# Computer Systems

by dcamenisch

## 1 Introduction

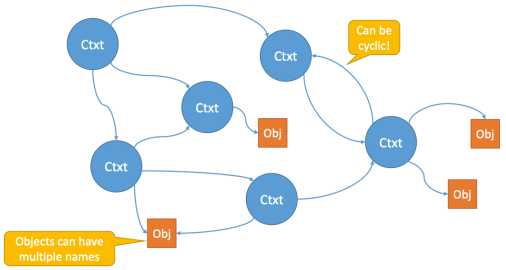
This document is a summary of the 2022 edition of the lecture *Computer Systems* at ETH Zurich. I do not guarantee correctness or completeness, nor is this document endorsed by the lecturers. If you spot any mistakes or find other improvements, feel free to open a pull request at `https://github.com/DannyCamenisch/systems-summary`. This work is published as CC BY-NC-SA.



# Operating Systems

## 2 Naming

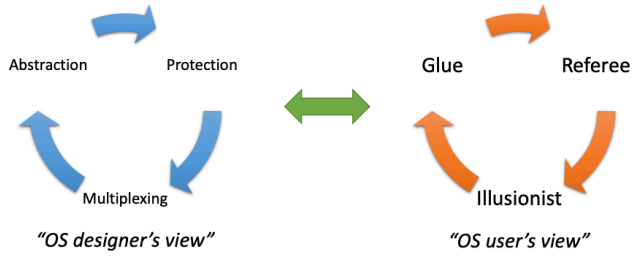
Naming is a fundamental concept, it allows resources to be bound at different times. Names are bound to objects, this is always relative to a context. One example of this would be path names, e.g. */usr/bin/emacs*. Name resolution can be seen as a function from context and name to some object. The resolved object can be a context in itself. This gives us a naming network.




Both synonyms (two names bound to the same object) and homonyms (the same name bound to two different objects) can occur.

## 3 The Kernel

There are three main functions to an OS. Commonly they are referred to from a designer's view, but the user's view can be much more helpful to actually understand how it works.




### 3.1 Bootstrapping

The term bootstrapping refers to pulling himself up from his own boots. In computer systems it is what we call the process of starting up a computer (booting). This boot process looks like this:

1. CPU starts executing code at a fixed address (Boot ROM)
2. Boot ROM code loads 2nd stage boot loader into RAM
3. Boot loader loads kernel and optionally initials file system into RAM
4. Jumps to kernel entry point

The first few lines are always written in assembly, but generally we want to switch to C as soon as possible.

### 3.2 Mode Switch

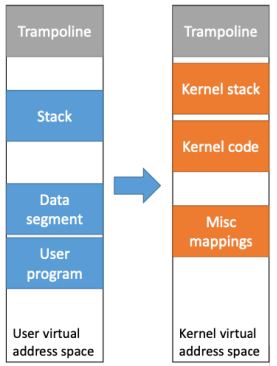
One of our main goals is to protect the OS from applications that could harm it (intentionally or not). For this purpose we introduce two different modes:

- **Kernel Mode** - execution with full privileges, read/write to any memory, access and I/O, etc. Code here must be carefully written
- **User Mode** - limited privileges, only those granted by the OS kernel

These two (or more) modes are already implemented in hardware. The main reason for a mode switch is when we encounter a processor exception (mode switch from user to kernel mode). If this is the case, we want the following to happen:

1. Finish executing current instruction
2. Switch mode from user to kernel
3. Look up exception cause in exception vector table
4. Jump to this address

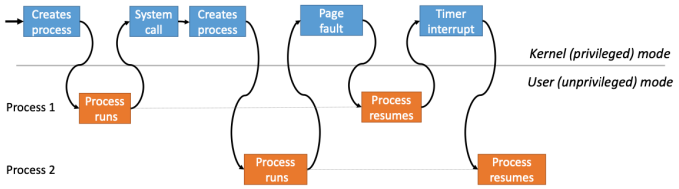
Further we may also want to save the registers and switch page tables. When switching between the modes we also have to change our address space, but we might want to access some informations from the user mode address space. One way of doing this is to use a so called **trampoline**, which is a part of the address space that gets mapped to the same location in user and kernel mode.




Mode switches can also occur the other way around (from kernel mode to user mode). The main reasons for this are:

- New process / thread start
- Return from exception
- Process / thread context switch
- User-level upcall (UNIX signal)

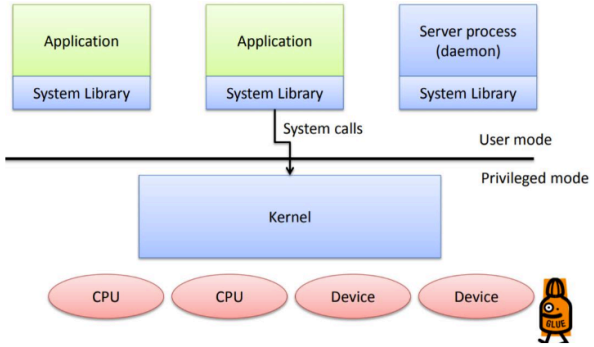
This leads us to the following perspective:




The mode switch is fundamental to modern computers:

- It enables virtualization of the processor
- It creates the illusion of multiple computers
- It referees access to the CPU

## 3.3 General Model of OS Structure




- **Kernel**: The kernel consists of software run in privileged mode. There might be more than one kernel in an OS. It can handle system calls, h/w interrupts etc.
- **System Libraries**: They are part of any application run on the machine and should provide an interface for the kernel.
- **Daemon**: This is a process running as part of the OS. Its not run in privileged mode and may use system libraries. An example might be a file system.

We can differentiate between monolithic kernels and microkernels, depending on the amount of code in kernel mode.

## 3.4 System Calls

System calls are the only way for user mode programs to enter kernel mode. System calls are a type of exception, but they try to look a lot more like a procedure call. Therefore the kernel system call handler has to first locate arguments, copy these arguments into kernel memory, validate these arguments and then copy the results back into user memory after execution. An example of such a system call would be *write()*.

## 3.5 Hardware Timers

What happens if a user mode program does not cause any exception and does not give control back to the kernel? Hardware timers are a solution for this problem, the hardware device periodically interrupts the processor and returns control to the kernel handler, which sets the time of the next interrupt.

# 4 Processes

When you run a program, the OS creates a process to execute the program in. A process is an illusion created by the OS, it creates an execution environment for a program. This environment gives the program limited rights (access, name spaces, threads, etc.) and therefore it is both a security and a resource principal.

## 4.1 Creating a Process

There are two main approaches to creating new processes:

- **Spawn** - constructs a running process from scratch
- **Fork / Exec** - creates a copy of the calling process or replaces the current program with another in the same process

**Spawn**

- Create and initialize the process control block (PCB) in the kernel
- Create and initialize a new address space
- Load the program into the address space
- Copy arguments into memory in the address space
- Initialize the hardware context to start execution at "start"
- Inform the scheduler that the new process is ready to run

Spawn is very complex, we have to specify everything about the new environment. If we omit a key argument a new process might have insufficient rights or resources or it might fail to function due to a security fault.

**Fork**
Fork on the other hand is less complex. The child process is almost an exact copy of the parent, with a different PID. We know which process we are in from the return value of the *fork()* call (0 for child, $> 0$ for parent, $< 0$ for error). The complete UNIX process management API also includes:

- *exec()* - system call to change the program being run by the current process
- *wait()* - system call to wait for a process to finish
- *signal()* - system call to send a notification to another process

In contrast to *spawn()*, here the child revokes rights and access explicitly before *exec()*, further we can use the full kernel API to customize the execution environment.

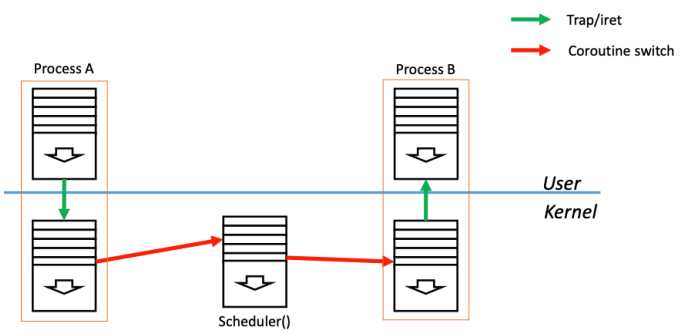
## 4.2 The Process Control Block

The PCB is the main kernel data structure used to represent a process. It has to hold or refer to the page table, trap frame, kernel stack, open files, program name, scheduling state, PID, etc.

## 4.3 Process Context Switching

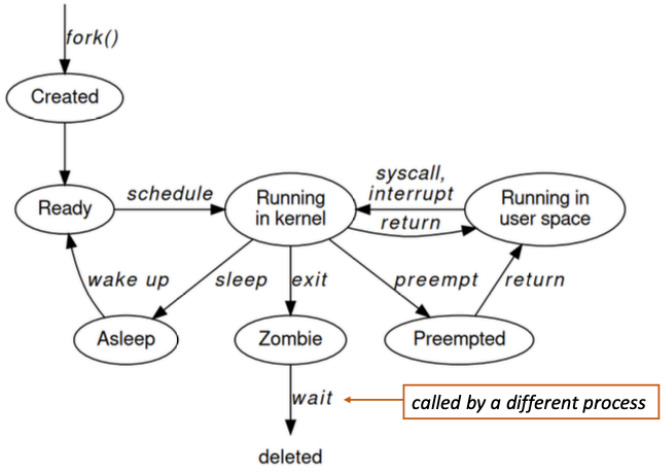
Context switching is the process of switching between different processes running in user mode or kernel mode. It is one of the key elements of the illusion that multiple programs can run in parallel.

There are two main reasons for the kernel to switch processes: either when a process has run for too long and gets interrupted by a hardware timer or when a process blocks. The second case happens when a system call can not complete immediately. The process then often calls *sleep()* and other processes can be executed.




### 4.4 Process Hierarchy

By forking and spawning new processes we create sort of a hierarchy. If a child process dies, but the parent does not call *wait()*, the child process becomes a zombie - it is dead, but still around since nobody asked for the return code. If a parent dies, but the child does not, the child becomes an orphan and gets reparented to the first process (PID #1, *init*).




The *init* process is basically an infinite loop calling *wait*(NULL), it gets rid of any zombies.

### 4.5 Threads

Threads are used as an abstraction for concurrency. They allow the usage of parallel hardware, e.g. multiple cores. A thread basically consist of a stack and some register values.

There is a distinction between **user threads** and **kernel threads**. The former are implemented entirely in users process. If a user thread is about to block (e.g. when executing a system call), the thread library usually interrupts this call and does something non blocking instead while the system call is served. Kernel threads are implemented by the OS kernel directly. They thus appear as different virtual processors to the user process. In this model, a set of threads share a virtual address space together. Each thread is now scheduled by the kernel itself, which keeps track of threads are part of which process. However, this makes the kernel more complicated.

## 5 Inter-Process Communication

It is often the case, that we want different processes to work together, e.g. DB and web-scraper. For this we need ways to exchange information between processes, we call this inter-process communication or IPC.

One of the most basic ways to exchange information is system calls. We save our data on the stack or in a register and execute the system call, the kernel then switches execution to the other process with the information about the data that should be exchanged. This is a really flawed approach some reasons are that context switches are expensive and it is unsafe to introduce new system calls for each task.

### 5.1 Shared Memory

We introduce shared memory that can only be accessed by the processes that communicate with each other. This requires us to define a common interface between the processes. The main building blocks for this are:

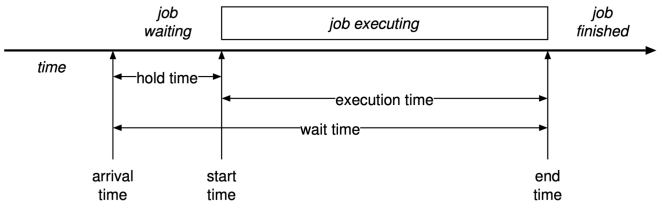
- Shared area: registers, memory - define the layout of the memory
- Indicating status: shared variables, signals
- Updating status: changing variables
- Consistency: use synchronization primitives

When checking the state of a shared variable polling can be very inefficient. As an alternative we can use a signal handler. When the first process calls the signal handler, the kernel notifies the second process of the update. When the kernel issues such a system call to a process, we call it an **upcall**. To guarantee consistency, we use synchronization primitives that are already known from previous lectures (spin locks with CAS, TAS, etc.).

A more modern approach is to use transactional memory, hereby we work with transactions that can fail on race condition.

## 6 Scheduling

Scheduling is the problem of deciding which process/thread on every core in the system is currently executing. This decision must be dynamically, not static. First we need to introduce some terminology. Notice how the wait time is defined as the combination of hold time and execution time.




The important metrics for scheduling are throughput (rate of completing jobs) and overhead (time spent without a job executing).

In the following part we will look at various scheduling algorithms, starting from a very simple approach.

### 6.1 Non-Preemptive Batch Oriented

There are two basic algorithms:

- First Come First Served
- Shortest Job First

Both are algorithms are very simple to implement but their applications are limited.

## 6.2 Preemptive Batch Oriented

We introduce preemption, meaning that we interrupt the execution after some finite time interval. After such an interrupt we decide which program to run next.

**Modified SJF:** Go through the sorted list of jobs and execute each until interrupted. Note that the length of a job does not get recomputed after an execution.

## 6.3 Interactive Scheduling

When running interactive workload, we can encounter events that block the execution (I/O, page-faults, etc.). During this time we would like to run another program instead of wasting execution time.

**Round Robin Scheduling:** Let $R$ be a queue and $q$ be the scheduling quantum:

1. Set an interval timer for an interrupt $q$ seconds in the future
2. Dispatch the job at the head of $R$
3. If blocking happens or the timer runs out, return to the scheduler
4. Push the previously running job to the tail of $R$

**Priority Based Scheduling:** We assign a priority to each task and dispatch the highest priority task first, if we have ties we use RR to break them. To avoid starvation we might want to use dynamic priorities (increased priority depending on wait time). An even more complex approach is to introduce multi-level queues, meaning that we have a fixed number of queues that assign priorities within each queue. Then we use RR scheduling between the queues to execute tasks.

Priority scheduling runs into problems when a high priority process has to wait for a lock from a low priority process (Priority Inversion). To fix this, we can either introduce inheritance - the holder of the lock acquires the priority of the highest waiting process - or ceiling - the holder of the lock runs at the highest priority.

## 6.4 Linux o(1) Scheduler

Linux uses 140 multilevel feedback queues, each with a different priority. Multilevel feedback queues penalize CPU bound tasks and prioritize I/O operations, as I/O tasks will eventually block.

The priority range 0-99 is used for high priority, static tasks and it uses FCFS or RR scheduling. The range 100-139 is for user tasks and uses RR together with priority ageing for I/O tasks.

# 7 Input / Output

Every OS has an I/O subsystem, which handles all interaction between the machine and the outside world. The I/O subsystem abstracts individual hardware devices to present a more or less uniform interface, provides a way to name I/O devices, schedules I/O operations and integrates them with the rest of the system, and contains the low-level code to interface with individual hardware devices.

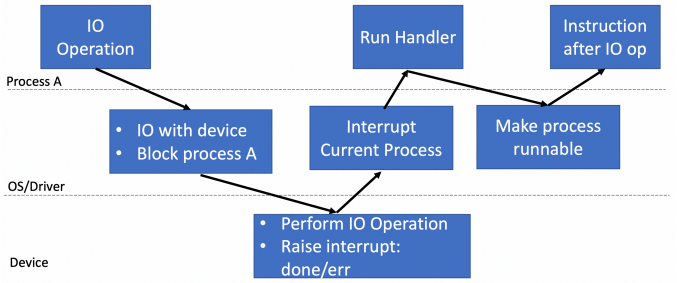
To an OS programmer, a **device** is a piece of hardware visible from software. It typically occupies some location on a bus or I/O interconnect, and exposes a set of hardware registers which are either memory mapped or in I/O space. A device is also usually a source of interrupts, and may initiate Direct Memory Access (DMA) transfers.

The **device driver** for a particular device is the software in the OS which understands the specific register and descriptor formats, interrupt models, and internal state machines of a given device and abstracts this to the rest of the OS.

## 7.1 Data Transfer

**Programmed I/O** consists of causing input/output to occur by reading/writing data values to hardware registers. This is the simples form of communication. It is fully synchronous, so the CPU always has to be involved. Further it is polled, the device has no way to signal that new data is ready.

**Interrupts** can be used to signal the availability of new data and solve the polling problem. But the problem of CPU involvement still persists.




**Direct Memory Access** or DMA, a device can be given a pointer to buffers in main memory and transfer data to and from those buffers without further involvement from the CPU. A single interrupt is used to signal the end of data transfer. DMA is, for the most part, physical (not virtual) access to memory. Further DMA transfers to and from main memory may or may not be coherent with processor caches.

## 7.2 Asynchrony

Device drivers have to deal with the fundamentally asynchronous nature of I/O: the system must respond to unexpected I/O events, or to events which it knows are going to happen, but not when.

The **First-level Interrupt Service Routine** (FLISR) is the code that executes immediately as a result of the interrupt. It runs regardless of what else is happening in the kernel. As a result, it can't change much since the normal kernel invariants might not hold.

