- **Application state**
  - Data handled by an application
  - We extend *statelessness* to also incorporate application state
  - Example: customer's shipping information stored by an application

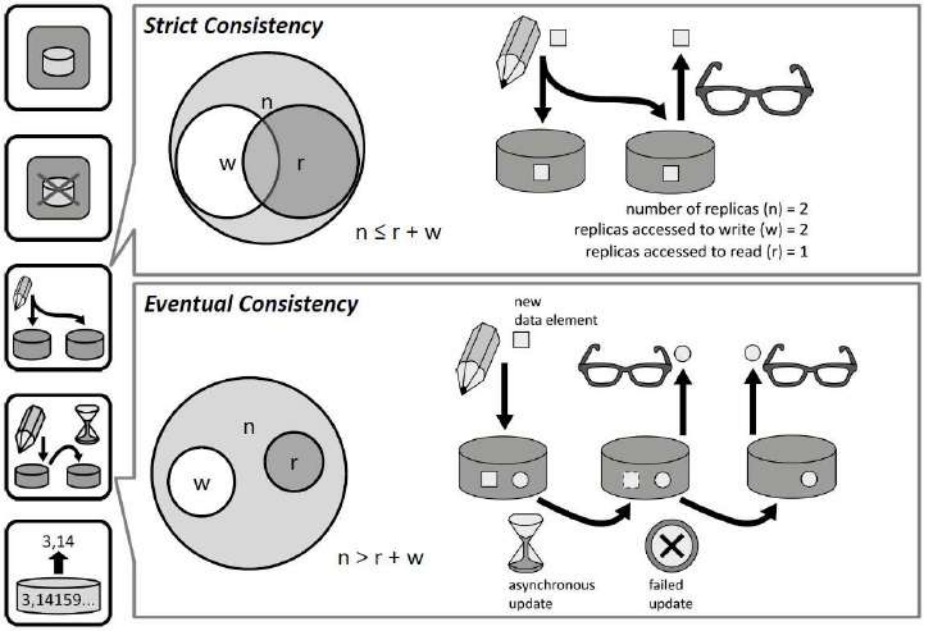
Stateful Component
Multiple instances of a scaled-out application component synchronize their internal state to provide a unified behavior.

Stateless Component
State is handled external of application components to ease their scaling-out and to make the application more tolerant to component failures.



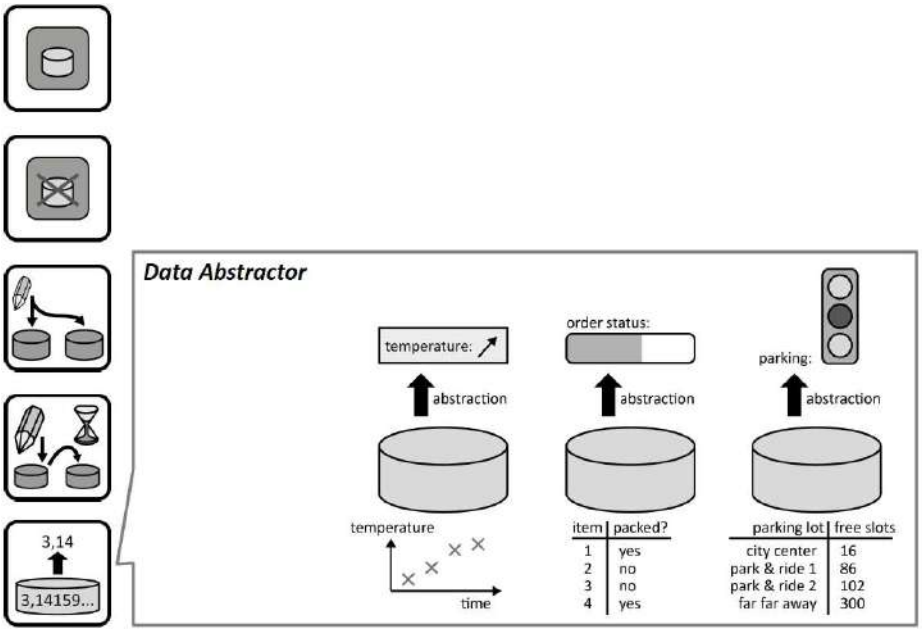
Strict Consistency
Data is stored at different locations to improve response time and to avoid data loss in case of failures while consistency of replicas is ensured at all times.

Eventual Consistency
Performance and availability of data in case of network partitioning are enabled by ensuring data consistency eventually and not at all times.



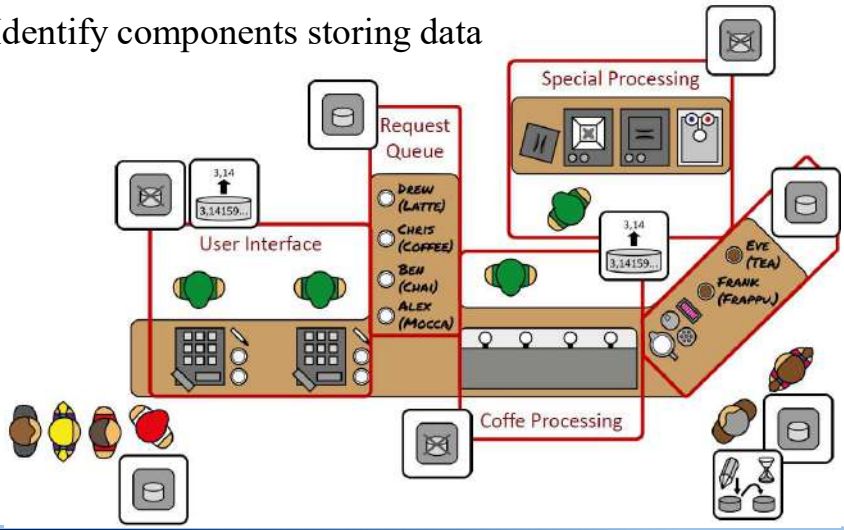
Data Abstractor

Data is abstracted to inherently support eventually consistent data storage through the use of abstractions and approximations.



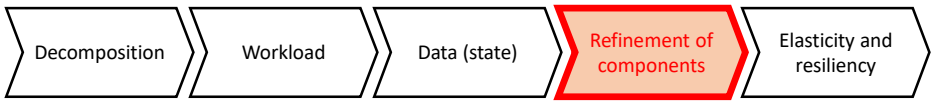
Coffee shop – Data

- Identify components storing data

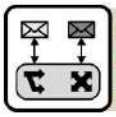


Data – Lessons learned

- State should not be handled by components whenever possible
- Handle session state in
  - Requests (has to be provided with every request)
  - Provider-supplied storage and communication offerings
- *Lie* about state whenever possible/acceptable

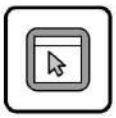


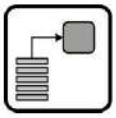
How are components implemented?

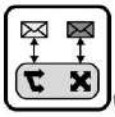


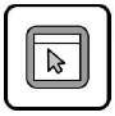
Message-oriented Middleware

Asynchronous communication is provided while hiding complexity of addressing, routing, or data formats to make interaction robust and flexible.

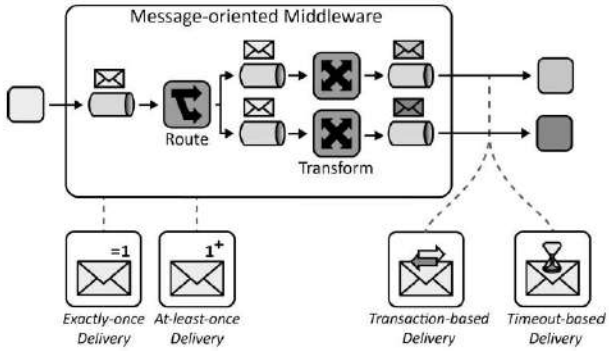








Message-oriented Middleware



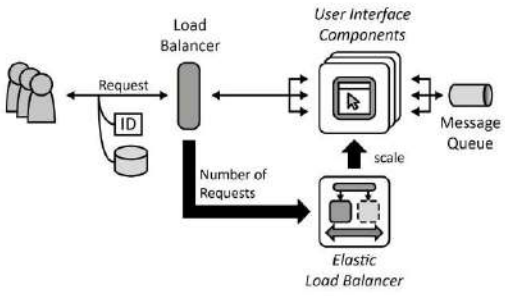
Assumptions of communication partners are reduced (***Loose Coupling***)

Platform: implementation language used
Reference: location of the communication partner (routing)
Time: communication partners are active at different time / speed
Format: message formats can change (transformation)

User Interface Component

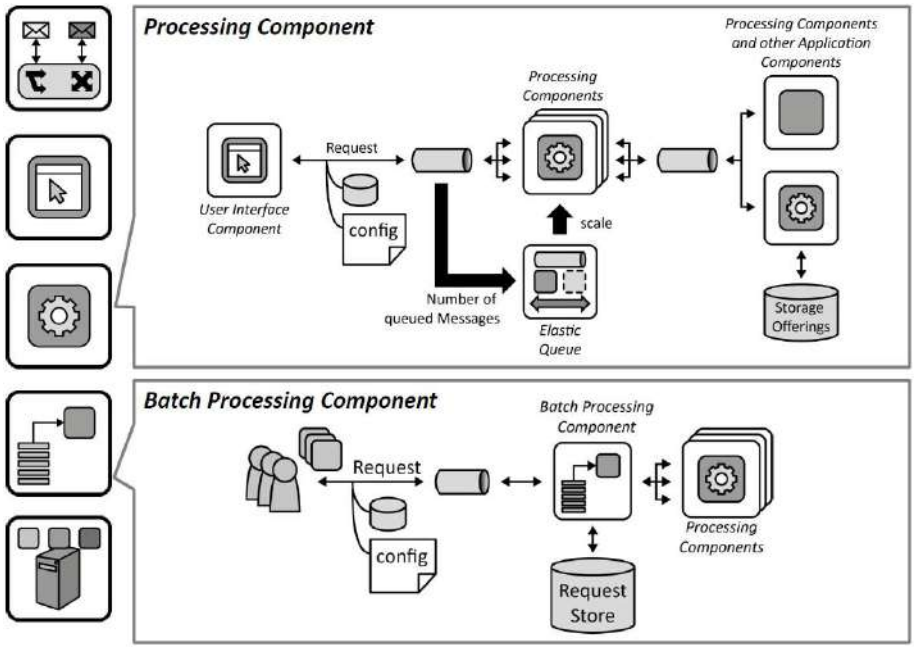
Synchronous user interfaces are accessed by humans, while application-internal interaction is realized asynchronously to ensure loose coupling.

User Interface Component



Processing Component
Processing functionality is handled by elastically scaled components.

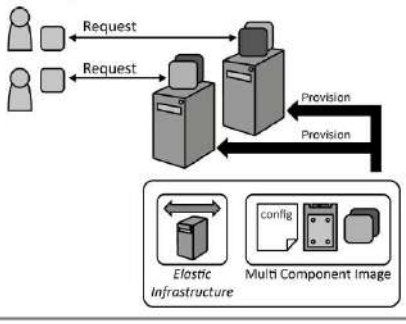
Batch Processing Component
Requests are delayed until environmental conditions make their processing feasible.



Multi-component Image

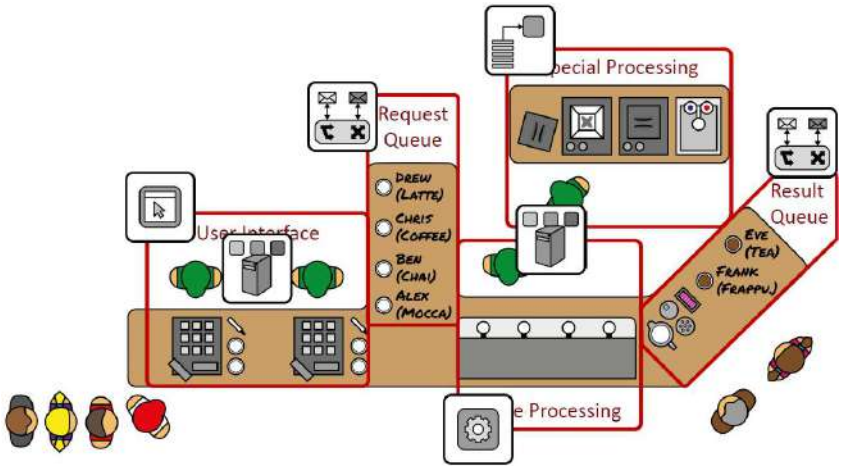
Virtual servers host multiple application components that may not be active at all times to reduce provisioning and decommissioning operations.

Multi-component Image



Coffee shop – Refinement of components

- Decide how to implement components

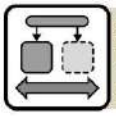


Refinement of components – Lessons learned

- Components should be integrated with messaging to ensure loose coupling
- Resources supporting functional components should be flexible (multi-component image)

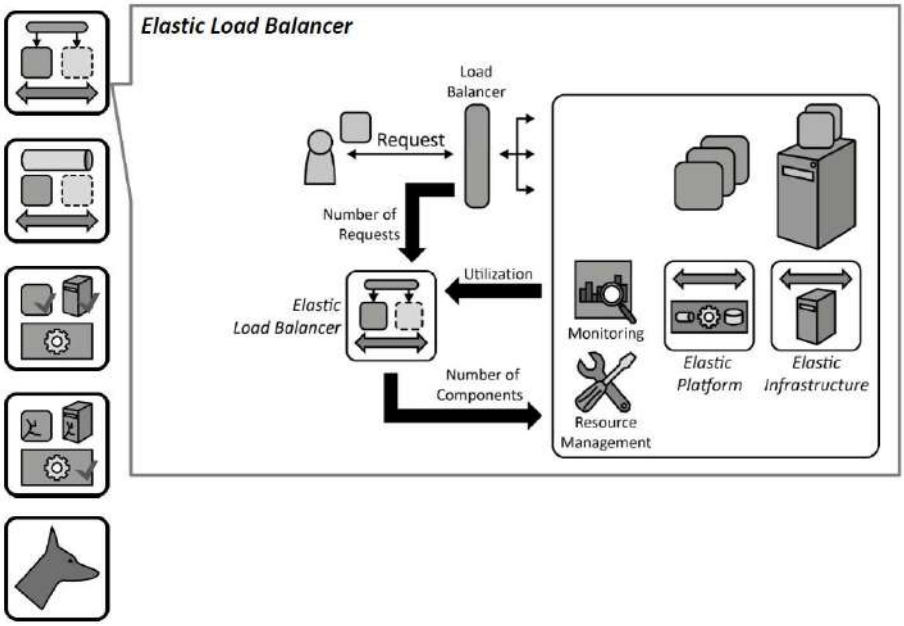
Decomposition → Workload → Data (state) → Refinement of components → Elasticity and resiliency

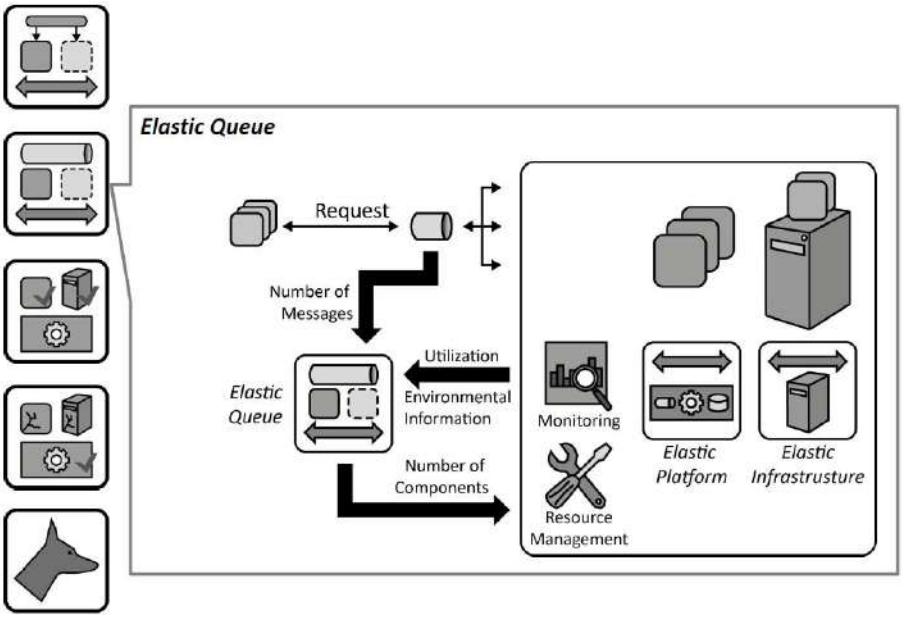
How to manage elasticity and resiliency?



Elastic Load Balancer
The number of synchronous accesses to an elastically scaled-out application is used to determine the number of required application component instances.

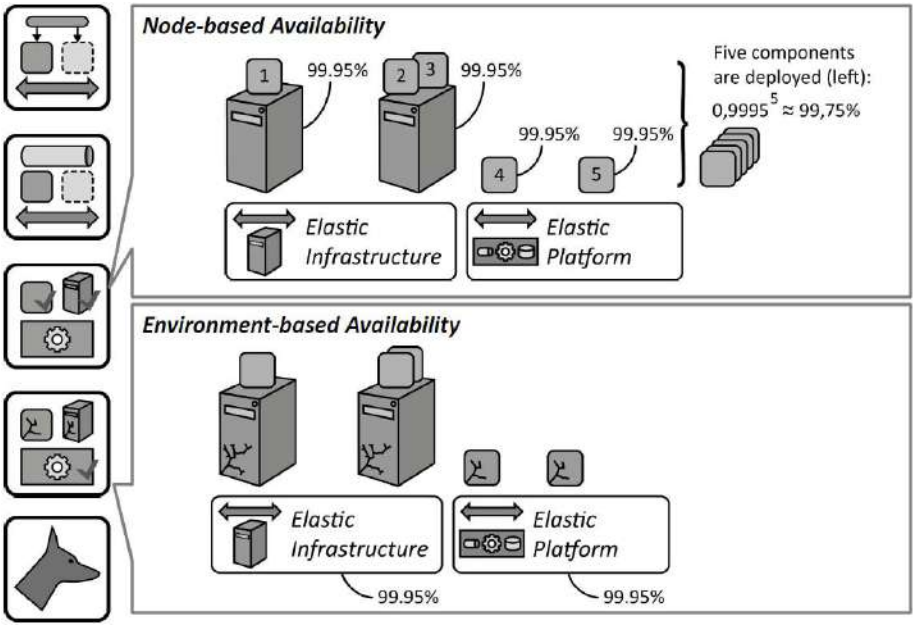
Elastic Queue
The number of accesses via messaging is used to adjust the number of required application component instances.


Elastic Load Balancer
Load Balancer
Request
Number of Requests
Utilization
Elastic Load Balancer
Number of Components
Monitoring
Resource Management
Elastic Platform
Elastic Infrastructure


Elastic Queue
Request
Number of Messages
Utilization
Environmental Information
Elastic Queue
Number of Components
Monitoring
Resource Management
Elastic Platform
Elastic Infrastrusture

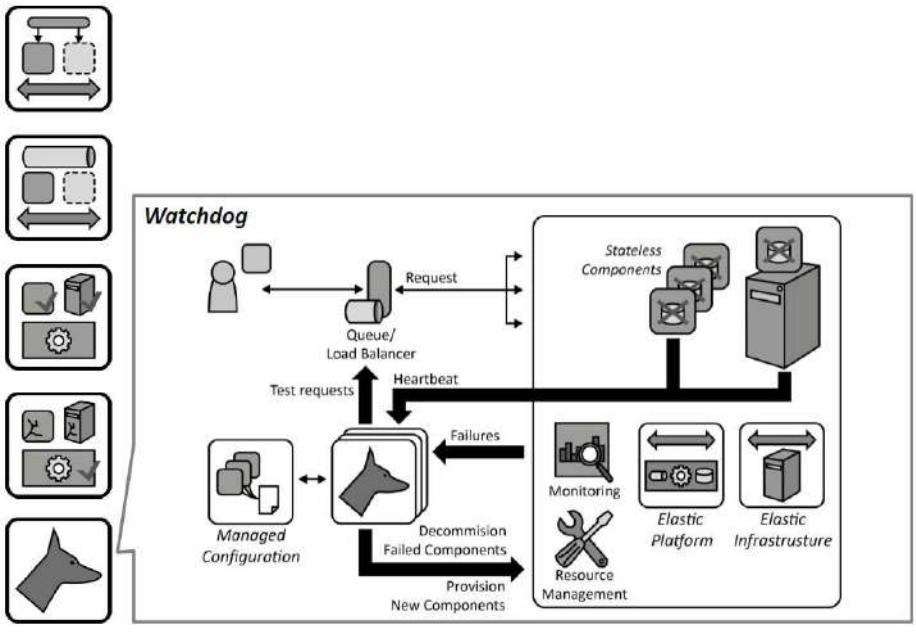
Node-based Availability
A cloud provider guarantees the availability of nodes, such as individual virtual servers, middleware components or hosted application components.

Environment-based Availability
A cloud provider guarantees the availability of the environment hosting individual nodes, such as virtual servers or hosted application components.



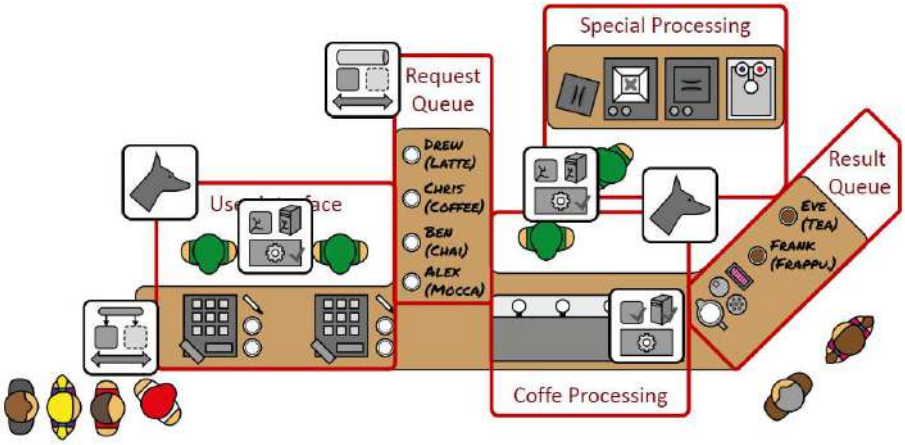
Watchdog

Applications cope with failures by monitoring and replacing application component instances if the provider-assured availability is insufficient.



Coffee shop – Elasticity and resiliency

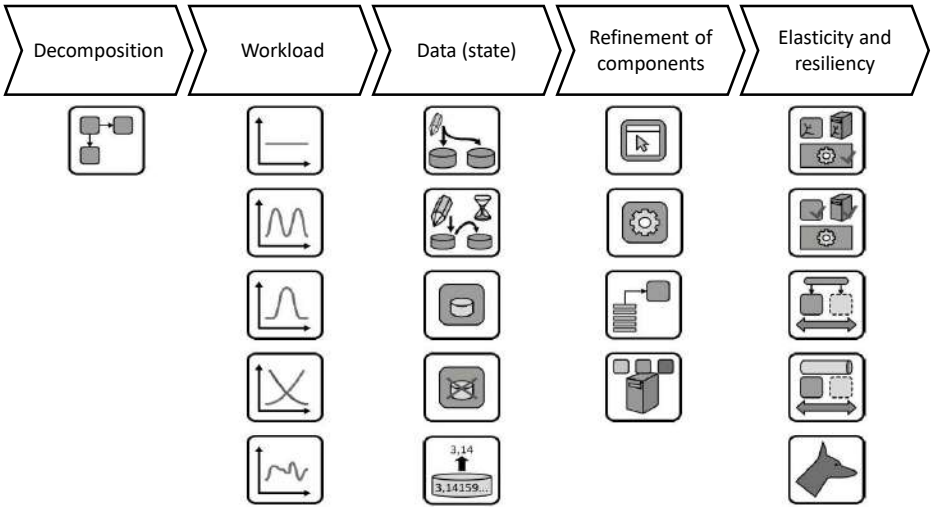
- What shall happen if workload changes or something fails?



Elasticity and resiliency – Lessons learned

- Analyze availability assured by provider (node-based or environment-based)
- In case of **low node-based availability** and **environment-based availability** implement a **watchdog**
- Use messages and requests to determine necessary component instances

Summary architectural questions



CLOUD SYSTEMS

The Geoscale Cloud

Lecture #7

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

Cloud systems have global scale

- They need information from global sources, consolidated to wherever a query occurs
  - With blockchain, anyone anywhere can see every transaction and validate it
  - With social networks, whether I am at home or visiting other places, I get updates from all my friends
- To work well, global cloud systems need global data replication

This is not only about replication

- We have point-to-point, subset, and true global replication patterns
  - Replication is one of a few dimensions of global state sharing
- Questions of consistency also arise: “How strong should the guarantees be?”

Overview

- Concept: Regions with Availability Zones (data centers) in them
- Constraints: Why we cannot just use TCP or RDMA or other standards
- Point to point solutions via "Zone aware" message queuing and apps
- Replication, including Google's unique Spanner atomic multicast
- 5G and how this will leverage but also push beyond the limits

Beyond the data center

- We discussed Facebook's global blob cache, but since then talked primarily about technologies used inside a single datacenter
- How do cloud developers approach global-scale application design?
- We will discuss "georeplication" and look at some solutions

Availability zones

- Servicing a data center can require turning it off
  - Amazon and Microsoft faced a problem early in the cloud build-out
- Why?
  - Some hardware components are too critical to service while active, like the datacenter power and cooling systems, or the "spine" of routers
  - Some software components cannot easily be upgraded while running, like the datacenter management system
  - There is a very long list of cases like these

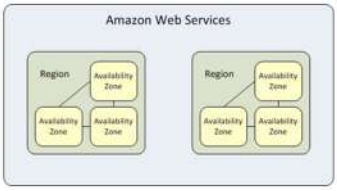
Availability zones

- Instead of building one massive datacenter, they would put two or even three side by side
- When all are active, they spread load over them, so everything stays busy
- This also gives an option for shutting one down entirely to do upgrades
  - With 3 datacenters, they would still be able to "tolerate" a major equipment failure in one of the two others

AWS Regions and Edge Locations

AWS Edge is less capable

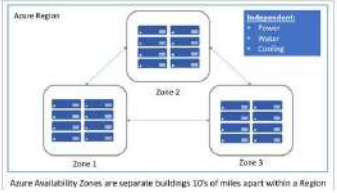
The AWS "region" has an availability zone structure



Azure Regions and Zones

Notice the blue circles. Those are regions with three data centers.

A "zone" additionally is physically spread out.



