Chapter 1

Introduction

> In designing an operating system one needs both theoretical insight and horse sense. Without the former, one designs an ad hoc mess; without the latter one designs an elephant in best Carrara marble (white, perfect, and immobile).
>
> — Roger Needham and David Hartley [NH69]

Welcome to Computer Systems! This is a brand-new course, so expect surprises, enthusiasm, experimentation, and occasional glitches as we try new ideas out. We're going to be covering a wide range of topics in this course, including (in no particular order):

- The operating system kernel
- Processes and threads
- Shared-memory interprocess communication
- Message-passing, client-server communication, and remote procedure call
- Scheduling
- Virtual memory and demand paging
- Files and file systems
- Performance analysis of computer systems
- Virtualization and virtual machine monitors
- Consensus and agreement protocols
- Time synchronization
- Logical time
- Blockchains and cryptocurrencies
- Byzantine fault tolerance
- Reliable distributed storage

– and possibly others ...

Definition 1.1 (Definition)**.** . *The script for this course is going to have a lot of* ***definitions****. They are all true, but note that they are not exclusive: a definition of a term in this refers to the meaning of the term in the context of the course (see Naming, chapter 2).*

Remarks:

- On the rare occasions where we adopt a somewhat idiosyncratic definition, we'll remark on it below the definition.

Definition 1.2 (Computer Systems)**.** . ***Computer Systems*** *(sometimes shortened to just "Systems") is the field of computer science that studies the design, implementation, and behaviour of real, complete systems of hardware and software.* ***Real*** *means that anything that can happen in the real world, in a real deployment, is within the scope of study.* ***Complete*** *means the ideas we introduce have to make sense in the context of a whole system (such as your phone, or a Google datacenter, or the totality of Facebook's infrastructure).*

Remarks:

- This course is being taught by Timothy Roscoe and Roger Wattenhofer, with help from lots of assistants. We come from different background: the more theoretical and the more practical ends of computer systems, and it will probably become clear during the course which of us is which. However, this mixture is deliberate, and hopefully makes the course more interesting and fun (including for us).

- Computer Systems itself has this mixture of theory and hard pragmatism. In some cases, the subject of computer systems sometimes runs considerably ahead of the applicable theory, which comes along later to formalize existing ideas. In others, systems people borrowed or stole theoretical concepts from other fields of computer science. In a few cases, the theory and the implementation (and deployment) actually did evolve together at the same time.

- Systems is thus as much an *approach* as a subject area. It's about embracing the whole chaotic mess of reality and avoiding idealizing problems too much, but also using the right theoretical concepts to approximate reality in useful ways. This is why really good Systems people are the most highly prized technical workers (even above machine learning experts) in large tech companies and other organizations [Mic13].

- This course is also a mixture of traditional operating systems, traditional distributed systems, and a bunch of newer concepts (like blockchain). This is also deliberate: in practice, the boundaries between all these are changing right now, and at least to us it makes perfect sense to mix them up and treat them as a single subject.

1.1 Format

We'll run the lectures a little bit differently to what you may have experienced in, e.g., Systems Programming.

We will distribute a *script* before each lecture, and you should read this in advance.

For the most part, there won't be any slides. Instead, we'll explain key concepts or examples on a whiteboard or via a laptop interactively. You should turn up to the lecture with questions that you would like us to cover. This is sometimes called a "flipped classroom" model: it's intended to be highly interactive, and mostly driven by the topics you'd like to go into in detail, rather than a pre-prepared set of slides. Since the core material is covered in the script, we have the freedom to talk about what is most useful in the lectures.

You might be wondering if we have objections to you reading email, watching YouTube, using Twitter, WhatsApp or SnapChat, or perusing Instagram or Facebook during lectures.

Remarkably, people who really understand computers, psychology, and education have actually looked into this and done Real Science, randomized trials and all. The results are quite scary [HG03, Fri07, LM11, WZG+11, ARWO12, SWC13, MO14, CGW17, RUF17]. It turns out that not only does using a laptop, phone, or tablet in classes *reduce* your academic performance and grade, but there's a passive effect as well: using a mobile device in a lecture lowers the grades of *other people around you*, even if they're not using the device themselves.

We strongly recommend going offline for the 2 × 45 minutes of each lecture. If you are desperately waiting for that key Instagram photo to show up, it's now been established scientifically that it's nicer to your fellow students if you just leave the lecture or not turn up instead.

1.2 Prerequisites

This is a 3rd-year elective course in the ETH Zurich Bachelor. As a result, we assume you've completed (and passed) the following courses:

- Design of Digital Circuits
- Parallel Programming
- Systems Programming and Computer Architecture
- Formal Methods and Functional Programming
- Data Modelling and Databases
- Computer Networks

We'll build on the concepts in these courses. Given the format of the course, and the compact nature of the notes we'll distribute beforehand, it's best to be pretty familiar with this prior material.

1.3 Grading

We'll have an examination in the Prüfungsession early next year. The course grade will be based on this examination. Examinable material will be anything in the script, plus whatever is covered in these interactive sessions.

In addition, an additional quarter-point (0.25) grade can be obtained as a bonus by creating and submitting to us an original exam question including a (correct) solution and grading scheme for one of the topics in this course.

We reserve the right to use your question as inspiration in the actual exam ...

Please follow the following guidelines:

- You must submit your question before the end of semester.
- The question must be non-trivial, that is, not a pure knowledge question.
- Both the question and the solution should be as precise as possible, including how points are awarded. Consider up to 20 points for the question.
- You must come up with your own idea. Do not try to cheat – we know our exercise sheets and old exams, and we check material which obviously comes from third-party sources (other books, Wikipedia, ...). Minimal variations of such copied questions are not acceptable either.
- Your submission should be in PDF format.
- Send your exam question to Manuel Eichelberger. Please use "[CompSys Bonus] `lastname firstname`" as subject line, name your file `lastname_firstname.pdf` and indicate the topic of your question.

We'll post the deadline for your question submission about two to three weeks into the course. There will also be a second chance to submit a revised question before the end of the course – further details will follow.

1.4 Mistakes

We expect the script for the whole course to evolve somewhat over the course of the semester – we'll update it based on the discussions in the class, and also in response to any errors you find and report to us.

Please send typos, etc to the following email address:
`compsys-erratalists.inf.ethz.ch`

Bibliography

[ARWO12] Nancy Aguilar-Roca, Adrienne Williams, and Diane O'Dowd. The impact of laptop-free zones on student performance and attitudes in large lectures. *Computers & Education*, 2012.

[CGW17] Susan Payne Carter, Kyle Greenberg, and Michael S. Walker. The impact of computer usage on academic performance: Evidence from a randomized trial at the united states military academy. *Economics of Education Review*, 2017.

[Fri07] Carrie B. Fried. In-class laptop use and its effects on student learning. *Computers & Education*, 2007.

[HG03] Helene Hembrooke and Geri Gay. The laptop and the lecture: The effects of multitasking in learning environments. *Journal of Computing in Higher Education*, 2003.

[LM11] Sophie Lindquist and John McLean. Daydreaming and its correlates in an educational environment. *Learning and Individual Differences*, 2011.

[Mic13] James Mickens. The night watch. *Usenix ;login:*, pages 5–8, November 2013.

[MO14] Pam A. Mueller and Daniel M. Oppenheimer. The pen is mightier than the keyboard: Advantages of longhand over laptop note taking. *Psychological Science*, 2014.

[NH69] R. M. Needham and D. F. Hartley. Theory and practice in operating system design. In *Proceedings of the Second Symposium on Operating Systems Principles*, SOSP '69, pages 8–12, New York, NY, USA, 1969. ACM.

[RUF17] Susan Ravizza, Mitchell Uitvlugt, and Kimberly Fenn. Logged in and zoned out: How laptop internet use relates to classroom learning. *Psychological Science*, 2017.

[SWC13] Faria Sana, Tina Weston, and Nicholas J. Cepeda. Laptop multitasking hinders classroom learning for both users and nearby peers. *Computers & Education*, 2013.

[WZG+11] Eileen Wood, Lucia Zivcakova, Petrice Gentile, Karin Archer, Domenica De Pasquale, and Amanda Nosko. Examining the impact of off-task multi-tasking with technology on real-time classroom learning. *Computers & Education*, 2011.

Chapter 2

Naming

Naming is fundamental to how we construct computer systems, in that naming issues (and naming decisions) pervade the design at all levels. The general principles of systematic naming and binding of objects in computer systems are well-established, but a surprising number of engineers are unaware of them.

That's the reason we start with the somewhat abstract concept of naming, before getting on to more obviously "systemsy" topics later on.

2.1 Basic definitions

Definition 2.1 (Name)**.** *A **name** is an indentifier used to refer to an object in a system.*

Remarks:

- This is an intentionally broad definition: a name might be a character string, or a string of bits – in principle any concrete value. For example, 64-bit virtual addresses are names that refer to data stored in memory pages. Part of the point we're trying to make in this chapter is that the same principles apply across scenarios as diverse as virtual addresses and web pages.

Definition 2.2 (Binding, Context, Namespace, Name Scope, Resolution)**.** *A **binding** is an association of a name to an object. A **context** is a particular set of bindings, and can also be referred to as a **namespace** or **name scope**. **Resolution** is the process of, given a name and a context, finding the object to which the name is bound in that context.*

Remarks:

- A binding – the association of a name to an object –is a separate thing from both the name and the object.
- For a name to designate an object, it must be bound to the object *in some context*. In other words, whenever names are being used, there is *always* a context, even if it's implicit. Many misunderstandings result from failing to recognize the context being used to resolve names in a particular situation.

Definition 2.3 (Catalog, Dictionary, Directory)**.** *A **catalog**, or **dictionary**, or **directory**, is an object which implements a table of bindings between names and objects – in other words, it is an object which acts as a context.*

2.2 Naming networks

Definition 2.4 (Naming Network, Pathname, Path Component)**.** *A **naming network** is a directed graph whose nodes are either naming contexts or objects, and whose arcs are bindings of names in one context to another context. A **pathname** is an ordered list of names (called **path components**) which therefore specify a path in the network.*

Definition 2.5 (Naming Hierarchy, Tree name, Root)**.** *A naming network which is a tree is called a **Naming hierarchy**. Pathnames in a naming hierarchy are sometimes called **tree names**. The unique starting point in a naming hierarchy is called the **root**.*

Example 2.6 (UNIX name resolution)**.** *. A* UNIX *filename is a pathname, but there are a few wrinkles. The* UNIX *file system is, for the most part, a naming hierarchy of directories which contain references to other directories or files.*

A UNIX *filename which starts with "/", such as `/usr/bin/dc`, is resolved using the "root" of the file system as the first context.*

Alternatively, a filename which doesn't start with a "/" is resolved using the current working directory as the first context.

Every context (i.e. directory) in the UNIX *file system has a binding of the name "`.`" to the context itself.*

Each context also has a binding of the name "`..`". This is is always bound to the "parent" directory of the context. The root directory binds "`..`" to itself.

*As we shall see later, a **file** object can have than one name bound to it, but a **directory** cannot. This that the naming network is not quite a tree, but a directed acyclic graph.*

2.3 Indirect entries and symbolic links

By this stage, you may be wondering where "symbolic links" or "shortcuts" fit it. A symbolic link is a UNIX file system construct (created using `ln -s`) which creates a kind of alias for a name; the equivalent facility in Windows is called a shortcut. They are both examples of:

Definition 2.7 (Indirect entry)**.** *An **indirect entry** is a name which is bound not to an object per se, but instead to another path name. The path name is resolved relative to the same context as the one in which the indirect entry is bound.*

Remarks:

- We'll see more about indirect entries later in the chapter on file systems.

- Indirect entries arbitrarily created by unprivileged users (Unix symbolic links, for example) can complicate name resolution since there is no guarantee that the binding will end up at an object at all, or if the name to which the indirect entry is bound is even syntactically valid in the context in which it is to be resolved.

2.4 Pure names and addresses

Definition 2.8 (Pure name)**.** *A **pure name** encodes no useful information about whatever object it refers to [Nee89].*

Remarks:

- The names we have considered so far are arguably pure names – in particular, the only thing we have done with them is to bind them to an object in a context, and look them up in a context to get the object again. Some people use the word "identifier" to mean a pure name.

- One might argue that Unix filenames are not pure, since `cv.pdf` surely states that the file format is PDF, but this isn't strictly true – it's just a convention in Unix at least.

Definition 2.9 (Address)**.** *An **address** is a name which encodes some information about the location of the object it refers to.*

Remarks:

- An IP address is not a pure name: it can be used to route a packet to the corresponding network interface without you needing to know anything more about the interface.

- In contrast, an Ethernet 802.11 MAC address (despite the terminology) *is* a pure name: it doesn't say anything about where to find the NIC and all you can do is look it up in a switch's routing table (we'll ignore the fact that a MAC address encodes the vendor in the first 3 octets).

2.5 Search paths

Definition 2.10 (Search path)**.** *A **search path** is an ordered list of contexts which are treated as a single context; to look up a name in such a context, each constituent context is tried in turn until a binding for the name is found, and which point resolution terminates.*

Remarks:

- The UNIX shell `PATH` variable, for example:

  `/home/troscoe/bin:/usr/bin:/bin:/sbin:/usr/sbin:/etc`

  – is a search path context for resolving command names: each directory in the search path is tried in turn to looking for a command.

- Search paths are a bit of a hack really, compared with the elegance of the naming networks we have considered so far. They certainly confuse questions like "what does it mean to remove a binding" – if I remove a name from a search path context, what actually happens? Is the first binding removed? Or all the bindings? Both are not necessarily desirable (and might surprise an unsuspecting user). Also, if the name being looked up is actually a path name, what happens? Is the search path searched for the first path component, or the full name? In general, it's best to avoid them if you possibly can.

2.6 Synonyms and Homonyms

Definition 2.11 (Synonyms, Homonyms)**.** ***Synonyms*** *are different names which ultimately resolve to the same object.* ***Homonyms*** *are bindings of the same name to different objects.*

Remarks:

- The existence of synonyms turns a naming hierarchy into a directed (possibly cyclic) graph. For this reason, the UNIX file system permits cycles by preventing directories from having synonyms (only inodes).

- It should be clear by now that comparing two objects for identity (i.e. are they the same object?) is different from comparing their names. Homonyms are another facet of this: names and objects really are completely separate entities (as are bindings).

Chapter Notes

Jerry Saltzer was one of the first to try to write down naming as a set of abstract principles [Sal78], drawing on experience in the Multics system [Org72], which was one of the first system design to take a highly principled approach to naming and protection. Saltzer's case studies in the above are paged virtual memory, programming languages (in particular linkage, which is all about names) and filing systems. It's a long and subtle read, going into more depth than we can here, but is worth reading despite its age and still relevant today.

So-called "union mounts" in a file system are also a search path context. Plan 9 from Bell Labs introduced union mounts as an attempt to come up with a cleaner solution to command line binaries than the UNIX `PATH` variable [PPT+93].

The design of good naming schemes, even those based on textual names, is fraught with subtle difficulties [PW85]. Simpler schemes are usually better.

Saltzer also addressed the design of naming schemes in reference to the early Internet architecture in RFC 1498 [Sal93]. It is interesting to judge the Internet's design against Saltzer's principles.

Naming is fundamental to computer science, and is surprisingly subtle. Throughout the rest of this course, you'll encounter a variety of naming schemes and resolution algorithms.

It is important to remember, however, that in human society naming is much richer than the rather precise and limited definition we use in systems - it's deeply cultural and political. For a glimpse of how complex it can get see, for example, Umberto Eco [Eco76].

Bibliography

[Eco76] Umberto Eco. *A Theory of Semiotics.* Advances in Semiotics. Indiana University Press, Bloomington, Indiana, USA, 1976.

[Nee89] R. M. Needham. Distributed systems. chapter Names, pages 89–101. ACM, New York, NY, USA, 1989.

[Org72] Elliott I. Organick. *The Multics System: An Examination of Its Structure.* MIT Press, Cambridge, MA, USA, 1972.

[PPT+93] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. *SIGOPS Oper. Syst. Rev.*, 27(2):72–76, April 1993.

[PW85] Rob Pike and Peter Weinberger. The hideous name. In *Proceedings of the USENIX Summer 1985 Conference*, Portland, OR, 1985.

[Sal78] Jerome H. Saltzer. Naming and binding of objects. In *Operating Systems, An Advanced Course*, pages 99–208, London, UK, UK, 1978. Springer-Verlag.

[Sal93] J. Saltzer. On the Naming and Binding of Network Destinations. RFC 1498 (Informational), August 1993.

Chapter 3

Classical Operating Systems and the Kernel

This course is about Computer Systems, and specifically systems software. It includes material that would traditionally be split between a course on operating systems, and a different one on distributed systems.

Treating these as a single topic is, we think, a much more accurate picture of reality. A good illustration of this is a basic definition of what an operating system actually is.

3.1 The role of the OS

Anderson and Dahlin [AD14], roughly speaking, define an OS *functionally*:

Definition 3.1 (Operating System)**.** . *The* ***operating system*** *for a unit of computing hardware is that part of the software running on the machine which fulfils three particular roles: Referee, Illusionist, and Glue.*

- *As a* ***Referee****, the OS multiplexes the hardware of the machine (CPU cores, memory, devices, etc.) among different principals (users, programs, or something else), and protects these princpals from each other: preventing them from reading or writing each others' data (memory, files, etc.) and using each others' resources (CPU time, space, memory, etc.). The OS is basically a* ***resource manager*** *whose goal is to provide some kind of fairness, efficiency, and/or predictability.*

- *As an* ***Illusionist****, the OS provides the illusion of "real" hardware resources to resource principals through* ***virtualization****. These virtual resources may resemble the underlying physical resources, but frequently look very different.*

- *Finally, as* ***Glue****, the OS provides abstractions to tie different resources together, communications functionality to allow principals to exchange data with each other and synchronise, and hide details of the hardware to allow programs portability across different platforms.*

Remarks:

- Most older OS textbooks (which are not recommended for this course) do claim that the OS essentially provides multiplexing, protection, communication, and abstraction, but then go on to define it ontologically as a kernel, libraries, and daemons – we look at these components below, but only in the context of an OS for a single, simplified machine.

- Note, however, that this definition is pretty broad in what the hardware actually *is*. It might be a small Raspberry PI, or a robot, or a phone, laptop, a tightly coupled cluster of PC-based servers in a rack, or an entire datacenter – the definition still makes sense in all these cases. That's why we're not separating operating systems and distributed systems in this course.

- In this course, we'll cover many of these scales. They have different properties; for example, whether part of the machine can fail when the rest keeps going (as in a datacenter or rackscale machine) or it all fails together (as in a phone or PC). A good way to make sense of this is the concept of a *domain*.

3.2 Domains

Domains are useful concepts for thinking about computing systems at different scales.

Definition 3.2 (Domain)**.** *. A **domain** is a collection of resources or principals which can be treated as a single uniform unit with respect to some particular property (depending on the domain). Some examples:*

- ***NUMA domain**: cores sharing a group of memory controllers in a NUMA system.*
- ***Coherence domain** or **coherence island**: caches which maintain coherence between themselves.*
- ***Failure domain** or **unit of failure**: the collection of computing resources which are assumed to fail together.*
- ***Shared memory domain**: cores which share the same physical address space (not necessarily cache coherent)*
- ***Administrative domain**: resources which are all managed by a single central authority or organization.*
- ***Trust domain**: a collection of resources which mutually trust each other.*
- ***Protection domain**: the set of objects which are all accessible to a particular security principal.*
- ***Scheduling domain**: set of processes or threads which are scheduled as a single unit.*

Remarks:

- Each topic we will cover is going to be tied to some combination of domain boundaries. Sometimes we'll mention which ones, but it's always good to figure out which ones are in play.
- Domains often correspond to the assumptions the software can make inside a domain, or the functionality the software must provide between domains. Domains also often can be expressed as invariants or uniform properties of the units inside each domain.

3.3 OS components

A *classical* mainstream OS (like UNIX or Windows) operates on a single machine. That's a single coherence domain, a single unit of failure, a single administrative domain (hence hierarchical authority), and a single resource allocation domain (centralized resource control).

Such an OS consists of the kernel, libraries, and daemons.

Definition 3.3 (Kernel). *The* ***kernel*** *is that part of an OS which executes in privileged mode. Historically, it has also been called the* ***nucleus****, the* ***nub****, the* ***supervisor****, and a variety of other names.*

Remarks:

- The kernel is a large part of OS in UNIX and Windows, less so in L4, Barrelfish, etc.
- While most computer systems have a kernel, very small embedded systems do not.
- The kernel is just a (special) computer program, typically an event-driven server. It responds to multiple entry points: system calls, hardware interrupts, program traps. It can also have its own long-running threads, but not always.

Definition 3.4 (System libraries). ***System Libraries*** *are libraries which are there to support all programs running on the system, performing either common low-level functions or providing a clean interface to the kernel and daemons.*

Remarks:

- The standard C library is a good example: it provides convenience functions like `strncmp()`, essential facilities like `malloc()`, and interfaces to the kernel (system calls like `open()` and `sbrk()`.

Definition 3.5 (Daemon). *A* ***daemon*** *is a user-space process running as part of the operating system.*

Remarks:

- Daemons are different from in-kernel threads. They execute OS functionality that can't be in a library (since it encapsulates some privilege), but is better off outside the kernel (for reasons of modularity, fault tolerance, and ease of scheduling).

3.4 Operating System models

There are a number of different architectural models for operating systems. They are all "ideals": any real OS is at best an approximation. Nevertheless, they are useful for classifying different OS designs.

Definition 3.6 (Monolithic kernel)**.** *A* ***monolithic kernel****-based OS implements most of the operating systems functionality inside the kernel.*

Remarks:

- This is the best-known model, and UNIX adheres closely to it.
- It can be efficient, since almost all the functionality runs in a single, privileged address space.
- Containing faults (software or hardware bugs) in a monolithic kernel is hard. Recent evidence suggests that this does result in reduced reliability.
- Monolithic kernels tend to have many threads which execute entirely in kernel mode, in addition to user-space application processes and daemons.

Definition 3.7 (Microkernel)**.** *A* ***microkernel****-based OS implements minimal functionality in the kernel, typically only memory protection, context switching, and inter-process communication. All other OS functionality is moved out into user-space server processes.*

Remarks:

- Microkernels use process boundaries to modularize OS functionality. Device drivers, file systems, pagers, all run as separate processes which communicate with each other and applications.
- The motivation for microkernels is to make the OS more robust to bugs and failures, and make it easier to structure and evolve the OS over time since dependencies between components are in theory more controlled.
- Microkernels can be slower since more kernel-mode transitions are needed to achieve any particular result, increasing overhead. However, the very small size of microkernels can actually improve performance due to much better cache locality. The debate remains controversial.
- Examples of contemporary microkernels include Minix and the L4 family. There is a myth that microkernels have never been successful. If you use a phone with a Qualcomm processor, you're running L4. If you use an Intel machine with a management engine, you're running Minix. That's well over a billion deployed units right there.

Definition 3.8 (Exokernel)**.** *In contrast to microkernels,* ***exokernel****-based systems move as much functionality as possible out of the kernel into system libraries linked into each application.*

Remarks:

- Moving OS functionality into application-linked libraries is, at first sight, an odd idea, but greatly simplifies reasoning about security in the system and providing performance guarantees to applications which also need to invoke OS functionality.
- Exokernel designs came out of academic research (MIT and Cambridge), but found their biggest impact in *virtual machine monitors*: VMware ESX Server and Xen are both examples of exokernels.

Definition 3.9 (Multikernel)**.** *A **multikernel**-based system targets multiprocessor machines, and runs different kernels (or different copies of the same kernel) on different cores in the system. Each kernel can be structured as a monolithic, micro- or exo-kernel.*

Remarks:

- Multikernels are a relatively new idea (although, if you look back in history, several old systems look like this as well).
- The individual kernels can be anything (or a mixture); people have built multikernels using exokernels (e.g. Barrelfish) or complete monolithic systems (e.g. Popcorn Linux). The key characteristic is the kernels themselves do not share memory or state, but communicate via messages.
- They reflect modern trends in hardware: heterogeneous processors (making a single kernel program impossible), increasingly networked interconnects (meaning programmers already worry about inter-core communication explicitly), and increasingly low-latency inter-machine networks (in other words, we increasingly need to consider a rack of servers or an entire datacenter as a single machine). Indeed, a theme of this course is that the boundary between "operating systems" and "distributed systems", always problematic, is now basically meaningless. They are the same topic.

3.5 Bootstrap

Definition 3.10 (Bootstrap)**.** *. **Bootstrapping**, or more commonly these days simply **booting**, is the process of starting the operating system when the machine is powered on or reset up to the point where it is running regular processes.*

Remarks:

- When a processor is reset or first powered on, it starts executing instructions at a fixed address in memory. A computer design must ensure that the memory at this address holds the right code to start the machine up in a controlled manner.
- While this code could be the initializing code for the kernel, in practice it is another program which sets up the hardware. This can be highly

complex process for two reasons: firstly, modern processors and memory systems are incredibly complicated these days, and there is a lot to do (potentially hundreds of thousands of lines of code) to execute before the OS itself can be loaded. This even includes starting memory controllers: this initialization code often runs only out of cache and initializes the DRAM controllers itself. This code is sometimes called the *Basic Input/Ouput System (BIOS)*, or its functionality is standardized as the *Unified Extensible Firmware Interface (UEFI)*. The BIOS is generally built into the machine as *Read-Only Memory (ROM)*.

- The BIOS also sets up a standard execution environment for the next program to run, so that it doesn't need to know at compile time the precise devices that the computer has (storage devices, network interfaces, how much memory, etc.). This next program can therefore be the same across a wide range of computers.

- The next program is typically the *boot loader*, and its job is to find the operating system kernel itself, load it into memory, and start it executing.

- The OS kernel itself, once it is entered, initializes its own data structures and creates the first process (known as `init` in traditional UNIX). Finally, it starts this new process executing, and the system is now in a regular steady state.

3.6 Entering and leaving the kernel

Definition 3.11 (Mode transfer)**.** . ***Mode transfer*** *is the process of software execution transitioning between different hardware processor modes.*

Remarks:

- This typically involves switching between user mode (running regular programs and daemons) and kernel mode (running the kernel). However, other modes are possible (such as with virtual machine support).

- The key goal of user to kernel mode transfer is to protect the kernel from a malicious or buggy user process.

- The kernel is entered from userspace as a result of a procesor exception: either a synchronous *trap* or an asynchrounous *fault* (as we saw in the Systems Programming course).

- Mode transfer to the kernel cannot be done via any kind of jump instruction: we cannot allow a user program to start executing at an arbitrary point in the kernel. Instead, as with faults, when a user program executes a system call the processor starts executing in the kernel at a fixed address, and then kernel code has to figure out the reason for the system call based on the call's *arguments*.

- In contrast, transfering from kernel to user mode happens under different circumstances: returning from a system call, creating a new process, or switching to different process after the current process has been interrupted.

Definition 3.12 (Execution state)**.** . *The **execution state** of a process consists of the current values of user mode processor registers, stack pointer, program counter, and other state such as a page table base address.*

Remarks:

- When entering the kernel from user mode, the kernel must save the execution state of the currently running process so it can resume it later.
- The most common way of creating a new process is to create this "saved" execution state, and then "resume" it as if it had previously entered the kernel.
- There is one exception to this "resume" model of process control, which we will say later when we look at communication primitives: the *upcall*.

Definition 3.13 (System call)**.** . *A **system call** is a trap (synchronous exception) deliberately invoked by a user program to request a service from the kernel. A system call is defined as taking arguments and returning values.*

Example 3.14 (UNIX `write()`)**.** *Consider the `write()` system call in* UNIX *(type "`man 2 write`" for more documentation).*

`write()` has the following functional prototype:

```
ssize_t write(int fd, const void *buf, size_t count);
```

Such a call would be implemented as follows:

Algorithm 3.15 System call for `write( fd, buf, count )`

Procedure in user space

1: Load `fd`, `buf`, and `count` into processor registers
2: Load system call number for `write` into a register
3: Trap
4: Read result from a register
5: **return** Result

Execution in the kernel (the trap handler)

6: Set up execution environent (stack, etc.)
7: Read system call number from register
8: Jump to `write` code based on this
9: Read `fd`, `buf`, and `count` from processor registers
10: Check `buf` and `count` for validity
11: Copy `count` bytes from user memory into kernel buffer
12: Do the rest of the code for `write`
13: Load the return value into a register
14: Resume the calling process, transfer to user mode

Remarks:

- This is very similar to a *Remote Procedure Call*, which we will see later.
- The code sequences in user space and kernel that *marshal* the arguments and return values are called *stubs*.
- In traditional operating systems, this code is mostly written by hand. However, it requires care to get right, and the consequences of a bug here can result in corruption of the kernel, or worse.

Bibliography

[AD14] Thomas Anderson and Michael Dahlin. *Operating Systems: Principles and Practice.* Recursive Books, 2nd edition, 2014.

Chapter 4

Processes

The process is a fundamental concept in operating systems. In this chapter we look at what a process is, the concept of an execution environment, how processes are created and destroyed, and how they interact with threads. In the next chapter we'll look at scheduling, which is the process of picking which process to run.

4.1 Basic definitions

Definition 4.1 (Process)**.** *A* ***process*** *is the execution of a program on a computer with restricted rights.*

Remarks:

- A process can be thought of as an *instance* of a program – there can be multiple processes in a computer executing different instances of the same program.

- A process combines *execution* – running the program – and *protection*. An operating system protects processes (and their data and resources) from other processes in the computer.

- A process is a resource *principal*: In terms of implementation, a process bundles a set of hardware and software resources together: some memory, some CPU cycles, file descriptors, etc.

- A process can be thought of as a *virtualized* machine executing a program, albeit a machine very different from the underlying hardware.

- Processes are typically named in the system using *process identifiers*, or *PIDs*.

- The complete software of a running computer system can be thought as the set of running processes plus the kernel.

4.2 Execution environment

Definition 4.2 (Execution environment)**.** *The* ***execution environment*** *of a process is the (virtual) platform on which is executes: the virtual address space, available system calls, etc.*

Remarks:

- The execution environment of the kernel is purely defined by the machine hardware (in the absence of virtualization). In contrast, in a process it is defined by the user-mode processor architecture plus whatever the kernel chooses to present to the process, for example the virtual address space

- The set of system calls constitutes the API to the kernel, but can also be thought of as *extensions* to the set of machine instructions the process can execute. A process can be thought of (and older texts on operating systems talk about this much more) as a *virtual machine* for executing the user's program. This machine has a processor (or more than one processor, if the program is multi-threaded), a *lot* of memory (due to paging), etc.

- The process's *virtual processor* (or processors) doesn't have a simple relationship to the real processors that the OS kernel is managing. However, in Unix and Windows, the process appears to have one or more virtual processors exclusively to itself. In practice, the OS is constantly preempting the process, running other processes, handling interrupts, etc. Because the process is always resumed exactly where it left off when the kernel was entered, the program running in the process behaves *as if* it had the processor entirely to itself and nothing else happened.

- Some OS designs go beyond this "resume" model. Unix also has *signals* (which we will see in a later chapter) that are analogous to hardware interrupts, except that they are generated by the kernel (sometimes on behalf of another process) and delivered to the process in user space as an *upcall*.

- Other OSes are more radical: instead of resuming a process after the kernel has been entered, they always jump back into the process at a single, fixed address, and let the process' own code at that address figure out how to resume from where it was. This mechanism is called *scheduler activations*, and they even available in new versions of Windows.

4.3 Process creation

How is a process created? There are a number of ways operating systems create processes, but they boil down to one of two models.

Definition 4.3 (Child, Parent)**.** *When a process creates (via the OS) another new process, the creating process is called the **parent** and the newly created process the **child**.*

*This creates a **process tree**: every process in the system has a parent (except one, the root of the tree).*

Definition 4.4 (Spawn)**.** *An OS **spawns** a child process by creating it from scratch in a single operation, with a program specified by the parent.*

Remarks:

- Unless you're familiar with `fork()` (below), this is the obvious way to create a process.
- Windows creates processes by spawning using the `CreateProcess()` system call.
- The spawn operation has to explicitly specify *everything* about a process that needs to be fixed before it starts: what program it will run, with which arguments, what protection rights it will have, etc. This can be quite complex; on Windows `CreateProcess()` takes 10 arguments, two of which are pointers to more (optional) arguments.

Definition 4.5 (Fork)**.** *In* UNIX*, a **fork** operation creates a new child process as an exact copy of the calling parent.*

Remarks:

- Since the child is an exact copy of the parent, one process calls `fork()`, but both parent and child return from it.
- The only difference between the parent and child processes is the return value from `fork()`: the parent gets the PID of the child it just created (or `-1` upon error). In the child, the same invocation of `fork()` returns `0`.
- In contrast to spawn, fork doesn't need any arguments at all: the OS doesn't need to know anything about the child except that it's an exact copy of the parent.
- On the face of it, `fork()` is expensive: it involves copying the entire address space of the parent. This is, indeed, what it used to do, but in the chapter on virtual memory we will encounter a technique called *copy-on-write* which makes `fork()` much cheaper (though not free).

Definition 4.6 (Exec)**.** *In* UNIX*, an **exec** operation replaces the contents of the calling process with a new program, specified as a set of command-line arguments.*

Remarks:

- `exec()` does *not* create a new process, instead it is the complement to `fork()` – without it, you could not run any new programs.
- `exec()` never returns (except if it fails), instead the new program starts where you might expect in `main()`.
- Splitting process creation in UNIX into `fork` and `exec` allows the programmer to change whatever features of the child process (e.g. protection rights, open files, etc.) themselves in the code between `fork()` returning and `exec()` being called, rather than having to specify all of this to a spawn call.

Definition 4.7 (The initial process)**.** *The **initial process**, often called `init`, is the first process to run as a program when a machine boots.*

Remarks:

- On UNIX, the first process naturally cannot be `fork`ed. Instead, it is constructed by the kernel, and given the PID of 0. PID 0 never runs a user program; in older UNIX versions it was called `swapper`. Instead, PID 0 calls the kernel entry point to `fork` to create `init`, which therefore has a PID of 1.

4.4 Process life cycle

What happens to a process after it has been created?

Definition 4.8 (Process states)**.** *Each process is said to be one of a set of states at any point in time. **Running** processes are actually executing code, either in kernel mode or in user space. **Runnable** (also called **waiting** or **ready**) processes can execute, but are not currently doing so. **Blocked** (also called **asleep**) processes are waiting for an event to occur (such as an I/O operation to finish, or a page fault to be serviced) before they can run.*

Example 4.10 (UNIX process lifecycle)**.** *Figure 4.9 shows a slightly simplified process state machine for* UNIX*. A running process moves between kernel and user space, but can stop running due to either blocking (being put to sleep) or some other process running instead (preemption).*

Remarks:

- A process which executes a blocking system call (like `read()` or `recv()`) enters the *asleep* state until the operation is ready to complete.
- A process running in user space must always first enter the kernel (via a system call, or an asynchronous trap) before it can change to another state.
- A process which exits (either voluntarily or otherwise) becomes a *zombie.*

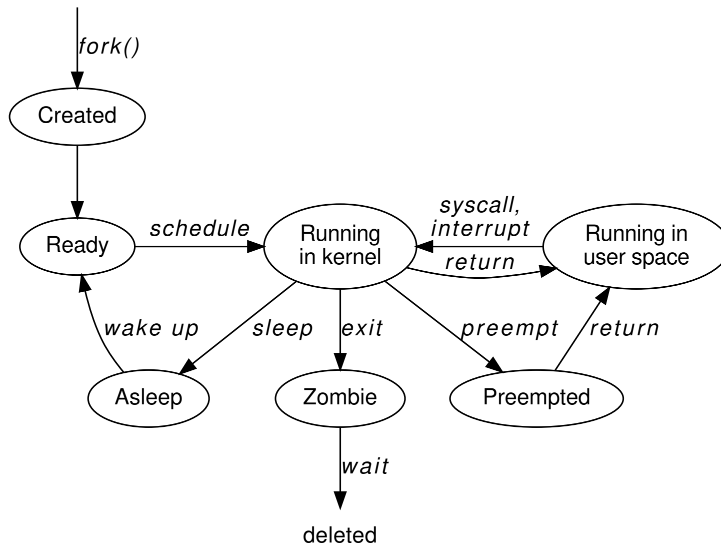


Figure 4.9: Simplified lifecycle of a UNIX process

Definition 4.11 (Zombie). *A process which exits is generally not removed completely from the system, but enters a state between being alive and being deleted from the OS. A process in this state is a* ***zombie****.*

Remarks:

- Zombies solve a problem. If a process which exited is completely deleted from the system, there is no record of it left. This means that the process' parent cannot determine anything about it (such as its exit code). Instead, the exiting process hangs around until its parent asks for its exit status, and can then be safely deleted.
- In UNIX, the operation for reading a process' exit status (and thus deleting it) is called `wait()`.

Definition 4.12 (Orphan). *A process which is alive, but whose parent has exited, is an* ***orphan****.*

Remarks:

- In UNIX, orphans are adopted not by the parent's parent, but by *init*, i.e. process ID 1.
- A key role of the init process is to "reap" orphan processes by calling `wait`.

4.5 Coroutines

Before we get into threads, it's worth pointing out the variety of different ways of expressing concurrency and parallelism.

Definition 4.13 (Coroutines)**.** *A* ***coroutine*** *is a generalization of the concept of a subroutine. A coroutine can be entered at multiple times, at multiple points, and return multiple times. Programming with coroutines is sometimes referred to as* ***cooperative multitasking****.*

Example 4.14. *The classic simple use-case of coroutines is a pair of activities, one of which processes data created by the other:*

Algorithm 4.15 Test-And-Set

```
1: inputs
2:   q {A queue data structure}
3: coroutine producer:
4:   loop
5:     while q not full do
6:       i ← new item
7:       q.insert(i)
8:     end while
9:     yield to consume
10:  end loop
11: end coroutine
12: coroutine consume:
13:  loop
14:    while q not empty do
15:      i ← q.remove()
16:      process(i)
17:    end while
18:    yield to produce
19:  end loop
20: end coroutine
```

This requires no threads, merely a different form of control flow to what you may be used to. Coroutines are an old idea, and express *concurrency* (more than one thing going on at the same time) without *parallelism* (actual simultaneous execution of multiple tasks). They are the basis not only for subroutines (which are strictly nested coroutines) but also iterators, which you might have seen in C++ or Python, for example. However, coroutines are more general than these.

4.6 Threads

Early multiprocessing operating systems (like the original UNIX) did not provide threads: each process was single-threaded. This was not a problem when most computers were single-processor machines and programming languages were relatively low-level. However, the model prevents a programmer from using threads as language abstractions to express concurrency in her program (as coroutines can), and also prevents her from exploiting parallel hardware such as multiple cores.

Definition 4.16 (User threads)**.** *.* ***User threads*** *are implemented entirely within a user process (as a library or as part of the language runtime). They*

are sometimes known as ***lightweight processes****, but this latter term is a bit ambiguous.*

Remarks:

- On a multiprocessor, user threads can be multiplex across multiple kernel threads (see below) to give a process true parallelism.
- Since user threads are context switched entirely in user space, they can be very fast.
- User threads can implement a "directed yield" to another thread, providing most of the functionality of coroutines.
- If a user thread is about to block (for example, when it is about to execute a system call which might block the whole process), it is typical for the thread library to intercept the call and turn it into a *non-blocking* variant so another user thread in the same process can run while the system call is serviced – otherwise, performance is severely impacted. Indeed, one use for user-level threads is as a convenient programming abstraction above non-blocking I/O primitives.
- This trick of intercepting blocking calls does not work, however, for unintentional synchronous processor exceptions, in an particular page faults. A page fault on one user-level thread will block the entire process, since the kernel has no way of scheduling one of the other threads (or indeed being aware of them).

Where the OS has no support for threads whatsoever, user threads are an attractive option. However, most mainstream OSes today provide threads in the kernel.

Definition 4.17 (Kernel threads)**.** *.* ***Kernel threads*** *are implemented by the OS kernel directly, and appear as different virtual processors to the user process.*

Remarks:

- This is the default model in, for example, Linux.
- Each thread is now scheduled by the kernel itself, which keeps track of which threads are part of which process.
- Whereas a process used to be a thread of execution in a virtual address space, in this model it becomes a non-empty set of threads which share a virtual address space and which might also be scheduled intelligently together.
- Each kernel thread can now block (including on page faults) without stopping other threads in the same process.
- The kernel is now more complicated, since it has to track the relation between threads, address spaces, and processes.
- Thread creation, and context-switching between threads, is slower since it requires the kernel to be entered.

Chapter Notes

Processes have been around for a long time; they were a well-established idea when UNIX was created [RT73].

Scheduler activations were described as such by Anderson (he of the textbook) and others [ABLL92], though the Psyche operating system (presented in the same session of the same conference!) also used upcall-based dispatch [MSLM91], and the basic idea may have appeared earlier in IBM mainframes. While microkernel-based operating systems [Lie93, Lie95] tend to provide just threads and address spaces, exokernel-based systems [EKO95, LMB+96], tend to use upcalls since thread-management is pushed into user space anyway, and the basic philosophy is to expose as much as possible to the user process (or its libraries, at least) [EK95]. This also is a natural fit for virtual machine monitors [BDGR97, BDR+12], since the upcall is delivered as a "hardware" interrupt to the guest operating system.

Bibliography

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, February 1992.

[BDGR97] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997.

[BDR+12] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Trans. Comput. Syst.*, 30(4):12:1–12:51, November 2012.

[EK95] D. R. Engler and M. F. Kaashoek. Exterminate all operating system abstractions. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, HOTOS '95, pages 78–, Washington, DC, USA, 1995. IEEE Computer Society.

[EKO95] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[Lie93] Jochen Liedtke. Improving ipc by kernel design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, New York, NY, USA, 1993. ACM.

[Lie95] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[LMB+96] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–129, September 1996.

[MSLM91] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 110–121, New York, NY, USA, 1991. ACM.

[RT73] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *SIGOPS Oper. Syst. Rev.*, 7(4):27–, January 1973.

Chapter 5

Inter-process communication

5.1 Hardware support for synchronization

You should have already seen a number of hardware mechanisms for synchronizing threads. The most basic one available to an OS is to disable interrupts:

Algorithm 5.1 Protecting a critical section by disabling interrupts

1: Disable all interrupts and traps
2: Access state in a critical section
3: Enable interrupts

Remarks:

- This technique doesn't work in a situation with multiple cores or hardware threads running concurrently. Neither does it take into account DMA devices.

- Processes can't be rescheduled inside critical section. Indeed, this is effectively a mutex on the entire state of the machine.

- That said, inside the kernel on a uniprocessor it is extremely efficient for short critical sections.

To provide synchronization on a multiprocessor in user space, you need help from the memory system and instruction set.

5.1.1 Shared-memory synchronization instructions

Most of this section should be a recap from the Systems Programming and Parallel Programming courses.

Algorithm 5.2 Test-And-Set

```
1: inputs
2:    p {Pointer to a word in memory}
3: outputs
4:    v {Flag indicating if the test was successful}
5: do atomically:
6:    v ← *p
7:    *p ← 1
8: end do atomically
9: return v
```

Remarks:

- Plenty of processors provide an instruction for this, or something equivalent like Read-And-Clear.
- Some Systems-on-Chip also provide peripheral hardware registers that function this way as well when you read from them.

Algorithm 5.3 Compare-and-Swap

```
1: inputs
2:    p {Pointer to a word in memory}
3:    v1 {Comparison value}
4:    v2 {New value}
5: outputs
6:    {Original value}
7: do atomically:
8:    if *p = v1 then
9:       *p ← v2
10:      return v1
11:   else
12:      return *p
13:   end if
14: end do atomically
```

Remarks:

- This is available on most modern processors as an instruction.
- It is strictly more powerful than TAS. In fact, it is as powerful as needed: you can show [HFP02] that any other atomic operation can be efficiently simulated with CAS, though not with TAS.

Algorithm 5.4 Load-linked / Store Conditional (LL/SC)

Load-linked

```
1: inputs
2:    p {Pointer to a word in memory}
3: outputs
4:    v {Value read from memory}
5: do atomically:
6:    v ← *p
7:    mark p as “locked”
8: end do atomically
9: return v
```

Store-conditional

```
10: inputs
11:    p {Pointer to a word in memory}
12:    v {Value to store to memory}
13: outputs
14:    r {Result of store}
15: do atomically:
16:    if *p has been updated since load-linked then
17:       r ← 1
18:    else
19:       *p ← v
20:       r ← 0
21:    end if
22: end do atomically
23: return r
```

Remarks:

- Also known as “Load-locked”, “Load-reserve”, etc.
- Well-suited to RISC load-store architectures
- Often implemented by marking the line in the cache

5.1.2 Hardware Transactional Memory

LL/SC can be viewed as providing highly restricted form of transaction (on a single word), which aborts if a conflicting update to the word has taken place during the transaction.

Definition 5.5 (Transactional Memory)**.** . ***Transactional Memory*** *is a programming model whereby loads and stores on a particular thread can be grouped into transactions. The* ***read set*** *and* ***write set*** *of a transaction are the set of addresses read from and written to respectively during the transaction. A* ***data conflict*** *occurs in a transaction if another processor reads or writes a value from the transaction's write set, or writes to an address in the transaction's read set. Data conflicts cause the transaction to abort, and all instructions executed since the start of the transaction (and all changes to the write set) to be discarded.*

Example 5.6. *Intel's Transactional Synchronization Extensions (TSX) provide three basic instructions for implementing "Restricted Transactional Memory" or RTM:* `XBEGIN` *(which starts a transaction),* `XEND` *(which commits a transaction), and* `XABORT` *(which forces the transaction to abort). There is also* `XTEST`*, which returns whether it is executing under a transaction.*

If a transaction aborts, the processor rolls back the write set and jumps to a ***fallback instruction address*** *specified by the* `XBEGIN` *instruction, with register information saying why the transaction aborted. This code can then choose to retry the transaction, or so something else (like take out a conventional lock instead).*

TSX also provides an alternative to RTM called Hardware Lock Elision (HLE). Under HLE, the code is written to take out locks using atomic instructions as above, but the processor doesn't actually do this the first time. Instead, it executes the critical section under a transaction, and only if it aborts does it try again, this time really taking out the lock.