Since I/O is for the most part interrupt-driven, but data is transferred to and from processes which perform explicit operations to send and receive it. Consequently, data must be buffered between the process and the interrupt handler, and the two must somehow rendezvous to exchange data. There are three canonical solutions to this problem:

A **deferred procedure call**, is a program closure created by the 1st-level interrupt handler. It is run later by any convenient process, typically just before the kernel is exited.

A **driver thread**, sometimes called an interrupt handler thread, serves as an intermediary between ISR and processes. The thread starts blocked waiting for a signal either from the user process or the ISR. When an interrupt occurs

or a user process issues a request, the thread is unblocked (this operation can be done inside an ISR) and it performs whatever I/O processing is necessary before going back to sleep. Driver threads are heavyweight: even if they only run in the kernel, the still require a stack and a context switch to and from them to perform any I/O requests.

The third alternative, is to have the FLISR convert the interrupt into a message to be sent to the driver process. This is conceptually similar to a DPC, but is even simpler: it simply directs the process to look at the device. However, it does require the FLISR to synthesize an IPC message, which might be expensive. In non-preemptive kernels which only process exceptions serially, however, this is not a problem, since the kernel does not need locks.

**Bottom-half handler** - the part of a device driver code which executes either in the interrupt context or as a result of the interrupt.

**Top-half handler** - the part of a device driver which is called "from above", i.e. from user or OS processes.

## 7.3 Device Models

The device model of an OS is the set of key abstractions that define how devices are represented to the rest of the system by their individual drivers. It includes the basic API to a device driver, but goes beyond this: it encompasses how devices are named throughout the system, and how their interconnections are represented as well.

**UNIX device model:**

- Character Devices - used for unstructured I/O and present a byte-stream interface with no block boundaries.
- Block Devices - used for structured I/O and deal with blocks of data at a time.
- Network Devices - correspond to a network interface adapter. It is accessed through a rather different API.
- Pseudo Devices - a software service provided by the OS kernel which it is convenient to abstract as a device, even though it does not correspond a physical piece of hardware (e.g. */dev/random*).

## 7.4 Protection

Another function of the I/O subsystem is to perform protection. Ensuring that only authorized processes can access devices or services offered by the device driver and that a device can't be configured to do something harmful.

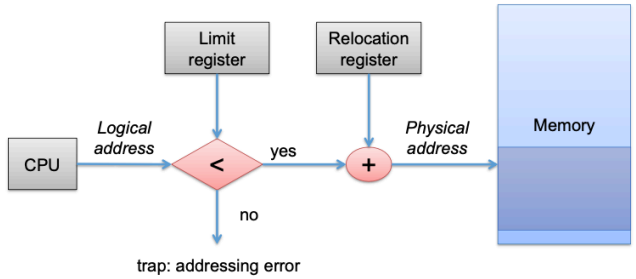
UNIX controls access to the drivers themselves by representing them as files, and thereby leveraging the protection model of the file system. DMA-capable devices are in principle capable of writing to physical memory anywhere in the system, and so it is important to check any addresses passed to them by the device driver. Even if you trust the driver, it has to make sure that its not going to ask the device to DMA somewhere it should not. One approach is to put a memory management unit (MMU) on the path between the device and main memory, in addition to each core having one.

# 8 Virtual Memory

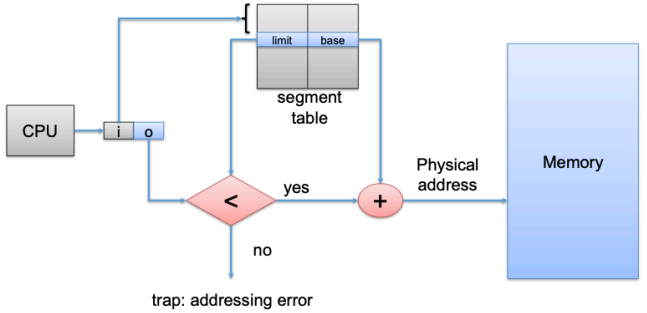
From previous courses we know MMUs, TLBs and basic paged virtual memory operations. The uses for address translation include: Process Isolation, Shared Code Segments, Dynamic Memory Allocation and many more.

## 8.1 Segments

Before paging, there were segments. Before segments there where base and limit registers. These contained two addresses $B$ and $L$. A CPU access to an address $a$ is permitted iff $B \leq a < L$. Relocation registers are an enhanced form of base register. All CPU accesses are relocated by adding the offset: a CPU access to an address $a$ is translated to $B + a$. This allows each program to be compiled to run at the same address (e.g. 0x0000).



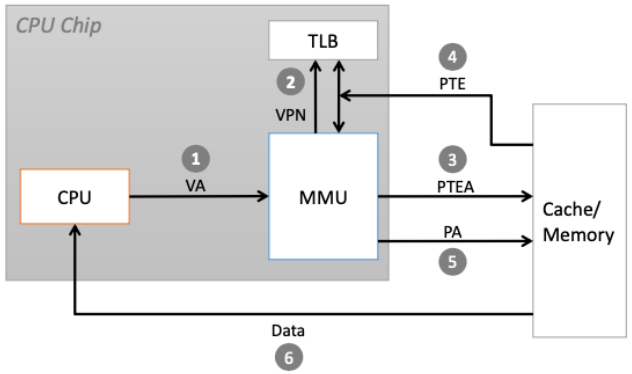

A segment is a triple $(I, B_I, L_I)$ of values specifying a contiguous region of memory address space with base $B_I$, limit $L_I$, and an associated segment identifier $I$ which names the segment. Memory in a segmented system uses a form of logical addressing: each address is a pair $(I, O)$ of segment identifier and offset. A Segment Table is an in-memory array of base and limit values $(B_I, L_I)$ indexed by segment identifier, and possibly with additional protection information.



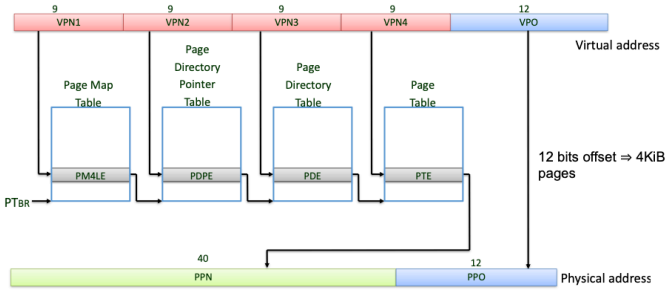

This enables sharing code/data segments between processes, further it adds protection and allows transparently growing stack/heap as needed. The principal downside of segmentation is that segments are still contiguous in physical memory, which leads to external fragmentation.

## 8.2 Paging

This is a short recap of paging. Virtual memory is divided into (virtual) pages of the same size having a VPN, physical memory gets divided into frames / physical pages having a PFN / PPN. Then a page table gets used to map VPNs to PFNs. This is implemented in hardware as the MMU. To speed up translation the TLB is used. Getting data from memory can work as follows (in this case we have a TLB miss):



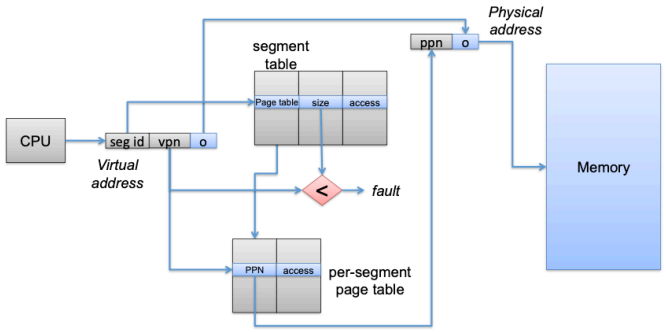

We have also seen how multi-level page tables work. Multi-level translation allows us to allocate only page table entries that are in use and makes memory allocation simpler.




On a context switch we need to store/restore the pointer to the page table and its size. One of the downsides of paging is if our page is very small and we do not need all of the space, this leads to internal fragmentation.

## 8.3 Paged Segmentation

It is possible to combine segmentation and paging. A paged segmentation memory management scheme is one where memory is addressed by a pair (segment.id, offset), as in a segmentation scheme, but each segment is itself composed of fixed-size pages whose page numbers are then translated to physical page numbers by a paged MMU.




One of the main benefits here is that each segment can have its own size of page table.

## 8.4 Zero-on Reference

The question how much physical memory is needed, is difficult to answer. So when a program runs out of memory, the system does the following:

1. Page fault into OS kernel
2. Kernel allocates some memory
3. Zeroes the memory
4. Modify page table
5. Resume process

## 8.5 Fill On Demand

Most programs do not need all their code to start running and might never use some code in its execution. Therefore we want to do something similar, we want to start a program before its code is in physical memory:

1. Set all page table entries to invalid
2. On first page reference, kernel trap
3. Kernel brings page in from disk
4. Resume execution
5. Remaining pages can be transferred in the background while the program is running

## 8.6 Copy On Write

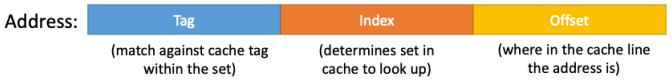
Remember that we said *fork()* copies the entire address space. This can be expensive and might not feasible in performance. On *fork()* we copy the page table and set all mappings to read-only in both address spaces. Reads are now possible for both processes. If a write in either process causes a protection fault the kernel allocates a new frame and copies the referenced frames content into it. The faulting process now maps to the new copy and the protection changes to read/write (also for the non-faulting process).

## 8.7 Managing Caches and the TLB

### 8.7.1 TLBs

A problem with TLBs on context switches is that they can hold content inaccessible to the new process. To avoid having to flush the TLB, we introduce tags. Each TLB entry has a 6-bit tag and the OS keeps track of a mapping between processes and tags.

### 8.7.2 Caches




Remember the different types of caches:

- Virtually indexed, virtually tagged - simple and fast, but context switches are hard
- Physically indexed, physically tagged - can only be accessed after address translation
- Virtually indexed, physically tagged - overlap cache and TLB lookups
- Physically indexed, virtually tagged

Also remember the different write (write through, write back) and allocate (write allocate, non-write allocate) policies. In virtually tagged caches we can encounter homonyms, the same virtual address maps to multiple physical address spaces. To avoid this we can use physical tags, add address space identifiers, try to ensure disjoint address spaces or flush the cache on a context switch.

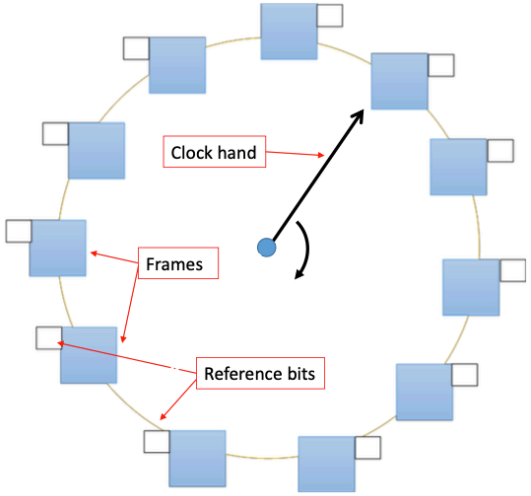
There is also a synonyms problem, where two virtual addresses map to the same physical address. This leads to inconsistent cache entries. The solutions to the homonym problem do not help here. To solve this problem we restrict VM mappings, so that synonyms map to the same cache set.

## 8.8 Demand Paging

Demand paging solves the problem how to find a frame to use for missing pages. The goal is to minimize the page fault rate $p$ ($0 \leq p \leq 1$). The metric we are interested in is the effective access time $(1-p) * l_m + p * l_f$, where $l_m$ is the latency of a memory access and $l_p$ the latency of a page fault handling. Generally speaking the performance of a paging system depends on how many frames it has, but there is a diminishing return on adding new frames.

### 8.8.1 Page Replacement Policies

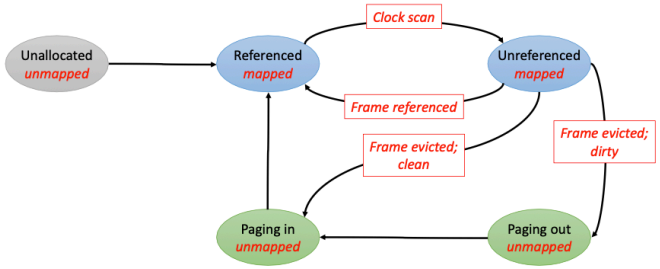
If a page has to be evicted, which one should be choose? We have already seen Least Recently Used (LRU) and First In First Out (FIFO). Both of these are not optimal, LRU is too expensive and FIFA works poorly for most workloads. A good compromise is the **Clock** or **2nd Chance** algorithm. It approximates LRU but much cheaper. It requires a linear table of all PFNs, with associated referenced bits. This can be best visualized as a circular buffer.




It works as follows:

- Mark each frame when referenced for the first time
- If we want to replace a frame, process as follows:
  - If marked, unmark and advance the clock hand
  - If unmarked, allocate this frame and mark it

### 8.8.2 Frame States




This only works if the corresponding bits are present in the page table and are properly set, in RISC-V for example these bits are present but do not have to be set. We can emulate these bits using faults.

# 9 File System

## 9.1 Abstractions

File systems are a very important concept and it makes sense to define an abstract file system. The main task of a file system is to virtualize, this means it should allow for multiplexing, have the same abstraction for multiple file systems implementation.

### 9.1.1 Access Control

We want to enforce some access control for our files. We call user or group of users a **principal** and files **objects**. One of the simplest way of representing this information would be in a matrix format. Each row would be a principal and columns correspond to objects. A entry in this matrix describes the rights the principal has in regards to the object. The problem with this idea is that such a matrix would become too large.

Another idea would be access control lists, a more compact representation. For each file you need to specify user and its access right, so you do not have to save any information on principals that do not have any rights to a file. If a new file gets created we use mandatory access control (MAC) to set the rights to a file. MAC defines the access policy centrally, so the principals cannot make policy decisions. This is easy to implement but hard to customize if you need special permissions.

A third idea would be to use capabilities, this means that there are tokens that give the holder of the token specific rights.

POSIX access control uses a hybrid approach. As principals there exist users but also groups. Access rights are divided into read, write and execute ($rwx$).

### 9.1.2 File Abstraction

Files consist of two parts. The first part is the **data**, consisting of unstructured bytes or blocks. The second part is the **metadata**, containing informations like type (structures or unstructured), time stamps, location, permissions, etc. The filename is not part of its metadata.

The filename is part of the **namespace**. Depending on the OS there are different restrictions on the filename, e.g. allowed characters. Filenames can be thought of as pointers to a file, meaning a file does not need a name or it could even have multiple names. The POSIX namespace results in a tree like structure, where each filename is preceded by the directories it is contained in. There are other location based bindings '.' meaning the current location and '..' being the parent directory. The challenge with this tree structure is to prevent cycles.

### 9.1.3 File Descriptors

**File descriptors** are used to open files and then perform operations on it. A file descriptor is basically an id that the OS give to a process to access a file (a FD comes with some metadata, e.g. type of access). Each processor has its own file descriptors. This concept is similar to capabilities.

There are different access types:

- Direct - unrestricted access to the file without offset, curser starts at zero

- Sequential - access without rights to move the curser, writing happens at the end of the file
- Structured - defined agreement on how data gets written

### 9.1.4 Memory-Mapped Files

Alternatively, we can open files by mapping its content to virtual memory. This allows for anonymous memory, meaning we can treat memory regions as files even though this is not the case. It also allows us to have shared memory, by loading the same file on different processes. But if we use memory-mapped files, we have to take care of synchronisation between the memory and the file system.

### 9.1.5 Executable Files

Executing a file creates a new process, checks if the file is valid and then uses the ELF format to load the file and start execution.

## 9.2 Implementations

After having specified how an interface for a file system might look like, we know want to have a look at the implementations.

First we introduce an additional abstraction, **volumes**. A volume is a generic name for a storage device, consisting of a contiguous set of fixed-size blocks. To access blocks, we need **logical block addresses** (LBA), these are numbers for blocks on a volume. Closer to the hardware, there are **disk partitions**. Partitions are physical volumes divided into contiguous regions. For this to work we need to store a partition table at the start of the physical volume.

A file system implementation consists of a set of data structures. These are stored on a volume and allow for naming and protection. These things together form the **FS API**.

This API allows us to mount several file systems on top of each other. At the top we have the root file system and all other mount points are under the root (e.g. */dev/sda1*).

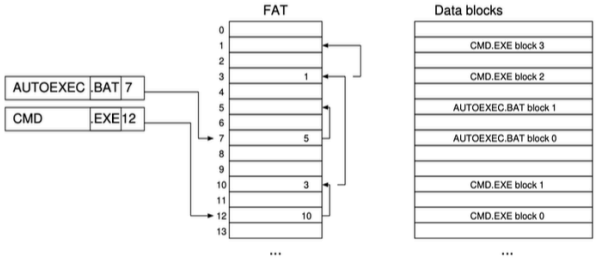
From the OS point of view, it implements a **virtual file system** layer in the kernel. This layer tries to resolve the type of file system that is being used. This allows for different file system implementations to coexist.

The main goals of concrete file system implementations depend on the device it is used on. Often they are a mixture of performance and reliability. In a typical file system implementation we will see the following things:

- Directories and Indexes - where on the disk is the data for each file?
- Index Granularity - what is the unit of allocation for files?
- Free Space Maps - how to allocate more sectors on the disk?
- Locality Optimizations - how to make it go fast in the common case?