Azure Availability Zones are separate buildings 10's of miles apart within a Region

Connectivity limitations

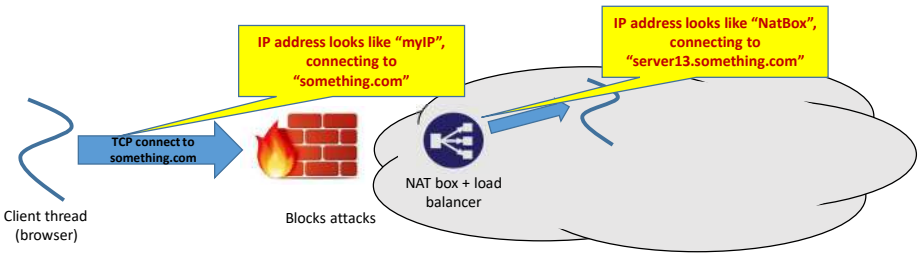
- Every datacenter has a firewall forming a very secure perimeter
  - Applications cannot pass data through it without following the proper rules
- Zone-redundant services are provided by AWS and Azure and others to help you mirror data across zones, or even communicate from your service in Zone *A* to a "sibling" in Zone *B*
- Direct connections via TCP would probably be blocked
  - It is easy to connect into a datacenter but **hard to connect out from inside**

Considering IP restriction

- The modern datacenter network can have millions of IP addresses inside each single datacenter
  - They are not actually unique IP addresses across different data centers (the addresses only make sense within a data center)
  - In fact, IP addresses only make sense within your own "private cloud"
- A computer in datacenter *A* often would not have an IP address visible to a computer in datacenter *B*
  - Blocking connectivity
- We send a TCP connect request to the datacenter over one of a small set of datacenter IP addresses covering the full datacenter
  - E.g., AWS has a few IP addresses per site
  - Some other systems mimic this approach, others have their own ways of ensuring that traffic gets to correct servers, even if hosted on their framework

Arriving traffic to NAT Box

- On arrival, packets are scanned for possible bot traffic or DDoS attacks, then the IP addresses are translated to "load balance" work over the proper servers
  - This would be costly, but high-speed parallel hardware is used to do the needed work at line rates



Connections behind NAT box

- The NAT box maintains a table of "internal IP addresses (and port numbers) and the matching "external" ones
- As messages arrive, if they are TCP traffic, it does a table lookup and substitution, then adjusts the packet header to correct the checksum
- Thus "server13.something.com" cannot be accessed directly and yet your messages are routed to it, and vice versa

Cannot connect to itself when considering georeplication

- If you have machine *A*.somthing.com inside AWS or Azure, and then try to connect to *B*.something.com, it works inside a single datacenter
  - In fact, you will be running in an "enterprise VLAN" or "VPC" (a virtual private cloud)
    - If you were to launch a network monitor you only see traffic from your own machines, not traffic from other datacenter tenants
- But if *B* was in a different datacenter, the connection simply will not work
  - Both *A* and *B* are behind NAT boxes, so neither can see the other

Solutions

- Because two NAT boxes are employed here, we could allow connectivity using NUTSS, a special way of tunnelling
  - Cloud vendors do not do so because they do not want uncontrolled connections
- Some vendors (not AWS or Azure) do offer ways to make an *A*-to-*B* connection across datacenters in the same region (availability zone)
  - You need to use a special library they provide and otherwise, the connection would fail
  - They charge for this service
- With AWS and Azure, you use an existing "Zone-aware" service
  - Many side-by-side TCP links, on a dedicated optical network

Example zone-aware services in Azure

- Linux Virtual Machines
- Windows Virtual Machines
- Virtual Machine Scale Sets
- Managed Disks
- Load Balancer
- Public IP address
- Zone-redundant storage
- SQL Database
- Event Hubs
- Service Bus
- VPN Gateway
- ExpressRoute
- Application Gateway

Zone-aware storages

- Zone-aware storage is a storage mirroring option
- Files written in zone *A* would be available as read-only replicas in zone *B*
  - *B* sees them as a read-only file volume under path /Zone/A/…
- This is a very common way to share information between data centers

Zone-aware features

- The Azure Service Bus in its "Premium" configuration
  - This is a message bus used by services within your VPC to communicate
  - Azure offers a configuration that automatically transfers data across zones under your control
  - You need to follow their instructions to set it up (they charge but the performance and rate have historically favored this model)
  - Basically, two queues hold messages, and then they use a set of side by side TCP connections to shuttle data in both directions which is very efficient

Zone-aware networking

- Azure's zone-redundant virtual network gateway
  - Used when you really do want a connection of your own, via TCP
  - Setup is fairly sophisticated, but they walk you through it
  - Creates a special "pseudo-public" IP address for your services, which can then connect to each other (not visible to other external users)
  - But performance might be balky (this is not their most performant option) and they charge by the megabyte transferred over the links

More remarks about IP addresses

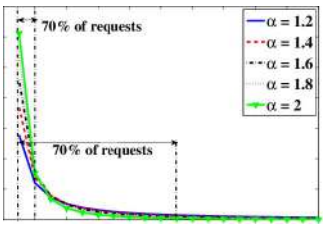
- We have seen that with cloud systems, you could connect to something.com and be routed to amazon.com instead
- In fact the modern cloud has considerable control over this, and you can influence that layer
- For each external client, when it initially connects, you can program the cloud DNS to route that specific request to a particular data center, and control whether or not the DNS record will be cached

More control

- You can use information in a cookie left on the client to advise the cloud on which of your servers would be the best one to handle a connection
- And you can select between a wide range of elasticity and routing and load-balancing "recipes" that are offered by the vendor
- In the next generation of routers (SDN routers) you will be able to even provide packet inspection rules that could route based on values in specific fields of the incoming request

Heavy tailed latency

- A big concern with georeplication is erratic delays
- Within an availability zone, the issue is minimal
  - The networks are short (maybe next building, maybe a few kilometers), so latencies are tiny
- But with global WAN links, latencies can be huge and variation grows
  - Mean delay within globe are ~90 ms
- Latencies follow Zipf's law (probability distribution)
  - Most traffic gets through very quickly
  - But some traffic takes extremely long, these distributions are heavy-tailed if the "very slow" traffic adds up to a substantial portion of the total traffic



Causes

- Packet drops
  - When a WAN link gets noisy, it might suddenly drop a slew of packets, then recover
  - The effect is that some packets rush through, but others show up very late and TCP will need to delay the early ones to deliver in order
- Routing or link speed changes
  - At WAN scale these events are infrequent, but they do happen and can cause a few seconds disruption
- Node crashes
  - Especially in the machines handling the links or queuing logic

Issues that arise

- Should we want to wait for all sites to respond, or just a quorum?
  - A subset of the sites large enough to include a majority
  - Any two quorums are certain to overlap ($Q_R + Q_W > N$)
- On the other hand, perhaps we would want all sites within an availability zone, but then would not need to wait for georeplicas to respond?
  - Leads to a three-level concept of Paxos stability
    - Locally stable in datacenter
    - Availability-zone stable
    - WAN stable

What have experiments shown (in Google)?

- Basically, Paxos performs poorly with high, erratic latencies
- A big issue is that delay is not symmetric
  - The path from Zone *A* to Zone *B* might be slow
  - Yet the path from Zone *B* to Zone *A* could be fast at that same instant
- The outgoing proposals experience one set of delays, and replies from Paxos members experience different delays
  - You end up waiting "for everyone"

Atomic multicast via Spanner (Google idea)

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, Dale Woodford. **Spanner: Google's Globally Distributed Database.** In *ACM Transactions on Computer Systems*, Volume 31, Issue 3, Article 8, 2013. https://doi.org/10.1145/2491245

- Google was one of the first to use a service built from Paxos in real datacenter settings, called Chubby
- In a single data center, Chubby worked well, but very slow
- But in a WAN setting, Google struggled to try and build a global version of Chubby
  - They could not pull it off

TrueTime

- Google uses actual time as a way to build an asynchronous totally ordered data replication solution called Spanner
- Google TrueTime is a global time service that uses atomic clocks together with GPS-based clocks to issue timestamps with the following guarantee

- For any two timestamps T_0 and T_1, if T_1 was generated after generation of T_0 finished, then $T_0 < T_1$

Why such definition?

- With the guarantee they offer, if some operation *B* was timestamped *T*, and could possibly have seen the effects of operation *A* with timestamp *T'*, then we can order *B* and *A* so that *A* occurs before *B*
- The full API is very elegant and simple
  - TT.now(), TT.before(), TT.after()
  - TT.now() returns a range from TT.before() to TT.after()
- The basic guarantee is that the time is definitely in the range TT.now()
- TT.before() is definitely a past time
- TT.after() is definitely in the future

Implementing TrueTime

- Start with a basic observation
  - Suppose clocks are synchronized to precision δ
  - It is 10:00:00.000 on my clock, and someone wants to run transaction *X*
  - What is the largest possible timestamp any zone could have issued?
- My clock could be slow, and some other clock could be fast
- The largest (worst case) possible will be $T + 2\delta$

Minimizing δ in TrueTime

- A mix of atomic clocks and GPS synchronization
  - They synchronize against the mix every 30 s, then might drift in between
- GPS can be corrected for various factors, including Einstein's relativistic time dilation, and they take those steps
- In the end their value of δ is quite small (0 – 6 ms)
  - At actual time 10:00:00.000, transaction X might get a TT.after() timestamp like (10:00:00.006, zone-id, uid)
    - The zone id and uid are to break ties

Ordering in TrueTime

- With T_0 and T_1 both TrueTime timestamps, we can look at two types of ordering

- The first of these **genuinely implies that T_0 happened before T_1**
  - **$T_0 < T_1$** is evaluated as T_0.after() < T_1.before()
  - If T_0 and T_1 are concurrent, then neither $T_0 < T_1$ nor $T_0 > T_1$

- The second is a **total ordering** but in fact **T_0 could be concurrent with T_1**
  - Each timestamp also has an actual time value, and we can order timestamps by this value, e.g. T_0.now() < T_1.now(), even if the times are concurrent
  - This ordering is used for sorting

Use of each kind of ordering

- Given a set of transactions, we can sort them by TT.now()
  - Spanner does this
- If applications hold "leases" (locks with timeouts) on resources, we can use the $T_0 < T_1$ form of ordering to know, with certainty, when a lease has expired
  - This works even without *A* being able to talk to *B*
- Use case for partitioning: If we track when processes last touched bases with a membership service, we can ensure that if *B* loses connectivity, it will shut down – and *A* can deduce this, too
  - For example, if *B* was in control of a drone, but loses contact to the main system, *B* shuts down and *A* can safely take over

How Spanner uses TrueTime?

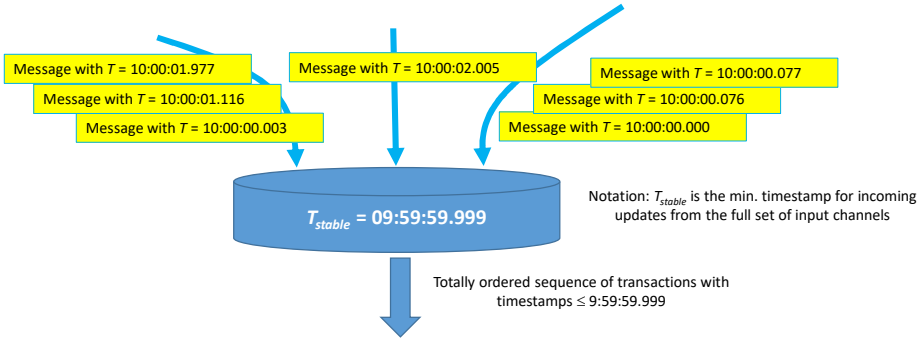
- Spanner is a transactional database system (key-value structure)
- Any application can generate a timestamped Spanner transaction
  - It will not be executed right away – they use stored procedures triggered by a message that **holds arguments for the triggered procedure**
- These are relayed over the Google version message services
  - Their service **delivers messages from Zone *X* to Zone *Y* in timestamp order**

How to use Google Spanner?

- First, get a job at Google – this is an internal-only product :-)
- You register your transactional logic as stored procedures, on all the Spanner instances
- When you wish to run a transaction you encode the arguments to your procedure into a message
  - Spanner will deliver it globally, causing your stored logic to run in a fixed order on all Google's datacenters worldwide

Spanner instance

- Spanner has connections to perhaps hundreds of remote zones
- Transactions flow in on these connections, and it stores them until every zone has sent some transaction with a timestamp of value T_{stable} or larger
- Then it can apply all transactions with timestamps $\leq T_{stable}$, in order



Why does it work?

- If Spanner has received messages with timestamps > T_{stable} from every zone, it has seen every transaction with timestamp ≤ T_{stable}
  - This is because the connections deliver messages in order, by timestamp
  - If an earlier transaction were to show up, it would violate this rule
- So, if it now places those transactions into timestamp order (breaking ties by zone-id, uid), they can be applied to the global database in total order

How long could an update be delayed?

- If incoming messages from any single datacenter pause, Spanner will have to wait for them (because they could have timestamps that order those messages "next")
  - When an incoming stream pauses, Spanner pauses
- Eventually they declare the datacenter itself to be down – like with virtual synchrony membership management
  - This step is tricky and is the complex part of the Spanner technology
  - They need a form of atomic multicast so that if any datacenter executes an update, every datacenter does so

Nice feature of Spanner

- Spanner can safely assume that the links to all its data centers are working, and it just waits to hear from all of them
  - If a data center is taken offline, Spanner is told, so then it will not wait for that link
- Spanner can make ordering decisions "as soon as possible"
- This is leading to interesting work on using RAID-style erasure coding to overcome the delay impact of a slow TCP link side by side with other TCP links that are not so slow

What limits Spanner?

- One limiting factor is that although the zone-to-zone data transfers run over large numbers of parallel TCP connections, the messages need to be put into order to obtain this "monotonicity" property
- A second limiting factor is Zipf-like latency with heavy tails
  - Spanner will often have to wait for "laggards"
- At global scale the effect can be significant (many seconds)

Paxos all over again?

- With the classic implementation of Paxos, we have a back-and-forth interaction that traverses every link several times in both directions
  - **Paxos experiences delays repeatedly**
- With Google Spanner, there are global asynchronous flows but no back-and-forth coordination of this kind
  - **Spanner experiences delays once**

What limits Spanner?

- Google researchers report that the most frustrating issue is that a transaction cannot even be processed **locally** without waiting this way
- In some situations, a speculative result may be acceptable
  - "If there are no conflicts, my transaction would run and you would win the auction"
- But in other settings, consistency is absolutely needed, so there is no choice

Tricks to minimize impact

- If a zone has not been sending any transactions, it should "pause"
  - Send a special "End of Sequence" message
  - Pause to send new transactions
  - Other zones no longer need to wait
- Later, to resume, it will have to get permission to restart
  - Send a "Resume Request" message
  - Every other zone must acknowledge this before new transactions can start

Other options?

- With a "primary zone" model, we can shard our database and assign each shard a primary owner (only the owner zone can update that shard)
- Then you can make the rule that the primary owner can always do transactions on shards that it owns, without waiting
- But if any transaction would need to access shards for which it is not primary, then all must use the Spanner ordering mechanism (you cannot mix methods)

What limits the primary zone method?

- In some situations it is hard to know what shards a transaction would access before the transaction code actually executes
- For example, the keys a transaction will touch might be a function of the data it reads in some initial step
  - So, until it executes, we do not know the key set, and cannot know if those will all be local shards
- Spanner is really aimed at cases like these

Mobility and georeplication

- With the advent of 5G, more and more IoT systems will include mobile clients in which the sensors are actually inside a vehicle like a car or plane
- 5G hubs are very much like Azure IoT Edge, and in fact Azure IoT Edge may be a leading platform option for 5G services
- With mobile users, the user itself may have a dynamically changing IP address, and might be using multipath TCP to maintain connectivity

5G will enable a new edge cloud

- Many experts believe that as these 5G point of presence systems roll out (seems it will be postponed to 6G), they will be so interconnected at the edge that they will rapidly become a new form of cloud
- Just like the standard cloud, we will be able to run apps on it
- The application will talk to the local 5G PoP, but those systems will need to use tools like messages queues or replication to share data

The path to 5G

- Is 5G just an IoT cloud integrated more closely to telecommunications?

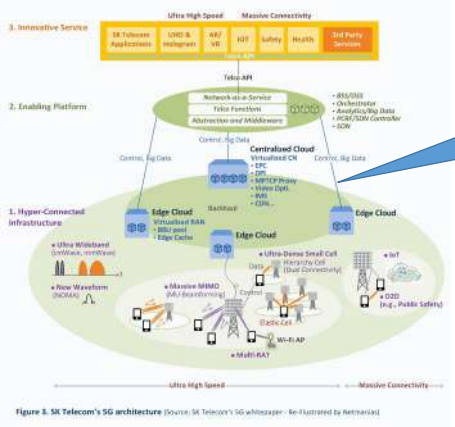


Figure 3. SK Telecom's 5G architecture (Source: SK Telecom's 5G whitepaper - Re-Illustrated by Rethinkx)

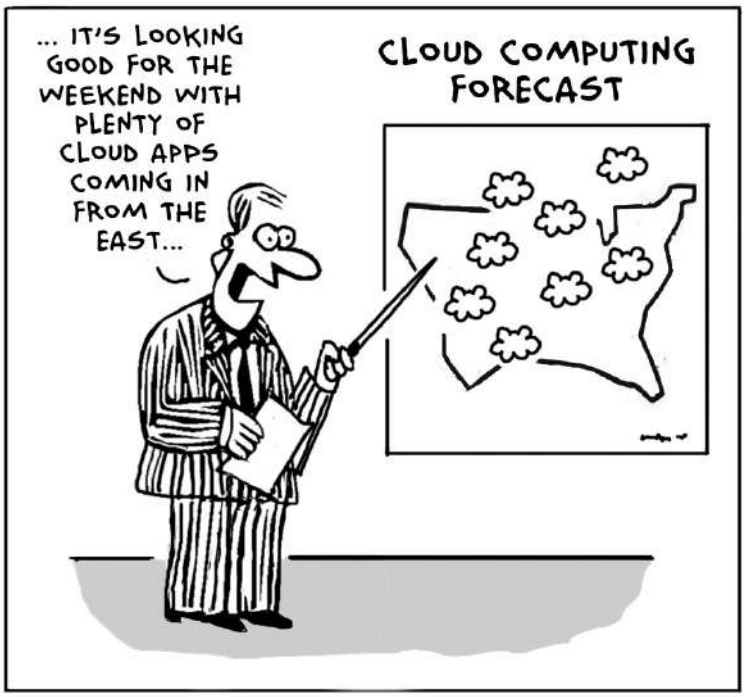
They envision a service infrastructure of their own living on Azure IoT Edge or similar

Key insights

- 5G is just another georeplication scenario, and the cloud can evolve to support it while retaining its main features
- Main areas where change will occur
  - Very large number of cellular regions ("5G edge cloudlets")
  - Mobility implies growing need to move data in anticipation of demand
  - Opportunity create a new hosting infrastructure for "5G apps"

Summary

- Georeplication is best viewed as having two scales
- Availability Zone
  - Just neighboring data centers
  - With some cost, you can use TCP and build "normal" protocols (based on Paxos)
- True global scope
  - Researchers have experimented with Paxos at global scope, but it performs poorly due to high-latency links
  - Techniques like Google Spanner are best
  - Derecho – a new version of Paxos with asynchronous state machine replication
- The notion of wide-area stability is useful and might be worth using in other contexts
- 5G mobility will introduce an additional layer of services



CLOUD SYSTEMS

Tracking State in Big Datacenters – Gossip Protocols

Lecture #8

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

Building scalable infrastructures

- Within a cloud computing environment we often need to manage very large pools of computers or services
- What is the best way to monitor and manage this kind of deployment?
- We will discuss the concept of using "gossip" as the basis for an unusually scalable style of solution
  - Amazon uses it in S3

Gossip is used in blockchain

- Gossip is used to distribute updates in **permissionless** blockchain, where the participants are anonymous but might not be cooperating
- Gossip makes it very hard for attackers to disrupt updates

Byzantine participants

- Blockchain must tolerate (a few) Byzantine participants
  - Not cooperating participants
- They seek to damage or disrupt the chain or to favor or delay transactions from certain parties, attempt to DDoS links, etc.
- Even a Byzantine participant cannot modify or forge messages
  - Messages are cryptographically signed

Gossiping (revisiting DS course)

- Suppose that Jana tells Peter something
- Peter is sitting next to David, and tells him
- Later, he tells Lucia and Peter tells Marek

- Each round doubles the number of people who know the secret

- This is an example of a **push** epidemic
- **Push-pull** occurs if we **exchange** data

Push-Pull in action

Push-Pull gossip

- Combines push and pull, but requires an RPC-style of interaction
  - Process *P* decides to gossip with process *Q*
  - *P* sends *Q* some form of concise "digest" of information available
  - *Q* sends back its own digest, plus a list of items it wants from *P*
  - *P* responds by sending those items, plus a list of items it wants from *Q*
  - *Q* sends the requested items
- This avoids sending large duplicate objects

Each message is treated separately

- We send requests but do not actually "wait" for a response
- This way if our peer turns out to be faulty or uncooperative, nothing bad can happen
  - We sent a first stage push-pull message but got not reply and do not have any associated state that lingers, etc.
- If it works, great
- If not, the next message will go to someone else

Limited work per round

- Think about maximum size gossip messages – there is always a limit
- All of these patterns have a fixed maximum number of messages that will be sent and received
- So, each process has a limit on how many bytes it will need to send for each gossip round

Reasonable network assumptions

- Gossip does not assume all-to-all connectivity
  - Outside the datacenter we often see more of a graph of peering connections
- Instead, we tend to assume an "expander" graph
  - Every node is reachable from every other node within $O(\log n)$ hops
  - We also assume that the network is robust to link failures or Byzantine DDoS attack
- These are considered to be very practical, reasonable assumptions

Size constraint

- For gossip to really have constant cost at each participant, we need to decide on a maximum message size
- Messages can grow to that maximum, but not beyond
- But with unlimited numbers of processes, even if the events we gossip about are rare, the amount of information to share could grow as $O(n)$

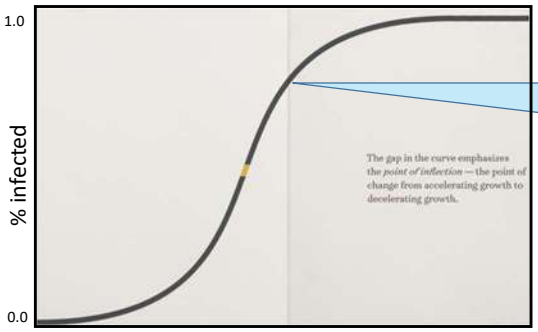
Tricks for limiting message size

- Many systems only gossip about "recent" information
- The theory is that older data is probably stale or wrong in any case
- Then the issue is "how many events can happen in delta time?"
  - This may be more manageable

Gossip scales very nicely but pure push or pure pull is not ideal

- Participant load is **constant**, independent of size of the system
- Total network load linear in system size
- Information spreads in log *n* time, yet that limit on work per process remains in effect

Pure push case: Needs a long time to reach the last few processes

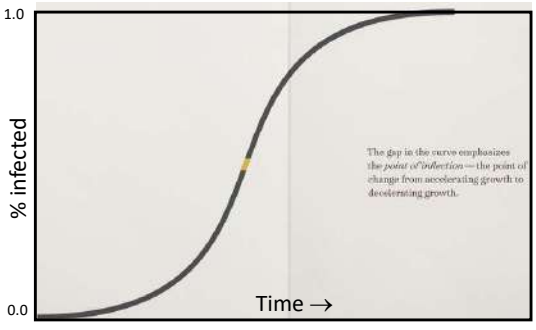


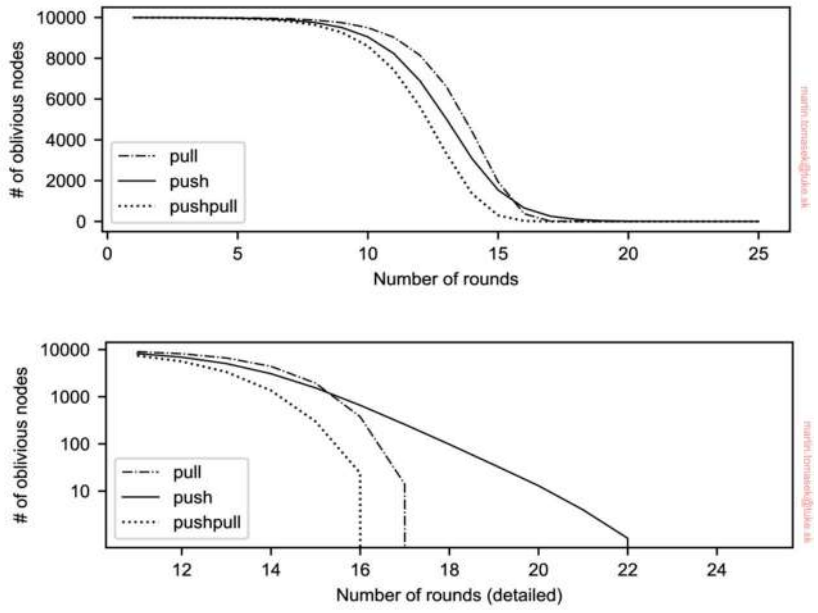
"Reinfection" eventually becomes dominant

Gossip with push-pull is best

- Participant load is **constant**, independent of size of the system
- Total network load linear in system size
- Information spreads in log *n* time, yet that limit on work per process remains in effect

Push-Pull terminates quickly and reaches every non-faulty process





One small risk

- What if everyone decides to gossip to the same process all at once?
- Selection of the target is random – it could happen
- But it is very unlikely and in fact the receiver could just ignore some messages
  - **Gossip does not require reliable messaging**

Gossip in distributed systems

- We can even gossip about membership
  - Need a bootstrap mechanism, but then discuss failures, new members
  - This feature is **not** used in permissionless blockchain, but it is used in permissioned blockchain
    - In fact this is what distinguishes those models
- Gossip to repair faults in replicated data
  - "I have 6 updates from server 12"
- If we are not in a hurry, gossip to replicate data too

Why "if we are not in a hurry?"

- Gossip is very robust, but log n time might not be fast
- Normally we run one round every second or so
- A data center with 100,000 computers would have log n = 17, so when something important happens, it would take 17 seconds to reach all nodes
- Size limit of messages can also be an issue

What if we actually **are** in a hurry?

- One option is to mix gossip with a second mechanism
- UDP multicast can be useful here – This is an old and not-often used feature of the UDP protocol (in contrast to TCP)
  - Instead of having just one server for each IP address, UDP datagrams allow multiple servers to attach to the same shared IP address
  - With this feature, the UDP multicast will go to all receivers

- BIRMAN, K. P. – HAYDEN, M. – OZKASAP, O. – XIAO, Z. – BUDIU, M. – MINSKY, Y.: *Bimodal Multicast*. ACM Transactions on Computer Systems, Volume 17, Issue 2, 1999, pp. 41 – 88.
  - https://doi.org/10.1145/312203.312207

Some warnings

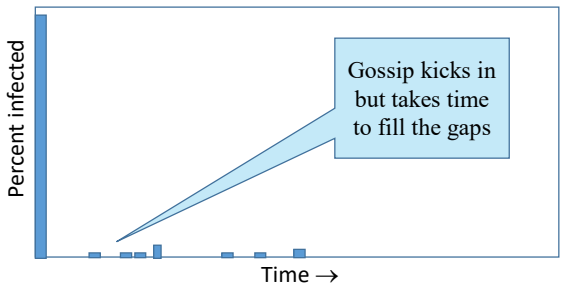
- Many datacenters disable the router feature UDP multicast requires
- If they do this, it will not work even though Linux might allow you to bind to that shared "class D" multicast IP address, and to send to it
  - The messages just will not reach other machines
- Also, because UDP multicast is not reliable, some receivers could receive the message, but others might drop it silently
  - No retransmissions occur

Bimodal multicast

- This was a protocol that uses UDP multicast as a first step, then "fills any gaps" using gossip
- Good trick: If some node is asked to share the same message twice via gossip, instead it resends the UDP multicast
  - That way if a few processes seem to have missed some message, it gets retransmitted soon
- We get two delivery delay "curves", one for UDP multicast, the second for gossip to fill the tiny number of remaining gaps

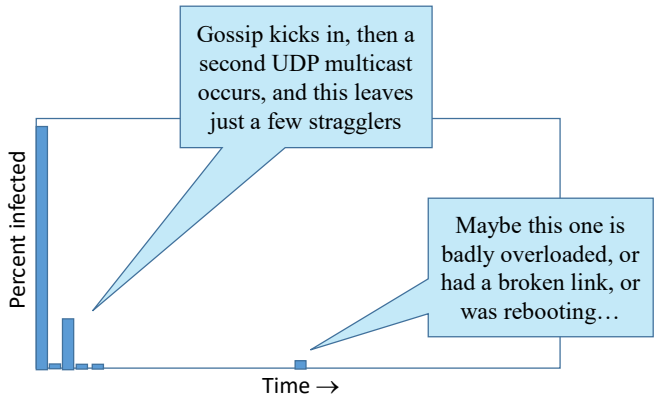
A bimodal delivery curve

- In this picture, 99.9 % of the messages are received via UDP multicast.
  - So in a datacenter with 100,000 machines, only 100 or so miss it
- But then the gossip spread could be a bit slow



A bimodal delivery curve

- With a second UDP multicast, our curve looks much better



Is gossip atomic multicast?