Remarks:

- As wiith LL/SC, HTM is usually implemented using the cache coherency protocol to make lines in the cache as part of the read and write sets. Coherency messages signal remote accesses, which then cause aborts. For this reason, conflict detection is actually done at the granularity of entire cache lines rather than just words.

- As with many speculation-based CPU features, HTM is notoriously difficult to get right. The first Intel Haswell and Broadwell processors to be sold supporting TSX had to have the functionality disabled in microcode after serious bugs came to light.

- There are many things other than conflicts or an explicit instruction that can cause an HTM transaction to abort (false sharing, interrupts, etc.) The "false abort rate" is an important measure of the effectiveness of a HTM implementation.

- There is a limit to the size of read and write sets that can be checked (such as in the L1 cache). If this is exceeded, the transaction aborts. It's important not to retry a transaction like this, since it's always going to abort. The abort handling code is therefore usually supplied with information about whether the CPU thinks the transaction should be retried or not.

5.2 Shared-memory synchronization models

We'll assume you're already familiar with semaphores (and P, V operations), mutexes (Acquire, Release), condition variables (Wait, Signal/Notify, Broadcast/NotifyAll), and monitors (Enter, Exit).

Our focus here is the interaction of these operations with the rest of the OS, in particular the scheduler. Assuming for a moment a priority-based scheduler (as in UNIX, Windows, etc.).

Definition 5.7. *Spinlock. A **spinlock** is a multiprocessor mutual exclusion primitive based on one processor spinning on a memory location written by another.*

Algorithm 5.8 TAS-based spinlock

```
1: inputs
2:    p is an address of a word in memory
   Acquire the lock
3: repeat
4:    v ← TAS(*p)
5: until v = 0
6: ...
   Release the lock
7: *p ← 0
```

Remarks:

- Spinlocks only make sense on a multiprocessor: if you're spinning, nobody else is going to release the lock.
- A pure spinlock only makes sense if the duration that any process holds the lock is short, otherwise, it's better to block.

Definition 5.9 (Spin-block problem)**.** . *The **spin-block** problem is to come up with a strategy for how long a thread should spin waiting to acquire a lock before giving up and blocking, given particular values for the cost of blocking, and the probability distribution of of lock hold times.*

Theorem 5.10. *(Competitive spinning): in the absence of any other information about the lock hold time, spinning for a time equal to the cost of a context switch results in overhead at most twice that of the optimal offline algorithm (which has perfect knowledge about the future). This bound is also tight: no online algorithm can do better than a factor of two. Proof: see Anna Karlin et al. [KMMO90].*

Remarks:

- The proof is subtle (and worth reading!), but the intuition is as follows: in the best case, you avoid a context switch and save time. Otherwise, your overhead is at worst twice as bad as immediately blocking.

5.3 Messages: IPC without shared memory

The alternative to communication using shared data structures protected by thread synchronization primitives is to send messages instead. You've already seen this in networking using sockets.

Message passing is best thought of as an abstraction: it's perfectly possible to implement thread synchronization using only messages, and vice versa. This was famously demonstrated by Hugh Lauer and Roger Needham [LN79], which showed that the two models are essentially equivalent, but can vary greatly in performance based on the properties of the underlying hardware.

Definition 5.11 (Asynchronous IPC)**.** *In **asynchronous** or **buffered IPC** the sender does not block, but the send operation instead returns immediately. If the receiving process is not waiting for the message, the message is buffered until the receive call is made. On the receive side, the receive call blocks if no message is available.*

Definition 5.12 (Synchronous IPC)**.** *In contrast, in a **synchronous** or **unbuffered IPC** system, both sender and receiver may block until both are ready to exchange data.*

Remarks:

- Asynchrous IPC is the model you're probably most familiar with from network sockets. "Asynchronous" and "synchronous" are heavily overloaded terms in computer science, but here "asynchronous" means that the send and receive operation do not need to overlap in time: a send can complete long before the corresponding receive starts.
- Asynchronous IPC implies a buffer to hold messages which have been sent but not yet received. If this buffer becomes full, it's not clear what to do: you can discard messages (as in UDP), or block the sender from sending (as in TCP) until the buffer is drained.
- Synchronous IPC, on the other hand, requires no buffering, merely two threads synchronizing in the OS kernel.
- You've probably heard of "non-blocking" I/O, which is an orthogonal concept.

Definition 5.13 (Non-blocking I/O)**.** ***Blocking** communication operations may block the calling thread (such as the asynchronous receive call described above). **Non-blocking** variants of these operations instead immediately return a code indicating that the operation should be retried.*

Remarks:

- Non-blocking operations can be thought of as *polling*; there's usually some kind of operation that can tells which potential non-blocking operations would succeed if they were tried right now: see `select()` or `epoll()` in Unix.
- You can have *synchronous*, *non-blocking* operations: the send call only succeeds when the receiver is waiting.

Example 5.14. Unix *pipes:* *Pipes are the more fundamental IPC mechanism in* UNIX*, and are closely related to* `fork()`*; one might reasonable claim that* UNIX ***is*** *any OS based on* `fork()` *and* `pipe()`*.*

A pipe is a unidirectional, buffered communication channel between two processes, created by:

```
int pipe(int pipefd[2])
```

Each end is identified by a file descriptor, returned by reference in the array `pipefd`*. One sets up a pipe between two processes by creating the pipe, then forking the other the process. This is, at heart, how the shell works.*

When you create a pipe, you immediately get both end-points in one go. We can make this model more flexible, for example allowing the processes at each end of the pipe to be created independently.

Example 5.15. Unix *domain sockets.* *Like network sockets,* UNIX *domain sockets can be bound to an address, which in this case is a filename. The filename can then be used by clients to connect to the socket.*

```
int s = socket(AF_UNIX, type, 0);
```

This allows us to split up, on the client side, the name of a communication end-point (the filename in this case) from the reference you use to actually send and receive data (the file descriptor you get back from open).

Example 5.16. Unix *named pipes.* *named pipes (also called "FIFO"s) go one step further by allowing both client and server to open the FIFO based on its name (it's also a special file type). You can create a FIFO from the command line:*

```
$ mkfifo /tmp/myfifo
```

5.4 Upcalls

So far, every operation to do with communication that we have seen has involved the sender or receiver (in other words, a userspace process) calling "down" into the OS to perform an operation (send, receive, etc.).

Definition 5.17 (Upcall)**.** . *An* ***upcall*** *is an invocation by the operating system (usually the kernel) of a function inside a user process. The called function in the user program is called the* ***entry point****, or the* ***upcall handler****.*

Remarks:

- This is the inverse of a regular system call: the OS calls the program. It is a very important structuring concept for systems, and yet not widely known among non-systems programmers. One way to view an upcall is as the generalization of an interrupt.

- Obviously, the kernel has to know *what* to call in the user program, i.e. the address of the upcall handler.
- If the OS is running conventional processes, and the process has been preempted when the kernel is entered, this naturally raise the question of what happens to the previous thread context that was saved by the OS.
- One approach is to keep this around, and treat the upcall as running in a "special" context which only exists until it returns (to the kernel).
- Alternatively, the OS might choose to pass the previously-saved thread context to the user program, in case it wants to do someting with it (like resume the thread).

Example 5.18 (UNIX signals)**.** *are an example of a the first type of upcall mechanism: the user program registers "signal handlers" as functions that can be called to deliver a signal.* UNIX *systems have a fixed number of signal types (see "`man 7 signal`"). For each signal, an* ***action*** *can be specified: ignore the signal, call a handler function, terminate the process completely, etc.*

Remarks:

- Signals raise some interesting concurrency questions. Which stack does the signal handler run on, for example?
- Another is: what happens if the signal handler issues a system call? Since it's not really part of the regular process, what happens? It turns out that signal handlers are quite limited in what they are allowed to do. For example, "`man 7 signal-safety`" will list the system calls that a signal handler *is* allowed to make, and there are not many of them. They do include `signal()` and `sigaction()`, however.
- Signal handlers can't, in general, safely access program global or static variables, since the main process might have these protected under a lock when the signal handler is called. This includes many standard C library calls cannot (including the reentrant "`_r`" variants of functions like `strtok()`.
- It is possible to `longjmp()` out of a signal handler (and into the process) if you are careful. It's a good exercise to figure out what the OS needs to do so that the process keeps running smoothly.
- As with all upcalls, what happens if another signal arrives? If multiple signals of the same type are to be delivered, UNIX will discard all but one – signals of the same type are basically indistinguishabe. If signals of different types are to be delivered, UNIX will deliver them all, but is free to do so in any order.

Example 5.19 (Scheduler activations)**.** *take the idea of upcalls much further than signals. Every time a process is resumed by the kernel, instead of simply restoring its registers, the kernel upcalls into the process letting it know where the previous execution state has been stored. This allows the process to resume*

it, or do something different: the original motivation for this design was to run highly efficient user-level threads that were aware of when the process itself was preempted and rescheduled. Indeed, the first implementatins also upcalled the process (on one core) whenever it was ***de****scheduled (on another core), just to let it know. The upcall handler is basically the entry point to a user-level thread scheduler.*

Remarks:

- Scheduler activations allow a thread implementation that elegantly combines the performance of user-space threads, and the predictability and flexibility of kernel threads, and this why they were adopted in recent versions of Windows.
- As with signals, what happens if more than one schedular activation is pending? The original implementation used a stack and a reentrant activation handler to allow multiple scheduler activations to be active at a time; but an alternative approach (published at the same time, but under a different name) simply disables upcalls until the activation handler tells the kernel it's OK. In the meantime, the kernel simply resumes the process instead.

5.5 Client-Server and RPC

Message-passing can be fast, and has quite nice semantics (either a message is sent or it isn't, and it's either received or it isn't). Moreover, over a network (as we'll see later in the course), it's the only way to communicate. We will soon encounter cases which, for the moment at least, we do not see in a single-machine OS: lost messages, reordered messages, or messages that are delayed by some unbounded time.

Until then, consider just two parties communicating by messages (often the common case). Typically, this interaction is asymmetric: one end of the communication is offering a service, while the other is using it.

Definition 5.20 (Client-Server)**.** *In the* ***client-server*** *paradigm distributed computing, a* ***server*** *offers a service to potentially multiple* ***clients****, who connect to it to invoke the service.*

Remarks:

- This is a distributed computing concept, but it applies even in the single-machine OS case (which is why we introduce it here). Indeed, the distinction between *inter-process communication* (with a single OS) and *networked communication* (between machines over a network) is increasingly blurred these days, and we'll see more of this later. Rather than focussing on whether one or more machines is involved, it's better to think about what *network model* is assumed between endpoints: can messages be lost? Reordered? Delayed indefinitely? etc.

- Pipes can't handle client-server communication, since either the client or server (or a common ancestor) must have forked the other. Client-server requires a way to *name* the end-point where the server is offering the service, so that clients can connect to it. You have seen one way to deal with this: sockets, where the server address is passed to the `connect()` call by the client.

- If you write client-server code using sockets, however, you immediately encounter an issue: you find yourself writing the same "wrapper" code over and over again for every service and every client.

Definition 5.21 (Remote Procedure Call)**.** ***Remote Procedure Call*** *or* ***RPC*** *is a programming technique whereby remove client-server interactions are made to look to the programmer of both the client and the server as simple procedure calls: the client program calls the server using a simple procedure call, and the server program implements the service purely as a procedure with the appropriate name.*

How this works is as follows: the ***signature*** *of the remote procedure is fed into a* ***stub compiler****, which outputs two chunks of code that go into libraries.*

The first is the ***client stub*** *(or* ***proxy*** *which implements the client side procedure: this takes its arguments,* ***marshals*** *them into a buffer, sends the buffer to the server, and waits for a message in reply. When this comes back, it* ***unmarshals*** *the return value(s) from the call, and returns to the client program.*

The second is the ***server stub****, which performs the correspnding actions on the server side: wait for a message, unmarshal the arguments, call the server code with the arguments, marshal the return value(s) into a buffer, and send it back to the client.*

Remarks:

- As described, this allows only one procedure to be made available remotely. In practice, this generalizes. The stub compiler can generate code for a *interface*, which is a collection of related procedures. All RPCs to the same interface go over the same connection, with an extra, hidden argument marshalled in: the *procedure number* of the function to call on the other side.

- For languages without strong type systems like C, a separate language is needed to define interfaces.

Definition 5.22 (Interface Definition Language)**.** *. An* ***Interface Definition Language*** *or* ***IDL*** *is a small, domain-specific language for writing RPC interface definitions.*

Remarks:

- If the RPC is to be carried over a network, both sides need to agree on a common representation for the arguments and results of the RPC. This requires, in networking terms, a *presentation-layer protocol.* An example is XDR, the *eXternal Data Representation* used for, among other applications, the UNIX Network File System (NFS).

- If the RPC is, instead, local to a machine (in which case it is called, without apparent irony, Local RPC), the use of a presentation-layer protocol is less important. However, the performance tradeoff is now different. For classical networked RPC, the time to execute the simplest possible RPC (the "Null RPC", which just returns) is dominated by the network propagation delay. On a single machine, through the kernel, it can be dominated by the cost of entering and exiting the kernel. For a server which is executing lots of requests, or a client which needs to send many requests in sequence, this can be a bottleneck.

Example 5.23 (RPC over synchronous IPC)**.** . *A client executing an RPC needs to perform two operations, which are typically system calls: first, the send call, and second, the receive call to get the reply. Similarly, a server processing requests need to receive the message, then execute another system call to send the result back.*

For this reason, high-performance local RPC systems allow the OS the combine two operations in a single system call. There are two important cases.

The first is sometimes called "send with closed receive": the operation sends a message to a given destination, and then the thread blocks waiting for a reply ***from that destination****. This performs the whole of the client side of an RPC communication in one syscall.*

The second is sometimes called "send with open receive": this sends a message to a destination and then blocks waiting for a message from ***any*** *source. This is both halves of the server side combined, but in reverse order: the server calls this to send a reply and then block waiting for the next call.*

5.6 Distributed objects

How does this get set up? As with TCP network connections, a server needs to create an end-point (in the TCP case, a listening socket) and advertise its address somewhere, while a client has to look up this address and connect to it. This usually requires a 3rd party.

Definition 5.24 (Name server)**.** . *A* ***name server*** *is a service (usually invoked using RPC) which holds the addresses of other RPC services. Servers* ***register*** *their services with the name server, and clients* ***lookup*** *the service they want to get the address to connect to.*

The data that the name server stores and hands out for a service is sometimes called an ***interface reference****. It's a name for the service, and it can be passed around freely (which is why the nameserver can be an RPC server like any other).*

Definition 5.25 (RPC binding)**.** . *To contact an RPC service, a client has to acquire an* ***interface reference*** *for the service, and then* ***establish a binding*** *to the service. Establishing the binding is basically setting up a connection, and results in an* ***invocation reference****, which is the required client stub.*

Binding can be ***explicit****, in which case the client has to call some kind of "bind" or "connect" procedure to establish it. However,* ***implicit*** *binding is also possible: as part of unmarshalling an interface reference, the binding is established immediately and an invocation reference returned to the client (or server) instead.*

Remarks:

- This is similar to the notion of a binding we saw early on, but here the binding is an association between the client stub and the remote service: the local pointer or reference the client program has to its stub is now, implicitly, bound to the remote server.

- We didn't talk about this in Chapter 2, but the client binding (and the analogous binding on the server side) are often first-class objects themselves: you can perform operations on them to manipulate the connection or query its status, for example.

By now, you're probably thinking that this is beginning to feel somewhat object-oriented.

Definition 5.26. *Distributed object system. A* ***distributed object system*** *is an RPC system with implicit binding where interface references are viewed as object references, and the IDL (if present) defines classes of which services are instances.*

A local datastructure called the ***object table*** *holds a mapping from interface references to invocation references, so that when an interface reference arrives twice, only a single client stub is created.*

Remarks:

- Note that a distributed object system need not be tied to a particular language: well-known exaples like CORBA and DCOM are not. However, they do need to define tbeir own type system if they cannot lift one from a single programming language. Most examples use C primitive types, plus records and (discriminated) unions, plus variable lengh arrays (sequences) and, of course, interface/object references.

- The type system for interfaces can be arbitrarily sophisticated, supporting subtyping (interface inheritance), and (in some cases) distributed garbage collection.

RPC, and distributed objects, are intuitive for programmers, but also are predicated on things mostly working: either your RPC was delivered, was executed, and you got a reply, or you get a program exception saying it didn't.

However, when implementing some *distributed algorithms* which are designed to handle lost messages, delays, node failures, etc., this model isn't realistic (often because these algorithms are intended to create this reliability). Hence, when discussing things like consensus, we tend to talk in terms of messages, whereas talking to web services is couched in terms of RPCs.

Bibliography

[HFP02] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. In Dahlia Malkhi, editor, *Distributed Computing*, pages 265–279, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[KMMO90] Anna R. Karlin, Mark S. Manasse, Lyle A. McGeoch, and Susan Owicki. Competitive randomized algorithms for non-uniform problems. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 301–309, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[LN79] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, April 1979.

Chapter 6

Introduction to Distributed Systems

Why Distributed Systems?

Today's computing and information systems are inherently *distributed*. Many companies are operating on a global scale, with thousands or even millions of machines on all the continents. Data is stored in various data centers, computing tasks are performed on multiple machines. At the other end of the spectrum, also your mobile phone is a distributed system. Not only does it probably share some of your data with the cloud, the phone itself contains multiple processing and storage units. Your phone is a complicated distributed architecture.

Moreover, computers have come a long way. In the early 1970s, microchips featured a clock rate of roughly 1 MHz. Ten years later, in the early 1980s, you could get a computer with a clock rate of roughly 10 MHz. In the early 1990s, clock speed was around 100 MHz. In the early 2000s, the first 1 GHz processor was shipped to customers. In 2002 one could already buy a processor with a clock rate between 3 and 4 GHz. If you buy a new computer today, chances are that the clock rate is still between 3 and 4 GHz, since clock rates basically stopped increasing. Clock speed can apparently not go beyond a few GHz without running into physical issues such as overheating. Since 2003, computing architectures are mostly developing by the multi-core revolution. Computers are becoming more parallel, concurrent, and distributed.

Finally, data is more reliably stored on multiple geographically distributed machines. This way, the data can withstand regional disasters such as floods, fire, meteorites, or electromagnetic pulses, for instance triggered by solar superstorms. In addition, geographically distributed data is also safer from human attacks. Recently we learned that computer hardware is pretty insecure: spectre, meltdown, rowhammer, memory deduplication, and even attacks on secure hardware like SGX. If we store our data on multiple machines, it may be safe assuming hackers cannot attack all machines concurrently. Moreover, data and software replication also helps availability, as computer systems do not need to be shut down for maintenance.

In summary, today almost all computer systems are distributed, for different reasons:

- Geography: Large organizations and companies are inherently geographically distributed, and a computer system needs to deal with this issue anyway.
- Parallelism: To speed up computation, we employ multicore processors or computing clusters.
- Reliability: Data is replicated on different machines to prevent data loss.
- Availability: Data is replicated on different machines to 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. Some say that going from one computer to two is a bit like having a second child. When you have one child and all cookies are gone from the cookie jar, you know who did it! Coordination problems are so prevalent, they come with various flavors and names: consistency, agreement, consensus, blockchain, ledger, event sourcing, etc.

Coordination problems will happen quite often in a distributed system. Even though every single node (computer, core, network switch, etc.) of a distributed system will only fail once every few years, with millions of nodes, you can expect a failure every minute. On the bright side, one may hope that a distributed system with multiple nodes may tolerate some failures and continue to work correctly.

Distributed Systems Overview

We introduce some basic techniques to building distributed systems, with a focus on fault-tolerance. We will study different protocols and algorithms that allow for fault-tolerant operation, and we will discuss practical systems that implement these techniques.

We will see different models (and even more combinations of models) that can be studied. We will not discuss them in detail now, but simply define them when we use them. Towards the end of the course a general picture should emerge, hopefully!

The focus is on protocols and systems that matter in practice. In other words, in this course, we do not discuss concepts because they are fun, but because they are practically relevant.

Nevertheless, have fun!

Chapter Notes

Many good textbooks have been written on the subject, e.g. [AW04, CGR11, CDKB11, Lyn96, Mul93, Ray13, TS01]. James Aspnes has written an excellent freely available script on distributed systems [Asp14]. Similarly to our course, these texts focus on large-scale distributed systems, and hence there is some overlap with our course. There are also some excellent textbooks focusing on small-scale multicore systems, e.g. [HS08].

Some chapters of this course have been developed in collaboration with (former) PhD students, see chapter notes for details. Many colleagues and students have helped to improve exercises and script. Thanks go to Pascal Bissig, Philipp Brandes, Christian Decker, Manuel Eichelberger, Klaus-Tycho Förster, Arthur Gervais, Barbara Keller, Rik Melis, Darya Melnyk, Peter Robinson, Selma Steinhoff, David Stolz, and Saravanan Vijayakumaran. Jinchuan Chen, Qiang Lin, Yunzhi Xue, and Qing Zhu translated this text into Simplified Chinese, and a long the way found improvements to the English version as well. Thanks!

Bibliography

[Asp14] James Aspnes. Notes on Theory of Distributed Systems, 2014.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition).* John Wiley Interscience, March 2004.

[CDKB11] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design.* Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[CGR11] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Mul93] Sape Mullender, editor. *Distributed Systems (2nd Ed.).* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[Ray13] Michel Raynal. *Distributed Algorithms for Message-Passing Systems.* Springer Publishing Company, Incorporated, 2013.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

Chapter 7

Fault-Tolerance & Paxos

How do you create a fault-tolerant distributed system? In this chapter we start out with simple questions, and, step by step, improve our solutions until we arrive at a system that works even under adverse circumstances, Paxos.

7.1 Client/Server

Definition 7.1 (node)**.** *We call a single actor in the system* ***node****. In a computer network the computers are the nodes, in the classical client-server model both the server and the client are nodes, and so on. If not stated otherwise, the total number of nodes in the system is n.*

Model 7.2 (message passing)**.** *In the* ***message passing model*** *we study distributed systems that consist of a set of nodes. Each node can perform local computations, and can send messages to every other node.*

Remarks:

- We start with two nodes, the smallest number of nodes in a distributed system. We have a *client* node that wants to "manipulate" data (e.g., store, update, . . .) on a remote *server* node.

Algorithm 7.3 Naïve Client-Server Algorithm

```
1: Client sends commands one at a time to server
```

Model 7.4 (message loss)**.** *In the message passing model with* ***message loss****, for* ***any*** *specific message, it is not guaranteed that it will arrive safely at the receiver.*

Remarks:

- A related problem is message corruption, i.e., a message is received but the content of the message is corrupted. In practice, in contrast to message loss, message corruption can be handled quite well, e.g. by including additional information in the message, such as a checksum.

- Algorithm 7.3 does not work correctly if there is message loss, so we need a little improvement.

Algorithm 7.5 Client-Server Algorithm with Acknowledgments

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, the client resends the command

---

Remarks:

- Sending commands "one at a time" means that when the client sent command c, the client does not send any new command c' until it received an acknowledgment for c.
- Since not only messages sent by the client can be lost, but also acknowledgments, the client might resend a message that was already received and executed on the server. To prevent multiple executions of the same command, one can add a *sequence number* to each message, allowing the receiver to identify duplicates.
- 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.
- What about multiple clients?

Model 7.6 (variable message delay)**.** *In practice, messages might experience different transmission times, even if they are being sent between the same two nodes.*

Remarks:

- Throughout this chapter, we assume the variable message delay model.

Theorem 7.7. *If Algorithm 7.5 is used with multiple clients and multiple servers, the servers might see the commands in different order, leading to an inconsistent state.*

Proof. Assume we have two clients u_1 and u_2, and two servers s_1 and s_2. Both clients issue a command to update a variable x on the servers, initially $x = 0$. Client u_1 sends command $x = x + 1$ and client u_2 sends $x = 2 \cdot x$.

Let both clients send their message at the same time. With variable message delay, it can happen that s_1 receives the message from u_1 first, and s_2 receives the message from u_2 first.[1] Hence, s_1 computes $x = (0 + 1) \cdot 2 = 2$ and s_2 computes $x = (0 \cdot 2) + 1 = 1$.

□

[1]For example, u_1 and s_1 are (geographically) located close to each other, and so are u_2 and s_2.

Definition 7.8 (state replication)**.** *A set of nodes achieves* ***state replication****, if all nodes execute a (potentially infinite) sequence of commands* $c_1, c_2, c_3, \dots$*, in the same order.*

Remarks:

- State replication is a fundamental property for distributed systems.
- For people working in the financial tech industry, state replication is often synonymous with the term blockchain. The Bitcoin blockchain we will discuss in Chapter 16 is indeed one way to implement state replication. However, as we will see in all the other chapters, there are many alternative concepts that are worth knowing, with different properties.
- Since state replication is trivial with a single server, we can designate a single server as a *serializer*. By letting the serializer distribute the commands, we automatically order the requests and achieve state replication!

Algorithm 7.9 State Replication with a Serializer

1: Clients send 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

Remarks:

- This idea is sometimes also referred to as *master-slave replication*.
- What about node failures? Our serializer is a single point of failure!
- Can we have a more *distributed* approach of solving state replication? 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*.

Algorithm 7.10 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 if**

Remarks:

- This idea appears in many contexts and with different names, usually with slight variations, e.g. *two-phase locking (2PL)*.
- Another example is the *two-phase commit (2PC)* protocol, typically presented in a database environment. The first phase is called the *preparation* of a transaction, and in the second phase the transaction is either *committed* or *aborted*. The 2PC process is not started at the client but at a designated server node that is called the *coordinator*.
- It is often claimed that 2PL and 2PC provide better consistency guarantees than a simple serializer if nodes can *recover* after crashing. In particular, alive nodes might be kept consistent with crashed nodes, for transactions that started while the crashed node was still running. This benefit was even improved in a protocol that uses an additional phase (3PC).
- The problem with 2PC or 3PC is that they are not well-defined if exceptions happen.
- Does Algorithm 7.10 really handle node crashes well? No! In fact, it is even worse than the simple serializer approach (Algorithm 7.9): Instead of needing one available node, Algorithm 7.10 requires *all* servers to be responsive!
- Does Algorithm 7.10 also work if we only get the lock from a subset of servers? Is a majority of servers enough?
- What if two or more clients concurrently try to acquire a majority of locks? Do clients have to abandon their already acquired locks, in order not to run into a deadlock? How? And what if they crash before they can release the locks?
- Bad news: It seems we need a slightly more complicated concept.
- Good news: We postpone the complexity of achieving state replication and first show how to execute a single command only.

7.2 Paxos

Definition 7.11 (ticket)**.** *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 t is the most recently issued ticket.*

Remarks:

- There is no problem with crashes: If a client crashes while holding a ticket, the remaining clients are not affected, as servers can simply issue new tickets.
- Tickets can be implemented with a counter: Each time a ticket is requested, the counter is increased. When a client tries to use a ticket, the server can determine if the ticket is expired.
- What can we do with tickets? Can we simply replace the locks in Algorithm 7.10 with tickets? We need to add at least one additional phase, as only the client knows if a majority of the tickets have been valid in Phase 2.

Algorithm 7.12 Naïve Ticket Protocol

Phase 1

1: Client asks all servers for a ticket

Phase 2

2: **if** a majority of the servers replied **then**
3: Client sends command together with ticket to each server
4: Server stores command only if ticket is still valid, and replies to client
5: **else**
6: Client waits, and then starts with Phase 1 again
7: **end if**

Phase 3

8: **if** client hears a positive answer from a majority of the servers **then**
9: Client tells servers to execute the stored command
10: **else**
11: Client waits, and then starts with Phase 1 again
12: **end if**

Remarks:

- There are problems with this algorithm: Let u_1 be the first client that successfully stores its command c_1 on a majority of the servers. Assume that u_1 becomes very slow just before it can notify the servers (Line 9), and a client u_2 updates the stored command in some servers to c_2. Afterwards, u_1 tells the servers to execute the command. Now some servers will execute c_1 and others c_2!
- How can this problem be fixed? We know that every client u_2 that updates the stored command after u_1 must have used a newer ticket than u_1. As u_1's ticket was accepted in Phase 2, it follows that u_2 must have acquired its ticket after u_1 already stored its value in the respective server.

- Idea: What if a server, instead of only handing out tickets in Phase 1, also notifies clients about its currently stored command? Then, u_2 learns that u_1 already stored c_1 and instead of trying to store c_2, u_2 could support u_1 by also storing c_1. As both clients try to store and execute the same command, the order in which they proceed is no longer a problem.
- But what if not all servers have the same command stored, and u_2 learns multiple stored commands in Phase 1. What command should u_2 support?
- Observe that it is always safe to support the most recently stored command. As long as there is no majority, clients can support any command. However, once there is a majority, clients need to support this value.
- So, in order to determine which command was stored most recently, servers can remember the ticket number that was used to store the command, and afterwards tell this number to clients in Phase 1.
- If every server uses its own ticket numbers, the newest ticket does not necessarily have the largest number. This problem can be solved if clients suggest the ticket numbers themselves!

Algorithm 7.13 Paxos

Client (Proposer)	**Server (Acceptor)**
Initialization	
c ◁ *command to execute*	$T_{\text{max}} = 0$ ◁ *largest issued ticket*
$t = 0$ ◁ *ticket number to try*	
	$C = \perp$ ◁ *stored command*
	$T_{\text{store}} = 0$ ◁ *ticket used to store C*
Phase 1	
1: $t = t + 1$	
2: Ask all servers for ticket t	
	3: **if** $t > T_{\text{max}}$ **then**
	4: $T_{\text{max}} = t$
	5: Answer with `ok`(T_{store}, C)
	6: **end if**
Phase 2	
7: **if** a majority answers `ok` **then**	
8: Pick (T_{store}, C) with largest T_{store}	
9: **if** $T_{\text{store}} > 0$ **then**	
10: $c = C$	
11: **end if**	
12: Send `propose`(t, c) to same majority	
13: **end if**	
	14: **if** $t = T_{\text{max}}$ **then**
	15: $C = c$
	16: $T_{\text{store}} = t$
	17: Answer `success`
	18: **end if**
Phase 3	
19: **if** a majority answers `success` **then**	
20: Send `execute`(c) to every server	
21: **end if**	

Remarks:

- Unlike previously mentioned algorithms, there is no step where a client explicitly decides to start a new attempt and jumps back to Phase 1. Note that this is not necessary, as a client can decide to abort the current attempt and start a new one *at any point* in the algorithm. 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 phases 1 and 2 if the ticket expired.
- The contention between different clients can be alleviated by randomizing the waiting times between consecutive attempts.

Lemma 7.14. *We call a message* `propose`(t,c) *sent by clients on Line 12 a* ***proposal for (t,c)****. A proposal for (t,c) is* ***chosen****, if it is stored by a majority of servers (Line 15). For every issued* `propose`(t',c') *with* $t' > t$ *holds that* $c' = c$*, if there was a chosen* `propose`(t,c)*.*

Proof. Observe that there can be at most one proposal for every ticket number τ since clients only send a proposal if they received a majority of the tickets for τ (Line 7). Hence, every proposal is uniquely identified by its ticket number τ.

Assume that there is at least one `propose`(t',c') with $t' > t$ and $c' \neq c$; of such proposals, consider the proposal with the smallest ticket number t'. Since both this proposal and also the `propose`(t,c) have been sent to a majority of the servers, we can denote by S the non-empty intersection of servers that have been involved in both proposals. Recall that since `propose`(t,c) has been chosen, this means that that at least one server $s \in S$ must have stored command c; thus, when the command was stored, the ticket number t was still valid. Hence, s must have received the request for ticket t' *after* it already stored `propose`(t,c), as the request for ticket t' invalidates ticket t.

Therefore, the client that sent `propose`(t',c') must have learned from s that a client already stored `propose`(t,c). Since a client adapts its proposal to the command that is stored with the highest ticket number so far (Line 8), the client must have proposed c as well. There is only one possibility that would lead to the client not adapting c: If the client received the information from a server that some client stored `propose`(t^*,c^*), with $c^* \neq c$ and $t^* > t$. In this case, a client must have sent `propose`(t^*,c^*) with $t < t^* < t'$, but this contradicts the assumption that t' is the smallest ticket number of a proposal issued after t. □

Theorem 7.15. *If a command c is executed by some servers, all servers (eventually) execute c.*

Proof. From Lemma 7.14 we know that once a proposal for c is chosen, every subsequent proposal is for c. As there is exactly one first `propose`(t,c) that is chosen, it follows that all successful proposals will be for the command c. Thus, only proposals for a single command c can be chosen, and since clients only tell servers to execute a command, when it is chosen (Line 20), each client will eventually tell every server to execute c. □

Remarks:

- If the client with the first successful proposal does not crash, it will directly tell every server to execute c.
- However, if the client crashes before notifying any of the servers, the servers will execute the command only once the next client is successful. Once a server received a request to execute c, it can inform every client that arrives later that there is already a chosen command, so that the client does not waste time with the proposal process.

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

- The original description of Paxos uses three roles: Proposers, acceptors and learners. Learners have a trivial role: They do nothing, they just learn from other nodes which command was chosen.

- We assigned every node only one role. In some scenarios, it might be useful to allow a node to have multiple roles. For example in a peer-to-peer scenario nodes need to act as both client and server.

- Clients (Proposers) must be trusted to follow the protocol strictly. However, this is in many scenarios not a reasonable assumption. In such scenarios, the role of the proposer can be executed by a set of servers, and clients need to contact proposers, to propose values in their name.

- So far, we only discussed how a set of nodes can reach decision for a single command with the help of Paxos. We call such a single decision an *instance* of Paxos.

- For state replication as in Definition 7.8, 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.

Chapter Notes

Two-phase protocols have been around for a long time, and it is unclear if there is a single source of this idea. One of the earlier descriptions of this concept can found in the book of Gray [Gra78].

Leslie Lamport introduced Paxos in 1989. But why is it called Paxos? Lamport described the algorithm as the solution to a problem of the parliament of a fictitious Greek society on the island Paxos. He even liked this idea so much, that he gave some lectures in the persona of an Indiana-Jones-style archaeologist! When the paper was submitted, many readers were so distracted by the descriptions of the activities of the legislators, they did not understand the meaning and purpose of the algorithm. The paper was rejected. But Lamport refused to rewrite the paper, and he later wrote that he *"was quite annoyed at how humorless everyone working in the field seemed to be"*. A few years later, when the need for a protocol like Paxos arose again, Lamport simply took the paper out of the drawer and gave it to his colleagues. They liked it. So Lamport decided to submit the paper (in basically unaltered form!) again, 8 years after he wrote it – and it got accepted! But as this paper [Lam98] is admittedly hard to read, he had mercy, and later wrote a simpler description of Paxos [Lam01].

Leslie Lamport is an eminent scholar when it comes to understanding distributed systems, and we will learn some of his contributions in almost every

chapter. Not suprisingly, Lamport has won the 2013 Turing Award for his fundamental contributions to the "theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency" [**?**]. One can add arbitrarily to this official citation, for instance Lamport's popular LaTeX typesetting system [**?**], based on Donald Knuth's TeX.

This chapter was written in collaboration with David Stolz.

Bibliography

[Gra78] James N Gray. *Notes on data base operating systems.* Springer, 1978.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

Chapter 8

Consensus

8.1 Two Friends

Alice wants to arrange dinner with Bob, and since both of them are very reluctant to use the "call" functionality of their phones, she sends a text message suggesting to meet for dinner at 6pm. However, texting is unreliable, and Alice cannot be sure that the message arrives at Bob's phone, hence she will only go to the meeting point if she receives a confirmation message from Bob. But Bob cannot be sure that his confirmation message is received; if the confirmation is lost, Alice cannot determine if Bob did not even receive her suggestion, or if Bob's confirmation was lost. Therefore, Bob demands a confirmation message from Alice, to be sure that she will be there. But as this message can also be lost...

You can see that such a message exchange continues forever, if both Alice and Bob want to be sure that the other person will come to the meeting point!

Remarks:

- Such a protocol cannot terminate: Assume that there are protocols which lead to agreement, and P is one of the protocols which require the least number of messages. As the last confirmation might be lost and the protocol still needs to guarantee agreement, we can simply decide to always omit the last message. This gives us a new protocol P' which requires less messages than P, contradicting the assumption that P required the minimal amount of messages.
- Can Alice and Bob use Paxos?

8.2 Consensus

In Chapter 7 we studied a problem that we vaguely called agreement. We will now introduce a formally specified variant of this problem, called *consensus*.

Definition 8.1 (consensus)**.** *There are n nodes, of which at most f might crash, i.e., at least $n - f$ nodes are **correct**. Node i starts with an input value v_i. The nodes must decide for one of those values, satisfying the following properties:*

- **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 a* node.

Remarks:

- We assume that every node can send messages to every other node, and that we have reliable links, i.e., a message that is sent will be received.
- There is no broadcast medium. If a node wants to send a message to multiple nodes, it needs to send multiple individual messages. If a node crashes while broadcasting, not all nodes may receive the broadcasted message. Later we will call this best-effort broadcast.
- Does Paxos satisfy all three criteria? If you study Paxos carefully, you will notice that Paxos does not guarantee termination. For example, the system can be stuck forever if two clients continuously request tickets, and neither of them ever manages to acquire a majority.

8.3 Impossibility of Consensus

Model 8.2 (asynchronous)**.** *In the **asynchronous model**, algorithms are event based ("upon receiving message ..., do ..."). Nodes do not have access to a synchronized wall-clock. A message sent from one node to another will arrive in a finite but unbounded time.*

Remarks:

- The asynchronous time model is a widely used formalization of the variable message delay model (Model 7.6).

Definition 8.3 (asynchronous runtime)**.** *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 (every legal input, every execution scenario), assuming that each message has a delay of **at most** one time unit.*

Remarks:

- The maximum delay cannot be used in the algorithm design, i.e., the algorithm must work independent of the actual delay.
- Asynchronous algorithms can be thought of as systems, where local computation is significantly faster than message delays, and thus can be done in no time. Nodes are only active once an event occurs (a message arrives), and then they perform their actions "immediately".
- We will show now that crash failures in the asynchronous model can be quite harsh. In particular there is no deterministic fault-tolerant consensus algorithm in the asynchronous model, not even for binary input.

Definition 8.4 (configuration)**.** *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 (sent but not yet received).*

Definition 8.5 (univalent)**.** *We call a configuration C* ***univalent****, if the decision value is determined independently of what happens afterwards.*

Remarks:

- We call a configuration that is univalent for value v *v-valent*.
- Note that a configuration can be univalent, even though no single node is aware of this. For example, the configuration in which all nodes start with value 0 is 0-valent (due to the validity requirement).
- As we restricted the input values to be binary, the decision value of any consensus algorithm will also be binary (due to the validity requirement).

Definition 8.6 (bivalent)**.** *A configuration C is called* ***bivalent*** *if the nodes might decide for* 0 *or* 1*.*

Remarks:

- The decision value depends on the order in which messages are received or on crash events. I.e., the decision is not yet made.
- We call the initial configuration of an algorithm C_0. When nodes are in C_0, all of them executed their initialization code and possibly, based on their input values, sent some messages. These initial messages are also included in C_0. In other words, in C_0 the nodes are now waiting for the first message to arrive.

Lemma 8.7. *There is at least one selection of input values V such that the according initial configuration C_0 is bivalent, if $f \geq 1$.*

Proof. As explained in the previous remark, C_0 only depends on the input values of the nodes. Let $V = [v_0, v_1, \ldots, v_{n-1}]$ denote the array of input values, where v_i is the input value of node i.

We construct $n+1$ arrays $V_0, V_1, \ldots, V_n$, where the index i in V_i denotes the position in the array up to which all input values are 1. So, $V_0 = [0, 0, 0, \ldots, 0]$, $V_1 = [1, 0, 0, \ldots, 0]$, and so on, up to $V_n = [1, 1, 1, \ldots, 1]$.

Note that the configuration corresponding to V_0 must be 0-valent so that the validity requirement is satisfied. Analogously, the configuration corresponding to V_n must be 1-valent. Assume that all initial configurations with starting values V_i are univalent. Therefore, there must be at least one index b, such that the configuration corresponding to V_{b-1} is 0-valent, and configuration corresponding to V_b is 1-valent. Observe that only the input value of the bth node differs from V_{b-1} to V_b.

Since we assumed that the algorithm can tolerate at least one failure, i.e., $f \geq 1$, we look at the following execution: All nodes except b start with their initial value according to V_{b-1} respectively V_b. Node b is "extremely slow";

i.e., all messages sent by b are scheduled in such a way, that all other nodes must assume that b crashed, in order to satisfy the termination requirement. Since the nodes cannot determine the value of b, and we assumed that all initial configurations are univalent, they will decide for a value v independent of the initial value of b. Since V_{b-1} is 0-valent, v must be 0. However we know that V_b is 1-valent, thus v must be 1. Since v cannot be both 0 and 1, we have a contradiction.

□

Definition 8.8 (transition)**.** *A **transition** from configuration C to a following configuration C_τ is characterized by an event $\tau = (u, m)$, i.e., node u receiving message m.*

Remarks:

- Transitions are the formally defined version of the "events" in the asynchronous model we described before.
- A transition $\tau = (u, m)$ is only applicable to C, if m was still in transit in C.
- C_τ differs from C as follows: m is no longer in transit, u has possibly a different state (as u can update its state based on m), and there are (potentially) new messages in transit, sent by u.

Definition 8.9 (configuration tree)**.** *The **configuration tree** is a directed tree of configurations. Its root is the configuration C_0 which is fully characterized by the input values V. The edges of the tree are the transitions; every configuration has all applicable transitions as outgoing edges.*

Remarks:

- For any algorithm, there is exactly *one* configuration tree for every selection of input values.
- Leaves are configurations where the execution of the algorithm terminated. Note that we use termination in the sense that the system as a whole terminated, i.e., there will not be any transition anymore.
- Every path from the root to a leaf is one possible asynchronous execution of the algorithm.
- Leaves must be univalent, or the algorithm terminates without agreement.
- If a node u crashes when the system is in C, all transitions $(u, *)$ are removed from C in the configuration tree.

Lemma 8.10. *Assume two transitions $\tau_1 = (u_1, m_1)$ and $\tau_2 = (u_2, m_2)$ for $u_1 \neq u_2$ are both applicable to C. Let $C_{\tau_1\tau_2}$ be the configuration that follows C by first applying transition τ_1 and then τ_2, and let $C_{\tau_2\tau_1}$ be defined analogously. It holds that $C_{\tau_1\tau_2} = C_{\tau_2\tau_1}$.*

Proof. Observe that τ_2 is applicable to C_{τ_1}, since m_2 is still in transit and τ_1 cannot change the state of u_2. With the same argument τ_1 is applicable to C_{τ_2}, and therefore both $C_{\tau_1\tau_2}$ and $C_{\tau_2\tau_1}$ are well-defined. Since the two transitions are completely independent of each other, meaning that they consume the same messages, lead to the same state transitions and to the same messages being sent, it follows that $C_{\tau_1\tau_2} = C_{\tau_2\tau_1}$. □

Definition 8.11 (critical configuration)**.** *We say that a configuration C is **critical**, if C is bivalent, but all configurations that are direct children of C in the configuration tree are univalent.*

Remarks:

- Informally, C is critical, if it is the last moment in the execution where the decision is not yet clear. As soon as the next message is processed by any node, the decision will be determined.

Lemma 8.12. *If a system is in a bivalent configuration, it must reach a critical configuration within finite time, or it does not always solve consensus.*

Proof. Recall that there is at least one bivalent initial configuration (Lemma 8.7). Assuming that this configuration is not critical, there must be at least one bivalent following configuration; hence, the system may enter this configuration. But if this configuration is not critical as well, the system may afterwards progress into another bivalent configuration. As long as there is no critical configuration, an unfortunate scheduling (selection of transitions) can always lead the system into another bivalent configuration. The only way how an algorithm can *enforce* to arrive in a univalent configuration is by reaching a critical configuration.

Therefore we can conclude that a system which does not reach a critical configuration has at least one possible execution where it will terminate in a bivalent configuration (hence it terminates without agreement), or it will not terminate at all.

□

Lemma 8.13. *If a configuration tree contains a critical configuration, crashing a single node can create a bivalent leaf; i.e., a crash prevents the algorithm from reaching agreement.*

Proof. Let C denote critical configuration in a configuration tree, and let T be the set of transitions applicable to C. Let $\tau_0 = (u_0, m_0) \in T$ and $\tau_1 = (u_1, m_1) \in T$ be two transitions, and let C_{τ_0} be 0-valent and C_{τ_1} be 1-valent. Note that T must contain these transitions, as C is a critical configuration.

Assume that $u_0 \neq u_1$. Using Lemma 8.10 we know that C has a following configuration $C_{\tau_0\tau_1} = C_{\tau_1\tau_0}$. Since this configuration follows C_{τ_0} it must be 0-valent. However, this configuration also follows C_{τ_1} and must hence be 1-valent. This is a contradiction and therefore $u_0 = u_1$ must hold.

Therefore we can pick one particular node u for which there is a transition $\tau = (u, m) \in T$ which leads to a 0-valent configuration. As shown before, all transitions in T which lead to a 1-valent configuration must also take place on u. Since C is critical, there must be at least one such transition. Applying the same argument again, it follows that all transitions in T that lead to a 0-valent

configuration must take place on u as well, and since C is critical, there is no transition in T that leads to a bivalent configuration. Therefore *all* transitions applicable to C take place on the *same* node u!