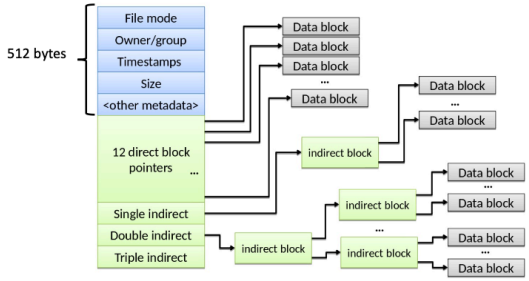
### 9.2.1 The FAT File System

FAT (file allocation table) is a very basic file system.




For naming, the FAT system uses the filename and the extension. It then remembers the first block of that file. The file allocation table itself works like a linked list of blocks, where at the end of each block there is a pointer to the next block. To allocate new space, we simply need to scan the table. This can lead to very poor locality (fragmentation).

### 9.2.2 The Berkeley Fast Filing System

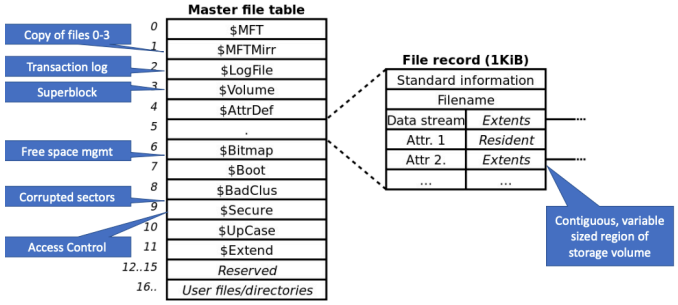



This system uses an index node (inode) for each file in the file system. The inode consists of the metadata and block pointers (or data directly if it fits). To allocate blocks it uses a bitmap indicating if a block is free or not.

Block groups are continuous subset of disk track where related inodes, directories, free space map and file data blocks are gathered together. A superblock then holds all the informations about the overall layout and where the block groups are.

### 9.2.3 Windows NTFS

NTFS treats everything as a file, e.g. files system and file metadata. The master file table contains file entries for each file and is itself a file with the very first entry in the table. A MFT entry consists of metadata and a list of variable length attributes. One attribute is all the names for a file. If the data is small enough, it gets stored as part of the attributes, else it is stored in an extent. If works very similar to Berkeley FFS.




In NFTS file descriptors are basically pointers to extents. Additionally there are basic file descriptors.

# 10 Network Stack

In this part we want to look at what happens in the OS when we do networking on a low level. The role of the OS network stack is to handle all network related I/O. This includes delivering and transmitting packets, multiplex and demultiplex packets, and processing protocols.

The following parts play a role in the networking stack:

- the NIC (networking interface card)
- first-level interrupt handlers

- NIC driver bottom half, e.g. deferred procedure calls
- NIC driver top half, kernel code
- app libraries, daemons, utils

Together they provide the following functionalities:

- Multiplexing - taking packets from the user space, deciding the protocol to use and sending them out
- Encapsulation - taking raw data from an application and encapsulate it in a packet
- Protocol State Processing - advance the state of a protocol to process packets arriving
- Buffering and Data Movement - store data until the application decides to process it

## 10.1 Header Space

The header space is the set of all possible packet headers. The OS needs to process these headers as fast as possible and deliver them to the applications. Part of processing the header is to decide if the header is valid.

## 10.2 Protocol Graphs

Some OS maintain a protocol graph. The protocol graph of a network stack is the directed-graph representation of the forwarding and multiplexing rules. Nodes in the graph represent a protocol acting on a communication channel and perform de-/encapsulation and possibly de-/multiplexing.

If a node has multiple outgoing edges, its probably demultiplexing packets (vice versa for multiplexing). Note that this graph can well be cyclic due to tunnelling.

## 10.3 Network I/O

If the NIC receives a packet it copies it to a OS buffer, enqueues it on a descriptor ring and returns the buffer to the OS. If the OS wants to send a packet it enqueues it on a descriptor ring, notifies the NIC and after processing the NIC returns the buffer to the OS. A simplified implementation of the first-level- interrupt handler for packet receive looks as follows:

**Algorithm 1:** First-level interrupt handler for receiving packets

```
  /* Inputs                                    */
  /* rxq:  the receiver description queue      */
1 Acknowledge interrupt
2 while not(rxq.empty()) do
3 |   buf = rxq.dequeue()
4 |   sk_buf = sk_buf_allocate(buf)
5 |   enqueue(sk_buf) for processing
6 |   post a DCP (software interrupt)
7 end
8 return
```

Note that this only copies the packet and does not process it. This allows the OS to free the space for the NIC to receive new packets with deferring the processing the packet to later.

## 10.4 Top-Half Handling

In the top half of the stack, a socket interface is used: we can call *bind(), listen(), connect(), send(), recv()* etc. from user space.

Some protocol processing happens in the kernel directly as a result of top half invocations, but for the most part the top half is concerned with copying network payload data and metadata between queues of protocol descriptors in the kernel and user-space buffers.

## 10.5 Polling

To increase performance, instead of the conventional interrupt-driven descriptor queues, a network receive queue can be serviced by a processor polling it contiguously. This eliminates the overhead of interrupts, context switches and maybe even kernel entry/exit. It requires a dedicated processor spinning, waiting for packets.

But even polling is insufficient to handle modern high-speed networks. Its not clear how to scale this approach to multiple cores. Thus one relies on hardware support.

## 10.6 Hardware Acceleration

Today many operations are accelerated using dedicated hardware, e.g. by smart NICs. This works by having multiple physical queues for each connection. The OS then can bind an application to a hardware queue when needed. These smart NICs have dedicated hardware to perform operations on these queues, e.g. calculate checksums or applying filtering rules.

**RDMA** devices allow for direct memory accesses between different devices. This is extremely fast and specially useful in a datacenter context. On a downside allowing a remote devices to access a devices memory can be dangerous.

## 10.7 Routing and Forwarding

An OS also implements the core functionality of routers (forwarding and routing). For forwarding a set of hardware tables is used. Forwarding a packet on a receive queue essentially involves reading its header, then using this information to transfer the packet to one or more transmit queues to be sent. The forwarding tables are a result of the routing calculations, which mostly happen in user space.

# 11 Virtual Machines

A **virtual machine monitor** (VMM) virtualizes an entire system. The execution environment of the VMs well look at provide a simulation of the raw machine hardware.

While a VMM is the functionality required to create the illusions of real hardware, the **hypervisor** is the software that runs on real, physical hardware and supports multiple VMs (each with its associated virtual machine monitor). There is one hypervisor on which many VMMs can run. We call a hypervisor running on bare metal a type-1 (native) hypervisor and one running on a real OS a type-2 (hosted) hypervisor.

OS-level virtualization uses a single OS to provide the illusion of multiple **containers** of that OS. Code running in a container have the same system call interface as the underlying OS, but cannot access any device. This is achieved by limiting the file system namespace (by changing the root for each container) and the process namespace, so processes can only see processes which share their container.

In general, using containers is more efficient than using hypervisors.

## 11.1 The Uses of Virtual Machines

When multiple applications contend for resources the performance of one or more may degrade in ways outside the control of the OS. **Resource isolation** guarantees to one application that its performance will not be impacted by others, this is done by running the application in a VM.

**Cloud computing** is the business of renting computing resources as a utility to paying customers rather than selling hardware. They are primarily based on renting computing resources in the form of a VM (similar to resource isolation).
The term **server consolidation** refers to taking a set of services, each running on a dedicated server, and consolidating them onto a single physical machine so that each runs in a VM.

**Backward compatibility** is the ability of a new machine to run programs (including OSes) written for an old machine.

## 11.2 Virtualizing the CPU

To run an OS inside a VM, we need to completely virtualize the processor, including the kernel (else we simply could use threads). The processor in the VM clearly cannot execute a privileged instruction "for real". Instead, the default result is a trap/fault:

*Trap-and-emulate* is a technique for virtualization which runs privileged code in non-privileged mode. Any privileged instruction causes a trap to the VMM, which then emulates the instruction and returns to the VM guest code.

The problem is that there might be some instructions which do not cause a trap when run in non-privileged mode but have a different behavior when executed in kernel mode (e.g. *POPF* in x86). This cannot happen with a strictly virtualizable ISA.

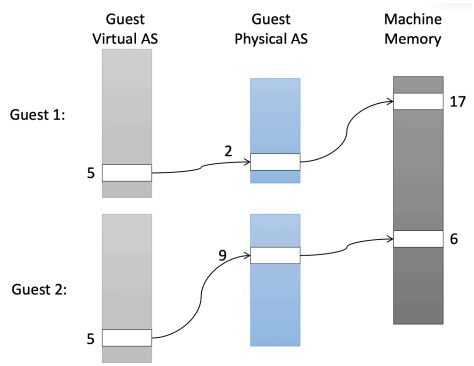
An ISA is **strictly virtualizable** iff it can be perfectly emulated over itself with all non-privileged instructions executed naively and all privileged instructions emulated via traps.

There are different approaches to dealing with non strictly virtualizable ISA:

- **Full software emulation**: Creates a virtual machine by interpreting all kernel-mode code in software. This is very slow, especially for many I/O operations.
- **Paravirtualization**: A paravirtualized guest OS is one which has been specifically modified to run inside a VM. Critical calls are replaced with explicit trap instructions.
- **Binary rewriting**: Scans compiled kernel code for unvirtualizable instructions and rewrites them essentially patching the kernel on the fly. This is done on demand: All kernel pages are first protected and when first accessed (i.e., the pro- tection trap occurs), they get scanned and rewritten.
- **Virtualization extensions**: Convert ISA by adding virtualization extensions. This typically takes the form of a new processor mode. Today, both ARM and x86 do have hardware support for virtualization.

## 11.3 Viratualizing the MMU

With virutalization, there is a second level of indirection with memory addresses. Now, a physical memory address is not unique in the machine, but in one VM (guest OS thinks it is physical). Thus, we define the **machine address** to be a real address on the machine which gets translated from the guest OSs physical address. From the view of the hypervisor, the machine address is the physical address.




The hypervisor thus needs to translate a guest virtual address not to a guest physical address but to a machine address instead. There are several ways to do so:

- Directly writable tables: The guest OS creates the page tables that the hardware uses to directly translate guest virtual to machine addresses. This requires paravirtualization. The VMM needs to check all writes to any PTE in the system. To change a PTBR, a hypercall is needed.
- A shadow page table is a page table maintained by the hypervisor which contains the result of translating virtual addresses through the guest OSs page tables, and then the VMMs physical-to-machine page table. The guest OS thus sets up its own PT, but they get never used. The VMM maintains the shadow PT which maps directly from guest VAs to machine addresses.

  The VMM must keep the shadow table consistent with both the guests PT and the hypervisors own physical-to-machine table. It achieves this by write-protecting all the guest OS's PT and trapping writes on them. When a write happens, it applies the update to the shadow page as well.
- Nested Paging/$2^{nd}$ level page translation is an enhancement to the MMU hardware that allows it to translate through two page tables (guest virtual to guest physical and guest physical to machine), caching the result (virtual to machine) in the TLB. This can be fast, but a TLB miss is costly.

## 11.4 Viratualizing the Physical Memory

How can the hypervisor allocate memory to a guest OS? The guest OS expects a fixed area of physical memory which does not change dynamically. In theory, this problem can be solved with paging. However, there is a phenomenon called **double paging**. Consider the following sequence of events:

1. The hypervisor pages out a guest page $P$ to storage
2. A guest OS decides to page out the virtual page associated with $P$ and touches it.
3. This triggers a page fault in the hypervisor, hence $P$ gets paged back in memory.

4. The page is immediately written out to disk and discarded by the guest OS.

So to throw away a page in a guest OS, there are three I/O operations and one extra page fault! We could solve this problem with paravirtualization, but this introduces more complexity.

**Memory ballooning** is an elegant solution to this problem. It allows hypervisors to reallocate machine memory between VMs without incurring the overhead of double paging. A device driver, the balloon driver, is installed in the guest kernel. This driver is VM-aware, i.e. it can make hypercalls and receive messages from the VMM. The principle is to block a large area of physical memory in the guest OS, which then can be allocated to the OS by unblocking it.

Memory can also be reclaimed from a guest OS (inflating the balloon):

1. The VMM asks the balloon driver to return $n$ physical pages from the guest OS to the hypervisor.
2. The balloon driver notifies the OS to allocate $n$ pages of memory for its private use.
3. It communicates the guest-physical addresses of these frames to the VMM using a hypercall.
4. The VMM unmaps these pages from the guest OS kernel and reallocates them elsewhere.

Reallocating machine memory to the VM (deflating the balloon) can be done similarly:

1. The VMM maps the newly allocate machine pages into guest-physical pages inside the balloon.
2. The VMM then notifies the balloon driver that these pages are now returned.
3. The balloon driver returns these guest-physical pages to the rest of the guest OS.

### 11.5 Viratualizing the Devices

To software, a device is something that the kernel communicates using memory mapped I/O registers, interrupts from the device to the CPU and DMA access by the device to/from main memory. The hypervisor needs to virtualize all of this, too.

A device model is a software model of a device that can be used to emulate a hardware device, using trap-and-emulate to catch CPU writes to device registers. Interrupts from the emulated device are simulated using upcalls from the hypervisor into the guest OS kernel at its interrupt vector.

A **paravirtualized device** is a hardware device design which only exists as an emulated device. The driver of the device in the guest OS is aware that it is running in a VM and can communicate efficiently with the hypervisor using shared memory buffers and hypercalls.

For the device drivers talking to the real devices, we have the option to put them in the hypervisor kernel. Alternatively, one could use device passthrough, mapping a real hardware device into the physical address space of a guest OS. However, this does not solve the problem of sharing a real device among multiple virtualized guest OSes.

A **driver domain** is a virtual machine whose purpose it is to provide drivers for devices using device passthrough. With this, we can share devices across multiple VMs, by exporting a different to these devices using inter-VM communication channels. They are great for compatibility, but can be very slow, due to the communication overhead.

A **self-virtualizing** device is a hardware device which is designed to be shared be- tween different VMs by having different parts of the device mapped into each VMs physical address space. **SR-IOV** is one form of this.

Single-Root I/O Virtualization (SR-IOV) is an extension to the PCIe standard which is designed to give VMs fast, direct but safe access to real hardware. An SR-IOV capable device appears initially as a single PCI device. This device can be configured to make further virtual functions appear in the PCI device space: each of this is a restricted version, but otherwise looks like a completely different, new device.

### 11.6 Viratualizing the Network

A soft switch is a network switch implementation inside a hypervisor which switches network packets sent from paravirtualized network interfaces in VMs to other VMs and/or one or more physical network interfaces.

The soft switch can be quite powerful but it needs to be fast. We can address a network interface inside a VM by giving each virtual network interface a MAC address on its own and letting DHCP do the rest.

## Distributed Systems

Today almost all computer systems are distributed, for different reasons:

- Geography
- Parallelism - speed up computation
- Reliability - prevent data loss
- Availability - allow for access at any time, without bottlenecks, minimizing latency

Even though distributed systems have many benefits, such as increased storage or computational power, they also introduce challenging coordination problems.

# 12 Fault Tolerance and Paxos

In this section we want to create a fault-tolerant distributed system. We start out with a simple approach and improve our solution until we arrive at a system that works even under adverse circumstance, Paxos.

A **node** is a single actor in the system. In the message passing model we study distributed systems that consist of a set of nodes, where each node can perform local computations and send messages to every other node. Message loss means that there is no guarantee that a message will arrive safely at the receiver. This leads us to the first algorithm

**Algorithm 2:** Naive Client-Server Algorithm

```
1 Client sends commands one at a time to server
2 Server acknowledges every command
3 If the client does not receive an acknowledgment
    within a reasonable time, it resends the command
```

This simple algorithm is the basis of many reliable protocols, e.g. TCP. The algorithm can easily be extended to work with multiple servers: The client sends each command to every server, and once the client received an acknowledgment from each server, the command is considered to be executed successfully.

In practice, messages might experience different transmission times, even if they are being sent between the same two nodes. A set of nodes achieves **state replication**, if all nodes execute a sequence of commands in the same order. Since state replication is trivial with a single server, we can desig- nate a single server as a serializer.

**Algorithm 3:** State Replication with a Serializer

1 Client sends commands one at a time to the serializer
2 Serializer forwards commands one at a time to all other servers
3 Once the serializer received all acknowledgments, it notifies the client about the success

The downside of this algorithm is that the serializer is a single point of failure.

## 12.1 Two-Phase Protocol

**Algorithm 4:** Two-Phase Protocol

```
   /* Phase 1                                          */
1  Client asks all servers for the lock
   /* Phase 2                                          */
2  if client receives lock from every server then
3  |  Client sends command reliably to each server
   |  and gives the lock back
4  else
5  |  Clients gives the received locks back
6  |  Client waits, and then starts with Phase 1
   |  again
7  end
```

Instead of directly establishing a consistent order of commands, we can use a different approach: We make sure that there is always at most one client sending a command; i.e., we use mutual exclusion, respectively locking.

Still there are quite some problems with this algorithm. What happens if the node holding the locks crashes?

