ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract Cloud Computing Architecture

Roman Böhringer

June 1, 2021

Department of Computer Science, ETH Zürich

Contents

Cloud computing is the delivery of compute and storage resources on-demand, to offer high performance, cost efficiency (economies of scale), flexibility, and high availability. We distinguish between private clouds (limited to a single organization) and public clouds that can be used by multiple organizations. There are three phases in the evolution of cloud computing:

1. Cloud 1.0: Virtualization (enabled multiple users to share hardware)
2. Cloud 2.0: Hardware on demand (rent VMs by the hour/second from public cloud providers)
3. Cloud 3.0: "Serverless computing" (an abstraction of compute/storage services, not servers)

A datacenter is a cloud computing facility. Software, hardware, and the building infrastructure are co-designed to optimize performance, power, and the economies of scale. Trends in cloud computing are the exponential growth (of users, data, compute), the evolution of the hardware landscape (networking, storage, accelerators) and software frameworks (machine learning, serverless computing, resource management). A common theme in cloud design is the tradeoff between performance, cost, availability, and security.

Chapter 1

Datacenter Infrastructure

There are a lot of commodity components, high-bandwidth networking gear, and distributed computing software in datacenters. Furthermore, there is power (voltage converters/regulators, generator, UPSs, etc...) and cooling infrastructure (A/C, cooling towers, heat exchangers, etc...). Storage is either directly attached to compute nodes or disaggregated in storage racks. Racks often have 48 rack units (where 1RU is 4.45cm) and include cabling for power distribution/networking. The servers are often equipped with 2 sockets (to amortize the cost of other server components and increase the memory capacity). One needs to decide between chips with high-end or low-end cores. Low-end cores provide lower performance (e.g., 2x) at much lower power (e.g., 5x). But because of Amdahl's law, performance is limited by the serial part of a program and more nodes increase performance variability (tail latency). In datacenters, hardware will fail, so fault tolerance needs to be considered, even when using highly reliable hardware.

10-15 years ago, accelerators were not common in datacenters because their application domain was too narrow. Nowadays, with Moore's Law ending and the increasing popularity of ML applications, accelerators for ML can power a broad range of ML applications and there is a shift towards specialized accelators. They can improve performance per watt up to 100-times and the complexity is usually hidden behind online APIs. But we need to decide what type of accelerator to use (e.g., FPGA or ASIC) and where to place them.

The bisection bandwidth is the available bandwidth between any two equal network halves. Scaling it is difficult because the design of switches is power and pin limited. Therefore, switches are cascaded in hierarchical topologies and links higher in the topology are over-subscribed.

In datacenters, remote memory is often faster than local disk and the bandwidth for remote reads is limited by the network bandwidth. Some important numbers are:

Operation	**Time**
L1 cache reference	1.5 ns
L2 cache reference	5 ns
Branch misprediction	6 ns
Uncontended mutex lock/unlock	20 ns
L3 cache reference	25 ns
Main memory reference	100 ns
Decompress 1KB with Snappy	500 ns
"Far memory"/Fast NVM reference	1,000 ns (1 us)
Compress 1KB with Snappy	2,000 ns (2 us)
Read 1MB sequentially from memory	12,000 ns (12 us)
SSD Random Read	100,000 ns (100 us)
Read 1 MB sequentially from SSD	500,000 ns (500 us)
Read 1 MB sequentially from 10Gbps network	1,000,000 ns (1 ms)
Read 1 MB sequentially from disk	10,000,000 ns (10 ms)
Disk seek	10,000,000 ns (10 ms)
Send packet California->Netherlands->California	150,000,000 ns (150 ms)

Often, there are only a few system configurations to allow reuse and simplify maintenance. The power budget per volume (e.g., per rack) is kept roughly constant (consumed power should be the same for a storage or a compute-heavy rack). Because compute and storage requirements are often uncorrelated, lack of flexibility can lead to imbalanced resource usage. Furthermore, there can be diurnal traffic patterns, spikes, and design for future growth. Disaggregating resources (physically or logically, i.e. when allocating) can help to build balanced systems and therefore improve total cost of ownership.

The total cost of ownership is the key metric in datacenters. It consists of capital expenses CAPEX (facilities, compute, storage, networking) and operational expenses OPEX (energy, maintenance, employees). Facilities and equipment is amortized over time (e.g., 12 years for a building and 3 years for servers). Hardware dominates TCO with approx. 60%.

The power efficiency of a datacenter is:

$$\text{Efficiency} = \frac{\text{Computation}}{\text{Total Power}} = \frac{1}{PUE}\frac{1}{SPUE}\frac{\text{Computation}}{\text{Total Power to Electronic Components}}$$

Where the power usage effectiveness (PUE) is a standard metric for how

much power goes towards useful work:

$$PUE = \frac{\text{Total Facility Power}}{\text{IT Equipment Power}}$$

Ideal would be 1, but because of cooling, the best today is 1.07 - 1.12. Cooling losses are three times greater than power conversion losses. To minimize inefficiencies due to air mixing, hot and cold isles are often isolated. Overall, CPUs consume around 60% of the power. Systems still consume significant energy at low load, but CPU power proportionality has improved over the years (because of different sleep states, frequencies, etc...). SPUE is the server power conversion efficiency.

1.1 Server CPUs

The key differences to take into account for CPU server design (compared to desktop environments) are the higher requests rates, that the requests often involve processing/moving large volumes of data, and that the hardware is shared (which requires isolation mechanisms for performance/security). There are "brawny" high-end (superscalar, out of order, sophisticated speculation) and "wimpy" low-end cores. Low-end cores provide lower performance at much lower power, but because of Amdahl's Law, performance is limited by the serial part of a program, so simply using more CPUs is not always sufficient. In recent years, some providers started to build their own ARM-based processors and NVIDIA is building a CPU to overcome the limited bandwidth between memory and the GPU.

Another consideration is the number of sockets per server. Because many applications are often too large even for the largest server, communication (with its performance penalty) is often needed anyway, which makes clusters of smaller servers often more cost effective.

Even when applications run on separate cores, they end up contending for resources ("noisy neighbor" problem). Intel Cache Allocation Technology (CAT) enables partitioning the last-level cache into several subsets with smaller associativity. However, this partitioning is only based on capacity, not on bandwidth. Similarly, workloads can be isolated across NUMA channels in multi-socket servers, which is also capacity-based.

Dynamic voltage frequency scaling (DVFS) enables scaling the frequency of CPU cores independently. CPU cores can have multiple power states, where p-states are different levels of performance/power consumption while executing code and c-states result in different power consumption/wake-up times while idle. For instance, the CPU is active at highest frequency in

P0 and at lowest frequency in Pn. In C1, the core clock is off (but caches still warm), in C3/C4, voltage is reduced and there is a partial L2 cache flush, and in C6, L2 cache is flushed completely and the state saved to SRAM. The states are controlled by the OS or the CPU hardware.

To know what kind of CPU to build and what to optimize, it is important to measure/characterize datacenter workloads. A survey by Google showed that datacenter workloads are very diverse (with no "killer application"), that the "datacenter tax" (moving data, RPCs, hashing, etc...) consumes roughly 30% of CPU cycles, and that there can be significant front-end (fetching/decoding instructions) and back-end (data cache hierarchy & lack of ILP) stalls. The front-end stalls occur because the instruction working set sizes grew much faster than the L1 instruction cache sizes.

Another challenge is imbalanced resource usage (over time and because of heterogeneous applications). A solution to that is resource disaggregation where compute & storage components are (logically) disaggregated and used over the network. This is very common for storage today, but can be challenging for memory because of performance issues (data plane challenge). Furthermore, there is the control plane challenge, i.e. deciding which resources to allocate to which job.