If this node u crashes while the system is in C, *all transitions are removed*, and therefore the system is stuck in C, i.e., it terminates in C. But as C is critical, and therefore bivalent, the algorithm fails to reach an agreement.

□

Theorem 8.14. *There is no deterministic algorithm which always achieves consensus in the asynchronous model, with* $f > 0$.

Proof. We assume that the input values are binary, as this is the easiest non-trivial possibility. From Lemma 8.7 we know that there must be at least one bivalent initial configuration C. Using Lemma 8.12 we know that if an algorithm solves consensus, all executions starting from the bivalent configuration C must reach a critical configuration. But if the algorithm reaches a critical configuration, a single crash can prevent agreement (Lemma 8.13). □

Remarks:

- If $f = 0$, then each node can simply send its value to all others, wait for all values, and choose the minimum.
- But if a single node may crash, there is no deterministic solution to consensus in the asynchronous model.
- How can the situation be improved? For example by giving each node access to randomness, i.e., we allow each node to toss a coin.

8.4 Randomized Consensus

Algorithm 8.15 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 current round arrived
7:   if all messages contain the same value v then
8:     Broadcast propose(v, round)
9:   else
10:    Broadcast propose(⊥, round)
11:  end if

12:  if decided then
13:    Broadcast myValue(v_i, round+1)
14:    Decide for v_i and terminate
15:  end if
     Vote
16:  Wait until a majority of propose messages of current round arrived
17:  if all messages propose the same value v then
18:    v_i = v
19:    decided = true
20:  else if there is at least one proposal for v then
21:    v_i = v
22:  else
23:    Choose v_i randomly, with Pr[v_i = 0] = Pr[v_i = 1] = 1/2
24:  end if
25:  round = round + 1
26:  Broadcast myValue(v_i, round)
27: end while
```

Remarks:

- The idea of Algorithm 8.15 is very simple: Either all nodes start with the same input bit, which makes consensus easy. Otherwise, nodes toss a coin until a large number of nodes get – by chance – the same outcome.

Lemma 8.16. *As long as no node sets* ***decided*** *to true, Algorithm 8.15 does not get stuck, independent of which nodes crash.*

Proof. The only two steps in the algorithm when a node waits are in Lines 6 and 16. Since a node only waits for a majority of the nodes to send a message, and since $f < n/2$, the node will always receive enough messages to continue, as long as no correct node set its value decided to true and terminates. □

Lemma 8.17. *Algorithm 8.15 satisfies the validity requirement.*

Proof. Observe that the validity requirement of consensus, when restricted to binary input values, corresponds to: If all nodes start with v, then v must be chosen; otherwise, either 0 or 1 is acceptable, and the validity requirement is automatically satisfied.

Assume that all nodes start with v. In this case, all nodes propose v in the first round. As all nodes only hear proposals for v, all nodes decide for v (Line 17) and exit the loop in the following round. □

Lemma 8.18. *Algorithm 8.15 satisfies the agreement requirement.*

Proof. Observe that proposals for both 0 and 1 cannot occur in the same round, as nodes only send a proposal for v, if they hear a *majority* for v in Line 8.

Let u be the first node that decides for a value v in round r. Hence, it received a majority of proposals for v in r (Line 17). Note that once a node receives a majority of proposals for a value, it will adapt this value and terminate in the next round. Since there cannot be a proposal for any other value in r, it follows that no node decides for a different value in r.

In Lemma 8.16 we only showed that nodes do not get stuck as long as no node decides, thus we need to be careful that no node gets stuck if u terminates.

Any node $u' \neq u$ can experience one of two scenarios: Either it also receives a majority for v in round r and decides, or it does not receive a majority. In the first case, the agreement requirement is directly satisfied, and also the node cannot get stuck. Let us study the latter case. Since u heard a majority of proposals for v, it follows that every node hears *at least one* proposal for v. Hence, all nodes set their value v_i to v in round r. Therefore, all nodes will broadcast v at the end of round r, and thus all nodes will propose v in round $r + 1$. The nodes that already decided in round r will terminate in $r + 1$ and send one additional `myValue` message (Line 13). All other nodes will receive a majority of proposals for v in $r + 1$, and will set decided to true in round $r + 1$, and also send a `myValue` message in round $r + 1$. Thus, in round $r + 2$ some nodes have already terminated, and others hear enough `myValue` messages to continue in Line 6. They send another `propose` and a `myValue` message and terminate in $r + 2$, deciding for the same value v. □

Lemma 8.19. *Algorithm 8.15 satisfies the termination requirement, i.e., all nodes terminate in expected time $O(2^n)$.*

Proof. We know from the proof of Lemma 8.18 that once a node hears a majority of proposals for a value, all nodes will terminate at most two rounds later. Hence, we only need to show that a node receives a majority of proposals for the same value within expected time $O(2^n)$.

Assume that no node receives a majority of proposals for the same value. In such a round, some nodes may update their value to v based on a proposal (Line 20). As shown before, all nodes that update the value based on a proposal, adapt the same value v. The rest of the nodes choses 0 or 1 randomly. The probability that all nodes choose the same value v in one round is hence at least $1/2^n$. Therefore, the expected number of rounds is bounded by $O(2^n)$. As every round consists of two message exchanges, the asymptotic runtime of the algorithm is equal to the number of rounds. □

Theorem 8.20. *Algorithm 8.15 achieves binary consensus with expected runtime $O(2^n)$ if up to $f < n/2$ nodes crash.*

Remarks:

- How good is a fault tolerance of $f < n/2$?

Theorem 8.21. *There is no consensus algorithm for the asynchronous model that tolerates $f \geq n/2$ many failures.*

Proof. Assume that there is an algorithm that can handle $f = n/2$ many failures. We partition the set of all nodes into two sets N, N' both containing $n/2$ many nodes. Let us look at three different selection of input values: In V_0 all nodes start with 0. In V_1 all nodes start with 1. In V_{half} all nodes in N start with 0, and all nodes in N' start with 1.

Assume that nodes start with V_{half}. Since the algorithm must solve consensus independent of the scheduling of the messages, we study the scenario where all messages sent from nodes in N to nodes in N' (or vice versa) are heavily delayed. Note that the nodes in N cannot determine if they started with V_0 or V_{half}. Analogously, the nodes in N' cannot determine if they started in V_1 or V_{half}. Hence, if the algorithm terminates before any message from the other set is received, N must decide for 0 and N' must decide for 1 (to satisfy the validity requirement, as they could have started with V_0 respectively V_1). Therefore, the algorithm would fail to reach agreement.

The only possibility to overcome this problem is to wait for at least one message sent from a node of the other set. However, as $f = n/2$ many nodes can crash, the entire other set could have crashed before they sent any message. In that case, the algorithm would wait forever and therefore not satisfy the termination requirement.

□

Remarks:

- Algorithm 8.15 solves consensus with optimal fault-tolerance – but it is awfully slow. The problem is rooted in the individual coin tossing: If all nodes toss the same coin, they could terminate in a constant number of rounds.

- Can this problem be fixed by simply always choosing 1 at Line 22?!

- This cannot work: Such a change makes the algorithm deterministic, and therefore it cannot achieve consensus (Theorem 8.14). Simulating what happens by always choosing 1, one can see that it might happen that there is a majority for 0, but a minority with value 1 prevents the nodes from reaching agreement.

- Nevertheless, the algorithm can be improved by tossing a so-called *shared coin*. A shared coin is a random variable that is 0 for all nodes with constant probability, and 1 with constant probability. Of course, such a coin is not a magic device, but it is simply an algorithm. To improve the expected runtime of Algorithm 8.15, we replace Line 22 with a function call to the shared coin algorithm.

8.5 Shared Coin

Algorithm 8.22 Shared Coin (code for node u)

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 the local coin set C_u
4: Broadcast `mySet`(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**

Remarks:

- Since at most f nodes crash, all nodes will always receive $n - f$ coins respectively coin sets in Lines 3 and 5. Therefore, all nodes make progress and termination is guaranteed.
- We show the correctness of the algorithm for $f < n/3$. To simplify the proof we assume that $n = 3f + 1$, i.e., we assume the worst case.

Lemma 8.23. *Let u be a node, and let W be the set of coins that u received in at least $f + 1$ different coin sets. It holds that $|W| \geq f + 1$.*

Proof. Let C be the multiset of coins received by u. Observe that u receives exactly $|C| = (n-f)^2$ many coins, as u waits for $n-f$ coin sets each containing $n - f$ coins.

Assume that the lemma does not hold. Then, at most f coins are in all $n-f$ coin sets, and all other coins $(n - f)$ are in at most f coin sets. In other words, the total number of coins that u received is bounded by

$$|C| \leq f \cdot (n - f) + (n - f) \cdot f = 2f(n - f).$$

Our assumption was that $n > 3f$, i.e., $n-f > 2f$. Therefore $|C| \leq 2f(n-f) < (n - f)^2 = |C|$, which is a contradiction. □

Lemma 8.24. *All coins in W are seen by all correct nodes.*

Proof. Let $w \in W$ be such a coin. By definition of W we know that w is in at least $f + 1$ sets received by u. Since every other node also waits for $n - f$ sets before terminating, each node will receive at least one of these sets, and hence w must be seen by every node that terminates. □

Theorem 8.25. *If $f < n/3$ nodes crash, Algorithm 8.22 implements a shared coin.*

Proof. Let us first bound the probability that the algorithm returns 1 for all nodes. With probability $(1 - 1/n)^n \approx 1/e \approx 0.37$ all nodes chose their local

coin equal to 1 (Line 1), and in that case 1 will be decided. This is only a lower bound on the probability that all nodes return 1, as there are also other scenarios based on message scheduling and crashes which lead to a global decision for 1. But a probability of 0.37 is good enough, so we do not need to consider these scenarios.

With probability $1 - (1 - 1/n)^{|W|}$ there is at least one 0 in W. Using Lemma 8.23 we know that $|W| \geq f + 1 \approx n/3$, hence the probability is about $1 - (1 - 1/n)^{n/3} \approx 1 - (1/e)^{1/3} \approx 0.28$. We know that this 0 is seen by all nodes (Lemma 8.24), and hence everybody will decide 0. Thus Algorithm 8.22 implements a shared coin. □

Remarks:

- We only proved the worst case. By choosing f fairly small, it is clear that $f + 1 \not\approx n/3$. However, Lemma 8.23 can be proved for $|W| \geq n - 2f$. To prove this claim you need to substitute the expressions in the contradictory statement: At most $n - 2f - 1$ coins can be in all $n - f$ coin sets, and $n - (n - 2f - 1) = 2f + 1$ coins can be in at most f coin sets. The remainder of the proof is analogous, the only difference is that the math is not as neat. Using the modified Lemma we know that $|W| \geq n/3$, and therefore Theorem 8.25 also holds for *any* $f < n/3$.
- We implicitly assumed that message scheduling was random; if we need a 0 but the nodes that want to propose 0 are "slow", nobody is going to see these 0's, and we do not have progress. There exist more complicated protocols that solve this problem.

Theorem 8.26. *Plugging Algorithm 8.22 into Algorithm 8.15 we get a randomized consensus algorithm which terminates in a constant expected number of rounds tolerating up to $f < n/3$ crash failures.*

Chapter Notes

The problem of two friends arranging a meeting was presented and studied under many different names; nowadays, it is usually referred to as the *Two Generals Problem*. The impossibility proof was established in 1975 by Akkoyunlu et al. [AEH75].

The proof that there is no deterministic algorithm that always solves consensus is based on the proof of Fischer, Lynch and Paterson [FLP85], known as FLP, which they established in 1985. This result was awarded the 2001 PODC Influential Paper Award (now called Dijkstra Prize). The idea for the randomized consensus algorithm was originally presented by Ben-Or [Ben83]. The concept of a shared coin was introduced by Bracha [Bra87]. A shared coin that can withstand worst-case scheduling has been developed by Alistarh et al. [AAKS14]; this shared coin was inspired by earlier shared coin solutions in the shared memory model [Cha96].

Apart from randomization, there are other techniques to still get consensus. One possibility is to drop asynchrony and rely on time more, e.g. by assuming partial synchrony [DLS88] or timed asynchrony [CF98]. Another possibility is to add failure detectors [CT96].

This chapter was written in collaboration with David Stolz.

Bibliography

[AAKS14] Dan Alistarh, James Aspnes, Valerie King, and Jared Saia. Communication-efficient randomized consensus. In *28th International Symposium of Distributed Computing (DISC), Austin, TX, USA, October 12-15, 2014*, pages 61–75, 2014.

[AEH75] EA Akkoyunlu, K Ekanadham, and RV Huber. Some constraints and tradeoffs in the design of network communications. In *ACM SIGOPS Operating Systems Review*, volume 9, pages 67–74. ACM, 1975.

[Ben83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30. ACM, 1983.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[CF98] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. In *Digest of Papers: FTCS-28, The Twenty-Eigth Annual International Symposium on Fault-Tolerant Computing, Munich, Germany, June 23-25, 1998*, pages 140–149, 1998.

[Cha96] Tushar Deepak Chandra. Polylog randomized wait-free consensus. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA*, pages 166–175, 1996.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

Chapter 9

CPU scheduling

In general, scheduling is deciding how to allocate a single temporal resource among multiple clients, in what order and for how long. This is a highly complex subject, as well as being a problem much older and broader than computer science. Here we focus on CPU scheduling, but the OS also schedules other resources (e.g., disk and network IO).

CPU scheduling involves deciding which task to run next on a given CPU, how long to run it for, and why CPU a given task should run on. “Task” here is intentionally vague: it might be a process, thread, batch job, dispatcher, etc.

Definition 9.1 (Scheduling)**.** ***Scheduling*** *is the problem of deciding, at any point in time, which process or thread on every core (or hardware thread) in a system is currently executing.*

Definition 9.2 (Dispatch)**.** *In contrast to scheduling,* ***dispatching*** *refers to the mechanism for (re)starting a particular process or thread running on a particular core.*

Remarks:

- We distinguish between scheduling and dispatch here so we can focus on actual scheduling algorithms in this chapter, and ignore dispatch (which we dealt with earlier in chapters 4 and 5), but it’s a useful distinction in practice. You will see people use the term “scheduling a process” to mean actually running it, but we’ll try and avoid that usage.

To break it down, we start here with very simple scheduling problems and gradually complicate them; this happens to roughly coincide with the computing chronology as well.

9.1 Non-preemptive uniprocessor batch-oriented scheduling

Definition 9.3 (Uniprocessor scheduling)**.** ***uniprocessor scheduling*** *is the problem of scheduling tasks on a single processor core (or, more precisely, hardware execution context).*

Definition 9.4 (Batch scheduling)**.** *A* ***batch workload*** *consists of a set of* ***batch jobs****, each of which runs for a finite length of time and then terminates.* ***Batch scheduling*** *is the problem of scheduling a (potentially unbounded) set of batch jobs, which appear according to some arrival process.*

Definition 9.5 (Non-preemptive scheduling)**.** *A* ***non-preemptive*** *scheduler always allows a job to run to completion once it has started.*

Remarks:

- Batch scheduling is typically performed on mainframes, supercomputers, and large-scale compute clusters such as those used in machine learning or internet search.

9.1.1 Batch scheduling terminology

See Figure 9.6.

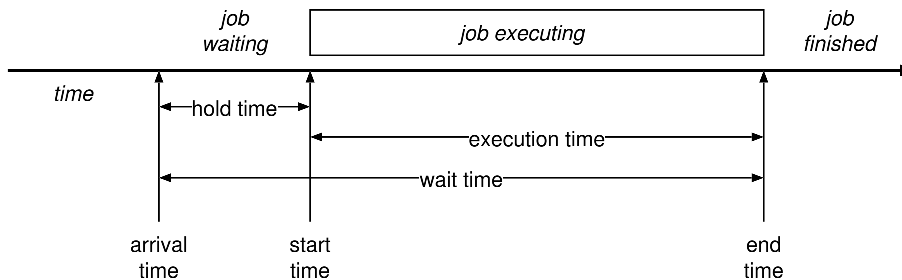


Figure 9.6: Batch job terms

Definition 9.7 (Arrival time)**.** *The* ***arrival time****, or* ***request time****, or* ***release time*** *of a job is the point at which it enters the scheduling system (i.e. the time at which it becomes runnable).*

Definition 9.8 (Start time)**.** *The* ***start time*** *of a batch job is the point in time that it starts executing.*

Definition 9.9 (End time)**.** *The* ***end time*** *or* ***completion time*** *of a job is the point in time that it terminates.*

Definition 9.10 (Execution time)**.** *The* ***execution time*** *or* ***run time*** *of a job is the duration of the job: the number of seconds it takes to complete from its start time.*

Definition 9.11 (Hold time)**.** *The* ***hold time*** *of a job is the time taken to start executing the job from the point where it arrives.*

Definition 9.12 (Wait time)**.** *The* ***wait time*** *or* ***turnaround time*** *of a job is the time take to finish the job from the point where it entered the system.*

9.1.2 Batch scheduling metrics

What makes a good scheduler? A scheduler is generally trying to optimize some metric relative to some workload.

Definition 9.13 (Batch scheduler throughput)**.** *The **throughput** of a batch scheduler is the number of jobs the scheduler completes per unit time.*

Definition 9.14 (Overhead)**.** *The **overhead** of a scheduler is the proportion of CPU time spend running the scheduler itself, as opposed to a client job. Overhead consists of the **context switch time** (strictly speaking, the time to do two half-context-switches) plus the **scheduling cost**.*

Remarks:

- For run-to-completion (non-preemptive) uniprocessor schedulers, throughput for a given workload really only the overhead. However, as things get more complex the throughput starts to depend on the algorithmic properties of the scheduler as well.
- There are plenty of more interesting scheduler metrics, even in the simplistic case we are considering here: mean (or median, or maximum, or ...) wait time, for example. Other examples are various definitions of "fairness", or keeping to some kind of external policy for sharing resources between paying clients.
- Overhead can be a problem for jobs with short run times.
- You should have already spotted that we're assuming the CPU runs at fixed speed. This is not true in reality for power reasons (and cache effects, etc.). In many batch cases, however, the most efficient way to use the CPU is to run it flat-out until there are no more runnable jobs.

Example 9.15. *Suppose the scheduling cost plus context switch time is 1ms, and each job runs for 4ms. The overhead is therefore* $1/(4 + 1) = 20\%$.

Algorithm 9.16 First-come-first-served (FCFS) scheduling

1: Assume each job P_i arrives at time t_i

When the scheduler is entered:

2: Dispatch the job P_j with the earliest arrival time t_j

Remarks:

- This pretty much the simplest scheduling algorithm possible, though it is used in some cases.
- It's low-overhead, but even for simple cases its performance is unpredictable.

Example 9.17. *Three jobs a, b, c arrive in quick succession, with execution times of 24, 3, and 3 seconds respectively. Dispatching them in this order leads to a mean wait time of* $(24 + 27 + 30)/3 = 27s$*. If the arrival order instead was c, b, a we would see a mean wait time of* $(3 + 6 + 24)/3 = 13s$*.*

Definition 9.18 (Convoying)**.** *The* ***convoy phenomenon*** *occurs in FIFO schedulers (among others) when many short processes back up behind long-running processes, greatly inflating mean wait times.*

Remarks:

- This is a well-known (and widely seen!) problem, famously first identified in databases with disk I/O.
- It is a simple form of *self-synchronization*, a wider class of generally undesirable effects in systems.
- Despite this, FIFO is still used, for example in `memcached`, Facebook's front-tier cache.

Algorithm 9.19 Shortest-Job First (SJF) scheduling

1: Assume each job P_i has an execution time of t_i seconds

When the scheduler is entered:

2: Dispatch the job P_j with the shortest execution time t_j

Theorem 9.20. *Shortest-job first is optimal in the sense that it minimizes the average (mean) waiting time for all jobs in the system, at least for the case when all jobs have the same release time.*

Proof. By contradiction: consider a sequence of n jobs with execution times t_k, for $0 < k \leq n$. Let w_k be the waiting time for job k. Then $w_k = w_{k-1} + t_k$, where $w_0 = 0$, and the average waiting time is W/n where W is the total waiting time:

$$W = \sum_{k=1}^{n} w_k = \sum_{k=1}^{n} (n-k)t_k$$

Suppose this sequence minimizes the mean wait time, but is **not** sorted by increasing execution time. Then $\exists j, 0 < j < n$, such that $t_j > t_{j+1}$.

Now consider the alternative sequence obtained by swapping the positions of jobs j and $j + 1$. The total wait time for this new sequence is:

$$W' = \sum_{k=1}^{j-1} (n-k)t_k + (n-j)t_{j+1} + (n-j-1)t_j + \sum_{k=j+2}^{n} (n-k)t_k$$

Subtracting:

$$\begin{aligned} W - W' &= (n-j)t_j + (n-j-1)t_{j+1} - (n-j)t_{j+1} - (n-j-1)t_j \\ &= t_j - t_{j+1} > 0 \end{aligned}$$

Consequently, $W' < W$ and either the original sequence cannot have been optimal, or there was no j (i.e the sequence was sorted). □

Remarks:

- This algorithm is already more complex computationally: it required a sort. If execution times are discrete (e.g. integer minutes), this requires time linear in the number of jobs. This points to a fundamental tradeoff in *scheduling complexity* (which hopefully corresponds to efficiency of the resulting schedule!) and the scheduling overhead. A simpler, less theoretically efficient scheduler may be better than an optimal scheduler that takes too long to schedule.
- In the real world, jobs arrive at any time. SJF can only make a scheduling decision when a job terminates, which means that newly arrived job with a short runtime can experience long hold times because a long job is already running.
- SJF requires knowledge in advance of the job execution times. This is notoriously hard to estimate in practice, but can be finessed by having the clients guess it and penalize them if they get it wrong, or by *preempting* unexpected long jobs.

9.2 Uniprocessor preemptive batch scheduling

Definition 9.21 (Preemption)**.** *A scheduler which can interrupt a running job and switch to another one is* ***preemptive****.*

Algorithm 9.22 SJF with preemption

When a new job enters the system or the running job terminates:

1: Preempt and suspend the currently running job (if there is one)
2: Dispatch (start or resume) the job P_j with the shortest execution time

---

Remarks:

- This requires the OS to have a mechanism for interrupting the current job, such as a programmable interrupt timer.
- SJF with preemption is still problematic: new, short jobs may preempt longer jobs already running, extending their wait time unacceptably.
- "Shortest *remaining* time next" is a variant of preemptive SJF which mitigates, but does not solve, this problem.
- Preemption means that jobs (and, below, processes) are dispatched and descheduled without warning. This is the norm in modern operating systems, and neatly handles other reasons why a job might stop running (page faults, device interrupts, blocking I/O operations, etc.).
- Despite this, there cases where *preventing* preemption is a good thing: hard real-time scheduling, for example, or low-latency communication-intensive parallel jobs.

- Some workloads which are not batch-oriented can nevertheless be approximated as a sequence of jobs. Some interactive workloads can be modeled by viewing each CPU burst as a job (and using exponential averages of previous bursts to predict the execution time). Transaction-based workloads like web page serving can also be viewed as jobs with (hopefully) predictable run times.

9.3 Uniprocessor interactive scheduling

Definition 9.23 (Interactive scheduling)**.** *In contrast to batch-oriented job scheduling, an* ***interactive workload*** *consists of long-running processes most of which are blocked waiting for an external event, such as user input.*

Remarks:

- Interactive scheduling covers a wide range of workloads: almost anything that runs for a long time (such as online services, databases, media servers, user interfaces, etc.), and where the CPU demand is dynamic and unpredictable.

- In addition to cases where the OS preempts the running process, preemptive schedule also captures the case where the process is paused for other reasons: page faults, I/O requests, etc.

- There have always been interactive, non-preemptive systems. Often called "cooperative multitasking" systems, they have included Windows up to version 3.1, the Macintosh OS before version 7, and many embedded systems. Such systems require each process to explicitly give up the processor to the scheduler by performing an I/O request or executing a `yield()` system call every so often.

Definition 9.24 (Response time)**.** *The* ***response time*** *of an interactive program is the time taken to respond to a request for service.*

Remarks:

- Response time is different from wait time: it refers to long-running processes which handle a sequence of external requests (possibly in addition to other computation). Examples include a game responding to user control, a word processor responding to typing, or Facebook responding to a request for a home page.

- Response time, or some statistical measure of it, is often the key metric of interest for interactive scheduling.

Algorithm 9.25 Round-robin (RR) scheduling

1: Let R be a double-ended queue of runnable processes
2: Let q be the scheduling quantum (a fixed time period)

When the scheduler is entered:

3: Push the previously-running job on the tail of R
4: Set an interval timer for an interrupt q seconds in the future
5: Dispatch the job at the head of R

Remarks:

- RR is the simplest interactive scheduling algorithm – in the absence of I/O or other activity, it runs all runnable tasks for a fixed quantum in turn. It is the interactive counterpart to FIFO.
- RR is easy to implement, understand, and analyze.
- Unless you're testing an OS, it's rarely what you want. For one thing, it allocates all processes in the system the same share of CPU time. Moreover, if a process blocks, it implicitly donates the rest of its current time quantum to the rest of the system.
- The response time of a process highly unpredictable, since it essentially depends on where the process is in the run queue.
- RR has a fundamental tradeoff between response time and scheduling overhead, determined by the choice of quantum q. However, it is exacerbated that RR usually switches the running process more than is necessary for the system to make progress.

Example 9.26. *Suppose we have* 50 *processes, the process switch time is* $10\mu s$, *and the scheduling quantum is* $100\mu s$. *This leads to a scheduling overhead of about* 9%, *but an average response time of* $100 \times 110/2 = 2750\mu s$.

Alternatively, if we increase the quantum to $1000\mu s$, *the overhead is reduced to* 0.99%, *but average response time increases to* $100 \times 1010/2 = 50500\mu s$ *or* $50ms$.

9.3.1 Priority-based scheduling

Definition 9.27 (Priority)**.** ***Priority-based scheduling*** *is a broad class of scheduling algorithms in which each process is assigned a numeric priority, and the scheduler always dispatches the highest priority runnable task. A* ***strict priority scheduling*** *algorithm is one where these priorities do not change.*

Remarks:

- Processes with the same priority can be scheduled using some other algorithm, such as RR.

Definition 9.28 (Starvation)**.** *Strict priority scheduling can lead to* ***starvation****: low-priority processes may be starved by high-priority ones which remain runnable and do not block. For this reason, strict priority systems are rare, and*

processes that run at high priority in such systems are carefully written to avoid hogging the processor. Instead most priority-based schedulers are not strict but ***dynamic****: the priorities of tasks change over time in response to system event and application behavior.*

Definition 9.29 (Process aging)**.** ***Aging*** *is one solution to starvation: Tasks which have waited a long time are gradually increased in priority. Eventually, any starving task ends up with the highest priority and runs. The original priorities periodically reset.*

Definition 9.30 (Multi-level queues)**.** *In practice, priority-based schedulers are based on* ***multi-level queues****: there are a finite number priorities (e.g. 256), and each has a queue of processes at that priority. Priority levels are grouped into classes; queues in different classes are scheduled differently. For example, interactive queues are high priority and scheduled using round-robin, batch and background tasks are low-priority and scheduled FCFS, etc.*

Definition 9.31 (Priority Inversion)**.** *Priority inversion occurs when a low-priority P_l process holding a lock R is preempted by a high-priority process P_h, which then attempts to acquire R. If when P_h blocks, a runnable medium-priority process P_m gets to run, this inverts the effect of priority in the schedule.*

Remarks:

- In the worst case, the medium-priority process can prevent the high-priority process from running for an arbitrarily long duration.
- Priority inversion is an old and well-studied problem, but it recurs with disturbing frequency as in the infamous case of the Mars Pathfinder rover.
- Classically, there are two approaches to dealing with priority inversion.

Definition 9.32 (Priority inheritance)**.** *In a system with* ***priority inheritance****, a process holding a lock temporarily acquires the priority of the highest-priority process waiting for the lock until it releases the lock.*

Remarks:

- Priority inheritance mostly solves the priority inversion problem, but at a cost: the scheduler must now be involved in every lock acquire/release. This increases runtime overhead.

Definition 9.33 (Priority ceiling)**.** *In a system with* ***priority ceiling****, a process holding a lock runs at the priority of the highest-priority process which can ever hold the lock, until it releases the lock.*

Remarks:

- Priority ceiling incurs much less runtime overhead than priority inheritance, but potentially requires static analysis of the entire system to work. Its use is therefore restricted to embedded real-time systems.

- A (rather conservative) approximation to priority ceiling is to disable interrupts during the lock hold time, but this is only applicable in limited situations.

Definition 9.34 (Hierarchical scheduling)**.** *A* ***hierarchical scheduler*** *is a further generalization of multi-level queues: queues are instead organized in a nested hierarchy or tree of* ***scheduling domains****. Within each domain (node in tree), sub-nodes are scheduled according to a potentially different policy.*

Definition 9.35 (Multilevel Feedback Queues)**.** *A* ***multilevel feedback queue scheduler*** *is a class of multi-level queue which aims to deliver good response for interactive jobs plus good throughput for background tasks. The key idea is to penalize CPU-bound tasks in favor of I/O bound tasks. Processes which do not block but run continuously during a time interval have their priority reduced (a form of aging). I/O bound (including interactive) tasks tend to block, and therefore remain at high priority. CPU-bound tasks are eventually re-promoted.*

Remarks:

- MLFQ schedulers are a very general class of algorithm. Almost any non-real-time scheduling algorithm can be approximated by multi-level feedback queues.

Example 9.36 (The Linux o(1) scheduler)**.** . *This version of the Linux scheduler is a 140-level Multilevel Feedback Queue. Levels 0-99 (high priority) are static, fixed, "real-time" priorities scheduled with FCFS or RR. Levels 100-139 are user tasks scheduling using RR, with priority aging for interactive (I/O intensive) tasks.*

This makes the complexity of scheduling independent of the number of tasks. The scheduler uses two arrays of queues: "runnable" and "waiting"; when no tasks remain in the "runnable" array, the two arrays are simply swapped.

Example 9.37 (The Linux "completely fair scheduler")**.** . *In the CFS as described in the documentation, a task's priority is determined by how little progress it has made adjusted by fudge factors over time. The task gets a "bonus" if it yields or blocks early (this time is distributed evenly). The implementation uses a Red-Black tree to maintain a sorted list of tasks, meaning that operations are now $O(logn)$, but still fast.*

In fact, this is the very old idea of "fair queuing" from packet networks, also known as "generalized processor scheduling". It ensures a guaranteed service rate for all processes, although CFS does not expose (or maintain) this guarantee.

Remarks:

- Stepping back a bit from the details, the schedulers we have seen in UNIX conflate *protection domains* with *resource principals*: priorities and scheduling decisions are per-process (or threads). In practice, *applications* may span multiple processes, and at the same time share server processes with other applications. This means we may want to allocate resources across processes, or provide separate resource allocations within a single process – think of a web server, for instance.

- Scheduling processes can also lead to unfairness between users: If I run more compiler jobs than you, I get more CPU time.
- The algorithms we have seen do not deal cleanly with in-kernel processing, for example interrupts or the overhead of scheduling itself.
- Some (though not all) of these issues are addressed by *virtual machines* or *containers*.

9.4 Real-time scheduling

Definition 9.38 (Hard real-time)**.** *An application is **hard real-time** if its correctness depends not only on the I/O actions it performs, but also the time it takes to execute. Hard real-time task correctness is often expressed in terms of **deadlines**: each task has a specific point in time by which it must have completed in order to be correct.*

Remarks:

- Hard real-time systems include engine management units (EMUs) for cars, control systems for critical machinery, avionics, etc.
- In the general case, hard real-time scheduling is impossible: tasks can appear at any time, with any deadlines. Hard real-time systems in practice must impose constraints on the set of tasks they are prepared to schedule so that they can guarantee correctness (including every task meeting its deadline).
- In hard real-time systems, the execution time of each task is generally known in advance along with the deadline
- Tasks in a hard real-time system can be *periodic* (they recur at regular intervals) or aperiodic.
- If the task set is not know in advance, the system must reject tasks for which no *feasible schedule* is possible, a process called *admission control.*
- *Real-time* does not mean fast! Both hard- and soft-real-time scheduling are about predictability, not performance. Hard real-time systems in particular are often quite slow.

Definition 9.39 (Rate-monotonic scheduling)**.** ***Rate-monotonic scheduling** (RMS) schedules periodic tasks by always running the task with shortest period first. This is a static (offline) scheduling algorithm.*

Suppose there are m (periodic) tasks, each task i has execution time C_i and period P_i. Then RMS will find a feasible schedule if:

$$\sum_{i=1}^{n} \frac{C_i}{P_i)} \leq n(2^{1/n} - 1)$$

Remarks:

- The proof [LL73] is beyond scope of this course, but worth reading.
- This condition puts a limit on the processor utilization (the left-hand side of the inequality) before deadlines get missed. As the number of tasks increases, this tends to:

$$\lim_{n \to \infty} n(\sqrt[n]{2} - 1) = \ln 2 \approx 0.693147\ldots$$

  The implication here is that it's hard to use more than 69% of the system under RM.

- RM is one of the two classic hard real-time schedulers: it is extremely efficient provided that tasks are periodic and the full workload is known in advance – this is the case in many embedded control applications,

So what should we do if we need to run online (that is, we don't know the job mix in advance)?

Definition 9.40 (Earliest-deadline first)**.** ***Earliest deadline first*** *[EDF] scheduling sorts tasks by deadline and always runs the earliest deadline first. It is dynamic and online, and tasks are not necessarily periodic.*

EDF is guaranteed to find a a feasible schedule if:

$$\sum_{i=1}^{n} \frac{C_i}{P_i)} \leq 1$$

I.e. it can use 100% of a processor, if we ignore the overhead of scheduling and context switching.

Remarks:

- EDF is more complex (scheduling decisions can be $o(\log n)$ in the number of tasks). It is typically implemented by maintaining a *priority queue* of jobs sorted by deadline, often represented as a heap.
- It is much more flexible, and admission control can be performed online.
- If utilization exceeds 100%, however, EDF is *unstable*: its behavior is unpredictable.

Definition 9.41 (Soft real-time scheduling)**.** *In contrast to hard real-time systems, a* ***soft real-time*** *task has timing requirements which are non-strict, but nevertheless affect the quality and utility of the result.*

Remarks:

- A classic example of a soft real-time task is multimedia, like video playback. In practice, users can tolerate some degradation in the quality of a video they are watching, but the task is still organized around displaying a sequence of video frames at the correct time, and/or playing a set of audio samples with the correct synchronization.

Definition 9.42 (Reservation-based scheduling)**.** *In contrast to conventional priority-based schedulers, a* ***reservation-based scheduling policy*** *guarantees a minimum number of CPU time units to each job.*

Remarks:

- EDF scheduling can be used to provide a long-running task (such as video playback) with a *guaranteed processor rate*, by breaking the task into a set of *periods* (for example, video frame times) during which the task is guaranteed a certain number of CPU cycles. This provides a sequence of jobs to EDF, where the execution time of each one is the number of cycles required, and the deadline is the end of the period.
- Such a scheduler actually provides a good approximation to *weighted fair queuing*, which you may be familiar with from networking.

9.5 Multiprocessor scheduling

So far we've considered scheduling a single processor (and its workload) in isolation. As soon as we have more than one processor to manage, things get much more complicated. Fully general multiprocessor scheduling is NP-hard - it tends to reduce to 2-dimensional bin-packing. The two-dimensionality comes from having to decide which core to run a given thread on as well as when to dispatch it on that core.

In general, multiprocessor scheduling is beyond the scope of this course. We just present a simplified overview here, starting with some simplifying assumptions:

- The system can always preempt a task. This rules out some very small embedded systems or hard-real-time systems (and early PC and Macs, it turns out) but otherwise is reasonable.
- The scheduler is *work-conserving.*

Definition 9.43 (Work conserving)**.** *A scheduler is* ***work conserving*** *if no processor is ever idle when there is a runnable task.*

9.5.1 Sequential programs on multiprocessors

Scheduling a collection of sequential programs on multiprocessors is relatively simple, although more complex than uniprocessor scheduling.

Definition 9.44 (Naive sequential multiprocessor scheduling)**.** *The simplest model for multiprocessor scheduling maintains a single system-wide run queue. Whenever an individual processor makes a scheduling decision, it picks a thread from the run queue to remove and dispatch.*

Remarks:

- As described, this scheduler is work-conserving (modulo overhead).
- We haven't said anything about the per-core scheduling algorithm used when a core looks at the run queue. It can be almost any uniprocessor scheme, but note that most of the guarantees vanish. For example, priority invariants might not be maintained across the whole system.
- Basic multi-queue models from queuing theory can be applied to analyze a system like this, but one must also take into account the overheads of locking and sharing the queue.
- In many situations, the single run queue becomes a significant bottleneck due to the need to globally lock it whenever any core needs to reschedule.
- Threads or tasks in this scheme also end up being allocated arbitrarily to cores, and so tend to move frequently between cores and, more critically, between different groups of cores sharing a cache. This dramatically reduces locality and hence performance.

Definition 9.45 (Affinity-based scheduling)**.** *To remove the bottleneck of a single run queue and improve cache locality of running processes, **affinity-based scheduling** tries to keep jobs on one core as much as possible. Each core has its own run queue, and jobs are periodically re-balanced between all the individual queues.*

Remarks:

- This is much more efficient, but note that it is not work conserving any more. A processor can end up with an empty run queue when other queues have jobs which are runnable, but not currently running,
- One way to mitigate this is for a processor which is idle to "steal" a job from a more heavily loaded processor.

Definition 9.46 (Work-stealing)**.** *A **work-stealing** scheduler allows one core which would otherwise be idle to "steal" runnable jobs from neighboring cores so as keep doing useful work.*

9.5.2 Parallel programs on multiprocessors

Things get much more complex when we consider jobs not as single threads, but as collections of parallel threads which coordinate among themselves. For example, global barriers in parallel applications present a significant challenge: one slow thread has huge effect on performance.

Multiple threads in many (but not all) applications would benefit from cache sharing, and different competing applications on the same machine can pollute each others' caches. However, in other cases, it's better to have each thread have its own cache and thereby maximize use of the cache across the whole machine.

The first case leads to clustering threads of the same process on a single socket, whereas the second leads to spreading them across a machine.

This topic is huge, and an active area of research (the more so when we consider that not all cores these days are uniform).

Bibliography

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

Chapter 10

Input / output

> I wouldn't trust any critical program created by someone who has never written an interrupt service routine.
>
> Chandu Thekkath

Definition 10.1 (I/O subsystem)**.** *Every OS has an **I/O subsystem**, which handles all interaction between the machine and the outside worlds. 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.*

Remarks:

- The I/O subsystem usually divides neatly into *device drivers* and *generic functionality*.
- In some OSes, like Linux, the I/O subsystem evolved organically over time and the boundary between it and the rest of the OS is sometimes a little blurred. In others, like MacOS, it's clearly delineated (the I/O subsystem in MacOS is called DeviceKit).
- Not all the I/O subsystem is necessarily in the kernel. It is possible to run most of it (including the device drivers) in user space, as in a microkernel.

We'll start with recapping the basic concepts of I/O.

Definition 10.2 (Device)**.** *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 the case of x86 machines) in **I/O space**. A device is also usually a source of **interrupts**, and may initiate **Direct Memory Access** (DMA) transfers.*

Remarks:

- Hopefully, the above definition should not be new to you (if you have taken the ETH Systems Programming course).
- This definition doesn't say a lot about what a device is physically. In practice this is difficult: in the past, a device really was just that: a piece of metal with connectors on it. Today, it might be some bundle of functionality somewhere on a chip. Anything that isn't a processor, RAM, or interconnect is often a device.

The software component of the OS that talks to a particular device is called the driver.

Definition 10.3 (Device driver)**.** *The **device driver** for a particular device is the software in the OS which is 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.*

Remarks:

- The driver can be thought of as sitting between hardware and rest of the OS.
- A given OS has a *driver model* which defines the kinds of abstractions a device driver will provide.
- In Unix, drivers run in the kernel for the most part, but there is not necessary reason for this.
- The concept of a driver long predates object-oriented program. This can lead to some confusion, since the term can equally refer to the *body of code* which is written to manage a particular piece of hardware (or a group of similar models of hardware device), and the runtime *software object* which manages a single device.

In the rest of this chapter, we're going to talk about "devices" or "the device" at lot, when we actually mean "the representation inside the OS that corresponds to a hardware device". To the OS, what matters is what it thinks a device is, rather than what the device physically is. Indeed, we'll see pseudo-devices which don't physically exist at all.

10.1 Devices and data transfer

The whole of this section should be familiar to you from the "Systems Programming and Computer Architecture" course.

Definition 10.4 (Device registers)**.** *A **device register** is a physical address location which is used for communicating with a device using reads and writes.*

Remarks:

- A hardware register is not memory, but it sits in the physical address space. There is no guarantee that reading from a device register will return the same value that was last written to it.
- Reading a device register can return device input data, or status information about the device.
- Writing a device register can send it data to be output, or configure the device.
- In addition to the above, however, both read and writing to the register can *trigger actions* in the device hardware.

Definition 10.5 (Programmed I/O)**.** ***Programmed I/O*** *consists of causing input/output to occur by writing data values to hardware registers from software (output), or reading values from hardware registers into CPU registers in software (input).*

Algorithm 10.6 Programmed I/O input