## 12.2 Paxos

A **ticket** is a weaker form of a lock, with the following properties:

- Reissuable: A server can issue a ticket, even if previously issued tickets have not yet been returned.
- Ticket expiration: If a client sends a message to a server using a previously acquired ticket $t$, the server will only accept $t$, if it is the most recently issued ticket.

There is no more problem with crashes: If a client crashes while holding a ticket, the remaining clients are not affected. (At this point the naive ticket protocol is left out)

**Algorithm 5:** Paxos Client / Proposer

```
   /* Initialization                                   */
1  c                          /* command to execute  */
2  t = 0                      /* ticket number to try */
   /* Phase 1                                          */
3  t = t + 1
4  Ask all servers for ticket t
   /* Phase 2                                          */
5  if a majority answers ok then
6  |  Pick(T_store, C) with largest T_store
7  |  if T_store > 0 then
8  |  |  c = C
9  |  end
10 |  Send propose(t, c) to same majority
11 end
   /* Phase 3                                          */
12 if a majority answers success then
13 |  Send execute(c) to every server
14 end
```

**Algorithm 6:** Paxos Server / Acceptor

```
   /* Initialization                                   */
1  T_max = 0              /* largest issued ticket    */
2  C = ⊥                  /* stored command           */
3  T_store = 0            /* ticket used to store C   */
   /* Phase 1                                          */
4  if t > T_max then
5  |  T_max = t
6  |  Answer with ok(T_max, C)
7  end
   /* Phase 2                                          */
8  if t = T_max then
9  |  C = c
10 |  T_store = t
11 |  Answer success
12 end
```

Unlike previously mentioned algorithms, there is no step where a client explicitly decides to start a new attempt and jumps back to Phase 1. This has the advantage that we do not need to be careful about selecting good values for timeouts, as correctness is independent of the decisions when to start new attempts. The performance can be improved by letting the servers send negative replies in Phase 1 or 2 if the ticket expired. Using randomized backoff we can eliminate contention between clients.

**Theorem:** If a command c is executed by some servers, all servers (eventually) execute c.

Note that Paxos cannot make progress if half (or more) of the servers crash, as clients cannot achieve a majority anymore.

For state replication we need to be able to execute multiple commands, we can extend each instance with an instance number, that is sent around with every message. Once the 1st command is chosen, any client can decide to start a new instance and compete for the 2nd command. If a server did not realize that the 1st instance already came to a decision, the server can ask other servers about the decisions to catch up.

# 13 Consensus

## 13.1 Two Friends

A protocol used in a network with unreliable connections that relies on ACKing might not terminate: If $A$ sends to $B$ and expects an ACK, $B$ also needs $A$ to ACK the ACK message and so on. This cannot terminate.

## 13.2 Consensus

There are $n$ nodes, $f$ of which might crash. $n - f$ nodes are correct. Each node $i$ starts with an input $v_i$. All nodes must decide on one of those values, satisfying the following:

- **Agreement**: all correct nodes decide for the same value
- **Termination**: all correct nodes terminate in finite time
- **Validity**: the decision value must be the input value of some node

We assume that the links are reliable and that each node can send to each other node. However there is no broadcast, a node can only send individual messages. If we study Paxos carefully, we will notice that **Paxos does not guarantee termination**.

## 13.3 Impossibility of Consensus

We restrict the input values to be either 0 or 1. Even with this simplification, there is no algorithm which solves the consensus problem.

In the **asynchronous model**, algorithms are event based (upon receiving msg, do ...). Nodes cannot access a synchronous clock. A message from a node to another will arrive in finite but unbounded time. This is a formalization of the variable message delay model.

For algorithms in the asynchronous model, the **runtime** is the number of time units from the start of the execution to its completion in the worst case (assuming a delay of at most one time unit).

We say that a system is **fully defined** (at any point during the execution) by its configuration $C$. The configuration includes the state of every node, and all messages that are in transit. A configuration if **univalent**, if the decision value is determined independently of what happens afterwards, else we call it **bivalent**. We might also call a configuration that is univalent for a value $v$ $v$-valent.

**Lemma:** There is at least one selection of input values $V$ such that the according initial configuration $C_0$ is bivalent, if $f > 1$. (We are gonna omit the proofs)

A **transition** from configuration $C$ to a following configuration $C_\tau$ is characterized by an event $\tau = (u, m)$, where node $u$ receives a message $m$.

The **configuration tree** is a directed tree of configurations. Its root is $C_0$ which is fully characterized by the input values $V$. The edges of the tree are transitions; every configuration has all applicable transitions as outgoing edges. Leaves are terminal configurations.

**Lemma:** Assume two transitions $\tau_1, \tau_2$ for $u_1 \neq u_2$ which are both applicable to $C$. It holds that $C_{\tau_1\tau_2} = C_{\tau_2\tau_1}$.

A configuration is **critical** if $C$ is bivalent, but all configurations that are direct children of $C$ are univalent. Informal we can say that $C$ is the last moment in the execution where the decision is not yet taken.

**Lemma:** If a system is in a bivalent configuration, it must reach a critical configuration within finite time, or it does not always solve consensus.

**Lemma:** If a configuration tree contains a critical configuration, crashing a single node can create a bivalent leaf.

From these lemmas we derive that there is no deterministic algorithm which always achieves consensus in the asynchronous model, while $f > 0$. If $f = 0$ all nodes can simply send their value to all other nodes and then choose the minimum.

## 13.4 Randomized Consensus

As there exists no deterministic solution, we use randomness. In the following we assume $f < n/2$.

The basic idea of a possible algorithm is that each node chooses $v_i$ at random until a majority gets the same resulting value.

**Algorithm 7:** Randomized Consensus (Ben-Or)

```
1  v_i ∈ [0,1]                                   /* input bit */
2  round = 1
3  decided = false
4  Broadcast myValue(v_i, round)
5  while true do
       /* Propose                                              */
6      Wait until a majority of myValue messages of
         the current round arrived
7      if all messages have the same value v then
8          Broadcast propose(v, round)
9      else
10         Broadcast propose(⊥, round)
11     end
12     if decided then
13         Broadcast myValue (v_i, round+1)
14         Decide for v_i and terminate
15     end
       /* Vote                                                 */
16     Wait until a majority of propose messages of
         current round arrived
17     if all message propose the same value v then
18         v_i = v
19         decided = true
20     else
21         if there is at least on proposal for v then
22             v_i = v
23         else
24             Choose v_i randomly, with Pr[v_i = 0] =
                 Pr[v_i = 1] = 1/2
25         end
26     end
27     round += 1
28     Broadcast myValue(v_i, round)
29 end
```

As long as no node decides and terminates, the algorithm will never get stuck, independent of which nodes crash. Further the algorithm satisfies the validity, the agreement and the termination requirement (expected $\mathcal{O}(2^n)$).

There is no consensus algorithm for the asynchronous model that tolerates $f \geq n/2$.

We have seen that the algorithm solves consensus with optimal fault-tolerance but it is awfully slow. The reason for this is the individual coin tossing. We can improve this by tossing a so-called shared coin.

### 13.5 Shared Coin

Instead of picking each value with probability 1/2, we use the shared coin to introduce randomness.

**Algorithm 8:** Shared Coin

```
1  Choose local coin c_u = 0 with probability 1/n, else
     c_u = 1
2  Broadcast myCoin(c_u)
3  Wait for n − f coins and store them in a local coin
     set C_u
4  Broadcast myCoinSet(C_u)
5  Wait for n − f coin sets
6  if at least one coin is 0 among all coins in the coin
     sets then
7  |  return 0
8  else
9  |  return 1
10 end
```

If $f < n/3$, the algorithm implements a shared coin. Plugging this into our randomized algorithm, we get a consensus algorithm which terminates in a constant expected number of rounds tolerating up to $f < n/3$ crash failures.

## 14 Byzantin Agreement

In some sort, byzantine agreement is the generalization of consensus. A node which can have arbitrary behavior is called byzantine. This includes sending wrong messages, different messages to different neighbors lying about input values and crashing.

Fining consensus in a system with byzantine nodes is called **byzantine agreement**. An algorithm is $f$-resilient if it still works correctly with $f$ byzantine nodes. If a algorithms solves the byzantine agreement, it solves consensus, too.

For an byzantine agreement, we need agreement, termination and validity. While the first two aspects are defined as before, validity is not that straight-forward.

### 14.1 Validity

There are four possible definitions for validity:

- **Any-Input Validity** - The decision value must be the input of any node. This does not make sense for byzantine nodes, as they can lie about their inputs.
- **Correct-Input Validity** - The decision value must be the input of a correct node.
- **All-Same Validity** - If all nodes start with the same input value, the decision value must be this input value.
- **Median Validity** - If the input values are orderable, byzantine outliers can be prevented by agreeing on a value close to the median of the correct input values. How close depends on the number of byzantine nodes $f$.

First, we will only look at the synchronous model, where nodes operate in synchronous rounds. In each round, each node may send a message, receive messages, and do some local computation. For algorithms in the synchronous model, the runtime is simply the number of rounds from the start of the execution to its completion in the worst case.

### 14.2 How Many Byzantine Nodes?

**Algorithm 9:** Byzantine Agreement with $f = 1$

```
1  Code for node u, with input value x
   /* Round 1                                          */
2  Send tuple(u, x) to all other nodes
3  Receive tuple(v, y) from all other nodes
4  Store all received tuples in a set S_u
   /* Round 2                                          */
5  Send set S_u to all other nodes
6  Receive sets S_v from all nodes v
7  T = set of tuples seen in at least two sets S_v,
     including own S_u
8  Let y be the smallest value in T
9  Decide on value y
```

If a byzantine node does not follow the protocol, it can be easily detected and its messages discarded. However, if it sends syntactically correct messages, bad things might happen, e.g. if a byzantine node sends different values to different nodes in the first round.

If $n \geq 4$, all nodes have the same set $T$. Thus, each one will see every correct value twice. So all correct values are in $T$. Based on this insight, we can show that this algorithm solves byzantine agreement for $n \geq 4$.

Further we can show that three nodes cannot reach byzantine agreement and that a larger network cannot reach byzantine agreement for $f \geq n/3$ byzantine nodes.

### 14.3 The King Algorithm

**Algorithm 10:** King Algorithm for $f < n/3$

```
1  x = my input value
2  for phase = 1 to f + 1 do
   |  /* Vote                                          */
3  |  Broadcast value(x)
   |  /* Propose                                       */
4  |  if value(y) received at least n  f times then
5  |  |  Broadcast propose(y)
6  |  end
7  |  if propose(z) received more than f times then
8  |  |  x = z
9  |  end
   |  /* King                                          */
10 |  Let node v_i be the predefined king of phase i
11 |  The king v_i broadcasts its current value w
12 |  if received strictly less than n - f propose(y)
   |   then
13 |  |  x = w
14 |  end
15 end
```

This algorithm fulfils the all-same validity. Further if a correct node proposes $x$, no other correct nodes proposes $y \neq x$, if $n > 3f$.

**Lemma:** There will be at least one phase with a correct king.

Also, after a round with a correct king, the correct nodes will not change their values anymore. Using all this facts, its easy to show that this algorithm solves the byzantine agreement. However, the algorithm needs $f+1$ predefined kings. If they are not given beforehand, finding those kings is a byzantine agreement task by itself, so this must be done before the King algorithm.

A synchronous algorithm solving consensus in the presence of $f$ crashing nodes needs at least $f+1$ rounds, if the nodes decide for the minimum value. Since byzantine nodes can also just crash, this lower bound also holds for byzantine agreement, so our algorithm has an asymptotically optimal runtime.

### 14.4 Asynchronous Byzantine Agreement

What if the nodes only work in a asynchronous manner?

**Algorithm 11:** Asynchronous Byzantine Agreement (Ben-Or for $f < n/10$)

```
1  x_u ∈ {0,1}
2  round = 1
3  while true do
4      Broadcast propose(x_u, round)
5      Wait until n − f propose messages of current
         round arrived
6      if at least n/2 + 3f + 1 propose messages
         contain same value x then
7          Broadcast propose(x, round + 1)
8          Decide for x and terminate
9      else
10         if at least n/2 + f + 1 propose messages
             contain same value x then
11             x_u = x
12         else
13             choose x_u randomly with prob. 1/2
14         end
15     end
16     round += 1
17 end
```

If a correct node chooses $x$ in line 11, then no other correct node chooses a value $y \neq x$ in line 11. The algorithm above solves binary byzantine agreement for up to $f < n/10$. There are other algorithms for asynchronous byzantine agreement which tolerate up to $f < n/3$. Nearly all developed algorithms for byzantine agreement (both synchronous and asynchronous) only satisfy all-same validity, with some exceptions.

### 14.5 Random Oracle and Bitstring

A **random oracle** is a trusted (non-byzantine) random source which can generate random values.

In the previous algorithm, replace line 13 by "return $c_i$, where $c_i$ is $i$-th random bit by oracle". So instead of every node throwing a local coin (and hoping that they all show the same), the nodes will base their random decision on the proposed algorithm. Using this, we could solve asynchronous byzantine agreement in expected constant number of rounds.

Unfortunately, random oracles do not really exist in the world. We thus try to use random bitstrings to simulate the behavior of a random oracle: this is a string of random binary values known to all participating nodes when starting a protocol. So in line 13 we choose the $i$-th bit in the random bitstring. But is this really random? Not quite! As the string is known beforehand, byzantine nodes know in the $i$-th step of a protocol the value $c_{i+1}$. Thus the algorithm might not terminate.

## 15 Broadcast and Shared Coin

Our asynchronous byzantine agreement solution is awfully slow or has unrealistic assumptions. Can we at least solve asynchronous (assuming worst-case scheduling) consensus if we have crash failures?

### 15.1 Shared Coin on Blackboard

The **blackboard** is a trusted authority which supports two operations. A node can **write** its message to the blackboard and a node can **read** all the values from that have been written to the blackboard so far. We assume that the nodes cannot reconstruct the order in which the messages are written to the blackboard since the system is asynchronous.

**Algorithm 12:** Crash-Resilient Shared Coin with Blackboard

```
1 while true do
2     Choose new local coin c_u with
          c_u = { 1   with probability 1/2
                { −1  with probability 1/2
3     Write c_u to the blackboard
4     Set C = read all coins on the blackboard
5     if |C| ≥ n^2 then
6         return sign(sum(C))
7     end
8 end
```

This is a wait-free algorithm that works even for a worse-case scheduler for crashing nodes. A single node can single-handedly generate all $n^2$ coinflips, without waiting. However, it does not work for byzantine nodes, as such a node could write in rapid succession at the beginning.

Assuming a trusted blackboard does not seem practical. However, fortunately, we can use advanced broadcast methods in order to implement something like a blackboard with just messages.

### 15.2 Broadcast Abstractions

A message received by a node $v$ is called **accepted** if node $v$ can consider this message for its computation. **Best-effort broadcast** ensures that a message that is sent from a correct node $u$ to another correct node $v$ will eventually be received and accepted by $v$.

**Reliable broadcast** ensures that the nodes eventually agree on all accepted messages. That is, if a correct node $v$ considers message $m$ as accepted, then every other node will eventually consider message $m$ as accepted. The following algorithm satisfies this definition:

**Algorithm 13:** Asynchronous Reliable Broadcast

```
1 Broadcast own message msg(u)
2 upon receiving msg(v) from v or echo(w, msg(v))
    from n − 2f nodes w:
3 Broadcast echo(u, msg(v)
4 end upon
5 upon receiving echo(w, msg(v)) from n − f nodes
    w:
6 Accept msg(v)
7 end upon
```

This algorithm satisfies the following three properties:

- If a correct node broadcasts a message reliably, it will eventually be accepted by every other correct node.
- If a correct node has not broadcast a message, it will not be accepted by any other correct node.
- If a correct node accepts a message, it will be eventually accepted by every correct node.

This algorithm can tolerate $f < n/3$ byzantine nodes or $f < n/2$ crash failures.

It only makes sure that all messages of correct nodes will be accepted *eventually*. This algorithm allows byzantine nodes to issue arbitrarily many messages, which may result in problems for protocols where each node is only allowed to send one message per round.

**FIFO reliable broadcast** defines an order in which the messages are accepted in the system. If a node $u$ broadcasts message $m_1$ before $m_2$, than any node $v$ will accept $m_1$ before $m_2$.

**Algorithm 14:** FIFO Reliable Broadcast

```
1 Broadcast own round r message msg(u, r)
2 upon receiving first msg(v, r) from v for round r
    or echo(w, msg(v, r)) from n − 2f nodes w:
3 Broadcast echo(u, msg(v, r)
4 end upon
5 upon receiving echo(w, msg(v, r)) from n − f
    nodes w:
6 if accepted msg(v, r − 1) then
7 |   Accept msg(v, r)
8 end
9 end upon
```

This algorithm can tolerate $f < n/5$ byzantine nodes or $f < n/2$ crash failures. Further it only only accepts one message per node. **Atomic broadcast** makes sure that all messages are accepted in the same order by every node. Now we finally have all tools to solve asynchronous consensus.