1.2 Storage & Memory Technology

1.2.1 Memory

SRAM uses latching circuitry to store individual bits, whereas DRAM stores charge on node capacitance (which requires periodic refresh). SRAM is faster and simpler to manufacture (compatible with the logic process), but the density is lower and the cost higher. Parallelism in DRAM is achieved with multiple channels, ranks, banks, and chips. Only whole rows can be read, but the column address is decoded to only select a subset of the row if needed. The memory-access protocol knows five basic commands, ACTIVATE (opening a row), READ (reading a column), WRITE, PRECHARGE (closing a row), and REFRESH. As long as a row is activated, it is in the row buffer and different columns can be read. Therefore, there is an open row policy (keeping the row open after an access) and a closed row policy. In practice, speculative adaptive policies are used. DRAM banks share command/address/data pins, but operate independently, which allows overlapping the ACTIVATE and PRECHARGE latencies. DRAM controller translate memory requests into DRAM command sequences, buffer and schedule requests, refresh the RAM, and manage power consumption. Different parts of the physical address are mapped to the row, bank, and

column address, which gives the OS (by the virtual to physical address mappings) influence over where an address maps to in DRAM, but controllers often also randomize the address such that consecutive addresses end up in different banks (and the OS does not need to randomize the addresses by itself).

1.2.2 Flash Storage

Flash cell store bits as charge trapped in a floating gate. This changes the voltage/current characteristic of the transistor and values can therefore be read by measuring this characteristic (i.e., the current after applying a certain voltage). In single-level cells (SLC), only two voltage/current characteristics are distinguished, whereas multi-level cells distinguish more, which allows storing multiple bits per cell. SLC is faster and more reliable, but less dense. In NOR flash, the cells are arranged in parallel, which results in very fast read, but slow writes and erase. This is sometimes used as an alternative to ROM (e.g., for instruction memory in mobile systems). In NAND flash (used for data storage), they are arranged in serial. Read is still faster than write/erase, but the differences are not that large. Controllers, channels, chips, blocks, and pages provide multiple levels of parallelism. A page (512 bytes - 8KB) is the minimum unit of read/write, whereas a block (64 - x256 pages) is the minimum unit of erasing. Erasing is the process of setting all bits in a block to 1 and it is needed because write can only perform 1 to 0 transitions. Because each chip can process only one operation at at time, there can be significant read/write interference and isolation mechanisms are needed for sharing flash (I/O schedulers, rate limit). Flash cells are physically damaged programming and erasing them (wear out), programming pages can corrupt the values of other pages in the block (writing disturb), and reading data can corrupt the data in the block (read disturb).

Flash is made useful by a flash transaction layer (FTL) that exposes a block-based interface, manages the program/erase granularity mismatch, equalizes wear, and delivers high performance. The user provide a logical block address (LBA) and the FTL performs the logical to physical mapping. It stores the mapping of the LBAs to physical pages centrally and stores various information (erased, erase count, valid page count, sequence number, bad block indicator) about every page. The write point is a pointer to the physical page that will be written next. On a write, the block mappings are updated, the old data is not overwritten in-place. On erase, the valid pages are moved (which requires updates in the mappings, valid page counts, etc...) and the FTL therefore performs extra writes to do an erase. The number of physical pages written to flash for every logical page

written by the user is the write amplification and it should be reduced (by reserving extra capacity, making the program write sequentially, or storing frequently written data in memory). The FTL performs wear leveling, i.e. it maximizes the device lifetime with uniform block wear by keeping statistics on the write/erase cycles. FTLs use DRAM to store mappings, queue requests and buffer writes. But because this information is lost on a power cycle, metadata about blocks and the mappings (in the summary page - the last page of a block) are also stored on the device.

Flash SSDs are attached via Serial ATA (SATA), Serial Attached SCSI (SAS), or PCI Express with the NVM Express (NVMe) command set that was designed for flash. NVMe is a command set & queuing interface for connecting flash to PCIe, each app has pairs of queues that allow for parallelism & isolation and NVMe provides lower latency and higher bandwidth.

1.2.3 Non-volatile memory

NVM technology such as phase change memory, resistive RAM or STT (Spin-Torque-Transfer) RAM provides "storage class memory" and introduces a new layer in the memory/storage hierarchy. They can be attached and look like fast flash devices (Optane SSD), but there is also byte-addressable NVM with lower latency and higher bandwidth (e.g., Optane DIMM).

Because moving data is expensive, a recent trend is processing in memory (PIM).

1.3 Networking I/O

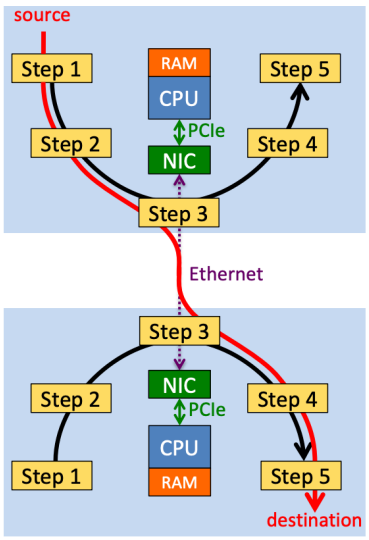
There are different I/O interface mechanisms such as register/queue based device interfaces, memory mapped I/O, direct memory access (DMA), or interrupts and polling. Memory-Mapped I/O uses load/store instructions to read/write locations on devices, which is often used for device configuration. In DMA, there is a separate engine to transfer blocks of data between an I/O device & memory without the CPU, which just describes the transfer to the engine. Transfers therefore do not pollute the CPU cache and can work in parallel to the CPU. But memory becomes inconsistent with CPU caches. Possible solutions for that are:

- Mapping DMA buffers non-cacheable (large performance hit as data is often processed anyway)
- Snooping DMA bus transactions (does not scale beyond small systems)

- Explicitly flushing/invalidating cache regions (important part of device drivers)
- Direct cache access: Inbound I/O traffic is delivered straight into core caches (particularly useful with packet header splitting for placing the headers into the cache).

Another issue is the decision to use physical (no translation needed, but need to use pinned addresses and the physical addresses may not be contiguous, which requires the splitting into page-size DMA transfers) or virtual addresses (very useful for VMs sharing I/O devices). An IOMMU intercepts DMA transactions/interrupts and performs virtual address translation, interrupt remapping (for virtualization), and memory protection for DMA.

Software knows when data is available with interrupts (higher latency) or by polling, i.e. periodically checking the status of a register (processor is always busy). The life of a network packet looks like this: There are



Source (TX)

1. Allocate DMA-able buffer with packet data
2. Write DMA descriptor to NIC
3. NIC DMA engine copies data for transmission
4. TX completion notification (interrupt)
5. Deallocate buffer

Destination (RX)

1. Allocate buffer for received packet
2. Write DMA descriptor to NIC
3. NIC DMA engine copies data to host
4. RX notification to host (interrupt)
5. Forward received packet to user application

different NIC acceleration techniques:

- Checksum offload: The NIC checks the TCP and IP Checksum
- Large segment offloading (at transmit): The NIC breaks up large TCP segments into smaller packets and generates the headers for them
- Large receive offload: Several incoming packets are combined into a larger one.

- Header splitting: The header & payload are placed in separate buffers, which improves locality and allows more prefetching & zero copy

Interrupt coalescing, i.e. processing many packets before raising an interrupt is another optimization technique. A software approximation is to stay in polling mode for a while after an interrupt. Interrupts can also be steered, either all to a single core (high locality of code and network state, but also high imbalance and low scalability), spread across all cores (round-robin or based on hashing; improves scalability, but poor locality and possible interference), or to the app waiting for data (e.g., Intel's flow director based on advanced filters). A SmartNIC, also called Data Processing Unit (DPU), is a programmable NIC that can accelerate networking-related tasks. They can be implemented as ASICs (highest performance, limited capabilities), FPGA (very flexible, difficult to program), or as a System On Chip (very flexible and simpler to program).