- Not really, it lacks a total order guarantee
- But some systems add a timestamp and deliver multicasts in timestamp order, breaking ties using the IP address of the senders
- At time τ, they have some estimated δ such that no messages are expected with timestamp $\leq \tau - \delta$ (if one were to turn up, they would silently discard it)
  - When messages become "stable" in this sense, they deliver them

Stragglers can miss messages

- A machine that ran very slow for a while and missed messages will see them via gossip, but it may be too late to deliver them in correct order
- In effect, a mistake was already made and it is too late to fix it
- So, bimodal multicast can approximate atomic multicast, but only to some probabilistic limit
  - It will not (cannot) be flawless

Gossip about membership

- Start with a bootstrap protocol
  - For example, processes go to some web site
  - On it they find a dozen nodes where the system has been stable for a long time
  - Pick one at random
- It sends back a membership list, and now your node is up and running
  - Then track "processes I've heard from recently" and "processes other nodes have heard from recently"
  - Generally, use push gossip to spread the word

Astrolabe

- Intended as help for applications adrift in a sea of information
- Structure emerges from a randomized gossip protocol

- VAN RENESSE, R. – BIRMAN, K. P. – VOGELS, W.: *Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining*. ACM Transactions on Computer Systems, Volume 21, Issue 2, 2003, pp. 164 – 206.
  - https://doi.org/10.1145/762483.762485

Astrolabe is a flexible monitoring overlay

Name	Time	Load	Weblogic?	SMTP?	Word Version
swift	2271	1.8	0	1	6.2
falcon	1971	1.5	1	0	4.1
cardinal	2004	4.5	1	0	6.0

Periodically, pull data from monitored systems

Astrolabe is a flexible monitoring overlay

swift.something.com

Name	Time	Load	Weblogic?	SMTP?	Word Version
swift	2271	1.8	0	1	6.2
falcon	1971	1.5	1	0	4.1
cardinal	2004	4.5	1	0	6.0

Periodically, pull data from monitored systems

cardinal.something.com

Name	Time	Load	Weblogic ?	SMTP?	Word Version
swift	2003	.67	0	1	6.2
falcon	1976	2.7	1	0	4.1
cardinal	2231	1.7	1	1	6.0

Astrolabe in a single domain

- Each node owns a single tuple, like the management information base (MIB)
- Nodes discover one-another through a simple broadcast scheme (“anyone out there?”) and gossip about membership
  - Nodes also keep replicas of one-another’s rows
  - Periodically (uniformly at random) merge your state with some else

State Merge: Core of Astrolabe epidemic

swift.something.com

Name	Time	Load	Weblogic?	SMTP?	Word Version
swift	2011	2.0	0	1	6.2
falcon	1971	1.5	1	0	4.1
cardinal	2004	4.5	1	0	6.0

cardinal.something.com

Name	Time	Load	Weblogic ?	SMTP?	Word Version
swift	2003	.67	0	1	6.2
falcon	1976	2.7	1	0	4.1
cardinal	2201	3.5	1	1	6.0

State Merge: Core of Astrolabe epidemic

swift.something.com

Name	Time	Load	Weblogic?	SMTP?	Word Version
swift	2011	2.0	0	1	6.2
falcon	1971	1.5	1	0	4.1
cardinal	2004	4.5	1	0	6.0

swift	2011	2.0

cardinal	2201	3.5

cardinal.something.com

Name	Time	Load	Weblogic ?	SMTP?	Word Version
swift	2003	.67	0	1	6.2
falcon	1976	2.7	1	0	4.1
cardinal	2201	3.5	1	1	6.0

State Merge: Core of Astrolabe epidemic

swift.something.com

Name	Time	Load	Weblogic?	SMTP?	Word Version
swift	2011	2.0	0	1	6.2
falcon	1971	1.5	1	0	4.1
cardinal	2201	3.5	1	0	6.0

cardinal.something.com

Name	Time	Load	Weblogic ?	SMTP?	Word Version
swift	2011	2.0	0	1	6.2
falcon	1976	2.7	1	0	4.1
cardinal	2201	3.5	1	1	6.0

Observations

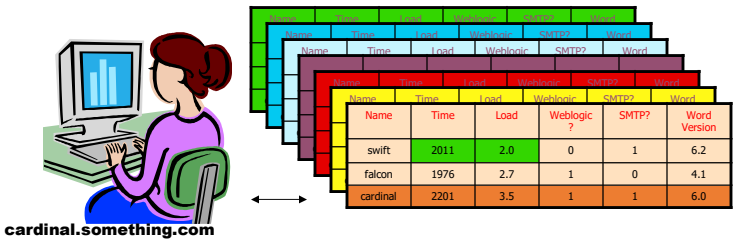
- Merge protocol has constant cost
  - One message sent, received (on average) per unit time
  - The data changes slowly, so no need to run it quickly
    - We usually run it every five seconds or so
  - Information spreads in $O(\log n)$ time
- But this assumes bounded region size
  - In Astrolabe, we limit them to 50 – 100 rows

Big systems

- A big system could have many regions
- Looks like a pile of spreadsheets
- A node only replicates data from its neighbors within its own region

Scaling up… and up…

- With a stack of domains, we do not want every system to "see" every domain
  - Cost would be huge
- So instead, we will see a summary



Astrolabe builds a hierarchy without any servers

- Dynamically changing query output is visible system-wide
- SQL query "summarizes" data

Name	Avg Load	WL contact	SMTP contact
SF	2.2	123.45.61.3	123.45.61.17
NJ	1.6	127.16.77.6	127.16.77.11
Paris	2.7	14.66.71.8	14.66.71.12

Name	Load	Weblogic?	SMTP?	Word Version	...
swift	1.7	0	1	6.2	
falcon	2.1	1	0	4.1	
cardinal	3.9	1	0	6.0	

San Francisco

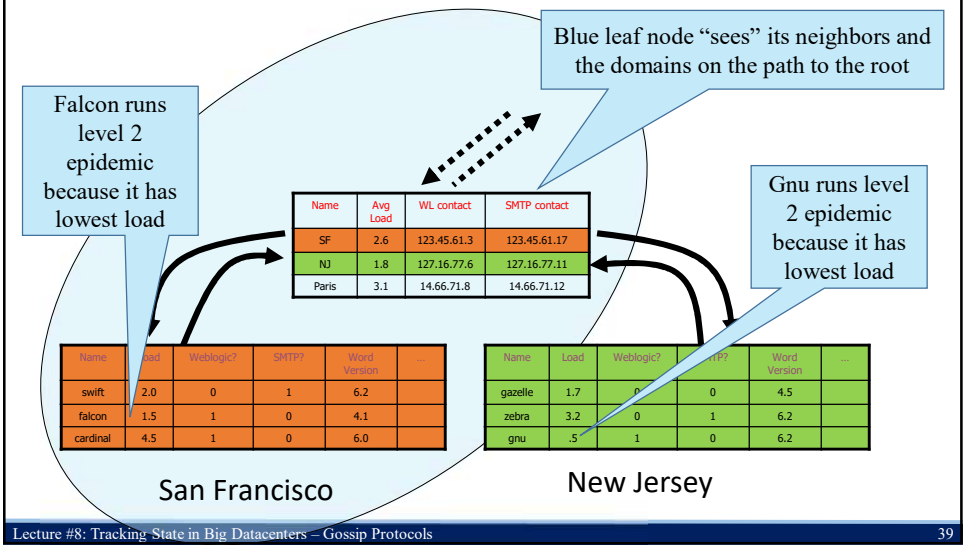
Name	Load	Weblogic?	SMTP?	Word Version	...
gazelle	4.1	0	0	4.5	
zebra	0.9	0	1	6.2	
gnu	2.2	1	0	6.2	

New Jersey

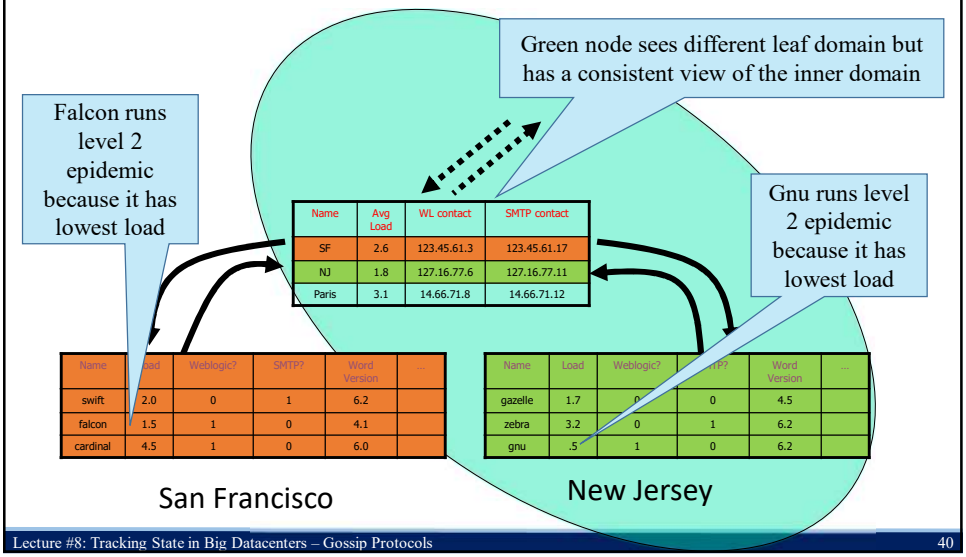
Large scale: "virtual" regions

- These are
  - Computed by queries that summarize a whole region as a single row
  - Gossiped in a read-only manner within a leaf region
- But who runs the gossip?
  - Each region elects *k* members to run gossip at the next level up
  - Can play with selection criteria and *k*

Hierarchy is virtual… data is replicated



Hierarchy is virtual… data is replicated



Worst case load?

- A small number of nodes end up participating in $O(\log_f n)$ epidemics
  - The f (fanout) is something like 50
  - In each epidemic, a message is sent and received roughly every 5 seconds
- We limit message size so even during periods of turbulence, no message can become huge

Criticism of Astrolabe

- Uses complained that Astrolabe did not feel natural
- People expect a database, not a distributed query system
  - However, they do like the idea of dynamically searching for the root cause of a problem
- Gossip is slow, and management systems may need to react quickly
  - So if they do use Astrolabe, they might ask for a feature like the bimodal multicast UDP "acceleration", which can speed things up when events occur

Who uses Astrolabe?

- When Werner Vogels joined Amazon, they adopted Astrolabe inside the S3 storage system
- It evolved substantially over time, but the gossip pattern was retained
- Today, many management systems use ideas similar to these, but the Astrolabe hierarchical approach is not currently seen even at Amazon

Blockchains

- Blockchains have emerged as a new application for gossip, but in wide-area settings, not inside datacenters
- The area was pioneered by Bitcoin cryptocurrency
- Bitcoin is defined over an append-only tamperproof log (like Paxos)
  - A gossip protocol is used to share proposed updates robustly

Blockchains

- The Bitcoin log is a bit complicated because of the cryptographic model
- But the policy for disseminating updates (log appends) is definitely a gossip protocol
  - Every node talks to some neighbors in the Bitcoin network "overlay" and they exchange data using gossip techniques
- The idea is that even if a few nodes are Byzantine, the overall blockchain can route around their bad behavior

How to use Gossip in a project?

- For example, Lonnie Princehouse's MiCA platform
  - https://github.com/mica-gossip/MiCA
- MiCA uses a Java-based coding style
- You create and compose gossip and can even configure them to run at different gossip rates

Summary

- Gossip can be a powerful tool for building stable distributed protocols
- It is extremely easy to use, and robust
- But it can be challenging to create entire systems based on gossip
  - The technology works best as a tool for building subsystems with specific roles

Why not use gossip everywhere?

- There are many tasks where the fit seems quite good, like blockchain and system management
- In a cloud datacenter, gossip is appealing for checking to see if systems have hung processes, monitoring loads, or tracking storage capacity
- The underlying values change slowly, so even a “slow” tracker will still be pretty accurate

But there are some cautionary tales

- For example, gossip once caused all of Amazon S3 to crash
- This nearly resulted in a congressional inquiry
  - When S3 crashes, a great many companies also freeze up – any company that depends on the cloud depends on the S3 file system storage solution
- How does S3 use gossip?

Amazon S3: The "Simple Storage System"

- S3 is a huge pool of storage nodes
- Plus, a "meta-data" server that keeps track of file names and where they can be found
- To store data, an application asks the meta-data service to allocate space, then sends the data to the appropriate storage servers

Why do we use the term "meta-data"?

- When you think about a file, you tend to think of the file name and the file contents
  - Like a key and a value
- But in fact files also have owners, permissions, create time, last access time, length (and perhaps, size on disk, which can be much smaller), etc.
- We associate this data with the file
  - In Linux the inode plays this role
  - In S3 and other big-data systems, the meta-data service does it

How does the S3 meta-data service track space available on storage units?

- You might expect this to be easy, because the meta-data service does the allocations
- But in fact the meta-data service itself is sharded, so any single shard within it only knows (for sure) about files it is responsible for
  - Additionally, sometimes a server needs to take some storage offline
- To know the full state of the full S3 deployment we would need to sum across all meta-data services

Load balancing

- For each server, use gossip to track an estimate of the current amount of free space. Line them up on a "space available" line

- For a new allocation, pick a random spot in this line
- This spreads the incoming load around but will be biased to favor servers with more space

Gossip is used for tracking storage

- Amazon used a gossip protocol in this role, specialized to S3 meta-data
- The basic idea is to use gossip to keep track of how much space each S3 storage node is reporting that it has available
- This is inexpensive and because each storage unit holds hundreds of gigabytes, the values do not change rapidly
  - A good match for gossip

…until it went wrong

- Once upon a time, when S3 was working perfectly well, a storage server needed to take some storage offline
- Because of doing this, it suddenly went from having 10 % excess capacity to being slightly over-full
  - This was not a bug – the servers actually have a tiny bit of reserve space, so "available capacity" **could become slightly negative**
  - The idea was that meta-data managers would omit that server from the line
- To support this, space available used **signed** 32 bit integers, but the S3 meta-data service declared the field to be a 32 bit **unsigned** integer

Signed-to-unsigned conversion is a bug

- In older C programs and some other weakly typed languages, storing a signed value into an unsigned variable is not flagged as an error
- C++ and Rust are examples of languages that DO complain about this
- Amazon was using C at that time, the compiler did not complain, and in fact their servers had so much capacity that for a long time, none ever actually went into "overload" in any case

I have –3 GB free capacity

- But then one day, we did overloaded. What happens if a signed integer becomes negative, and then we interpret it as an unsigned integer?
- The sign bit will be set, 2^{31} is a large number
- In effect, small negative numbers will suddenly be interpreted as big positive numbers
  - Our server suddenly reports: "I have 2147483645 GB of free capacity"

Suddenly lots of new files were sent to this storage server

- Since it was full, it refused the requests
- S3 **did** have logic to handle that situation
  - But it became a bottleneck
- S3 became extremely slow

It took Amazon nearly a day to figure this out

- S3 was actually working, it did store new files, but it was weirdly slow
- Higher level applications that depend on S3 began to have request timeouts, causing a cascade of failures
  - Every AWS product was broken
- This issue of one failure triggering other failures is a major problem see in the cloud and causes a whole series of outages all to happen at once

From bad… to worse?

- They eventually found the issue, and came up with a great idea
- They shut down the bad server, but nothing happened
  - Gossip is very slow to spread the word
- So, then they noticed that meta-data server md1 was gossiping that server53 had infinite space
- They killed it
- Suddenly, md2 took over and started gossiping that server53 had infinite space

Issue you see in this story

- With gossip, fresher data might not always spread faster than stale data
- Gossip is very robust to servers being down, which means that just rebooting a single node will not fix anything
- Pushing an urgent patch did not help either: many computers were in a thrashing state and some of those would wake up after a random amount of time
  - They were still gossiping these huge “free space” numbers

Questions to think of

- What is the best way to
  - Count the number of nodes in a system?
  - Compute the average load, or find the most loaded nodes, or least loaded nodes?
- Options to consider
  - Pure gossip solution?
  - Construct a service that actively tracks the nodes, or the load, etc.?

The answers

- Gossip is not very good for some of these tasks
  - There are gossip solutions for counting nodes, but they give approximate answers and run slowly
  - Tricky to compute something like an average because of "re-counting" effect (best algorithm: Kempe et al.)
- On the other hand, gossip works well for finding the *c* most loaded or least loaded nodes (constant *c*)
- Gossip solutions run in time $O(\log n)$ and generally give probabilistic solutions

Lesson learned

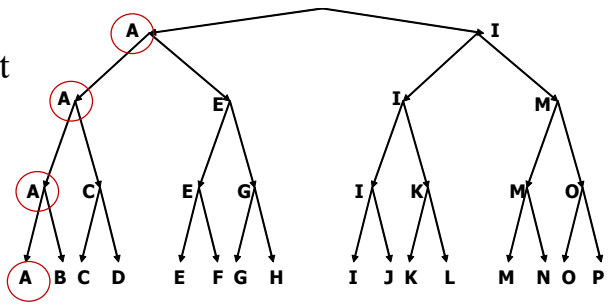
- In retrospect, many mistakes were made
  - Use of a weakly typed language, C
  - Poor communication about a feature (using a negative number to report that a server is over capacity), so some people did not know about it
  - Poor testing of the combined elements (this bug should have been seen before it was put into service)
  - It is not even completely obvious that this design was the best way to solve their actual S3 storage balancing task

Another really bad story

- A company experimented with using Astrolabe
- In their experiment, which was never deployed in practice, they had the idea that instead of the least loaded leaf nodes playing the inner gossip role, every node would have an equal role
- So, they came up with a new kind of Astrolabe tree

A normal aggregation tree

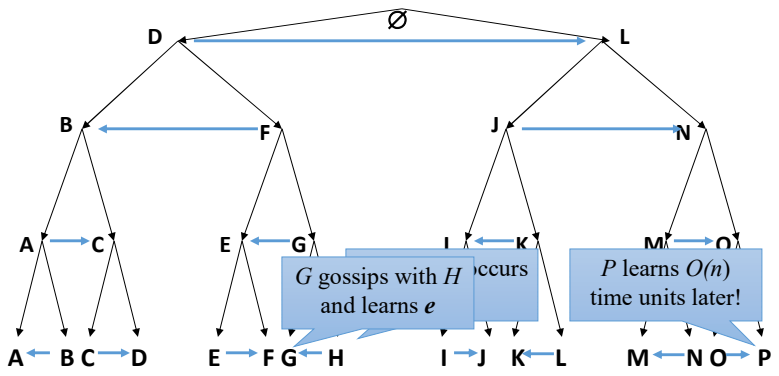
- In this tree, the lowest level has fanout of 2, whereas Astrolabe used 100, but this is still fine
- Notice that node *A* is elected to gossip at several levels of the hierarchy



Boss gets worried… is this "fair"?

- When a company acquires a technology they often redesign some aspects
- In this particular scenario, the new owners knew that Astrolabe was kind of slow (due to gossip), but had the idea that maybe a more even gossip role sharing would help
- When people explain that all of Astrolabe is only using 0.00001 % of the network, and that 2 or 3 gossip messages per second versus 1 per second is not a big issue, boss brushes that away
  - "I insist! We must fix this problem! And patent the improvement!"

A different aggregation tree



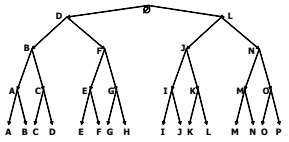
Wait! *P* learns *n* time-steps later?

- Wasn't Astrolabe supposed to run in $O(\log n)$ time?
- Why is it suddenly running in time $O(n)$?
- What went wrong?

- In this horrendous tree, each node has equal "work to do", but the information-space diameter is larger
- Astrolabe was actually benefitting from "instant" knowledge because the epidemic at each level is run by someone elected from the level below

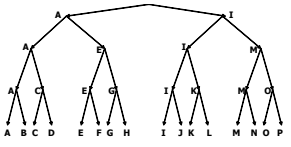
Information space perspective

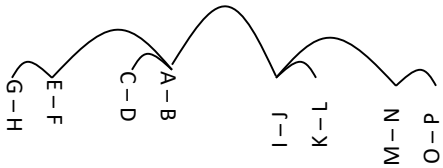
- Bad aggregation graph: diameter $O(n)$



H–G–E–F–B–A–C–D–L–K–I–J–N–M–O–P

- Astrolabe version: diameter $O(\log n)$



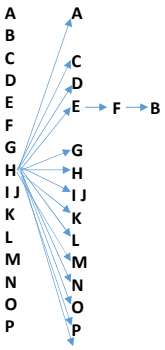


So… we fixed that

- But then they had another idea
- Recall how UDP multicast was used to speed up urgent notifications with Bimodal Multicast
- Could something like that be used to speed up Astrolabe?

Information space perspective

- UDP multicast causes a fast "all to most" exchange
- Then a few stragglers need to catch up in the next gossip round or two



In this UDP-multicast accelerated graph, we get a very accelerated convergence

It seems like it might be a good idea, but we will not investigate it in details

UDP multicast problem

- Let's look and understand UDP multicast in more detail, and to hear about an issue it already had
- This is unrelated to using any kind of gossip
- This issue arose in a very famous product, let's call it "A Message Bus"

UDP multicast was very popular

- In fact we have used two terms
  - IP multicast – This is part of the definition of the Internet IP packet protocol. Like IP itself, it works for packets of maximum size 1400 bytes
  - UDP multicast – "Slices" larger packets into 1400 byte segments
- Neither is a reliable protocol: these are for sending a packet, which will be rapidly routed to its destination(s), but there are no automatic retransmissions if the packet gets lost along the way

Building blocks

- Infrastructure tools designers think about technology as building blocks
- They focus on modular components and match the properties of the resulting infrastructure tool to the available building blocks
- But each new combination can bring unexpected problems caused by interactions between elements that work perfectly well "on their own"

How UDP multicast works

- First, the IP system reserves a class of IP addresses for use in UDP multicast
  - They are "class D" addresses, and we can think of each one as a unique id plus a unique port number shared by a set of receivers
- For example, "A Message Bus" might allocate IP address and port for messages about "extra food"
  - Every pub-sub topic would have its own (IP address, port) pair

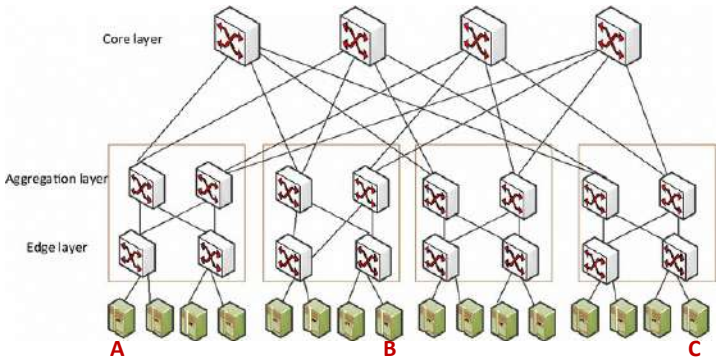
Roles of hardware

- In UDP multicast, the hardware itself is supposed to route packets only to where they are wanted
- For "A Message Bus", this will be "nodes subscribing to the topic"
  - Each (ip, port) pair corresponds to a topic, and we want our packets to go only to subscribers
- So, the network becomes active, and **filters** traffic

The basic mechanism

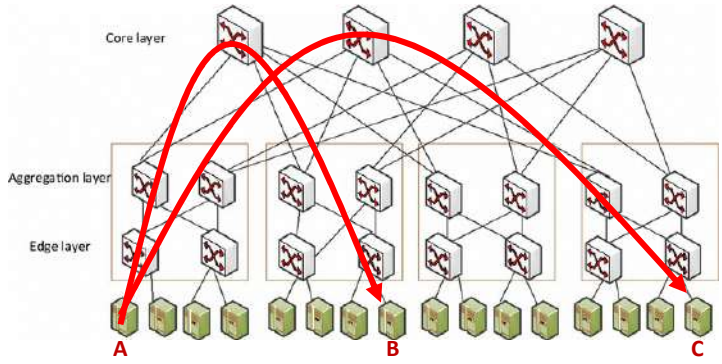
- When a machine boots, the message bus server instance launches and creates a socket and binds this standard IP address and port to it
- This causes the NIC to begin to watch for messages that match
  - In addition, the top of rack switch and datacenter routers are informed that there is a new multicast listener on this segment of the network
- The routers use this knowledge to filter on each forwarding link

Example: Nodes *A*, *B* and *C* are in IP multicast group of "A Message Bus"



So-called FAT tree topology has at least 2 routes between each pair of nodes for fault-tolerance

Goal: Forward messages efficiently



Ideally, *B* and *C* are the only nodes that receive *A*'s publication on this topic. And ideally the route picked is perfect

How can hardware help?

- Central idea
- When a router receives a multicast message *m* on link *l*, for each outgoing link from it, ask "is there a subscriber *down* this link"?
- With a very quick test for "the destination of *m* will match some subscriber reachable via this link", we can forward efficiently
- But routers run very fast, the test has to occur in one clock cycle

Set membership question

- The message *m* has a destination (address, port) pair
- Think of this as a value *x*
- In UDP multicast cases *x* would be 64-bits long
- Now think of "subscribers" (for any UDP multicast group) "down" (reachable) via each link
  - Call this a set *S*
- We want to know "is *x* in *S*?", for each link

Bloom filter: Fast "is x in S" testing

- A vector of bits (0 and 1):

- Answers question "is x in set S", the bits represent keys that are in the set

- Filter starts at all 0
  - To tell the filter "x is in S", hash key x, and set that bit
  - To test "is x in S?", hash key x, and return that bit

Issue: False positives

- In modern routers, the Bloom filter needs to be implemented with a special form of ultra-fast memory and associated logic
  - This is costly
- Limits the bit vector size to around 16 kbits
- But with 16 kbits, we could have a collision where x and x' hash to the same bit
  - That would cause us to deliver m to machines that are not interested
  - They will discard m, but will pay a small overhead cost

Bloom filter: Fast "is x in S" testing

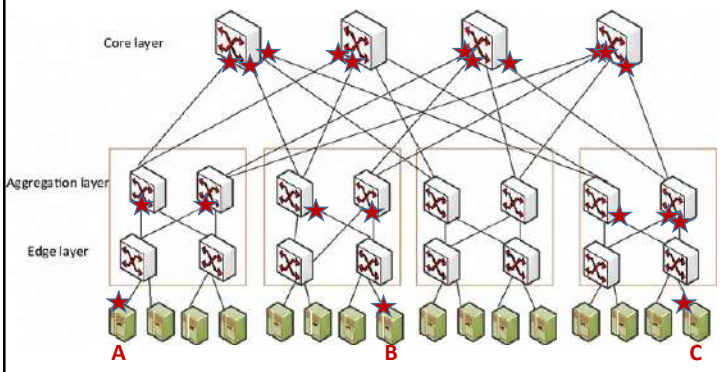
- Can a Bloom filter be extended to better handle hash collisions?

- Bloom's idea: just hash x, $(x + 1)$, $(x + 2)$, … (usually 3 is enough)
- The odds of a collision on 1 bit are pretty high
  - But is it likely that all k bits would "accidentally" be 1? Not very!

Use of these filters?

- The NIC uses a Bloom filter to recognize incoming IP multicast packets it should accept
- The TOR switch uses a Bloom filter to track which links lead to machines listening for a particular IP multicast address
- The fat-tree of datacenter routers uses this to remember which subnetworks have a machine listening for an IP multicast address

Example: Nodes *A*, *B* and *C* are in IP multicast group of "A Message Bus"



★ means: If you receive a message to address *x* (on any incoming link), forward it via this outgoing link

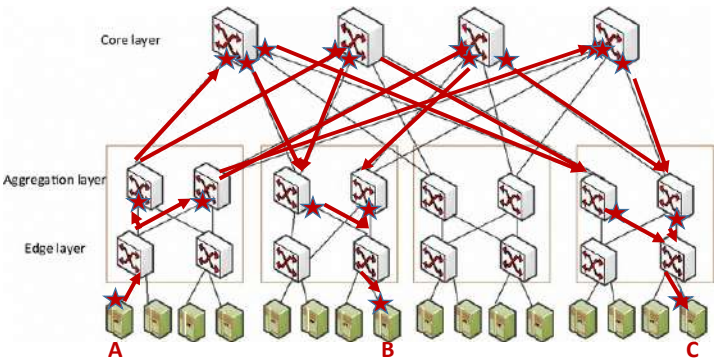
A sends a multicast

- Suppose we want to publish some event from *A* to the "A Message Bus" "group"?
- *A* prepares a UDP packet, puts the special address in it, and calls sendto
- At each stage it will be forwarded towards any receivers

Bloom Filter role?