## 15.3 Blackboard with Message Passing

**Algorithm 15:** Crash-Resilient Shared Coin

```
1  while true do
2  |   Choose new local coin c_u with
   |
   |                  { 1    with probability 1/2
   |       c_u =      { −1   with probability 1/2
   |
3  |   FIFO-broadcast coin(c_u, r) to all nodes
4  |   Save all received coins coin(c_v, r) in a set C_u
5  |   Wait until accepted own coin(c_u)
6  |   Request C_v from n − f nodes v, and add newly
   |    seen coins to C_u
7  |   if |C_u| ≥ n^2 then
8  |   |   return sign(sum(C_u))
9  |   end
10 end
```

This solves asynchronous binary agreement for $f < n/2$ crash failures. But what about byzantine agreement? We need even more powerful methods!

## 15.4 Using Cryptography

Let $t, n \in \mathbb{N}$ with $1 \leq t \leq n$. An algorithm that distributes a secret among $n$ participants such that $t$ participants need to collaborate to recover the secret is called a $(t, n)$-**threshold secret sharing scheme**.

Every node can **sign** its messages in a way that no other node can forge, thus nodes can reliably determine which node a signed message originated from. We denote a message $x$ signed by node $u$ with $\text{msg}(x)_u$.
Those methods allow us to create an algorithm like this:

**Algorithm 16:** $(t, n)$-Treshold Secret Sharing

```
1 Input: A secret s, represented as a real number
  /* Secret distribution by dealer d          */
2 Generate t − 1 random numbers a_1, ..., a_{t−1} ∈ ℝ
3 Obtain a polynomial p of degree t − 1 with
                p(x) = s + a_1 x + ... + a_{t−1} x^{t−1}
4 Generate n distinct x_1, ..., x_n ∈ ℝ\{0}
5 Distribute share msg(x_1, p(x_1))_d to node v_1, ...,
    msg(x_n, p(x_n))_d to node v_n
  /* Secret recovery                          */
6 Collect t shares msg(x_u, p(x_u))_d from at least t
    nodes
7 Use Lagrange's interpolation formula to obtain
    p(0) = s
```

This algorithm relies on a trusted dealer who cannot by byzantine and creates the polynomial. Further the communication between the dealer and the nodes must be private, i.e., a byzantine party cannot see the shares sent to the correct nodes. Using an $(f+1, n)$-threshold secret sharing scheme, we can encrypt messages in such a way that byzantine nodes alone cannot decrypt them. This allows us to solve byzantine agreement for $f < n/10$ in expected 3 number of rounds.

A **hash function** $U \mapsto S$ is called **cryptographic**, if for a given $z \in S$ it is computationally hard to find an element $x \in U$ with $\text{hash}(x) = z$. Examples are the Secure Hash Algorithm (SHA) and the Message-Digest Algorithm (MD). With cryptographic hashing, we can implement a synchronous byzantine shared coin:

**Algorithm 17:** Simple Synchronous Byzantine Shared Coin

```
1 Each node has a public key that is known to all
    nodes
2 Let r be the current round of Alg. 11
3 Broadcast msg(r)_u, i.e., round number r signed by
    node u
4 Compute h_v = hash(msg(r)_v) for all received
    messages msg(r)_v
5 Let h_min = min_v h_v
6 return least significant bit of h_min
```

This algorithm plugged into Alg. 11 solves synchronous byzantine agreement in expected 3 rounds (roughly) for up to $f < n/10$ byzantine failures.

# 16 Consistency and Logical Time

This section involves concepts already seen in Parallel Programming and DMDB.

## 16.1 Consistency Models

An **object** is a variable or a data structure storing information. Object is a general term for any entity that can be modified. An **operation** $f$ accesses or manipulates an object. The operation $f$ starts at wall-clock time $f_*$ and ends at wall-clock time $f_\dagger$. If $f_\dagger < g_*$ we simply write $f < g$.

An **execution** $E$ is a set of operations on one or multiple objects that are executed by a set of nodes. An execution restricted to a single node is a **sequential execution**. This means that no two operations $f$ and $g$ are concurrent, i.e., we have $f < g$ or $g < f$.

Two executions are **semantically equivalent** if they contain exactly the same operations. Moreover, each pair of corresponding operations has the same effect in both executions.

An execution $E$ is called **linearizable** (or atomically consistent), if there is a sequence of operations (sequential execution) $S$ such that:

- $S$ is correct and semantically equivalent to $E$
- Whenever $f < g$ for two operations $f, g$ in $E$, then also $f < g$ in $S$

A linearization point of operation $f$ is some $f_\circ \in [f_*, f_\dagger]$. $E$ is linearizable if and only if there exist linearization points such that the sequential execution $S$ that results in ordering the operations according to those linearization points is semantically equivalent to $E$.

An execution $E$ is called **sequentially consistent**, if there is a sequence of operations $S$ such that:

- $S$ is correct and semantically equivalent to $E$
- Whenever $f < g$ for two operations $f, g$ on the same node in $E$, then also $f < g$ in $S$

Every linearizable execution is also sequentially consistent.

$$\text{linearizability} \implies \text{sequential consistency}$$

An execution $E$ is called **quiescently consistent**, if there is a sequence of operations $S$ such that:

- $S$ is correct and semantically equivalent to $E$
- Let $t$ be some quiescent point, i.e., for all operations $f$ we have $f_\dagger < t$ or $f_* > t$. Then for every $t$ and every pair of operations $g, h$ with $g_\dagger < t$ and $h_* > t$ we also have $g < h$ in $S$

Every linearizable execution is also quiescently consistent.

$$\text{linearizability} \implies \text{quiescently consistency}$$

However, sequentially consistent and quiescent consistency do not imply one another. A system or an implementation is called linearizable if it ensures that every possible execution is linearizable. Analogous definitions exist for sequential and quiescent consistency.

Let $E$ be an execution involving operations on multiple objects. For some object $o$ we let the **restricted execution** $E|o$ be the execution $E$ filtered to only contain operations involving object $o$.

A consistency model is called **composable** if the following holds: If for every object $o$ the restricted execution $E|o$ is consistent, then also $E$ is consistent. Sequential consistency is not composable, but linearizability is.

## 16.2 Logical Clocks

To capture dependencies between nodes in an implementation, we can use logical clocks. These are supposed to respect the so-called happened-before relation.

Let $S_u$ be a sequence of operations on some node $u$ and define $\to$ to be the **happened-before relation** on $E := S_1 \cup ... \cup S_n$ that satisfies the following three conditions:

1. If a local operation $f$ occurs before operation $g$ on the same node (i.e. $f < g$), then $f \to g$
2. If $f$ is a send operation of one node and $g$ is the corresponding receive operation of another node, then $f \to g$
3. If $f, g, h$ are operations such that $f \to g, g \to h$ then also $f \to h$

If for two distinct operations $f, g$ neither $f \to g$ nor $g \to f$, then we also say $f$ and $g$ are independent and write $f \sim g$. Sequential computations are characterized by $\to$ being a total order, whereas the computation is entirely concurrent if no operations $f, g$ with $f \to g$ exist.

An execution $E$ is called **happened-before consistent**, if there is a sequence of operations $S$ such that:

- $S$ is correct and semantically equivalent to $E$
- Whenever $f \to g$ in $E$, then also $f < g$ in $S$

This is actually an equivalent definition to sequential consistency.

A **logical clock** is a family of functions $c_u$ that map every operation $f \in E$ on node $u$ to some logical time $c_u(f)$ such that the happened-before relation is respected, i.e., for two operations $g$ on node $u$ and $h$ on node $v$:

$$g \to \implies c_u(g) < c_u(h)$$

If the reverse implication also holds, then the clock is called a **strong logical clock**.

**Algorithm 18:** Lamport Clock

```
1 Initialize c_u := 0
2 Upon local operation: Increment current local
    time c_u := c_u + 1
3 Upon send operation: Increment c_u := c_u + 1 and
    include c_u as T in message
4 Upon receive operation: Extract T from message
    and update c_u := max(c_u, T) + 1
```

This is not an implementation of a strong log- ical clock. To achieve this, nodes also have to gather information about other clocks in the system. We can do this with **vector clocks**:

**Algorithm 19:** Lamport Clock

1 Initialize $c_u[v] := 0$ for all other nodes $v$
2 Upon local operation: Increment current local time $c_u[u] := c_u[u] + 1$
3 Upon send operation: Increment $c_u[u] := c_u[u] + 1$ and include the vector $c_u$ as $d$ in message
4 Upon receive operation: Extract vector $d$ from message and update $c_u[v] := \max(d[v], c_u[v])$ for all entries $v$. Increment $c_u[u] := c_u[u] + 1$

We define $c_u < c_v$ if $c_u[w] \leq c_v[w]$ for all entries $w$ and $c_u[x] < c_v[x]$ for at least one entry. Then vector clocks are strong logical clocks.

Usually, the number of interacting nodes is small compared to the overall number of nodes. Therefore, we do not have to send the whole vector, but only some entries of the nodes that are actually communicating. This is called the **differential technique**.

## 16.3 Consistent Snapshots

A **cut** is some prefix of a distributed execution. More precisely, if a cut contains an operation $f$ on some node $u$, then it also contains all the preceding operations of $u$. The set of last operations on every node included in the cut is called the **frontier** of the cut. A cut $C$ is called **consistent** if for every operation $g$ in $C$ with $f \to g$, $C$ also contains $f$.

A **consistent snapshot** is a consistent cut $C$ plus all messages in transit at the frontier of $C$. In a consistent snapshot it is forbidden to see an effect without its cause.

We say that a system is **fully defined** (at any point during the execution) by its configuration. The configuration includes the state of every node, and all messages that are in transit (sent but not yet received).

The following algorithm collects a consistent snapshot:

**Algorithm 20:** Distributed Snapshot Algorithm

1 **Initiator:** Save local state, send a snap message to all other nodes and collect incoming states and messages of all other nodes
2 **All other Nodes:**
3 Upon receiving a snap message for the first time: send own state (before message) to the initiator and propagate snap by adding snap tag to future messages
4 If afterwards receiving a message $m$ without snap tag: Forward $m$ to the initiator

Let $q_u$ be the number of operations on node $u$. Then the number of consistent snapshots (including the empty cut) in thesequential case is $\mu_s := 1 + q_1 + q_2 + ... + q_n$.

The number of consistent snapshots in the concurrent case is $\mu_c := (1 + q_1) \cdot (1 + q_2) \cdot ... \cdot (1 + q_n)$.

The concurrency measure of an execution $E = (S_1, ..., S_n)$ is defined as the ratio:

$$m(E) = \frac{\mu - \mu_s}{\mu_c - \mu_s}$$

Where $\mu$ denotes the number of consistent snapshot of $E$. This measure of concurrency is normalized.

## 16.4 Distributed Tracing

A microservice architecture refers to a system composed of loosely coupled services. These services communicate by various protocols and are either decentrally coordinated (also known as "choreography") or centrally ("orchestration"). Microservices are the architecture of choice to implement a cloud based distributed system.

Due to the often heterogeneous technology, a uniform debugging framework is not feasible. Tracing enables tracking the set of services which participate in some task, and their interactions.

A **span** $s$ is a named and timed operation representing a contiguous sequence of operations on one node. A span $s$ has a start time $s_*$ and finish time $s_\dagger$.

Spans represent tasks, like a client submitting a request or a server processing this request. Spans often trigger several child spans or forwards the work to another service. A span may causally depend on other spans. The two possible relations are **ChildOf** and **FollowsFrom** references.

In a ChildOf reference, the parent span depends on the result of the child, and therefore parent and child span must overlap. In FollowsFrom references parent spans do not depend in any way on the result of their child spans (the parent just invokes the child).

A **trace** is a series-parallel directed acyclic graph representing the hierarchy of spans that are executed to serve some request. Edges are annotated by the type of the reference, either ChildOf or FollowsFrom.

The following algorithm shows what is needed if you want to trace requests to your system.

**Algorithm 21:** Inter-Service Tracing

1 Upon requesting another service: Inject information of current trace and span (IDs or timing information) into the request header.
2 Upon receiving request from another service: Extract trace and span information from the request header and create new span as child span.

All tracing information is collected and has to be sent to some tracing backend which stores the traces and can provide a frontend to understand what is going on.

# 17 Time, Clock and GPS

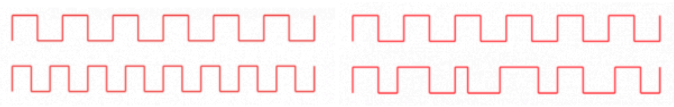
## 17.1 Time and Clocks

We define a **second** based on the oscillation cycles of a caesium-133 atom.

The wall-clock time $t_*$ is the true time (a perfectly accurate clock would show).

A **clock** is a device which tracks and indicates time. Its time $t$ is a function of the wall-clock time $t_*, t = f(t_*)$. Ideally, $f$ is the identity function.

The **clock error** or clock skew is the difference between two clocks. In practice the clock error is often modeled as $t = (1 + \delta) \cdot t_* + \epsilon \cdot t_*$. Accurate timekeeping and clock synchronization are very important problems.



The **drift** $\delta$ (left) is the predictable clock error. It is relatively constant over time, but can vary with supply voltage, temperature etc. Stable clock sources are more expensive and larger. Clock drift is indicated in parts per million (ppm). One ppm corresponds to a time error growth of one microsecond per second.

The **jitter** $\epsilon$ is the unpredictable, random noise of the clock error. It is the irregularity of the clock and can vary fast.

## 17.2 Clock Synchronization

Clock synchronization is the process of matching multiple clocks to have a common time. There is a trade-off between accuracy, convergence time and cost.

**Algorithm 22:** Network Time Protocol (NTP)

Client $u$ and Server $v$
**while** *true* **do**
  Node $u$ sends request to $v$ at time $t_u$
  Node $v$ receives request at time $t_v$
  Node $v$ processes the request and replies at time $t'_v$
  Node $u$ receives the response at time $t'_u$
  Propagation delay: $\delta = \frac{(t'_u - t_u) - (t'_v - t_v)}{2}$
  Clock skew: $\theta = \frac{(t_v - (t_u + \delta)) - (t'_u - (t'_v + \delta))}{2} = \frac{(t_v - t_u) + (t'_v - t'_u)}{2}$
  Node $u$ adjusts clock by $+\theta$
  Sleep before next synchronization
**end**

Most NTP servers are public and answer to UDP packets. There is a hierarchy of NTP servers in a forest structure. Despite unpredictable clock errors, we can limit the maximum error by using regular synchronization.

The Precision Time Protocol (PTP) is a clock synchronization protocol similar to NTP, but which uses medium access control (MAC) layer timestamps. This removes the unknown time delay incurred through message passing through the network stack. This way, we can achieve sub-microsecond accuracy in local networks.

Global synchronization establishes a common time between any two nodes in the system. NTP and PTP both optimize for global synchronization. However, two nodes may receive their timestamps through different paths of the forest, accumulating different errors. Thus, a message from $u$ might arrive at $v$ with a timestamp from the future.

We can easily achieve local time synchronization by constantly exchanging the current time with the one from the direct neighbors and taking the average/median. This is a method of choice for time-division multiple access (TDMA) and coordination in wireless networks.

In wireless networks, one can simplify synchronization:

**Algorithm 23:** Wireless Clock Synchronization with Known Delays

Given: transmitter $s$, receivers $u, v$, with known transmission delays $d_u, d_v$ from transmitter $s$, respectively
$s$ sends signal at time $t_s$
$u$ receives signal at time $t_u$
$v$ receives signal at time $t_v$
$\Delta_u = t_u - (t_s + d_u)$
$\Delta_v = t_v - (t_s + d_v)$
Clock skew between $u$ and $v$: $\theta = \Delta_v - \Delta_u$

## 17.3 Time Standards

The **International Atomic Time** (TAI) is a time standard derived from over 400 atomic clocks distributed worldwide. Using a weighted average over all involved clocks, TAI is more stable than the best known clock.

A leap second is an extra second added to a minute to make it irregularly 61 instead of 60 seconds long. This is used to compensate for the slowing of the Earths rotation. There are also negative leap seconds (59 instead of 60), but they were never used so far.

The Coordinated Universal Time (UTC) is a time standard based on TAI with leap seconds added at irregular intervals to keep it close to mean solar time at 0° longitude. Before, we had GMT which is not based on atomic clocks and thus got replaced by UTC in 1967.

A standardized format for timestamps, mostly used for processing by computers, is the ISO 8601 standard. According to this standard, a UTC timestamp looks like this:

$$1712 - 02 - 30\mathrm{T}07 : 39 : 52\mathrm{Z}$$

T separates the date and time parts while Z indicates the time zone with zero offset from UTC.

## 17.4 Clock Sources

There are different clock sources that were used over the period of time:

- An atomic clock is a clock which keeps time by counting oscillations of atoms. Such clocks are the most accurate clocks known.
- The system clock in a computer is an oscillator used to synchronize all components on the motherboard. Usually, a quartz crystal oscillator with some tens / hundreds of MHz is used (precision of some ns). The CPU clock is generated using a multiple of the system clock (through a clock multiplier).

  To guarantee nominal operation of the computer, the system clock must have low jitter. The drift, on the other hand, is irrelevant. The system clock only runs when the computer is on.
- The real-time clock (RTC) in a computer is a battery backed oscillator which is running even if the computer is shut down or unplugged. It is read at startup to initialize the system clock. Even if a computer is not connected to a network, it keeps its time close to the GTC. It's frequency is often exactly 215 Hz, which allows more simple binary counter circuits to be used.
- A radio time signal is a time code transmitted via radio waves by a time signal station, referring to a time in a given standard such as UTC. Radio-controlled clocks are the most common application of such clocks.
- A power line clock measures the oscillations from electric AC powerlines, e.g. at 50Hz. E.g. clocks in kitchen ovens are driven by such clocks. It is relatively stable and uses very little energy, but it is quite imprecise.