1.4 Accelerators

Because exponential compute growth is no longer possible, more specialized accelerators are produced. One type of accelerators are GPUs. They were originally designed to provide the processing capacity for multimedia applications, but became more powerful over time. Neural networks were a perfect match for GPU architectures because of SIMT (single instruction, multiple threads) parallelism and data parallelism. In contrast, CPUs achieve task parallelism through cores (GPUs through many threads), use the SIMD (single instruction, multiple data) model, and make heavy use of caches (whereas GPUs use fast memory). GPUs consist of processing cluster with texture processing clusters and streaming multiprocessors (SMs). The SMs have multiple FP32, INT32, FP64, and tensor (used for ML training) ALUs. They are very useful for large scale multi-threading with no synchronization between threads and for smaller data sizes where processing can be batched. Latency should not be very critical.

Because GPUs are power hungry, expensive, and have limited memory capacity & a high latency, Google designed TPUs for matrix multiplication in reduced precision. They provide higher I/O per watt, no support for graphic processing, large scale parallelism, and fit in a hard drive slot. Like GPUs, the design is hierarchical with modules that are assembled into pods. Single TPUs are driven by a host machine (VM), multiple TPUs are connected to each other through high speed interconnects (without host involvement). TPUs are good for computations that are dominated by low precision matrix multiplication, they are bad for computations with frequent branching.

There are things that the CPU is not good at where GPUs/TPUs do not help, e.g. networking, pipeline parallelism, or data parallelism with irregular memory accesses. For these tasks, FPGAs are used often. They are between ASICs and traditional processors in terms of flexibility/programmability and energy efficiency. Code is written in HDLs (Verilog, VHDL, HLS, ...), a synthesized circuit (collection of logic gates & connections) is created, and then placing & routing (which can be expensive) is done. FPGAs can be used on the side, in the data-path, or as a co-processor. Amazon uses them in their Aqua cloud cache for query acceleration, Microsoft has also developed a system called Catapult with many use cases and to offload common tasks (encryption, network virtualization, etc...).

Chapter 2

Datacenter Software

Software systems are divided into different tiers (also called layers) which can be conceptual or real. The tiers usually appeared because of changes in technology (computers, hardware, networking) and allow to describe the architecture of modern IT infrastructures, abstracting from the system details. The basic tiers are:

- **Presentation Layer** (client, external API layer): Interacting with the outside world (user, other systems)
- **Application Logic** (business rules, business processes): Part of the system that defines what it does/its functionality
- **Data Layer** (databases, storage, business objects): Storage and management of the data being used/processed

These tiers can be distributed and may be complete systems on their own. More modular, interconnected systems open up more opportunities for distribution and parallelism and allow encapsulation, component based design, and reuse. However, it means also that more sessions need to be maintained, more coordination is necessary, and the performance gets worse because of context switches/intermediate steps. There is therefore a big trade-off between clean designs with a clear logical and physical separation of layers and performant designs with highly optimized data paths.

2.1 Multi-Tier Architectures

2.1.1 One Tier

The fully centralized, one tier architecture was the first architecture where all layers were built as a monolithic entity and users/programs accessed

the system through display terminals. There were no forced context switches, everything was centralized (making the management and control of resources easy) and the design can be optimized. Scalability was achieved through scale up and there was no need for interconnections (which was established with screen scrapers at some point later). This still exists as an architectural patterns with similar advantages (no software distribution, centralized upgrades, SaaS with charge per access/volume, centralized control) and disadvantages (no use of resources at the client, integration difficult).

2.1.2 Two Tier

In a client/server architecture, the presentation layer (or parts of it) is moved to the clients. This allows several different presentation layers, one can take advantage of the computing power of the client, and it helped with performance as the resource manager only sees one client (the application logic). This also introduced the notion of service, service interface, and APIs, which started many standardization efforts. The server design is still tightly coupled and its still relatively easy to manage and control. However, the server now has to deal with all possible client connections, clients are tied to the systems since there is no standard presentation layer, and there is no failure or load encapsulation (when the server fails, nobody can work). Furthermore, the client is the point of integration and the responsibility of dealing with heterogeneous systems is shifted to it. The internet brought this architecture to its limit on the server (many clients) and client side (many servers with different presentation layers), but the two tier architecture is the most basic architectural pattern.

We distinguish between thin clients with minimal functionality (small code base, easy to update and upgrade, easy to port, complexity in the server, but no computing capabilities at client, functionality of client limited, and not everything can be done at the server) and fat clients with extended functionality (offloading to the client, added value at the client side, but larger code base, increased maintenance/upgrade costs, backward compatibility issues, and it is difficult to port across platforms).

2.1.3 Three Tier

In a three tier or middleware system, the layers are fully separated and often distributed. To connect the components from many different systems, a middleware is introduced which is a level of indirection between clients and other layers. It simplifies the design by reducing the number of interfaces, provides transparent access to the systems, acts as the platform

for inter-system functionality/high level application logic, and takes care of locating resources and accessing them. But it can hurt performance, is a complex software, and needs to be standardized.

2.1.4 Micro-Services

Nowadays, there are less structured architectures as they allow faster evolution and less standardization because of the cloud. The emphasis is on quick development and reducing the complexity of each module, which allows reuse and simplifies testing and maintenance of each component. This often leads to "death star architectures" of many small, connected services. However, in data centers and 24/7 systems with long lived, statically deployed applications, multi-tier architectures are still common.

2.2 Software Architecture

In enterprise data centers, the main cost is people and is measured in work/$, which often results in consolidation (scale up). In clouds, the main cost is hardware and the scales are larger. Cost is measured in work/watt and the focus is on utilization and scaling out (decentralized systems that provide more flexibility, but are harder to manage).

In cloud environments, storage and compute is usually separated such that compute nodes do not need to invest time for managing their storage. Systems are often specialized and consist of many subsystems. Load balancers, caching layers (intermediate or at compute nodes), key value stores, etc... compensate for design problems.

2.2.1 Performance

Performance has three components: Throughput, latency, and reliability/availability. Many systems operate under a SLA and try to maximize throughput under a latency upper bound.

Throughput is improved via parallelism, which works well for embarrassingly parallel workloads and opens up the possibility of elastic scalability. But, this is limited by cost, system bottlenecks, potentially correlated request, and efficiency/load balancing.

Latency can also be improved by parallelism (dividing a task into smaller chunks), which has an overhead of communication, task splitting, and result gathering. It also requires automating the deployment of parallel tasks. Furthermore, one can shorten the critical path by consolidating

(reducing network communication, making steps faster with more powerful machines/accelerators, making the network faster). But this often worsens throughput because it puts more load on the processing node. Another approach is caching and moving processing to the storage layers.

Reliability is improved through replication, which needs to be done to the entire system/all layers and can create consistency problems. Separating functions helps with reliability as every tier can be made as reliable as needed. Dealing with failures also drives the trend towards stateless (REST) APIs.

2.3 Virtualization

2.3.1 Virtual Machines

Virtualization is a level of indirection, which is similar to abstraction, but the details are not necessarily hidden. A machine can be seen from different perspectives (OS, compiler, or application developer), which leads to different types of VMs (system, process, or language VMs). A system VM introduces the virtual machine monitor (VMM)/hypervisor between the physical host hardware and the VMs which honors existing hardware interfaces to create virtual copies of a complete hardware system. We call the physical platform (hardware and sometimes also the host OS) the host and the additional platforms that run on the VM (OS, apps) the guest. Virtualization allows partitioning (with resource sharing and isolation), as well as encapsulation of the state (which enables checkpoints/restoring, migrations, and execution replays). In the cloud, it is used to share hardware efficiently and securely between multiple users. For instance, it enables server consolidation and load balancing (which can in turn allow to power down some servers).