```
1: inputs
2:    l: number of words to read from input
3:    d: buffer of size l
4: d ← empty buffer
5: while length(d) < l do
6:    repeat
7:       s ← read from status register
8:    until s indicates data ready
9:    w ← read from data register
10:   d.append(w)
11: end while
12: return
```

Remarks:

- You can construct the corresponding output algorithm by symmetry arguments.
- Programmed I/O is the simplest way for software on a core to communicate with a device.
- It is fully synchronous: the CPU has to write to the register to cause output to happen there and then, and must read from the register for input. There is no buffering.
- It is also polled: there is no way for a device to signal that it has data ready, or it is ready to send data.

Definition 10.7 (Interrupt)**.** *An* ***interrupt*** *is a signal from a device to a CPU which causes the latter to take an exception and execute an* ***interrupt service routine*** *(ISR), also known as an* ***interrupt handler****.*

Algorithm 10.8 Interrupt-driven I/O cycle

1 Initiating (software)

1: Process A performs a blocking I/O operation
2: OS initiates an I/O operation with the device
3: Scheduler blocks Process A, runs another process

2 Processing (hardware)

4: Device performs the I/O operation
5: Raises device interrupt when complete, or an error occurs

3 Termination (software)

6: Currently running process is interrupted
7: Interrupt handler runs, processes any input data
8: Scheduler makes Process A runnable again

4 Resume (software)

9: Process A restarts execution.

Remarks:

- Interrupts solve the polling problem, and so decouple the software and hardware to some extent.
- The CPU still has to copy output data to the device when it initiates the request, and/or copy input data from the device when the operation completes.
- To further decouple the two, we need to allow the device to do the copy itself.

Definition 10.9 (Direct Memory Access)**.** *Using **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.*

Remarks:

- DMA in its simple form mean that the processor's involvement at the start and end of an I/O operation is minimized, since the data itself does not need to be copied.
- DMA is typically performed by the device, though in older hardware a separate *DMA engine* was provided which essentially performed programmed I/O on a device while the CPU did something else.
- DMA saves bus bandwidth, since the data doesn't need to be copied to through the CPU's registers.
- Typically only a single interrupt is needed, so signal the end of a complete DMA copy.

- The real value of DMA comes when the device uses it both to transfer data to and from main memory, but also to read new I/O operations from a list in memory, and write completion information back to main memory. This is the concept of "descriptor rings" that you saw last year, and further decouples CPU and device.
- DMA is, for the most part, *physical* (not virtual) access to memory (but see IOMMUs, below). This means that a virtual address in a user program or the kernel (if the kernel runs in virtual address space, which is common) must be translated to a physical address before being handed to the device.
- DMA transfers to and from main memory may or may not be coherent with processor caches. If not, device drivers must take great care to *flush* (*clean*) and *invalidate* processor caches before and after DMA transactions.

10.2 Dealing with 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. Input data arrives without warning, and an input operations takes an unknown period of time. A busy device becomes capable of accepting more output data at an unspecified time in the future, and it's not clear when an output operation is going to complete.

Definition 10.10 (First-level interrupt service routine)**.** *The **First-level Interrupt Service Routine** or FLISR is the code that executes immediately as a result of the interrupt.*

Remarks:

- An FLISR runs regardless of what else is happening in the kernel (unless interrupts are disabled).
- As a result, it can't change much since the normal kernel invariants might not hold: it can't allocate memory from the kernel heap (if the kernel has one), it can't acquire or release locks, and it should not take too long to finish.

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: deferred procedure calls, driver threads, and non-blocking kernels.

Definition 10.11. *Deferred procedure calls. A **Deferred procedure call**, sometimes known as a **2nd-level interrupt handler**, a **soft interrupt handler**, or a **slow interrupt handler**, is a program closure created by the 1st-level interrupt handler. It is run later (hence the name) by any convenient process, typically just before the kernel is exited.*

Remarks:

- DPCs are extremely efficient, and a common solution to the rendezvous problem (e.g. in VMS, Windows, BSD Unix, etc.).
- The closure itself is small (a few words), and can be statically allocated (since you only ever need one outstanding per-device) so doesn't need to be dynamically allocated inside the FLISR.
- DPCs do need to be queued up for execution once the interrupt context has finished, so we need a lock-free queue to hold them.

Definition 10.12 (Driver thread). *A* ***driver thread****, sometimes called an* ***interrupt handler thread****, serves as an intermediary between interrupt service routines 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.*

Remarks:

- Driver threads are heavyweight: even if they only run in the kernel, the still require a stack and a (same address space) context switch to and from them to perform any I/O requests. They therefore take time (cycles) and space (memory). The latter also translates into time (more cache misses).
- They are conceptually simple, and can be understood more intuitively than DPCs. Consequently, if many kernel and/or driver developers are of intermediate skill and ability (such as with Linux), this may be the preferred option from an engineering perspective.

The third alternative, used in microkernels and exokernels, 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.

Definition 10.13 (Bottom-half handler). *The part of a device driver code which executes either in the interrupt context or as a result of the interrupt (like a DPC) is the* ***bottom half****.*

Definition 10.14 (Top-half handler). *The part of a device driver which is called "from above", i.e. from user or OS processes, is the* ***top half****.*

Remarks:

- Note that the top half can be scheduled, and so the time it uses can be accounted to some process, whereas the bottom half either isn't scheduled (the FLISR) or is run by whatever is handy (the driver thread or the current process).

- Note that this is **not** the Linux terminology, but it is the one used by pretty much all other OSes (including other Unix systems, all of which predate Linux). In Linux, for unknown reasons, the "top half" is the FLISR, while the DPC or driver thread is the "bottom half".

10.3 Device models

Definition 10.15 (Device model)**.** *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.*

Remarks:

- The device model fulfills the role of device abstraction in the system. 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. It also specifies the relationship between physical devices and device drivers.

As a rough example, we'll discuss the Unix device model here. Unix divides devices into three classes, character, block, and network devices.

Definition 10.16 (Character devices)**.** *A **character device** in* Unix *is used for "unstructured I/O", and presents a byte-stream interface with no block boundaries.*

Remarks:

- Character devices are accessed by single byte or short string get/put operations.
- In practice, buffering implemented by libraries
- Examples include keyboards, serial lines (UARTS), mice, USB-controlled missile launchers, etc.

Definition 10.17 (Block devices)**.** *A **block device** in* Unix *is intended for "structured I/O", and deals with "blocks" of data at a time - for example, disk blocks.*

Remarks:

- Block devices often resemble files more than simply in being named by the filing system: storage devices like disks or SSDs are seekable and mappable like files. Access to them often uses the Unix buffer cache. We'll see more of block devices as a basis for file systems in Chapter 20.
- In practice, the distinction between character and block devices is somewhat arbitrary.

Definition 10.18 (Network devices)**.** *A **network device** in* Unix *corresponds to a (real or virtual) network interface adapter. It is accessed through a rather different API to character and block devices.*

Remarks:

- Arguably, network interfaces don't fit nicely into either of these models (they came along much later in the development of UNIX), and indeed NICs per se in UNIX are generally not abstracted as device. Instead, streams of packets can be send and received through sockets or special character devices like `/dev/tun`.
- We'll look at network devices in more detail later in Chapter **??**.

Definition 10.19 (Pseudo-devices)**.** *A **pseudo-device** is 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.*

Example 10.20. UNIX *systems have a variety of pseudo-devices, such as:*

- *`/dev/mem` A character device corresponding to the entire main memory of the machine*
- *`/dev/random` Generates random numbers when read from.*
- *`/dev/null` Anything written is discarded, read always returns end-of-file.*
- *`/dev/zero` Always reads as zeroes.*
- *`/dev/loop` Block device for making a file look like an entire file system.*
- *`/dev/loop` Block device for making a file look like an entire file system.*

How are devices identified inside the kernel?

Example 10.21 (Traditional UNIX device configuration)**.** *In older* UNIX *systems, devices were named inside the kernel by a pair of bytes: the **major** and **minor** device numbers.*

*The **Major device number** identified the class of device (e.g., disk, CD-ROM, keyboard).*

*The **Minor device number** identified a specific device within a class.*

In addition, a third "bit" determined whether the device was a character or block device.

As a naming scheme, this was fine when there were very few different devices and device types, but things have changed a lot since then. Not only are there a large number of different models of device that might be plugged into a computer (literally tens of millions), they can also be connected in a variety of different ways.

Definition 10.22 (Device discovery)**.** *Most modern OSes (with the exception of some small embedded systems) perform **device discovery**: the process of finding and enumerating all hardware devices in the system and storing metadata about them (where and what they are) in some kind of queryable data store.*

Example 10.23 (Linux `sysfs`)**.** *. Modern versions of Linux store this information in the kernel, but expose it to curious user programs via a "pseudo file system" (i.e. something that looks like a file system but isn't) called `sysfs`.*

While `sysfs` *is a hierarchical directory structure, devices appear in it multiple times, organized by type, connection, etc.*

`sysfs` *is a very strange way to build a database (which is what it essentially is), but does fit in the Unix philosophy of "everything is a file".*

When devices are plugged or unplugged, the contents of `sysfs` *change. User programs can get notification of these changes by listening on a special socket.*

Linux device ***drivers*** *are also more dynamic than in the old days: they are typically implemented as loadable "kernel modules", to be loaded on demand when the kernel discovers a device that needs a particular driver. The initial list of drivers is given at boot time, but a daemon can load more on demand if required.*

10.4 Device configuration

In addition to simply discovering a device, and finding out how to access it, the OS often has to configure the device *and other devices* in order to make it work.

Example 10.24 (USB hotplug)**.** *When a USB device (such as a USB thumb drive) is plugged in, a number of different devices are involved, at the very least:*

- *The USB drive itself*
- *The USB* ***host bus adapter*** *or HBA, which interfaces the USB device network to the rest of the computer.*
- *The USB* ***hub*** *that the device was plugged into. This can easily not be a physically separate hub, but one integrated onto the motherboard or built into another device (such as the HBA).*

Broadly speaking, when the device is plugged in, the HBA notifies the OS that something has been plugged in. The HBA driver then talks to the HBA to enumerate the devices attached to it, including the hubs – USB is organized approximately as a tree of devices where the non-leaf nodes are hubs. The HBA adapter then has to assign new bus and device identifiers to anything that has changed and reconfigure the HBA and switches. It also discovers the new device by finding out what it is – USB devices, like PCI devices, describe themselves with a 4-byte code.

After this, the OS can start the appropriate driver for the device, and tell it where in the USB hierarchy the new device is.

Sometimes, more complex resource allocation is required.

Example 10.25 (PCI configuration)**.** *Configuration of PCI (or, today, PCI Express) devices at a high level looks very similar to that of USB, except that the "PCIe Root Complex" is used instead of the HBA, and "PCIe bridges" are used instead of USB hubs.*

However, unlike in USB, where a driver for something like a USB stick talks to the physical device by sending it messages via the HBA, in PCIe all devices are memory mapped. The regions of memory they need to be mapped have to be allocated by the OS from the physical address space of the machine. This is a complex process, which continues to cause problems for OS developers.

Moreover, almost all devices require interrupt routing.

Definition 10.26 (Interrupt routing)**.** ***Interrupt routing*** *is the process of configuring* ***interrupt controllers*** *in the system to ensure that when a device raises an interrupt, it is delivered to the correct vector on the correct core (or group of cores).*

Remarks:

- Interrupt routing is one of the things that is getting much more complex over time. It is not unusual for a modern PC to have 4 or 5 interrupt controllers between a device and the CPUs, and even phones can have 3 or more.
- This is also a resource allocation problem: vector numbers and intermediate interrupt numbers between interrupt controllers must all be allocated.
- This problem, like discovery and other kinds of configuration, is a generic issue: it is not the job of a single device driver, but instead the function of the common part of the I/O subsystem (though the PCI bridge drivers, USB HBA drivers, etc. may also play a role).

10.5 Naming devices

Once configured, an OS needs a way to refer to devices (again, driver instances really) from user space (for both user programs and system daemons). This is, of course, a naming problem, and it is important to understand what kind of problem. For example, it's useful if the same device has the same name on different computers, rather than giving every device in the world a unique name.

Example 10.27. *In older versions of* Unix*, where every device was identified by a (major, minor) pair of integers, devices were named using the file system by creating a special kind of file to represent each device, using the* `mknod` *command.*

We'll cover the Unix *file system later on in Chapters 19 and 20, but as a preview, the major and minor device numbers were stored in the* ***inode****, meaning the "device file" took up very little space.*

Devices are traditionally grouped in the directory `/dev`*. For example:*

- `/dev/sda`*: First SCSI/SATA/SAS disk*
- `/dev/sda5`*: Fifth partition on the above*
- `/dev/cdrom0`*: First DVD-ROM drive*
- `/dev/ttyS1`*: Second UART*

In the truly old days, all drivers were compiled into the kernel. Each driver probed the hardware itself for any supported devices, and the system administrator populated `/dev` *manually using* `mknod` *when a new device was connected.*

Modern hardware trends have resulted in a huge explosion of devices, and this approach is unworkable. People simply want to plug a device in and have it work.

Example 10.28. *In modern versions of Linux (including Android),* `/dev` *still contains files corresponding to all the devices, but* `/dev` *itself is no longer a "real" file system (i.e. residing on storage). Instead, it is an illusion created by a device discovery process (called* `udev`*) which repeatedly polls* `sysfs`*.*

10.6 Protection

Another function of the I/O subsystem is to perform protection:

- Ensuring that only authorized processes (such as user-space drivers, or the kernel) can directly access devices.
- Ensuring that only authorized processes can access the services offered by the device driver
- Ensuring that a device cannot be configured to do some malicious to the rest of the system

There are a number of mechanisms for achieving this.

Putting device drivers in the kernel makes it easy to control access to the hardware, but you have to trust the device drivers to do the right thing since they are now part of the kernel.

Unix controls access to the drivers themselves by representing them as files, and thereby leveraging the protection model of the file system.

The last point is more difficult. 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 it's not going to ask the device to DMA somewhere it shouldn't.

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. Such a unit is called an IOMMU, and its main purpose is to provide I/O virtualization to virtual machines, which we'll cover in a later chapter.

10.7 More on I/O

There's a lot more to say about I/O, since it's one of the most important things the OS does. However, we'll see more of this later when we look at virtual memory 17, paging 18, storage 20, and networking 21.

Chapter 11

Byzantine Agreement

In order to make flying safer, researchers studied possible failures of various sensors and machines used in airplanes. While trying to model the failures, they were confronted with the following problem: Failing machines did not just crash, instead they sometimes showed arbitrary behavior before stopping completely. With these insights researchers modeled failures as arbitrary failures, not restricted to any patterns.

Definition 11.1 (Byzantine)**.** *A node which can have arbitrary behavior is called **byzantine**. This includes "anything imaginable", e.g., not sending any messages at all, or sending different and wrong messages to different neighbors, or lying about the input value.*

Remarks:

- Byzantine behavior also includes collusion, i.e., all byzantine nodes are being controlled by the same adversary.
- We assume that any two nodes communicate directly, and that no node can forge an incorrect sender address. This is a requirement, such that a single byzantine node cannot simply impersonate all nodes!
- We call non-byzantine nodes *correct* nodes.

Definition 11.2 (Byzantine Agreement)**.** *Finding consensus as in Definition 8.1 in a system with byzantine nodes is called **byzantine agreement**. An algorithm is f-resilient if it still works correctly with f byzantine nodes.*

Remarks:

- As for consensus (Definition 8.1) we also need agreement, termination and validity. Agreement and termination are straight-forward, but what about validity?

11.1 Validity

Definition 11.3 (Any-Input Validity)**.** *The decision value must be the input value of **any** node.*

Remarks:

- This is the validity definition we used for consensus, in Definition 8.1.
- Does this definition still make sense in the presence of byzantine nodes? What if byzantine nodes lie about their inputs?
- We would wish for a validity definition which differentiates between byzantine and correct inputs.

Definition 11.4 (Correct-Input Validity)**.** *The decision value must be the input value of a **correct** node.*

Remarks:

- Unfortunately, implementing correct-input validity does not seem to be easy, as a byzantine node following the protocol but lying about its input value is indistinguishable from a correct node. Here is an alternative.

Definition 11.5 (All-Same Validity)**.** *If **all** correct nodes start with the same input v, the decision value must be v.*

Remarks:

- If the decision values are binary, then correct-input validity is induced by all-same validity.
- If the input values are not binary, but for example from sensors that deliever values in $\mathbb{R}$, all-same validity is in most scenarios not really useful.

Definition 11.6 (Median Validity)**.** *If the input values are orderable, e.g. $v \in \mathbb{R}$, 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.*

Remarks:

- Is byzantine agreement possible? If yes, with what validity condition?
- Let us try to find an algorithm which tolerates 1 single byzantine node, first restricting to the so-called synchronous model.

Model 11.7 (synchronous)**.** *In the **synchronous model**, nodes operate in synchronous rounds. In each round, each node may send a message to the other nodes, receive the messages sent by the other nodes, and do some local computation.*

Definition 11.8 (synchronous runtime)**.** *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 (every legal input, every execution scenario).*

11.2 How Many Byzantine Nodes?

Algorithm 11.9 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 v
4: Store all received tuple(v, y) 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 tuple(v, y) seen in at least two sets S_v, including own S_u
8: Let tuple$(v, y) \in T$ be the tuple with the smallest value y
9: Decide on value y

Remarks:

- Byzantine nodes may not follow the protocol and send syntactically incorrect messages. Such messages can easily be deteced and discarded. It is worse if byzantine nodes send syntactically correct messages, but with a bogus content, e.g., they send different messages to different nodes.
- Some of these mistakes cannot easily be detected: For example, if a byzantine node sends different values to different nodes in the first round; such values will be put into S_u. However, some mistakes can and must be detected: Observe that all nodes only relay information in Round 2, and do not say anything about their own value. So, if a byzantine node sends a set S_v which contains a tuple(v, y), this tuple must be removed by u from S_v upon receiving it (Line 6).
- Recall that we assumed that nodes cannot forge their source address; thus, if a node receives tuple(v, y) in Round 1, it is guaranteed that this message was sent by v.

Lemma 11.10. *If $n \geq 4$, all correct nodes have the same set T.*

Proof. With $f = 1$ and $n \geq 4$ we have at least 3 correct nodes. A correct node will see every correct value at least twice, once directly from another correct node, and once through the third correct node. So all correct values are in T. If the byzantine node sends the same value to at least 2 other (correct) nodes, all correct nodes will see the value twice, so all add it to set T. If the byzantine node sends all different values to the correct nodes, none of these values will end up in any set T. □

Theorem 11.11. *Algorithm 11.9 reaches byzantine agreement if $n \geq 4$.*

Proof. We need to show agreement, any-input validity and termination. With Lemma 11.10 we know that all correct nodes have the same set T, and therefore

agree on the same minimum value. The nodes agree on a value proposed by any node, so any-input validity holds. Moreover, the algorithm terminates after two rounds. □

Remarks:

- If $n > 4$ the byzantine node can put multiple values into T.
- Algorithm 11.9 only provides any-input agreement, which is questionable in the byzantine context. One can achieve all-same validity by choosing the smallest value that occurs at least twice, if a value appears at least twice.
- The idea of this algorithm can be generalized for any f and $n > 3f$. In the generalization, every node sends in every of $f + 1$ rounds all information it learned so far to all other nodes. In other words, message size increases exponentially with f.
- Does Algorithm 11.9 also work with $n = 3$?

Theorem 11.12. *Three nodes cannot reach byzantine agreement with all-same validity if one node among them is byzantine.*

Proof. We will assume that the three nodes satisfy all-same validity and show that they will violate the agreement condition under this assumption.

In order to achieve all-same validity, nodes have to deterministically decide for a value x if it is the input value of every correct node. Recall that a Byzantine node which follows the protocol is indistinguishable from a correct node. Assume a correct node sees that $n-f$ nodes including itself have an input value x. Then, by all-same validity, this correct node must deterministically decide for x.

In the case of three nodes ($n - f = 2$) a node has to decide on its own input value if another node has the same input value. Let us call the three nodes u, v and w. If correct node u has input 0 and correct node v has input 1, the byzantine node w can fool them by telling u that its value is 0 and simultaneously telling v that its value is 1. By all-same validity, this leads to u and v deciding on two different values, which violates the agreement condition. Even if u talks to v, and they figure out that they have different assumptions about w's value, u cannot distinguish whether w or v is byzantine. □

Theorem 11.13. *A network with n nodes cannot reach byzantine agreement with $f \geq n/3$ byzantine nodes.*

Proof. Assume (for the sake of contradiction) that there exists an algorithm A that reaches byzantine agreement for n nodes with $f \geq \lceil n/3 \rceil$ byzantine nodes. We will show that A cannot satisfy all-same validity and agreement simultaneously.

Let us divide the n nodes into three groups of size $n/3$ (either $\lfloor n/3 \rfloor$ or $\lceil n/3 \rceil$, if n is not divisible by 3). Assume that one group of size $\lceil n/3 \rceil \geq n/3$ contains only Byzantine and the other two groups only correct nodes. Let one group of correct nodes start with input value 0 and the other with input value 1. As in Lemma 11.12, the group of Byzantine nodes supports the input value of each of the node, so each correct node observes at least $n - f$ nodes who support its own input value. Because of all-same validity, every correct node

has to deterministically decide on its own input value. Since the two groups of correct nodes had different input values, the nodes will decide on different values respectively, thus violating the agreement property. □

11.3 The King Algorithm

Algorithm 11.14 King Algorithm (for $f < n/3$)

```
1: x = my input value
2: for phase = 1 to f + 1 do
     Round 1
3:   Broadcast value(x)
     Round 2
4:   if some value(y) received at least n − f times then
5:     Broadcast propose(y)
6:   end if
7:   if some propose(z) received more than f times then
8:     x = z
9:   end if
     Round 3
10:  Let node v_i be the predefined king of this 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 if
15: end for
```

Lemma 11.15. *Algorithm 11.14 fulfills the all-same validity.*

Proof. If all correct nodes start with the same value, all correct nodes propose it in Round 2. All correct nodes will receive at least $n - f$ proposals, i.e., all correct nodes will stick with this value, and never change it to the king's value. This holds for all phases. □

Lemma 11.16. *If a correct node proposes x, no other correct node proposes y, with $y \neq x$, if $n > 3f$.*

Proof. Assume (for the sake of contradiction) that a correct node proposes value x and another correct node proposes value y. Since a good node only proposes a value if it heard at least $n-f$ `value` messages, we know that both nodes must have received their value from at least $n-2f$ distinct correct nodes (as at most f nodes can behave byzantine and send x to one node and y to the other one). Hence, there must be a total of at least $2(n - 2f) + f = 2n - 3f$ nodes in the system. Using $3f < n$, we have $2n - 3f > n$ nodes, a contradiction. □

Lemma 11.17. *There is at least one phase with a correct king.*

Proof. There are $f + 1$ phases, each with a different king. As there are only f byzantine nodes, one king must be correct. □

Lemma 11.18. *After a round with a correct king, the correct nodes will not change their values v anymore, if $n > 3f$.*

Proof. If all correct nodes change their values to the king's value, all correct nodes have the same value. If some correct node does not change its value to the king's value, it received a proposal at least $n - f$ times, therefore at least $n-2f$ correct nodes broadcasted this proposal. Thus, all correct nodes received it at least $n - 2f > f$ times (using $n > 3f$), therefore all correct nodes set their value to the proposed value, including the correct king. Note that only one value can be proposed more than f times, which follows from Lemma 11.16. With Lemma 11.15, no node will change its value after this round. □

Theorem 11.19. *Algorithm 11.14 solves byzantine agreement.*

Proof. The king algorithm reaches agreement as either all correct nodes start with the same value, or they agree on the same value latest after the phase where a correct node was king according to Lemmas 11.17 and 11.18. Because of Lemma 11.15 we know that they will stick with this value. Termination is guaranteed after $3(f + 1)$ rounds, and all-same validity is proved in Lemma 11.15. □

Remarks:

- Algorithm 11.14 requires $f + 1$ predefined kings. We assume that the kings (and their order) are given. Finding the kings indeed would be a byzantine agreement task by itself, so this must be done before the execution of the King algorithm.

- Do algorithms exist which do not need predefined kings? Yes, see Section 11.5.

- Can we solve byzantine agreement (or at least consensus) in less than $f + 1$ rounds?

11.4 Lower Bound on Number of Rounds

Theorem 11.20. *A synchronous algorithm solving consensus in the presence of f crashing nodes needs at least $f+1$ rounds, if nodes decide for the minimum seen value.*

Proof. Let us assume (for the sake of contradiction) that some algorithm A solves consensus in f rounds. Some node u_1 has the smallest input value x, but in the first round u_1 can send its information (including information about its value x) to only some other node u_2 before u_1 crashes. Unfortunately, in the second round, the only witness u_2 of x also sends x to exactly one other node u_3 before u_2 crashes. This will be repeated, so in round f only node u_{f+1} knows about the smallest value x. As the algorithm terminates in round f, node u_{f+1} will decide on value x, all other surviving (correct) nodes will decide on values larger than x. □

Remarks:

- A general proof without the restriction to decide for the minimum value exists as well.
- Since byzantine nodes can also just crash, this lower bound also holds for byzantine agreement, so Algorithm 11.14 has an asymptotically optimal runtime.
- So far all our byzantine agreement algorithms assume the synchronous model. Can byzantine agreement be solved in the asynchronous model?

11.5 Asynchronous Byzantine Agreement

Algorithm 11.21 Asynchronous Byzantine Agreement (Ben-Or, for $f < n/10$)

```
 1: x_u ∈ {0,1}          ◁ input bit
 2: r = 1                ◁ round
 3: decided = false
 4: Broadcast propose(x_u,r)
 5: repeat
 6:   Wait until n − f propose messages of current round r arrived
 7:   if at least n/2 + 3f + 1 propose messages contain same value x then
 8:     x_u = x, decided = true
 9:   else if at least n/2 + f + 1 propose messages contain same value x then
10:     x_u = x
11:   else
12:     choose x_u randomly, with Pr[x_u = 0] = Pr[x_u = 1] = 1/2
13:   end if
14:   r = r + 1
15:   Broadcast propose(x_u,r)
16: until decided (see Line 8)
17: decision = x_u
```

Lemma 11.22. *Let a correct node choose value x in Line 10, then no other correct node chooses value $y \neq x$ in Line 10.*

Proof. For the sake of contradiction, assume that both 0 and 1 are chosen in Line 10. This means that both 0 and 1 had been proposed by at least $n/2 + 1$ out of $n - f$ correct nodes. In other words, we have a total of at least $2 \cdot n/2 + 2 = n + 2 > n - f$ correct nodes. Contradiction! □

Theorem 11.23. *Algorithm 11.21 solves binary byzantine agreement as in Definition 11.2 for up to $f < n/10$ byzantine nodes.*

Proof. First note that it is not a problem to wait for $n - f$ propose messages in Line 6, since at most f nodes are byzantine. If all correct nodes have the same input value x, then all (except the f byzantine nodes) will propose the same value x. Thus, every node receives at least $n - 2f$ propose messages containing x. Observe that for $f < n/10$, we get $n - 2f > n/2 + 3f$ and the nodes will

decide on x in the first round already. We have established all-same validity! If the correct nodes have different (binary) input values, the validity condition becomes trivial as any result is fine.

What about agreement? Let u be the first node to decide on value x (in Line 8). Due to asynchrony another node v received messages from a different subset of the nodes, however, at most f senders may be different. Taking into account that byzantine nodes may lie (send different propose messages to different nodes), f additional propose messages received by v may differ from those received by u. Since node u had at least $n/2 + 3f + 1$ propose messages with value x, node v has at least $n/2 + f + 1$ propose messages with value x. Hence every correct node will propose x in the next round, and then decide on x.

So we only need to worry about termination: We have already seen that as soon as one correct node terminates (Line 8) everybody terminates in the next round. So what are the chances that some node u terminates in Line 8? Well, we can hope that all correct nodes randomly propose the same value (in Line 12). Maybe there are some nodes not choosing randomly (entering Line 10 instead of 12), but according to Lemma 11.22 they will all propose the same.

Thus, at worst all $n-f$ correct nodes need to randomly choose the same bit, which happens with probability $2^{-(n-f)+1}$. If so, all correct nodes will send the same propose message, and the algorithm terminates. So the expected running time is exponential in the number of nodes n in the worst case. □

Remarks:

- This Algorithm is a proof of concept that asynchronous byzantine agreement can be achieved. Unfortunately this algorithm is not useful in practice, because of its runtime.
- Note that for $f \in O(\sqrt{n})$, the probability for some node to terminate in Line 8 is greater than some positive constant. Thus, the Ben-Or algorithm terminates within expected constant number of rounds for small values of f.

Chapter Notes

The project which started the study of byzantine failures was called SIFT and was founded by NASA [WLG+78], and the research regarding byzantine agreement started to get significant attention with the results by Pease, Shostak, and Lamport [PSL80, LSP82]. In [PSL80] they presented the generalized version of Algorithm 11.9 and also showed that byzantine agreement is unsolvable for $n \leq 3f$. The algorithm presented in that paper is nowadays called *Exponential Information Gathering (EIG)*, due to the exponential size of the messages.

There are many algorithms for the byzantine agreement problem. For example the Queen Algorithm [BG89] which has a better runtime than the King algorithm [BGP89], but tolerates less failures. That byzantine agreement requires at least $f + 1$ many rounds was shown by Dolev and Strong [DS83], based on a more complicated proof from Fischer and Lynch [FL82].

While many algorithms for the synchronous model have been around for a long time, the asynchronous model is a lot harder. The only results were by

Ben-Or and Bracha. Ben-Or [Ben83] was able to tolerate $f < n/5$. Bracha [BT85] improved this tolerance to $f < n/3$.

Nearly all developed algorithms only satisfy all-same validity. There are a few exceptions, e.g., correct-input validity [FG03], available if the initial values are from a finite domain, median validity [SW15, MW18, DGM+11] if the input values are orderable, or values inside the convex hull of all correct input values [VG13, MH13, MHVG15] if the input is multidimensional.

Before the term *byzantine* was coined, the terms Albanian Generals or Chinese Generals were used in order to describe malicious behavior. When the involved researchers met people from these countries they moved – for obvious reasons – to the historic term byzantine [LSP82].

Hat tip to Peter Robinson for noting how to improve Algorithm 11.9 to all-same validity. This chapter was written in collaboration with Barbara Keller.

Bibliography

[Ben83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30. ACM, 1983.

[BG89] Piotr Berman and Juan A Garay. *Asymptotically optimal distributed consensus*. Springer, 1989.

[BGP89] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards optimal distributed consensus (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 410–415, 1989.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[DGM+11] Benjamin Doerr, Leslie Ann Goldberg, Lorenz Minder, Thomas Sauerwald, and Christian Scheideler. Stabilizing Consensus with the Power of Two Choices. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, June 2011.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[FG03] Matthias Fitzi and Juan A Garay. Efficient player-optimal protocols for strong and differential consensus. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 211–220. ACM, 2003.

[FL82] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. 14(4):183–186, June 1982.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[MH13] Hammurabi Mendes and Maurice Herlihy. Multidimensional Approximate Agreement in Byzantine Asynchronous Systems. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC, June 2013.

[MHVG15] Hammurabi Mendes, Maurice Herlihy, Nitin Vaidya, and Vijay K. Garg. Multidimensional agreement in Byzantine systems. *Distributed Computing*, 28(6):423–441, January 2015.

[MW18] Darya Melnyk and Roger Wattenhofer. Byzantine Agreement with Interval Validity. In *37th Annual IEEE International Symposium on Reliable Distributed Systems (SRDS), Salvador, Bahia, Brazil*, October 2018.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[SW15] David Stolz and Roger Wattenhofer. Byzantine Agreement with Median Validity. In *19th International Conference on Priniciples of Distributed Systems (OPODIS), Rennes, France*, 2015.

[VG13] Nitin H. Vaidya and Vijay K. Garg. Byzantine Vector Consensus in Complete Graphs. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC, July 2013.

[WLG+78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. In *Proceedings of the IEEE*, pages 1240–1255, 1978.

Chapter 12

Broadcast & Shared Coins

In Chapter 11 we have developed a fast solution for synchronous byzantine agreement (Algorithm 11.14), yet our *asynchronous* byzantine agreement solution (Algorithm 11.21) is still awfully slow. Is there a fast asynchronous algorithm, possibly based on some advanced communication methods?

12.1 Random Oracle and Bitstring

Definition 12.1 (Random Oracle)**.** *A random oracle is a trusted (non-byzantine) random source which can generate random values.*

Algorithm 12.2 Shared Coin with Magic Random Oracle

1: **return** c_i, where c_i is ith random bit by oracle

Remarks:

- Algorithm 12.2 as well as the following shared coin algorithms will for instance be called in Line 12 of Algorithm 11.21. So instead of every node throwing a local coin (and hoping that they all show the same), the nodes throw a *shared* coin. In other words, the value x_u in Line 12 of Algorithm 11.21 will be set to the return value of the shared coin subroutine.

- We have already seen a shared coin in Algorithm 8.22. This concept deserves a proper definition.

Definition 12.3 (Shared Coin)**.** *A* ***shared coin*** *is a binary random variable shared among all nodes. It is* 0 *for all nodes with constant probability, and* 1 *for all nodes with constant probability. The shared coin is allowed to fail (be* 0 *for some nodes and* 1 *for other nodes) with constant probability.*

Theorem 12.4. *Algorithm 12.2 plugged into Algorithm 11.21 solves asynchronous byzantine agreement in expected constant number of rounds.*

Proof. If there is a large majority for one of the input values in the system, all nodes will decide within two rounds since Algorithm 11.21 satisfies all-same-validity; the shared coin is not even used.

If there is no significant majority for any of the input values at the beginning of algorithm 11.21, all correct nodes will run Algorithm 12.2. Therefore, they will set their new value to the bit given by the random oracle and terminate in the following round.

If neither of the above cases holds, some of the nodes see an $n/2 + f + 1$ majority for one of the input values, while other nodes rely on the oracle. With probability $1/2$, the value of the oracle will coincide with the deterministic majority value of the other nodes. Therefore, with probability $1/2$, the nodes will terminate in the following round. The expected number of rounds for termination in this case is 3. □

Remarks:

- Unfortunately, random oracles are a bit like pink fluffy unicorns: they do not really exist in the real world. Can we fix that?

Definition 12.5 (Random Bitstring)**.** *A* ***random bitstring*** *is a string of random binary values, known to all participating nodes when starting a protocol.*

Algorithm 12.6 Naive Shared Coin with Random Bitstring

1: **return** b_i, where b_i is ith bit in common random bitstring

---

Remarks:

- But is such a precomputed bitstring really random enough? We should be worried because of Theorem 8.14.

Theorem 12.7. *If the scheduling is worst-case, Algorithm 12.6 plugged into Algorithm 11.21 does not terminate.*

Proof. We start Algorithm 12.6 with the following input: $n/2 + f + 1$ nodes have input value 1, and $n/2 - f - 1$ nodes have input value 0. Assume w.l.o.g. that the first bit of the random bitstring is 0.

If the second random bit in the bitstring is also 0, then a worst-case scheduler will let $n/2 + f + 1$ nodes see all $n/2 + f + 1$ values 1, these will therefore deterministically choose the value 1 as their new value. Because of scheduling (or byzantine nodes), the remaining $n/2 - f - 1$ nodes receive strictly less than $n/2 + f + 1$ values 1 and therefore have to rely on the value of the shared coin, which is 0. The nodes will not come to a decision in this round. Moreover, we have created the very same distribution of values for the next round (which has also random bit 0).

If the second random bit in the bitstring is 1, then a worst-case scheduler can let $n/2 - f - 1$ nodes see all $n/2 + f + 1$ values 1, and therefore deterministically choose the value 1 as their new value. Because of scheduling (or byzantine nodes), the remaining $n/2 + f + 1$ nodes receive strictly less than $n/2 + f + 1$ values 1 and therefore have to rely on the value of the shared coin, which is 0.

The nodes will not decide in this round. And we have created the symmetric situation for input value 1 that is coming in the next round.

So if the current and the next random bit are known, worst-case scheduling will keep the system in one of two symmetric states that never decide.

□

Remarks:

- Theorem 12.7 shows that a worst-case scheduler cannot be allowed to know the random bits of the future.
- Note that in the proof of Theorem 12.7 we did not even use any byzantine nodes. Just bad scheduling was enough to prevent termination.
- Worst-case scheduling is an issue that we have not considered so far, in particular in Chapter 8 we implicitly assumed that message scheduling was random. What if scheduling is worst-case in Algorithm 8.22?

Lemma 12.8. *Algorithm 8.22 has exponential expected running time under worst-case scheduling.*

Proof. In Algorithm 8.22, worst-case scheduling may hide up to f rare zero coinflips. In order to receive a zero as the outcome of the shared coin, the nodes need to generate at least $f+1$ zeros. The probability for this to happen is $(1/n)^{f+1}$, which is exponentially small for $f \in \Omega(n)$. In other words, with worst-case scheduling, with probability $1-(1/n)^{f+1}$ the shared coin will be 1. The worst-case scheduler must make sure that some nodes will always deterministically go for 0, and the algorithm needs n^{f+1} rounds until it terminates. □

Remarks:

- With worst-case asynchrony, some of our previous results do not hold anymore. Can we at least solve asynchronous (assuming worst-case scheduling) *consensus* if we have crash failures?
- This is indeed possible, but we need to sharpen our tools first.

12.2 Shared Coin on a Blackboard

Definition 12.9 (Blackboard Model)**.** *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 that have been written to the blackboard so far.*

Remarks:

- We assume that the nodes cannot reconstruct the order in which the messages are written to the blackboard, since the system is asynchronous.

Algorithm 12.10 Crash-Resilient Shared Coin with Blackboard (for node u)

1: **while** true **do**
2: Choose new local coin $c_u = +1$ with probability 1/2, else $c_u = -1$
3: Write c_u to the blackboard
4: Set C = Read all coinflips on the blackboard
5: **if** $|C| \geq n^2$ **then**
6: **return** sign(sum(C))
7: **end if**
8: **end while**

Remarks:

- In Algorithm 12.10 the outcome of a coinflip is -1 or $+1$ instead of 0 or 1 because it simplifies the analysis, i.e., "$-1 \approx 0$".
- The *sign* function is used for the decision values. The sign function returns $+1$ if the sum of all coinflips in C is positive, and -1 if it is negative.
- The algorithm is unusual compared to other asynchronous algorithms we have dealt with so far. So far we often waited for $n - f$ messages from other nodes. In Algorithm 12.10, a single node can singlehandedly generate all n^2 coinflips, without waiting.
- If a node does not need to wait for other nodes, we call the algorithm *wait-free.*
- Many similar definitions beyond wait-free exist: lock-free, deadlock-free, starvation-free, and generally non-blocking algorithms.

Theorem 12.11 (Central Limit Theorem)**.** *Let $\{X_1, X_2, \ldots, X_N\}$ be a sequence of independent random variables with $Pr[X_i = -1] = Pr[X_i = 1] = 1/2$ for all $i = 1, \ldots, N$. Then for every real number z,*

$$\lim_{N\to\infty} \Pr\left[\sum_{i=1}^{N} X_i \leq z\sqrt{N}\right] = \Phi(z) < \frac{1}{\sqrt{2\pi}} e^{-z^2/2},$$

where $\Phi(z)$ is the cumulative distribution function of the standard normal distribution evaluated at z.

Theorem 12.12. *Algorithm 12.10 implements a polynomial shared coin.*

Proof. Each node in the algorithm terminates once at least n^2 coinflips are written to the blackboard. Before terminating, nodes may write one additional coinflip. Therefore, every node decides after reading at least n^2 and at most $n^2 + n$ coinflips. The power of the adversary lies in the fact that it can prevent $n - 1$ nodes from writing their coinflips to the blackboard by delaying their writes. Here, we will consider an even stronger adversary that can hide up to n coinflips which were written on the blackboard.

We need to show that both outcomes for the shared coin ($+1$ or -1 in Line 6) will occur with constant probability, as in Definition 12.3. Let X be the sum of all coinflips that are visible to every node. Since some of the nodes might read

n more values from the blackboard than others, the nodes cannot be prevented from deciding if $|X| > n$. By applying Theorem 12.11 with $N = n^2$ and $z = 1$, we get:

$$Pr(X < -n) = Pr(X > n) = 1 - Pr(X \leq n) = 1 - \Phi(1) > 0.15.$$

□

Lemma 12.13. *Algorithm 12.10 uses n^2 coinflips, which is optimal in this model.*

Proof. The proof for showing quadratic lower bound makes use of configurations that are indistinguishable to all nodes, similar to Theorem 8.14. It requires involved stochastic methods and we therefore will only sketch the idea of where the n^2 comes from.

The basic idea follows from Theorem 12.11. The standard deviation of the sum of n^2 coinflips is n. The central limit theorem tells us that with constant probability the sum of the coinflips will be only a constant factor away from the standard deviation. As we showed in Theorem 12.12, this is large enough to disarm a worst-case scheduler. However, with much less than n^2 coinflips, a worst-case scheduler is still too powerful. If it sees a positive sum forming on the blackboard, it delays messages trying to write $+1$ in order to turn the sum temporarily negative, so the nodes finishing first see a negative sum, and the delayed nodes see a positive sum. □

Remarks:

- Algorithm 12.10 cannot tolerate even one byzantine failure: assume the byzantine node generates all the n^2 coinflips in every round due to worst-case scheduling. Then this byzantine node can make sure that its coinflips always sum up to a value larger than n, thus making the outcome -1 impossible.

- In Algorithm 12.10, we assume that the blackboard is a trusted central authority. Like the random oracle of Definition 12.1, assuming a blackboard does not seem practical. However, fortunately, we can use advanced broadcast methods in order to implement something like a blackboard with just messages.

12.3 Broadcast Abstractions

Definition 12.14 (Accept). *A message received by a node v is called* ***accepted*** *if node v can consider this message for its computation.*

Definition 12.15 (Best-Effort Broadcast). ***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.*

Remarks:

- Note that best-effort broadcast is equivalent to the simple broadcast primitive that we have used so far.
- Reliable broadcast is a stronger paradigm which implies that byzantine nodes cannot send different values to different nodes. Such behavior will be detected.

Definition 12.16 (Reliable Broadcast)**.** ***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.*

Algorithm 12.17 Asynchronous Reliable Broadcast (code for node u)

```
1: Broadcast own message msg(u)
2: if received msg(v) from node v then
3:    Broadcast echo(u, msg(v))
4: end if
5: if received echo(w, msg(v)) from n − 2f nodes w but not msg(v) then
6:    Broadcast echo(u, msg(v))
7: end if
8: if received echo(w, msg(v)) from n − f nodes w then
9:    Accept(msg(v))
10: end if
```

Theorem 12.18. *Algorithm 12.17 satisfies the following properties:*

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

Proof. We start with the first property. Assume a correct node broadcasts a message msg(v), then every correct node will receive msg(v) eventually. In Line 3, every correct node (including the originator of the message) will echo the message and, eventually, every correct node will receive at least $n - f$ echoes, thus accepting msg(v).

The second property follows from byzantine nodes being unable to forge an incorrect sender address, see Definition 11.1.

The third property deals with a byzantine originator b. If a correct node accepted message msg(b), this node must have received at least $n - f$ echoes for this message in Line 8. Since at most f nodes are byzantine, at least $n - 2f$ correct nodes have broadcast an echo message for msg(b). Therefore, every correct node will receive these $n - 2f$ echoes eventually and will broadcast an echo itself. Thus, all $n - f$ correct nodes will have broadcast an echo for msg(b) and every correct node will accept msg(b).

□

Remarks:

- Algorithm 12.17 does not terminate. Only *eventually*, all messages by correct nodes will be accepted.
- The algorithm has a linear message overhead, since every node again broadcasts every message.
- Note that byzantine nodes can issue arbitrarily many messages. This may be a problem for protocols where each node is only allowed to send one message (per round). Can we fix this, for instance with sequence numbers?

Definition 12.19 (FIFO Reliable Broadcast)**.** *The* ***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, then any node v will accept message m_1 before m_2.*

Algorithm 12.20 FIFO Reliable Broadcast (code for node u)

1: Broadcast own round r message `msg`(u, r)
2: **if** received first message `msg`(v, r) from node v for round r **then**
3:    Broadcast `echo`$(u, \text{msg}(v, r))$
4: **end if**
5: **if** not echoed any `msg'`(v, r) before **then**
6:    **if** received `echo`$(w, \text{msg}(v, r))$ from $f + 1$ nodes w but not `msg`(v, r) **then**
7:       Broadcast `echo`$(u, \text{msg}(v, r))$
8:    **end if**
9: **end if**
10: **if** received `echo`$(w, \text{msg}(v, r))$ from $n - f$ nodes w **then**
11:    **if** accepted `msg`$(v, r - 1)$ **then**
12:       Accept(`msg`(v, r))
13:    **end if**
14: **end if**

Theorem 12.21. *Algorithm 12.20 satisfies the properties of Theorem 12.18. Additionally, Algorithm 12.20 makes sure that no two messages* `msg`(v, r) *and* `msg'`(v, r) *are accepted from the same node. It can tolerate $f < n/3$ Byzantine nodes or $f < n/2$ crash failures.*

Proof. Just as reliable broadcast, Algorithm 12.20 satisfies the first two properties of Theorem 12.18 by simply following the flow of messages of a correct node.

For the third property, assume again that some message originated from a byzantine node b. If a correct node accepted message `msg`(b), this node must have received at least $n - f$ echoes for this message in Line 10.

- Byzantine case: If at most f nodes are byzantine, at least $n - 2f > f + 1$ correct nodes have broadcast an echo message for `msg`(b).
- Crash-failure case: If at most f nodes can crash, at least $n - f > f + 1$ nodes have broadcast an echo message for `msg`(b).

In both cases, every correct node will receive these $f+1$ echoes eventually and will broadcast an echo. Thus, all $n-f$ correct nodes will have broadcast an echo for msg(b) and every correct node will accept msg(b).

It remains to show that at most one message will be accepted from some node v in a round r.

- Byzantine case: Assume that some correct node u has accepted msg(v,r) in Line 12. Then, u has received $n-f$ echoes for this message, $n-2f$ of which were the first echoes of the correct nodes. Assume for contradiction that another correct node accepts msg'(v,r). This node must have collected $n-f$ messages echo(w, msg'(v,r)). Since at least $n-2f$ of these messages must be the first echo messages sent by correct nodes, we have $n-2f+n-2f=2n-4f>n-f$ (for $f<n/3$) echo messages sent by the correct nodes as their first echo. This is a contradiction.

- Crash-failure case: At least $n-2f$ not crashed nodes must have echoed msg(v,r), while $n-f$ nodes have echoed msg'(v,r). In total $2n-3f>n-f$ (for $f<n/2$) correct nodes must have echoed either of the messages, which is a contradiction.

□

Definition 12.22 (Atomic Broadcast)**.** ***Atomic broadcast*** *makes sure that all messages are received in the same order by every node. That is, for any pair of nodes u,v, and for any two messages m_1 and m_2, node u receives m_1 before m_2 if and only if node v receives m_1 before m_2.*

Remarks:

- Definition 12.22 is equivalent to Definition 7.8, i.e., atomic broadcast = state replication.

- Now we have all the tools to finally solve asynchronous consensus.

12.4 Blackboard with Message Passing

Algorithm 12.23 Crash-Resilient Shared Coin (code for node u)