- Sunlight time synchronization is a method of reconstructing global timestamps by correlating annual solar patterns from light sensors length of day measurement. Its relatively inaccurate, but well-suited for long-time measurements with data storage and post-processing.

The most popular source of time is probably GPS.

## 17.5 GPS

The **global positioning system** (GPS) is a global navigation satellite system (GNSS), consisting of at least 24 satellites orbiting around the world, each continuously transmitting its position and times code.

Positioning is done in space and time! GPS provides information of those two anywhere on Earth where at least four satellite signals can be received. Signal delay is between 64 and 89 ms.

**Pseudo-Random Noise** (PRN) sequences are pseudo-random bit strings. Each GPS satellite uses a unique PRN sequence with a length of 1023 bits for its signal transmissions. To simplify our math, each PRN bit is either 1 or -1.

**Navigation Data** is the data transmitted from satellites, which includes orbit parameters to determine satellite positions, timestamps of signal transmission, atmospheric delay estimations and status information of the satellites and GPS as a whole, such as the accuracy and validity of the data.

**Algorithm 24:** GPS Satellite

```
1  Given: Each satellite has a unique PRN sequence,
     plus some current navigation data D (±1).
2  The code below is simplified, only concentrating
     on the digital aspects.
3  while true do
4  |  for all bits D_i ∈ D do
5  |  |  for j = 0...19 do
6  |  |  |  for k=0...1022 do
7  |  |  |  |  Send bit PRN_k · D_i
8  |  |  |  end
9  |  |  end
10 |  end
11 end
```

For better robustness, we repeat a single bit multiple times in the following algorithm (line 5).

The **circular cross-correlation** is a similarity measure between two vectors of length $N$, circularly shifted by a given displacement $d$:

$$\text{cxcorr}(a, b, d) = \sum_{i=0}^{N-1} a_i \cdot b_{i+d} \mod N$$

The two vectors are most similar at the displacement $d$ where the sum is maximum. The vector of cross-correlation values with all $N$ displacements can efficiently be computed using a fast Fourier transform.

Acquisition is the process in a GPS receiver that finds the visible satellite signals and detects the delays of the PRN sequences and the Doppler shifts of the signals. The relative speed between satellite and receiver introduce a significant Doppler shift to the carrier frequency which needs to be found in order to decode a signal:

**Algorithm 25:** Acquisition

```
1  Received 1 ms signal s with sampling rate r · 1023
     kHz
2  Possible Doppler shifts F, e.g.
     {−10kHz, −9.8kHz, ..., +10kHz}
3  Tensor A = 0: Satellite × carrier frequency × time
4  for all satellites i do
5  |  PRN'_i = PRN_i stretched with ratio r
6  |  for all Doppler shifts f ∈ F do
7  |  |  Build modulated PRN''_i with PRN'_i and
   |  |    Doppler frequency f
8  |  |  for all delays d ∈ {0, ..., 1023 · r − 1} do
9  |  |  |  A_i(f, d) = |cxcorr(s, PRN''_i, d)|
10 |  |  end
11 |  end
12 end
13 Select d* that maximizes max_d max_f A_i(f, d)
14 Signal arrival time r_i = d*/(r · 1023kHz)
```

Using all the facts from above, we can build a classic GPS receiver:

**Algorithm 26:** Classic GPS Receiver

```
1  h: Unknown receiver handset position
2  θ: Unknown handset time offset to GPS system
     time
3  r_i: Measured signal arrival time in handset time
     system
4  c: Signal propagation speed (GPS: speed of light)
5  Perform acquisition
6  Track signal and decode navigation data
7  for all satellites i do
8  |  Using navigation data, determine signal
   |    transmit time s_i and position p_i
9  |  Measured satellite transmission delay
   |    d_i = r_i − s_i
10 end
11 Solve the following system of equations for h and θ:
12 ||p_i − h||/c = d_i − θ for all i
```

GPS satellites carry precise atomic clocks, but the receiver is not synchronized with the satellites. We thus can- not determine the exact distance between satellite and receiver (even though timestamps are included). In total, the positioning problem contains four unknown variables, three for the handset's spatial position and one for its time offset from the system time. Therefore, signals from at least four transmitters are needed to find the correct solution. However, more received signals reduce the measurement noise and increase the accuracy.

An **assisted GPS** (A-GPS) receiver fetches the satellite orbit parameters and other navigation data form the Internet. We use this method to reduce the data transmission time and thus the TTFF from a max. of 30s to 6s.

Another GPS improvement is **Differential GPS** (DGPS), where a receiver with a fixed location within a few kilometres of a mobile receiver compares the observed and actual satellite distances. This error is then subtracted at the mobile receiver. We can achieve accuracies in the order of 10cm.

A **snapshot receiver** is a GPS receiver that captures one or a few milliseconds of raw GPS signal for a position fix. They aim at the remaining latency that results from the transmission of timestamps from the satellite every 6 seconds.

**Coarse Time Navigation** (CTN) is a snapshot receiver positioning technique measuring sub-millisecond satellite ranges from correlation peaks, like conventional GPS receivers. A CTN receiver determines the signal transmit times and satellite positions from its own approximate location by subtracting the signal propagation delay from the receive time. As receiver location/time is not exactly known, we round to the nearest whole millisecond. However, this way noise cannot be averaged out well and may lead to wrong signal arrival time estimates. Usually, this renders the system of equations unsolvable, making positioning infeasible!

**Collective detection** (CD) is a maximum likelihood snapshot receiver localization method which does not determine an arrival time for each satellite, but rather combines all the available information and take a decision only at the end of the computation. It is more robust than CTN as it can tolerate a few low quality signals.

**Algorithm 27:** Collective Detection Receiver

```
Given: A raw 1 ms GPS sample s, a set H of location/time hypotheses
In addition, the receiver learned all navigation and atmospheric data
for all hypotheses h ∈ H do
    Vector r = 0
    Set V = satellites that should be visible with hypothesis h
    for satellites i in V do
        r = r + r_i, where r_i is expected signal of satellite i. The data of vector r_i incorporates all available information: distance and atmospheric delay between satellite and receiver, frequency shift because of Doppler shift due to satellite movement, current navigation data bit of satellite, etc.
    end
    P_h = cxcorr(s, r, 0)
end
Solution: hypothesis h ∈ H maximizing P_h
```

## 17.6 Lower Bounds

In the clock synchronization problem, we are given a network (graph) with $n$ nodes. The goal for each node is to have a clock such that the clock values are well synchronized, and close to real time. We assume a bounded but variable drift, i.e. each node has a hardware clock producing pulses between $[1-\epsilon, 1+\epsilon], \epsilon << 1$ and an arbitrary jitter in the delivery times. The goal is to design a message-passing algorithm that ensures that the logical clock skew of adjacent nodes is as small as possible at all times.

In a network of nodes, the local clock skew is the skew between neighboring nodes, while the global clock skew is the maximum skew between any two nodes.

The global clock skew is $\Omega(D)$, where $D$ is the diameter of the network graph. Many natural algorithms manage to achieve a global clock skew of $O(D)$, so we look at local clock skew:

**Algorithm 28:** Local Clock Synchronization at node $v$

```
while not done do
    send logical time t_v to all neighbors
    if Receive logical time t_u, where t_u > t_v, from any neighbor u then
        t_v = t_u
    end
end
```

This algorithm has a local skew of $\Omega(n)$. It was shown that the local clock skew is in $\Theta(\log D)$, but any natural clock synchronization algorithm has a bad local skew. However, all of these are worse-case bounds; in practice, clock drift and message delay may not be the worse possible. Under more realistic (weaker) assumptions, better protocols exist in theory and in practice.

# 18 Quorum Systems

In chapters before, we took decisions based on a majority-approach: if $\lfloor n/2 \rfloor + 1$ nodes decide for a value, clearly all nodes must decide for that value. This is very fault tolerant, but highly inefficient and not scalable at all! Instead, we will look at so called **quorum**.

Let $V = \{v_1, ..., v_n\}$ be a set of nodes. A quorum $Q \subseteq V$ is a subset of these nodes s.t. every two quorums intersect. When a quorum system is being used, a client selects a quorum $Q$, acquires a lock on all nodes of $Q$ and when done leases all locks again. No matter which quorum is chosen, its nodes will intersect with each other quorum.

A quorum system $S$ is called **minimal** if $\forall Q_1, Q_2 \in S : Q_1 \not\subset Q_2$. The simplest quorum system imaginable consists of just one quorum, which in turn just consists of one server. It is known as **Singleton**. In the **Majority quorum system**, every quorum has $\lfloor n/2 \rfloor + 1$ nodes.

## 18.1 Load and Work

An access strategy $Z$ defines the probability $P_Z(Q)$ of accessing a quorum $Q \in S$ s.t. $\sum_{Q \in S} P_Z(Q) = 1$. We can define two measurements for quorum systems:

**Load**:

- The load of access strategy $Z$ on a node $v_i$ is

$$L_z(v_i) = \sum_{Q \in S, v_i \in Q} P_Z(Q)$$

  The load is the probability that $v_i \in Q$ if $Q$ is sampled from $S$.

- The load induced by access strategy $Z$ on a quorum system $S$ is the maximal load induced by $Z$ on any node in $S$:

$$L_Z(S) = \max_{v_i \in S} L_Z(v_i)$$

- The load of a quorum system $S$ is:

$$L(S) = \min_Z L_Z(S)$$

**Work**

- The work of a quorum $Q \in S$ is the number of nodes in $Q$:

$$W(Q) = |Q|$$

- The work induced by access strategy $Z$ on a quorum system $S$ is the expected number of nodes accessed:

$$W_Z(S) = \sum_{Q \in Z} P_Z(Q) \cdot W(Q)$$

- The work of a quorum system $S$ is:

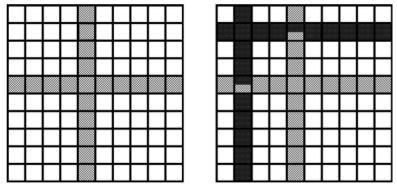
$$W(S) = \min_Z W_Z(S)$$

Note that you cannot choose different access strategies $Z$ for work and load, you have to pick a single $Z$ for both. If every quorum $Q$ in a quorum system $S$ has the same number of elements, $S$ is called **uniform**.

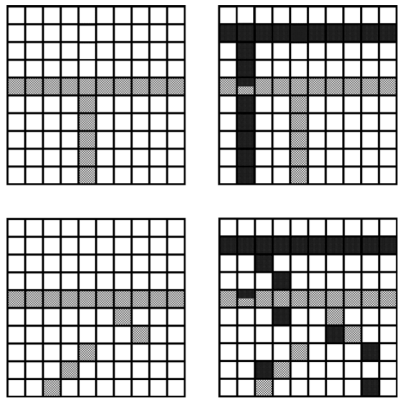
Let $S$ be a quorum system. Then $L(S) \geq 1/\sqrt{n}$ holds.

## 18.2 Grid Quorum System

We try to achieve this lower bound for the load with so called grid quorum systems. Assume $\sqrt{n} \in \mathbb{N}$, then arrange the $n$ nodes in a $\sqrt{n} \times \sqrt{n}$ grid. The basic grid quorum consists of $\sqrt{n}$ quorums, each containing the full row and column $i$.



However, with the simple quorum system, two quorums intersect at two nodes. They could enter a deadlock, if both quora try to get locks at the same time. By introducing some kind of ordering (e.g. the one which holds the highest lock, i.e. the one with the right-most lock in the last row, will get the lock), we can solve this problem. There are different quorum systems which only intersect in one node, e.g. the ones below:



In the script, there is a detailed algorithm for locking strategies that prevent deadlocks. There are also other problems, e.g., what happens when a quorum holds locks and crashes? Instead of locks we could e.g. use leases instead of locks (which have a time out).

## 18.3 Fault Tolerance

If any $f$ nodes from a quorum system $S$ can fail s.t. there is still a quorum $Q \in S$ without failed nodes, then $S$ is $f$-resilient. The largest such $f$ is the resilience $R(S)$.

If $S$ is a Grid quorum system where each of the $n$ quorums consists of a full row and a full column, S has a re- silience of $\sqrt{n} - 1$: Either, all $\sqrt{n}$ nodes on the diagonal fail. Else, no matter which $\leq \sqrt{n} - 1$ nodes fail, there is always a row and column without failed nodes.

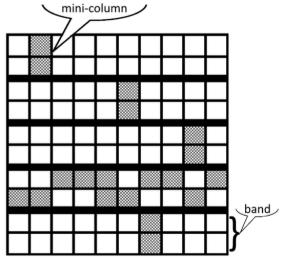
Assume that every node works with a fixed probability $p$. The failure probability $F_p(S)$ of a quorum system $S$ is the probability that at least one node of every quorum fails. The asymptotic failure probability is $F_p(S)$ for $n \to \infty$.

We have proven the following asymptotic failure probabilities:

- Majority quorum system: failure probability 0
- Grid quorum system: failure probability 1

As we can see, we have a system with optimal load and one with fault-tolerance. If we want to achieve both, we need B-Grind quorum systems.

Consider $n = dhr$ nodes, arranged in a rectangular grid with $h \cdot r$ rows and $d$ columns. Each group of $r$ rows is a band, and $r$ elements in a column restricted to a band are called a mini-column. A quorum consists of one mini-column in every band and one element from each mini-column of one band. Thus, every quorum has $d + hr - 1$ elements. The B-Grid quorum system consists of all such quorums.




Indeed, the asymptotic failure probability of the B-Grid quorum system is 0.

| | **Singleton** | **Majority** | **Grid** | **B-Grid*** |
|---|---|---|---|---|
| Work | $\mathbf{1}$ | $> n/2$ | $\Theta(\sqrt{n})$ | $\Theta(\sqrt{n})$ |
| Load | 1 | $> 1/2$ | $\Theta(\mathbf{1}/\sqrt{\mathbf{n}})$ | $\Theta(\mathbf{1}/\sqrt{\mathbf{n}})$ |
| Resilience | 0 | $< \mathbf{n}/\mathbf{2}$ | $\Theta(\sqrt{n})$ | $\Theta(\sqrt{n})$ |
| F. Prob.** | $1 - p$ | $\to \mathbf{0}$ | $\to 1$ | $\to \mathbf{0}$ |

## 18.4 Byzantine Quorum System

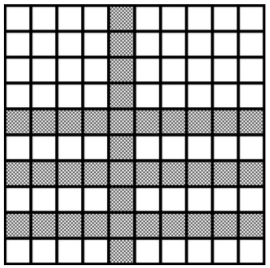
While failed nodes are bad, they are still easy to deal with: just access another quorum where all nodes can respond! Byzantine nodes make life more difficult.

A quorum system $S$ is $f$-disseminating if the intersection of two different quorums always contains $f + 1$ nodes (1), and for any set of $f$ byzantine nodes, there is at least one quorum without byzantine nodes (2). If we have some authentication mechanism (the data is self-verifying) in our system, then (1) is strong enough as the correct node could verify the data. If this does not hold, then we need another mechanism.

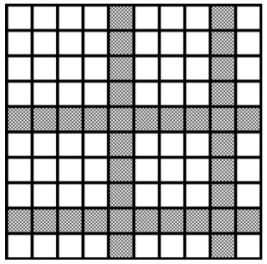
A quorum system $S$ is $f$-masking if (1) the intersection of two different quorums always contains $2f+1$ nodes and (2) for any set of $f$ byzantine nodes, there is at least one quorum without byzantine nodes. The idea behind (1) is that $f + 1$ nodes can outvote all the byzantine nodes. Masking quorum systems need more than $4f$ nodes.

The loads for these systems are $L(S) \geq \sqrt{(f+1)/}$ for the $f$-disseminating and $L(S) \geq \sqrt{(2f+1)/2}$ for the $f$-masking.

A $f$-masking Grid quorum system is constructed as the grid quorum system, but each quorum contains one full column and $f + 1$ rows of nodes, with $2f + 1 \leq \sqrt{n}$. In such a system, two quorums overlap by their columns intersecting each others rows the overlap is thus at least $2f + 2$ nodes. The $f$-masking Grid nearly hits the lower bound, but is not optimal.



The $M$-Grid quorum system on the other hand hits the lower bound: it is constructed as the grid quorum system, but each quorum contains $\sqrt{f+1}$ rows and $\sqrt{f+1}$ columns of nodes, with $f \leq \frac{\sqrt{n}-1}{2}$. For both of these systems, the load is in $\Theta(\sqrt{f/n})$.



We achieved nearly the same load as without byzantine nodes! However, as mentioned earlier, what happens if we access a quorum that is not up-to-date, except for the intersection with an up-to-date quorum?

We might want to ensure that the number of correct up-to-date nodes accessed will be larger than the number of out-of-date nodes and byzantine nodes.

A quorum system $S$ is $f$-opaque if the following two properties hold for any set of $f$ byzantine nodes $F$ and any two different quorums $Q_1, Q_2$:

$$|(Q_1 \cap Q_2)/F| > |(Q_2 \cap F) \cup (Q_2/Q_1)|$$

$$(F \cap Q) = \emptyset \text{ for some } Q \in S$$

For a $f$-opaque system, we need $n > 5f$. For a $f$-opaque quorum system $S$, we also have $L(S) \geq 1/2$.