- At the "line rate" of the network (75 M packets/second) we have very little time to decide where to forward copies
- The Bloom filter can be implemented in hardware (the hashing policy is the expensive step) and runs fast enough to make the decision before the switch or router exceeds its available time
- So we get a very clean UDP multicast **that might show up multiple times per receiver**, but will not bother non-receivers

Example: Nodes *A*, *B* and *C* are in IP multicast group of "A Message Bus"



***B* gets *m* twice, and *C* gets it four times, but at least it was reliable!**

…nobody else got it at all

Some uses

- Maybe "A Message Bus" is super popular
  - **Monitoring:** Publish a message for any potentially monitored value. If nobody is watching, the network will filter them away!
  - **Debugging:** Publish a message for any kind of debugging purpose. If debugger is not in use, the network will filter them away!
  - **Other kinds of news reporting…**
  - **Any sort of messaging-style application…**

What does this tell us?

- **In a small data center, the approach will work amazingly well**

- Our Bloom filters have 16 kbits, we want no more than about 1/4 of the bits set, so we can handle about 4 kbits different pub-sub topics
- In a small data center we are not likely to hit this threshold

What does this tell us?

- **When the datacenter was small, it worked awesomely**
- But then the boss decides to scale up by 3 times
- Yesterday our data center "used" 4 kbits out of the 16 kbits, and now it uses 12 kbits, or more
  - **Suddenly we get a lot of false positives**

False positives

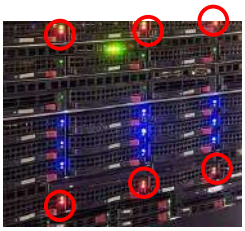
- As we scale up the data center, more and more of the UDP packets are going to be forwarded to more and more machines, due to Bloom Filter matches
- **In effect we go from the network filtering out "no receiver" packets to forwarding every packet, many copies each, on every link**
- This overloads the network, and it becomes lossy, it may also overload individual machines if some machines are listening for many IP multicast addresses

We call these multicast storms

- The term refers to an all-to-all message pattern that overwhelms the entire data center
- Basically, a single event ended up crashing the whole datacenter within seconds
- Huge percentage of messages get dropped
  - All the machines see a huge overload, they are receiving packets they did not subscribe to, and must "manually" discard them, which takes time
  - The whole data center grinds to a halt
  - Lots of other services begin to get timeouts due to unresponsive servers, causing even more errors to report

In fact, it crashes the whole data center

- When this happens, the load is so extreme that every computer running this logic, and all their routers, get swamped
- But those are **also** the hosts and routers for other services
- Those end up crashing too, even though they are not using "A Message Bus"

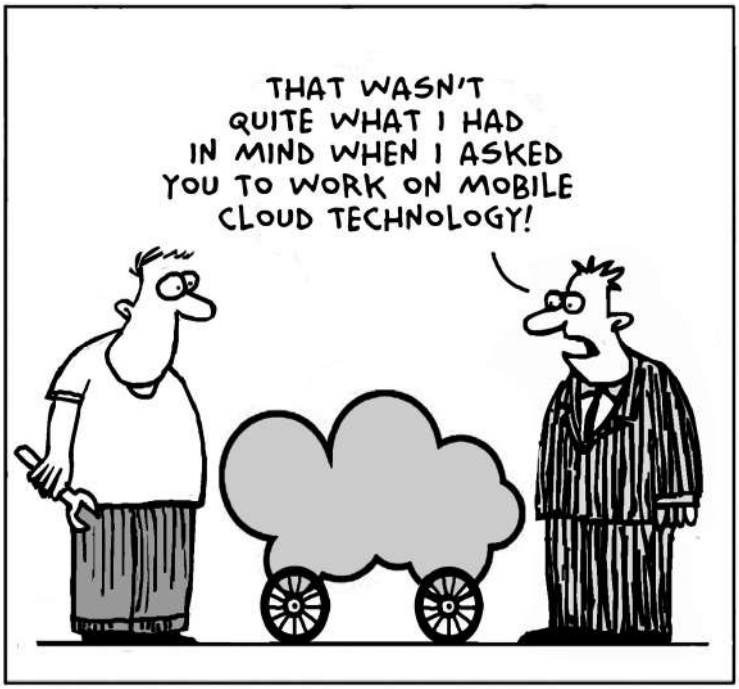


These stories were all real

- For example, modern data centers **disable** hardware for UDP multicast
- If you try and use UDP multicast on a cloud, inside your VPC, the cloud will automatically set up TCP connections and "tunnel" your messages via TCP
- This works so well that people have talked about changing the official UDP multicast implementation to work this way

Summary

- New technology can be tricky to deploy, even the best intentions can blow up and disrupt the entire data center
- Gossip really is very valuable. A well-designed gossip mechanism will have constant, low overheads and very predictable delay, provided the information sharing graph is of low diameter
  - This is what blockchain gossip layers assume
- But careless designs and inadequate testing can yield solutions that actually malfunction in major ways, potentially shutting down entire datacenters



CLOUD SYSTEMS

Blockchain for IoT

Lecture #9

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

What is blockchain?

- **The most disruptive tech in decades!**

- "The distributed ledger technology, better known as blockchain, has the potential to eliminate huge amounts of record-keeping, save money and disrupt IT in ways not seen since the internet arrived."

Lucas Mearian

ComputerWorld Staff Writer

A more technical answer

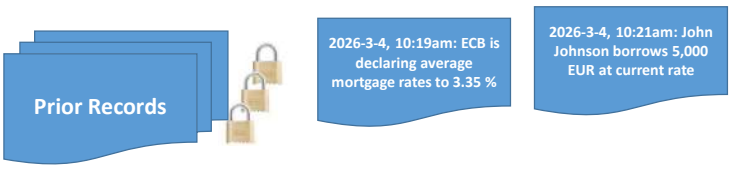
- "A blockchain, originally block chain, is a growing list of records, called blocks, which are linked using cryptography. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data (generally represented as a Merkle Tree root hash).
  By design, a blockchain is resistant to modification of the data."

Wikipedia

What are best uses for blockchain?

- Secure, trustworthy shared log (append-only file)
- Records describe some form of announcement or transaction of broad value to the users
  - There are standard (web-like) languages for recording things in these records
  - Famous example: cyber coin purchases/sales, NFTs for digital art ownership and trade
- Other examples
  - Bank announcements of variable mortgage loan rates
  - Public announcements of weddings, divorces, births, deaths
  - Auditable log of events occurred within an IoT system

How might bank use blockchain?



How might bank use blockchain?

- If the blockchain is public, shared and tamperproof there can never be any basis for disagreement about the information
- The blockchain is publicly replicated to ensure that even the bank can not cheat
- Cryptography prevents tampering

History: Bitcoin as an original idea

- Satoshi Nakamoto in 2008
  - Most likely a group of cryptography activists (Cyberpunks) or their sympathizers
- Basic properties
  - Break away from fiat currencies and create independent financial system (cryptocurrency) based on cryptographic techniques
  - Anonymous transactions
  - Decentralized trust achieved by cryptographic blockchain records (ledger)
  - Proof-of-work as distributed timestamping and prevention of DDoS
- From the beginnings ethics of Bitcoin (and other cryptocurrencies) is a big issue
  - Do we want a technology to support criminal activities e.g. drug selling, pornography, prostitution, weapons trading?
- **After some years a lot of technological advantages of blockchain were identified**
  - Regardless the questionable activism and ethics issues

Terminology

- In blockchain settings, a **transaction** is a digital record describing some event
- Some transactions are complete and self-contained
- But blockchain also supports transaction languages in which one transaction might refer to events defined by a past **or future** transaction

Example transactions

- **Owner**
  d968256677f114410957b0d76dedcb8e862e9858dfef87b8c7893715163574acf724f4a593bacb30f1e639ba63867dced37b5e88a2c7efc463d4fbb570c827dc0286bdb433247530aeb122d3f3c25e0
  transfers coin
  29091fff1d2844a3ace94a46dea3dd441a8d0816bb192dc1622f6caf219e6b38baf5926302bb1a6288a2fc437fdbd3bc3ad27b7615530765f0419a9c004250513f92e0d34d483c52fa1fc816ee6ec767
  to user
  a50179c64f02bec56c854f40c4fd1b04df72a556b652602dd593a46f100160d4ffd5bd256282a10a52453405ac65c75ec19180459e002c1c17acf6041f058158bbac2b10ba8f997f43be0fd58c92a253
  paying a 0.1 % fee to platform operator
  405ac65c75ec19180459e002c1c17acf6041f058158bbac2b10ba8f997f43be0fd58c92a253a50179c64f02bec56c854f40c4fd1b04df72a556b652602dd593a46f100160d4ffd5bd256282a10a52453

Things to note

- This does not "look like code" but in fact could be code
- There are transactional languages specifically for expressing complex transactions
  - HyperLedger
  - Ethereum
- But they also are deeply linked to cryptography
  - The user names, coins, wallets, chain operators are all identified by **"random numbers" (cryptographic keys)**

Roles for cryptography in blockchains

- Blockchains use cryptography to represent operations
- They use it for "irrefutable proofs", such as to seal the blockchain ledger for committed transactions
  - These center on **digital signatures**
- It brings anonymity
  - "It is not anyone's business what I am buying or selling"
- It avoids lock-in to specific national currencies, oversight rules (laws), political systems, patriotic goals, etc.
  - "No fiat roles"
- But it is not easy to prosecute a fraud
  - A stolen coin can quickly vanish into cyberspace
  - And conversely, with enough compute power, a big country might be able to **deanonymize** by connecting users to externalities

But this is not the same as anonymity in a public setting

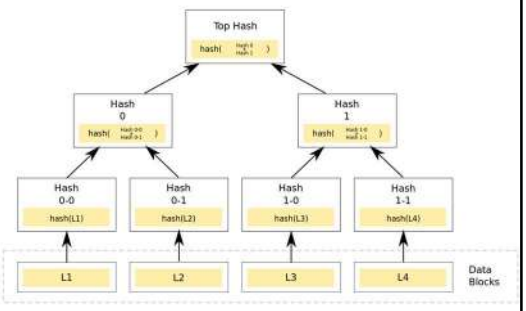
- In the real world, you are anonymous if nobody knows your identity
- But with blockchain, the main uses are **financial**
  - Create or purchase coins, sell them, wallets (like bank accounts), etc.
  - The blockchain can be anonymous, **but when you convert cash to coins or coins to cash, that links the chain to the outside world**
  - A smart investigator can link the two "graphs" and deduce a lot about you
    - Tax evasion or money laundering could actually be traced
    - Widely used by current crime investigation organizations

Blockchain record

- In the case of blockchain, each block contains a set of transactions represented as binary records
  - These records might be huge, hence slow to hash (and very slow to encrypt, were you to try that)
- By having the creator store the record someplace reasonable and then just storing signatures in the blockchain, we use it as efficiently as possible

Merkle Tree: A tree of signed records

- Rather than making one list of *N* records and then hashing them, we often create a binary tree of hashes
- Very common in blockchains, permits us to run SHA-256 or SHA-512 in its fastest "mode" of operation

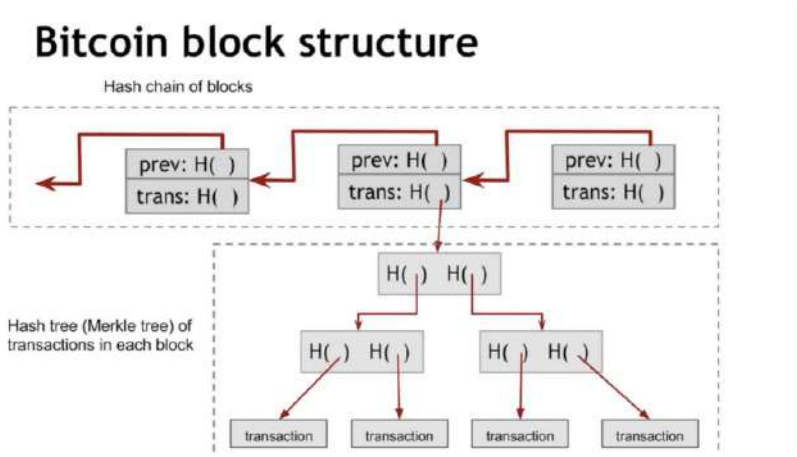


- Often, we think of the entire block chain as a sequence of Merkle trees that change (only) by appending new subtrees (which also changes the root node)

This already gives us a basic solution

- Compute a series of records, each containing transactions signed by the initiator
  - The record needs to include the "name" of the initiator so that anyone needing to do so can look up the matching public key
- Associate each record with a key for lookup, and insert the (key, record) objects into the Merkle tree
- Then create some form of cryptographic proof that the new tree extends the prior tree
  - Logs are just one possible representation

Basic solution



Why is it secure?

- If anyone tampers with any record in the chain, we can sense this by recomputing the Merkle tree
  - The signature will not match
- To verify the entire chain, block by block recompute the Merkle tree, then recompute the sequence of pairwise hash values
- Verification is required when an untrusted blockchain is initially loaded

Permissioned / permissionless

- Blockchain solutions split into two categories
- A **permissioned** blockchain is managed by an authorized group of servers, which could be geodistributed or inside some datacenter
  - We generally assume that they do not have true Byzantine faults
- A **permissionless** blockchain is managed by an anonymous group of servers that volunteer to play the role, and might come and go at will
  - It must be able to resist Byzantine faults/participants

And two more sub-cases for the permissioned case

- **Permissioned blockchains** can arise in two settings
  - **In a data center**, just use Paxos plus logic to compute the needed cryptographic signature chain
    - WAN mirrors protect against attempts to compromise the whole data center and rewrite the whole log
  - **At geoscale**, a permissioned blockchain just means the miners get an authorization to mine – it can be withdrawn if they misbehave
- When people say "permissioned blockchain" they normally mean **"in a wide-area environment, with Byzantine behavior controlled by the permissioning key-management technique"**

Attacks are an issue

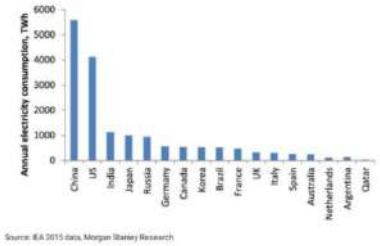
- WAN replicated blockchain can come under many forms of attack
  - Compromised server might try to hand out "fake" versions of the chain
  - It might try to generate huge rates of transactions on its own, and not include your transactions
    - This is a form of DDoS attack
  - It could try to corrupt individual transactions or records

Proof-of-work (1)

- A proof-of-work mechanism adds one more field to the blocks – a **nonce**
- The nonce is just a bit string of some size
- The rule is that to append B_{k+1} to the chain, in addition to hashing it with the hash of the prior block, P must also find a nonce such that when the nonce is included and a new hash is computed, the hash value ends with some desired number of 0 bits
  - Solve a so called cryptographic puzzle

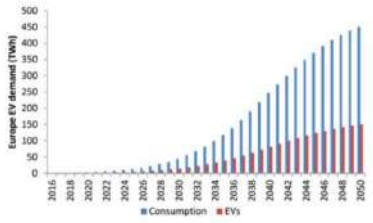
Proof-of-work (2)

Exhibit 1: According to the IEA, global electricity consumption is c.22k TWh annually, with cryptos annualising nearly 40TWh (in line with Qatar), but could get to c.125TWh in 2018 (in line with Argentina)



Source: IEA 2015 data, Morgan Stanley Research

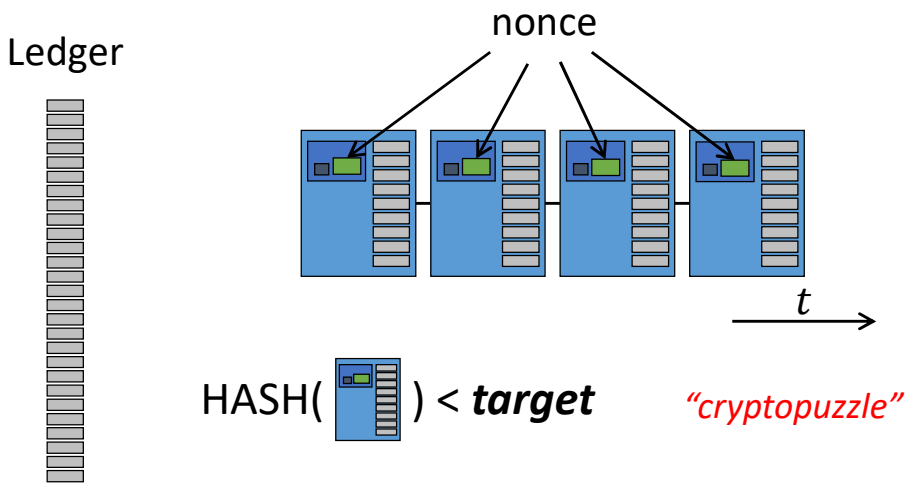
Exhibit 2: To put this in to context, we see EV electricity consumption in Europe at 1-2TWh presently, and 25TWh by 2025, with global EV demand at 125TWh only by 2025



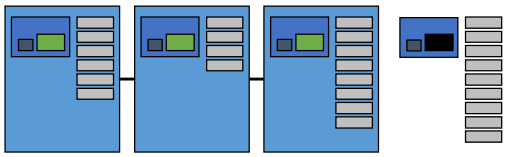
Source: Morgan Stanley Research estimates

- Finding such a nonce is hard work
- So, while *P* could be keen to append its block, it may need to search for this nonce for many seconds or minutes
- The difficulty of finding a nonce value that will work prevents DDoS attacks

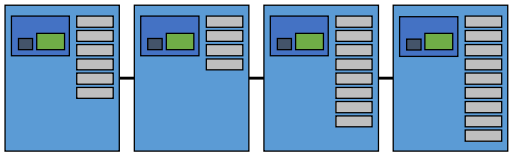
The blockchain



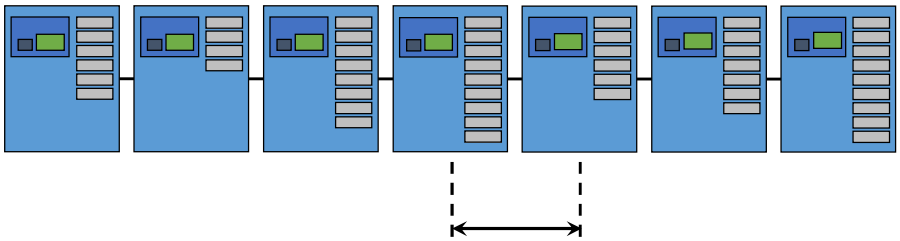
The blockchain



The blockchain



The blockchain



Exponentially distributed rate of new blocks, with constant mean interval

Target automatically adjusted every 2016 blocks so that mean interval is **10 minutes**

Witness and proof-of-stake

- Ethereum has shifted to a new model
- A pool of "witness" servers is running
  - Anyone with enough money in a cryptographic wallet can join the pool
- To commit a transaction, obtain a majority set of witnesses for your record (or for the block containing your record)
  - Based on deterministic consensus

How most blockchains work today

- You can download the software as a VM or even compile it yourself
- You launch the code on your own servers – this makes you a "miner" as soon as the system has initialized itself
- **The system downloads the entire current blockchain**, from other machines already running the blockchain software
  - There is a web site listing some you can contact for copies

Why the whole tree?

- There is a lot of research (meaning, a huge amount) on ways to download and verify just a portion of the blockchain
  - None is really secure, yet
- In some settings this is going to be absolutely obligatory, but right now it is not how the big public blockchains work
- The most promising approaches provide "countersignatures" for the portion you want, provided by authorities you happen to trust

Next, you verify the blockchain

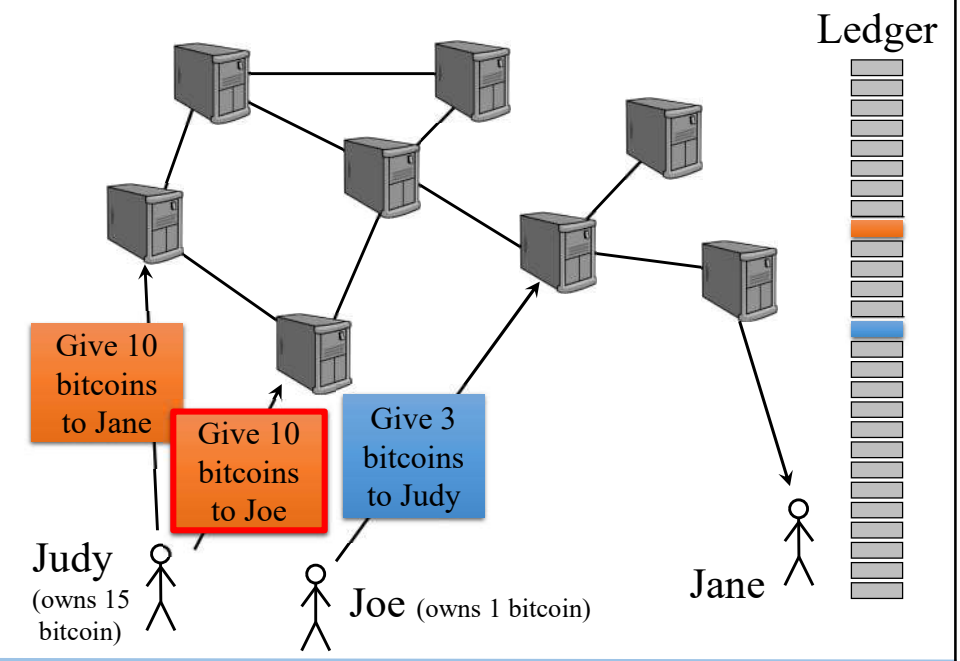
- You will need to recompute all the Merkle trees and the chain of hashes
- In fact this may not take enormously long today
  - Few blockchains have huge amounts of content
- But someday, we might have blockchains with hundreds of billions of records and total sizes in the petabytes
  - Then download speed and verification time and storage will become an issue

Meanwhile, new blocks arrive

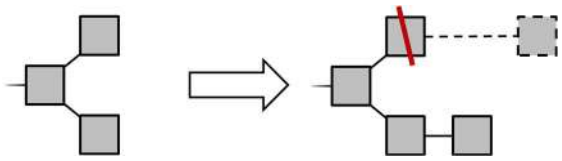
- Each block extends some specific sequence of prior blocks
- If you turn out to have downloaded the wrong sequence, you may have to truncate your chain and download the longer sequence
- This is a “rollback”
  - During startup, substantial rollbacks can occur
  - Later they should not (assumes a fully connected network of mostly “correct” miners)

At this point you can create transactions

- You will generate the transaction (think "credit card payment slip") and submit it to the system
  - It enters a pool of pending transactions
- When will your transaction go through?
  - Within an hour or so, you should see that your transaction got included into some block, and also that everyone seems to have adopted that block
  - The chain has moved six or more blocks into the future

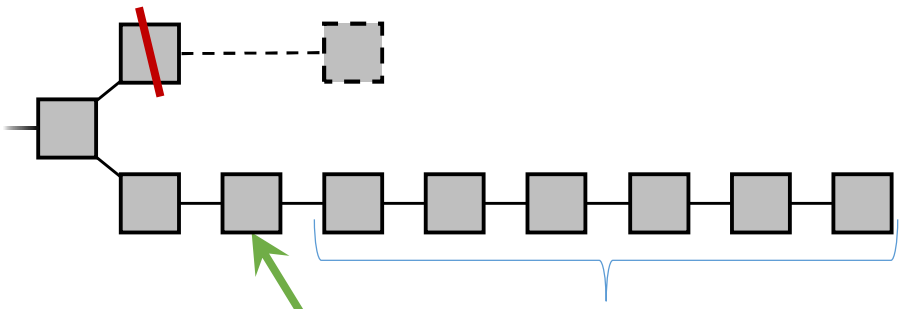


Forks



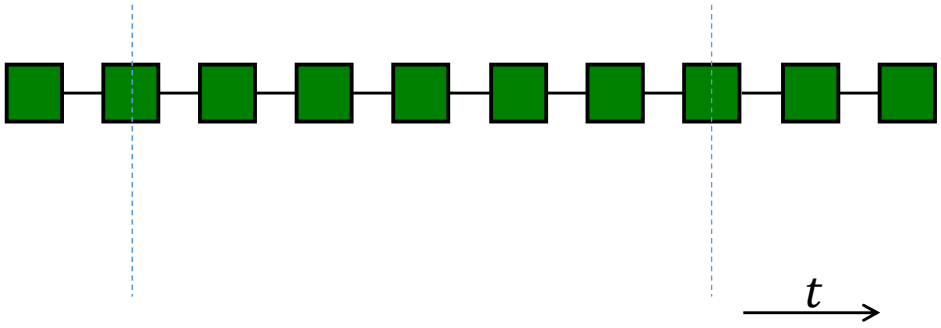
- Rule: Adopt the longest chain your system learns about
  - A fork arises when two miners concurrently discover candidates for the next block
  - Some adopt one, some adopt the other
- When an extension to one chain is found, the shorter chain is abandoned (unless someone shows up with an even longer extension that builds on the short one)

Fork resolution

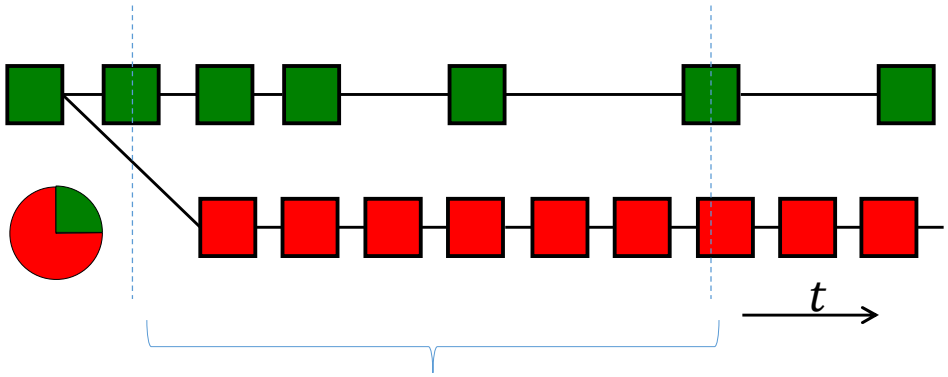


A transaction is **confirmed** when
it is **buried** "deep enough"
(typically 6 blocks – i.e., one hour)

Security threat



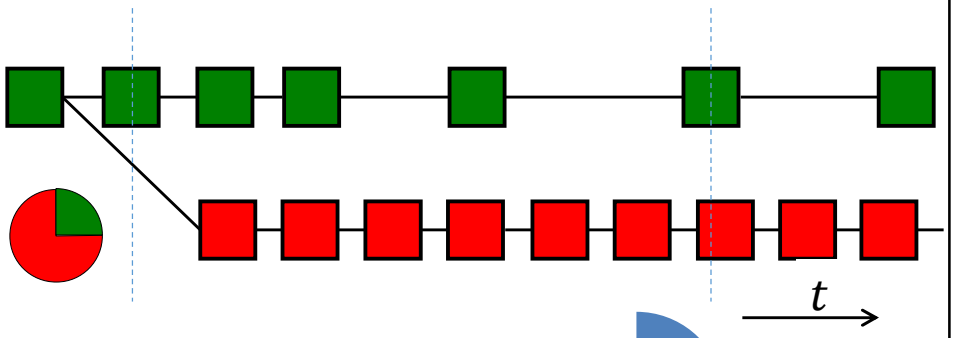
Security threat



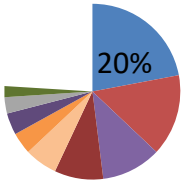
Red (attack) generates 6 blocks in time that the good miners generate 4

Threat: attacker outruns good miners, causing their transactions to roll back

Response: An assumption

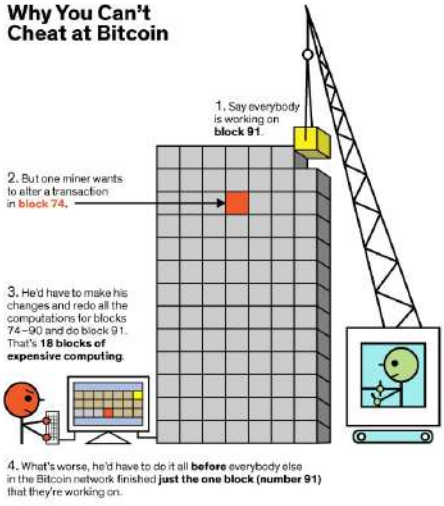


Security Assumption: good miners own >50 % of the total compute power



But nobody can cheat

- To modify a past record you need to also modify every signature subsequent to that record
- The step where you have to find these nonce values will be very slow and you will lose the race



What if attacker is a country?