```
1: while true do
2:    Choose local coin c_u = +1 with probability 1/2, else c_u = −1
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 if
10: end while
```

Theorem 12.24. *Algorithm 12.23 solves asynchronous binary agreement for $f<n/2$ crash failures.*

Proof. The upper bound for the number of crash failures results from the upper bound in 12.21. The idea of this algorithm is to simulate the read and write operations from Algorithm 12.10.

Line 3 simulates a read operation: by accepting the own coinflip, a node verifies that $n - f$ correct nodes have received its most recent generated coinflip $\texttt{coin}(c_u, r)$. At least $n - 2f > 1$ of these nodes will never crash and the value therefore can be considered as stored on the blackboard. While a value is not accepted and therefore not stored, node u will not generate new coinflips. Therefore, at any point of the algorithm, there is at most n additional generated coinflips next to the accepted coins.

Line 6 of the algorithm corresponds to a read operation. A node reads a value by requesting C_v from at least $n - f$ nodes v. Assume that for a coinflip $\texttt{coin}(c_u, r)$, f nodes that participated in the FIFO broadcast of this message have crashed. When requesting $n - f$ sets of coinflips, there will be at least $(n - 2f) + (n - f) - (n - f) = n - 2f > 1$ sets among the requested ones containing $\texttt{coin}(c_u, r)$. Therefore, a node will always read all values that were accepted so far.

This shows that the read and write operations are equivalent to the same operations in Algorithm 12.10. Assume now that some correct node has terminated after reading n^2 coinflips. Since each node reads the stored coinflips before generating a new one in the next round, there will be at most n additional coins accepted by any other node before termination. This setting is equivalent to Theorem 12.12 and the rest of the analysis is therefore analogous to the analysis in that theorem. □

Remarks:

- So finally we can deal with worst-case crash failures *and* worst-case scheduling.

- But what about byzantine agreement? We need even more powerful methods!

12.5 Using Cryptography

Definition 12.25 (Threshold Secret Sharing)**.** *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.*

Definition 12.26 (Signature)**.** *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* $\texttt{msg}(x)_u$.

Algorithm 12.27 (t, n)-Threshold Secret Sharing

1: Input: A secret s, represented as a real number.

Secret distribution by dealer d

2: Generate $t-1$ random numbers $a_1, \ldots, a_{t-1} \in \mathbb{R}$
3: Obtain a polynomial p of degree $t-1$ with $p(x) = s + a_1 x + \cdots + a_{t-1}x^{t-1}$
4: Generate n distinct $x_1, \ldots, x_n \in \mathbb{R} \setminus \{0\}$
5: Distribute share $\texttt{msg}(x_1, p(x_1))_d$ to node v_1, ..., $\texttt{msg}(x_n, p(x_n))_d$ to node v_n

Secret recovery

6: Collect t shares $\texttt{msg}(x_u, p(x_u))_d$ from at least t nodes
7: Use Lagrange's interpolation formula to obtain $p(0) = s$

Remarks:

- Algorithm 12.27 relies on a trusted dealer, who broadcasts the secret shares to the 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.

Algorithm 12.28 Preprocessing Step for Algorithm 12.29 (code for dealer d)

1: According to Algorithm 12.27, choose polynomial p of degree f
2: **for** $i = 1, \ldots, n$ **do**
3: Choose coinflip c_i, where $c_i = 0$ with probability 1/2, else $c_i = 1$
4: Using Algorithm 12.27, generate n shares $(x_1^i, p(x_1^i)), \ldots, (x_n^i, p(x_n^i))$ for c_i
5: **end for**
6: Send shares $\texttt{msg}(x_u^1, p(x_u^1))_d, \ldots, \texttt{msg}(x_u^n, p(x_u^n))_d$ to node u

Algorithm 12.29 Shared Coin using Secret Sharing (ith iteration)

1: Request shares from at least $f+1$ nodes
2: Using Algorithm 12.27, let c_i be the value reconstructed from the shares
3: **return** c_i

Theorem 12.30. *Algorithm 11.21 together with Algorithm 12.28 and Algorithm 12.29 solves asynchronous byzantine agreement for $f < n/3$ in expected* 3 *number of rounds.*

Proof. In Line 1 of Algorithm 12.29, the nodes collect shares from $f+1$ nodes. Since a byzantine node cannot forge the signature of the dealer, it is restricted to either send its own share or decide to not send it at all. Therefore, each correct node will eventually be able to reconstruct secret c_i of round i correctly in Line 2 of the algorithm. The running time analysis follows then from the analysis of Theorem 12.4. □

Remarks:

- In Algorithm 12.28 we assume that the dealer generates the random bitstring. This assumption is not necessary in general.
- We showed that cryptographic assumptions can speed up asynchronous byzantine agreement.
- Algorithm 11.21 can also be implemented in the synchronous setting.
- A randomized version of a synchronous byzantine agreement algorithm can improve on the lower bound of $t+1$ rounds for the deterministic algorithms.

Definition 12.31 (Cryptographic Hash Function). *A hash function hash : $U \to S$ is called* ***cryptographic****, if for a given $z \in S$ it is computationally hard to find an element $x \in U$ with $hash(x) = z$.*

Remarks:

- Popular hash functions used in cryptography include the Secure Hash Algorithm (SHA) and the Message-Digest Algorithm (MD).

Algorithm 12.32 Simple Synchronous Byzantine Shared Coin (for node u)

1: Each node has a public key that is known to all nodes.
2: Let r be the current round of Algorithm 11.21
3: Broadcast $\texttt{msg}(r)_u$, i.e., round number r signed by node u
4: Compute $h_v = \text{hash}(\text{msg}(r)_v)$ for all received messages $\texttt{msg}(r)_v$
5: Let $h_{min} = \min_v\ h_v$
6: **return** least significant bit of h_{min}

Remarks:

- In Algorithm 12.32, Line 3 each node can verify the correctness of the signed message using the public key.
- Just as in Algorithm 11.9, the decision value is the minimum of all received values. While the minimum value is received by all nodes after 2 rounds there, we can only guarantee to receive the minimum with constant probability in this algorithm.
- Hashing helps to restrict byzantine power, since a byzantine node cannot compute the smallest hash.

Theorem 12.33. *Algorithm 12.32 plugged into Algorithm 11.21 solves synchronous byzantine agreement in expected* 5 *rounds for up to $f < n/10$ byzantine failures.*

Proof. With probability 1/3 the minimum hash value is generated by a byzantine node. In such a case, we can assume that not all correct nodes will receive the byzantine value and thus, different nodes might compute different values for the shared coin.

With probability 2/3, the shared coin will be from a correct node, and with probability 1/2 the value of the shared coin will correspond to the value which was deterministically chosen by some of the correct nodes. Therefore, with probability 1/3 the nodes will reach consensus in the next iteration of Algorithm 11.21. The expected number of rounds is:

$$1 + \sum_{i=0}^{\infty} 2 \cdot \left(\frac{2}{3}\right)^i = 5$$

□

Chapter Notes

Asynchronous byzantine agreement is usually considered in one out of two communication models – shared memory or message passing. The first polynomial algorithm for the shared memory model that uses a shared coin was proposed by Aspnes and Herlihy [AH90] and required exchanging $O(n^4)$ messages in total. Algorithm 12.10 is also an implementation of the shared coin in the shared memory model and it requires exchanging $O(n^3)$ messages. This variant is due to Saks, Shavit and Woll [SSW91]. Bracha and Rachman [BR92] later reduced the number of messages exchanged to $O(n^2 \log n)$. The tight lower bound of $\Omega(n^2)$ on the number of coinflips was proposed by Attiya and Censor [AC08] and improved the first non-trivial lower bound of $\Omega(n^2/\log^2 n)$ by Aspnes [Asp98].

In the message passing model, the shared coin is usually implemented using reliable broadcast. Reliable broadcast was first proposed by Srikanth and Toueg [ST87] as a method to simulate authenticated broadcast. There is also another implementation which was proposed by Bracha [Bra87]. Today, a lot of variants of reliable broadcast exist, including FIFO broadcast [AAD05], which was considered in this chapter. A good overview over the broadcast routines is given by Cachin et al. [CGR14]. A possible way to reduce message complexity is by simulating the read and write commands [ABND95] as in Algorithm 12.23. The message complexity of this method is $O(n^3)$. Alistarh et al. [AAKS14] improved the number of exchanged messages to $O(n^2 \log^2 n)$ using a binary tree that restricts the number of communicating nodes according to the depth of the tree.

It remains an open question whether asynchronous byzantine agreement can be solved in the message passing model without cryptographic assumptions. If cryptographic assumptions are however used, byzantine agreement can be solved in expected constant number of rounds. Algorithm 12.28 presents the first implementation due to Rabin [Rab83] using threshold secret sharing. This algorithm relies on the fact that the dealer provides the random bitstring. Chor et al. [CGMA85] proposed the first algorithm where the nodes use verifiable secret sharing in order to generate random bits. Later work focuses on improving resilience [CR93] and practicability [CKS00]. Algorithm 12.32 by Micali [Mic18] shows that cryptographic assumptions can also help to improve the running time in the synchronous model.

This chapter was written in collaboration with Darya Melnyk.

Bibliography

[AAD05] Ittai Abraham, Yonatan Amit, and Danny Dolev. Optimal resilience asynchronous approximate agreement. In *Proceedings of the 8th International Conference on Principles of Distributed Systems*, OPODIS'04, pages 229–239, Berlin, Heidelberg, 2005. Springer-Verlag.

[AAKS14] Dan Alistarh, James Aspnes, Valerie King, and Jared Saia. Communication-efficient randomized consensus. In Fabian Kuhn, editor, *Distributed Computing*, pages 61–75, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995.

[AC08] Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *J. ACM*, 55(5):20:1–20:26, November 2008.

[AH90] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441 – 461, 1990.

[Asp98] James Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *J. ACM*, 45(3):415–450, May 1998.

[BR92] Gabriel Bracha and Ophir Rachman. Randomized consensus in expected $o(n^2 logn)$ operations. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, WDAG '91, pages 143–150, Berlin, Heidelberg, 1992. Springer-Verlag.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130 – 143, 1987.

[CGMA85] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 383–395, Oct 1985.

[CGR14] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2014.

[CKS00] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18:219–246, 2000.

[CR93] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 42–51, New York, NY, USA, 1993. ACM.

[Mic18] Silvio Micali. Byzantine agreement , made trivial. 2018.

[Rab83] M. O. Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409, Nov 1983.

[SSW91] Michael Saks, Nir Shavit, and Heather Woll. Optimal time randomized consensus – making resilient algorithms fast in practice. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '91, pages 351–362, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.

[ST87] T. K. Srikanth and S. Toueg. Optimal Clock Synchronization. *Journal of the ACM*, 34:626–645, 1987.

Chapter 13

Consistency & Logical Time

You submit a comment on your favorite social media platform using your phone. The comment is immediately visible on the phone, but not on your laptop. Is this level of consistency acceptable?

13.1 Consistency Models

Definition 13.1 (Object)**.** *An **object** is a variable or a data structure storing information.*

Remarks:

- Object is a general term for any entity that can be modified, like a queue, stack, memory slot, file system, etc.

Definition 13.2 (Operation)**.** *An **operation** f accesses or manipulates an object. The operation f starts at wall-clock time f_* and ends at wall-clock time $f_\dagger$.*

Remarks:

- An operation can be as simple as extracting an element from a data structure, but an operation may also be more complex, like fetching an element, modifying it and storing it again.
- If $f_\dagger < g_*$, we simply write $f < g$.

Definition 13.3 (Execution)**.** *An **execution** E is a set of operations on one or multiple objects that are executed by a set of nodes.*

Definition 13.4 (Sequential Execution)**.** *An execution restricted to a single node is a **sequential execution**. All operations are executed sequentially, which means that no two operations f and g are concurrent, i.e., we have $f < g$ or $g < f$.*

Definition 13.5 (Semantic Equivalence)**.** *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.*

Remarks:

- For example, when dealing with a stack object, corresponding `pop` operations in two different semantically equivalent executions must yield the same element of the stack.
- In general, the notion of semantic equivalence is non-trivial and dependent on the type of the object.

Definition 13.6 (Linearizability)**.** *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.*

Definition 13.7. *A* ***linearization point*** *of operation f is some $f_\bullet \in [f_*, f_\dagger]$.*

Lemma 13.8. *An execution 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.*

Proof. Let f and g be two operations in E with $f_\dagger < g_*$. Then by definition of linearization points we also have $f_\bullet < g_\bullet$ and therefore $f < g$ in S. □

Definition 13.9 (Sequential Consistency)**.** *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.*

Lemma 13.10. *Every linearizable execution is also sequentially consistent, i.e., linearizability $\implies$ sequential consistency.*

Proof. Since linearizability (order of operations *on any* nodes must be respected) is stricter than sequential consistency (only order of operations *on the same node* must be respected), the lemma follows immediately. □

Definition 13.11 (Quiescent 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.*

Lemma 13.12. *Every linearizable execution is also quiescently consistent, i.e., linearizability $\implies$ quiescent consistency.*

Proof. Let E be the original execution and S be the semantically equivalent sequential execution. Let t be a quiescent point and consider two operations g, h with $g_\dagger < t < h_*$. Then we have $g < h$ in S. This is also guaranteed by linearizability since $g_\dagger < t < h_*$ implies $g < h$. □

Lemma 13.13. *Sequentially consistent and quiescent consistency do not imply one another.*

Proof. There are executions that are sequentially consistent but not quiescently consistent. An object initially has value 2. We apply two operations to this object: *inc* (increment the object by 1) and *double* (multiply the object by 2). Assume that $inc < double$, but *inc* and *double* are executed on different nodes. Then a result of 5 (first double, then inc) is sequentially consistent but not quiescently consistent.

There are executions that are quiescently consistent but not sequentially consistent. An object initially has value 2. Assume to have three operations on two nodes u and v. Node u calls first *inc* then *double*, node v calls *inc* once with $inc^v_* < inc^u_\dagger < double^u_* < inc^v_\dagger$. Since there is no quiescent point, quiescent consistency is okay with a sequential execution that doubles first, resulting in $((2 \cdot 2) + 1) + 1 = 6$. The sequential execution demands that $inc^u < double^u$, hence the result should be strictly larger than 6 (either 7 or 8). □

Definition 13.14. *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.*

Remarks:

- In the introductory social media example, a linearizable implementation would have to make sure that the comment is immediately visible on any device, as the *read* operation starts after the *write* operation finishes. If the system is only sequentially consistent, the comment does not need to be immediately visible on every device.

Definition 13.15 (restricted execution)**.** *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.*

Definition 13.16. *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.*

Remarks:

- Composability enables to implement, verify and execute multiple concurrent objects independently.

Lemma 13.17. *Sequential consistency is not composable.*

Proof. We consider an execution E with two nodes u and v, which operate on two objects x and y initially set to 0. The operations are as follows: u_1 reads $x = 1$, u_2 writes $y := 1$, v_1 reads $y = 1$, v_2 writes $x := 1$ with $u_1 < u_2$ on node u and $v_1 < v_2$ on node v. It is clear that $E|x$ as well as $E|y$ are sequentially consistent as the write operations may be before the respective read operations. In contrast, execution E is *not* sequentially consistent: Neither u_1 nor v_1 can possibly be the initial operation in any correct semantically equivalent sequential execution S, as that would imply reading 1 when the variable is still 0. □

Theorem 13.18. *Linearizability is composable.*

Proof. Let E be an execution composed of multiple restricted executions $E|x$. For any object x there is a sequential execution $S|x$ that is semantically consistent to $E|x$ and in which the operations are ordered according to wall-clock-linearization points. Let S be the sequential execution ordered according to all linearization points of all executions $E|x$. S is semantically equivalent to E as $S|x$ is semantically equivalent to $E|x$ for all objects x and two object-disjoint executions cannot interfere. Furthermore, if $f_\dagger < g_*$ in E, then also $f_\bullet < g_\bullet$ in E and therefore also $f < g$ in S. □

13.2 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.

Definition 13.19. *Let S_u be a sequence of operations on some node u and define "$\rightarrow$" to be the* ***happened-before relation*** *on $E := S_1 \cup \cdots \cup S_n$ that satisfies the following three conditions:*

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

Remarks:

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

Definition 13.20 (Happened-before consistency)**.** *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 $f \rightarrow g$ for two operations f, g in E, then also $f < g$ in S.*

Lemma 13.21. *Happened-before consistency = sequential consistency.*

Proof. Both consistency models execute all operations of a single node in the sequential order. In addition, happened-before consistency also respects messages between nodes. However, messages are also ordered by sequential consistency because of semantic equivalence (a receive cannot be before the corresponding send). Finally, even though transitivity is defined more formally in happened-before consistency, also sequential consistency respects transitivity.

In addition, sequential consistency orders two operations o_u, o_v on two different nodes u, v if o_v can see a state change caused by o_u. Such a state change does not happen out of the blue, in practice some messages between u and v (maybe via "shared blackboard" or some other form of communication) will be involved to communicate the state change. □

Definition 13.22 (Logical clock)**.** *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 $\rightarrow$ is respected, i.e., for two operations g on node u and h on node v*

$$g \rightarrow h \implies c_u(g) < c_v(h).$$

Definition 13.23. *If it additionally holds that $c_u(g) < c_v(h) \implies g \rightarrow h$, then the clock is called a* ***strong logical clock****.*

Remarks:

- In algorithms we write c_u for the current logical time of node u.
- The simplest logical clock is the *Lamport clock*, given in Algorithm 13.24. Every message includes a timestamp, such that the receiving node may update its current logical time.

Algorithm 13.24 Lamport clock

1: (Code for node u)
2: Initialize $c_u := 0$.
3: Upon local operation: Increment current local time $c_u := c_u + 1$.
4: Upon send operation: Increment $c_u := c_u + 1$ and include c_u as T in message.
5: Upon receive operation: Extract T from message and update $c_u := \max(c_u, T) + 1$.

Theorem 13.25. *Lamport clocks are logical clocks.*

Proof. If for two operations f, g it holds that $f \rightarrow g$, then according to the definition three cases are possible.

1. If $f < g$ on the same node u, then $c_u(f) < c_u(g)$.
2. Let g be a receive operation on node v corresponding to some send operation f on another node u. We have $c_v(g) \geq T + 1 = c_u(f) + 1 > c_u(f)$.
3. Transitivity follows with $f \rightarrow g$ and $g \rightarrow h \Rightarrow g \rightarrow h$, and the first two cases.

□

Remarks:

- Lamport logical clocks are not strong logical clocks, which means we cannot completely reconstruct $\rightarrow$ from the family of clocks c_u.
- To achieve a strong logical clock, nodes also have to gather information about other clocks in the system, i.e., node u needs to have a idea of node v's clock, for every u, v. This is what *vector clocks* in Algorithm 13.26 do: Each node u stores its knowledge about other node's logical clocks in an n-dimensional vector c_u.

Algorithm 13.26 Vector clocks

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

Theorem 13.27. *Define $c_u < c_v$ if and only if $c_u[w] \leq c_v[w]$ for all entries w, and $c_u[x] < c_v[x]$ for at least one entry x. Then the vector clocks are strong logical clocks.*

Proof. We are given two operations f, g, with operation f on node u, and operation g on node v, possibly $v = u$.

If we have $f \to g$, then there must be a happened-before-path of operations and messages from f to g. According to Algorithm 13.26, $c_v(g)$ must include at least the values of the vector $c_u(f)$, and the value $c_v(g)[v] > c_u(f)[v]$.

If we do not have $f \to g$, then $c_v(g)[u]$ cannot know about $c_u(f)[u]$, and hence $c_v(g)[u] < c_u(f)[u]$, since $c_u(f)[u]$ was incremented when executing f on node u.

□

Remarks:

- Usually the number of interacting nodes is small compared to the overall number of nodes. Therefore we do not need to send the full length clock vector, but only a vector containing the entries of the nodes that are actually communicating. This optimization is the called the *differential technique.*

13.3 Consistent Snapshots

Definition 13.28 (cut)**.** *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.*

Definition 13.29 (consistent snapshot)**.** *A cut C is a* ***consistent snapshot****, if for every operation g in C with $f \to g$, C also contains f.*

Remarks:

- In a consistent snapshot it is forbidden to see an effect without its cause.

- The number of possible consistent snapshots gives also information about the degree of concurrency of the system.

- One extreme is a sequential computation, where stopping one node halts the whole system. Let q_u be the number of operations on node $u \in \{1, \ldots, n\}$. Then the number of consistent snapshots (including the empty cut) in the sequential case is $\mu_s := 1 + q_1 + q_2 + \cdots + q_n$.

- One the other hand, in an entirely concurrent computation the nodes are not dependent on one another and therefore stopping one node does not impact others. The number of consistent snapshots in this case is $\mu_c := (1 + q_1) \cdot (1 + q_2) \cdots (1 + q_n)$.

Definition 13.30 (measure of concurrency)**.** *The concurrency measure of an execution $E = (S_1, \ldots, S_n)$ is defined as the ratio*

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

where μ denotes the number of consistent snapshot of E.

Remarks:

- This measure of concurrency is normalized to $[0, 1]$. F

- In order to evaluate the extent to which a computation is concurrent, we need to compute the number of consistent snapshots μ. This can be done via vector clocks.

- Imagine a bank having lots of accounts with transactions all over the world. The bank wants to make sure that at no point in time money gets created or destroyed. This is where consistent snapshots come in: They are supposed to capture the state of the system. Theoretically, we have already used snapshots when we discussed configurations in Definition 8.4:

Definition 13.31 (configuration)**.** *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).*

Remarks:

- One application of consistent snapshots is to check if certain invariants hold in a distributed setting. Other applications include distributed debugging or determining global states of a distributed system.

- In Algorithm 13.32 we assume that a node can record only its internal state and the messages it sends and receives. There is no common clock so it is not possible to just let each node record all information at precisely the same time.

Theorem 13.33. *Algorithm 13.32 collects a consistent snapshot.*

Algorithm 13.32 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.

Proof. Let C be the cut induced by the frontier of all states and messages forwarded to the initiator. For every node u, let t_u be the time when u gets the first snap message m (either by the initiator, or as a message tag). Then C contains all of u's operations before t_u, and none after t_u (also not the message m which arrives together with the tag at t_u).

Assume for the sake of contradiction we have operations f, g on nodes u, v respectively, with $f \rightarrow g$, $f \notin C$ and $g \in C$, hence $t_u \leq f$ and $g < t_v$. If $u = v$ we have $t_u \leq f < g < t_v = t_u$, which is a contradiction. On the other hand, if $u \neq v$: Since $t_u \leq f$ we know that all following send operations must have included the snap tag. Because of $f \rightarrow g$ we know there is a path of messages between f and g, all including the snap tag. So the snap tag must have been received by node v before or with operation g, hence $t_v \leq g$, which is a contradiction to $t_v > g$. □

Remarks:

- It may of course happen that a node u sends a message m before receiving the first snap message at time t_u (hence not containing the snap tag), and this message m is only received by node v after t_v. Such a message m will be reported by v, and is as such included in the consistent snapshot (as a message that was *in transit* during the snapshot).

13.4 Distributed Tracing

Definition 13.34 (Microservice Architecture)**.** *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").*

Remarks:

- There is no exact definition for microservices. A rule of thumb is that you should be able to program a microservice from scratch within two weeks.

- Microservices are the architecture of choice to implement a cloud based distributed system, as they allow for different technology stacks, often also simplifying scalability issues.

- In contrast to a monolithic architecture, debugging and optimizing get trickier as it is difficult to detect which component exactly is causing problems.
- 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.

Definition 13.35 (Span)**.** *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 $s_\dagger$.*

Remarks:

- 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.

Definition 13.36 (Span Reference)**.** *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 (the parents asks the child and the child answers), 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).*

Definition 13.37 (Trace)**.** *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.*

Remarks:

- The advantage of using an open source definition like opentracing is that it is easy to replace a specific tracing by another one. This mitigates the lock-in effect that is often experienced when using some specific technology.
- Algorithm 13.38 shows what is needed if you want to trace requests to your system.

Algorithm 13.38 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.

Remarks:

- All tracing information is collected and has to be sent to some tracing backend which stores the traces and usually provides a frontend to understand what is going on.
- Opentracing implementations are available for the most commonly used programming frameworks and can therefore be used for heterogeneous collections of microservices.

13.5 Mutual Exclusion

When multiple nodes compete for exclusive access to a shared resource, we need a protocol which coordinates the order in which the resource gets assigned to the nodes. The most obvious algorithm is letting a leader node organize everything:

Algorithm 13.39 Centralized Mutual Exclusion Algorithm

1: To access shared resource: Send request message to leader and wait for permission.
2: To release shared resource: Send release message to leader.

Remarks:

- An advantage of Algorithm 13.39 is its simplicity and low message overhead with 3 messages per access.
- An obvious disadvantage is that the leader is single point of failure and performance bottleneck. Assuming an asynchronous system, this protocol also does not achieve first come first serve fairness.
- To eliminate the single bottleneck we pass an access token from node to node. This token contains the time t of the earliest known outstanding request.
- We assume a ring of nodes, i.e., there is an order of the nodes given such that every node knows its successor and predecessor.

Algorithm 13.40 Token-Based Mutual Exclusion Algorithm

1: To access shared resource at time T_R: Wait for token containing time t of earliest known outstanding request.
2: Upon receiving token:
3: **if** $T_R = t$ **then**
4:    Hold token and access shared resource.
5: **else if** $T_R > t$ **then**
6:    Pass on token to next node.
7: **else if** $t = null$ or $T_R < t$ **then**
8:    Set $t = T_R$ and pass on token.
9: **end if**
10: To release access: Set $t = null$ and pass on token.

Remarks:

- Algorithm 13.40 achieves in-order fairness if all nodes stick to the rules.
- One issue is the breakdown if one node does not manage to pass on the token. In this case some new token has to be created and assigned to one of the remaining nodes.
- We can get rid of the token, if access to the token gets decided on a first come first serve basis with respect to logical clocks. This leads to Algorithm 13.41.

Algorithm 13.41 Distributed Mutual Exclusion Algorithm

1: To access shared resource: Send message to all nodes containing the node ID and the current timestamp.
2: Upon received request message: If access to the same resource is needed and the own timestamp is lower than timestamp in received message, **defer** response. Otherwise send back a response.
3: Upon responses **from all nodes** received: enter critical section. Afterwards send deferred responses.

Remarks:

- The algorithm guarantees mutual exclusion without deadlocks or starvation of a requesting process.
- The number of messages per entry is $2(n-1)$, where n is the number of nodes in the system: $(n-1)$ requests and $(n-1)$ responses.
- There is no single point of failure. Yet, whenever a node crashes, it will not reply with a response and the requesting node waits forever. Eeven worse, the requesting process cannot determine if the silence is due to the other process currently accessing the shared resource or crashing. Can we fix this? Indeed: Change step 2 in Algorithm 13.41 such that upon receiving request there will always be an answer, either Denied or OK. This way crashes will be detected.

Chapter Notes

In his seminal work, Leslie Lamport came up with the happened-before relation and gave the first logical clock algorithm [Lam78]. This paper also laid the foundation for the theory of logical clocks. Fidge came some time later up with vector clocks [JF88]. An obvious drawback of vector clocks is the overhead caused by including the whole vector. Can we do better? In general, we cannot if we need strong logical clocks [CB91].

Lamport also introduced the algorithm for distributed snapshots, together with Chandy [CL85]. Besides this very basic algorithm, there exist several other algorithms, e.g., [LY87], [SK86].

Throughout the literature the definitions for, e.g., consistency or atomicity slightly differ. These concepts are studied in different communities, e.g., linearizability hails from the distributed systems community whereas the notion of serializability was first treated by the database community. As the two areas converged, the terminology got overloaded.

Our definitions for distributed tracing follow the OpenTracing API [1]. The opentracing API only gives high-level definitions of how a tracing system is supposed to work. Only the implementation specifies how it works internally.There are several systems that implement these generic definitions, like Uber's open source tracer called *Jaeger*, or *Zipkin*, which was first developed by Twitter. This technology is relevant for the growing number of companies that embrace a microservice architecture. Netflix for example has a growing number of over 1,000 microservices.

This chapter was written in collaboration with Julian Steger.

Bibliography

[CB91] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, July 1991.

[CL85] K Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. 3:63–75, 02 1985.

[JF88] Colin J. Fidge. Timestamps in message-passing systems that preserve partial ordering. 10:56–66, 02 1988.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978.

[LY87] Ten H. Lai and Tao H. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153 – 158, 1987.

[SK86] Madalene Spezialetti and Phil Kearns. Efficient distributed snapshots. In *ICDCS*, pages 382–388. IEEE Computer Society, 1986.

[1] http://opentracing.io/documentation/

Chapter 14

Time, Clocks & GPS

> "A man with a clock knows what time it is – a man with two is never sure." (Segal's Law)

14.1 Time & Clocks

Definition 14.1 (Second)**.** *A* ***second*** *is the time that passes during 9,192,631,770 oscillation cycles of a caesium-133 atom.*

Remarks:

- This definition is a bit simplified. The official definition is given by the *Bureau International des Poids et Mesures*.
- Historically, a second was defined as one in 86,400 parts of a day, dividing the day into 24 hours, 60 minutes and 60 seconds.
- Since the duration of a day depends on the unsteady rotation cycle of the Earth, the novel oscillation-based definition has been adopted. Leap seconds are used to keep time synchronized to Earth's rotation.

Definition 14.2 (Wall-Clock Time)**.** *The* ***wall-clock time*** t^* *is the true time (a perfectly accurate clock would show).*

Definition 14.3 (Clock)**.** *A* ***clock*** *is a device which tracks and indicates time.*

Remarks:

- A clock's time t is a function of the wall-clock time t^*, i.e., $t = f(t^*)$. Ideally, $t = t^*$, but in reality there are often errors.

Definition 14.4 (Clock Error)**.** *The* ***clock error*** *or clock skew is the difference between two clocks, e.g.,* $t - t^*$ *or* $t - t'$*. In practice the clock error is often modeled as* $t = (1 + \delta)t^* + \xi(t^*)$*.*

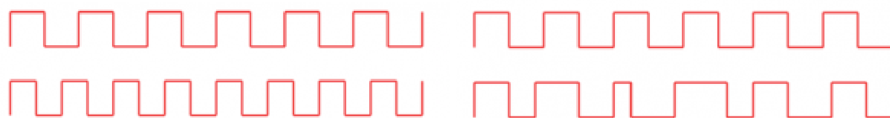


Figure 14.8: Drift (left) and Jitter (right). On top is a square wave, the wall-clock time t^*.

Remarks:

- The importance of accurate timekeeping and clock synchronization is reflected in the following statement by physicist Steven Jefferts: "We've learned that every time we build a better clock, somebody comes up with a use for it that you couldn't have foreseen."

Definition 14.5 (Drift)**.** *The **drift** δ is the predictable clock error.*

Remarks:

- Drift is relatively constant over time, but may change with supply voltage, temperature and age of an oscillator.

- Stable clock sources, which offer a low drift, are generally preferred, but also more expensive, larger and more power hungry, which is why many consumer products feature inaccurate clocks.

Definition 14.6 (ppm)**.** *Clock drift is indicated in **parts-per-million (ppm)**. One ppm corresponds to a time error growth of one microsecond per second.*

Remarks:

- In PCs, the so-called *real-time clock* normally is a crystal oscillator with a maximum drift between 5 and 100 ppm.

- Applications in signal processing, for instance GPS, need more accurate clocks. Common drift values are 0.5 to 2 ppm.

Definition 14.7 (Jitter)**.** *The **jitter** ξ is the unpredictable, random noise of the clock error.*

Remarks:

- In other words, jitter is the irregularity of the clock. Unlike drift, jitter can vary fast.

- Jitter captures all the errors that are not explained by drift. Figure 14.8 visualizes the concepts.

14.2 Clock Synchronization

Definition 14.9 (Clock Synchronization)**.** ***Clock synchronization*** *is the process of matching multiple clocks (nodes) to have a common time.*

Remarks:

- A trade-off exists between synchronization accuracy, convergence time, and cost.
- Different clock synchronization variants may tolerate crashing, erroneous or byzantine nodes.

Algorithm 14.10 Network Time Protocol NTP

1: Two nodes, client u and server v

2: **while** true **do**
3: Node u sends request to v at time t_u
4: Node v receives request at time t_v
5: Node v processes the request and replies at time t'_v
6: Node u receives the response at time t'_u

7: Propagation delay $\delta = \frac{(t'_u - t_u) - (t'_v - t_v)}{2}$ (assumption: symmetric)
8: Clock skew $\theta = \frac{(t_v - (t_u + \delta)) - (t'_u - (t'_v + \delta))}{2} = \frac{(t_v - t_u) + (t'_v - t'_u)}{2}$
9: Node u adjusts clock by $+\theta$
10: Sleep before next synchronization
11: **end while**

Remarks:

- Many NTP servers are public, answering to UDP packets.
- The most accurate NTP servers derive their time from atomic clocks, synchronized to UTC. To reduce those server's load, a hierarchy of NTP servers is available in a forest (multiple trees) structure.
- The regular synchronization of NTP limits the maximum error despite unpredictable clock errors. Synchronizing clocks just once is only sufficient for a short time period.

Definition 14.11 (PTP)**.** *The* ***Precision Time Protocol (PTP)*** *is a clock synchronization protocol similar to NTP, but which uses* ***medium access control (MAC)*** *layer timestamps.*

Remarks:

- MAC layer timestamping removes the unknown time delay incurred through messages passing through the software stack.
- PTP can achieve sub-microsecond accuracy in local networks.

Definition 14.12 (Global Synchronization)**.** ***Global*** *synchronization establishes a common time between* ***any*** *two nodes in the system.*

Remarks:

- For example, email needs global timestamps. Also, event detection for power grid control and earthquake localization need global timestamps.

- Earthquake localization does not need real-time synchronization; it is sufficient if a common time can be reconstructed when needed, also known as "post factum" synchronization.

- NTP and PTP are both examples of clock synchronization algorithms that optimize for global synchronization.

- However, two nodes that constantly communicate may receive their timestamps through different paths of the NTP forest, and hence they may accumulate different errors. Because of the clock skew, a message sent by node u might arrive at node v with a timestamp in the future.

Algorithm 14.13 Local Time Synchronization

```
1: while true do
2:    Exchange current time with neighbors
3:    Adapt time to neighbors, e.g., to average or median
4:    Sleep before next synchronization
5: end while
```

Definition 14.14 (Local Synchronization)**.** ***Local*** *synchronization establishes a common time between close-by (neighbor) nodes.*

Remarks:

- Local synchronization is the method of choice to establish *time-division multiple access (TDMA)* and coordination of wake-up and sleeping times in wireless networks. Only close-by nodes matter as far-away nodes will not interfere with their transmissions.

- Local synchronization is also relevant for precise event localization. For instance, using the speed of sound, measured sound arrival times from co-located sensors can be used to localize a shooter.

- While global synchronization algorithm such as NTP usually synchronize to an external time standard, local algorithms often just synchronize among themselves, i.e., the notion of time does not reflect any time standards.

- In wireless networks, one can simplify and improve synchronization.

Algorithm 14.15 Wireless Clock Synchronization with Known Delays

1: Given: transmitter s, receivers u, v, with known transmission delays d_u, d_v from transmitter s, respectively.

2: s sends signal at time t_s
3: u receives signal at time t_u
4: v receives signal at time t_v

5: $\Delta_u = t_u - (t_s + d_u)$
6: $\Delta_v = t_v - (t_s + d_v)$

7: Clock skew between u and v: $\theta = \Delta_v - \Delta_u = t_v - d_v + d_u - t_u$

14.3 Time Standards

Definition 14.16 (TAI)**.** *The **International Atomic Time (TAI)** is a time standard derived from over 400 atomic clocks distributed worldwide.*

Remarks:

- Using a weighted average of all involved clocks, TAI is an order of magnitude more stable than the best clock.
- The involved clocks are synchronized using simultaneous observations of GPS or geostationary satellite transmissions using Algorithm 14.15.
- While a single satellite measurement has a time uncertainty on the order of nanoseconds, averaging over a month improves the accuracy by several orders of magnitude.

Definition 14.17 (Leap Second)**.** *A leap second is an extra second added to a minute to make it irregularly 61 instead of 60 seconds long.*

Remarks:

- Time standards use leap seconds to compensate for the slowing of the Earth's rotation. In theory, also negative leap seconds can be used to make some minutes only 59 seconds long. But so far, this was never necessary.
- For easy implementation, not all time standards use leap seconds, for instance TAI and GPS time do not.

Definition 14.18 (UTC)**.** *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.*

Remarks:

- The global time standard *Greenwich Mean Time (GMT)* was already established in 1884. With the invention of caesium atomic clocks and the subsequent redefinition of the SI second, UTC replaced GMT in 1967.
- Before time standards existed, each city set their own time according to the local mean solar time, which is difficult to measure exactly. This was changed by the upcoming rail and communication networks.
- Different notations for time and date are in use. 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: `1712-02-30T07:39:52Z`. `T` separates the date and time parts while `Z` indicates the time zone with zero offset from UTC.
- Why UTC and not "CUT"? Because France insisted. Same for other abbreviations in this domain, e.g. TAI.

Definition 14.19 (Time Zone)**.** *A **time zone** is a geographical region in which the same time offset from UTC is officially used.*

Remarks:

- Time zones serve to roughly synchronize noon with the sun reaching the day's highest apparent elevation angle.
- Some time zones' offset is not a whole number of hours, e.g. India.

14.4 Clock Sources

Definition 14.20 (Atomic Clock)**.** *An **atomic clock** is a clock which keeps time by counting oscillations of atoms.*

Remarks:

- Atomic clocks are the most accurate clocks known. They can have a drift of only about one second in 150 million years, about 2e-10 ppm!
- Many atomic clocks are based on caesium atoms, which led to the current definition of a second. Others use hydrogen-1 or rubidium-87.
- In the future, atoms with higher frequency oscillations could yield even more accurate clocks.
- Atomic clocks are getting smaller and more energy efficient. Chip-scale atomic clocks (CSAC) are currently being produced for space applications and may eventually find their way into consumer electronics.
- Atomic clocks can serve as a fallback for GPS time in data centers.

Definition 14.21 (System Clock)**.** *The **system clock** in a computer is an oscillator used to synchronize all components on the motherboard.*

Remarks:

- Usually, a quartz crystal oscillator with a frequency of some tens to hundreds MHz is used.
- Therefore, the system clock can achieve a precision of some ns!
- The *CPU clock* is usually a multiple of the system clock, generated from the system clock through a clock multiplier.
- To guarantee nominal operation of the computer, the system clock must have low jitter. Otherwise, some components might not get enough time to complete their operation before the next (early) clock pulse arrives.
- Drift however is not critical for system stability.
- Applications of the system clock include thread scheduling and ensuring smooth media playback.
- If a computer is shut down, the system clock is not running; it is reinitialized when starting the computer.

Definition 14.22 (RTC)**.** *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.*

Remarks:

- The RTC is read at system startup to initialize the system clock.
- This keeps the computer's time close to UTC even when the time cannot be synchronized over a network.
- RTCs are relatively inaccurate, with a common maximum drift of 5, 20 or even 100 ppm, depending on quality and temperature.
- In many cases, the RTC frequency is 32.768 kHz, which allows for simple timekeeping based on binary counter circuits because the frequency is exactly 2^{15} Hz.

Definition 14.23 (Radio Time Signal)**.** *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.*

Remarks:

- Time signal stations use atomic clocks to send as accurate time codes as possible.
- Radio-controlled clocks are an example application of radio signal time synchronization.
- In Europe, most radio-controlled clocks use the signal transmitted by the *DCF77* station near Frankfurt, Germany.

- Radio time signals can be received much farther than the horizon of the transmitter due to signal reflections at the ionosphere. DCF77 for instance has an official range of 2,000 km.
- The time synchronization accuracy when using radio time signals is limited by the propagation delay of the signal. For instance the delay Frankfurt-Zurich is about 1 ms.

Definition 14.24 (Power Line Clock)**.** *A **power line clock** measures the oscillations from electric AC power lines, e.g. 50 Hz.*

Remarks:

- Clocks in kitchen ovens are usually driven by power line oscillations.
- AC power line oscillations drift about 10 ppm, which is remarkably stable.
- The magnetic field radiating from power lines is strong enough that power line clocks can work wirelessly.
- Power line clocks can be synchronized by matching the observed noisy power line oscillation patterns.
- Power line clocks operate with as little as a few ten µW.

Definition 14.25 (Sunlight Time Synchronization)**.** ***Sunlight time synchronization** is a method of reconstructing global timestamps by correlating annual solar patterns from light sensors' length of day measurements.*

Remarks:

- Sunlight time synchronization is relatively inaccurate.
- Due to low data rates from length of day measurements, sunlight time synchronization is well-suited for long-time measurements with data storage and post-processing, requiring no communication at the time of measurement.
- Historically, sun and lunar observations were the first measurements used for time determination. Some clock towers still feature sun dials.
- ... but today, the most popular source of time is probably GPS!

14.5 GPS

Definition 14.26 (Global Positioning System)**.** *The **Global Positioning System (GPS)** is a **Global Navigation Satellite System (GNSS)**, consisting of at least 24 satellites orbiting around the Earth, each continuously transmitting its position and time code.*

Remarks:

- Positioning is done in space and *time*!
- GPS provides position and time information to receivers anywhere on Earth where at least four satellite signals can be received.
- Line of sight (LOS) between satellite and receiver is advantageous. GPS works poorly indoors, or with reflections.
- Besides the US GPS, three other GNSS exist: the European Galileo, the Russian GLONASS and the Chinese BeiDou.
- GPS satellites orbit around Earth approximately 20,000 km above the surface, circling Earth twice a day. The signals take between 64 and 89 ms to reach Earth.
- The orbits are precisely determined by ground control stations, optimized for a high number of satellites being concurrently above the horizon at any place on Earth.

Algorithm 14.27 GPS Satellite

```
1: Given: Each satellite has a unique 1023 bit (±1, see below) PRN sequence,
   plus some current navigation data D (also ±1).
2: The code below is a bit simplified, concentrating on the digital aspects,
   ignoring that the data is sent on a carrier frequency of 1575.42 MHz.