# 19 Distributed Storage

How can we store a huge amount of large files across many nodes in a network? Which components in the network change over time? Can we do something better than a global index (what file is stored on which node)?

## 19.1 Consistent Hashing

We might want to use some sort of hashing, e.g. consistent hashing (we already have seen this concepts in other lectures).

**Algorithm 29:** Consistent Hashing

1. Hash the unique file name of each file $x$ with a known set of hash functions $h_i(x) \mapsto [0,1)$, for $i = 1, ..., k$
2. Hash the unique name (e.g. IP address and port number) of each node with the same hash function $h(u) \mapsto [0,1)$
3. Store a copy of movie $x$ on node $u$ if $h(x) \approx h(u)$, for any $i$. More formally, store movie $x$ on node $u$ if :

$$|h_i(x) - h_(u)| \min_v \{|h_i(x) - h_(v)|\}, \text{ for any } i$$

In expectation, each node in this algorithm stores $km/n$ files, where $k$ is the number of hash functions, $m$ the number of different files and $n$ the number of nodes.

We can also choose to use pointers, then we can store the files on any node we like (e.g. a data centre) and let the weaker nodes simply return the forward pointer to the actual location. For better load balancing, we might want to hash multiple times.

In this chapter, we consider nodes with high churn: nodes are very unreliable and may only be available for a short amount of time. In this scenario, as hundreds of nodes will change every second, no single node can have an accurate picture of all the other nodes in the system. It is impossible to have a consistent view at any time.

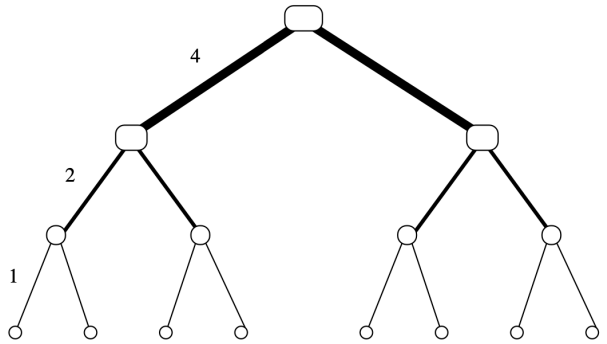
A node will have information about its neighbors (a small subset of all nodes). Thus, it does not directly know which node is responsible for what file. Instead, it asks its neighbor who recursively asks its neighbor, too. Thus, the nodes form a virtual network (an overlay network).

## 19.2 Hypercubic Networks

Our virtual network should have the following properties:

- The network should be more or less **homogeneous**, i.e. no node plays a dominant role and there is no single point of failure.
- The nodes should have **IDs**. all the IDs should span the universe $[0,1)$.
- Every node should have a **small degree**. This allows a node to maintain a persistent connection with each neighbor, allowing us to deal with churn.
- The network should have a **small diameter** and routing should be easy. If a node doesnt have the required information itself, it should know which neighbor it must ask. Within a few hops, we should find the node containing the correct information.

One possible network topology that can be used are trees. Routing is very easy, but basic trees are not homogeneous: the root is a bottleneck. Using **fat trees** where every edge connecting $v$ to its parent $u$ has a capacity that is proportional to the number of leaves in the subtree of $v$.

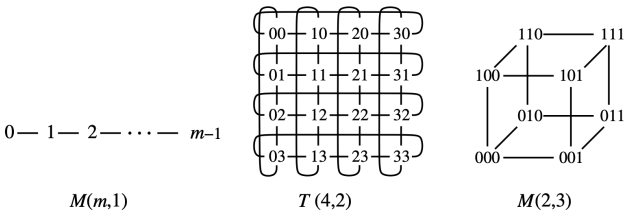



Another topology are tori and meshes: Let $m, d \in \mathbb{N}$, the $(m,d)$-mesh $M$ is a graph with node set $V = [m]^d$ (vectors of length $d$ of numbers $\{1, ..., n\}$) and edge set

$$E = \left\{ \{(a_1, ..., a_d), (b_1, ..., b_d) \mid a_i, b_i \in [m], \sum_{i=1}^{d} |a_i, b_i| = 1\} \right\}$$

The $(m,d)$-torus $T(m,d)$ is a graph that consists of an $(m,d)$-mesh and additionally wrap-around edges

from nodes $(a_1, ..., a_{i-1}, m-1, a_{i+1}, ..., a_d)$ to nodes $(a_1, ..., a_{i-1}, 0, a_{i+1}, ..., a_d)$ for all $i \in \{1, ..., d\}$ and all $a_j \in [m]$ with $j \neq i$. $M(m,1)$ is a path, $T(m,1)$ a cycle and $M(2,d) = T(2,d)$ a $d$-dimensional hypercube.




Routing on a mesh, torus, or hypercube is trivial. On a $d$-dimensional hypercube, to get from a source bitstring $s$ to a target bitstring $t$ one only needs to fix each "wrong" bit, one at a time. There are $k!$ routes with $k$ hops.

We need to map the $d$-bit IDs to the universe $[0, 1)$. We can do so, by interpreting the bitstring $b = b_1...b_d$ as the number $\sum_{i=1}^{d} 2^{-i} b_i = (0.b_1...b_d)_2$ in binary.

There are many topologies that are similar to hypercubes, e.g. the Chord architecture. The hypercube connects every node with an ID in $[0, 1)$ with every node in exactly distance $2^{-i}$. Chord instead connects nodes with approximately distance $2^{-i}$. Many of the following examples are also derivatives of hypercubes.

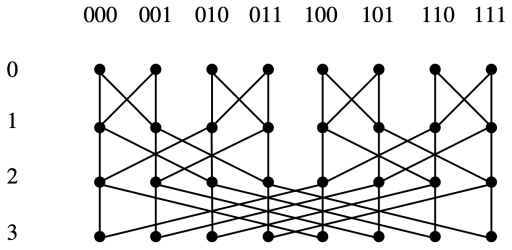
### 19.2.1 Butterfly

The $d$-dimensional butterfly $BF(d)$ is a graph with node set $V = [d+1] \times [2]^d$ and edge set $E = E_1 \cup E_2$ where:

$$E_1 = \{\{(i, \alpha), (i+1, \alpha)\} \mid i \in [d], \alpha \in [2]^d\}$$

$$E_2 = \{\{(i, \alpha), (i+1, \beta)\} \mid i \in [d], \alpha, \beta \in [2]^d, \alpha \oplus \beta = 2^i\}$$

The node set $\{(i, \alpha) \mid \alpha \in [2]^d\}$ forms the level $i$ of the butterfly.




The $d$-dimensional wrap-around butterfly $W - BF(d)$ is defined by taking the $BF(d)$ and having $(d, \alpha) = (0, \alpha)$ for all $\alpha$.

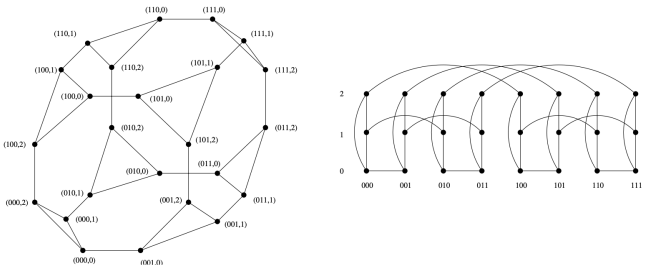
Butterflies have the advantage of a constant node degree over hypercubes, whereas hypercubes feature more fault-tolerant routing.

### 19.2.2 Cube-Connected-Cycles

The cube-connected-cycles network $CCC(d)$ is a graph with node set $V = \{(a, p) \mid a \in [2]^d, p \in [d]\}$ and edge set

$E = \{\{(a, p), (a, p+1 \mod d)\} \mid \alpha \in [2]^d, p \in [d]\} \cup \{\{(a, p), (b, p)\} \mid a, b \in [2]^d, p \in [d], |a - b| = 2^p\}$

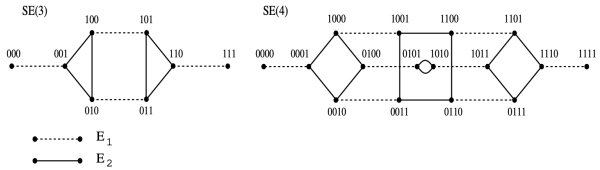
$CCC(3)$ results from the hypercube by replacing the corners by cycles. We can represent it in 2 different ways:



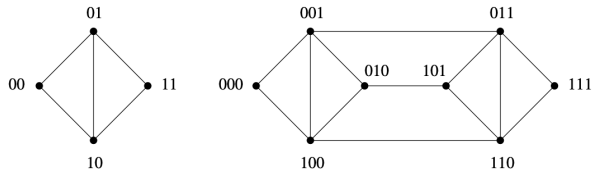
### 19.2.3 Shuffle-Exchange and DeBrujin

The shuffle-exchange and the DeBruijn network are other ways of transforming the hypercubic interconnection structure into a constant degree network.

**Shuffle-Exchange**




**DeBrujin**



### 19.2.4 Skip List

The skip list is an ordinary ordered linked list of objects, augmented with additional forward links. The ordinary linked list is the level 0 of the skip list. In addition, every object is promoted to level 1 with probability 1/2. As for level 0, all level 1 objects are connected by a linked list. In general, every object on level $i$ is promoted to the next level with probability 1/2. A special start-object points to the smallest/first object on each level.

Search, insert, and delete can be implemented in $\mathcal{O}(\log n)$ expected time in a skip list. There are obvious variants of the skip list, e.g., the skip graph.

Back to more general properties of hypercubic networks. In general, there is a trade o between degree and diameter. Every graph of maximum degree $d > 2$ and size $n$ must have a diameter of at least $\lceil \log n / \log(d-1) \rceil - 2$. In other words, constant-degree hypercubic networks feature an asymptotically optimal diameter $D$. Other hypercubic graphs manage to have a different tradeoff between node degree $d$ and diameter $D$.

## 19.3 DHT and Churn

A distributed hash table (DHT) is a distributed data structure that implements a distributed storage. A DHT should support at least (i) a search (for a key) and (ii) an insert (key, object) operation, possibly also (iii) a delete (key) operation.

A DHT can be implemented as a hypercubic overlay network with nodes having identifiers such that they span the ID space $[0, 1)$. We assume that a joining node knows a node which already belongs to the system. One way to do this is using some authority for a list of IP addresses of nodes that might be in the system.

To analyze this network against adversary, we assume that an adversary can remove and add a bounded number of nodes. It can choose which nodes to crash/join. Also, the adversary does not have to wait until the system is recovered before it crashes the next bash of nodes.

Our system is never fully repaired, but always fully functional. An adversary can add/remove at most $\mathcal{O}(\log n)$ nodes in a constant time interval. This covers repeatedly

taking nodes down in a DDoS attack. Also we assume no message delays which can be achieved using time synchronization.

**Algorithm 30:** DHT

1 Given: a globally known set of hash functions $h_i$ and a hypercube network
2 Each hypercube virtual node ("hypernode") consist of $\Theta(\log n)$ nodes
3 Nodes have connections to all other nodes of their hypernode and to nodes of their neighboring hypernodes
4 Because of churn, some of the nodes have to change to another hypernode such that up to constant factors, all hypernodes own the same number of nodes at all times
5 If the total number of nodes $n$ grows or shrinks above or below a certain threshold, the dimension of the hypercube is increased or decreased by one, respectively

Thus, each node has $\Theta(\log^2 n)$ neighbors. One can achieve $\Theta(\log n)$ with some additional effort.

The balancing of nodes can be seen as a dynamic token distribution problem on the hypercube. Each hypernode has a certain number of tokens and the goal is to distribute them along the edges s.t. all hypernodes end up with roughly the same number of tokens.

Using DHT with churn, we have a fully scalable, efficient distributed storage system which tolerates $\mathcal{O}(\log n)$ worse case joins and/or crashes per constant time interval. Nodes have $\mathcal{O}(\log n)$ overlay neighbors and search/insert take time $\mathcal{O}(\log n)$.

# 20 Eventual Consistency

How would one implement an ATM? A naive algorithm could block if a connection problem occurs. A network partition is a failure where a network splits into at least two parts that cannot communicate with each other. We look at tradeoffs between consistency, availability and partition tolerance.

## 20.1 Consistency, Availability and Partitions

A system is **consistent** if all nodes in the system agree on the current state of the system. **Availability** means that the system is operational and instantly processing incoming requests. **Partition tolerance** is the ability of a distributed system to continue operating correctly even in the presence of a network partition.

It is impossible for a distributed system to simultaneously provide consistency, availability and partition tolerance, a distributed system can fulfil two of these but not all three.

The following algorithm is both partition tolerant and available:

**Algorithm 31:** Partition tolerant, available ATM

1 **if** *bank reachable* **then**
2 | Synchronize local view of balances between ATM and bank
3 | **if** *balance of customer insufficient* **then**
4 | | ATM displays error and aborts
5 | **end**
6 **end**
7 ATM dispenses cash
8 ATM logs withdrawal for synchronization

The ATM's local view of the balances may diverge from the balances as seen by the bank, therefore consistency is no longer guaranteed.

**Eventual Consistency** - If no new updates to the shared state are issued, then eventually the system is in a quiescent state, i.e., no more messages need to be exchanged between nodes, and the shared state is consistent. Eventual consistency is a form of weak consistency. A conflict resolution mechanism is required to resolve the conflicts and allow the nodes to eventually agree on a common state.

## 20.2 Bitcoin

The Bitcoin network is a randomly connected overlay network of a few tens of thousands of individually controlled nodes.

The lack of structure is intentional: it ensures that an attacker cannot strategically position itself in the network and manipulate the information exchange. Nodes communicate via a broadcasting protocol.

Users can generate any number of private keys. From each private key a corresponding public key can be derived using arithmetic operations over a finite field. A **public key** may be used to identify the recipient of funds in Bitcoin, and the corresponding **private key** can spend these funds. The Bitcoin network collaboratively tracks the balance in bitcoins of each address.

**Bitcoin**, the currency, is an integer value that is transferred in Bitcoin transactions. This integer value is measured in **Satoshi**; 100 million Satoshi are 1 Bitcoin.

### 20.2.1 Transactions

A **transaction** is a data structure that describes the transfer of bitcoins from spenders to recipients. It consists of inputs and outputs. Outputs are tuples consisting of an amount of bitcoins and a spending condition. Inputs are references to outputs of previous transactions. Spending conditions are scripts with a variety of options, e.g. including conditions etc. An output is either spent or unspent, it can only be spent once.

The set of unspent transaction outputs (**UTXO**s) an some additional parameters are the shared state of Bitcoin. Local replicas of this state may temporarily diverge, but consistency is eventually reestablished. The inputs result in the referenced outputs spent (removed from the UTXO) and the new outputs being added to the UTXO. Transactions are broadcast and processed by every node.

**Algorithm 32:** Node Receives Transaction

```
Receive transaction t
for each input (h,i) in t do
    if output (h, i) is not in local UTXO set or
     signature invalid then
        Drop t and stop
    end
end
if sum of values of inputs < sum of values of new
 outputs then
    Drop t and stop
end
for each input (h,i) in t do
    Remove (h, i) from local UTXO set
end
for each output o in t do
    add o to local UTXO set
end
Forward t to neighbors in the Bitcoin network
```

The effect of a transaction on the state is deterministic, if all nodes receive the same set of transactions in the same order, the state across all nodes is consistent. The outputs of a transaction may assign less than the sum of inputs, in which case the difference is called the **transaction fee**. Incoming transactions are unconfirmed and are added to a pool of transactions, the **memory pool**.

### 20.2.2 Doublespend

A **doublespend** is a situation in which multiple transactions attempt to spend the same output. Only one transaction can be valid since outputs can only be spent once. When nodes accept different transactions in a doublespend, the shared state across nodes becomes inconsistent.

If doublespends are not resolved the shared state diverges. We therefore need a conflict resolution mechanism to decide which of the conflicting transactions is to be confirmed.

### 20.2.3 Proof-of-Work (PoW)

Proof-of-Work (PoW) is a mechanism that allows a party to prove to another party that a certain amount of computational resources has been utilized for a period of time.

A function $\mathcal{F}_d(c, x) \mapsto \{\text{true}, \text{false}\}$, where **difficulty** $d$ is a positive number, while **challenge** $c$ and **nonce** $x$ are bitstrings, is called PoW function if it has the following properties:

1. $\mathcal{F}_d(c, x)$ is fast to compute if $d, c, x$ are given
2. For fixed parameters $d, c$ finding $x$ s.t. $\mathcal{F}_d(c, x) =$ true is computationally difficult but feasible. The difficulty $d$ is used to adjust the time to find an $x$

The Bitcoin PoW function is given by:

$$\mathcal{F}_d(c, x) = \text{SHA256}(\text{SHA256}(c \mid x)) < \frac{2^{224}}{d}$$

This function concatenates $c$ and $x$, and hashes them twice using SHA256. There is no better algorithm known to find a nonce $x$ s.t. $\mathcal{F}_d(c, x)$ returns true rather than iterating over all $x$. Each node attempts to find a valid nonce for a node-specific challenge.