Implementation

The main requirements are safety (isolation), equivalency (guests should not be aware that they are running in a VM), and efficiency. The main approaches are:

- **Hosted interpretation**: The VMM is run as a regular user application on a host OS and steps through instructions in the VM code, while updating the virtual hardware. This provides complete isolation and it is easy to handle privileged instructions, but emulating a modern processor is difficult and the approach is slow.

- **Direct Execution, Trap-and-Emulate**: In this approach, the VMM uses a policy to handle traps (e.g., executes the instruction on behalf of the guest OS or kills the VM). Generally, the kernel runs in ring 0 (security mechanism that is enforced by the CPU and where calls between rings can only happen through hardware-enforced mechanisms) and applications in ring 3. In this approach, the VMM runs in ring 0, the host OS in ring 1, and apps in ring 3. This approach is faster than hosted interpretation, but still slow because of emulation and the processor needs to be virtualizable. This is the case when there are at least two execution modes/rings and all sensitive instructions (that change the HW configuration or whose outcome depends on it) are also privileged instructions (that cause a trap). In this case, the VMM can interpose any sensitive instructions and can control how the VM interacts with the outside world. However, x86 was not virtualizable for many years, because e.g. `push %cs` exposed that the privilege level is not ring 0, but 1.
- **Direct Execution, Binary Translation**: With binary translation, the VMM dynamically rewrites non-virtualizable instructions and adds instructions that will trap. This allows running unmodified guest OSes / apps and most instructions run at bare-metal speed. But implementing the VMM is difficult and translation impacts performance.
- **Para-virtualization**: Here, the guest OS is modified to remove sensitive, but unprivileged instructions (e.g. done in Xen). The guest applications are unmodified. The problem of the approach is that it requires substantial modifications to the OS.
- **Direct Execution, Hardware Support**: Modern processors (Intel VT-x, AMD-V) provide HW support for virtualization. They have a new privilege mode "VT root mode" that is entered/exited on special instructions (`vmentry`, `vmexit`). The virtual machine control structure (VMCS) allows to configure which instructions trigger `vmexit`.

When virtualizing memory, a challenge is that the guest OS expects contiguous, zero-based memory (guest physical) memory. One approach to do this is page-table shadowing where the VMM intercepts paging operations (usually with traps by marking the memory read-only) and constructs copy of page tables. Extended page tables provide hardware support for memory virtualization. They map guest physical to host physical addresses. On a TLB miss (which caches guest physical to host physical addresses), the hardware page-table walker has therefore to walk two tables (guest virtual to guest physical and guest physical to host physical).

For virtualizing I/O, virtual I/O devices are presented to guest VMs. They translate DMA operations, inject virtual interrupts and trap device commands. There is I/O device emulation and paravirtualization with specialized guest device drivers.

2.3.2 Containers

Containers provide lightweight operating system level virtualization and share the host OS, but have their own system binaries, libraries and dependencies. They are lighter-weight than VMs (higher density sharing, faster startup/shutdown, bare-metal like performance), but provide less secure isolation and require applications to run on the same host OS. Docker provides a standard way to package an application and its dependencies, it is built on top of linux containers (LXC) whose goal is performance isolation. Resource isolation is done with namespaces and CPU cores, memory, and bandwidth limits are managed with cgroups. Namespaces abstract a global resource (e.g., PID for what other processes are visible) and its goal is to restrict what a container can see (changes are only visible to other processes in the same namespace). Control groups (cgroups) limit the resources that a container has access to, they are implemented with a virtual file system `cgroupfs` and have several subsystems. With `seccomp-bpf`, the system calls a container can call are limited and `chroot` provides an isolated filesystem.

A docker image is a read-only template for creating a container, a container is the runnable instance of an image. Images are divided into a sequence of layers that built upon each other (defined in a Dockerfile). The container runtime is responsible for starting and managing the container (unsharing of the namespace, creation of cgroup, etc...).

gvisor creates a container and sandboxes it (by intercepting system calls and performing them in a secure user space) which can be good for running untrusted applications.

Container orchestration tools like Kubernetes manage the complexity of the container lifecycle (Scheduling, Lifecycle/Health, Scaling, Naming/Discovery, Load Balancing, Storage Volumes, Logging/Monitoring, Debugging/Introspection, Identity/Authorization). The Kubernetes master manages different nodes (kubelets) and can be administrated over different APIs. A pod is a bunch of containers with the same lifecycle, network, and storage volume and is intended to run a common task. It is the unit of scheduling and migration in Kubernetes. Services are a group of pods that work together and enable load balancing among pod replicas. They can have a stable virtual IP address (managed by kube-proxy which runs

on all nodes). Service meshes like `lstio` make it easier to connect, manage and secure traffic between services. They consist of a data plane that establishes connections to other services (with proxies that are deployed within each container) and a control plane which allows configuration of communication management, load balancing, security policies, etc... Labels can be added to Kubernetes nodes and these labels can be queried to place containers on specific nodes. ReplicaSets ensure that N copies of a pod are running and the horizontal pod autoscaler automatically scales pods as needed based on CPU utilization (but own controllers can be implemented as well). Kubernetes knows the notion of requests and limits: Requests for a resource (CPU/RAM) provide a strong guarantee of availability (no over-commitment) whereas limits set the maximal amount of a resource the container can access. Based on this, there are three QoS levels:

- Guaranteed: request > 0 && limit == request (highest protection)
- Burstable: request > 0 && limit > request (medium protection)
- Best effort: request == 0 (lowest protection)

Firecracker

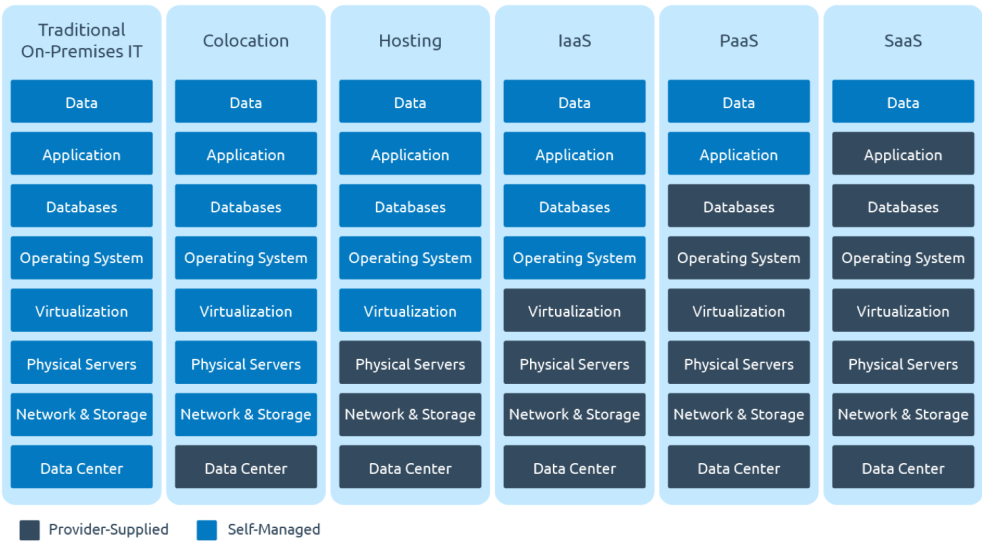
Firecracker is a VMM that is specialized for serverless workloads. Its goal is to provide both strong security (like VMs) and minimal overhead (like containers). The minimal VMM (small codebase, unnecessary drivers removed, etc...) is built on top of KVM and provides fast boot times (150ms) with a minimal Linux kernel and low overhead. Devices can be rate-limited and the system has a minimal REST API. It therefore is able to provide characteristics of containers with virtualization. Amazon uses Firecracker in Lambda and Fargate to run untrusted user input with a small pool of pre-booted "MicroVMs" and thousands of MicroVMs on every host.