3: while true do
4:   for all bits D_i ∈ D do
5:     for j = 0...19 do
6:       for k = 0...1022 do {this loop takes exactly 1 ms}
7:         Send bit PRN_k · D_i
8:       end for
9:     end for
10:  end for
11: end while
```

Definition 14.28 (PRN)**.** ***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.*

Remarks:

- The GPS PRN sequences are so-called *Gold codes*, which have low cross-correlation with each other.
- To simplify our math (abstract from modulation), each PRN bit is either 1 or -1.

Definition 14.29 (Navigation Data)**.** ***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.*

Remarks:

- As seen in Algorithm 14.27 each bit is repeated 20 times for better robustness. Thus, the navigation data rate is only 50 bit/s.
- Due to this limited data rate, timestamps are sent every 6 seconds, satellite orbit parameters (function of the satellite position over time) only every 30 seconds. As a result, the latency of a first position estimate after turning on a receiver, which is called *time-to-first-fix (TTFF)*, can be high.

Definition 14.30 (Circular Cross-Correlation)**.** *The **circular cross-correlation** is a similarity measure between two vectors of length N, **circularly** shifted by a given displacement d:*

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

Remarks:

- The two vectors are most similar at the displacement d where the sum (cross-correlation value) is maximum.
- The vector of cross-correlation values with all N displacements can efficiently be computed using a fast Fourier transform (FFT) in $\mathcal{O}(N \log N)$ instead of $\mathcal{O}(N^2)$ time.

Algorithm 14.31 Acquisition

1: Received 1 ms signal $\boldsymbol{s}$ with sampling rate $r \cdot 1,023$ kHz
2: Possible Doppler shifts F, e.g. {-10 kHz, -9.8 kHz, ..., +10 kHz}
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 $f \in F$ **do**
7: Build modulated PRN''_i with PRN'_i and Doppler frequency f
8: **for all** delays $d \in \{0, 1, \dots, 1,023 \cdot r - 1\}$ **do**
9: $A_i(f, d) = |cxcorr(\boldsymbol{s}, \boldsymbol{PRN''_i}, d)|$
10: **end for**
11: **end for**
12: Select d^* that maximizes $\max_d \max_f A_i(f, d)$
13: Signal arrival time $r_i = d^*/(r \cdot 1,023 \text{ kHz})$
14: **end for**

Remarks:

- Multiple milliseconds of acquisition can be summed up to average out noise and therefore improve the arrival time detection probability.

Definition 14.32 (Acquisition)**.** ***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.*

Remarks:

- The relative speed between satellite and receiver introduces a significant Doppler shift to the carrier frequency. In order to decode the signal, a frequency search for the Doppler shift is necessary.
- The nested loops make acquisition the computationally most intensive part of a GPS receiver.

Algorithm 14.33 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 (Algorithm 14.31)
6: Track signals 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 for**
11: Solve the following system of equations for h and θ:
12: $||p_i - h||/c = d_i - \theta$, for all i

Remarks:

- GPS satellites carry precise atomic clocks, but the receiver is not synchronized with the satellites. The arrival times of the signals at the receiver are determined in the receiver's local time. Therefore, even though the satellite signals include transmit timestamps, the exact distance between satellites and receiver is unknown.
- 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.
- Since the equations are quadratic (distance), with as many observations as variables, the system of equations has two solutions in principle. For GPS however, in practice one of the solutions is far from the Earth surface, so the correct solution can always be identified without a fifth satellite.
- More received signals help reducing the measurement noise and thus improving the accuracy.
- Since the positioning solution, which is also called position fix, includes the handset's time offset Δ, this establishes a global time for all handsets. Thus, GPS is useful for global time synchronization.

- For a handset with unknown position, GPS timing is more accurate than time synchronization with a single transmitter, like a time signal station (cf. Definition 14.23). With the latter, the unknown signal propagation delays cannot be accounted for.

Definition 14.34 (A-GPS)**.** *An **Assisted GPS (A-GPS)** receiver fetches the satellite orbit parameters and other navigation data from the Internet, for instance via a cellular network.*

Remarks:

- A-GPS reduces the data transmission time, and thus the TTFF, from a maximum of 30 seconds per satellite to a maximum of 6 seconds.
- Smartphones regularly use A-GPS. However, coarse positioning is usually done based on nearby Wi-Fi base stations only, which saves energy compared to GPS.
- Another GPS improvement is *Differential GPS (DGPS)*: A receiver with a fixed location within a few kilometers of a mobile receiver compares the observed and actual satellite distances. This error is then subtracted at the mobile receiver. DGPS achieves accuracies in the order of 10 cm.

Definition 14.35 (Snapshot GPS Receiver)**.** *A **snapshot receiver** is a GPS receiver that captures one or a few milliseconds of raw GPS signal for a position fix.*

Remarks:

- Snapshot receivers aim at the remaining latency that results from the transmission of timestamps from the satellites every six seconds.
- Since time changes continuously, timestamps cannot be fetched together with the satellite orbit parameters that are valid for two hours.
- Snapshot receiver can determine the ranges to the satellites modulo 1 ms, which corresponds to 300 km. An approximate time and location of the receiver is used to resolve these ambiguities without a timestamp from the satellite signals themselves.

Definition 14.36 (CTN)**.** ***Coarse Time Navigation (CTN)*** *is a snapshot receiver positioning technique measuring sub-millisecond satellite ranges from correlation peaks, like conventional GPS receivers.*

Remarks:

- 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. The receiver location and time is not exactly known, but since signals are transmitted exactly whole milliseconds, rounding to the nearest whole millisecond gives the signal transmit time.

- With only a few milliseconds of signal, noise cannot be averaged out well and may lead to wrong signal arrival time estimates. Such wrong measurements usually render the system of equations unsolvable, making positioning infeasible.

Algorithm 14.37 Collective Detection Receiver

1: Given: A raw 1 ms GPS sample $\boldsymbol{s}$, a set H of location/time hypotheses
2: In addition, the receiver learned all navigation and atmospheric data

3: **for all** hypotheses $h \in H$ **do**
4: Vector $\boldsymbol{r} = 0$
5: Set V = satellites that should be visible with hypothesis h
6: **for all** satellites i in V **do**
7: $\boldsymbol{r} = \boldsymbol{r} + \boldsymbol{r_i}$, where $\boldsymbol{r_i}$ is expected signal of satellite i. The data of vector $\boldsymbol{r_i}$ 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.
8: **end for**
9: Probability $P_h = cxcorr(\boldsymbol{s}, \boldsymbol{r}, 0)$
10: **end for**
11: Solution: hypothesis $h \in H$ maximizing P_h

Definition 14.38 (Collective Detection)**.** ***Collective Detection (CD)*** *is a maximum likelihood snapshot receiver localization method, which does not determine an arrival time for each satellite, but rather combine all the available information and take a decision only at the end of the computation.*

Remarks:

- CD can tolerate a few low quality satellite signals and is thus more robust than CTN.
- In essence, CD tests how well position hypotheses match the received signal. For large position and time uncertainties, the high number of hypotheses require a lot of computation power.
- CD can be sped up by a branch and bound approach, which reduces the computation per position fix to the order of one second even for uncertainties of 100 km and a minute.

14.6 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 (logical) clock such that the clock values are well synchronized, and close to real time. Each node is equipped with a hardware (system) clock, that ticks more or less in real time, i.e., the time between two pulses is arbitrary between $[1-\epsilon, 1+\epsilon]$, for a constant $\epsilon \ll 1$. We assume that messages sent over the edges of the graph have a delivery time

between $[0, 1]$. In other words, we have a bounded but variable drift on the hardware clocks 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.

Definition 14.39 (Local and Global Clock Skew)**.** *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.*

Remarks:

- Of interest is also the *average global clock skew*, that is the average skew between any pair of nodes.

Theorem 14.40. *The global clock skew (Definition 14.12) is $\Omega(D)$, where D is the diameter of the network graph.*

Proof. For a node u, let t_u be the logical time of u and let $(u \to v)$ denote a message sent from u to a node v. Let $t(m)$ be the time delay of a message m and let u and v be neighboring nodes. First consider a case where the message delays between u and v are $1/2$. Then, all the messages sent by u and v at time t according to the clock of the sender arrive at time $t + 1/2$ according to the clock of the receiver.

Then consider the following cases

- $t_u = t_v + 1/2$, $t(u \to v) = 1$, $t(v \to u) = 0$
- $t_u = t_v - 1/2$, $t(u \to v) = 0$, $t(v \to u) = 1$,

where the message delivery time is always fast for one node and slow for the other and the logical clocks are off by $1/2$. In both scenarios, the messages sent at time i according to the clock of the sender arrive at time $i + 1/2$ according to the logical clock of the receiver. Therefore, for nodes u and v, both cases with clock drift seem the same as the case with perfectly synchronized clocks. Furthermore, in a linked list of D nodes, the left- and rightmost nodes l, r cannot distinguish $t_l = t_r + D/2$ from $t_l = t_r - D/2$. □

Remarks:

- From Theorem 14.40, it directly follows that any reasonable clock synchronization algorithm must have a global skew of $\Omega(D)$.
- Many natural algorithms manage to achieve a global clock skew of $\mathcal{O}(D)$.
- As both message jitter and hardware clock drift are bounded by constants, it feels like we should be able to get a constant drift at least between neighboring nodes.
- Let us look at the following algorithm:

Lemma 14.42. *The clock synchronization protocol of Algorithm 14.41 has a local skew of $\Omega(n)$.*

Algorithm 14.41 Local Clock Synchronization (at node v)

```
1: repeat
2:    send logical time t_v to all neighbors
3:    if Receive logical time t_u, where t_u > t_v, from any neighbor u then
4:       t_v = t_u
5:    end if
6: until done
```

Proof. Let the graph be a linked list of D nodes. We denote the nodes by $v_1, v_2, \ldots, v_D$ from left to right and the logical clock of node v_i by t_i. Apart from the left-most node v_1 all hardware clocks run with speed 1 (real time). Node v_1 runs at maximum speed, i.e. the time between two pulses is not 1 but $1 - \epsilon$. Assume that initially all message delays are 1. After some time, node v_1 will start to speed up v_2, and after some more time v_2 will speed up v_3, and so on. At some point of time, we will have a clock skew of 1 between any two neighbors. In particular $t_1 = t_D + D - 1$.

Now we start playing around with the message delays. Let $t_1 = T$. First we set the delay between the v_1 and v_2 to 0. Now node v_2 immediately adjusts its logical clock to T. After this event (which is instantaneous in our model) we set the delay between v_2 and v_3 to 0, which results in v_3 setting its logical clock to T as well. We perform this successively to all pairs of nodes until v_{D-2} and v_{D-1}. Now node v_{D-1} sets its logical clock to T, which indicates that the difference between the logical clocks of v_{D-1} and v_D is $T - (T - (D - 1)) = D - 1$. □

Remarks:

- The introduced examples may seem cooked-up, but examples like this exist in all networks, and for all algorithms. Indeed, it was shown that any natural clock synchronization algorithm must have a bad local skew. In particular, a protocol that averages between all neighbors (like Algorithm 14.13) is even worse than Algorithm 14.41. An averaging algorithm has a clock skew of $\Omega(D^2)$ in the linked list, at all times.

- It was shown that the local clock skew is $\Theta(\log D)$, i.e., there is a protocol that achieves this bound, and there is a proof that no algorithm can be better than this bound!

- Note that these are worst-case bounds. In practice, clock drift and message delays may not be the worst possible, typically the speed of hardware clocks changes at a comparatively slow pace and the message transmission times follow a benign probability distribution. If we assume this, better protocols do exist, in theory as well as in practice.

Chapter Notes

Atomic clocks can be used as a GPS fallback for data center synchronization [CDE+13].

GPS has been such a technological breakthrough that even though it dates back to the 1970s, the new GNSS still use essentially the same techniques. Several people worked on snapshot GPS receivers, but the technique has not penetrated into commercial receivers yet. Liu et al. [LPH+12] presented a practical CTN receiver and reduced the solution space by eliminating solutions not lying on the ground. CD receivers are studied since at least 2011 [ABD+11] and have recently been made practically feasible through branch and bound [BEW17]

It has been known for a long time that the global clock skew is $\Theta(D)$ [LL84, ST87]. The problem of synchronizing the clocks of nearby nodes was introduced by Fan and Lynch in [LF04]; they proved a surprising lower bound of $\Omega(\log D/\log\log D)$ for the local skew. The first algorithm providing a non-trivial local skew of $\mathcal{O}(\sqrt{D})$ was given in [LW06]. Later, matching upper and lower bounds of $\Theta(\log D)$ were given in [LLW10]. The problem has also been studied in a dynamic setting [KLO09, KLLO10] or when a fraction of nodes experience byzantine faults and the other nodes have to recover from faulty initial state (i.e., self-stabilizing) [DD06, DW04]. The self-stabilizing byzantine case has been solved with asymptotically optimal skew [KL18].

Clock synchronization is a well-studied problem in practice, for instance regarding the global clock skew in sensor networks, e.g. [EGE02, GKS03, MKSL04, PSJ04]. One more recent line of work is focussing on the problem of minimizing the local clock skew [BvRW07, SW09, LSW09, FW10, FZTS11].

This chapter was written in collaboration with Manuel Eichelberger.

Bibliography

[ABD+11] Penina Axelrad, Ben K Bradley, James Donna, Megan Mitchell, and Shan Mohiuddin. Collective Detection and Direct Positioning Using Multiple GNSS Satellites. *Navigation*, 58(4):305–321, 2011.

[BEW17] Pascal Bissig, Manuel Eichelberger, and Roger Wattenhofer. Fast and Robust GPS Fix Using One Millisecond of Data. In *Information Processing in Sensor Networks (IPSN), 2017 16th ACM/IEEE International Conference on*, pages 223–234. IEEE, 2017.

[BvRW07] Nicolas Burri, Pascal von Rickenbach, and Roger Wattenhofer. Dozer: Ultra-Low Power Data Gathering in Sensor Networks. In *International Conference on Information Processing in Sensor Networks (IPSN), Cambridge, Massachusetts, USA*, April 2007.

[CDE+13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[DD06] Ariel Daliot and Danny Dolev. Self-Stabilizing Byzantine Pulse Synchronization. *Computing Research Repository*, 2006.

[DW04] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of Byzantine faults. September 2004.

[EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained Network Time Synchronization Using Reference Broadcasts. *ACM SIGOPS Operating Systems Review*, 36:147–163, 2002.

[FW10] Roland Flury and Roger Wattenhofer. Slotted Programming for Sensor Networks. In *International Conference on Information Processing in Sensor Networks (IPSN), Stockholm, Sweden*, April 2010.

[FZTS11] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. Efficient Network Flooding and Time Synchronization with Glossy. In *Proceedings of the 10th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 73–84, 2011.

[GKS03] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. Timing-sync Protocol for Sensor Networks. In *Proceedings of the 1st international conference on Embedded Networked Sensor Systems (SenSys)*, 2003.

[KL18] Pankaj Khanchandani and Christoph Lenzen. Self-Stabilizing Byzantine Clock Synchronization with Optimal Precision. January 2018.

[KLLO10] Fabian Kuhn, Christoph Lenzen, Thomas Locher, and Rotem Oshman. Optimal Gradient Clock Synchronization in Dynamic Networks. In *29th Symposium on Principles of Distributed Computing (PODC), Zurich, Switzerland*, July 2010.

[KLO09] Fabian Kuhn, Thomas Locher, and Rotem Oshman. Gradient Clock Synchronization in Dynamic Networks. In *21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Calgary, Canada*, August 2009.

[LF04] Nancy Lynch and Rui Fan. Gradient Clock Synchronization. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2004.

[LL84] Jennifer Lundelius and Nancy Lynch. An Upper and Lower Bound for Clock Synchronization. *Information and Control*, 62:190–204, 1984.

[LLW10] Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. Tight Bounds for Clock Synchronization. In *Journal of the ACM, Volume 57, Number 2*, January 2010.

[LPH+12] Jie Liu, Bodhi Priyantha, Ted Hart, Heitor Ramos, Antonio A.F. Loureiro, and Qiang Wang. Energy Efficient GPS Sensing with Cloud Offloading. In *10th ACM Conference on Embedded Networked Sensor Systems (SenSys 2012)*. ACM, November 2012.

[LSW09] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. Optimal Clock Synchronization in Networks. In *7th ACM Conference on Embedded Networked Sensor Systems (SenSys), Berkeley, California, USA*, November 2009.

[LW06] Thomas Locher and Roger Wattenhofer. Oblivious Gradient Clock Synchronization. In *20th International Symposium on Distributed Computing (DISC), Stockholm, Sweden*, September 2006.

[MKSL04] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The Flooding Time Synchronization Protocol. In *Proceedings of the 2nd international Conference on Embedded Networked Sensor Systems*, SenSys '04, 2004.

[PSJ04] Santashil PalChaudhuri, Amit Kumar Saha, and David B. Johnson. Adaptive Clock Synchronization in Sensor Networks. In *Proceedings of the 3rd International Symposium on Information Processing in Sensor Networks*, IPSN '04, 2004.

[ST87] T. K. Srikanth and S. Toueg. Optimal Clock Synchronization. *Journal of the ACM*, 34:626–645, 1987.

[SW09] Philipp Sommer and Roger Wattenhofer. Gradient Clock Synchronization in Wireless Sensor Networks. In *8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), San Francisco, USA*, April 2009.

Chapter 15

Quorum Systems

What happens if a single server is no longer powerful enough to service all your customers? The obvious choice is to add more servers and to use the majority approach (e.g. Paxos, Chapter 7) to guarantee consistency. However, even if you buy one million servers, a client still has to access more than half of them per request! While you gain fault-tolerance, your efficiency can at most be doubled. Do we have to give up on consistency?

Let us take a step back: We used majorities because majority sets always overlap. But are majority sets the only sets that guarantee overlap? In this chapter we study the theory behind overlapping sets, known as quorum systems.

Definition 15.1 (quorum, quorum system). *Let $V = \{v_1, \ldots, v_n\}$ be a set of nodes. A **quorum** $Q \subseteq V$ is a subset of these nodes. A **quorum system** $\mathcal{S} \subset 2^V$ is a set of quorums s.t. every two quorums intersect, i.e., $Q_1 \cap Q_2 \neq \emptyset$ for all $Q_1, Q_2 \in \mathcal{S}$.*

Remarks:

- When a quorum system is being used, a client selects a quorum, acquires a lock (or ticket) on all nodes of the quorum, and when done releases all locks again. The idea is that no matter which quorum is chosen, its nodes will intersect with the nodes of every other quorum.
- What can happen if two quorums try to lock their nodes at the same time?
- A quorum system $\mathcal{S}$ is called **minimal** if $\forall Q_1, Q_2 \in \mathcal{S} : Q_1 \not\subset Q_2$.
- 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 $\lfloor \frac{n}{2} \rfloor + 1$ nodes.
- Can you think of other simple quorum systems?

15.1 Load and Work

Definition 15.2 (access strategy)**.** *An* ***access strategy*** *Z defines the probability $P_Z(Q)$ of accessing a quorum $Q \in \mathcal{S}$ s.t. $\sum_{Q\in\mathcal{S}} P_Z(Q) = 1$.*

Definition 15.3 (load)**.**

- *The* ***load*** *of access strategy Z on a node v_i is $L_Z(v_i) = \sum_{Q\in\mathcal{S};v_i\in Q} P_Z(Q)$.*
- *The* ***load*** *induced by access strategy Z on a quorum system $\mathcal{S}$ is the maximal load induced by Z on any node in $\mathcal{S}$, i.e., $L_Z(\mathcal{S}) = \max_{v_i\in\mathcal{S}} L_Z(v_i)$.*
- *The* ***load*** *of a quorum system $\mathcal{S}$ is $L(\mathcal{S}) = \min_Z L_Z(\mathcal{S})$.*

Definition 15.4 (work)**.**

- *The* ***work*** *of a quorum $Q \in \mathcal{S}$ is the number of nodes in Q, $W(Q) = |Q|$.*
- *The* ***work*** *induced by access strategy Z on a quorum system $\mathcal{S}$ is the expected number of nodes accessed, i.e., $W_Z(\mathcal{S}) = \sum_{Q\in\mathcal{S}} P_Z(Q) \cdot W(Q)$.*
- *The* ***work*** *of a quorum system $\mathcal{S}$ is $W(\mathcal{S}) = \min_Z W_Z(\mathcal{S})$.*

Remarks:

- Note that you cannot choose different access strategies Z for work and load, you have to pick a single Z for both.
- We illustrate the above concepts with a small example. Let $V = \{v_1, v_2, v_3, v_4, v_5\}$ and $\mathcal{S} = \{Q_1, Q_2, Q_3, Q_4\}$, with $Q_1 = \{v_1, v_2\}$, $Q_2 = \{v_1, v_3, v_4\}$, $Q_3 = \{v_2, v_3, v_5\}$, $Q_4 = \{v_2, v_4, v_5\}$. If we choose the access strategy Z s.t. $P_Z(Q_1) = 1/2$ and $P_Z(Q_2) = P_Z(Q_3) = P_Z(Q_4) = 1/6$, then the node with the highest load is v_2 with $L_Z(v_2) = 1/2+1/6+1/6 = 5/6$, i.e., $L_Z(\mathcal{S}) = 5/6$. Regarding work, we have $W_Z(\mathcal{S}) = 1/2 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 3 + 1/6 \cdot 3 = 15/6$.
- Can you come up with a better access strategy for $\mathcal{S}$?
- If every quorum Q in a quorum system $\mathcal{S}$ has the same number of elements, $\mathcal{S}$ is called *uniform.*
- What is the minimum load a quorum system can have?

Primary Copy vs. Majority		**Singleton**	**Majority**
How many nodes need to be accessed?	(Work)	**1**	$> n/2$
What is the load of the busiest node?	(Load)	1	$> \mathbf{1/2}$

Table 15.5: First comparison of the Singleton and Majority quorum systems. Note that the Singleton quorum system can be a good choice when the failure probability of every single node is $> 1/2$.

Theorem 15.6. *Let $\mathcal{S}$ be a quorum system. Then $L(\mathcal{S}) \geq 1/\sqrt{n}$ holds.*

Proof. Let $Q = \{v_1, \ldots, v_q\}$ be a quorum of minimal size in $\mathcal{S}$, with $|Q| = q$. Let Z be an access strategy for $\mathcal{S}$. Every other quorum in $\mathcal{S}$ intersects in at least one element with this quorum Q. Each time a quorum is accessed, at least one node in Q is accessed as well, yielding a lower bound of $L_Z(v_i) \geq 1/q$ for some $v_i \in Q$.

Furthermore, as Q is minimal, at least q nodes need to be accessed, yielding $W(\mathcal{S}) \geq q$. Thus, $L_Z(v_i) \geq q/n$ for some $v_i \in Q$, as each time q nodes are accessed, the load of the most accessed node is at least q/n.

Combining both ideas leads to $L_Z(\mathcal{S}) \geq \max(1/q, q/n) \Rightarrow L_Z(\mathcal{S}) \geq 1/\sqrt{n}$. Thus, $L(\mathcal{S}) \geq 1/\sqrt{n}$, as Z can be *any* access strategy. □

Remarks:

- Can we achieve this load?

15.2 Grid Quorum Systems

Definition 15.7 (Basic Grid quorum system)**.** *Assume $\sqrt{n} \in \mathbb{N}$, and arrange the n nodes in a square matrix with side length of $\sqrt{n}$, i.e., in a grid. The basic **Grid** quorum system consists of $\sqrt{n}$ quorums, with each containing the full row i and the full column i, for $1 \leq i \leq \sqrt{n}$.*

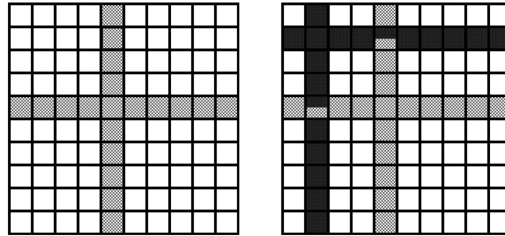


Figure 15.8: The basic version of the Grid quorum system, where each quorum Q_i with $1 \leq i \leq \sqrt{n}$ uses row i and column i. The size of each quorum is $2\sqrt{n} - 1$ and two quorums overlap in exactly two nodes. Thus, when the access strategy Z is uniform (i.e., the probability of each quorum is $1/\sqrt{n}$), the work is $2\sqrt{n} - 1$, and the load of every node is in $\Theta(1/\sqrt{n})$.

Remarks:

- Consider the right picture in Figure 15.8: The two quorums intersect in two nodes. If both quorums were to be accessed at the same time, it is not guaranteed that at least one quorum will lock all of its nodes, as they could enter a deadlock!
- In the case of just two quorums, one could solve this by letting the quorums just intersect in one node, see Figure 15.9. However, already with three quorums the same situation could occur again, progress is not guaranteed!
- However, by deviating from the "access all at once" strategy, we can guarantee progress if the nodes are totally ordered!

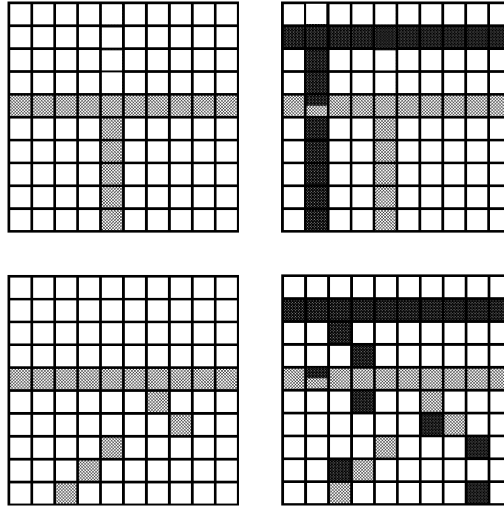


Figure 15.9: There are other ways to choose quorums in the grid s.t. pairwise different quorums only intersect in one node. The size of each quorum is between $\sqrt{n}$ and $2\sqrt{n}-1$, i.e., the work is in $\Theta(\sqrt{n})$. When the access strategy Z is uniform, the load of every node is in $\Theta(1/\sqrt{n})$.

Algorithm 15.10 Sequential Locking Strategy for a Quorum Q

1: Attempt to lock the nodes one by one, ordered by their identifiers
2: Should a node be already locked, release all locks and start over

Theorem 15.11. *If each quorum is accessed by Algorithm 15.10, at least one quorum will obtain a lock for all of its nodes.*

Proof. We prove the theorem by contradiction. Assume no quorum can make progress, i.e., for every quorum we have: At least one of its nodes is locked by another quorum. Let v be the node with the highest identifier that is locked by some quorum Q. Observe that Q already locked all of its nodes with a smaller identifier than v, otherwise Q would have restarted. As all nodes with a higher identifier than v are not locked, Q either has locked all of its nodes or can make progress – a contradiction. As the set of nodes is finite, one quorum will eventually be able to lock all of its nodes. □

Remarks:

- But now we are back to sequential accesses in a distributed system? Let's do it concurrently with the same idea, i.e., resolving conflicts by the ordering of the nodes. Then, a quorum that locked the highest identifier so far can always make progress!

Theorem 15.13. *If the nodes and quorums use Algorithm 15.12, at least one quorum will obtain a lock for all of its nodes.*

Algorithm 15.12 Concurrent Locking Strategy for a Quorum Q

Invariant: Let $v_Q \in Q$ be the highest identifier of a node locked by Q s.t. all nodes $v_i \in Q$ with $v_i < v_Q$ are locked by Q as well. Should Q not have any lock, then v_Q is set to 0.

```
1: repeat
2:   Attempt to lock all nodes of the quorum Q
3:   for each node v ∈ Q that was not able to be locked by Q do
4:     exchange v_Q and v_Q' with the quorum Q' that locked v
5:     if v_Q > v_Q' then
6:       Q' releases lock on v and Q acquires lock on v
7:     end if
8:   end for
9: until all nodes of the quorum Q are locked
```

Proof. The proof is analogous to the proof of Theorem 15.11: Assume for contradiction that no quorum can make progress. However, at least the quorum with the highest v_Q can always make progress – a contradiction! As the set of nodes is finite, at least one quorum will eventually be able to acquire a lock on all of its nodes. □

Remarks:

- What if a quorum locks all of its nodes and then crashes? Is the quorum system dead now? This issue can be prevented by, e.g., using leases instead of locks: leases have a timeout, i.e., a lock is released eventually.

15.3 Fault Tolerance

Definition 15.14 (resilience). *If any f nodes from a quorum system $\mathcal{S}$ can fail s.t. there is still a quorum $Q \in \mathcal{S}$ without failed nodes, then $\mathcal{S}$ is f-**resilient**. The largest such f is the **resilience** $R(\mathcal{S})$.*

Theorem 15.15. *Let $\mathcal{S}$ be a Grid quorum system where each of the n quorums consists of a full row and a full column. $\mathcal{S}$ has a resilience of $\sqrt{n} - 1$.*

Proof. If all $\sqrt{n}$ nodes on the diagonal of the grid fail, then every quorum will have at least one failed node. Should less than $\sqrt{n}$ nodes fail, then there is a row and a column without failed nodes. □

Remarks:

- The Grid quorum system in Theorem 15.15 is different from the Basic Grid quorum system described in Definition 15.7. In each quorum in the Basic Grid quorum system the row and column index are identical, while in the Grid quorum system of Theorem 15.15 this is not the case.

Definition 15.16 (failure probability). *Assume that every node works with a fixed probability p (in the following we assume concrete values, e.g. $p > 1/2$). The **failure probability** $F_p(\mathcal{S})$ of a quorum system $\mathcal{S}$ is the probability that at least one node of every quorum fails.*

Remarks:

- The **asymptotic failure probability** is $F_p(\mathcal{S})$ for $n \to \infty$.

Facts 15.17. *A version of a **Chernoff bound** states the following:*
Let $x_1, \ldots, x_n$ be independent Bernoulli-distributed random variables with $Pr[x_i = 1] = p_i$ and $Pr[x_i = 0] = 1 - p_i = q_i$, then for $X := \sum_{i=1}^{n} x_i$ and $\mu := \mathbb{E}[X] = \sum_{i=1}^{n} p_i$ the following holds:

$$\text{for all } 0 < \delta < 1\text{: } Pr[X \leq (1-\delta)\mu] \leq e^{-\mu\delta^2/2} .$$

Theorem 15.18. *The asymptotic failure probability of the Majority quorum system is 0.*

Proof. In a Majority quorum system each quorum contains exactly $\lfloor \frac{n}{2} \rfloor + 1$ nodes and each subset of nodes with cardinality $\lfloor \frac{n}{2} \rfloor + 1$ forms a quorum. The Majority quorum system fails, if only $\lfloor \frac{n}{2} \rfloor$ nodes work. Otherwise there is at least one quorum available. In order to calculate the failure probability we define the following random variables:

$$x_i = \begin{cases} 1, & \text{if node } i \text{ works, happens with probability } p \\ 0, & \text{if node } i \text{ fails, happens with probability } q = 1-p \end{cases}$$

and $X := \sum_{i=1}^{n} x_i$, with $\mu = np$,

whereas X corresponds to the number of working nodes. To estimate the probability that the number of working nodes is less than $\lfloor \frac{n}{2} \rfloor + 1$ we will make use of the Chernoff inequality from above. By setting $\delta = 1 - \frac{1}{2p}$ we obtain $F_P(\mathcal{S}) = Pr[X \leq \lfloor \frac{n}{2} \rfloor] \leq Pr[X \leq \frac{n}{2}] = Pr[X \leq (1-\delta)\mu]$.

With $\delta = 1 - \frac{1}{2p}$ we have $0 < \delta \leq 1/2$ due to $1/2 < p \leq 1$. Thus, we can use the Chernoff bound and get $F_P(\mathcal{S}) \leq e^{-\mu\delta^2/2} \in e^{-\Omega(n)}$. □

Theorem 15.19. *The asymptotic failure probability of the Grid quorum system is* 1*.*

Proof. Consider the $n = d \cdot d$ nodes to be arranged in a $d \times d$ grid. A quorum always contains one full row. In this estimation we will make use of the Bernoulli inequality which states that for all $n \in \mathbb{N}, x \geq -1 : (1+x)^n \geq 1 + nx$.

The system fails, if in each row at least one node fails (which happens with probability $1 - p^d$ for a particular row, as all nodes work with probability p^d). Therefore we can bound the failure probability from below with:

$F_p(\mathcal{S}) \geq Pr[\text{at least one failure per row}] = (1-p^d)^d \geq 1 - dp^d \underset{n\to\infty}{\longrightarrow} 1.$ □

Remarks:

- Now we have a quorum system with optimal load (the Grid) and one with fault-tolerance (Majority), but what if we want both?

Definition 15.20 (B-Grid quorum system)**.** *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.*

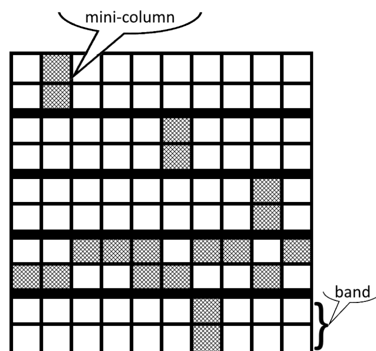


Figure 15.21: A B-Grid quorum system with $n = 100$ nodes, $d = 10$ columns, $h \cdot r = 10$ rows, $h = 5$ bands, and $r = 2$. The depicted quorum has a $d + hr - 1 = 10 + 5 \cdot 2 - 1 = 19$ nodes. If the access strategy Z is chosen uniformly, then we have a work of $d + hr - 1$ and a load of $\frac{d+hr-1}{n}$. By setting $d = \sqrt{n}$ and $r = \log n$, we obtain a work of $\Theta(\sqrt{n})$ and a load of $\Theta(1/\sqrt{n})$.

Theorem 15.22. *The asymptotic failure probability of the B-Grid quorum system is* 0.

Proof. Suppose $n = dhr$ and the elements are arranged in a grid with d columns and $h \cdot r$ rows. The B-Grid quorum system does fail if in each band a complete mini-column fails, because then it is not possible to choose a band where in each mini-column an element is still working. It also fails if in a band an element in each mini-column fails. Those events may not be independent of each other, but with the help of the union bound, we can upper bound the failure probability with the following equation:

$$\begin{aligned} F_p(\mathcal{S}) &\leq Pr[\text{in every band a complete mini-column fails}] \\ &\quad + Pr[\text{in a band at least one element of every m.-col. fails}] \\ &\leq (d(1-p)^r)^h + h(1-p^r)^d \end{aligned}$$

We use $d = \sqrt{n}, r = \ln d$, and $0 \leq (1-p) \leq 1/3$. Using $n^{\ln x} = x^{\ln n}$, we have $d(1-p)^r \leq d \cdot d^{\ln 1/3} \approx d^{-0.1}$, and hence for large enough d the whole first term is bounded from above by $d^{-0.1h} \ll 1/d^2 = 1/n$.

Regarding the second term, we have $p \geq 2/3$, and $h = d/\ln d < d$. Hence we can bound the term from above by $d(1 - d^{\ln 2/3})^d \approx d(1 - d^{-0.4})^d$. Using $(1 + t/n)^n \leq e^t$, we get (again, for large enough d) an upper bound of $d(1 - d^{-0.4})^d = d(1 - d^{0.6}/d)^d \leq d \cdot e^{-d^{0.6}} = d^{(-d^{0.6}/\ln d)+1} \ll d^{-2} = 1/n$. In total, we have $F_p(\mathcal{S}) \in O(1/n)$. □

15.4 Byzantine Quorum Systems

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 however, as they can pretend to be a regular node, i.e., one needs more sophisticated methods to deal with them. We need to ensure that the intersection of two quorums always contains a non-byzantine (correct) node and furthermore, the byzantine nodes should not be allowed to infiltrate every quorum. In

	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	$< \mathbf{n/2}$	$\Theta(\sqrt{n})$	$\Theta(\sqrt{n})$
F. Prob.**	$1-p$	$\to \mathbf{0}$	$\to 1$	$\to \mathbf{0}$

Table 15.23: Overview of the different quorum systems regarding resilience, work, load, and their asymptotic failure probability. The best entries in each row are set in bold.

* Setting $d = \sqrt{n}$ and $r = \log n$

**Assuming prob. $q = (1-p)$ is constant but significantly less than 1/2

this section we study three counter-measures of increasing strength, and their implications on the load of quorum systems.

Definition 15.24 (f-disseminating)**.** *A quorum system $\mathcal{S}$ is f-**disseminating** if* (1) *the intersection of two different quorums always contains $f+1$ nodes, and* (2) *for any set of f byzantine nodes, there is at least one quorum without byzantine nodes.*

Remarks:

- Thanks to (2), even with f byzantine nodes, the byzantine nodes cannot stop all quorums by just pretending to have crashed. At least one quorum will survive. We will also keep this assumption for the upcoming more advanced byzantine quorum systems.
- Byzantine nodes can also do something worse than crashing - they could falsify data! Nonetheless, due to (1), there is at least one non-byzantine node in every quorum intersection. If the data is self-verifying by, e.g., authentication, then this one node is enough.
- If the data is not self-verifying, then we need another mechanism.

Definition 15.25 (f-masking)**.** *A quorum system $\mathcal{S}$ 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.*

Remarks:

- Note that except for the second condition, an f-masking quorum system is the same as a $2f$-disseminating system. The idea is that the non-byzantine nodes (at least $f+1$ can outvote the byzantine ones (at most f), but only if all non-byzantine nodes are up-to-date!
- This raises an issue not covered yet in this chapter. If we access some quorum and update its values, this change still has to be disseminated to the other nodes in the byzantine quorum system. Opaque quorum systems deal with this issue, which are discussed at the end of this section.

- f-disseminating quorum systems need more than $3f$ nodes and f-masking quorum systems need more than $4f$ nodes. Essentially, the quorums may not contain too many nodes, and the different intersection properties lead to the different bounds.

Theorem 15.26. *Let $\mathcal{S}$ be a f-disseminating quorum system. Then $L(\mathcal{S}) \geq \sqrt{(f+1)/n}$ holds.*

Theorem 15.27. *Let $\mathcal{S}$ be a f-masking quorum system. Then $L(\mathcal{S}) \geq \sqrt{(2f+1)/n}$ holds.*

Proofs of Theorems 15.26 and 15.27. The proofs follow the proof of Theorem 15.6, by observing that now not just one element is accessed from a minimal quorum, but $f+1$ or $2f+1$, respectively. □

Definition 15.28 (f-masking Grid quorum system)**.** *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 $2f+1 \leq \sqrt{n}$.*

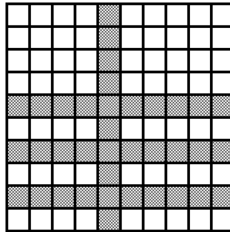


Figure 15.29: An example how to choose a quorum in the f-masking Grid with $f = 2$, i.e., $2+1 = 3$ rows. The load is in $\Theta(f/\sqrt{n})$ when the access strategy is chosen to be uniform. Two quorums overlap by their columns intersecting each other's rows, i.e., they overlap in at least $2f+2$ nodes.

Remarks:

- The f-masking Grid nearly hits the lower bound for the load of f-masking quorum systems, but not quite. A small change and we will be optimal asymptotically.

Definition 15.30 (M-Grid quorum system)**.** *The M-**Grid** quorum system is constructed as the grid quorum as well, but each quorum contains $\sqrt{f+1}$ rows and $\sqrt{f+1}$ columns of nodes, with $f \leq \frac{\sqrt{n}-1}{2}$.*

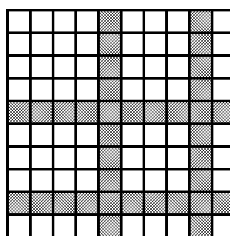


Figure 15.31: An example how to choose a quorum in the M-Grid with $f = 3$, i.e., 2 rows and 2 columns. The load is in $\Theta(\sqrt{f/n})$ when the access strategy is chosen to be uniform. Two quorums overlap with each row intersecting each other's column, i.e., $2\sqrt{f+1}^2 = 2f + 2$ nodes.

Corollary 15.32. *The f-masking Grid quorum system and the M-Grid quorum system are f-masking quorum systems.*

Remarks:

- 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? Surely we can fix that as well without too much loss?
- This property will be handled in the last part of this chapter by *opaque* quorum systems. It will ensure that the number of correct up-to-date nodes accessed will be larger than the number of out-of-date nodes combined with the byzantine nodes in the quorum (cf. (15.33.1)).

Definition 15.33 (f-opaque quorum system)**.** *A quorum system $\mathcal{S}$ 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:*

$$|(Q_1 \cap Q_2) \setminus F| > |(Q_2 \cap F) \cup (Q_2 \setminus Q_1)| \tag{15.33.1}$$

$$(F \cap Q) = \emptyset \text{ for some } Q \in \mathcal{S} \tag{15.33.2}$$

Theorem 15.35. *Let $\mathcal{S}$ be a f-opaque quorum system. Then, $n > 5f$.*

Proof. Due to (15.33.2), there exists a quorum Q_1 with size at most $n-f$. With (15.33.1), $|Q_1| > f$ holds. Let F_1 be a set of f (byzantine) nodes $F_1 \subset Q_1$, and with (15.33.2), there exists a $Q_2 \subset V \setminus F_1$. Thus, $|Q_1 \cap Q_2| \leq n - 2f$. With (15.33.1), $|Q_1 \cap Q_2| > f$ holds. Thus, one could choose f (byzantine) nodes F_2 with $F_2 \subset (Q_1 \cap Q_2)$. Using (15.33.1) one can bound $n - 3f$ from below: $n - 3f > |(Q_2 \cap Q_1)| - |F_2| \geq |(Q_2 \cap Q_1) \cup (Q_1 \cap F_2)| \geq |F_1| + |F_2| = 2f.$ □

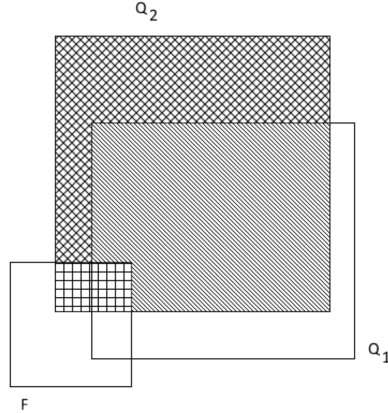


Figure 15.34: Intersection properties of an opaque quorum system. Equation (15.33.1) ensures that the set of non-byzantine nodes in the intersection of Q_1, Q_2 is larger than the set of out of date nodes, even if the byzantine nodes "team up" with those nodes. Thus, the correct up to date value can always be recognized by a majority voting.

Remarks:

- One can extend the Majority quorum system to be f-opaque by setting the size of each quorum to contain $\lceil (2n+2f)/3 \rceil$ nodes. Then its load is $1/n \lceil (2n+2f)/3 \rceil \approx 2/3 + 2f/3n \geq 2/3$.
- Can we do much better? Sadly, no...

Theorem 15.36. *Let $\mathcal{S}$ be a f-opaque quorum system. Then $L(\mathcal{S}) \geq 1/2$ holds.*

Proof. Equation (15.33.1) implies that for $Q_1, Q_2 \in \mathcal{S}$, the intersection of both Q_1, Q_2 is at least half their size, i.e., $|(Q_1 \cap Q_2)| \geq |Q_1|/2$. Let $\mathcal{S}$ consist of quorums $Q_1, Q_2, \ldots$. The load induced by an access strategy Z on Q_1 is:

$$\sum_{v \in Q_1} \sum_{v \in Q_i} L_Z(Q_i) = \sum_{Q_i} \sum_{v \in (Q_1 \cap Q_i)} L_Z(Q_i) \geq \sum_{Q_i} (|Q_1|/2)\ L_Z(Q_i) = |Q_1|/2\ .$$

Using the pigeonhole principle, there must be at least one node in Q_1 with load of at least 1/2. $\square$

Chapter Notes

Historically, a quorum is the minimum number of members of a deliberative body necessary to conduct the business of that group. Their use has inspired the introduction of quorum systems in computer science since the late 1970s/early 1980s. Early work focused on Majority quorum systems [Lam78, Gif79, Tho79], with the notion of minimality introduced shortly after [GB85]. The Grid quorum system was first considered in [Mae85], with the B-Grid being introduced in [NW94]. The latter article and [PW95] also initiated the study of load and resilience.

The f-masking Grid quorum system and opaque quorum systems are from [MR98], and the M-Grid quorum system was introduced in [MRW97]. Both papers also mark the start of the formal study of Byzantine quorum systems. The f-masking and the M-Grid have asymptotic failure probabilities of 1, more complex systems with better values can be found in these papers as well.

Quorum systems have also been extended to cope with nodes dynamically leaving and joining, see, e.g., the dynamic paths quorum system in [NW05].

For a further overview on quorum systems, we refer to the book by Vukolić [Vuk12] and the article by Merideth and Reiter [MR10].

This chapter was written in collaboration with Klaus-Tycho Förster.

Bibliography

[GB85] Hector Garcia-Molina and Daniel Barbará. How to assign votes in a distributed system. *J. ACM*, 32(4):841–860, 1985.

[Gif79] David K. Gifford. Weighted voting for replicated data. In Michael D. Schroeder and Anita K. Jones, editors, *Proceedings of the Seventh Symposium on Operating System Principles, SOSP 1979, Asilomar Conference Grounds, Pacific Grove, California, USA, 10-12, December 1979*, pages 150–162. ACM, 1979.

[Lam78] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[Mae85] Mamoru Maekawa. A square root N algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.

[MR98] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[MR10] Michael G. Merideth and Michael K. Reiter. Selected results from the latest decade of quorum systems research. In Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors, *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*, pages 185–206. Springer, 2010.

[MRW97] Dahlia Malkhi, Michael K. Reiter, and Avishai Wool. The load and availability of byzantine quorum systems. In James E. Burns and Hagit Attiya, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, Santa Barbara, California, USA, August 21-24, 1997*, pages 249–257. ACM, 1997.

[NW94] Moni Naor and Avishai Wool. The load, capacity and availability of quorum systems. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 214–225. IEEE Computer Society, 1994.

[NW05] Moni Naor and Udi Wieder. Scalable and dynamic quorum systems. *Distributed Computing*, 17(4):311–322, 2005.

[PW95] David Peleg and Avishai Wool. The availability of quorum systems. *Inf. Comput.*, 123(2):210–223, 1995.

[Tho79] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[Vuk12] Marko Vukolic. *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012.

Chapter 16

Eventual Consistency & Bitcoin

How would you implement an ATM? Does the following implementation work satisfactorily?

Algorithm 16.1 Naïve ATM

```
1: ATM makes withdrawal request to bank
2: ATM waits for response from bank
3: if balance of customer sufficient then
4:    ATM dispenses cash
5: else
6:    ATM displays error
7: end if
```

Remarks:

- A connection problem between the bank and the ATM may block Algorithm 16.1 in Line 2.
- A *network partition* is a failure where a network splits into at least two parts that cannot communicate with each other. Intuitively any non-trivial distributed system cannot proceed during a partition *and* maintain consistency. In the following we introduce the tradeoff between consistency, availability and partition tolerance.
- There are numerous causes for partitions to occur, e.g., physical disconnections, software errors, or incompatible protocol versions. From the point of view of a node in the system, a partition is similar to a period of sustained message loss.

16.1 Consistency, Availability and Partitions

Definition 16.2 (Consistency)**.** *All nodes in the system agree on the current state of the system.*

Definition 16.3 (Availability)**.** *The system is operational and instantly processing incoming requests.*

Definition 16.4 (Partition Tolerance)**.** *Partition tolerance is the ability of a distributed system to continue operating correctly even in the presence of a network partition.*

Theorem 16.5 (CAP Theorem)**.** *It is impossible for a distributed system to simultaneously provide Consistency, Availability and Partition Tolerance. A distributed system can satisfy any two of these but not all three.*

Proof. Assume two nodes, sharing some state. The nodes are in different partitions, i.e., they cannot communicate. Assume a request wants to update the state and contacts a node. The node may either: 1) update its local state, resulting in inconsistent states, or 2) not update its local state, i.e., the system is no longer available for updates. □

Algorithm 16.6 Partition tolerant and 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 user interaction
5:   end if
6: end if
7: ATM dispenses cash
8: ATM logs withdrawal for synchronization
```

Remarks:

- Algorithm 16.6 is partition tolerant and available since it continues to process requests even when the bank is not reachable.
- The ATM's local view of the balances may diverge from the balances as seen by the bank, therefore consistency is no longer guaranteed.
- The algorithm will synchronize any changes it made to the local balances back to the bank once connectivity is re-established. This is known as eventual consistency.

Definition 16.7 (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.*

Remarks:

- Eventual consistency is a form of *weak consistency.*
- Eventual consistency guarantees that the state is eventually agreed upon, but the nodes may disagree temporarily.
- During a partition, different updates may semantically conflict with each other. A *conflict resolution* mechanism is required to resolve the conflicts and allow the nodes to eventually agree on a common state.

- One example of eventual consistency is the Bitcoin cryptocurrency system.

16.2 Bitcoin

Definition 16.8 (Bitcoin Network)**.** *The Bitcoin network is a randomly connected overlay network of a few thousand* ***nodes****, controlled by a variety of owners. All nodes perform the same operations, i.e., it is a homogenous network and without central control.*

Remarks:

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

Definition 16.9 (Address)**.** *Users may generate any number of private keys, from which a public key is then derived. An address is derived from a public key and may be used to identify the recipient of funds in Bitcoin. The private/public key pair is used to uniquely identify the owner of funds of an address.*

Remarks:

- The terms public key and address are often used interchangeably, since both are public information. The advantage of using an address is that its representation is shorter than the public key.
- It is hard to link addresses to the user that controls them, hence Bitcoin is often referred to as being *pseudonymous*.
- Not every user needs to run a fully validating node, and end-users will likely use a lightweight client that only temporarily connects to the network.
- The Bitcoin network collaboratively tracks the balance in bitcoins of each address.
- The address is composed of a network identifier byte, the hash of the public key and a checksum. It is commonly stored in base 58 encoding, a custom encoding similar to base 64 with some ambiguous symbols removed, e.g., lowercase letter "l" since it is similar to the number "1".
- The hashing algorithm produces addresses of size 20 bytes. This means that there are 2^{160} distinct addresses. It might be tempting to brute force a target address, however at one billion trials per second one still requires approximately 2^{45} years in expectation to find a matching private/public key pair. Due to the birthday paradox the odds improve if instead of brute forcing a single address we attempt to brute force any address. While the odds of a successful trial increase with the number of addresses, lookups become more costly.

Definition 16.10 (Output)**.** *An output is a tuple consisting of an amount of bitcoins and a spending condition. Most commonly the spending condition requires a valid signature associated with the private key of an address.*

Remarks:

- Spending conditions are scripts that offer a variety of options. Apart from a single signature, they may include conditions that require the result of a simple computation, or the solution to a cryptographic puzzle.
- Outputs exist in two states: unspent and spent. Any output can be spent at most once. The address balance is the sum of bitcoin amounts in unspent outputs that are associated with the address.
- The set of unspent transaction outputs (UTXOs) and some additional global parameters are the shared state of Bitcoin. Every node in the Bitcoin network holds a complete replica of that state. Local replicas may temporarily diverge, but consistency is eventually re-established.

Definition 16.11 (Input)**.** *An input is a tuple consisting of a reference to a previously created output and arguments (signature) to the spending condition, proving that the transaction creator has the permission to spend the referenced output.*

Definition 16.12 (Transaction)**.** *A transaction is a data structure that describes the transfer of bitcoins from spenders to recipients. The transaction consists of a number of inputs and new outputs. The inputs result in the referenced outputs spent (removed from the UTXO), and the new outputs being added to the UTXO.*

Remarks:

- Inputs reference the output that is being spent by a (h, i)-tuple, where h is the hash of the transaction that created the output, and i specifies the index of the output in that transaction.
- Transactions are broadcast in the Bitcoin network and processed by every node that receives them.

Algorithm 16.13 Node Receives Transaction

1: Receive transaction t
2: **for each** input (h, i) in t **do**
3: **if** output (h, i) is not in local UTXO **or** signature invalid **then**
4: Drop t and stop
5: **end if**
6: **end for**
7: **if** sum of values of inputs $<$ sum of values of new outputs **then**
8: Drop t and stop
9: **end if**
10: **for each** input (h, i) in t **do**
11: Remove (h, i) from local UTXO
12: **end for**
13: Append t to local history
14: Forward t to neighbors in the Bitcoin network

Remarks:

- Note that the effect of a transaction on the state is deterministic. In other words if all nodes receive the same set of transactions in the same order (Definition 7.8), then the state across 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*. The fee is used to incentivize other participants in the system (see Definition 16.19)
- Notice that so far we only described a local acceptance policy. Nothing prevents nodes to locally accept different transactions that spend the same output.
- Transactions are in one of two states: unconfirmed or confirmed. Incoming transactions from the broadcast are unconfirmed and added to a pool of transactions called the *memory pool*.