### 20.2.4 Blocks

A block is a data structure used to communicate incremental changes to the local state of a node. It consists of a list of transactions, a timestamp, a reference to a previous block and a nonce. It lists some transactions the block creator (**miner**) has accepted to its memory pool since the previous block. A node finds and broadcasts a block when it finds a valid nonce for its PoW function.

**Algorithm 33:** Node Creates (Mines) Block

```
block b_t = {coinbase_tx}
while size(b_t) ≤ 1 MB do
    Choose transaction t in the memory pool that
     is consistent with b_t and local UTXO set
    Add t to b_t
end
Nonce x = 0, difficulty d, previous block b_t1,
 timestamp = t_s
challenge c = (merkle(b_t), hash(b_{t-1}), t_s, d)
while F_d(c, x) not true do
    x = x + 1
end
Gossip block b_t
Update local UTXO set to reflect b_t
```

The function merkle creates a cryptographic representation of the set of transactions in $b_t$. With their reference to a previous block, the blocks build a tree, rooted in the so called **genesis block**. The primary goal for using the PoW mechanism is to adjust the rate at which blocks are found in the network, giving the network time to synchronize on the latest block.

Finding a block allows the finder to impose the transactions in its local memory pool to all other nodes. Upon receiving a block, all nodes roll back any local changes since the previous block and apply the new block's transactions. Transactions contained in a block are said to be confirmed by that block.

The first transaction in a block is called the **coinbase transaction**. The block's miner is rewarded for confirming transactions by allowing it to mint new coins. The coinbase transaction has a dummy input, and the sum of outputs is determined by a fixed subsidy plus the sum of the fees of transactions confirmed in the block.

### 20.2.5 Blockchain

The longest path from the genesis block to a leaf is called the **blockchain**. The blockchain acts as a consistent transaction history on which all nodes eventually agree. Only the longest path from the genesis block to a leaf is a valid transaction history, since branches may contradict each other because of doublespends.

Since only transactions in the longest path are agreed upon, miners have an incentive to append their blocks to the longest chain, thus agreeing on the current state. If multiple blocks are mined more or less concurrently, the system is said to have **forked**.

**Algorithm 34:** Node Receives Block

```
Current head b_max at height h_max
Receive block b
Connect b as a child of its parent p at height h_p + 1
if h_p + 1 > h_max and is_valid(b_t) then
    h_max = h_p + 1
    b_max = b
    Compute UTXO for the path leading to b_max
    Cleanup memory pool
end
```

Switching paths is referred to as **reorg** and may result in confirmed transactions no longer being confirmed because the blocks in the new blockchain do not include them.

Forks are eventually resolved and all nodes eventually agree on which is the longest blockchain. The system therefore guarantees eventual consistency.

The is_valid function represents the consensus rules of Bitcoin. All nodes will converge on the same shared state if and only if all nodes agree on this function.

If the set of valid transactions is expanded, we have a hard fork. If the set of valid transactions is reduced, we have a soft fork.

## 20.3 Smart Contracts

A **smart contract** is an agreement between two or more parties, encoded in such a way that the correct execution is guaranteed by the blockchain.

Contracts allow business logic to be encoded in Bitcoin transactions which mutually guarantee that an agreed upon action is performed. Scripts allow encoding complex conditions specifying who may spend the funds under what circumstances.

Bitcoin provides a mechanism to make transactions invalid until some time in the future: **timelocks**. A transaction may specify a **locktime**: the earliest time (either Unix timestamp or blockchain height) at which it may be included in a block and therefore be confirmed. Transactions with a timelock are not released into the network until the timelock expires. A node receiving the transaction needs to store it locally until the time expires, then broadcast it. A transaction $t_1$ can be replaced by another transaction $t_0$ spending some of the same outputs, if $t_0$ has a earlier timelock and thus gets broadcast in the network before $t_1$ becomes valid.

When an output can be claimed by providing a single signature it is called a **singlesig** output. In contrast the script of **multisig** outputs specifies a set of $m$ public keys and requires $k$-of-$m$ ($k \leq m$) valid signatures from distinct matching public keys from that set in order to be valid.

Most smart contracts begin with the creation of a 2-of-2 multisig output, requiring a signature from both parties.

**Algorithm 35:** Parties A and B create a 2-of-2 multisig output $o$

B sends a list $I_B$ of inputs with $c_B$ coins to A
A selects its own inputs $I_A$ with $c_A$ coins
A creates transaction

$$t_s\{[I_A, I_B], [o = c_A + c_B \to (A, B)]\}$$

A creates timelocked transaction

$$t_r\{[o], [c_A \to A, c_B \to B]\}$$

and signs it
A sends $t_s$ and $t_r$ to B
B signs both $t_s$ and $t_r$ and sends them to A
A signs $t_s$ and broadcasts it to the Bitcoin network

We call $t_s$ the **setup transaction**. It is used to lock in funds into a shared account. The fund could become unspendable if one of the parties could not collaborate to spend the multisig output. Thus, we also introduce $t_r$, the refund transaction, which guarantees that, if funds are not spend before the timelock expires, the funds are returned to the respective party.

Using this algorithm, we can implement a micropayment channel:

**Algorithm 36:** Simple Micropayment Channel from $s$ to $r$ with capacity $c$

$c_s = c,\ c_r = 0$
$s$ and $r$ create a smart contract with output $o$ with value $c$ from $s$
Create settlement transaction

$$t_f[o], [c_s \to s, c_r \to r]$$

**while** *channel open and* $c_r < c$ **do**
    In exchange for good with value $\delta$
    $c_r = c_r + \delta$
    $c_s = c_s - \delta$
    Update $t_f$ with outputs $[c_r \to r, c_s \to s]$
    $s$ signs and sends $t_f$ to $r$
**end**
$r$ signs last $t_f$ and broadcasts it

This channel can be used for rapidly adjusting micropayments from spender to recipient. Only 2 transactions are sent in the whole process, even though there may have been any number of updates to the settlement transaction, transferring more of the shared output to the recipient. The number $c$ of bitcoins used to create the channel is the maximum total that can be transferred over this channel.

At any time the recipient $r$ is guaranteed to eventually receive the bitcoins, since she holds a fully signed settlement transaction. The simple micropayment channel is intrinsically unidirectional. Since the recipient may choose any of the settlement transactions in the protocol, she will use the one with maximum payout for her.

## 20.4 Weak Consistency

Eventual consistency is only one form of weak consistency. A number of different tradeoffs between partition tolerance and consistency exist in literature.

In a system with **monotonic read consistency**, if a node $u$ has seen a particular value of an object, any subsequent accesses of $u$ will never return any older values.

Similarly, in a system with **monotonic write consistency**, a write operation by a node on a data item is completed before any successive write operation by the same node.

**Read-your-write consistency** means that after a node $u$ has updated a data item, any later reads from node $u$ will never see an older value.

The following pairs of operations are said to be **causally related**:

- Two writes by the same node to different variables.
- A read followed by a write of the same node.
- A read that returns the value of a write from any node.
- Two operations that are transitively related according to the above conditions.

A system provides **causal consistency** if operations that potentially are causally related are seen by every node of the system in the same order. Concurrent writes are not causally related, and may be seen in different orders by different nodes.

# 21 Advanced Blockchain

## 21.1 Selfish Mining

A **selfish miner** hopes to earn the reward of a larger share of blocks than its hardware would allow. The selfish miner achieves this by temporarily keeping newly found blocks secret.

**Algorithm 37:** Selfish Mining

```
Idea: Mine secretly, without immediately publishing newly found blocks
Let d_p be the depth of the public blockchain
Let d_s be the depth of the secretly mined blockchain
if a new block b_p is published then
    if d_s < d_p then
        Start mining on the newly found block b_p
    end
    if d_p = d_s then
        Publish secretly mined block b_s
        Mine on b_s and publish newly found block immediately
    end
    if d_p = d_s − 1 then
        Publish all secretly mined blocks
    end
end
```

It may be rational to mine selfishly, depending on two parameters $\alpha$ and $\gamma$, where $\alpha$ is the ratio of the mining power of the selfishly miner, and $\gamma$ is the share of the altruistic mining power the selfishly miner can reach in the network if the selfishly miner publishes a block right after seeing a newly published block. Precisely, the selfishly miner share is:

$$\frac{\alpha(1-\alpha)^2(4\alpha+\gamma(1-2\alpha))-\alpha^3}{1-(1+(2-\alpha)\alpha)}$$

If the miner is honest (altruistic), then a miner with computational share $\alpha$ should expect to find an $\alpha$ fraction of the blocks.

## 21.2 Ethereum

**Ethereum** is a distributed state machine. Unlike Bitcoin, Ethereum promises to run arbitrary computer programs in a blockchain.

Like the Bitcoin network, Ethereum consists of nodes that are connected by a random virtual network. These nodes can join or leave the network arbitrarily. There is no central coordinator. Users broadcast cryptographically signed transactions in the network. Nodes collate these transactions and decide on the ordering of transactions by putting them in a block on the Ethereum blockchain.

**Smart contracts** are programs deployed on the Ethereum blockchain that have associated storage and can execute arbitrarily complex logic. They are written in higher level programming languages like Solidity, Vyper, etc. and are compiled down to EVM (Ethereum Virtual Machine) bytecode. They cannot be changed after deployment. But most smart contracts contain mutable storage, and this storage can be used to adapt the behavior of the smart contract.

Ethereum knows two kinds of accounts. **Externally Owned Accounts (EOAs)** are controlled by individuals, with a secret key. **Contract Accounts (CAs)** are for smart contracts. CAs are not controlled by a user.

An Ethereum **transaction** is sent by a user who controls an EOA to the Ethereum network. A transaction contains:

- Nonce: This "number only used once" is simply a counter that counts how many transactions the account of the sender of the transaction has already sent.
- 160-bit address of the recipient.
- The transaction is signed by the user controlling the EOA.
- Value: The amount of Wei (the native currency of Ethereum) to transfer.
- Data: Optional data eld, which can be accessed by smart contracts.
- StartGas: A value representing the maximum amount of computation this transaction is allowed to use.
- GasPrice: How many Wei per unit of Gas the sender is paying. Miners will probably select transactions with a higher GasPrice.

There are three types of transactions:

- A **Simple Transaction** in Ethereum transfers some of the native currency, called Wei, from one EOA to another.
- A **Smart Contract Creation Transaction** whose recipient address field is set to 0 and whose data field is set to compiled EVM code is used to deploy that code as a smart contract on the Ethereum blockchain. The contract is considered deployed after it has been mined in a block and is included in the blockchain at a sufficient depth.
- A **Smart Contract Execution Transaction** that has a smart contract address in its recipient field and code to execute a speciffic function of that contract in its data field.

Smart Contracts can execute computations, store data, send Ether to other accounts or smart contracts, and invoke other smart contracts. They can also be programmed to self destruct. This is the only way to remove them again from the Ethereum blockchain. Each contract stores data in 3 separate entities: storage, memory, and stack. Of these, only the storage area is persistent between transactions.

**Gas** is the unit of an atomic computation, like swapping two variables. Complex operations use more than 1 Gas, e.g., adding two numbers costs 3 Gas.

Transactions are an all or nothing affair. If the entire transaction could not be finished within the StartGas limit, an Out-of-Gas exception is raised. The state of the blockchain is reverted back to its values before the transaction. The amount of gas consumed is not returned back to the sender.

In Ethereum, like in Bitcoin, a **block** is a collection of transactions that is considered a part of the canonical history of transactions. Among other things, a block contains: pointers to parent and up to two uncles, the hash of the root node of a trie structure populated with each transaction of the block, the hash of the root node of the state trie (after transactions have been executed).

# 22 Game Theory

In this chapter, nodes no longer have a common goal but are selfish. They are not byzantine but try to benefit from

a distributed system. Game theory attempts to mathematically capture behavior in strategic situations, in which an individual's success depends on the choices of others.

## 22.1 Prisoner's Dilemma

The following is one of the classical examples of game theory. Two prisoners $u, v$ are questioned by the police. They are both held in solitary confinement and cannot talk to each other. The prosecutors offer a bargain to each prisoner: snitch on the other prisoner to reduce your prison sentence.

| | $u$ | Player $u$ | |
|---|---|---|---|
| $v$ | | Cooperate | Defect |
| Player $v$ | Cooperate | 1<br>1 | 0<br>3 |
| | Defect | 3<br>0 | 2<br>2 |

A game requires at least two rational players, and each player can choose from at least two options (strategies). In every possible outcome (strategy profile) each player gets a certain payoff (or cost). The payoff of a player depends on the strategies of the other players.

A strategy profile is called **social optimum** (SO) if and only if it minimizes the sum of all costs (or maximizes payoff).

A strategy is **dominant** if a player is never worse off by playing this strategy. A dominant strategy profile is a strategy profile in which each player plays a dominant strategy.

A **Nash Equilibrium** (NE) is a strategy profile in which no player can improve by unilaterally (the strategies of the other players do not change) changing its strategy. A game can have multiple Nash Equilibria. If every player plays a dominant strategy, then this is by definition a Nash Equilibrium.

Nash Equilibria and dominant strategy profiles are so called solution concepts. They are used to analyze a game.

The **best response** is the best strategy given a belief about the strategy of the other players.

## 22.2 Selfish Caching

Consider computers in a network who want to access a file regularly. Each node $v \in V$ has a demand $d_v$ for the file and wants to minimize the cost for accessing it. To access it, a file can either be cached locally at cost 1 or be requested from another node $u$ with cost $c_{u \to v}$. If we interpret this game as a graph, then the cost $c_{u \to v}$ is equivalent to the length of the shortest path times the demand $d_v$.

**Algorithm 38:** Nash Equilibrium for Selfish Mining

```
S = {}
while V not empty do
    Let v be the node maximum demand d_v in V
    S = S ∪ {v}, V = V\{v}
    Remove every node u from V with c_{v→u} ≤ 1
end
```

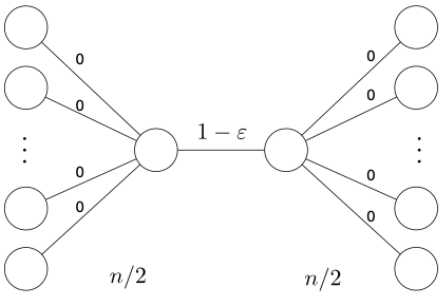
Let $NE$ denote the Nash Equilibrium with the highest cost (smallest payoff). The Price of Anarchy measures how much a distributed system degrades because of selfish nodes. The Price of Anarchy ($PoA$) is defined as:

$$PoA = \frac{\text{cost}(NE_-)}{\text{cost}(SO)}$$

Let $NE_+$ denote the Nash Equilibrium with the smallest cost (highest payoff). The Optimistic Price of Anarchy ($OPoA$) is defined as:

$$OPoA = \frac{\text{cost}(NE_+)}{\text{cost}(SO)}$$

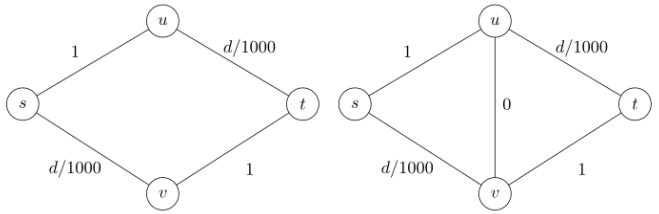
We have $PoA \geq OPoA \geq 1$. The following shows a network with a Price of Anarchy of $\Theta(n)$




The (Optimistic) Price of Anarchy of selfish caching can be $\Theta(n)$.

## 22.3 Braess' Paradox

Consider a game where cars want to travel from $s$ to $t$. Some road's cost depend on the amount of traffic going through them. There are 1000 cars. Adding a super fast road with delay 0 can increase the travel time from $s$ to $t$!




Each driver acts rationally, thus half of them takes the upper road and half of them the lower. The time for each traveler is then $1 + 500/1000 = 1.5$.

If we introduce the new road with cost 0 between $u$ and $v$, each driver now drives from $s \to v \to u \to t$, leading to a total cost of $2 > 1.5$.

## 22.4 Rock-Paper-Scissors

We will consider the classical game of rock-paper-scissors. No strategy for one player is a Nash Equilibrium: whatever $u$ chooses, $v$ can always switch its strategy s.t. $v$ wins.

| | $u$ | Player $u$ | | |
|---|---|---|---|---|
| $v$ | | Rock | Paper | Scissors |
| Player $v$ | Rock | 0<br>0 | 1<br>-1 | -1<br>1 |
| | Paper | -1<br>1 | 0<br>0 | 1<br>-1 |
| | Scissors | 1<br>-1 | -1<br>1 | 0<br>0 |

A **Mixed Nash Equilibrium** (MNE) is a strategy profile in which at least one player is playing a randomized strategy (choose strategy profiles according to probabilities), and no player can improve their expected payoff by unilaterally changing their (randomized) strategy. Every game has a mixed Nash Equilibrium.

The Nash Equilibrium of this game is if both players choose each strategy with probability 1/3.

## 22.5 Mechanism Design

Whereas game theory analyzes existing systems, there is a related area that focuses on designing games  mechanism design. The task is to create a game where nodes have an incentive to behave "nicely".

One good is sold to a group of bidders in an auction. Each bidder $v_i$ has a secret value $z_i$ for the good and tells his bid $b_i$ to the auctioneer. The auctioneer sells the good to one bidder for a price $p$.

For simplicity, we assume that no two bids are the same, and that $b_1 > b_2 > ....$