2.4 Business Model

Using the cloud results in operational expenses instead of capital expenses that depreciate and can enable scaling that was not possible before (because of the investment required). The cloud works because of utilization, servers are often unused and the TCO is very high compared to the usage. The cloud provides higher efficiencies by sharing resources. The largest cloud providers dominate the server market and are starting to build their own servers and processors.

2.4.1 Service Models

Besides On-Premise and the cloud, there are hybrid deployments where parts are local and parts in the cloud. Furthermore, there are different abstraction/service levels that vendors provide: In IaaS, a VM (or a ded-



icated server which is virtualized but not shared or even a bare metal server) is rented. Oftentimes, the providers offer other services such as VM image management, storage services, messaging services, etc... In PaaS, the provider offers common services (web servers, analytic engines, etc...) and the user needs to put the application together using the services provided. On the other hand, the user only provides the data and gets the application by the cloud when using SaaS. FaaS or serverless allows to encapsulate a small application into a function and the cloud executes this function. This is usually charged by the millisecond. For very short tasks executed rarely, this can be cheaper. Furthermore, functions can be started very quickly / automatically and elasticity is automatic. However, it has still some limitations (stateless, short lived, no communication) and only works well in concrete use cases.

2.4.2 Architectures

Synchronous RPC messages are often used for communication in cloud environments. An alternative are asynchronous queues that detach different components. To prevent tight coupling, cloud providers often offer a load balancing service. This allows higher availability and distributing

the load, but it adds latency. For security reasons, there is often a DMZ (perimeter network) and network groups to separate services with different privileges. The VMs are often divided into different tiers, like the web, business, and data tier. The data tier is implemented using a VM and running a database on it or using automatically managed data services. When storage is provided as a service, specialized extensions such as predicate pushdown or hardware accelerated data processing and caching are possible. DNS resolution (geo-routing, flow/traffic control) and content management (CDN) can be important services of cloud providers. Furthermore, all major cloud providers provide automatic scaling systems that scale resources based on certain parameters (availability, cost, utilization, etc...).

Availability zones and regions increase availability because systems are in different locations and the failure of one location (machine, network, rack, cluster, or even data center) does not affect all systems.

2.5 Cluster Management

Cluster management is about server configuration & package management, application scheduling, and application naming/discovery. The problems are very similar for VMs and containers. Cluster scheduling consists of resource allocation (how much resources to give to an application) and resource assignment (on which physical nodes to allocate the resources on). In private resource assignment, each app receives a private, static set of resources (static partitioning), which is simple, provides performance isolation and allows HW specialization. But it can result in low utilization and failures/maintenance may be hard to handle. In shared resource assignment, machine resources are shared between application. A scheduler ensures isolation, efficient resource usage, and flexibility. In fair sharing, we give $1/n$ of the shared resource to each of the n users. Max-min fairness handles the case if a user wants less than its fair share, the remaining portion is split fairly among users. This is generalized by weighted max-min fairness. Max-min fairness results in a share guarantee (each user gets at least $1/n$ of the resource) and is strategy-proof (users are not better off by asking for more than they need). Dominant resource fairness (DRF) extends it to multiple resources. The dominant resource is the resource that the user has the biggest share of and the dominant share the fraction of the dominant resource allocated. DRF then applies max-min fairness to the dominant shares. Mesos uses an online DRF scheduler and schedules a task to the user with the smallest dominant share.

For resource assignment, there are multiple cluster manager designs: In

a centralized cluster manager (e.g., Borg by Google or Quasar) there is a single manager that is responsible for the entire cluster. There are also two-level designs with one central master and multiple framework schedulers (e.g., Mesos or YARN) where the master sends resource offers and application frameworks select which offers to accept and which tasks to run. Finally, there are distributed managers (e.g., Omega or Sparrow) with multiple concurrently operating scheduling agents that probe workers independently and queue up tasks. This can be improved by batching (grouping tasks) and late binding (reserving slots and then submitting to the first slot where the reservation is at the head of the queue). Because of overprovisioning (wrong specification of resources), resource utilization in shared clusters is often relatively poor, which Quasar tries to solve.

2.5.1 Quasar

Translating performance goals to resource requirements can be very hard because they depend on many factors (interference, server heterogeneity, etc...). Because of that, the cluster manager Quasar performs resource allocation and assignment jointly. Users specify performance goals (queries per second and latency constraint, execution time, or instructions per second) and Quasar finds an allocation and assignment that satisfies it. It does so by profiling services (in parallel using a sandbox/containers) with respect to interference (using microbenchmarks), heterogeneity (server type), scale up, and scale out. Collaborative filtering is used to classify the services based on this sparse profiling data (with data from benchmark and previous runs). A greedy algorithm uses this classification results to select the least amount of resources that satisfy the performance constraint (by first choosing the servers with the best resource quality, and preferring scale up over scale out). Workload performance is continuously monitored (and interfering microbenchmarks are actively injected) in order to detect phase changes or wrong classifications and adapt to them by providing more or less resources.

2.6 Systems for Machine Learning

In deep learning training, we care about high throughput and fast convergence to a high accuracy. In (online) inference, we also care about low latency. ML applications are computationally intensive (room for optimization with hardware acceleration), error-tolerant (trade-off between accuracy, speed, cost) and need to process large training datasets (oftentimes not fitting in memory). "Software 2.0" consists of data and models and needs a new ecosystem. There is a lot of surrounding infrastructure re-

quired for it. End-to-end machine learning consists of model development, training, and inference. During model development, there is a data part (consisting of data collection and cleaning & visualization) and a model part (consisting of training & validation and feature engineering & model design). After model development, we ideally want a training pipeline that allows retraining, tracking data and code, capturing dependencies, and auditing training for compliance. Inference can be done by embedding the model inside an end-user application or with a prediction service.

There are different software frameworks for ML with the goal to make development easier, automatically compute gradients (using backpropagation), and run the models efficiently on hardware and at scale. A tensor is a multi-dimensional array with elements having a uniform type and the frameworks usually support tensor operations. There is a computation graph (DAG) where nodes are mathematical operations and edges data (tensors) flowing between operations. In static graphs (original TensorFlow), the framework first builds the graph and then executes it, whereas it is built during the execution in dynamic graphs (PyTorch, TensorFlow 2.0 with eager execution). Static graphs can result in more optimization opportunities whereas dynamic graphs are more intuitive/familiar for programmers and it is easier to support dynamic control flow.

For distributed training, there is data parallelism (partitioning the data and running multiple copies of the model) and model parallelism (partitioning the model). For model parallelism, pipelining can be used to increase parallelism. Distributed training can be synchronous (better model quality) or asynchronous (higher throughput).

2.7 Communication APIs

2.7.1 RPC

RPC was created when designs shifted from one tier to a two tier architecture, because communicating at the level of sockets is complicated and not efficient from a programming perspective. It needed to fit the model of the language and inherited many aspects of the languages (which is now considered a bad thing, because too much details are hidden and it is not the same to make a local call than a remote one). RPC turns a procedure call into a call to a remote procedure. Calls are replaced by a stub (automatically generated) that presents the same interface, but manages the remote call. Binding consists of finding the remote procedure (IP address, port, machine, etc.) using a name and directory service. Marshalling (transforming memory representation into a format suitable for transmission)

and serialization (which can be expensive) map the call parameters to a format suitable for transmission, which implies some standard intermediate representation. For these steps, there is usually a name and directory service (where servers register their procedures), an interface definition language (that defines how to declare a procedure as a remote procedure) and serialization which is done by the stub. There may be other services provided by the environment for security, system management, threads, files, etc... RPC is simple, follows the programming techniques of the time, and it allows modular/hierarchical design of large distributed systems. But it is not a standard (there are multiple implementations), it is only a low level construct and does not solve many of the problems distribution creates, and it was designed with only client/server in mind. Issues are if a remote procedure should be transparent, if the location should be transparent, and if there should be a centralized name system. RPC is synchronous, which can be made very efficient, but results in tight coupling and high latency. Furthermore, it is difficult to recover from failures (there are different semantics with regards to the number of times a call may be executed, at least once, at most once, and exactly once). CORBA and DCOM are RPC implemenations with many services attached to it. RPC is still in use today, but the implementations have changed significantly (e.g., gRPC which uses protocol buffers and HTTP/2).