- A country could build whole datacenters, equipped with hardware to compute SHA-256 at ultra-high speeds
- In this case *P* (using the datacenter) could generate a lot of blocks quickly, for which they would be paid
  - Or could have an entire second blockchains starting from months ago, and longer than the official main one
- To prevent most such attacks, blockchain solutions make the proof-of-work task harder as a function of the rate at which blocks are being found

What if attacker is a country?

- In effect, if *P* controls enough computing power, he can "gain control" of the blockchain
  - The proof-of-work can become so hard that only *P* has the compute power to solve the puzzle
- *P* could then refuse to post some transactions, or cause trouble in other ways
- But this form of attack has not (yet) been seen

What about races?

- Permissionless blockchains are at risk of a "race" situation in which one group of miners is working to append record *R*, and some other group, record *S*
  - A tie can easily occur
- Blockchains systems "adopt the longest chain" (may the best miners win)
  - This can cause a rollback if a few blocks were appended by group *A*, but then group *B* suddenly publishes a longer extension
- In practice, rollbacks longer than 6 blocks are never observed

Permissioned blockchains does not need proof-of-work

- A permissioned system is operated by known, trusted, authorized servers
- They will not attack the chain by trying to overload it with transactions in an unfair way, and they would charge for any transactions they append on behalf of external clients
- So we can avoid this costly step with datacenter blockchain solutions

What if you do not really trust the permissioned provider?

- We can mix methods: a global "proof" with a local "data store"
- Our permissioned provider can commit to some form of cryptographic root of each new version of the log (or tree), and to a proof that the new version extends the old version
- The "commit" is broadly shared and pins the provide down
  - Then for an append or a query, the provider can be asked to also provide a proof that they did the append, or that the query response is correct and complete

What is in a transaction?

- Some blockchain systems are very rigid
  - For example, a Bitcoin blockchain record can only support a few operations on Bitcoins
- These represent transactions: "John sells Peter a packet of cigarettes for 10"
  - In a permissionless scheme, John would probably wait for a while before handing the cigarettes to Peter
  - With permissions, rollback risk can be reduced or even completely eliminated

Fancier transactions

- There are several standards for encoding fancier "digital contracts" into records suitable for blockchain
- HyperLedger, uses HTML as its underlying "language"
- Ethereum, has a sophisticated language of its own, and can even encode computational tasks into the transaction record

- A really bad consequences: Unfortunately, it appears a lot of blockchain records currently store illegal content (e.g. pornography) and it is impossible to remove it

Validation issue

- In existing blockchain systems, every participant maintains a replica of the entire blockchain
- But as blockchains grow, this could become unworkable
  - A blockchain might be hundreds of gigabytes long, users will want to download portions
- If a user only downloads the record they are seeking, how can the system validate that the **entire chain** is intact and properly signed?

More issues

- With permissionless blockchain, is it really "safe" to trust that after six blocks have been appended, the chain will not roll back and invalidate my transaction?
  - "When should John give the cigarettes to Peter?"
- If a smart contract references future events, what would be the "semantics" of that contract?
- Does the meaning depend on waiting for the future to occur?
- Can chains of dependencies arise, or contracts that are undecidable, or infeasibly complex to "evaluate"?

A really big issue

- Quantum computers are advancing rapidly
- But many blockchains use cryptographic techniques known to be at risk if practical quantum computers emerge and can deal with really large quantum computations
  - Currently used cryptosystems: RSA or ECC
- People are asking: Is this becoming a real risk?
- Research on post-quantum cryptography
  - There is a theory of **semantic cryptography safe against quantum attacks** (developed by Goldwasser and Micali)
  - They proved that secure encryption schemes must be probabilistic, rather than deterministic, with many possible encrypted texts corresponding to each message
  - Latice-based cryptography

Today's limitations

- Blockchains work well for basic financial tasks, NFTs
- But they are hard to use for records from IoT settings, or collaborating companies that hoped the chain could be a shared ground truth
  - Rollback can be problematic
  - Huge energy costs of proof of work are a concern
  - The blockchain is not a "database" but companies want to treat it as one

Walking through the issues

- Which issues would arise on an industrial IoT application?
- Would a blockchain solve those issues?
- What **new** risks would it introduce?
- What limits the speed of new technology adoption?

Blockchain for an (industrial) IoT

- Main uses seem to be for audit trails of various kinds
  - Capture data about something we are supposed to trace or record
  - Write it digitally into the ledger, securely
    - Tamperproof and automatic
  - Auditors given access to the record
- But they will want to know
  - Why should I trust this blockchain record?
  - Is the underlying data valid?

Blockchain for sensors

- An IoT would store blockchain records that include sensor data
- Very likely the sensors themselves would be securely connected to their host machines, for example using Azure's IoT Hub or the AWS equivalent
- With an IoT Hub model, the sensor itself uses security keys, and will only connect and talk to the hub
  - This enables a **digital twin** model

Trust with sensors

- We can assume the connection to the sensor is secure
  - But how would a digital twin for an IoT work?
- Azure IoT Hub will only allow authorized sensors to be part of the system, and it patches the software and configuration automatically
  - Feeds events to the Azure IoT function server, where functions consume them
- So presumably these functions log the event records to the blockchain

Questions this might not answer

- Is this sensor the correct one for the information the application claims to have captured?
- Is the sensor working correctly?
- Has the data the sensor generated been modified before it was logged?

Can we answer the questions an auditor might ask?

- For example, a sensor records show
  - Cow 2143 was milked on Tuesday at 10am
  - Later the milk turned out to have a dangerous infection in it
  - It got through and a consumer became quite sick
- Was she properly clean when she was milked?
- Had she been evaluated for deseases as required by the health department?
- Was she periodically checked for overall health?
- Did she receive any "off records" meds?

Can we answer the questions an auditor might ask?

- Realistically, an audit using sensor data would provide "evidence" but cannot answer these questions
  - Non-technical people might find this surprising, but in cloud computing we have seen why many either **cannot** be answered, or **we cannot have confidence our answers will be correct**
- **Blockchain does protect against tampering, and does record what the sensors reported, and when**
  - **This is already valuable, as long as we are honest about the capabilities and limitations**

To have real confidence

- Azure IoT Hub would need to log **management** events too
- The audit-trail examination tool would need to visualize this information and be able to convince the auditor that yes, this is the proper sensor, it seems to be properly calibrated, the data was not tampered with, etc.
- The mention of time suggests that we might also need to log events related to the way the system tracks time, or at least have a "story" there

Connectivity issue

- A lot of the "events" that matter in IoT setting are in remote places, **disconnected** from the main system
- The blockchain would probably be used primarily as an audit trace, to track events in the production chain from sources to product
- So this raises issues like intermittent connectivity, how we know that the sensor that generated a record is the "correct one" for that role, etc.

IoT use case: Food chain supply?

- Supply chain management
  - Walmart is building one for the food supply chain
    - Food safety: fast identification of tainted foods
    - Consumers are demanding more information about the products they buy (organic, fair trade, etc.)
  - Simplify international transactions
- Help farmers
  - Want to know what happens to their products for fair pricing
  - What products should they be producing?
- Reduce food scandals
  - Illegal production, misrepresentation, loss and waste, etc.

Industrial uptake?

- ripe.io
  - A company that is building a “blockchain of food” with IoT interfaces
- Walmart
  - partnered with IBM and Tsinghua to identify sources of contaminated products and speed up recall
- But today’s blockchain technology may not be appropriate for all use cases
  - Too dependent on availability of plentiful power, networking, and storage

Desired blockchain properties

- Performance
  - High throughput, low latency
  - Energy-efficient
- Security
  - Always available for reading (verifying) and appending
  - Fair
  - Tamperproof (integrity)
  - Possibly confidentiality as well
- No single administrative domain
  - **No need to trust a single provider**
- Open membership (or not)

Open membership is hard

- Traditional secure logs are based on voting
- Members vote on which transactions to add to the log and in what order
- **Problem: "Sybil" or impersonation attacks**
  - A participant may try to vote multiple times
  - With closed membership, cryptographic signatures can identify the source of a vote
  - With open membership, anybody can create identities and that way vote many times

Permissionless vs. permissioned

	Permissionless	Permissioned
Approach	Competitive	Cooperative
Basic technique	Proof-of-Resource	Voting
Membership	Open	Closed
Energy-efficiency	Often terrible	Excellent
Transaction rate	At best hundreds / sec	Many thousands per second
Transaction latency	As high as many minutes	Less than a second

Permissionless vs. permissioned

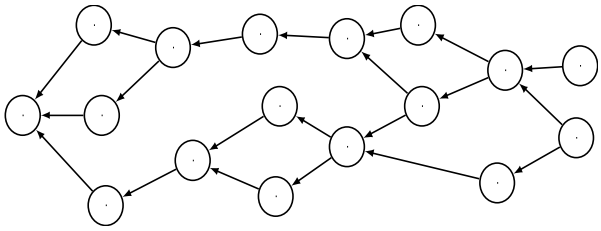
- Permissionless
  - Open membership, but inefficient
  - Vulnerable to 50% attacks
  - Examples include Bitcoin, Ethereum, IOTA
- Permissioned
  - Performance
    - High Throughput, Low Latency
    - Energy-efficient
  - Security
    - No forks!
  - **Closed membership**
  - Examples include Ripple, Hyperledger

Properties of blockchain for the IoT?

- Blockchains require strong network connectivity and lots of storage
- Permissionless blockchain are power-hungry
- Sensors have limited resources
  - Sensors for growing conditions, storage conditions, shipping conditions, etc.
- Blockchain for an IoT will generate records in a decentralized way, and hence it **must** work in a network-partitioned or network-challenged environment

Vegvisir: Tolerate branches

- https://vegvisir.cs.cornell.edu



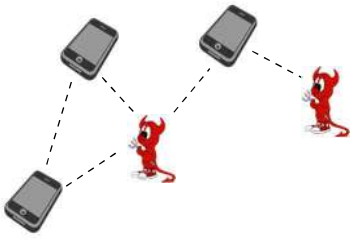
- The key innovation is to allow branching as a feature
- Leads to a graph (DAG) structure instead of linear blockchain
- DAG is a full causal history of events (respect Lamport's $\rightarrow$), but note that the DAG arrows run in the opposite direction than Lamport's $\rightarrow$

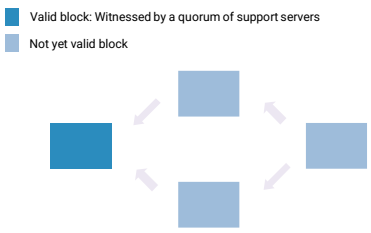
What about ← notation?

- When Lamport talks about $a \rightarrow b$, this is a statement about causality
- In the Vegvisir DAG, nodes represent operations but the notation is reversed: $a \leftarrow b$, meaning "b came after a"
  - They made this choice because b "countersigns" a, in the actual blockchain
- As long as you understand the notation it is easy to understand their examples

Proof-of-witness to persist blocks securely

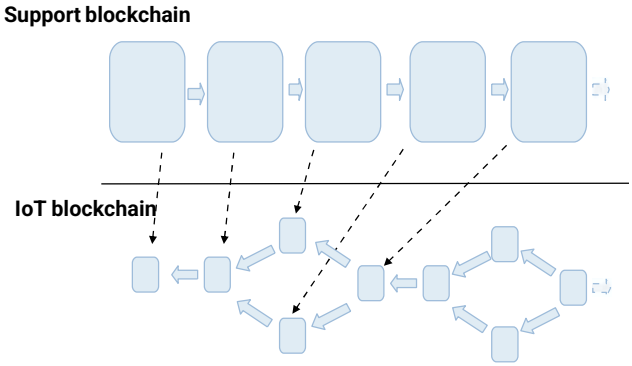
No more than k malicious nodes in any neighborhood





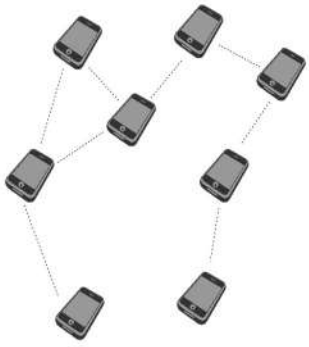
At least one copy of a valid block will survive if <k malicious peers

The Support blockchain reduces sensor storage needs



Allows regular peers to discard old blocks when storage space is low

Blocks are **gossiped** over ad-hoc network



Heterogeneous, opportunistic networking

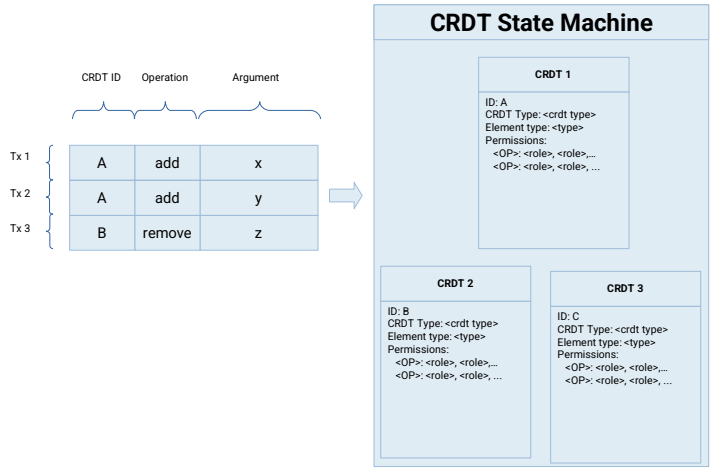
What would you find inside a record?

- There are two kinds of transactions
- One kind spends coins – these need to validated using a total ordering
- The others are commutative and the DAG ordering is sufficient
  - These are composed entirely from CRDT operations

CRDTs: strong semantics in DAGs

- **Conflict-Free Replicated Datatype**
- Updates must be associative, commutative, idempotent
- Replicas can be updated independently and concurrently
- Basic CRDTs form registers, counters, sets

Transactions manipulate CRDTs

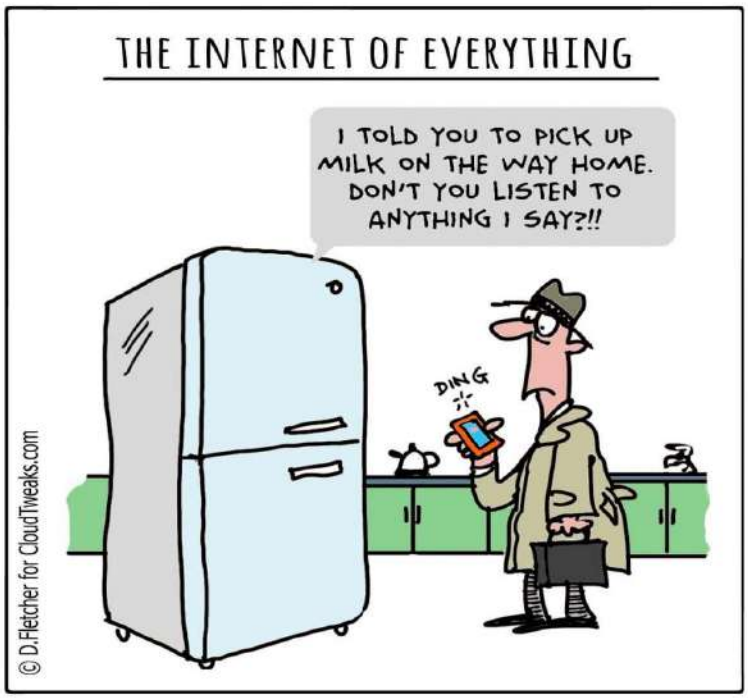


But some questions remain open

- When we merge the DAG to form a total order we might discover double-spending of cybercurrency
  - Vegvisir does not currently offer a solution here
  - It is easy to impose a total order, but only during periods of connectivity
- Vegvisir also leaves open many of the other questions we touched on (can we trust a sensor?), that an auditor might want to see answered

Conclusion

- Exciting possibilities for blockchains in the IoT (e.g. food supply chain)
- But current blockchain designs may not be compatible with some deployment scenarios in the IoT
- For example Vegvisir supports partitioned operation and has low power/networking/storage requirements



CLOUD SYSTEMS

Smart Contracts with Multiple Organizations

Lecture #10

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

Broad overview

- In the previous lecture we learned that
  - It is appealing to use blockchain for IoT data in settings like farms
  - This poses challenges not seen in cryptocurrency settings, like intermittent networking
  - We can extend the blockchain model to solve those problems, e.g. Vegvisir did this by tolerating forks (using a dag model) and introducing a support blockchain operated by witnesses (permissioned with proof of stake, not proof of work)
  - But that was a research system, today there are no products like Vegvisir

Broad overview

- We have mentioned hospitals that might use a blockchain to assist in electronic medical records management
- What **exactly** are the challenges here?
  - They include maintaining confidence that the EHR database and the blockchain are properly synchronized, and semantics of smart records
  - There is also a broader issue of trust, e.g. the Stellar system is one effort that tries to address the trust problem

Brainstorming about a **real** blockchain use case

- Consider the challenge of running a multi-hospital consortium structure in which one patient might be seen by physicians at more than one different medical center
- A patient with a complex condition could have treatments in all three
  - Mrs. Smith is admitted for abdominal pain at Hospital 1, various diagnostic procedures are performed, Hospital 1 starts treating an infection
  - She turns out to require surgery, which is done in the gastroenterology surgical unit at Hospital 2
  - There is a follow-up cancer treatment at the Hospital 3, meanwhile, she also has infection follow-ups at Hospital 1, and surgical post-op visits to Hospital 2

The issue? Three sets of medical records

- Each of these organizations has a distinct electronic medical record management system, which for legal reasons (GDPR in Europe or HiPPA in USA) must be managed by each individual hospital
- Yet each also needs a way to see the records created by the others, in order to ensure that the complex condition Mrs. Smith is being treated for is correctly managed
- GDPR does not deal very well with this form of sharing

Current approaches and issues

- Let's say, each hospital maintains its own distinct records
- Sometimes these records disagree in ways that may have billing consequences, so hospitals employ full time staff to resolve such issues
- Causes include different opinions about a diagnosis, errors entering data, different rules employed by the different hospital IT systems, etc.

Medical records and the cloud

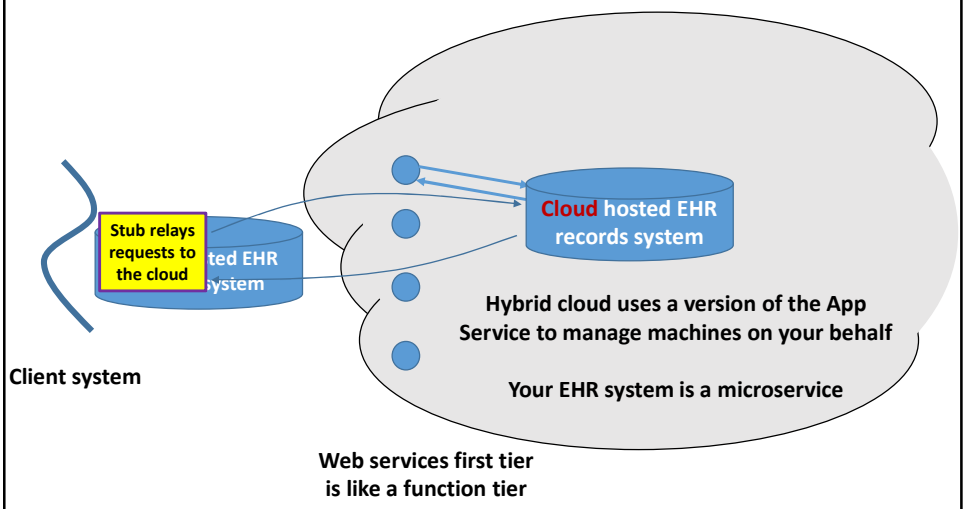
- What roles can the cloud play in medical record management?
- In early EHR systems, the answer was very simple: **none**
  - For a long time, medical records systems were treated as "on your own premises only"
- But over time, this became more and more expensive and unworkable
  - Eventually, a form of hybrid cloud emerged

Hybrid cloud

- **A hybrid cloud is any system that combines elements from the home system and the cloud, or even from multiple distinct cloud providers**

- These could run on the home system or be migrated to the cloud
- For example, a company could take a database that used to run on its own servers and "port" it to run on AWS
  - Hybrid cloud allows them do this in a very transparent way

Porting an enterprise system to a cloud



Porting an enterprise system to a cloud

- We end up "redirecting" what used to be remote procedure calls from clients to the home-hosted server
  - Now they become web-service requests forwarded to the cloud, and the answers come back in the same web-service format
- But this can be hidden in the library used by clients to talk to the service
- In principle, you just recompile and things work as they did previously

Why was this beneficial?

- Notice first that it could actually be slower than the original solution
- But modern networks are fast, and network delay is not the only concern
- The cloud has
  - Cutting edge hardware
  - Automated management tools
  - Sharing (amortized costs)
- Other advantages for a hospital
  - By adopting the cloud, the hospital can change the computer room into some other use of the space, like a surgical suite
  - The IT staff can become more effective and will not need to do as much hands-on management of the servers
  - Upgrades are done by the cloud vendor, not by the IT staff

Downsides?

- The hospital cannot access its own EHRs unless the network is up and the cloud provider is operating normally
- Security and privacy aspects of GDPR
  - What if Amazon has an evil employee who plans to sell data about celebrity patients?
- What if the disk on which data is stored gets upgraded and the old one is not wiped clean?

Some reassuring cloud features

- The web connections use TLS (https)
  - All the data is encrypted
- The cloud itself uses the Virtual Private Cloud concept
  - The microservice lives in an isolated “security environment”
- All the files stored by your application are automatically encrypted by the file system before being written, and the keys are kept at your home system, not on the cloud
  - So, if the disk were stolen, it cannot be deciphered

Technology really can help

- By splitting secret keys among multiple repositories, we can guard against failure that could otherwise destroy a key, while also ensuring that *k* out of *N* portions are needed to reconstruct the actual key
- In a secure cloud, keys are not kept in memory for more than a few microseconds
  - Instead, any key we will use for a long period is "mapped" to a different key managed by the hardware keys built into the trusted computing hardware module (TCB)
  - This temporary key can only be used on a particular machine, with the help of the hardware itself

More reassuring features

- There are also human-factors steps taken by the big vendors
- They ensure that any person who has access to secure keys does not have access to the physical data center, and that any person who has a way to modify the operating system code cannot access keys or hardware
- There are also steps taken to ensure that connectivity from the home site to the cloud is redundant, from multiple ISPs (tunneling over mTCP)

Hard disk drive "shredder"

- To reassure customers, cloud operators have begun to contractually commit that used storage devices will be destroyed
- This small device will shred a hard disk into recyclable metal waste
  - It even has (in some countries) a government seal of approval

Where does this get us?

- Individual organizations, like hospitals, have some elements of their electronic health records on the cloud, and the trend is to move more and more functionality over time
  - GDPR concerns slow this down, but it is advancing even so
- But even if they happen to be on the same cloud, they cannot just share records with one-another other than by creating specialized mirroring

Hospital *Y* treats a patient from Hospital *X*

- In today's approach, each pair of cooperating hospitals establishes a form of secure replication channel
- For example, *X* might send records about Mrs. Smith to *Y*, specifically needed for *Y* to carry out this treatment
- *Y* might then share back some of its records of the treatment, so that *X* has the needed data to track her overall plan of care

X and *Y* must audit all sharing

- Each hospital has a legal obligation to track which records have been shared (outgoing or incoming), and precisely why
- This extends to other organizations too, such as laboratories that do testing, private practices where physicians might see some patients in an office setting, etc.
- It is natural to think of blockchain as a technical tool for such uses

Blockchain: A "partial" match

- The tamperproof audit trail aspects are a good fit here
- We can track these EHR transfers in a blockchain, and it gives us a proof of which records moved from institution to institution
- We can also track individual accesses to those records
  - Now we know that Dr. House accessed the records of Mrs. Smith from hospital *Y*

The real problem is that a blockchain is not a database

- For higher level machine-learning tools that could ask "is it appropriate for Dr. House to make this access?", we would need more of a database representation
- In fact, in general, both human and computer users of this data will want to query it
- But a blockchain is not a database

There is a lot of work to close the gap

- Many companies offer SQL interfaces so that their blockchains can be viewed as if they were databases
- This is not always very performant, but by creating secondary indices (like our B+ tree from second hardware), we can ensure that common query patterns are executed very rapidly
- Then an administrator could, for example, check to see whether Dr. House made this access as part of a treatment activity

Would cloud safety features stop the exploit?

- A hacker wants you to **think** everything is being audited and tracked, but **actually** he plans to steal Mrs. Smith's health records and sell them without being caught
- If the records themselves are encrypted, he might be blocked, but a legitimate medical access might be delayed or blocked too
- He might modify the audit-update or query code to try and fool you
  - Perhaps he takes the legitimate sensor records, changes them right away, and then logs the modified versions
  - Is anything checking this risk?
- "Chief security officer: Check that all EHR accesses are properly authorized."
- "System: Confirmed, no problems detected."

A skeptic might want more proof

- …but here things get very difficult
- Does GDPR allows us to show the skeptic the entire audit trace?
  - What if the hacker gets a job as an auditor?
- Or could we somehow construct a machine-checkable proof that the query response is a correct and complete one?

WAN mirroring can help

- Suppose that as we generate our blockchain, we also make a lot of mirror copies on other datacenters
  - They "countersign" the replicas
- Thus if WAN datacenter *X* logs event *A*, a skeptic can actually fetch the signed log from *X* but can also get countersigned mirror copies at *Y*, *Z*, etc.
- Having many signed copies makes it hard to rewrite entire logs from start to finish
  - With so many copies to check, the auditor will sense tampering

Other settings with similar needs

- Suppose our hospital sees a surge of food poisoning, now the health workers wish to trace all the main events "between when a load of spinach was harvested to when it reached the consumers"
- The team is hoping to pinpoint an event, perhaps, "On the Smith family farm, the spinach cases were transported on a pickup truck that previously was used to transport a load of cow manure"
- But it can be extremely hard to capture enough state to do that

More such settings

- Inter-bank transactions
- Supply chain tracking for industrial activities such as manufacturing plants
- Tracking the maintenance of hardware in the field, for things like power grid technology with very sensitive roles

Many such cases need even more

- The smart contracts concept from Ethereum can appear to be a great fit for banking transactions or other complex kinds of records
- But here we trust that the Ethereum "code" describing the event is correct and unambiguous
- The potential of a blockchain to roll back (instability of the last records) is especially worrying

Example of how rollback causes issues

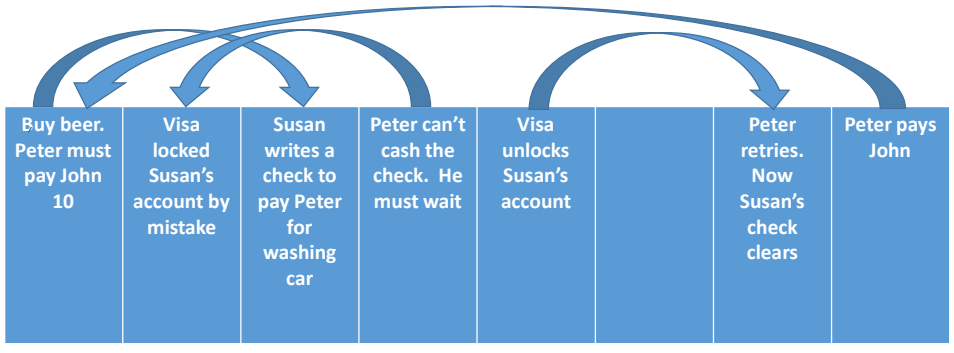
- Suppose that John sells beer to Peter for 10
- Our blockchain record may record that "Upon receipt of 10 from Peter, John will hand Peter the beer"
- But now Peter decides to earn some money
  - He washes Susan's car and for this she will pay him 30, and he will use 10 of this to buy the beer
  - The record is added to the blockchain

We can create endless dependencies

- …actually, Susan wrote him a check and Peter needs to deposit it
- …the bank needs to clear the check
- …Susan's bank account has plenty of money, but is blocked by her credit card, which is trying to collect what they claim is an unpaid bill
- …the credit card company actually made an error and Susan **did** pay the bill (someone else with the same first and last name is in arrears), and they agreed to fix this, but have not done so yet

We can create endless dependencies

- Visualizing these dependencies
- Remember: the blockchain can roll back at any time, erasing the last few records



Cascading effect of rollbacks

- The last 5 or 6 blocks of records are considered unstable, meaning they could be overwritten by other records
- Blockchain becomes stable after about an hour, but in Peter's case, that will not be enough, he needs the **last record** in the chain to stabilize
- Even worse: What if the amount being paid depends on some future thing?