Definition 16.14 (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 becomes inconsistent.*

Remarks:

- Doublespends may occur naturally, e.g., if outputs are co-owned by multiple users. However, often doublespends are intentional – we call these doublespend-attacks: In a transaction, an attacker pretends to transfer an output to a victim, only to doublespend the same output in another transaction back to itself.
- Doublespends can result in an inconsistent state since the validity of transactions depends on the order in which they arrive. If two conflicting transactions are seen by a node, the node considers the first to be valid, see Algorithm 16.13. The second transaction is invalid

since it tries to spend an output that is already spent. The order in which transactions are seen, may not be the same for all nodes, hence the inconsistent state.

- If doublespends are not resolved, the shared state diverges. Therefore a conflict resolution mechanism is needed to decide which of the conflicting transactions is to be confirmed (accepted by everybody), to achieve eventual consistency.

Definition 16.15 (Proof-of-Work)**.** *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* $\mathcal{F}_d(c, x) \to \{true, false\}$*, where difficulty* d *is a positive number, while challenge* c *and nonce* x *are usually bit-strings, is called a Proof-of-Work function if it has following properties:*

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

Definition 16.16 (Bitcoin PoW function)**.** *The Bitcoin PoW function is given by*

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

Remarks:

- This function concatenates the challenge c and nonce x, and hashes them twice using SHA256. The output of SHA256 is a cryptographic hash with a numeric value in $\{0, \dots, 2^{256} - 1\}$ which is compared to a target value $\frac{2^{224}}{d}$, which gets smaller with increasing difficulty.
- SHA256 is a cryptographic hash function with pseudorandom output. No better algorithm is known to find a nonce x such that the function $\mathcal{F}_d(c, x)$ returns true than simply iterating over possible inputs. This is by design to make it difficult to find such an input, but simple to verify the validity once it has been found.
- If the PoW functions of all nodes had the same challenge, the fastest node would always win. However, as we will see in Definition 16.19, each node attempts to find a valid nonce for a node-specific challenge.

Definition 16.17 (Block)**.** *A block is a data structure used to communicate incremental changes to the local state of a node. A block consists of a list of transactions, a reference to a previous block and a nonce. A block 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 16.18 Node Finds Block

1: Nonce $x = 0$, challenge c, difficulty d, previous block b_{t-1}
2: **repeat**
3: $\quad x = x + 1$
4: **until** $\mathcal{F}_d(c, x) = true$
5: Broadcast block $b_t = (memory\ pool, b_{t-1}, x)$

Remarks:

- 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. Bitcoin sets the difficulty so that globally a block is created about every 10 minutes in expectation.
- 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.

Definition 16.19 (Reward Transaction)**.** *The first transaction in a block is called the reward transaction. The block's miner is rewarded for confirming transactions by allowing it to mint new coins. The reward 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.*

Remarks:

- A reward transaction is the sole exception to the rule that the sum of inputs must be at least the sum of outputs.
- The number of bitcoins that are minted by the reward transaction and assigned to the miner is determined by a subsidy schedule that is part of the protocol. Initially the subsidy was 50 bitcoins for every block, and it is being halved every 210,000 blocks, or 4 years in expectation. Due to the halving of the block reward, the total amount of bitcoins in circulation never exceeds 21 million bitcoins.
- It is expected that the cost of performing the PoW to find a block, in terms of energy and infrastructure, is close to the value of the reward the miner receives from the reward transaction in the block.

Definition 16.20 (Blockchain)**.** *The longest path from the genesis block, i.e., root of the tree, to a leaf is called the blockchain. The blockchain acts as a consistent transaction history on which all nodes eventually agree.*

Remarks:

- The path length from the genesis block to block b is the height h_b.
- 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.
- The mining incentives quickly increased the difficulty of the PoW mechanism: initially miners used CPUs to mine blocks, but CPUs were quickly replaced by GPUs, FPGAs and even application specific integrated circuits (AS-ICs) as bitcoins appreciated. This results in an equilibrium today in which only the most cost efficient miners, in terms of hardware supply and electricity, make a profit in expectation.
- If multiple blocks are mined more or less concurrently, the system is said to have *forked*. Forks happen naturally because mining is a distributed random process and two new blocks may be found at roughly the same time.

Algorithm 16.21 Node Receives Block

1: Receive block b
2: For this node the current head is block b_{max} at height h_{max}
3: Connect block b in the tree as child of its parent p at height $h_b = h_p + 1$
4: **if** $h_b > h_{max}$ **then**
5: $h_{max} = h_b$
6: $b_{max} = b$
7: Compute UTXO for the path leading to b_{max}
8: Cleanup memory pool
9: **end if**

Remarks:

- Algorithm 16.21 describes how a node updates its local state upon receiving a block. Notice that, like Algorithm 16.13, this describes the local policy and may also result in node states diverging, i.e., by accepting different blocks at the same height as current head.
- Unlike extending the current path, switching paths may result in confirmed transactions no longer being confirmed, because the blocks in the new path do not include them. Switching paths is referred to as a *reorg*.
- Cleaning up the memory pool involves 1) removing transactions that were confirmed in a block in the current path, 2) removing transactions that conflict with confirmed transactions, and 3) adding transactions that were confirmed in the previous path, but are no longer confirmed in the current path.

- In order to avoid having to recompute the entire UTXO at every new block being added to the blockchain, all current implementations use data structures that store undo information about the operations applied by a block. This allows efficient switching of paths and updates of the head by moving along the path.

Theorem 16.22. *Forks are eventually resolved and all nodes eventually agree on which is the longest blockchain. The system therefore guarantees eventual consistency.*

Proof. In order for the fork to continue to exist, pairs of blocks need to be found in close succession, extending distinct branches, otherwise the nodes on the shorter branch would switch to the longer one. The probability of branches being extended almost simultaneously decreases exponentially with the length of the fork, hence there will eventually be a time when only one branch is being extended, becoming the longest branch. □

16.3 Smart Contracts

Definition 16.23 (Smart Contract)**.** *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.*

Remarks:

- Contracts allow business logic to be encoded in Bitcoin transactions which mutually guarantee that an agreed upon action is performed. The blockchain acts as conflict mediator, should a party fail to honor an agreement.
- The use of scripts as spending conditions for outputs enables smart contracts. Scripts, together with some additional features such as timelocks, allow encoding complex conditions, specifying who may spend the funds associated with an output and when.

Definition 16.24 (Timelock)**.** *Bitcoin provides a mechanism to make transactions invalid until some time in the future:* ***timelocks****. A transaction may specify a locktime: the earliest time, expressed in either a Unix timestamp or a blockchain height, at which it may be included in a block and therefore be confirmed.*

Remarks:

- Transactions with a timelock are not released into the network until the timelock expires. It is the responsibility of the node receiving the transaction to store it locally until the timelock expires and then release it into the network.
- Transactions with future timelocks are invalid. Blocks may not include transactions with timelocks that have not yet expired, i.e., they are mined before their expiry timestamp or in a lower block than specified. If a block includes an unexpired transaction it is invalid. Upon receiving invalid transactions or blocks, nodes discard them immediately and do not forward them to their peers.

- Timelocks can be used to replace or supersede transactions: a timelocked transaction t_1 can be replaced by another transaction t_0, spending some of the same outputs, if the replacing transaction t_0 has an earlier timelock and can be broadcast in the network before the replaced transaction t_1 becomes valid.

Definition 16.25 (Singlesig and Multisig Outputs)**.** *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 (with $k \leq m$) valid signatures from distinct matching public keys from that set in order to be valid.*

Remarks:

- Most smart contracts begin with the creation of a 2-of-2 multisig output, requiring a signature from both parties. Once the transaction creating the multisig output is confirmed in the blockchain, both parties are guaranteed that the funds of that output cannot be spent unilaterally.

Algorithm 16.26 Parties A and B create a 2-of-2 multisig output o

1: B sends a list I_B of inputs with c_B coins to A
2: A selects its own inputs I_A with c_A coins
3: A creates transaction $t_s\{[I_A, I_B], [o = c_A + c_B \rightarrow (A, B)]\}$
4: A creates timelocked transaction $t_r\{[o], [c_A \rightarrow A, c_B \rightarrow B]\}$ and signs it
5: A sends t_s and t_r to B
6: B signs both t_s and t_r and sends them to A
7: A signs t_s and broadcasts it to the Bitcoin network

Remarks:

- t_s is called a *setup transaction* and is used to lock in funds into a shared account. If t_s is signed and broadcast immediately, one of the parties could not collaborate to spend the multisig output, and the funds become unspendable. To avoid a situation where the funds cannot be spent, the protocol also creates a timelocked *refund transaction* t_r which guarantees that, should the funds not be spent before the timelock expires, the funds are returned to the respective party. At no point in time one of the parties holds a fully signed setup transaction without the other party holding a fully signed refund transaction, guaranteeing that funds are eventually returned.

- Both transactions require the signature of both parties. In the case of the setup transaction because it has two inputs from A and B respectively which require individual signatures. In the case of the refund transaction the single input spending the multisig output requires both signatures being a 2-of-2 multisig output.

Algorithm 16.27 Simple Micropayment Channel from S to R with capacity c

1: $c_S = c$, $c_R = 0$
2: S and R use Algorithm 16.26 to set up output o with value c from S
3: Create settlement transaction $t_f\{[o], [c_S \rightarrow S, c_R \rightarrow R]\}$
4: **while** channel open **and** $c_R < c$ **do**
5: In exchange for good with value δ
6: $c_R = c_R + \delta$
7: $c_S = c_S - \delta$
8: Update t_f with outputs $[c_R \rightarrow R, c_S \rightarrow S]$
9: S signs and sends t_f to R
10: **end while**
11: R signs last t_f and broadcasts it

Remarks:

- Algorithm 16.27 implements a Simple Micropayment Channel, a smart contract that is used for rapidly adjusting micropayments from a spender to a recipient. Only two transactions are ever broadcast and inserted into the blockchain: the setup transaction t_s and the last settlement transaction t_f. There may have been any number of updates to the settlement transaction, transferring ever more of the shared output to the recipient.

- The number of bitcoins c used to fund the channel is also the maximum total that may be transferred over the simple micropayment channel.

- At any time the recipient R is guaranteed to eventually receive the bitcoins, since she holds a fully signed settlement transaction, while the spender only has partially signed ones.

- 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. If we were to transfer bitcoins back, we would be reducing the amount paid out to the recipient, hence she would choose not to broadcast that transaction.

16.4 Weak Consistency

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

Definition 16.28 (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.*

Remarks:

- Users are annoyed if they receive a notification about a comment on an online social network, but are unable to reply because the web interface does not show the same notification yet. In this case the notification acts as the first read operation, while looking up the comment on the web interface is the second read operation.

Definition 16.29 (Monotonic Write Consistency)**.** *A write operation by a node on a data item is completed before any successive write operation by the same node (i.e., system guarantees to serialize writes by the same node).*

Remarks:

- The ATM must replay all operations in order, otherwise it might happen that an earlier operation overwrites the result of a later operation, resulting in an inconsistent final state.

Definition 16.30 (Read-Your-Write Consistency)**.** *After a node u has updated a data item, any later reads from node u will never see an older value.*

Definition 16.31 (Causal Relation)**.** *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.*

Remarks:

- The first rule ensures that writes by a single node are seen in the same order. For example if a node writes a value in one variable and then signals that it has written the value by writing in another variable. Another node could then read the signalling variable but still read the old value from the first variable, if the two writes were not causally related.

Definition 16.32 (Causal Consistency)**.** *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.*

Chapter Notes

The CAP theorem was first introduced by Fox and Brewer [FB99], although it is commonly attributed to a talk by Eric Brewer [Bre00]. It was later proven by Gilbert and Lynch [GL02] for the asynchronous model. Gilbert and Lynch also showed how to relax the consistency requirement in a partially synchronous system to achieve availability and partition tolerance.

Bitcoin was introduced in 2008 by Satoshi Nakamoto [Nak08]. Nakamoto is thought to be a pseudonym used by either a single person or a group of people; it is still unknown who invented Bitcoin, giving rise to speculation and conspiracy theories. Among the plausible theories are noted cryptographers Nick Szabo [Big13] and Hal Finney [Gre14]. The first Bitcoin client was published shortly after the paper and the first block was mined on January 3, 2009. The genesis block contained the headline of the release date's The Times issue "*The Times 03/Jan/2009 Chancellor on brink of second bailout for banks*", which serves as proof that the genesis block has been indeed mined on that date, and that no one had mined before that date. The quote in the genesis block is also thought to be an ideological hint: Bitcoin was created in a climate of financial crisis, induced by rampant manipulation by the banking sector, and Bitcoin quickly grew in popularity in anarchic and libertarian circles. The original client is nowadays maintained by a group of independent core developers and remains the most used client in the Bitcoin network.

Central to Bitcoin is the resolution of conflicts due to doublespends, which is solved by waiting for transactions to be included in the blockchain. This however introduces large delays for the confirmation of payments which are undesirable in some scenarios in which an immediate confirmation is required. Karame et al. [KAC12] show that accepting unconfirmed transactions leads to a non-negligible probability of being defrauded as a result of a doublespending attack. This is facilitated by *information eclipsing* [DW13], i.e., that nodes do not forward conflicting transactions, hence the victim does not see both transactions of the doublespend. Bamert et al. [BDE+13] showed that the odds of detecting a doublespending attack in real-time can be improved by connecting to a large sample of nodes and tracing the propagation of transactions in the network.

Bitcoin does not scale very well due to its reliance on confirmations in the blockchain. A copy of the entire transaction history is stored on every node in order to bootstrap joining nodes, which have to reconstruct the transaction history from the genesis block. Simple micropayment channels were introduced by Hearn and Spilman [HS12] and may be used to bundle multiple transfers between two parties but they are limited to transferring the funds locked into the channel once. Recently Duplex Micropayment Channels [DW15] and the Lightning Network [PD15] have been proposed to build bidirectional micropayment channels in which the funds can be transferred back and forth an arbitrary number of times, greatly increasing the flexibility of Bitcoin transfers and enabling a number of features, such as micropayments and routing payments between any two endpoints.

This chapter was written in collaboration with Christian Decker.

Bibliography

[BDE+13] Tobias Bamert, Christian Decker, Lennart Elsen, Samuel Welten, and Roger Wattenhofer. Have a snack, pay with bitcoin. In *IEEE Internation Conference on Peer-to-Peer Computing (P2P), Trento, Italy*, 2013.

[Big13] John Biggs. Who is the real satoshi nakamoto? one researcher may have found the answer. http://on.tcrn.ch/l/R0vA, 2013.

[Bre00] Eric A. Brewer. Towards robust distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*. ACM, 2000.

[DW13] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *IEEE International Conference on Peer-to-Peer Computing (P2P), Trento, Italy*, September 2013.

[DW15] Christian Decker and Roger Wattenhofer. A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In *Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2015.

[FB99] Armando Fox and Eric Brewer. Harvest, yield, and scalable tolerant systems. In *Hot Topics in Operating Systems*. IEEE, 1999.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 2002.

[Gre14] Andy Greenberg. Nakamoto's neighbor: My hunt for bitcoin's creator led to a paralyzed crypto genius. http://onforb.es/1rvyecq, 2014.

[HS12] Mike Hearn and Jeremy Spilman. Contract: Rapidly adjusting micro-payments. https://en.bitcoin.it/wiki/Contract, 2012. Last accessed on November 11, 2015.

[KAC12] G.O. Karame, E. Androulaki, and S. Capkun. Two Bitcoins at the Price of One? Double-Spending Attacks on Fast Payments in Bitcoin. In *Conference on Computer and Communication Security (CCS)*, 2012.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[PD15] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network. 2015.

Chapter 17

Memory Management and Virtual Memory

You've already seen MMUs, TLBs, and basic paged virtual memory operations in the Systems Programming and Computer Architecture course at ETHZ. Here we take this a bit further.

17.1 Segments

Before paging, there were segments. Segments evolved from basic protection mechanisms. Today they are little used (but still present on x86 machines), although there is some evidence they are making a comeback as memories become very large.

Definition 17.1 (Base and Limit)**.** *A **base and limit register pair** is a couple of hardware registers containing two addresses B and L. A CPU access to an address a is permitted iff $B \leq a < L$.*

Remarks:

- Base and limit registers were the first, primitive form of MMU, and at first operated entirely on physical addresses (as did user code). Each process in the system had its own base-and-limit region. To context switch, the OS reprogrammed the base and limit registers.
- Since the base address is not known to the program at compile time, programs had to be compiled to *position-independent code*, or a *relocation register* used.
- Base and limit registers still live on in the x86 architecture as a way of constraining *virtual* addresses.

Definition 17.2 (Relocation register)**.** *A **relocation register** is 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 $B + a$ and allowed iff $B \leq B + a < L$.*

Remarks:

- With relocation registers each program can be compiled to run at the same address (e.g. `0x0000`).
- Relocation registers don't allow sharing of code and data between processes, since each process has a single region of memory (and a different relocation address).
- Relocation registers allow a primitive form of *swapping*: a process' single memory region can be written out to storage, and later read back in to a different physical address and resumed.

The sharing problem with relocation registers was solved by generalizing base/limit regions, and making them independent of particular processes. The result is segments.

Definition 17.3 (Segments)**.** *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 load or store to or from a logical address (i, o) succeeds iff $0 \leq o < L_i$ and the running process is authorized to access segment i. If it does succeed, it will access physical address $B_i + o$.*

Remarks:

- In practice the segment identifier for an address might be specified explicitly in machine instructions, indirectly via a special processor "segment register", or explicitly in the address value used.

Definition 17.4 (Segment Table)**.** *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. The MMU in a segmentation system holds the location and size of this table in a **segment table base register** (STBR) and **segment table length register** (STLR). Logical memory accesses cause the MMU to look up the segment id in this table to obtain the physical address and protection information.*

Remarks:

- Segmentation is fast. As with pages, segment information can be cached in a TLB (and locality is *much* higher than with pages since segments are large and contiguous).
- Sharing is trivially easy at a segment granularity. If segments are allowed to overlap, finer-grained sharing is possible.
- The OS and hardware might have a single, system-wide segment table, or (more often) a per-process table. Context switching is fast: it just involves rewriting the STBR and STLR.
- Segment idenfitiers can be *virtualized* by having per-process segment tables indirect into a single, system-wide table. Protection is therefore applied at each process' level, while translation of addresses is the same everywhere.

- The principal downside of segmentation is that segments are still contiguous in physical memory, which leads to *external fragmentation* (recall this from Systems Programmming).

17.2 Paging

As we should already know from Systems Programing, paging solves the external fragmentation problem associated with segments, at some cost in efficiency.

Definition 17.5 (Paging)**.** *A paging system divides the physical address space into fixed-size regions called* ***frames*** *or* ***physical pages****, indexed by a* ***physical frame*** *(or* ***page****)* ***number (PFN)****, the high bits of the address. The virtual address space is similarly divided into fixed-size regions called* ***virtual pages****, identified by* ***virtual page number (VPN)*** *the high bits of the virtual address. The MMU translates from virtual addresses to physical addresses by looking up the VPN in a* ***page table*** *to obtain a PFN.* ***Page Table Entries (PTEs)*** *also hold protection metadata for the virtual page, including validity information. Access via an invalid PTE causes a* ***Page Fault*** *processor exception. VPN-to-PFN translations are cached in a* ***Translation Lookaside Buffer (TLB)****.*

Remarks:

- Virtual pages and physical pages are *almost* always the same size, and are always a power-of-two in size (often 4kB).
- Segmentation and paging can be combined. Either segments become regions of virtual memory, which is then paged, or each segment itself is divided into a set of pages and paged itself.

Definition 17.6 (Linear page table)**.** *A* ***Linear Page Table*** *is a page table implemented as a simple array of PTEs, indexed by VPN.*

Remarks:

- Linear page tables work great when there aren't many pages – early systems often had only 8 pages per virtual address space, and sometimes simply used registers to implement the page table.
- Linear page tables grow with the size of the virtual address space, and become very large on 32-bit and 64-bit machines. They're generally not used in modern machines.
- The DEC VAX, a 32-bit machine, *did* use a linear page table scheme, but ingeniously dealt with the size of the table by placing it in virtual memory [Leo87].

Definition 17.7 (Hierarchical page table)**.** *A* ***Hierarchical Page Table*** *organized in multiple layers, each one translating a different set of bits in the virtual address.*

Example 17.8 (64-bit x86 paging [Int18])**.** *. x86 has a page size of 4096 bytes, which can be addressed using 12 bits. It happens that every page table used in x86 is also 4kB in size, and virtual addresses are 48 bits wide.*

The page table has four levels, each one translating 9 bits of the virtual address leaving 12 left over for the page offset. Since each entry is 64 bits or 8 (i.e. 2^3 *bytes) in size, to translate* 2^9 *or 512 addresses requires* 2^{9+3} *or 4096 bits of page table – a page.*

The topmost level is called the ***Page Map Level 4*** *or PML4, and translates bits 58-47 of the virtual address to (among other things) the virtual address of the next level page table, the* ***Page Directory Page Table*** *or PDPT. This in turn points to a* ***Page Directory*** *or PD, and thence to a* ***Page Table*** *proper or PT. This then gives the* ***physical*** *address of the frame.*

Remarks:

- Most virtual address spaces for 64-bit machines are less than the full 64 bits in size. The address is typically sign-extended to 64 bits.
- Many architectures support two page table base registers, one for the upper portion of the virtual address space and one for the lower. This allows the kernel to run in virtual memory (using one page table) without needing to swap page tables for the kernel on a context switch.
- Not all architectures have the "everything is the size of a page" property of x86. For example, ARM page tables are very different sizes at different levels, though always a power of two bytes.

Definition 17.9 (Page table walking)**.** *The process of translating a virtual address to a physical address using a hierarchical page table is called a* ***Page Table Walk****. Most processor MMUs have a* ***hardware table walker*** *which is used on TLB misses to find and load the appropriate page table entry into the TLB, but others (like the MIPS [KH91]) require the OS to provide a* ***software page table walker****.*

In Systems Programming, we only looked at how a hardware walker and a TLB use a hierarchical page table to map virtual addresses to physical addresses. In practice, *software* in the operating system always has to map virtual to physical addresses:

- A software-loaded TLB simply has no hardware to walk the page table.
- In an case, when a page fault occurs, the OS needs to map the faulting (virtual) address to the relevant PTE, so that it can figure out what kind of fault occurred and also where to find a physical page to satisfy the fault. This requires a table walk.
- When a user process *unmaps* a region of virtual memory (including when the process exits), the OS has to identify all the physical pages associated with that mapping.

And so on. The OS can do this in two ways: first, it can walk the actual hardware page table in software. Alternatively, it can keep a "shadow page table" structure which is independent of the hardware page table. This might be easier to walk, and might occupy less space (the hardware table might even be constructed lazily from this structure in response to early page faults).

However, the OS also needs to map *physical addresses* to *virtual addresses* as well. For example, when a physical page needs to be reused, once it is clean, the OS needs to identify every virtual mapping to the page and mark them invalid. This requires a data structure for *reverse mapping* as well.

The bookkeeping involved is considerable.

Definition 17.10 (Virtual memory region)**.** *To facilitate tracking the mappings between virtual and physical addresses, an OS typically divides an address space into a set of contiguous* ***virtual memory regions****.*

Remarks:

- These include the "segments" of the process – not those in Definition 17.3, but the text, data, bss, stack, etc. segments used by the program loader.
- Regions also provide a convenient unit of *sharing* between address spaces, either at the same virtual address in all cases, or backed by a *memory object* that can appear at different virtual addresses in different process address spaces.

17.3 Segmented paging

It is possible to combine segmentation and paging.

Definition 17.11 (Paged segments)**.** *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 Memory Management Unit.*

Remarks:

- This is the approach adopted by Multics [Org72], but it's also supported (though little used) by older 16-bit and 32-bit x86 processors [Int18]. In the 64-bit x86 architecture, the segmentation registers used for this still exist, but simply hold linear offsets into a flat, paged address space.

17.4 Page mapping operations

Each process has its own page table. At a high level, all operating systems provide three basic operations on page mappings, which in turn manipulate the page table for a given process:

Definition 17.12 (Map)**.** *A page* ***map*** *operation on an address space A:*

$$A.map(v, p)$$

– takes a virtual page number v and a physical page (or frame) number p, and creates a mapping $v \rightarrow p$ in the address space.

Definition 17.13 (Unmap)**.** *A page **unmap** operation on an address space A:*

A.unmap(v)

– takes a virtual page number v and removes any mapping from v in the address space.

Definition 17.14 (Protect)**.** *A page **protect** operation on an address space A:*

A.protect(v, rights)

– takes a virtual page number v and changes the page protection on the page.

An MMU typically allows different protection rights on pages. The rights are specified in the page table entry, so they apply to *virtual* addresses. For example:

- READABLE: the process can read from the virtual address
- WRITEABLE: the process can write to the virtual address
- EXECUTABLE: the process can fetch machine code instructions from the virtual address

An OS will typically change protection rights on pages not only to protect pages, but also to cause a trap to occur when the process accesses a particular page in a particular way.

17.5 Copy-on-write

Recall that the `fork()` operation in UNIX makes a complete copy of the address space of the parent process, and that this might be an expensive operation.

Definition 17.15 (On-demand page allocation)**.** *. To avoid allocating all the physical pages needed for a process at startup time, **on-demand page allocation** is used to allocate physical pages lazily when they are first touched.*

Algorithm 17.16 On-demand page allocation

---

inputs
  A {An address space}
  $\{(v_i), i = 1 \ldots n\}$ {A set of virtual pages in A}

Setup the region

for $i = 1 \ldots n$ **do**
  A.unmap(v_i) {Ensure all mappings in region are invalid}
end for

Page fault

inputs
  v' {Faulting virtual address }
$n \leftarrow$ VPN(v')
$p \leftarrow$ AllocateNewPhysicalPage()
A.map($v_n \rightarrow p$)
return

Remarks:

- This technique uses the MMU to *interpose* on accesses to memory, and cause a trap when memory is touched for the first time. After the trap, the virtual memory has been backed and the cost of the trap doesn't occur again
- The general technique of "trap and fix things up" is very flexible. In particular, it is used to make `fork()` extremely efficient on modern machines via *copy-on-write*.

Definition 17.17 (Copy-on-Write)**.** . ***Copy-on-write*** *or COW is a technique which optimizes the copying of large regions of virtual memory when the subsequent changes to either copy are expected to be small.*

Algorithm 17.18 Copy-On-Write

1: **inputs**
2: A_p {Parent address space}
3: A_c {Child address space}
4: $\{(v_i, p_i), i = 1 \dots n\}$ {A set of virtual to physical mappings in A_p}

Setup

5: **for** $i = 1 \dots n$ **do**
6: A_p.protect(v_i, READONLY)
7: A_c.map($v_i \rightarrow p_i$)
8: A_c.protect(v_i, READONLY)
9: **end for**

Page fault in child

10: **inputs**
11: V {Faulting virtual address }
12: $n \leftarrow$ VPN(V)
13: $p' \leftarrow$ AllocateNewPhysicalPage()
14: CopyPageContents($p' \leftarrow p_n$)
15: A_c.map($v_n \rightarrow p'$)
16: A_p.protect(v_n, WRITEABLE)
17: **return**

Remarks:

- Essentially, we share the original physical pages between both address spaces, and mark both mappings read-only. When either process writes to the region, a protection fault will occur. When it does, we identify the page, allocate a new physical page, copy the contents across, and replace its mapping in the faulting process. Finally, we mark the original page mapping in the other process as writeable.
- There is no need for the processes involved to be parent or child, or for there to be only two of them. The technique is widely used, but the canonical example for this is copying the entire process address space in `fork()`.

17.6 Managing caches

We've already seen in Systems Programming how caches work, and how cache coherency protocols ensure varying levels of consistency among multiple caches. What about how the OS manages caches? Before looking at why and how the OS needs to manage the processor caches, let's go through the operations it can use from software on a cache.

Definition 17.19 (Cache Invalidate)**.** *An* ***invalidate*** *operation on a cache (or cache line) marks the contents of the cache (or line) as invalid, effectively discarding the data.*

Example 17.20. *The x86* `invd` *instruction invalidates* ***all*** *a processor's caches.*

Example 17.21. *The ARMv8-A* `dc isw` *instruction invalidates a specific set and way of the core's cache.* `dc ivac` *invalidates any line holding a specified virtual address "to the Point of Coherency".*

Remarks:

- Invalidate operations *generally* don't care about whether there is dirty data in the cache line or lines being invalidated – it just gets thrown away regardless. However, different hardware vendors use the word in different ways, so it's a good idea to check.

Definition 17.22 (Cache Clean)**.** *A* ***clean*** *operation on a cache writes any dirty data held in the (write-back) cache to memory.*

Example 17.23. *The ARMv8-A* `dc vac` *instruction writes dirty data in any line holding a specified virtual address back "to the Point of Coherency". It* ***may****, in addition, invalidate the line.*

Remarks:

- "Clean" is a term usually associated with the ARM architecture, but it is at least unambiguous in that it leaves the cache (or line) clean, but not necessarily invalid. Rather more ambiguous is...

Definition 17.24 (Flush)**.** *A* ***flush*** *operation writes back any dirty data from the cache and then invalidates the line (or the whole cache).*

Example 17.25. *The x86* `wbinvd` *instruction flushes* ***all*** *a processor's caches. The* `clflush` *instruction, in contrast, flushes a single cache line.*

Remarks:

- This is a reasonable definition of "Flush", and is at least consistent with the use of the term in Intel, ARM, and MIPS documentation (and most OS people's heads). However, don't assume when you read "flush" in some documentation that it always means this precisely.
- ARM do use this term, though when it really matters for clarity they write "clean-and-invalidate", which is unambiguous.

Given that caches are supposed to be transparent to software, though, why would you need these operations? Let's look at the kinds of problems that occur, and then survey the types of caches we have as well.

17.6.1 Homonyms and Synonyms

Definition 17.26 (Cache synonyms)**.** ***Synonyms*** *are different cache entries (virtual addresses) that refer to the same physical addresses. Synonyms can result in cache* ***aliasing****, where the same data appears in several copies in the cache at the same time.*

Remarks:

- Synonyms cause problems because an update to one copy in the cache will not necessarily update others, leaving the view of physical memory inconsistent.
- Worse, in a write-back cache, a dirty symonym may only be written back much later, causing an unexpected change to the contents of main memory. In a virtually-tagged cache (see below), this can even happen after virtual-to-physical page mappings have changed, creating a write to an arbitrary physical page. This is sometimes called a *cache bomb*.

Definition 17.27 (Cache homonyms)**.** *In contrast, cache* ***homonyms*** *are multiple physical addresses referred to using the same virtual address (for example, in different address spaces).*

Remarks:

- Homonyms are a problem since the cache tag may not uniquely identify cache data, leading to the cache accessing the wrong data. They are a common problem in conventional operating systems

Whether homonyms and synonyms are actually a problem the OS has to deal with depends on the cache hardware in use:

17.6.2 Cache types

These days, almost all processor caches are write-back, write-allocate, and set-associative. The main differences (other than size, and inclusivity) are to do with where the tag and index bits come from during a lookup.

Definition 17.28 (virtually-indexed, virtually-tagged)**.** *A* ***virtually indexed, virtually tagged*** *or VIVT cache (sometimes, ambiguously, also called a "virtual cache") is one where the virtual address of the access determines both the cache index and cache tag to look up.*

Remarks:

- VIVT caches are simple to implement, and fast. However, they suffer from homonyms: since the same virtual address in different address spaces refers to different physical addresses, it may be necessary for the OS to flush the cache on every address space switch, which is expensive. Early ARM processors had this "feature", for example.
- The problem can be alleviated by allowing cache entries to be annotated with "address space tags".

Definition 17.29 (Address space tags)**.** *, **Addresss-space tags**, or **ASIDs (Address Space Ids)** are small (e.g. 5-8 bits) additional cache or TLB tags which match different processes or address space and therefore allow multiple contexts to coexist in a cache or TLB at the same time*

Remarks:

- ASIDs improve performance because the cache doesn't need to be flused on a contact switch. In practice, VIVT caches are rare (see VIPT caches below), so their real impact is in TLBs, where they are almost ubiquitous these days.
- In practice, the "current" ASID is held in a register in the CPU or MMU which is altered by the OS kernel every time the address space switches. For example, in the MIPS R3000 this is the `EntryHi` register in coprocessor `CP0` [**?**].
- ASIDs are typically small, from 5 to 8 bits in size depending on the processor architecture. Obviously there are likely to be rather more processes active in a typical system. As a result, ASIDs should be regarded as a cache for true process or address space IDs: if the OS needs to dispatch a process which doesn't yet have an ASID assigned to it, it needs to reallocate an ASID, flush the cache to make sure no older entries for that ASID exist, and then dispatch the process.
- In the cast of a VIVT cache, ASIDs don't solve the further problem of *synonyms*: if two processes share physical memory, the same data can appear in the cache twice, making it hard to retain consistency between these two copies. For this reason, VIVT caches tend to be write-through.

Definition 17.30 (Physically-indexed, Physically-tagged)**.** *A physically-indexed, physically-tagged (PIPT) cache is one where the physical address after TLB translation determines the cache index and tag.*

Remarks:

- PIPT caches are easier to manage, since nothing needs to change on a context switch, and they do not suffer from homonyms or synonyms.
- The downside is that they are slow: you can only start to access a PIPT cache after the TLB has translated the address. For this reason, they are a common choice of L2 or L3 caches.

Definition 17.31 (Virtually-indexed, Physically-tagged)**.** *A virtually-indexed, physically-tagged (VIPT) cache is one where the virtual address before TLB translation determines the cache index, but the physical address after translation gives the tag.*

Remarks:

- VIPT caches are a great choice for L1 D-caches these days, *if* they can be made to work. Ideally, the virtual index is exactly what the corresponding physical index would be. This implies that the number of index bits, plus the number of cache offset bits, in a virtual address is less than the number of bits in the page size. This in turn restricts the size of a VIPT cache.

- If this condition doesn't hold, a VIPT cache can still work, but now the OS has to be careful about cache homonyms arising from the ambiguity now present in the cache contents. For example, in the MIPS R4400, the OS has to ensure that even and odd pages are allocated together and there is never a mapping from an even VPN to an odd PPN, or vice versa, since the cache index+offset bits are 1 bit larger than the page size.

Definition 17.32 (Physically-indexed, Virtually-tagged)**.** *A physically-indexed, virtually-tagged (PIVT) cache is one where the physical address after TLB translation determines the cache index to look up, but the virtual address before translation supplies the tag to then look up in the set.*

Remarks:

- It is hard to imagine anyone building such a cache, and even harder to figure out why, but they do exist: the L2 cache on the MIPS R6000 [**?**], for example, is PIVT. This requires considerable complexity on the part of the OS, and delivers questionable benefit. Thankfully, PIVT caches are extremely rare.

17.7 Managing the TLB

Like caches, the TLB is supposed to be transparent to user programs, although in practice as you have seen in Systems Programming it can be critical to know the TLB's dimensions to improve performance, for example by *tiling*, *blocking*, and *buffering* large dense matrics.

Definition 17.33 (TLB coverage)**.** *The* ***TLB coverage*** *of a processor is the total number of bytes of virtual address space which can be translated by a TLB at a given point in time.*

Example 17.34. *The venerable Motorola MC68851 Paged MMU has a 64-entry fully-associative TLB. When configured with a page size of 8k bytes, the coverage of this TLB is 64 x 8 = 512k bytes.*

Remarks:

- Modern processors are quite complex, with multiple TLBs: primary and secondary TLBs (as with L1 and L2 caches), separate TLBs for instructions and data, and even different TLBs for different page sizes (regular pages, superpages, etc.).

- However, looking purely at the regular page-sized TLB entries, TLB coverage has barely increased over the last 30 years. Even as far back as 2002, it was observed that TLB coverage as percentage of main memory size was dropping precipitously [NIDC02]. In some ways this is not too surprising: TLBs are highly-associative caches that need to be really fast, and Content-Addressable Memory circuits (CAMs) are very energy-inefficient, but it's a serious factor in the performance of computer software over time.

Regular processor caches on a multiprocessor machine typically try to remain *coherent*, in other words, they present a consistent view of the contents of main memory to the processors in the system. TLBs, by analogy, present a view of the structure of an address space to the different cores in a multiprocessor system, and so also need to be kept coherent. This this view changes less frequently than the contents of memory, it is done in software by the OS when mappings change.

Definition 17.35 (TLB shootdown)**.** ***TLB shootdown*** *on a multiprocessor is the process of ensuring that no out-of-date virtual-to-physical translations are held in any TLB in the system following a change to a page table.*

Remarks:

- Since the TLB is a cache, it should be coherent with other TLBs in the system, and with each process' page tables.
- Shootdown is basically a way to invalidate certain mappings in every TLB, if they refer to the affected process.
- Sometimes hardware does this, but more common is for one core to send an *inter-processor interrupt* to every other core which *might* have such a mapping, and have the kernel on that core remove the mappings from its own TLB.
- TLB shootdown is expensive, and so operating systems try to make it faster by, for example, trying to track which TLBs might be affected by a change in a page table. If the OS can prove that given TLB cannot hold a mapping, then it doesn't need to interrupt that TLB's processor when the mapping needs to be invalidated.

Bibliography

[Int18] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, volume 3. Intel Corp., 2018.

[KH91] Gerry Kane and Joseph Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 2nd edition, 1991.

[Leo87] Timothy E. Leonard. *VAX Architecture Reference Manual*. Digital Press, March 1987.

[NIDC02] Juan Navarro, Sitaram Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and implementationCopyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading*, OSDI '02, pages 89–104, Berkeley, CA, USA, 2002. USENIX Association.

[Org72] Elliott I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, USA, 1972.

Chapter 18

Demand Paging

Today, the prevailing view is that if you're paging to storage, either in a server or a mobile phone, you've already lost the game performance-wise.

It is difficult these days to imagine how transformative the idea of demand-paged virtual memory was at the time [KELS62]: it became possible to consider programs whose space requirements were not limited by a machine's memory. For the first time, a machine could precisely emulate a much more powerful machine (memory-wise).

However, the basic ideas may be returning: persistent main memory (byte-addressable flash, phase-change, memristor, etc.) may be the only option to scale to very large sizes (for power reasons), whereas traditional DRAM is the only thing that can endure lots of read/write cycles. We may end up paging between main memory addresses using `memcpy`.

The other important reason to cover demand paging in this course is that it is almost *the* canonical example of caching as a general system design principle. Many of the concepts below carry over into other caching scenarios which crop up again and again in systems.

18.1 Basic mechanism

Definition 18.1 (Demand paging)**.** ***Demand paging*** *uses page faults to exchange virtual pages on demand between physical pages in main memory, and locations in a larger* ***page file*** *held on cheaper persistent storage.*

Remarks:

- This is the traditional use of virtual memory: making a small (and expensive) main memory plus a large (and relatively cheaper) disk look like a large main memory.

- Ideally, this larger virtual memory would not be too much slower than a real one of the same size, for a typical workload.

- This is, essentially, caching. However, since the access time to fetch a virtual page from disk is much higher than in a processor cache, and the transfer unit (a page) is much larger than a typical cache line,

software can be much smarter about page replacement (though not as smart as in wide-area network caches).

- Demand paging is *lazy*: it only loads a page into memory when a virtual address in the page has been touched by the processor (or at least its cache) – a demand pager was sometimes called a *lazy swapper* for this reason.

- A note on terminology: I've removed any reference to **frames** from this script, but you'll hear me talk about them. A frame is the same as a physical page (think of it as something that contains virtual page).

The process works roughly as follows, but note that this algorithm assumes that all accesses are actually valid (the check that throws a fatal error on an invalid access is omitted), and we also ignore concurrency issues entirely for now:

Algorithm 18.2 Demand paging: page fault handling

On a page fault with faulting VPN v_{fault}*:*

1: **if** there are free physical pages **then**
2: $p \leftarrow$ `get_new_pfn()`
3: **else**
4: $p \leftarrow$ `get_victim_pfn()`
5: $v_{old} \leftarrow$ VPN mapped to p
6: invalidate all TLB entries and page table mappings to p
7: **if** p is dirty (modified) **then**
8: write contents of p into v_{old}'s area in storage
9: **end if**
10: **end if**
11: read page v_{fault} in from disk into physical page p
12: install mapping from v_{fault} to p
13: **return**

18.2 Paging performance

Remarks:

- The performance of a demand paging system is critically dependent on how many page faults are generated for a workload (see below). The goal is to minimize these page faults.

- The critical part of Algorithm 18.13 for performance is what happens in `get_victim_pfn()`, in other words what the *page replacement algorithm* is.

Definition 18.3 (Page replacement policy)**.** *The* ***page replacement policy*** *of a demand paging system is the algorithm which determines which physical page (the* ***victim page****) will be used when paging a virtual page in from storage.*

Page replacement algorithms matter because the performance of a computer when it is paging can be completely dominated by the overhead of paging.

How do we measure the performance or efficiency of a paging system?

Definition 18.4 (Effective Access time)**.** *The average time taken to access memory, over all memory references in a program, is the* ***Effective Access Time****.*

Consider a page fault rate $p, 0 \leq p \leq 1.0$. If $p = 0$, we have no page faults. Similarly, if $p = 1$, then every memory reference causes a page fault (unlikely, but not unheard of).

Then the Effective Access Time is:

$$EAT = ((1 - p) \times m) + (p \times (o + m))$$

– where m is the memory access latency, and o is the paging overhead.

Remarks:

- In practice, o is the sum of a number of factors: page fault overhead itself, the cost of swapping a dirty virtual page out if required, the cost of swapping the required virtual page in, and the cost of restarting the instruction.

Example 18.5. *Suppose $m = 50ns$, and on average o is 4ms (these are plausible figures for a disk-based system).*

Then the EAT in nanoseconds is $((1–p) \times 50) + (p \times 4,000,050))$

If only one access in 1,000 causes a page fault, i.e. $p = 0.001$, then $EAT = 4\mu s$, and we have a slowdown over main memory of a factor of 80.

Analyzing the performance of a paging system is a rather under-specified problem. In general, it's good to fix some things in advance.

Definition 18.6 (Reference String)**.** *A* ***Reference String*** *is a trace of page-level memory accesses. A given page replacement algorithm can be evaluated relative to a given reference string.*

Remarks:

- Reference strings capture the essential part of a workload, without having to look at what a program is actually doing.
- Generally, the best paging replacement policy minimizes the number of page faults for a given reference string.
- Reference strings are useful in analysing page replacement algorithms, as we will see below.

18.3 Page replacement policies

What is the optimal page replacement strategy?

Algorithm 18.7 Optimal page replacement

When a victim page is required

1: **return** The virtual page that will not be referenced again for the longest period of time.

Remarks:

- This algorithm is optimal, in that it minimizes the number of page faults for any given reference string. The proof is left as an exercise.
- It requires knowing the reference string in advance, which is generally not the case (except for some deterministic real-time systems). As a result, it's almost never used, but is useful to provide a baseline for performance comparisons.

The following algorithm requires no knowledge at all about the reference string:

Algorithm 18.8 FIFO page replacement

Return a new victim page on a page fault

1: **inputs**
2: pq: a FIFO queue of all used PFNs in the system.
3: $p \leftarrow pq.pop_head()$
4: $pq.push_tail(p)$
5: **return** p

Remarks:

- Intuitively, FIFO is not the best cache replacement policy in most cases (though you might be able to think of a case where it is), but it is simple to implement.
- You might also think, intuitively, that increasing the size of memory (i.e. increasing the number of physical pages available to hold virtual pages) would increase the efficiency of the paging algorithm by decreasing the number of times a victim page had to be evicted. Surprisingly, this is **not** the case for all page replacement algorithms, and is famously not the case for FIFO.

Definition 18.9 (Bélády's Anomaly)**.** ***Bélády's Anomaly*** *is the behavior exhibited by some replacement algorithms in caching systems (notably demand paging), where increasing the size of the cache actually reduces the hit rate for some reference strings.*

Example 18.10. *FIFO page replacement exhibits Bélády's Anomaly. Consider the following reference string (from the original paper by Bélády, Nelson, and Shedler [BNS69]):*

1 2 3 4 1 2 5 1 2 3 4 5

In a machine with 3 physical pages available, this reference string will result in 9 page faults. With 4 physical pages available, the result is 10 page faults.

In practice, we need a dynamic, online page replacement algorithm. Consider Least Recently Used:

Algorithm 18.11 A Least Recently Used (LRU) page replacement implementation

Initialization

1: **inputs**
2: S: Stack of all physical pages $p_i, 0 \leq i < N$
3: **for** all p_i **do**
4: p_i.referenced $\leftarrow$ False
5: S.push(p_i)
6: **end for**

When a page p_r is referenced

7: S.remove(p_r)
8: S.push(p_r)

When a victim page is needed

9: **return** Return S.remove_from_bottom()

Remarks:

- LRU is hard to beat performance-wise, though not impossible: see e.g. ARC [MM03].

- It can be easily implemented using a stack, as shown above. The general class of algorithms with this property are called *stack algorithms*. No stack-based algorithm exhibits Bélády's Anomaly.

- LRU needs detailed information about page references. The OS sees every page fault, but in general does not get informed about every memory access a program makes. This makes LRU impractical for a general-purpose OS on typical hardware, but it is the usual baseline choice for other types of cache.

- What a modern machine and OS *can* do is track which virtual pages have been referenced at some unspecified time in the recent past (and which ones have been modified). This is done by setting a flag associated with a page whenever the page is accessed, and then periodically clearing it after reading it.