2.7.2 REST

RPC was tied to TCP/IP (problematic for firewalls) and to programming languages. Representational State Transfer (REST) is a general concept and its most common implementation uses HTTP. APIs describe resources (instead of services) that can be anything. Resources have a unique identifier (URI) and data is exchanged using character formats, e.g. JSON. The interaction is stateless with no dependencies between calls. REST allows more than programs, decouples the service for the implementation and the transport protocol, uses standardized representations, simplifies the interfaces (GET, POST, PUT, DELETE), and is scalable because it is stateless (which means that calls can be treated independently).

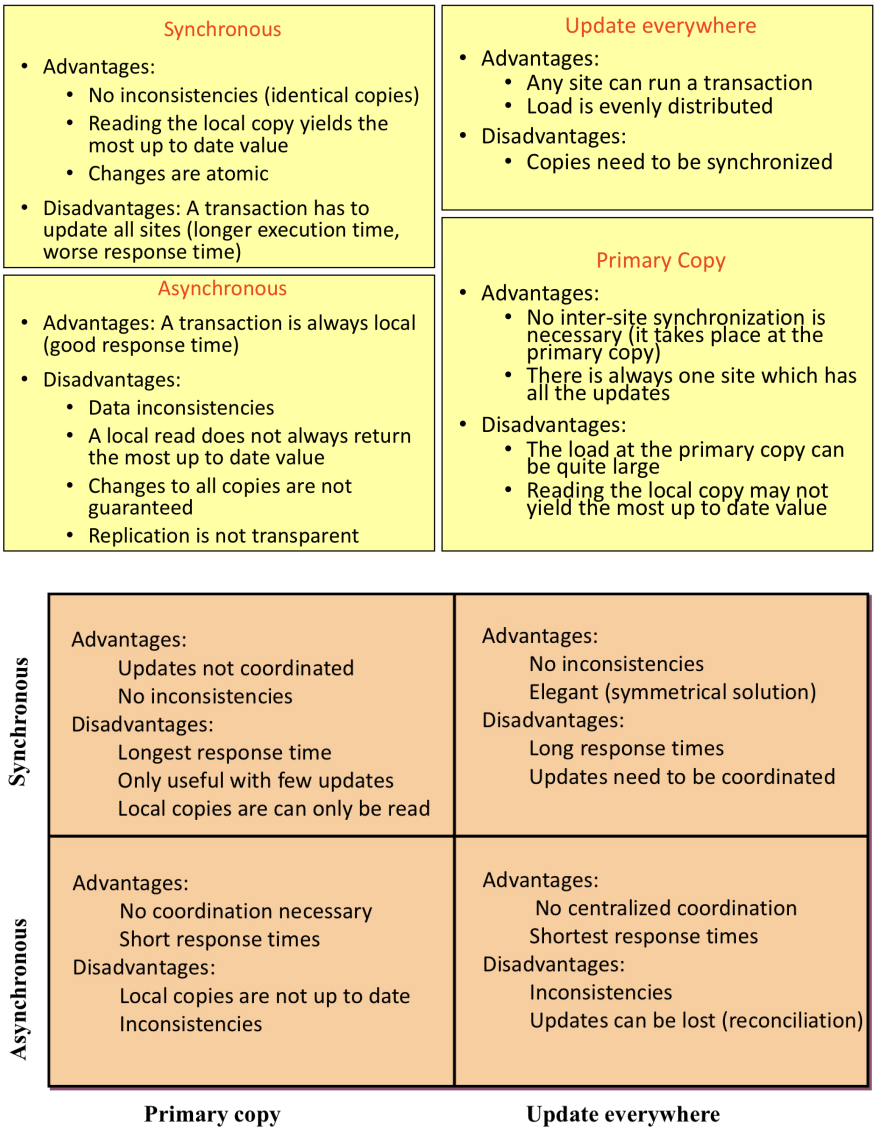
There are different design patterns for REST systems:

- **Materialized View Pattern**: To avoid long chains of calls, a local copy of a materialized view is kept. This improves performance, but there are now two copies of the data that need to be updated.
- **Service Aggregator Pattern**: In this pattern, a service enforces the sequence of calls from a centralized place.

In message queuing systems, there are queues to store messages persistently (and insert/delete them transactionally), which helps with reliable asynchronous communication. They can be implemented on top of databases or cloud storage (Azure Store Queues). When a message broker is added, one can implement publish subscribe (event based) systems where clients subscribe to different types of messages and get notified.

2.8 Data Storage

Storing data in the cloud has unique challenges, it cannot be based on local disks (no flexibility), must scale and provide reliability/availability at a reasonable cost, must work with existing applications and should take advantage of cloud services. The most distinct aspect is the clear separation between the storage and compute layer. Furthermore, data replication at all levels is needed because failures will occur (planned or unplanned events). Replication is common for performance and fault tolerance, it can reduce the load on certain sites, the response time and the criticality of systems. One needs to decide when the updates are propagated (synchronous/eager or asynchronous/lazy) and where the updates can take place (primary copy/master or update everywhere/group). This results in four possible replication strategies. When using synchronous replication, data is often replicated in the context of transactions. There is an agreement protocol (or locking) that sites use to consult with other sites before updating. Most database systems and almost all data storage systems use synchronous replication. It is expensive, but it hides the fact that data is replicated from the application. With asynchronous replication, the copies are inconsistent during propagation (which can take from milliseconds to days, depending on the application). Updates are propagated to all sites with push or pull models and there needs to be a reconciliation protocol for conflicts. These systems are usually called "eventual consistency" in the cloud and are used when the lack of consistency can be tolerated. Inconsistencies are pushed to the application (which may not be a good idea). Many systems use update everywhere and some require a quorum for updates: When we have a read quorum R, a write quorum W and the number of copies N, we need to have $N < R + W$ and $N < W + W$. Then (with reads from R copies and writes to W copies), we are guaranteed to always read the latest value. Primary copy approaches are used by databases or solutions where the copies are there for back-ups. The advantages/disadvantages are summarized in the following graphic: These are the advantages/disadvantages of the four possible replication strategies:



2.8.1 CAP Theorem

Any two of the following three properties can be achieved at the same time, but not all of them:

- (Atomic) **Consistency**: All nodes see the same data.
- **Availability**: It is possible to query the database at all times.
- **Partition Tolerance**: The database continues to function even if the network gets partitioned.

No distributed system is safe from network failures, thus network parti-

tioning generally has to be tolerated. In the presence of a partition, one is then left with two options: consistency or availability. When choosing consistency over availability, the system will return an error or a time out if particular information cannot be guaranteed to be up to date due to network partitioning. When choosing availability over consistency, the system will always process the query and try to return the most recent available version of the information, even if it cannot guarantee it is up to date due to network partitioning.

In the absence of network failure – that is, when the distributed system is running normally – both availability and consistency can be satisfied.

The tradeoff between the options is complex (involving performance and cost) and some systems (e.g., CP) might not make sense in practice. Availability has many aspects, a slow system might also be considered unavailable.

2.8.2 Storage Systems

We distinguish between:

- **Block Storage**: This mirrors the interface of a local disk, the access is by blocks/pages and the application manages the data through a low level interface. Application that manage their own data (e.g., databases) commonly use block storage. In clouds, it is provided by services that have similar interfaces and are available through the network, which allows replication, choice of HDD/SSD, etc... Performance can be optimized with local cache disks.
- **File Storage**: The interface to the user is a collection of blocks called a file with some metadata. It is built on top of block based devices. Oftentimes, automatic replication is supported in these systems (as well in block storage). When data is replicated, it can be read from different places, which improves performance.
- **Object Storage**: Objects consist of data and metadata plus system identifier, they are managed through a REST API. Usually built on top of file systems. The availability is usually very high, which can be achieved more easily because the REST API decouples the interface from the implementation and makes replication much easier. Objects are usually immutable and can have a lot of metadata attached to them.