Hospitals and banks should use **permissioned** blockchain with true commit

- The selection of the blockchain product becomes very important
- If you use a permissionless blockchain with rollback, the consequences of rollback become issues for your application
  - If you pick a permissioned blockchain with a fairly quick and firm commit, and no further rollbacks, your application will not experience problems as long as it waits for commit
- Only well-informed developers can safely use these technologies

Future dependency

- Peter, Susan and John are creating a startup company, it will revolutionize delivery services in big cities, and they will split the insane profits
- They call it "delivery.com", even the name will be worth ~~millions~~ billions
- The blockchain record of the agreement might say: "When delivery.com is sold, Peter and John will receive 30 % of the proceeds each, after deducting costs, and Susan will receive 40 %"

…but the "amount" could vary!

- Depending on the records added to the chain in the future, the costs could change, and hence the amounts they will be paid could change
- What are the **semantics** for "costs"?
- When should costs be evaluated?
- When can the transaction finalize?

Can IoT help?

- On the positive side, we can capture records from our IoT devices and store them into the blockchain, too
  - We can even have them signed by the devices to "prove" that this is really what the device detected
- And we could have enough sensors to make it hard to conceal things
- But ultimately, if a hacker understands how the system works, he will not have any real difficulty evading it

Patterns of trust

- One big area of discussion relates to situations where there are multiple blockchains managed by different organizations and you do not have equal trust in all of them
- For example, maybe a citizen only trusts small credit banks, but not big banks
- On the other hand a Wall Street person only trusts really big banks and would never trust small banks
- So how can the citizen get a mortgage that the Wall Street person is actually issuing?
  - Somehow their different blockchains need to talk to each other

Do I trust you?

- In our hospital setting, each big hospital has its own network of labs and providers and insurance partners
- Each **trusts** its own partners, but not the partners of the other big hospitals
  - In fact this is a common **legal obligation**: a hospital must only share EHRs with organizations and people who are trusted to access them
- Can blockchains incorporate a concept of trust?

Stellar

Stellar: Tackling this issue

- A company called Stellar has created a blockchain model with many blockchains
- The key idea is to use a version of RAFT state machine replication protocol to replicate blockchain records across the set of blockchains
- If person P trusts $\{X, Y, Z\}$ then any record associated with person P is required to be stored by a quorum (majority) of the set $\{X, Y, Z\}$

Stellar

Stellar with two companies

- What if a record involves this mortgage that *P* is basically obtaining from *Q*?
- Then we take their two trust sets: *P* trusts {*A*, *B*, *C*, *D*, *E*}, and *Q* trusts {*X*, *Y*, *Z*} (or they can share some trust, this is fine too)
- Stellar asks for a majority from **each of the trust sets**, so we end up putting the mortgage record into a quorum from {*A*, *B*, *C*, *D*, *E*} and also a quorum from {*X*, *Y*, *Z*}
  - Now anyone *P* "cares about" would be able to see the record, and similarly for anyone *Q* trusts

Stellar

Witnesses in Stellar

- This insight leads to a policy
  - A logged record is initially pending
  - When a majority of some trust set witness the record, it becomes trusted with respect to that majority
  - When the record is trusted by all the trust sets from the **transitive closure** of the parties involved in the record, the record is supported and can be treated as a committed database update

Stellar

Challenges with Stellar

- If the trust sets are static, the idea is pretty simple and it is obvious that it will work in a simple way
- But suppose that trust varies over time
- So now the quorum sets are not constant, and we have to "re-replicate" to preserve our invariant
  - With concurrency and failures it gets messy

Stellar

…but Stellar is up and running

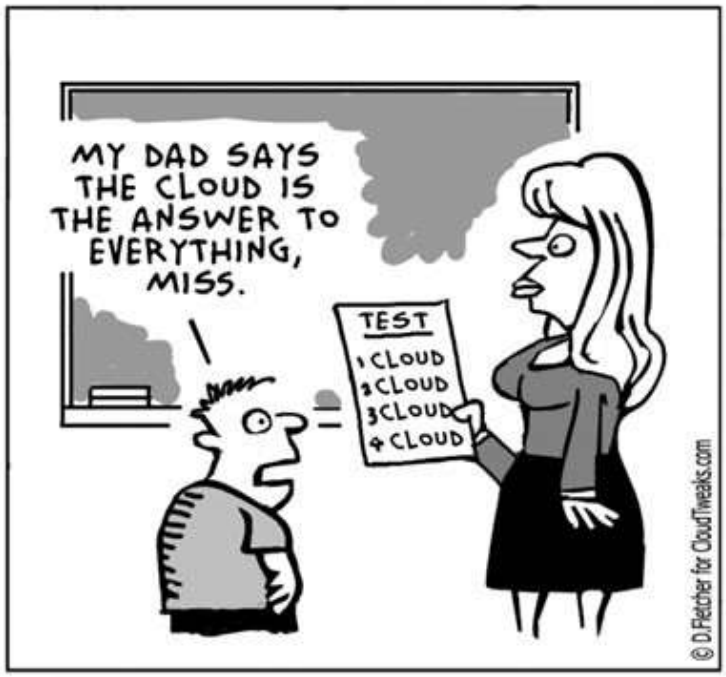
- The company is basically a financial clearing firm for big customers that need to do international money transactions and track them for audits
- They had $10B per day of average transactions in 2022, and are growing rapidly
- This demonstrates demand for **auditable** financial services, not just anonymous ones that evade scrutiny and regulatory oversight

What can we conclude?

- Mostly, the conclusion here is that the enthusiasm for blockchain is way ahead of the reality of the standard systems, but that some of the more mundane uses are exciting, real, impactful and perhaps very beneficial
- Everyone is going wild dreaming up **fancy** use cases, but perhaps the real value is in use cases that look technically easy and kind of dull
- The fancy scenarios are very speculative but the mundane ones are worth pursuing

Summary

- Real world uses that seem ideal for IoT and blockchain are very common
- Yet it can be a real puzzle to actually build the solutions
- Simply seeing a match does not really **solve** anything, the details are hard



CLOUD SYSTEMS

Privacy in Cloud Computing

Lecture #11

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

The privacy puzzle for IoT

- We have sensors everywhere, including in very sensitive settings
- They are capturing information you definitely do not want to share
- Seemingly arguing for brilliant sensors that do all the computing
  - But sensors are power and compute-limited
  - Sometimes, only cloud-scale datacenters can possibly do the job

Things that can only be done on the cloud

- Training models for high quality image recognition and tagging
  - Classifying complex images
- High quality speech, including regional accents and individual styles

- Correlating observations from video cameras with shared knowledge
  - Example: A smart highway where we are comparing observations of vehicles with previously computed motion trajectories
  - Is Cow 2437 likely to give birth soon? Will it be a difficult labor?
  - What plant disease might be causing this form of leaf damage?

But the cloud is not good on privacy

- Many cloud computing vendors are incented by advertising revenue
  - Google just wants to show ads that the user will click on
  - Amazon wants to offer products this user might buy
- Consider medications – it is a big business
  - But to show a relevant ad for a drug to treat mental health, or diabetes, entails knowing the user's health status
- Even **showing** the ad could leak information that a third party, like the ISP carrying network traffic, might "steal"

The law cannot help

- Lawrence Lessig: "East code versus West code"
- Main points
  - The law is far behind the technology curve, in the US
  - Europe may be better (see GDPR), but is a less innovative technology community
  - So, our best hope is to just build better technologies here

Some providers are not incented

- We should separate cloud providers into two groups
- One group of cloud providers has an inherent motivation to violate privacy for revenue reasons and will "fight against" constraints
  - Here we need to block their effort to spy on the computation
- A second group does not earn their revenue with ads
  - These cloud vendors might **cooperate** to create a secure and private model

Traffic analysis attacks

- Some attacks do not actual try to "see" the actual data
- Instead, the attacker might just try to monitor the system carefully, as a way to see who is talking to whom, or sending big objects
- A malicious operator can use this as indirect evidence, or try and disrupt the computation at key moments to cause trouble

Subversion attacks

- These are exploits that leave a system unchanged, but find ways of extracting keys or other sensitive information during normal interactions
- For example, suppose that if I check my account at a bank, various account information and personal information stays on their (cloud hosted) server
- But then suppose that a negative-length web page "put" operation is done and as part of the error code, the server tries to return the bad argument, but instead sends back a snapshot of its memory, revealing my private data

Even an machine-learned model can be subverted

- Machine learning systems generally operate in two stages
  - Given a model, they use **labeled data** to "train" the model (like fitting a curve to a set of data points, by finding parameters to minimize error)
  - Then the active stage takes unlabeled data and "classifies" it by using the model to estimate the most likely labels from the training set
- The model will look like a vector of numbers
  - At a glance it does not seem to encode the information it learned

Inverting a machine-learned model

- But such a model **might** encode private data
- For example, a model trained on your activities in your home might "know" all sorts of very private things even if the raw input is not retained
- It turns out that for certain input, like data that is all 0's or all 1's, or that has all 0's and a single 1, the classifier output will reveal a (noisy) version of the input it was trained on
  - The attacker does a legitimate operation but learns your secrets

Uncooperative provider

- Making things worse, the provider of some service might not really care much about the form of privacy you are worried about
- So, what can we do? If the provider is very cooperative, we could audit their code and design, but few providers are willing to allow that
- Or we could try and run their cloud platform inside a locked box

Sounds pretty bad

- If our cloud provider wants to game the system, there are a million ways to evade constraints, and they may even be legal
- So realistically, with an uncooperative cloud operator, our best bet is to just not use their cloud
- Even hybrid cloud models seem to be infeasible if you need to protect sensitive user data

Categories of responses (1)

1. The cryptography community is exploring **fully homomorphic computing**
   - In this approach, data is encrypted and never decrypted again, but there are still some ways to compute on it
2. A second concept is **differential privacy**
   - This approach involves a database that you trust, which holds the actual unencrypted data, but adds noise to any query results
   - The noise hides individual data but averages out at large scale, so "aggregated" queries return accurate answers

Categories of responses (2)

3. Extending of the hardware by adding a trusted computing module to the machine, holding secret keys and a CPU that can encrypt, decrypt, sign and perform other cryptographic operations
   - Intel turned this into a concept of a secure computing **enclave**

4. Extending any standard database, turning it into a secure database
   - It is a pure software solution and is open source

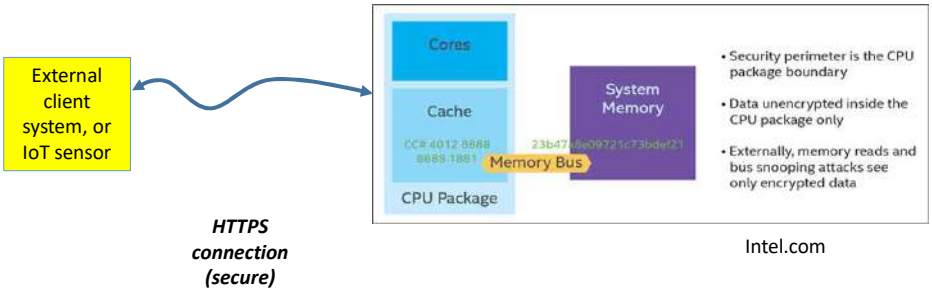
The lockbox model

- Intel has created special hardware to assist for this case: iSGX (Software Guard Extensions)
  - https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html
- Basically, they offer a way to run in a "secure context" within a vendor's cloud
  - If the operator wanted to, it cannot peek into the execution context

SGX concept

- The cloud launches the SGX program, which was supplied by the client
- The program can now read data from the cloud file system or accept a secured TCP connection (HTTPS) from an external application
- The client sends data, and the SGX-secured enclave performs the task and sends back the result
  - The cloud vendor can only see encrypted information, and never has any access to decrypted data or code

SGX example



HTTPS connection (secure)

Intel.com

SGX limitations

- In itself, SGX will not protect against monitoring attacks
  - Via side channel
- And it cannot stop someone from disrupting a connection or accosting a user and saying "Why are you using this secret computing concept?"
  - In some countries even using a system could be illegal
- And it is slow

SGX reception has been mixed

- Some adoption, but performance impact is a continuing worry
- There have been some successful exploits against SGX that leverage Intel's hardware caching and prefetching policies ("Leaks")
- Using SGX requires substantial specialized expertise
  - And SGX cannot leverage specialized hardware accelerators, like GPU or TPU or even FPGA (they could have "back channels" that leak data)

Cooperative privacy looks more promising

- If the vendor is willing to work with the cloud developer many new options emerge
  - Such a vendor guarantees: "We will not snoop, and we will isolate users so that other users **cannot** snoop"
- A first simple idea is for the vendor to provide a guaranteed "scrubbing" for container virtualization
  - Containers that start in a known and "clean" runtime context
  - After the task finishes, they clean up and leave no trace at all

ORAM model

- ORAM: Oblivious RAM (multiuser system that will not leak information)
- Idea here is that if the cloud operator can be trusted but "other users" on the same platform cannot, we should create containers that leak no data
- Even if an attacker manages to run on the same server, they will not learn anything
  - All leaks are blocked (if the solution covered all issues, that is)
- Turns out to be feasible with special design and compilation techniques

Enterprise VLAN, Virtually Private Network (VPN), Virtually Private Cloud (VPC)

- If the cloud vendor is able to "set aside" some servers, but cannot provide a private network, these tools let us create a form of VPN in which traffic for application shares the network with traffic for other platforms, but no leakage occurs
- In practice the approach is mostly via cryptography
- For this reason, "traffic analysis" could still reveal some data

Privacy with microservices

- Vendor or microservice developer will need to implement a similar "leave no trace" guarantee
- Use cryptography to ensure that data on the wire cannot be interpreted
  - With FPGA bump-in-the-wire model, this can be done at high speeds
  - So, we can pass data across the cloud message bus/queue safely as long as the message tag set does not reveal secrets
  - Cloud vendor could even audit the microservices, although this is hard to do and might not be certain to detect private data leakage

Databases with sensitive content

- Many applications turn out to need to create a **single** database with data from **multiple** clients, because some form of "aggregated" data is key to what the microservice is doing
  - Most customers who viewed product *A* want to compare with *B*
  - If you liked that book, you will probably like this one too
  - People like you who study at TUKE love Libresso
  - 88 % of people with this gene variant are descended from Genghis Khan

Issue with database queries

- Many people assume that we can anonymize databases, or limit users to queries that sum up ("aggregate") data over big groups
- But in fact, it is often surprisingly easy to de-anonymize the data, or use known information to "isolate" individuals
  - "How many bottles of wine are owned by people in Košice that have taught Cloud computing course?"
    - Seems to ask about a large population, but actually asks about me

Best possible? **Differential Privacy**

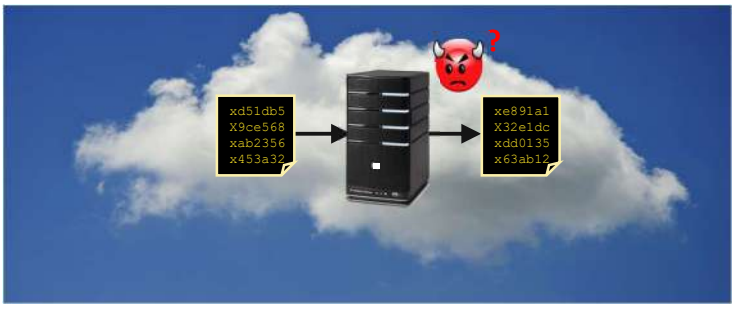
- Cynthia Dwork has invented a model called Differential Privacy
  - https://dl.acm.org/keyword/differential+privacy?expand=all&ContribRoleAndId=author%3A10.1145%2Fcontrib-81100183784
- We put our private database on a trusted server
  - It permits queries (normally, aggregation operations like average, min, max) but not retrieving individual data **and it injects noise into results**
- Noise level can be tuned to limit the rate at which leakage occurs

Bottles of wine query

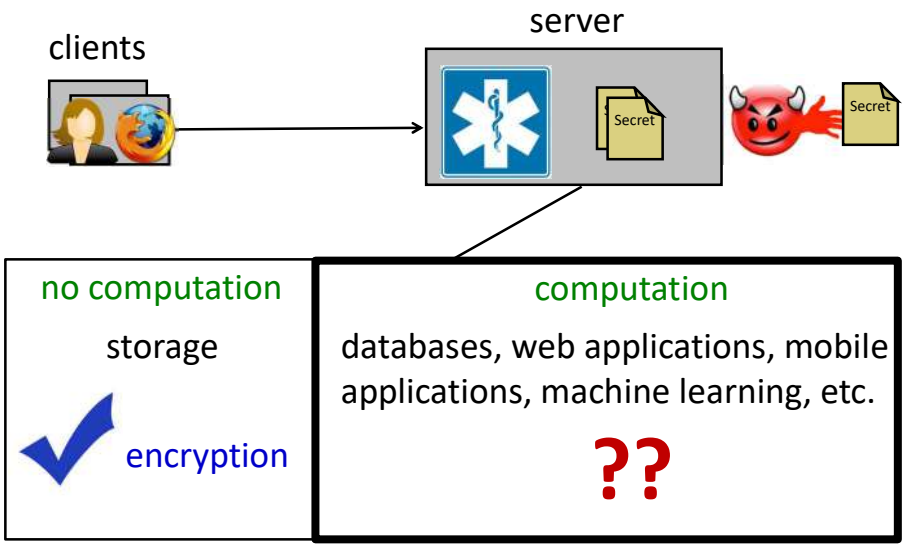
- For example, if the aggregation query includes a random extra number in the range [–10000, 10000], then an answer like 72 tells you nothing about my wine cellar
- There are several ways to add noise, and this is a "hot topic"
- But for many purposes, noisy results are not very useful
  - "I cannot see to the right. How many cars are coming?"

Building systems that compute on encrypted data

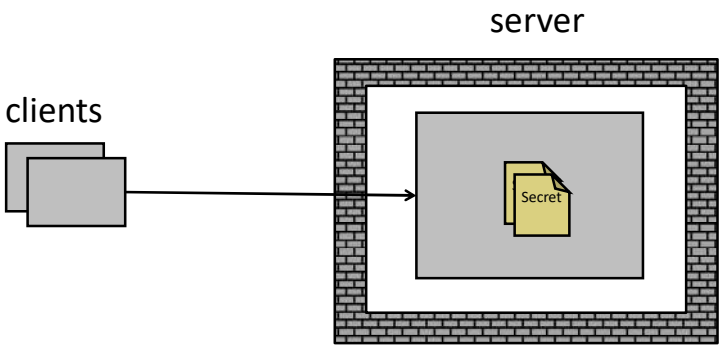
- Raluca Ada Popa
  - Her PhD work at MIT, now she is a professor at Berkeley
  - https://dl.acm.org/topic/ccs2012/10002978.10002979?expand=all&ContribRoleAndId=author%3A10.1145%2Fcontrib-99659462253



Problem setup



Current system strategy



Prevent attackers from breaking into servers

Lots of existing works

- Checks at the operating-system level
- Language-based enforcement of a security policy
- Static or dynamic analysis of application code
- Checks at the network level
- Trusted hardware
- …

- **Data still leaks even with these mechanisms because attackers eventually breaks in!**

Attacker:

hackers 

cloud employees 

increasingly many companies store data on external clouds

government 

accessed private data according to 

Reason they succeed:

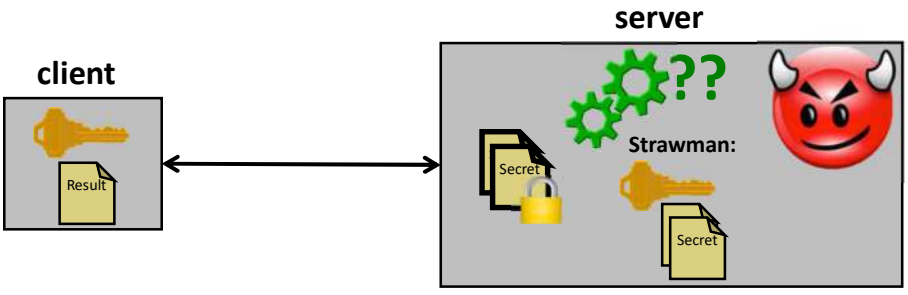
software is complex

insiders: legitimate server access!

e.g., physical access

Systems that protect confidentiality even against attackers with access to all server data

Servers store, process, and compute on encrypted data



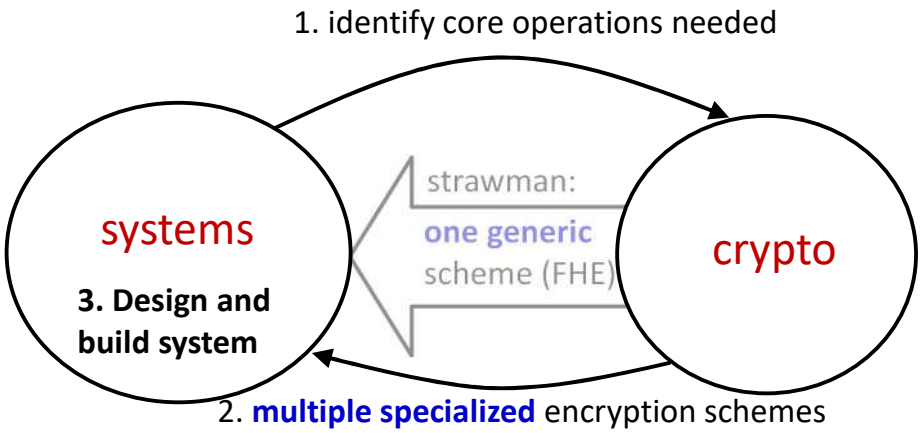
Approaches

- Computing on encrypted data in cryptography
  - [Rivest-Adleman-Dertouzos 1978]
- Fully homomorphic encryption (FHE)
  - [Gentry 2009]
- **Both are prohibitively slow, e.g., slowdown billion times**

- **"Practical systems"** by Raluca Ada Popa
  - Real-world performance
  - Large class of real applications
  - Menanigfull security

Raluca's contributions

- Theoretical
  - Functional cryptography [STOC'13], [CRYPTO'13]
- Practical systems
  - Databases
    - CryptDB [SOSP'11], [CACM'12]
    - mOPE, adjJOIN [Oakland'13]
  - Web applications
    - Mylar (multi-key search) [NSDI'14]
  - Mobile applications
    - PrivStats [CCS'11]
    - VPriv [Usenix Security'09]



New schemes:

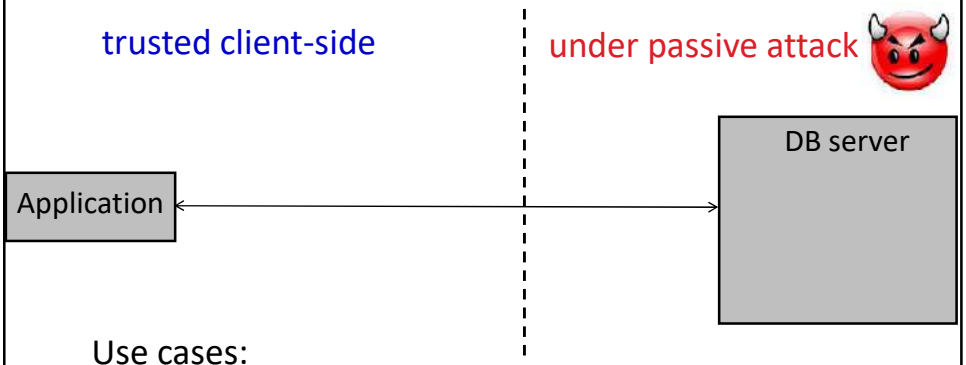
- mOPE, adjJOIN for CryptDB
- multi-key search for Mylar

CryptDB

- First **practical** database system (DBMS) to process most SQL queries on encrypted data

- Related works
  - Systems work [Hacigumus et al.'02], [Damiani et al.'03], [Ciriani et al'09]
    - No formal confidentiality guarantees
    - Restricted functionality
    - Client-side filtering
  - Theory work
    - General computation: FHE [Gentry'09]
      - Very strong security: **forces slowdown - many queries must always scan and return the whole DB**
      - Prohibitively slow (billion times)
    - Specialized schemes [Amanatidis et al.'07], [Song et al.'00], [Boldyreva et al.'09]

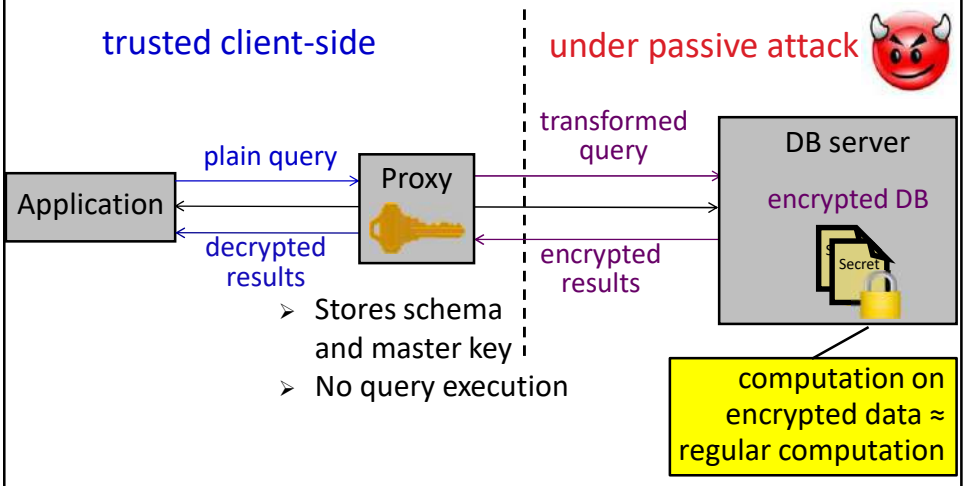
Setup



Use cases:

- Outsource DB to the cloud (DBaaS)
  - e.g. Encrypted BigQuery
- Local cluster: hide DB content from sys. admins.