Furthermore, there are higher level concepts such as key-value stores or databases where the content of the data is known/interpreted. To better

navigate the cost/performance tradeoff, cloud providers often offer intermediate caches or key-value stores. In key-value stores, replication is easy because the key can be hashed to get the location for the value. Data can be kept in memory, which allows the usage as caches.

Traditional databases in cloud environments run on a VM using a block storage, cloud native databases run on VMs using different forms of storage and cloud services, whereas databases as a service run on top of object storage but do not store data themselves, they use the object storage as input. However, they often generate too much traffic in terms of I/O. Amazon Aurora (on top of MySQL) therefore separates storage and cache management from transaction processing. Snowflake is a data warehouse specialized for analytical queries (cloud native). It separates compute from storage and uses S3. Worker nodes have a cache with a simple LRU replacement policy. Instead of extents, it uses micro-partitions between 50 and 500MB (before compression). Data in them is stored in columnar form and they contain metadata (that can be read without reading the whole micro-partition) which allows pruning (skipping whole micro-partitions if the looked for data is not in there). This pruning allows Snowflake to not use any indexes. Because S3 does not support updates in place, micro-partitions are updated in its entirety, which Snowflakes uses to implement snapshots of the data including time travel queries.

Chapter 3

Availability and Reliability

In the cloud, failures will occur simply because of the scale of the system. The most common causes are software, configuration, human error, and the network.

3.1 Reliability

Reliability is the ability of a system to offer the service it was designed for. It is influenced by errors, faults, failures, and performance issues and is typically measured in terms of system availability.

When we observe F failures during an interval of time T, the frequency of failures/failure rate is F/T and the mean Mean Time to Failures (MTTF) is T/F. This assumes uncorrelated failures and components in isolation and is therefore an upper bound (real failures are likely to happen more often). Assuming N independent systems, their MTTF is $\text{MTTF}_{\text{one}}/N$. This decrease in MTTF leads to the notion of "best effort": Systems that try their best to work correctly, but there is no guarantee (e.g., search engines that consider only partial results when failures happen). There is often also a trade off between parallelization and reliability because of this decrease.

The Mean Time Between Failures (MTBF) also includes the Mean Time To Repair (MTTR), which is the time between a failure occurring and the moment the system is operational again. It is defined as $\text{MTBF} = \text{MTTF} + \text{MTTR}$. Availability is then the: Time system is working / total time = $\text{MTTF}/(\text{MTTF} + \text{MTTR})$.

3.2 Availability

Availability is often defined as "number of 9s" with:

Uptime	Downtime in one year
99% (two 9s)	87.6 hours
99.9% (three 9s)	8.76 hours
99.99% (four 9s)	53 min
99.999% (five 9s)	5 min
99.9999% (six 9s)	32 sec
99.99999% (seven 9s)	3 sec

When modeling MTTF of multiple systems (e.g, backup systems), we often use the Markov model with memoryless distributions. For instance, when we have two components with a MTTF of 10 years and the other component takes over if one fails, the MTTF of the whole system is $10/2+10 = 15$ years.

Different system configurations are possible, parallel system with redundant components tolerate the failure of one component and the capacity doubles if both are active. On the other hand, with redundant components and a majority vote, errors can be detected.

We distinguish between errors, faults, and failures:

- **Error**: We get a result, but it is incorrect. This should be detected and corrected by the system.
- **Fault**: Some part of the system is not working and, as a result, some of its functionality might be compromised. This should be detected and is typically compensated through redundancy.
- **Failure**: The system is not working. This can be in many different ways, e.g. degraded performance, unstable behavior, unreachable service, or a service that is down (with either a clean shutdown or corrupted/lost data). These situations require different approaches.

Fault tolerance must be implemented at all levels and for all components (e.g., power & cooling, network, scheduling, and the application).

3.2.1 Mechanisms for Fault Tolerance

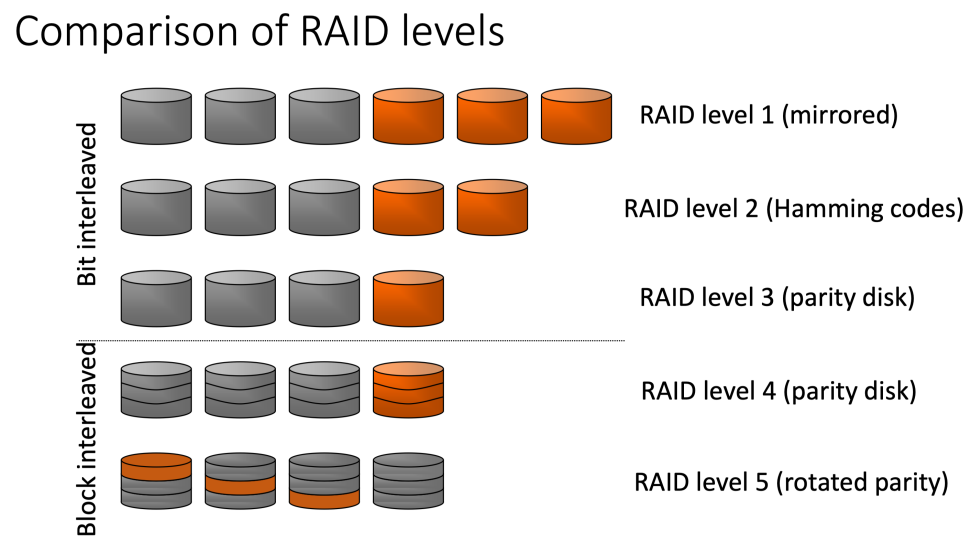
A basic mechanism for fault tolerance is replication. It is used for availability and/or performance. There is data replication, error correcting/detecting codes, and sharding (which is not exactly replication, but often combined). In data striping, data is distributed among several disks such that it can be accessed in parallel. This increases disk bandwidth (concurrent retrieval) and decreases seek time. There is the choice between fine grained (where all requests are serviced using all the disks and there is only one logical I/O request at a time) and coarse grained striping (which

uses large block, such that small requests can be serviced in parallel since they only access a few disks).

Striping and parity can be done bit-interleaved (at the bit level) or block interleaved (on a per block basis), which leads to different RAID levels:

- **Level 0**: Data is striped across disks without any redundancy on a per block basis. The I/O bandwidth is improved N times, but there is no fault tolerance.
- **Level 1**: Individual bits are mirrored and read operations are performed on the copy that offers the smallest seek time (with potential for two parallel read operations). Recovery is trivial and multiple failures can be handled.
- **Level 2**: Works on a bit level, but fault tolerance is provided with Hamming codes. Recovery is more complex, $\log N$ disks are needed for parity and the I/O bandwidth is therefore $N - \log N$.
- **Level 3**: Also works on individual bits with a disk devoted to storing the bit-wise parity of the other disks. Recovery is simple, one disk failure is tolerated and the I/O bandwidth is $N - 1$.
- **Level 4**: This level is block interleaved, i.e. works on blocks of arbitrary size (the striping unit). Like in level 3, there is a disk devoted to store the block-wise parity of the other disks. But because the disks work independent (in contrast to level 3), reads can be performed in parallel (when the blocks are on different disks).
- **Level 5**: RAID 5 is like RAID 4, but the block-wise parity is uniformly distributed across all disks, which allows parallel write operations (in contrast to level 4, where all write operations access the parity disk).
- **Level 6**: Like level 5, but with an additional parity block (tolerates two failures).
- **Level 10**: A level 0 controller is used to stripe the data, each striping unit is mirrored by a RAID level 1 controller.
- **Level 0 + 1**: A level 1 controller is used for mirroring the data, level 0 controllers for striping the mirrored disks. This results in worse failure behavior than level 10 (when a disk in both level 0 groups fail, the data is lost, whereas $N/2$ can fail in the best case for level 10), but reads can be executed in parallel.
- **Level 53**: A level 0 controller is used to stripe the data, each striping unit is controlled by a level 3 controller.

The different levels are visualized in the following graphic:



Another important decision when sharding is the size of shards (big shards may cause hot data to be on the same place and make load balancing harder) and the placement. Random placement is good for independent failures, but when we have correlated failures, the probability that the failing servers have all the data for a replica is large, which increases the probability of losing data. Copysets (a fixed set of nodes containing all copies of a data chunk, ideally such that correlated failures are avoided) are a solution to that. While the probability of data loss is minimized, the amount of data that is lost in case of a loss event increases. Furthermore, the recovery time increases.

A completely different mechanism for fault tolerance is eventual consistency which boosts both performance (no need to update all replicas) and availability (failures of one or more replicas can be ignored). However, applications must be able to operate with inconsistent data and bringing the system back to a consistent state can be expensive.

Load balancing combines well with sharding and is made much easier with REST (because of its independent requests).

Another approach to deal with stragglers/failures is redundant execution. Jobs are sent to multiple machines and as soon as one returns, the other jobs are killed.

There are also different techniques for monitoring. A job can be first run in a "canary" and only run at scale if it has no problems. With watchdog

timers, the responsiveness of machines can be regularly checked. Furthermore, regular integrity checks make sure that data is not lost or corrupted.

Chapter 4

Security

In the cloud, there are untrusted users sharing hardware (which requires isolation between users) and the cloud provider may not be trusted. The main security goals are confidentiality (not disclosing information to unauthorized users), integrity (not allowing unauthorized users to modify information), and availability. Other goals are access control, privacy, and attestation (non-forgeable evidence of identity/state, e.g. making sure that the provider is executing the right program). Data must be protected at rest, in transmission, and in use. The last point is the hardest, we can either compute on encrypted data (homomorophic encryption), which is usually too slow or compute in "trusted execution environments" (confidential computing) with hardware support. With confidential computing, we do not have to trust the OS/hypervisor and data is even secure when the provider is compromised.

The weakest link determines the security of the system (end-to-end model) and we need to get security policies and mechanisms right. Hardware's role is to implement security mechanisms, reduce the trusted code base, and accelerate expensive software mechanisms (e.g. SGX and AES crypto extensions).

4.1 Hardware Support

Most hardware provides acceleration for commonly used crypto codes, either with security instructions or a security coprocessor. Furthermore, it can often produce truly random number (e.g., Intel's DRNG that uses thermal noise).

The Trusted Platform Module (TPM) is a specialized chip that stores the encryption keys specific to the host system for hardware authentication.

It is used for secure boot (trusted boot block, stored on the TPM; used to measure the BIOS, the OS, and any software before running to ensure that the measurement matches the expected signature, e.g. the SHA-1 digest of the code) and certifying the system to external users.

The goal of confidential applications is to run applications with sensitive data in untrusted environments. The most common approach is to create trusted execution environments (TEEs) with hardware support, enabling software enclaves to execute. The code inside the enclave is encrypted and the root user cannot extract the secrets. Intel's Software Guard Extensions (SGX) are extensions that support enclaves, attestation (proving what code is running in enclaves), and a minimum trusted computing base (only the processor is trusted, all writes to memory are encrypted because DRAM is untrusted). The processor furthermore prevents access to cached enclave data outside of the enclave and enclaves can only run unprivileged code (no system call). The enclaves are a highly secure part of a bigger process (e.g. the one that handles the secret keys), not the whole application. Entry and exit is restricted and done with special instructions:

- `ECREATE`: Establishing the memory address for an enclave
- `EADD`: Coping memory pages into the enclave
- `EEXTEND`: Computing the hash of enclave contents
- `EINIT`: Verifying that the hashed content is properly signed and initializing enclave if so.
- `EENTER`: Calling a function inside the enclave
- `EEXIT`: Returning from the enclave

Secrets are inserted into the enclave at runtime with remote attestation. The app receives a public key and validates the certificate of the enclave. If validation succeeds, data is encrypted with this public key.

AWS provides Nitro enclaves based on the AWS Nitro system (custom hardware for network/storage I/O acceleration) and Google Confidential VMs based on the Asylo open-source framework. Systems like AWS Nitro Enclaves enable securing private keys in an enclave, tokenizing sensitive data, or multi-party computation where highly sensitive data is processed from multiple parties without sharing the actual data.

4.2 Side Channels

A side channel is a source of unintended communication between isolated components. A covert channel is a hidden channel used to get information out (whose existence is in contrast to side channels intended), e.g. malware that establishes a hidden channel. Covert channels can be established by observing usage or noise patterns on shared resources.

Spectre are a large class of side channel attacks which are enabled by speculation mechanisms in CPU hardware. They are hard to fix and some fixes come with high costs. The fundamental issue is that the ISA preserves functional behavior, but the side channels uses performance behavior to leak informations.

- Step 1: Train some microarchitectural state, e.g.:
  - Clear 256 blocks in cache at address X.
  - Prime branch predictor for a branch to “not taken.”
- Step 2: Save secret in the microarchitecture, e.g., cache state:

```
beq r0, r1, elsewhere  ; predicted “not taken”
                       ;  but will branch to “elsewhere” this time
lb r2, (r1)  ; load secret at “protected address(r1)”, load will abort
lw (X + r2) ; instruction aborts but causes cache fill
            ; the block loaded in cache depends on value of r2
            ; r2 is the secret from protected address r1!
```

- Step 3: Extract secret from the microarchitecture, e.g., cache state:
  - Time accesses to 256 cache blocks at X; the block that hits (and is faster) corresponds to the value of byte at address r1

4.3 Homomorphic Encryption

An alternative idea is to encrypt the data before it’s stored and to compute on encrypted data. In 2009, it was shown that any function can be calculated on data after it has been encrypted. However, the best encryption schemes took more than >1,000,000x as long to complete compared to unencrypted computations. CryptDB is a cal DBMS to process most SQL queries on encrypted data with only 26% overhead. They designed a SQL-aware encryption strategy and dynamically adjust the encryption based on the level necessary for the queries. Data is stored encrypted in regular tables, SQL user-defined functions are used for server-side cryptographic and a proxy rewrites SQL instructions.

Chapter 5

Appendix

5.1 Background

5.1.1 Interactive Law

For N users with a thinking time of Z and a response time of R, we have:

$$X = \frac{\text{Jobs}}{\text{Time}} = \frac{N\frac{T}{R+Z}}{T} = \frac{N}{R+Z}$$

Or:

$$R = \frac{N}{X} - Z$$

5.1.2 Amdahl's Law (Strong Scaling)

If we use the same load and add resources, the speedup $RT(1)/RT(N)$ is ideally a linear function. However, Amdahl's Law says that we have for a parallelizable part P:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

5.1.3 Weak Scaling

When we test the ability of the system under test to deal with larger loads by adding resources and use N units problem, $\text{ScaleUp}(N) = RT(1)/RT(N)$ is ideally a constant function.

5.1.4 OS terminology

- **Trap**: Any kind of transfer of control to the operating system.

- **System Call**: Synchronous (planned) program-to-kernel transfer, e.g. reading a file or allocating memory.
- **Exception**: Synchronous program-to-kernel transfer caused by exceptional events (divide by zero, page fault, page protection error, etc.)
- **Interrupt**: Asynchronous device-initiated transfer (network packet arrival, keyboard event, timer, etc.)