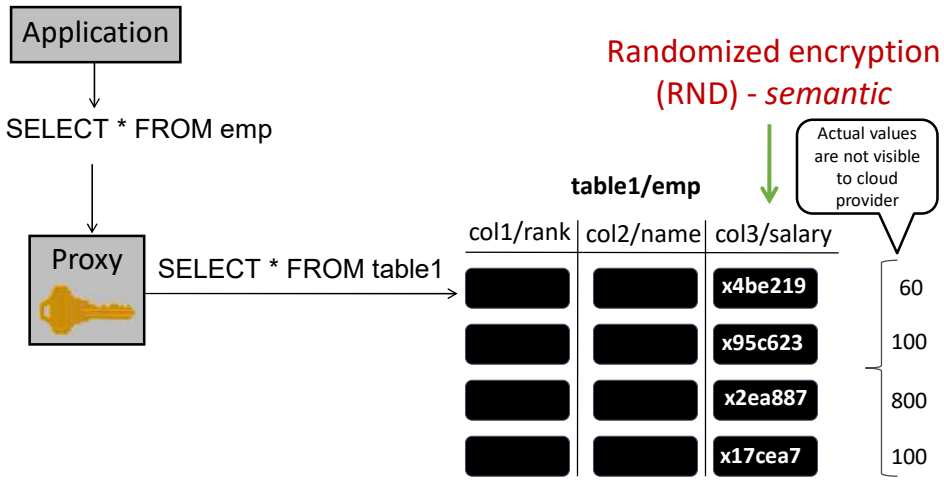
Setup



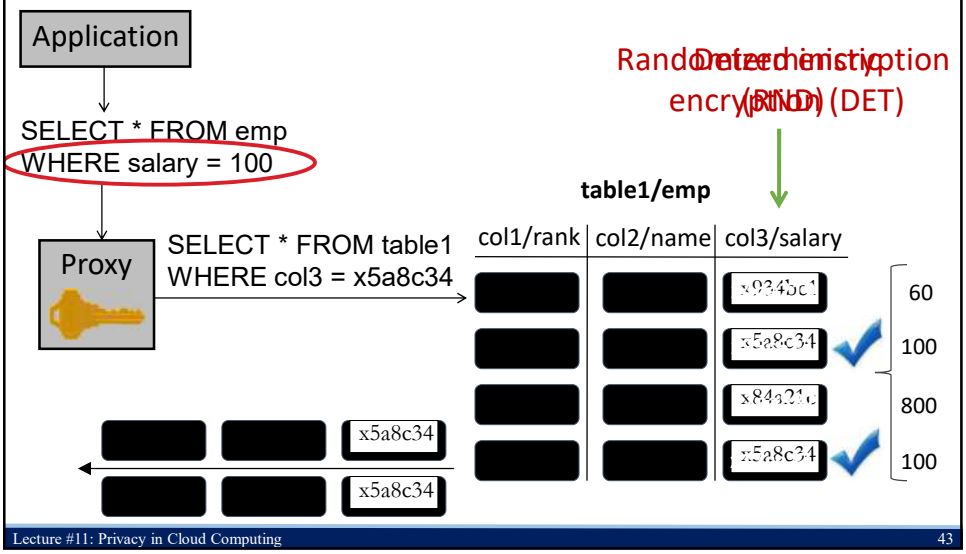
Goal

- We want to support standard queries, such as "retrieve the whole database", or "get the average salary for tier 3 employees"
- Yet we do not want the service operator to ever see the real data
  - So the step that selects matching data cannot decrypt

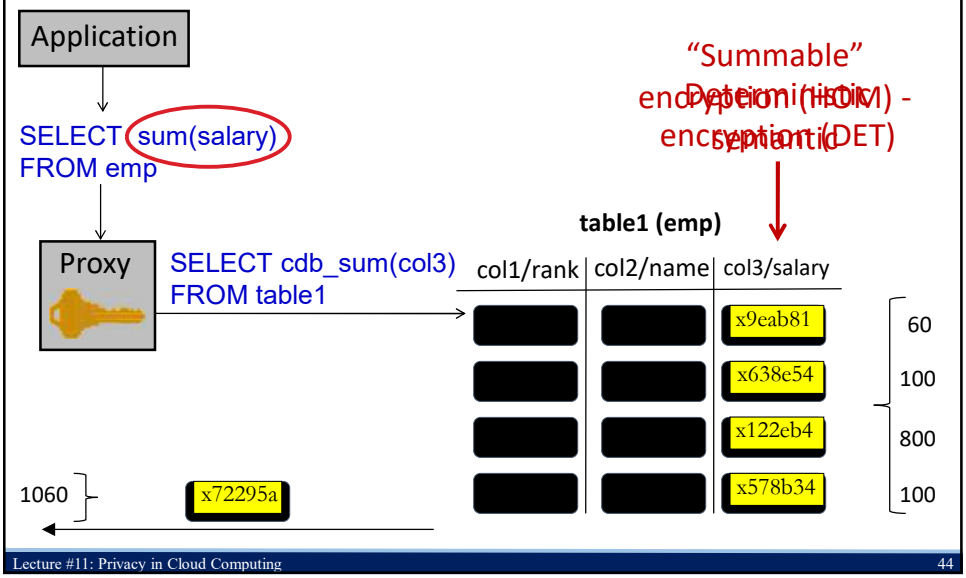
Example



Example



Example



Techniques

1. Use SQL-aware set of efficient encryption schemes (**meta technique**)
   - Most SQL can be implemented with a few core operations
2. Adjust encryption of data based on queries
3. Query rewriting algorithm

Security	Scheme	Construction	Function	SQL operations:
≈ semantic security	RND	AES in UFE	data moving	e.g., SELECT, UPDATE, DELETE, INSERT, COUNT
	HOM	Paillier	addition	
	SEARCH	Song et al.,'00	word search	
reveals only repeat pattern	DET	AES in CMC	equality	e.g., =, !=, IN, GROUP BY, DISTINCT
	JOIN	our new scheme	join	
reveals only order	OPE	our new scheme [Oakland'13]	order	e.g., >, <, ORDER BY, ASC, DESC, MAX, MIN, GREATEST, LEAST

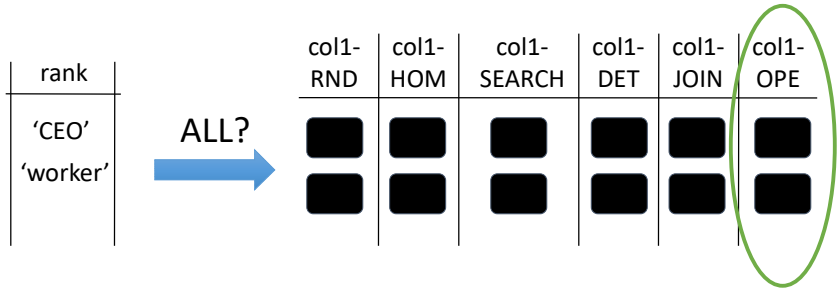
$x < y \iff Enc(x) < Enc(y)$

How to encrypt each data item?

Goals: 1. Support queries

2. Use most secure encryption schemes

Challenge: may not know queries ahead of time

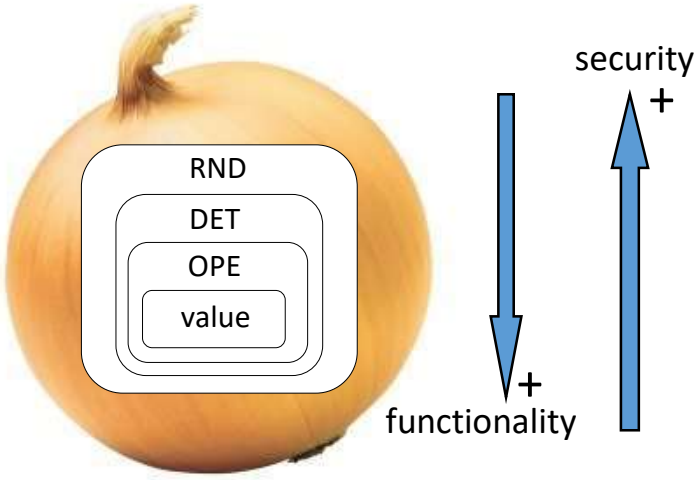


Leaks order!

Onion concept

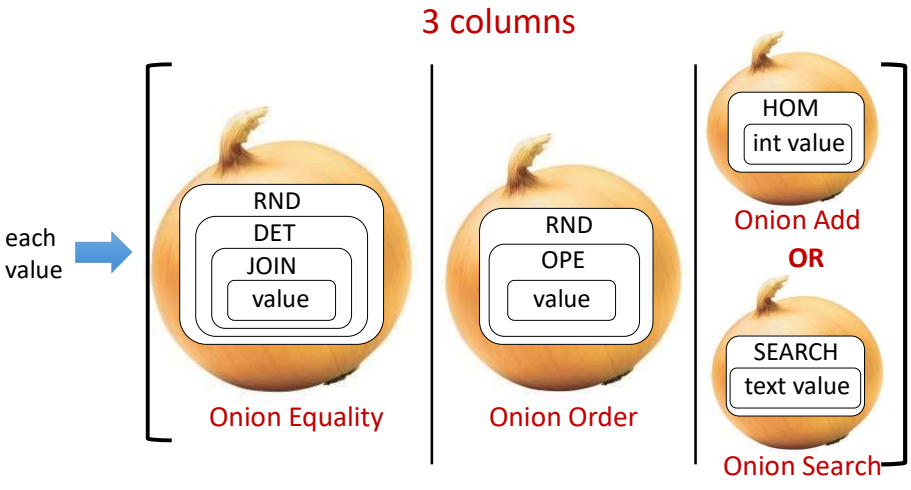
- Take some value, X
- Now encrypt it once: X_K
- What if we encrypt X_K a second time, with a different crypto system?
  - We now have a two-layer "onion"
  - We would have to decrypt twice to reveal X

Onion of encryptions



Adjust encryption: strip off layer of the onion

Onion of encryptions

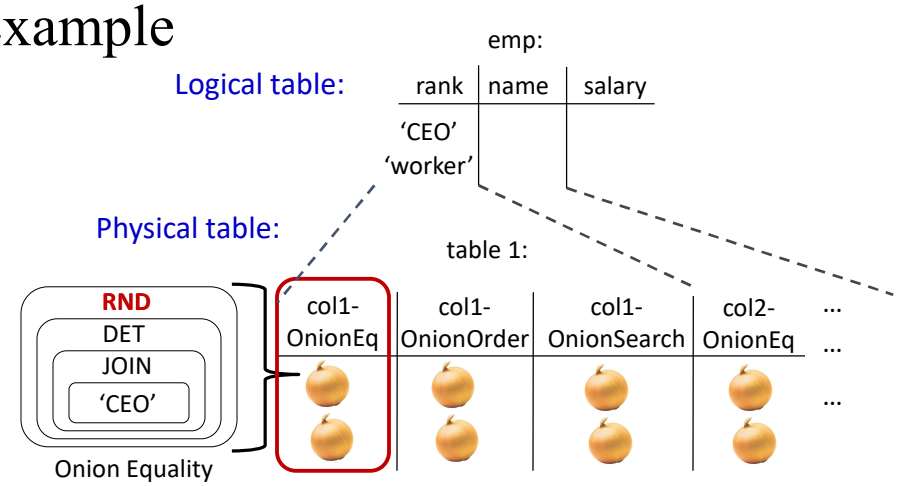


Same key for all items in a column for same onion layer

Onion evolution

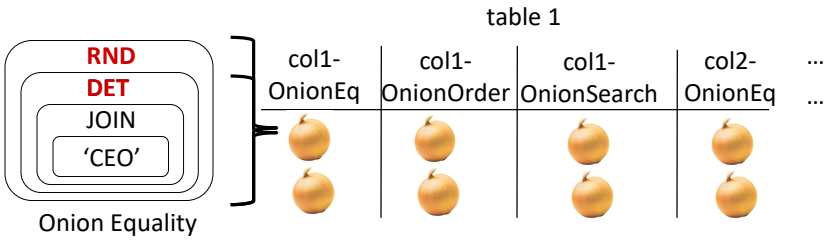
- Start out the database with the most secure encryption scheme
- If needed, adjust onion level
  - Proxy gives decryption key to server
  - Proxy remembers onion layer for columns
- **Lowest onion level is never removed**

Example



SELECT * FROM emp WHERE rank = 'CEO'

Example



SELECT * FROM emp WHERE rank = 'CEO'

⇩

UPDATE table1 SET col1-OnionEq =
Decrypt_RND(key, col1-OnionEq)

SELECT * FROM table1 WHERE col1-OnionEq = xda5c0407

Security treshold

- Data owner can specify minimum level of security

CREATE TABLE emp (…, credit_card SENSITIVE integer, …)

RND, HOM, DET for unique fields
≈ semantic security

Security guarantee

Columns annotated as sensitive have semantic security (or similar).

Encryption schemes exposed for each column are the most secure enabling queries.

equality ➡ repeats sum ➡ semantic no filter ➡ semantic

common in practice

➡ Never reveals plaintext

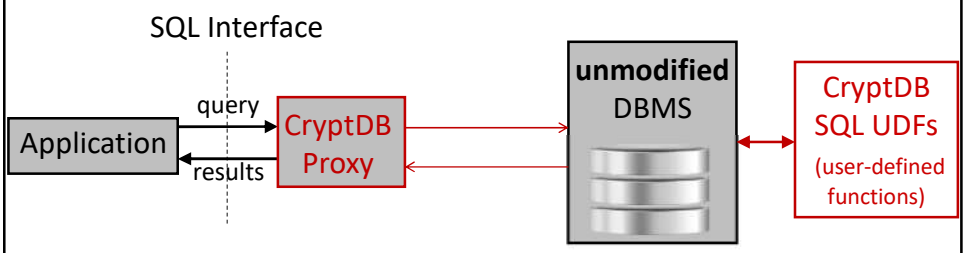
Limitations and workarounds

- Queries not supported
  - More complex operators, e.g., trigonometry
  - Certain combinations of encryption schemes:
    - e.g., salary + raise > 100K

HOM

➡ use query splitting, query rewriting

Implementation



No change to the DBMS!

Largely no change to apps!

Evaluation

- Does it support real queries/applications?
- What is the resulting confidentiality level?
- What is the performance overhead?

Real queries/application

Application	Encrypted columns	# cols with queries not supported
phpBB	23	0
HotCRP	22	0
grad-apply	103	0
TPC-C	92	0
sql.mit.edu	128,840	1,094

apps with sensitive columns

tens of thousands of apps

SELECT 1/log(series_no+1.2) …

… WHERE sin(latitude + PI()) …

Confidentiality level

Final onion state

Application	Encrypted columns	Min level: ≈semantic	Min level: DET/JOIN	Min level: OPE
phpBB	23	21	1	1
HotCRP	22	18	1	2
grad-apply	103	95	6	2
TPC-C	92	65	19	8
sql.mit.edu	128,840	80,053	34,212	13,131

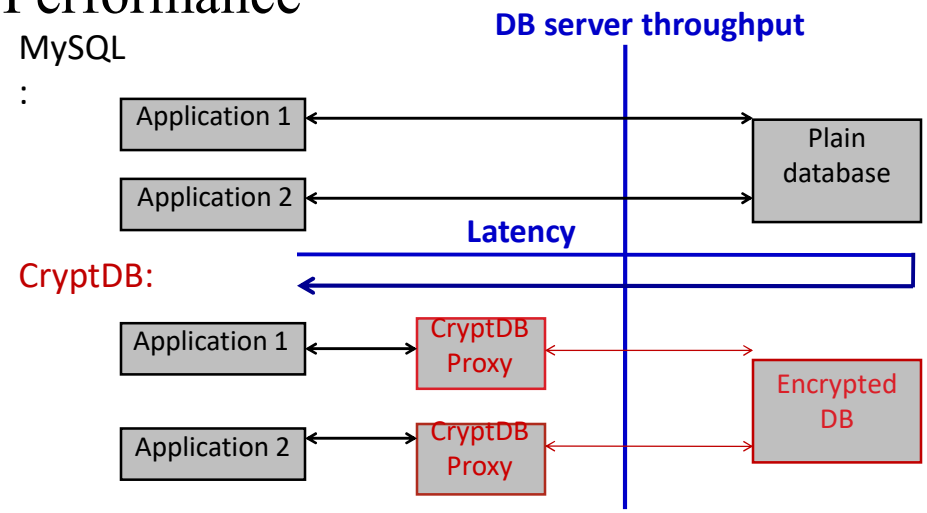
Most columns at semantic

Most columns at OPE were less sensitive

Performance

MySQL

:

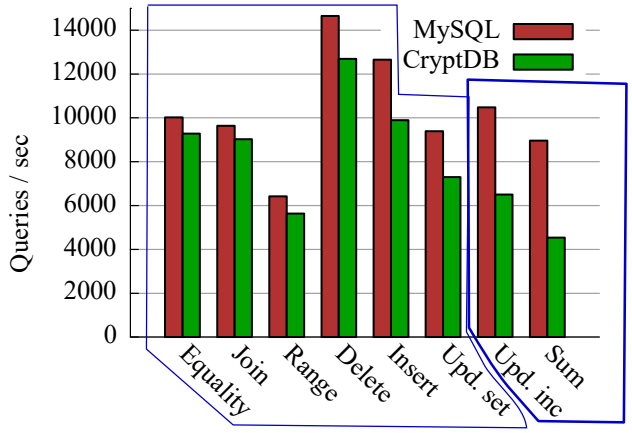


Hardware: 2.4 GHz Intel Xeon E5620 – 8 cores, 12 GB RAM

TPC-C performance

Latency (per query): 0.10ms MySQL vs. 0.72ms CryptDB

Throughput loss over MySQL: 26%



Homomorphic addition

No cryptography at the DB server in the steady state!

Adoption

Google Encrypted BigQuery [http://code.google.com/p/encrypted-bigquery-client/]

Úlfar Erlingsson, head of security research, Google

"CryptDB was really eye-opening in establishing the practicality of providing a SQL-like query interface to an encrypted database"

"CryptDB was [..] directly influential on the design and implementation of Encrypted BigQuery."

SAP SEEED implemented on top of the SAP HANA DBMS

MIT Lincoln Laboratory Encrypted version of the D4M Accumulo NoSQL engine

sql.mit.edu Users opted-in to run Wordpress over our CryptDB source code

Sources

- https://css.csail.mit.edu/cryptdb
- https://github.com/CryptDB

Concerns about CryptDB?

- The main criticisms stem from the "strip a layer" step
- Once we reduce the level of protection, we've leaked some information and the remaining data is "less protected"
  - Raluca's response: If you want to make use of operations like aggregation, you cannot easily avoid releasing some information
- Criticism response to Raluca: attacker might **trick** my code into doing the operation, and might do so in the future when some flaw in one of the crypto scheme is noticed, the logic would not protect itself in that case

Summary

- A "leave no trace" model could offer a practical way to leverage the cloud and yet not release private data to the public
- With a trusted vendor willing to audit operations and to "enclave" sensitive data computation, and clean up afterward, there is real hope for privacy without leaks
- SGX, costly but can be used where the vendor is not trusted
- For databases, techniques like CryptDB are not perfect but work well
  - Differential privacy is even better, but only if noise can be tolerated

CLOUD SYSTEMS

MapReduce

Lecture #12

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

Motivation

- Process lots of data
  - Google is processing about 24 petabytes of data per day
- A single machine cannot serve all the data
  - You need a distributed system to store and process in parallel
- Parallel programming?
  - Threading is hard!
  - How do you facilitate communication between nodes?
  - How do you scale to more machines?
  - How do you handle machine failures?

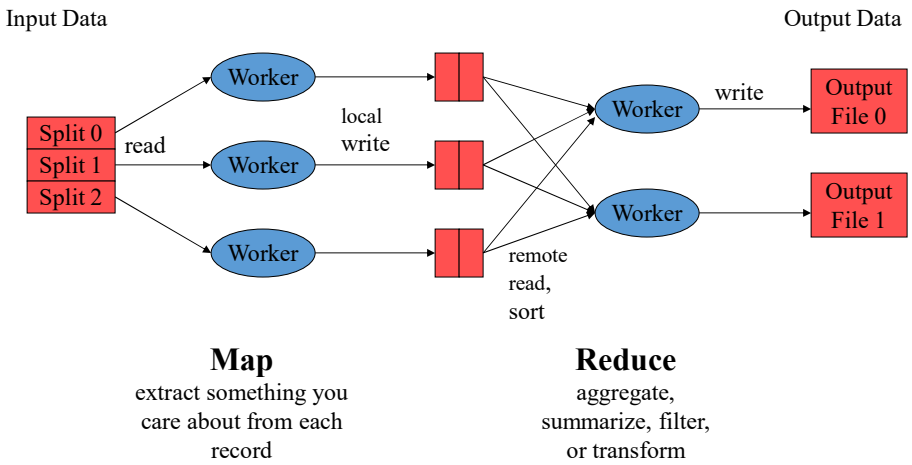
Usage of MapReduce

- MapReduce provides
  - Automatic parallelization, distribution
  - I/O scheduling
    - Load balancing
    - Network and data transfer optimization
  - Fault tolerance
    - Handling of machine failures
- Need more power: Scale out, not up!
  - Large number of commodity servers as opposed to some high-end specialized servers

Typical problem solved by MapReduce

- Read a lot of data
- **Map** – extract something you care about from each record
- Shuffle and Sort
- **Reduce** – aggregate, summarize, filter, or transform
- Write the results

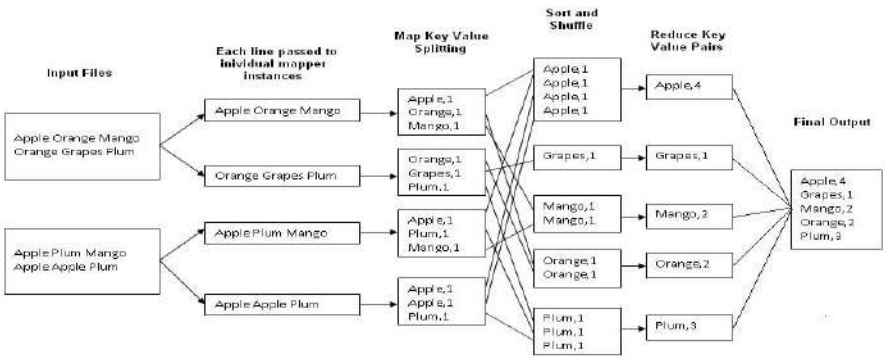
MapReduce workflow



Mappers and reducers

- Need to handle more data?
  - Just add more mappers and reducers
- No need to handle multithreaded code
  - Mappers and reducers are typically single threaded and deterministic
    - Determinism allows for restarting of failed jobs
  - Mappers and reducers run entirely independent of each other
    - In Hadoop, they run in separate JVMs

Example: Word count

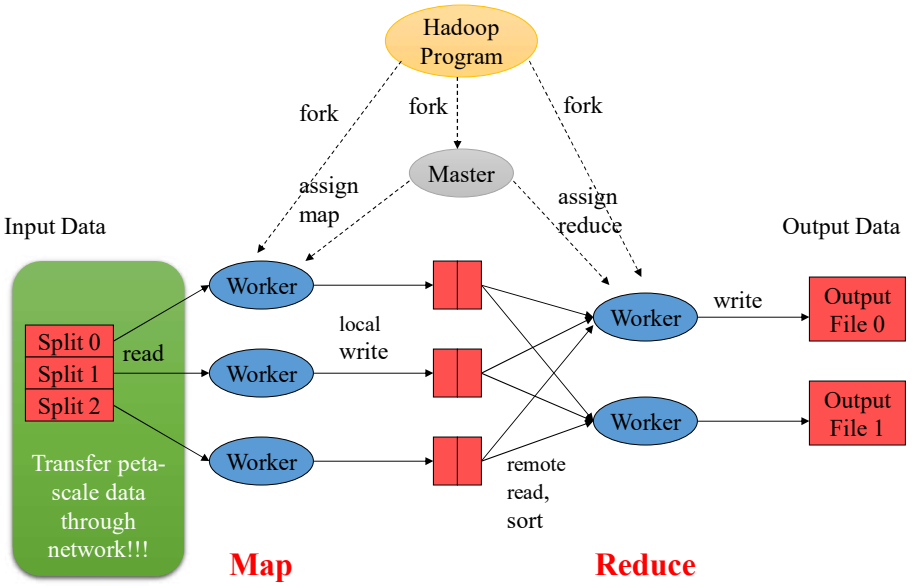


Mapper

- Reads in input pair <K, V>
- Outputs a pair <K', V'>
- Let's count number of each word in user queries (or Tweets, Blogs)
- The input to the mapper will be <QueryID, QueryText>
  - <Q1, "The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am.">
- The output would be
  - <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store,1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store,1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>

Reducer

- Accepts the Mapper output, and aggregates values on the key
- For our example, the reducer input would be
  - <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> **<store, 1>** <the, 1> **<store, 1>** <was, 1> <closed, 1> <the, 1> **<store, 1>** <opens,1> <in, 1> <the, 1> <morning, 1> <the 1> **<store, 1>** <opens, 1> <at, 1> <9am, 1>
- The output would be
  - <The, 6> <teacher, 1> <went, 1> <to, 1> **<store, 4>** <was, 1> <closed, 1> <opens, 2> <morning, 1> <at, 1> <9am, 1>

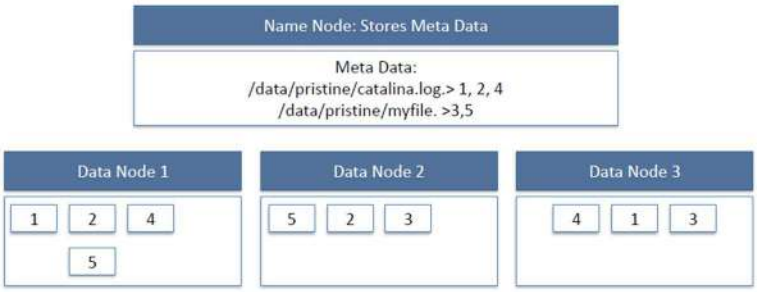


Distributed file systems

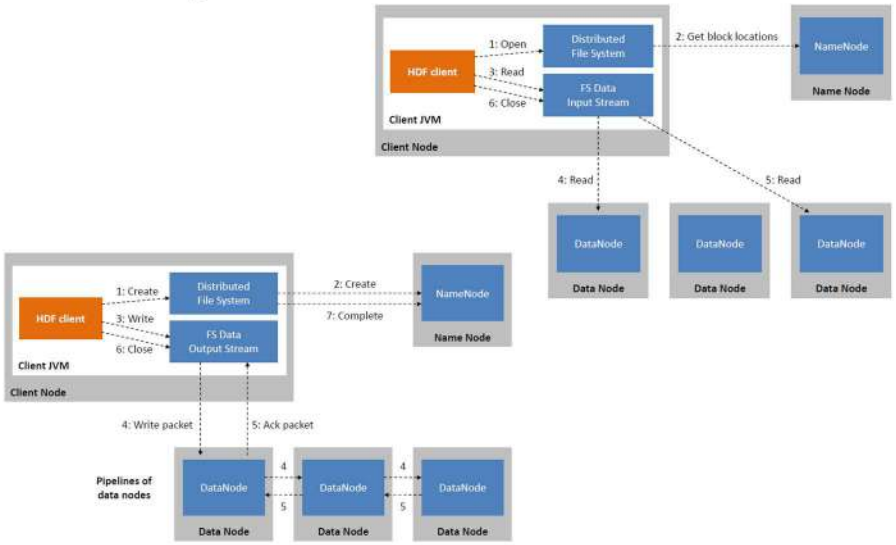
- Split data and store replicas on commodity servers
  - Google file system (GFS)
  - Hadoop distributed file system (HDFS)

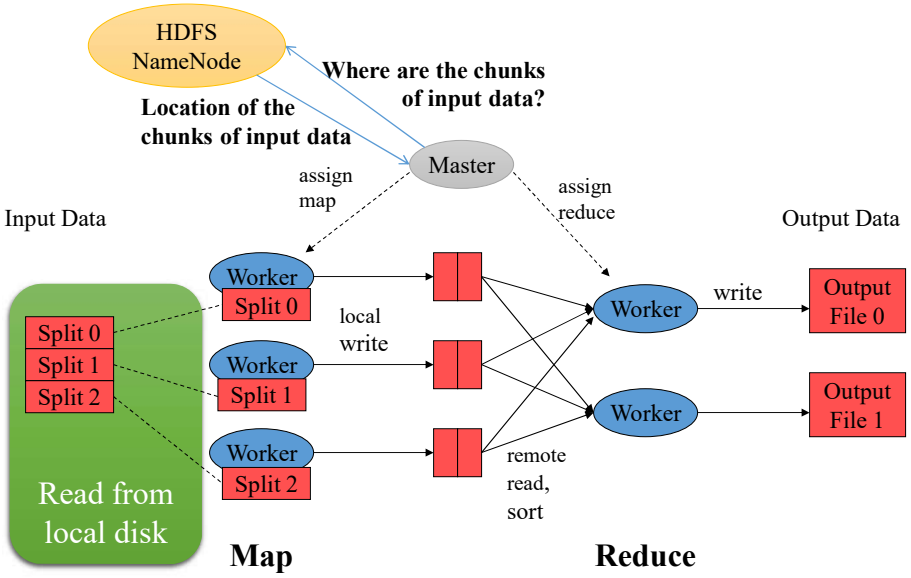
HDFS concepts

- Blocks – files are split to 128 MB blocks
- Data nodes – store and retrieve blocks
- Name node – metadata information of files



HDFS opertations

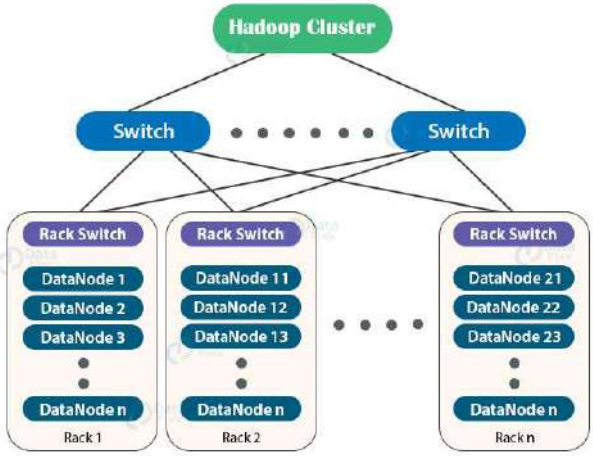




Locality optimization

- Master scheduling policy
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks scheduled so GFS input block replica are on same machine or same rack
- Effect
  - Thousands of machines **read input at local disk speed**
  - Eliminate network bottleneck

HDFS rack awareness



Why rack awareness?

- To reduce the network traffic while file read/write, which improves the cluster performance
- To achieve fault tolerance, even when the rack goes down
- Achieve high availability of data so that data is available even in unfavorable conditions
- To reduce the latency, that is, to make the file read/write operations done with lower delay

- NameNode uses a rack awareness algorithm while placing the replicas in HDFS

Fault tolerance

- Failures are norm in commodity hardware
- Worker failure
  - Detect failure via periodic heartbeats
  - **Re-execute** in-progress map and reduce tasks
- Master failure
  - Single point of failure
  - Resume from Execution Log
- Robust
  - Google's experience: lost 1600 of 1800 machines once, but finished fine

Refinement: Redundant execution

- Slow workers significantly lengthen completion time
  - Other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
  - Weird things: processor caches disabled
- Solution
  - Spawn backup copies of tasks
  - Whichever one finishes first "wins"

Refinement: Skipping bad records

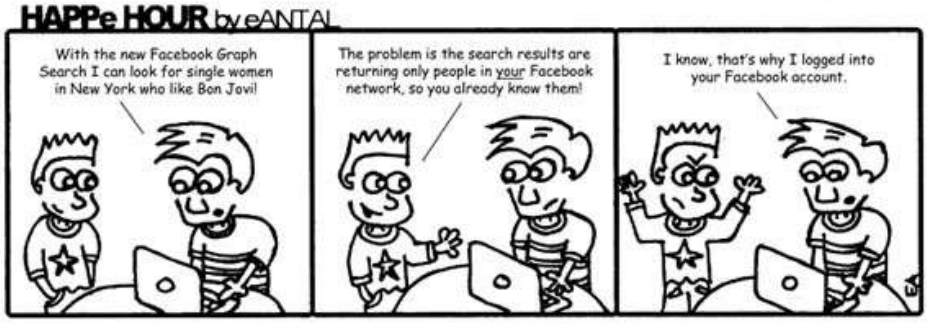
- Map and reduce functions sometimes fail for particular inputs
  - Best solution is to debug and fix, but not always possible
  - If master sees two failures for the same record next worker is told to skip the record


```
public class WordCount {
  public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
  }}}
  public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
  }}

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);
        P
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
  }}
```

Run this program as a MapReduce job

Summary

- Programming paradigm for data-intensive computing
- Distributed and parallel execution model
- Simple to program
  - The framework automates many tedious tasks (machine selection, failure handling, etc.)



CLOUD SYSTEMS

Social Networks Graphs

Lecture #13

doc. Ing. Martin Tomášek, PhD.
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice

2025/2026

Another view on big data

- We have looked at computing on sharded big data, but not all big data is actually stored in this form
- Facebook creates and manages big data TAO database for the social networking graph
  - This is a central form of big data in Facebook, and most cloud platforms have a similar system
- IoT will cause us to create new kinds of social network graphs (for example to track Covid spread)
  - This is a huge opportunity area for researchers

What is big data usually about?

- Early in the cloud era, research at companies like Google and Amazon made it clear that people respond well to social networking tools and smarter advertising placement and recommendations
- The idea is simple
  - "People with Martin's interest find this store fantastic."
  - "Jane really likes Chanel and might want to know about this 15 % off sale on spring clothing."
  - "Peter had a flat tire and needs new ones."

They had a lot of customers and data

- Web search and product search tools needed to deal with billions of web pages and hundreds of millions of products
- Billions of people use these modern platforms
- So simple ideas still involve enormous data objects that simply cannot fit in memory on modern machines
  - And yet in-memory computing is far faster than any form of disk-based storage and computing

What are the big data files?

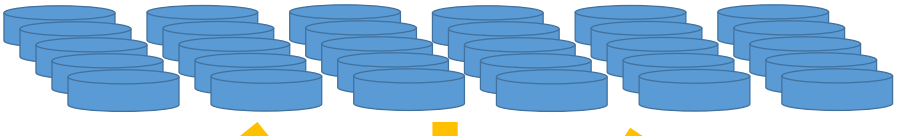
- Snapshot of all the web pages in the world, updated daily
- Current product data and price for every product Amazon knows about
- Social networking graph for all of Facebook

Many challenges

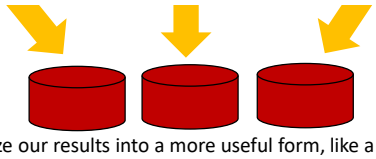
XenonStack.com

Visualizing the pipeline

Data starts out sharded over servers



Early pipeline stages are extremely parallel: they *extract, transform, summarize*



Eventually we squeeze our results into a more useful form, like a trained machine-learning model.
The first stages can run for a long time before this converges

Copy the model to wherever we plan to use it.

For web pages this is "easy"

- The early steps mostly extract words or phrases, and summarize by doing things like counting or making lists of URLs
- The computational stages do work similar to sorting (but at a very large scale), e.g. finding the "most authoritative pages" by organizing web pages in a graph and then finding the graph nodes with highest weight for a given search
- When we create a trained machine-learning model, the output is some sort of numerical data that parameterizes a "formula" for later use (like to select ads)

What about for social networks?

- Here we tend to be dealing with very large and very dynamic graphs
- The approaches used involve specialized solutions that can cope with the resulting dynamic updates
- Facebook's TAO is a great example

- Bronson, N. – Amsden, Z. – Cabrera, G. – Chakka, P. – Dimov, P. – Ding, H. – Ferris, J. – Giardullo, A. – Kulkarni, S. – Li, H. – Marchukov, M. – Petrov, D. – Puzar, L. – Song, Y. J. – Venkataramani, V.: ***TAO: Facebook's Distributed Data Store for the Social Graph***. 2013 USENIX Annual Technical Conference (USENIX ATC '13), 2013
  - https://www.usenix.org/system/files/conference/atc13/atc13-bronson.pdf

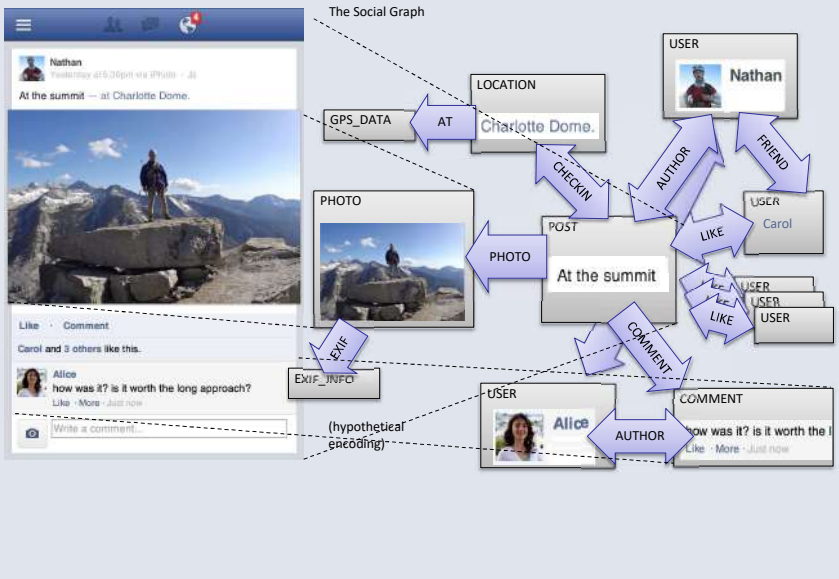
facebook

TAO

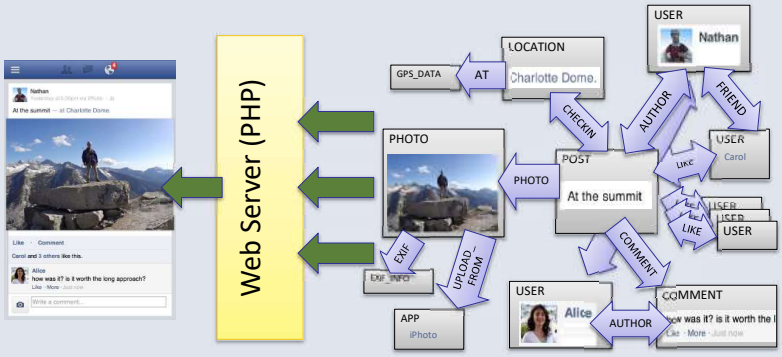
Facebook's Distributed Data Store for the Social Graph

Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, Venkat Venkataramani

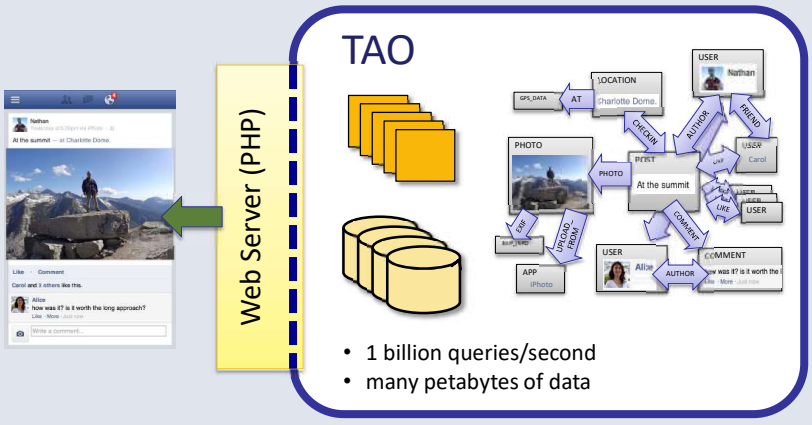
Presented at USENIX ATC – June 26, 2013



Dynamically Rendering the Graph



Dynamically Rendering the Graph



TAO opts for NoSQL Model

- Most TAO applications treat the graph like a very restricted form of SQL database: it looks like SQL.
- But first, they limit the operations: it isn't full SQL.
- And then they don't guarantee the ACID properties.

- In fact the *back end* of TAO actually is serializable, but it runs out of band, in a batched and high-volume way (BASE: eventually, consistency happens).
- The only edge consistency promise is that they try to avoid returning broken association lists, because applications find such situations hard to handle.

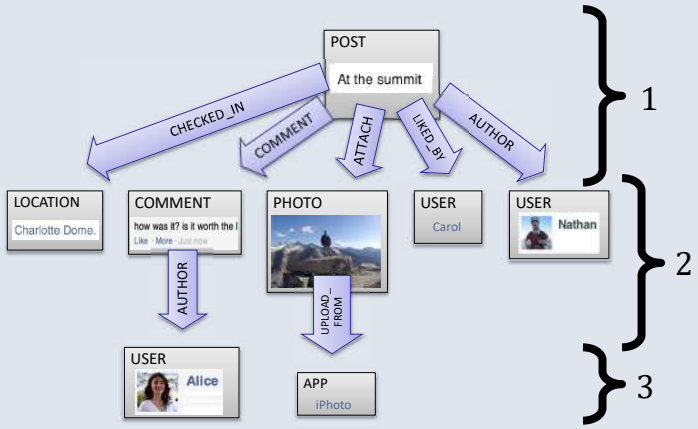
Concept: A *data dependency*

- For TAO, if a node has an edge to some other node, we say that there is a data dependency: the origin of that directed edge "depends" on the destination

- They think of the graph as "broken" if a dependent node can't find the node it depends upon.
  - It would show up as a "?" on the Facebook feed, and they don't like that
  - Ideally, you only see a "?" if they deleted some sort of spam, fake news, etc

- The graph has a depth and so we can talk about dependencies at depth *d*

Dynamic Resolution of Data Dependencies



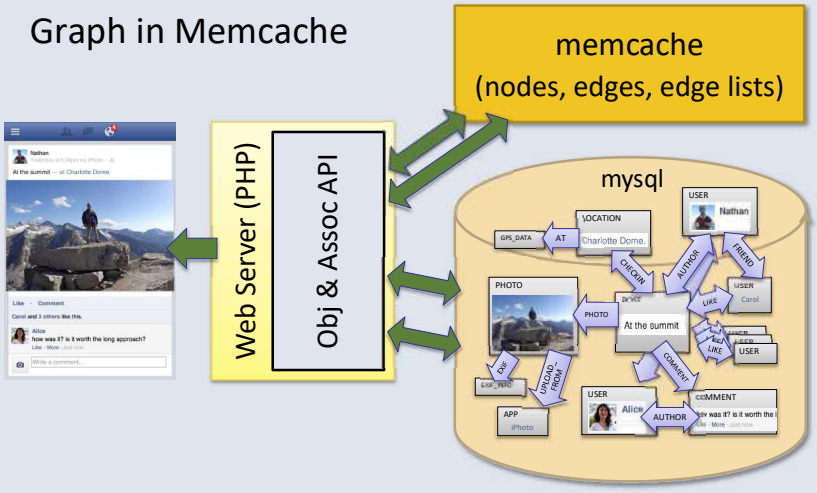
What Are TAO's Goals/Challenges?

- Efficiency at scale

What Are TAO's Goals/Challenges?

- Efficiency at scale
- Low read latency
- Timeliness of writes
- High Read Availability

Graph in Memcache



Objects = Nodes

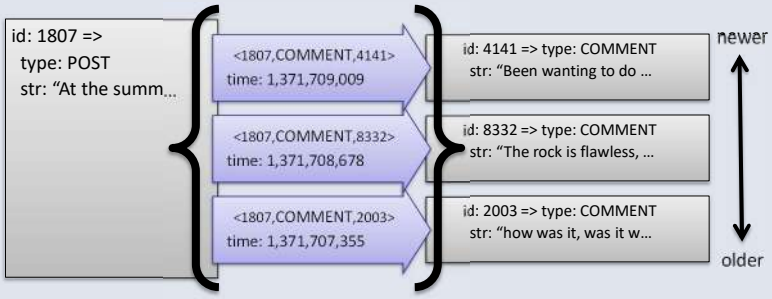
- Identified by unique 64-bit IDs
- Typed, with a schema for fields

Associations = Edges

- Identified by <id1, type, id2>
- Bidirectional associations are two edges, same or different type

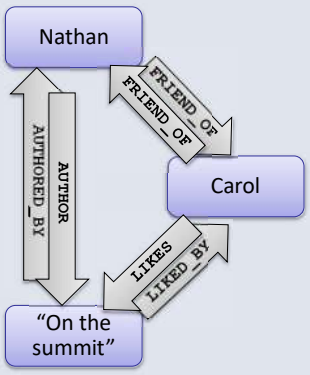
Association Lists

- <id1, type, *>
- Descending order by time
- Query sublist by position or time
- Query size of entire list



Inverse associations

- Bidirectional relationships have separate *a*→*b* and *b*→*a* edges
  - inv_type(**`LIKES`**) = **`LIKED_BY`**
  - inv_type(**`FRIEND_OF`**) = **`FRIEND_OF`**
- Forward and inverse types linked only during write
  - TAO **`assoc_add`** will update both
  - Not atomic, but failures are logged and repaired



Objects and Associations API

Reads – 99.8%

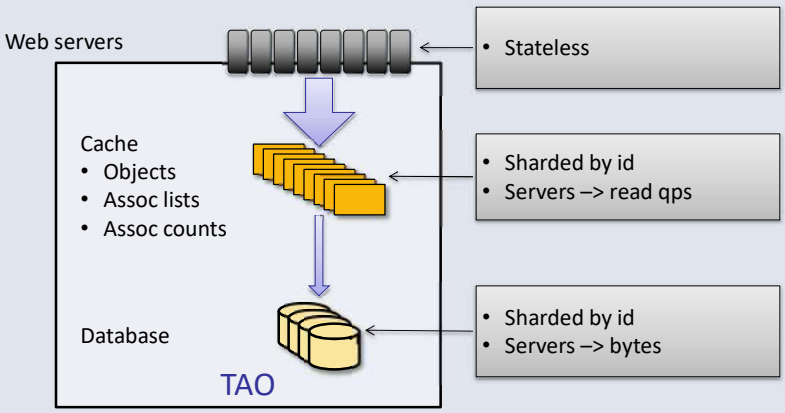
- Point queries
  - **`obj_get`** 28.9%
  - **`assoc_get`** 15.7%
- Range queries
  - **`assoc_range`** 40.9%
  - **`assoc_time_range`** 2.8%
- Count queries
  - **`assoc_count`** 11.7%

Writes – 0.2%

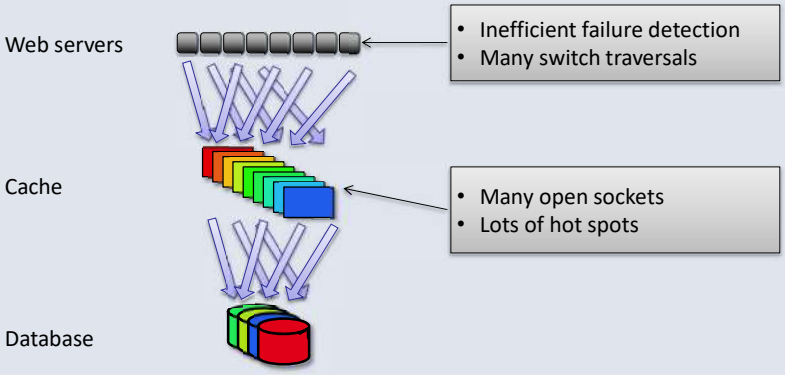
- Create, update, delete for objects
  - **`obj_add`** 16.5%
  - **`obj_update`** 20.7%
  - **`obj_del`** 2.0%
- Set and delete for associations
  - **`assoc_add`** 52.5%
  - **`assoc_del`** 8.3%

What Are TAO's Goals/Challenges?

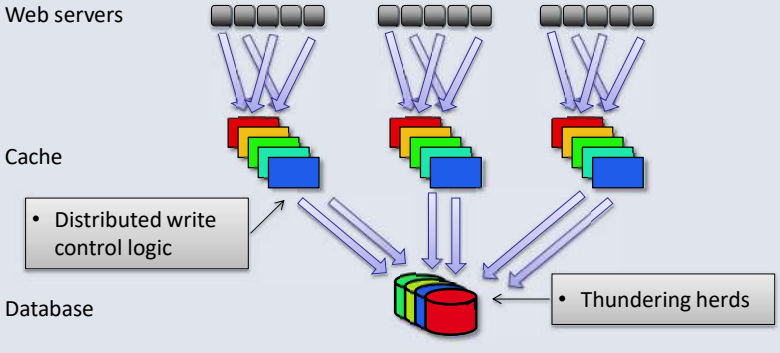
Independent Scaling by Separating Roles



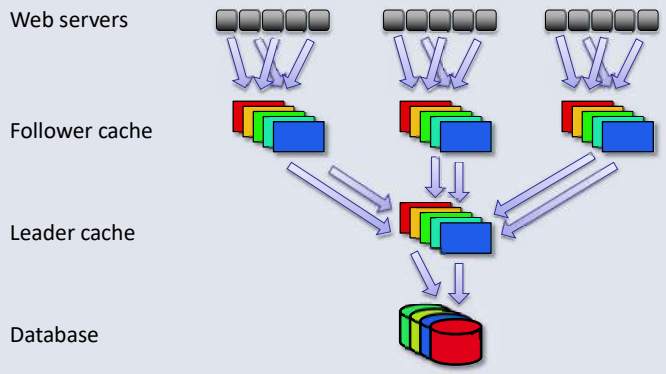
Subdividing the Data Center



Subdividing the Data Center



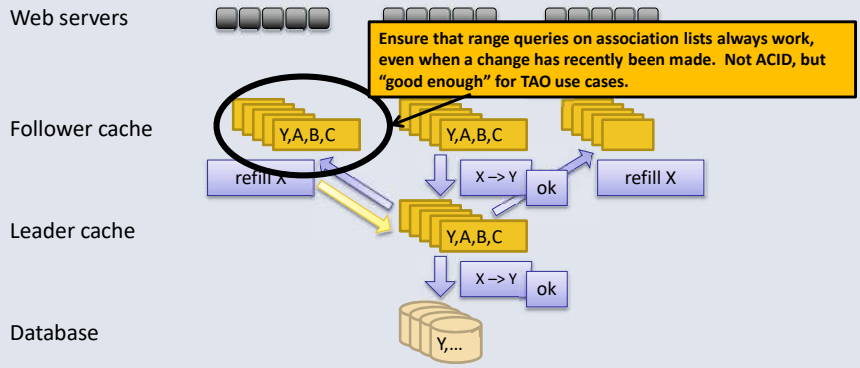
Follower and Leader Caches



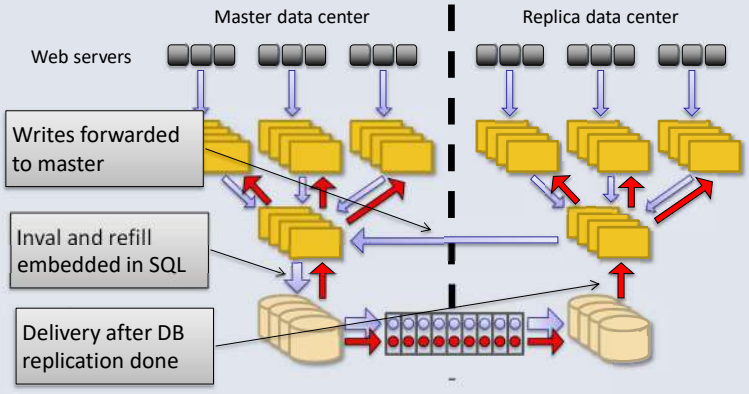
What Are TAO's Goals/Challenges?

- Efficiency at scale
- Low read latency
- Timeliness of writes
- High Read Availability

Write-through Caching – Association Lists



Asynchronous DB Replication



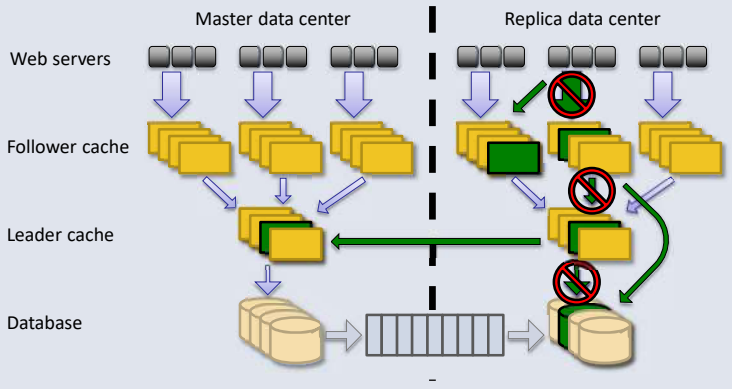
What Are TAO's Goals/Challenges?

- Efficiency at scale
- Low read latency
- Timeliness of writes
- High Read Availability

Key Ideas

- TAO has a "normal operations" pathway that offers pretty good properties, very similar to full ACID.
- But they also have backup pathways for almost everything, to try to preserve updates (like unfollow, or unfriend, or friend, or like) even if connectivity to some portion of the system is disrupted.
- This gives a kind of self-repairing form of fault tolerance. It doesn't promise a clean ACID model, yet is pretty close to that.

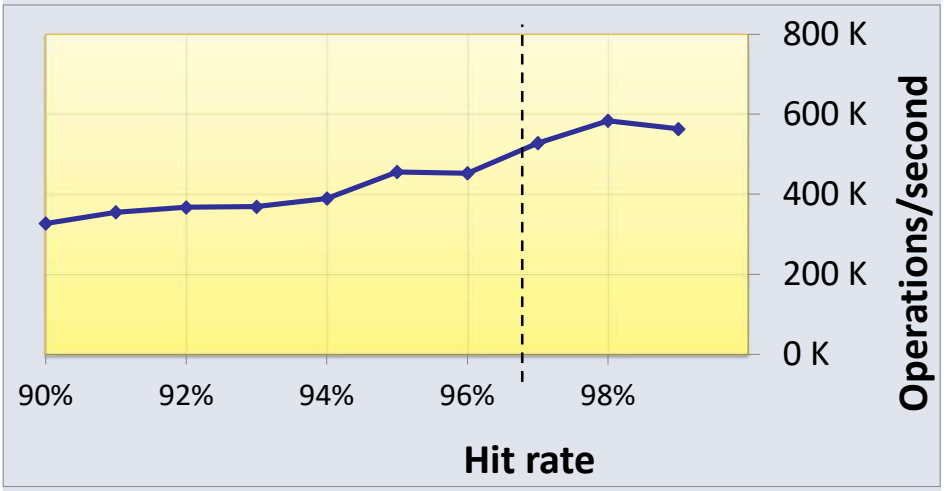
Improving Availability: Read Failover



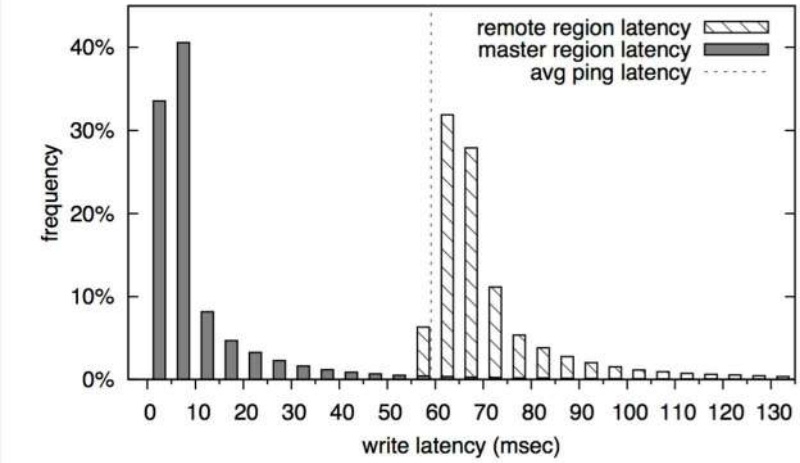
TAO Summary

Efficiency at scale Read latency	• Separate cache and DB • Graph-specific caching • Subdivide data centers
Write timeliness	• Write-through cache • Asynchronous replication
Read availability	• Alternate data sources

Single-server Cache Node Peak Observed Capacity



Write latency



Not on this slide set

- *Much* more data about performance under various conditions
- The role of association time in optimizing cache hit rates
- Optimized graph-specific data structures
- Write failover
- Failure recovery
- Workload characterization

As we look beyond Hadoop and TAO

- TAO showed us that the Hadoop style of computing on sharded data is not really enough
- We also need a data ingress infrastructure optimized for the expected workload and customized for the specifics of the target setting
- Because Facebook has such a social-network centrism, it makes sense that the company would innovate on social network graph infrastructure tools

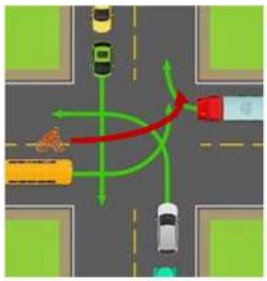
The edge and "real world" networks

- Edge computing will demand new architectures and tools, as well
- Up to now we have treated edge computing as if it would be mostly a job for standard big-data analytics
- But in fact edge applications have a "social structure" of their own, emerging from the roles the various elements are playing

Think of smart cars in a modern city

- Each has its own on-board computing infrastructure and knowledge of its current task, goals, plans
  - Even true of non-self-driving cars
- Around it are smart urban support solutions, like smart traffic intersections and people carrying smart devices
- The next generation of applications will need to put those together to create high value for consumers (and high value companies to provide those solutions)

Favorite examples



A smart intersection or roadway must send warnings within 25-30ms!

Are there "graphs" here?

- No, if we mean specifically Facebook TAO social networking graphs
- But the information of relevance to a vehicle or a pedestrian is a kind of graph in which the nodes are other entities, and the edges represent interactions, and the queries we might use have to run on that graph
- We want to warn "Watch out for that bicycle!" but do that we have to query vehicle paths and proximity and know which ones are impacted

TAO could inspire those next solutions

- Like TAO they need to be geo-scalable, timely, graph-oriented
- One big difference is that we can rapidly discard the past: these smart solutions often care mostly about current state, although they may have learned "characteristics" from past observations (like, "Susan is an insane motorcycle driver," but "Mark the Uber driver is slow and careful")
- We could start with a TAO-like model and build from there

What will motivate the developers?

- Facebook is motivated to serve their customers amazingly well but also cost-effectively
- They also earn revenue by placing advertisements in smart ways that lead to click-throughs and purchases
- Edge and 5G cellular solutions will align with revenue opportunities

Conclusions

- The big data world will be evolving rapidly under demand from IoT uses
- We should look for existing solutions and ask how much of the infrastructure can be reused for these edge-computing cases
  - For example, hybrid cloud can increase our confidence that a microservice model is genuinely viable
- But we should be "cautious" and not change lots of things all at once
  - The cloud is surprisingly delicate and not everything would scale, be easy to manage, be cost-effective, be performant, etc.
- Follow the money to understand which aspects will mature most quickly