Algorithm 18.12 2nd-chance page replacement using reference bits

Initialization

```
1: inputs
2:   F: FIFO queue of all physical pages p_i, 0 ≤ i < N
3: for all p_i do
4:   p_i.referenced ← False
5:   F.add_to_tail(p_i)
6: end for
```

When a physical page p_r is referenced

```
7: p_r.referenced ← True
```

What a victim page is needed

```
8: repeat
9:   p_h ← F.remove_from_head()
10:  if p_h.referenced = True then
11:    p_h.referenced ← False
12:    F.add_to_tail(p_h)
13:  end if
14: until p_h.referenced = False
15: return p_h
```

Remarks:

- On x86 machines, the MMU hardware provides a "referenced" (or "accessed" bit), in addition to a dirty bit, inside the PTE. On ARM machines, this is not the case.

- The workaround for lack of hardware reference bits is to mark the virtual page invalid at the point where one would otherwise clear the reference bit. The next time the virtual page is referenced the OS will take a trap on the reference, and can set the bit in software before marking the page valid and continuing. This works: you only pay the overhead of a trap for the first reference; after that it's free.

- Since the x86 hardware keeps the reference bit in the PTE, it is associated with the *virtual* page not the physical page (which is what you might want), so extra bookkeeping is needed anyway.

- This algorithm is so-called because each referenced virtual page gets a "second chance" before being evicted.

- This is a rough approximation to LRU, but we can get the same result more efficiently with conventional hardware.

Algorithm 18.13 "Clock" page replacement using reference bits

Initialization

1: **for** all physical pages $p_i, 0 \leq i < N$ **do**
2: p_i.referenced ← False
3: **end for**
4: Next physical page number $n \leftarrow 0$

When physical p_r is referenced

5: p_r.referenced ← True

When a victim page is needed

6: **while** p_n.referenced = True **do**
7: p_n.referenced = False
8: $n \leftarrow (n+1) mod N$
9: **end while**
10: **return** page p_n.

Remarks:

- This classic "clock" algorithm is the basis for most modern page replacement policies when detailed reference information is not available (note that in software caches, such as web caches, such reference information *is* easy to obtain since the software sees all web requests, and so LRU is practical).

- The term "clock" algorithm comes from the visual image of a clock hand (the next pointer n) sweeping round through memory clearing the referenced bits.

18.4 Allocating physical pages between processes

All the schemes we've seen so far operate on a single pool of physical pages, and don't differentiate between different processes or applications. What about trying to allocate different physical memory to different programs?

Definition 18.14 (Global physical page allocation)**.** *In **global physical page allocation**, the OS selects a replacement physical page from the set of all such pages in the system*

Remarks:

- This doesn't quite work: each process needs to have some minimum number of physical pages in order to be runnable at all. This can vary depending on the OS, but it's never zero, so you need to reserve some memory for each process.

- Also, we need to adjust for physical pages which are shared between multiple processes.

Definition 18.15 (Local physical page allocation)**.** *Instead, with **local physical page allocation** a replacement physical page is allocated from the set of physical pages currently held by the faulting process.*

Remarks:

- Local physical page allocation doesn't need to be fixed. The simplest scheme is indeed to allocate an equal share to all processes, but an alternative is to vary the proportion of physical pages dynamically, much as we have seen CPU schedulers vary to priority of a thread or process dynamically.
- Indeed, we can use priorities rather than quantitative measures of process "size" to give some processes more memory. However, this runs the risk of starving an important process which only needs a few physical pages, but suffers disproportionately when one of them is paged out.
- We could adopt a "rate-based" approach: we measure the number of page faults each process takes over some fixed time period, and reassign physical pages from processes with low fault rates to those with high rates.
- The problem with the rate-based approach is that processes with very large memory requirements can end up starving those with small requirements.
- This naturally leads to the question: how can we *define*, and then *measure*, the "size" of a running process?

Definition 18.16 (Working set)**.** *The **working set** $W(t, \tau)$ of a process at time t is the set of virtual pages referenced by the process during the previous time interval $(t - \tau, t)$. The **working set size** $w(t, \tau)$ of a process at time t is the $|W(t, \tau)|$.*

Remarks:

- The working set size of a process is usually a good approximation to how many pages of physical memory a process needs to avoid excessively paging pages to and from storage.
- It goes without saying that τ should be chosen carefully. However, it is often the case that above some reasonable value, the working set of a given execution typically doesn't increase with increasing window size τ beyond a certain point.
- Accurately measuring the working set of a process at any time t requires a complete list of virtual page references since time $t - \tau$. As with LRU page replacement, this information may be impractical to obtain.

The standard approach, listed in the original paper on the Working Set model [Den68], uses sampling at an interval $\sigma = \tau / K$, where K is an integer.

For each page table entry we keep a hardware-provided "accessed" bit u_0 and $K-1$ software-maintained "use bits" $u_i, 0 < i < K$.

An interval timer is programmed to raise an interrupt every σ time units.

Algorithm 18.17 Estimating the working set by sampling

On an interval timer every σ time units

1: $WS \leftarrow \{\}$
2: **for** all page table entries **do**
3: **for** all use bits $u_i, 0 < i < K$ **do**
4: $u_i \leftarrow u_{i-1}$ {Shift all use bits right}
5: **end for**
6: $u_0 \leftarrow 0$
7: $U \leftarrow \bigoplus_{i=0}^{K-1} u_i$ {Logical-or all the use bits together}
8: **if** $U = 1$ **then**
9: WS.add(page)
10: **end if**
11: **end for**
12: **return** WS {The working set $W(t, \sigma K)$}

Remarks:

- This principle, while developed for demand paging, is actually a general approach to any caching scenario.
- We find that, for the most part, when a program is allocated more physical pages than its working set, performance improves but not by much.
- When a program is allocated fewer physical pages than its working set, it starts to page excessively.

Definition 18.18 (Thrashing)**.** *A paging system is **thrashing** when the total working set significantly exceeds the available physical memory, resulting in almost all application memory accesses triggering a page fault. Performance falls to near zero.*

Remarks:

- The term "thrashing" comes from old, large hard disk drives. When a process thrashes, almost all the available time is spent paging pages in and out of memory to and from disk, and very little in actually executing the program. The load on the disk drive causes the disk arm the thrash back and forth, causing a lot of noise and vibration.

Working set estimation provides a basis for how many physical pages to ideally allocate to each process. After that, the problem is to come up with a good policy, much as with CPU scheduling. Indeed, as [Den68] points out, there is a close connection between CPU and memory allocation in a demand paging system.

This should not be surprising: the trick that makes a small machine look like a large machine can work because the (apparently) larger machine runs more slowly.

Nevertheless, this concept of *virtualization*, which we will see later in other forms, is fundamental to computer science, and is possibly unique to computers. The thing that makes a computer special is its ability to virtualize itself.

Bibliography

[BNS69] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM*, 12(6):349–353, June 1969.

[Den68] Peter J. Denning. The working set model for program behavior. *Commun. ACM*, 11(5):323–333, May 1968.

[KELS62] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. *IRE Transactions on Electronic Computers*, EC-11(2):223–235, April 1962.

[MM03] Nimrod Megiddo and Dharmendra Modha. Arc: A self-tuning, low overhead replacement cache. In *In Proceedings of the 2003 Conference on File and Storage Technologies (FAST*, pages 115–130, 2003.

Chapter 19

File system abstractions

Definition 19.1 (The filing system)**.** ***The Filing System*** *(as opposed to "a filing system", which we'll see below) is the functionality in an operating system which provides the abstractions of files in some system-wide namespace.*

Remarks:

- The filing system *virtualizes* the collection of stable storage devices in the system (and possibly some other resources as well), in the same way that the virtual memory system virtualizes main memory.
- As with any other virtualization function, this is a combination of *multiplexing* (sharing the storage between applications and users), *abstraction* (making the devices appear as a more convenient collection of files with consistency properties) and *emulation* (creating this illusion over an arbitrary set of storage devices).
- The Filing System comprises the core functionality for accessing files, together with a set of additional maintenance *utility programs* which are usually not needed by regular users.

This chapter is about the abstractions provided by the filing system; the next one is about how these are actually implemented.

The key filing system abstractions can be divided into two groups: (1) files, directories, name spaces, access control, and other issues relating to storage of data itself, and (2) open file descriptors and other abstractions that allow access to this data from programs.

19.1 Access control

Access control goes far beyond file systems, but this is a convenient place to introduce it. Access control is about deciding which *access rights* a given *principal* or *subject* has on a given *object*.

Definition 19.2 (Principal, Object, Right)**.** *A security* ***principal*** *or* ***subject*** *is the entity to which particular access* ***rights*** *are ascribed, which grant to ability to access particular* ***objects****. Note that a principal can also function as an object of rights.*

Example 19.3. *In a file system, for example,* ***principals*** *can be system users,* ***objects*** *are files, and* ***rights*** *include* `read`*,* `write`*,* `execute`*, etc.*

Definition 19.4 (Protection domain)**.** *The* ***protection domain*** *of a principal is the complete set of objects that the principal has some rights over.*

We think of access control for a complete system as represented by:

Definition 19.5 (Access control matrix)**.** *. The* ***access control matrix*** *is a table whose rows correspond to principals, and whose columns correspond to objects. Each element of the matrix is a list of the rights that the corresponding principal has over the corresponding object.*

Remarks:

- The Access control matrix was introduced by Butler Lampson in 1971 [Lam71].
- The rows of the matrix represent the protection domain of each principal, whereas the columns represent the *accessibility* of an object: who can do what do it.
- The access control matrix is fully general for *static* access control. Once you start to worry about which principals have the right (or not) to delegate a right that they have to other principals, you need a more sophisticated model, outside the scope of this course.
- If materialized, the access control matrix for a large file system would be prohibitively large. Consequently, access control models in file systems are all about trying to represent this matrix in a compact form, perhaps by limiting its expressivity.

Definition 19.6 (Access control list)**.** *An* ***access control list****, or* ***ACL****, is a compact list of non-zero entries for a column of the access control matrix; i.e. an encoding of the set of principals that have any rights over the object, together with what those rights are.*

Remarks:

- Historically, the ACL is one of the two main models of *discretionary access control.* The other is an encoding of the rows of the matrix, which correspond roughly to a list of *capabilities* held by the principal.
- ACLs make it easy to change the rights on an object quickly.
- This ACL vs. capabilities split is convenient, but is an oversimplification in practice.
- We concentrate here on ACLs, since they are the model adopted by most major file systems. The ACL can be stored *with the file* as part of its metadata, which means that ACLs can scale to large numbers of files (objects).
- ACLs conversely find it hard to represent very large numbers of *principals*, since the lists get very long. Most real ACL schemes have ways of grouping principals together to mitigate this.

Example 19.7 (Authorization in Microsoft Windows)**.** *Microsoft Windows has a powerful ACL system based on the MIT Kerberos authorization framework. ACLs can be applied to all kind of objects in Windows, not simply files, using the same scheme.*

Windows ACLs support the creation of arbitrary groups of principals. Each group is itself a principal, and there no serious limits on the number of principals on a ACL.

Moreover, Windows supports sophisticated rights, including delegation.

Example 19.8 (POSIX (UNIX) access control)**.** *Traditional* UNIX *access control is a rather simplified form of ACLs, which limits each ACL to exactly 3 principals:*

1. *A single user, the **owner** of the file.*
2. *A single group of users. Groups are defined on a system-wide basis.*
3. *"Everyone else", i.e. the implicit group composed of all users not the owner and not in the named group.*

Moreover, for each principal, the ACL defines 3 rights: `read` *(or list, if the file is a directory),* `write` *(or create a file, for directories), and **execute** (or traverse, if a directory).*

Remarks:

- Today, the POSIX standard now supports full ACLs similar to Windows, although they are rarely used. If you are interested, `man getfacl`.
- There's more to even traditional UNIX ACLs that we haven't covered yet: sticky bits, for example. We will see some of these later.

19.2 Files

What, exactly, is a file?

Definition 19.9 (File)**.** *A **file** is an abstraction created by the operating system that corresponds to a set of data items (often bytes), with associated metadata.*

Remarks:

- The idea of a file sits between the fixed-size "blocks" typically provided by the storage devices (disks, SSDs, etc.) and the programmer's abstraction of data.
- Files in UNIX are *unstructured*: their contents are simply vectors of bytes. This is not true on some other systems (such as mainframes) where the abstraction of a file can be quite rich and include fixed-length records, for example, bringing the filing system closer to a database.

Associated with a file aside from its data is metadata.

Definition 19.10 (File metadata)**.** *The **metadata** of a file is additional information about a file which is separate from its actual contents (data), such as its size, **file type**, access control information, time of creation, time of last update, etc. It also includes, critically, where the file's data is **located** on the storage devices.*

Remarks:

- The "type" of a file is not a well-defined concept, but can include whether the file is structured or not, if it's an executable (though in most systems that is part of the name rather than the metadata), or if the file is "special" (such as device files in UNIX, which actually represent hardware devices rather than data sets).

- What about the file name? It's common to *not* include the name in the metadata, since (as you will recall from Chapter **??**), there can be multiple names for given file. The name of a file is, instead, considered part of the directory or catalog entry for the file.

19.3 The Namespace

Definition 19.11 (Filename)**.** *A **filename** is a name which is bound to a file in the namespace implemented by the filing system.*

Remarks:

- More than one name can refer to the same file, in general, though some file systems forbid this.

- The organization of the file system namespace can vary quite a bit.

Example 19.12 (IBM MVS)**.** *IBM Mainframes actually have a flat namespace, though the names have to consist of sequences of 1-8 alpha-numeric characters separated by spaces. Users can set a "default prefix" to be applied to shortened versions of these names for files ("datasets" in IBM parlance). All the filenames in the system are held centrally, which makes for extremely fast name lookups.*

Example 19.13 (DOS)**.** *In the old MS-DOS operating system for PCs, all filenames consist of a "drive letter", followed by 8 characters of name, plus an extra 3 for "type" (such as `A:\AUTOEXEC.BAT`).*

Example 19.14 (UNIX)**.** *In UNIX, the namespace is hierarchical (actually, it's a directed **acyclic** graph - we will see why later).*

Since UNIX-like systems (including MacOS) are fairly common, and even Windows follows a similar structure these days, the rest of this section will assume we're talking about the UNIX file system. Be aware, though, that others can be different.

19.4 The POSIX namespace and directories

What follows is a lengthy description of the API to the UNIX file system, but written from a more abstract perspective, relating it to the ideas we saw back in the Naming chapter, for example. For a more programmer-oriented view, the manual pages for the system called (`link()`, `open()`, etc.) are helpful.

In UNIX, the file system consists of a single, system-wide namespace organized as a directed acyclic graph.

Definition 19.15 (Directory). *A UNIX **directory** (also called a **folder** on other systems) is a non-leaf node in the naming graph. It implements a naming context.*

Remarks:

- A directory supports typical namespace operations: binding a name to a file is done explicitly by a `link()` operation which creates a new finding to an existing file, or implicitly when a file (or directory) is created for the first time (using `open()`, `creat()`, etc.).
- Bindings can be removed from a directory using `unlink()`.
- You can also enumerate a directory's entries by `open()` it, and using the `readdir()` system call.
- Directories are, really, a special kind of file. The file metadata for a directory identifies it as such, and causes the filing system to treat it differently. We'll see more of this next time.
- One difference, however, is that the filing system only allows a directory to be bound *once* in the whole system, as opposed to files – in other words, you can't used `link()` to bound a name to an existing directory (try it – you should get an `EPERM` error). This forces the naming graph to be a DAG by preventing cycles.
- All directories must always contain two particular name bindings: "`.`" (dot) is always bound to the directory itself, and "`..`" (dot-dot) is bound to the "parent" of this directory.
- The existence of "`..`" is controversial, but at least has a well-defined meaning since the naming graph is acyclic (apart from the trivial cycles introduced by "`.`" and "`..`".

Definition 19.16 (Current working directory). *Every process has, as part of its context, a **current working directory**. Any path name which does not start with a "/" is resolved starting at this directory.*

Remarks:

- You can display the current working directory's name by typing "`pwd`".

Definition 19.17 (Root directory).

*There is a distinguished directory whose name is simple "/". This is called the **root** of the file system. All path names which start with "/" are resolved relative to this root.*

Remarks:

- In the root, both "`.`" and "`..`" are bound to the root. In other words, the root is its own parent.
- It is, in fact, possible to change the root directory of a process using `chroot()`. This results in the process only seeing a restricted subset of the file namespace. Implementing some form of security in this case is, actually, rather difficult and fraught with potential errors (see "open files" below).

Definition 19.18 (Symlink)**.** *A **symbolic link** or **symlink** is a name in a directory bound not to a file or directory, but instead to another name.*

Remarks:

- We saw symbolic links in the Naming chapter; here they are in UNIX.
- They are created in UNIX using the `symlink()` system call, or "`ln -s`" in the shell.
- Technically, a symlink is a special kind of directory entry, but the equivalent in Windows (the "shortcut") is a actually a special kind of *file*, whose contents is the destination name.

There is a clear distinction between a *file* itself, and the *directory entry* referring to the file – there can be multiple instances of the latter corresponding to the same file. Most of the metadata of a file in UNIX is actually stored in a separate object which actually represents the file: the *inode*, which we will see in the next chapter.

However, a more important distinction from the user's perspective is between a *file*, and an *open file*. The most important operation on a *file* is `open()`, which creates an open file.

19.5 Open Unix files

Definition 19.19 (Open file descriptor)**.** *An **open file** is an object which represents the context for reading from, writing to, and performing other operations on, a file. An open file is identified in user space by means of an **open file descriptor**.*

Remarks:

- Almost all operating systems have some equivalent object to an open file, although the details (and how it is referred to) vary.
- An open file descriptor in UNIX is a small integer which names the open file (note that this has nothing to do with the name used to open the original file), and is returned by the `open()` system call.
- Almost all the file operations that regular programs are interested in (read, write, seek, etc.) are, strictly speaking, operations on an *open files*, not a *file* per se.

- The naming context for file descriptors is local to the process, but it is possible to pass a file descriptor from one process to another through an IPC socket. When this happens, a new number is allocated in the destination process.
- File descriptors are, actually, a simple form of *capability*, as defined above.
- The capability-like properties of file descriptors mean that they are used to represent all kind of other objects in Unix apart from open files: sockets, shared memory segments, etc.

What can you actually do with an open file? To some extent this depends on the access method.

Definition 19.20 (File access method)**.** *An **access method** defines how the contents of a file are read and written.*

Definition 19.21 (Direct access file)**.** ***Direct access** (or **random access** is an access method whereby any part of the file can be accessed by specifying its offset.*

Remarks:

- This is the primary access method defined by UNIX. The interface for this in UNIX should be familiar to you by now: each open file has a *file position indicator* which keeps track of an offset in the file. `read()` and `write()` operations start at this position, and `lseek()` changes (and reports) the position indicator.

Definition 19.22 (Sequential access)**.** ***Sequential access** is an access method under which the file is read (or written) strictly in sequence from start to end.*

Remarks:

- For mainframes, sequential access is quite important. Many programs want to simply read a file from start to finish, and if you know this in advance, you know you can usefully prefetch the file. It's also a good match for tape drives, which are still big business.
- UNIX has a few limited forms of sequential access. Simply reading or writing a regular open file moves the file position indicator long by the number of bytes read or written. This is also the reason that a convenience function in the C library to set the file position indicator to 0 (the start of the file) is called `rewind()`.
- You can also open a file in "append" mode, in which case all writes are constrained to add sequentially to the end of the file.
- One of the simplifications that UNIX introduced, however, was to really have only a single kind of file, and not much file access. Tape access in UNIX is deliberately not provided by a file system at all, but instead built into utility programs like `tar` and `cpio`.

Definition 19.23. *Structured filesIn contrast to files which are presented simply as a vector of bytes,* ***structured files*** *contain* ***records*** *which are, to some extent, understood by the file system.*

Remarks:

- It's easy to forget that a plan Unix file is really an abstraction, with a complex implementation hidden behind it. The extreme simplicity of the abstraction (a vector of bytes) is the reason people often forget this.
- Again, in mainframes, structured files can be quite common, sometimes resembling database tables.
- In the early Macintosh operating system (MacOS), all files were structured as two "forks": the "data fork" which was an untyped byte vector, and the "resource fork" which was a key-value store mapping pairs of (type identifier, object identifier) 32-bit values to structured data.
- In Unix, structured files barely exist, but we have already seen one type: directories! A directory is a file, but is accessed not as bytes but as a set of entries using *readdir()*.

At some point, the boundary between structured files or exotic file access methods implemented by the operating system, and the same functionality provided on top of simple, byte-oriented, direct access files by application programs, begins to blur.

In Unix, the current consensus is towards putting file structure into database applications, or other applications, but as we have seen, it's not a hard boundary, and it can change. New storage technologies are likely to change this further, as are the constraints imposed by the current POSIX file interface on multiprocessor scaling.

19.6 Memory-mapped files

Definition 19.24 (Memory-mapped file)**.** *A* ***memory mapped file*** *is an open file which is accessed through the virtual memory system.*

Remarks:

- In Unix, a file can be memory mapped using the `mmap()` system call.
- Memory-mapping a file creates a region of virtual memory which is actually paged back to the named file (rather than the OS regular paging file or device).
- After memory mapping, file contents can be accessed using regular loads and stores to and from main memory
- The operation of memory mapping files is actually overloaded in Unix to perform a variety of other tasks, such as creating shared memory segments (which are also referred to via file descriptors).

19.7 Executable files

Definition 19.25 (Executable files)**.** *An **executable file** is one which the OS can use to create a process.*

Remarks:

- To some extent, an executable file is simply one whose metadata labels it as executable. However, in practice the OS needs to know *how* to create the process given the file, for example if it is an ELF file, a script, etc.
- The ELF files we saw last year (and other executable binary formats produced by a linker) are identified by a *magic* few bytes at the start of the file. The UNIX kernel *loader* then decodes the file format to figure out where the text, data, and bss segments go, etc.
- In UNIX, the magic bytes '`#`', '`!`' indicate a *script*: the loader reads the rest of the line to give the pathname of a binary which is then launched with the name of the original file as the first argument.
- Where possible, the text segment of a program (marked read-only) will page back to the original file, to save space in the regular page file.
- In Windows, any file which is currently executing is locked by the OS to prevent modifications to the code while the program is running.

Bibliography

[Lam71] Butler W. Lampson. Protection. In *Proc. Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971. reprinted in Operating Systems Review, 8,1, January 1974, pp. 18–24.

Chapter 20

File system implementation

How are file systems implemented? Let's start at the bottom.

20.1 Low-level file system basics

Definition 20.1 (Volume)**.** *A **volume** is the generic name for a storage device, or something which resembles it. A volume consists of a contiguous set of fixed-size **blocks**, each of which can be read from or written to.*

Remarks:

- Volumes are a convenient abstraction that includes spinning disks, flash disks, logical partitions (defined below), and other ways of creating a "virtual disk".
- In addition to splitting up a physical device into logical volumes, volumes can also be constructed by adding together a collection of devices and making them appear as a single volume, as with RAID.

Definition 20.2 (Logical block addresses)**.** *It is convenient to refer to every block on a volume using its **logical block address** (LBA), treating the volume as a compact linear array of usable blocks.*

Remarks:

- LBAs nicely abstract such features of disks as tracks, sectors, spindles, etc. They also hide wear-leveling mappings in Flash drives, storage-area networks, RAM disks, RAID arrays, etc. behind a simple, convenient interface.

Definition 20.3 (Disk partitions)**.** *At a level below files, disks or other **physical volumes** are divided into contiguous regions called **partitions**, each of which can function as a volume. A **partition table** stored in sectors at the start of the physical volume lays out the partitions.*

Remarks:

- Partitions coarsely multiplex a disk among file systems, but aren't really much of a file system themselves.

Example 20.4. *Here is a Linux system with a single hard disk drive with four different partitions on it:*

```
troscoe@emmentaler1:~$ sudo parted
GNU Parted 3.2
Using /dev/sda
Welcome to GNU Parted! Type 'help' to view a list of commands.
(parted) print
Model: SEAGATE ST3600057SS (scsi)
Disk /dev/sda: 600GB
Sector size (logical/physical): 512B/512B
Partition Table: msdos
Disk Flags:

Number  Start   End     Size    Type     File system     Flags
 1      1049kB  16.0GB  16.0GB  primary  ext4            boot
 2      16.0GB  20.0GB  4000MB  primary  ext4
 3      20.0GB  36.0GB  16.0GB  primary  linux-swap(v1)
 4      36.0GB  600GB   564GB   primary  ext4
```

Definition 20.5 (Logical volume)**.** *In addition to splitting a single physical device into multiple volumes,* ***logical volumes*** *are created by merging multiple physical devices into a single volume.*

Remarks:

- The best framework for thinking about logical volumes, and the reason we don't discuss them further in this chapter, is as *distributed storage*: data can be replicated across physical volumes for durability, or striped across them for performance, or all kinds of combinations, just as in more widely distributed storage systems.

Definition 20.6 (A file system)**.** ***A file system*** *(as opposed to* **the** *file system) is a set of data structures which fill a volume and collectively provide file storage, naming, and protection. The term can also mean the emphimplementation (class) of the file system.*

Remarks:

- The file system we saw in the previous chapter (that bit of the OS which provides files) provides the interface to, and implementation of, one or more of the *file systems* defined as above.
- Typically, each separate file system occupies a different part of the global name space, though this is not necessarily true.

Example 20.7. *In the following output, there are 15 different individual file systems which collectively make up the file name space of the machine:*

```
troscoe@emmentaler1:~$ df
Filesystem                      1K-blocks       Used   Available Use% Mounted on
udev                             12316068          0    12316068   0% /dev
tmpfs                             2467336       1120     2466216   1% /run
/dev/sda1                        15247760   10435252     4014916  73% /
tmpfs                            12336676        232    12336444   1% /dev/shm
tmpfs                                5120          0        5120   0% /run/lock
tmpfs                            12336676          0    12336676   0% /sys/fs/cgroup
/dev/sdb1                      1922728840  981409016   843627764  54% /mnt/local2
/dev/sda2                         3779640     339748     3228180  10% /tmp
/dev/sda4                       542126864  298706076   215859216  59% /mnt/local
fs.roscoe:/e/g/home/harness    1043992576  710082560   333910016  69% /home/harness
fs.systems:/e/g/netos          1043992576  812044288   231948288  78% /home/netos
fs.roscoe:/e/g/home/haeckir    1043992576  710082560   333910016  69% /home/haeckir
fs.roscoe:/e/g/home/gerbesim   1043992576  710082560   333910016  69% /home/gerbesim
iiscratch-zhang:/e/s/systems  77309411328 6010442752 71298968576   8% /mnt/scratch-n
fs.roscoe:/e/g/home/troscoe    1043992576  710082560   333910016  69% /home/troscoe
troscoe@emmentaler1:~$
```

How are all these different file systems tied into the single name space of the kernel's "File System"?

Definition 20.8 (Mount point)**.** *A **mount point** in a hierarchically-named OS (like* UNIX*) is a directory under which is **mounted** a complete other file system. One file system, the **root file system**, sits at the top, and all other file systems accessible in the name space are mounted in directories below this.*

Example 20.9. *In the example above, the root file system is the volume `/dev/sda1`, which is the first partition of the disk `/dev/sda`.*

Remarks:

- A file system can, in principle, be mounted over any existing directory in the name space, regardless of which file system *that* directory is part of.
- When a directory has a file system mounted over it, its contents and that of all its children become inaccessible.
- Having multiple file systems in the same name space is not completely transparent to users. For example, you can't create a hard link from a directory in one file system to a file in another, for reasons that should be clear from the discussion below.
- Mount points allow great flexibility in reconfiguring the name space, and have grown more flexible over the years. It's now possible for unprivileged users in Linux to mount their own file systems in directories in their home directories.

Having multiple types (implementations) of filing systems raises a final question: how are the operations on the file system (open, read, write, etc.) implemented when the sequence of operations is going to be different for different file systems?

Definition 20.10 (Virtual File System)**.** *A **virtual file system** or **VFS** interface is an abstraction layer inside the kernel which allows different file system implementations to coexist in different parts of the name space.*

Remarks:

- Most operating systems have some form of VFS interface. For POSIX-like file systems, they all look pretty much like the original [Kle86].
- Although in C, this is good example of the long-standing use of polymorphism in OS design.
- The details of the VFS interface are instructive: it's not quite the same as the user API to files, but similar.
- VFS is quite flexible: you can attach all kinds of file systems that we don't cover a lot here, such as networked systems (NFS and SMB).

20.2 File system goals

What does the file system need to provide? We've seen the user-visible *functionality* (access control, named data, controlled sharing), but the implementation is also concerned with:

Performance How long does it take to open a file? To read it? To write it? etc. The key challenge here used to be that I/O accesses (to spinning disks) were high-latency, and involved moving disk arms. The result was file systems which tried to lay data out to maximize locality, and to allow common sequences of operations to require a small number of large, sequential I/O operations.

Reliability What happens when a disk fails? Or Flash memory is corrupted? Or the machine crashes in the middle of a file operation? The challenges are the same as you have probably seen with databases: consistency and durability. Approaches are similar: atomic operations, even transactions (though hidden inside the file system implementation), write-ahead logging, and redundant storage of critical data.

20.3 On-disk data structures

For the rest of this section, it's better to describe specific examples of filing systems rather than try to present abstract principles (though a future version of this course might try to do so). We'll look at FAT, BSD FFS, and Windows NTFS.

For each, we'll look at:

- 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?

There is a fourth filing system that's worth looking at, since it adopts a range of different choices to the three we look at here. It's called ZFS, and it's described in the textbook [AD14], in Chapter 13.

20.3.1 The FAT file system

Example 20.11 (FAT)**.** *FAT is the file system created for MS-DOS and the original IBM PC, based on earlier designs that Microsoft had used for other microcomputers.*

Remarks:

- FAT dates back to the 1970s, though even then was a remarkable *un*sophisticated file system. It holds almost no metadata, has no access control, and no support for hard links (i.e. multiple names for the same file).
- Despite this, FAT won't die. It remains the file system of choice for preformated USB drives, cameras, etc.
- Over time, FAT has evolved to handle larger and larger storage devices which stressed hard limits in the original design: thus, we talk about FAT, FAT12, FAT16, FAT32, exFAT, etc. All use the same basic structure we describe here.

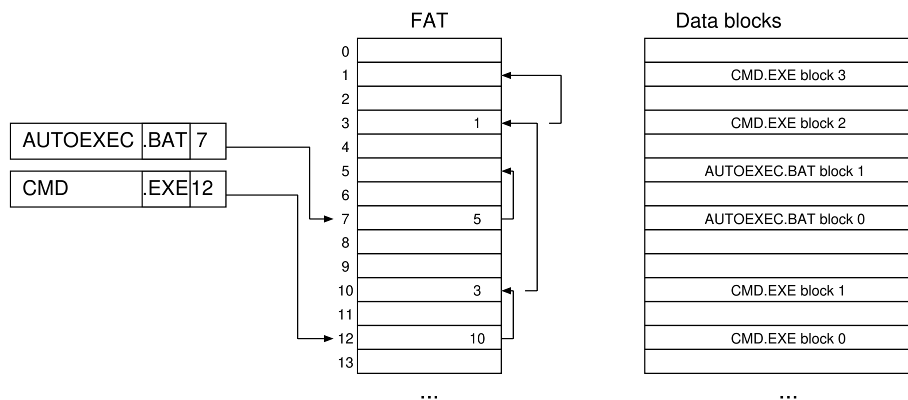


Figure 20.12: The FAT file system

The design of FAT is shown in Figure 20.12.

Definition 20.13 (FAT)**.** *The* ***File Allocation Table****, or FAT, is a linear array of blocks numbers, with an entry for every block on the volume.*

Remarks:

- Directories in FAT are a special kind of file which the OS interprets differently. The directory is a simple table of entries, which give the name, metadata, and an index into the FAT. This is simple, easy to extend (since the file can be extended using the standard file system implementation) but slow to search for large numbers of entries in a directory.
- Each FAT entry marks the corresponding block on the volume as free, or used. If it's used, it's part of a file, and holds the block number (and therefore FAT entry number) of the next block in the file.
- Directories in FAT are special files holding the file name, some metadata, and the block number of the start of the file, i.e. an initial pointer into the File Allocation Table.
- Files are consequently allocated at the granularity of disk blocks.
- Allocating from the free space on the volume is slow: it requires a linear search through the FAT.
- Random access to a file is slow, since it requires traversing the list of file blocks inside the FAT.
- Resilience is poor: if the FAT is corrupted or lost, you really have lost everything.
- The file system has poor locality, since there is nothing to prevent a file's block from being scattered over the disk. Over time, as the file system *ages*, this causes *fragmentation*, which reduces performance.

20.3.2 The Berkeley Fast Filing System

For more information of FFS check out [Mck15] and [MBKQ96].

Example 20.14 (FFS)**.** *FFS was introduced in Berkeley* UNIX *version 4.2 (4.2BSD).*

Definition 20.15 (Inode)**.** *An index node or* ***inode*** *in a file system is a contiguous region of data on a disk that contains all the necessary metadata for file (including where the file's data is stored).*

Remarks:

- The inode is the heart of the file: all names for the file point to the inode, and all data blocks comprising the file are listed by the inode.
- FFS has an area of disk called the *inode array*, which holds a table of complete inodes, indexed by number.
- Directories in FFS are lists of file names and corresponding inode numbers.
- Inodes are quite large, typically 4kB.

The structure of an inode (somewhat simplified) is shown in Figure 20.16.

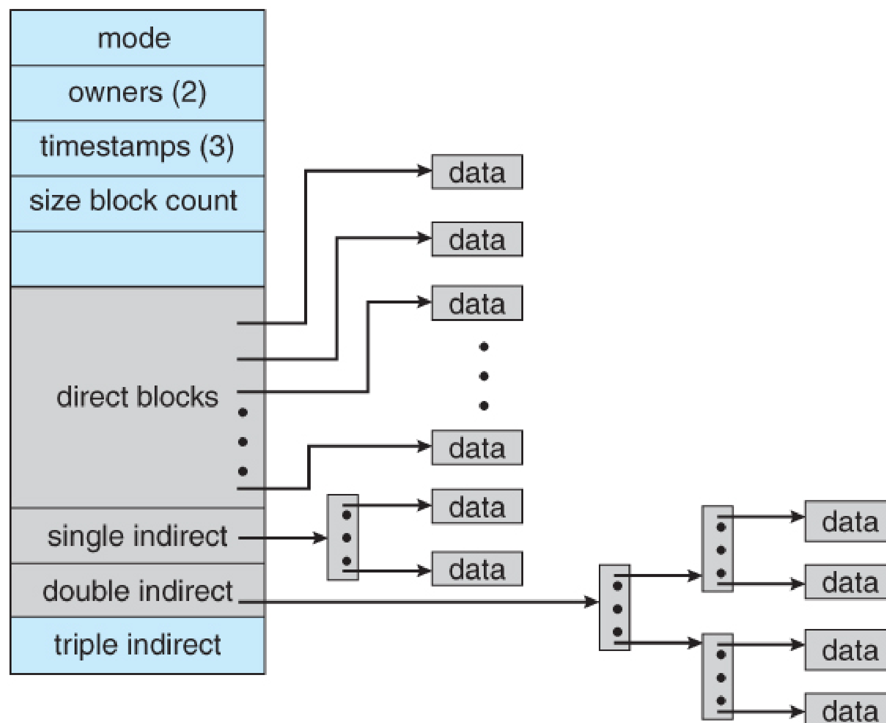


Figure 20.16: Structure of an inode

Remarks:

- The first part of the inode holds standard file metadata; this is typically about 512 bytes, leaving 4096-512 = 3584 bytes left over.
- For small files (¡ 3.5kB), the file's data is stored in the inode, making for very fast access. This is a big win: most files in Unix (particularly in those days) were small, and most accesses were to small files.
- Otherwise, the inode is filled with a list of addresses of the blocks holding the file data. If block addresses are 64 bits in size (8 bytes), there is room for (4,096-512) / 8 = 448 block pointers, so files up to 448 x 4096 = 1792kB in size could be represented with a single inode.
- Any file smaller than this size can be accessed randomly with a single disk access, given the inode.
- To circumvent this limit, the final 3 block addresses (the "indirect pointers" in the inode are special. The first (the "single indirect pointer") points to a block which only contains further block addresses, which more than doubles the maximum representable file size. Any block in the second half of the file can now be accessed with two (cached) disk accesses.
- The second (the "double indirect pointer") points to a block filled with the block addresses of further blocks which contain addresses of data blocks. This means that much larger files can now be accommodated, at the cost of one more disk access to read or write most of the large file.
- The final ("triple indirect") pointer takes this scheme one stage further. This allows truly huge files to be accessed randomly with an upper bound of 4 disk reads per block.

Free space on the volume is managed in FFS by using a bitmap, one per disk block, which can be cached in memory. This is reasonably efficient when searching for free blocks, and is quick to maintain.

Performance on a spinning disk in FFS is enhanced by block groups.

Definition 20.17 (Block group)**.** *A **Block Group** in FFS is a contiguous subset of the disk tracks where related inodes, directories, free space map, and file data blocks are gathered together.*

Remarks:

- Block groups are not perfect, but they aim at increasing the locality of reference for most file system operations. By dividing the disk into regions by track, they try to minimize the movement of the disk arms, which was (at the time) the biggest source of latency for disk I/O operations.

We have not yet specified how the OS figures out where all the block groups, free space maps, etc. are on a disk when the machine starts.

Definition 20.18 (Superblock)**.** *The **superblock** is the area of a disk which holds all information about the overall layout of the file system: how large it is, where the block groups are, etc.*

Remarks:

- Most file systems have some equivalent of the superblock (in FAT, for example it is stored in front of the File Allocation Table itself, in the *master boot record.*
- Losing the superblock is catastrophic. For this reason, it's replicated several times throughout the disk, at regular, predictable intervals. Since it doesn't often change, it's OK to duplicate the data in this way. If the first copy is corrupted, the OS can hopefully recover the file system from another copy.

20.3.3 Windows NTFS

Example 20.19 (NTFS)**.** *The Windows NT file system replaced FAT as the default filing system in Windows NT, and then Windows XP, Windows 2000, and so on (with various enhancements) up to the present day.*

In NTFS, *all file system and file metadata is stored in files.*

Definition 20.20 (Master File Table)**.** *NTFS is organized around a single **Master File Table** of **file entries**, each of which is 1kB in size. Each MFT entry describes a file.*

Remarks:

- The MFT itself is a file, described by the first entry (number 0) in the MFT. Its name is `$MFT`.
- Apart from standard metadata, each MFT entry holds a list of variable-length attributes, which serve a wide range of functions.
- Each possible name for a file is an attribute of its MFT entry.
- For very small files, the data itself is an attribute of its MFT entry.
- File data for non-tiny files is stored in *extents* (see below).
- If the MFT entry is not big enough to hold all the file attributes, the first attribute is an "attribute list" which indexes the rest. More attributes can be held in extra MFT entries for the file, or in extents.
- After file number 0 (the MFT itself), there are 10 more special files in NTFS including a free space bitmap (`$Bitmap`) as in FFS, `$BadClus` (the map of bad sectors to avoid), `$Secure` (the access control database), etc.
- What about the superblock? This is stored in file 3 (`$Volume`), but how is it made resilient? Well, file 1 (`$MFTirr`) actually a contains a copy of the first four entries of the MFT, files 0-3. Note that this includes the backup copy itself.

- The remaining critical file (number 2) is `$Logfile`, a transaction log of changes to the file system.
- The final part of the puzzle (how do you find where `$MFT` starts if you don't know where the MFT is?) is this: the first sector of the volume points to the first block of the MFT file.

Definition 20.21 (Extent). *An **extent** is a contiguous, variable-sized region of a storage volume, identified by its start LBA and length in blocks.*

Remarks:

- NTFS is not the only extent-based file system - the idea is very old. The "`ext`" in Linux `ext2`, `ext3`, and `ext4` file systems refers to extents.
- Extent-based systems are highly efficient for sequential file access (quite a common access pattern) since they keep arbitrarily large sequences of data contiguously on the volume.

20.4 In-memory data structures

When a file or directory is read in from a volume, it is translated into in-kernel data. Directory data, for example, is turned into an index in the kernel from names to (say) inode numbers and volume identifiers.

While the representation of directories is very OS-dependent, open files themselves mostly look the same at a high-level.

Definition 20.22 (Per-process open file table). *The **per-process open file table** is an array mapping file descriptors or file handles to entries in the **system-wide open file table**.*

Definition 20.23 (System open file table). *The **system open file stable** contains an entry for every open file in the OS, including its state: the seek pointer, the layout of the file itself on its volume, and any other state of the open file.*

Remarks:

- There are thus two levels of indirection involved in a file access in user-space: the local file descriptor is mapped to an system open file number, which looked up to give the open file object, which itself points to the file itself.
- Note that, if you're using `stdio` in C (or similar facilities in C++, or Java, or Rust, or...) note that an extra level of indirection happens: a `FILE *` in C itself contains a file descriptor number.
- In an FFS-like file system, the system open file table includes a cache of the inode for each open file.

How can this be made to go fast? Note that all the designs we've seen so far require quite a few disk block accesses to get at a single byte of data in a file. File systems get their performance from caching all this metadata, as well as file data itself.

Definition 20.24 (Block cache)**.** *The file system* ***block cache****, or* ***buffer cache****, is a system-wide cache of blocks read from and written to storage volumes.*

Remarks:

- The block cache can be optimized for particular access patterns: for example, if it expects a file to be accessed sequentially, it can use *read-ahead* to fetch blocks speculatively before they are actually requested by a program, and also *free-behind* to discard them once they have been read if it expects them not to be accessed again.
- Some operating systems, such as Linux, can use any unused physical page in memory as part of its block cache. Others reserve a fixed area of memory for it.

Recall that demand paging regards main memory as a cache for large address spaces on disk: a *paging file* or *paging device* is used to store all data in virtual memory, and this is paged in on demand.

Recall also that *memory mapped files* are given a region of the virtual address space, but are paged in not from the paging device, but from their original file (as are the text segments of running programs).

Both operations require disk I/O, which may or may not go, in turn through the buffer cache. This two-level caching duplicates functionality and, in many cases, data, leading some OS designs to combine the two:

Definition 20.25 (Unified buffer cache)**.** *A* ***unified buffer cache*** *manages most of a computers main memory, and allows cached pages to be accessed via loads and stores from user space (having been mapped into a virtual address space), and/or accessed as file contents using read/write system calls.*

Bibliography

[AD14] Thomas Anderson and Michael Dahlin. *Operating Systems: Principles and Practice.* Recursive Books, 2nd edition, 2014.

[Kle86] S R Kleinman. Vnodes: An architecture for multiple file system types in sun unix. In *Usenix Conference Proceedings, Atlanta, GA, Summer*, pages 238–247, 1986.

[MBKQ96] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4 BSD Operating System.* Addison-Wesley, 1st edition, 1996.

[Mck15] Marshall Kirk Mckusick. A brief history of the bsd fast filesystem. Keynote talk at FAST 2015, 2015. `https://www.youtube.com/watch?v=TMjgShRuYbg`.

Chapter 21

The Network Stack

In comparison with some other parts of OS design, networking has very little (if any) basis in formalism or algorithms – yet (some of us are working on this. . .).

Definition 21.1 (Network stack)**.** *The* ***network stack*** *is the component of the OS which handles all network I/O, including* ***packet transmit and receive****,* ***multiplexing*** *and* ***demultiplexing****, and other kinds of* ***protocol processing****.*

Remarks:

- Not all network protocols are handled by the OS network stack: some (such as HTTP) will be handled as part of the application. The division of responsibility for a complete network communication channel between an application and the OS network stack is somewhat arbitrary, but typically happens above the transport layer (e.g. TCP).
- Within the OS network stack, protocol processing can (and does) occur in many different places:
  - The network hardware, such as the Network Interface Card (NIC).
  - The first-level interrupt handlers for the NIC
  - The rest of the NIC driver bottom-half, e.g the DPCs.
  - The driver top-half, and other kernel code invoked from user space with a system call.
  - Libraries linked with the application

  When a packet is received by a NIC, the data *generally* traverses all these layers in order.
- Similarly, when data is to be sent from a process over the network, all these layers can be involved in reverse.
- In addition to these components, there also
  - System daemons, and
  - Utility programs

  – which are used on longer timescales to maintain the state of the network stack, and form part of the *control plane*.

21.1 Network stack functions

The network stack's purpose is to move data between user programs on different machines. Within a single end-system, getting this done involves converting between the user's data and the set of network protocols being used to communicate it.

This can be broken down into various functions, all of which happen at multiple places in the stack:

Definition 21.2 (Multiplexing)**.** *__Multiplexing__ is the process of sending packets from multiple connections at one layer in the protocol stack down a single connection at a lower later. __Demultiplexing__ is the reverse process: taking packets received on a connection at one layer, and directing each one to the appropriate channel in an upper layer.*

Definition 21.3 (Encapsulation)**.** *While multiplexing refers to sharing a single lower-level channel among multiple higher-level connections, __encapsulation__ is the mechanism by which is this usually achieved: by wrapping a packet on one channel as the payload of a lower-layer connection, adding a __header__ and/or __trailer__ in the process. __De-encapsulation__ is the reverse: interpreting the payload of one packet as a packet for a different, higher-layer protocol.*

Remarks:

- Multiplexing and demultiplexing, and encapsulation and de-encapsulation, are close related but not the same thing. Multiplexing generally requires some kind encapsulation so as to be able to distinguish packets from different connections, but encapsulation may not imply multiplexing.
- Demultiplexing is frequently the most important factor in the performance of a workload over a particular network stack. Fully demultiplexing an Ethernet packet to an application socket may require 10 or 20 different stages of de-encapsulation, if done naively.

Definition 21.4 (Protocol state processing)**.** *The networking stack needs to do more than simply move packets between the network interface an main memory. In addition, it must maintain, and execute, __state machines__ for some (though not all) network protocols. Such __protocol state processing__ not only means that the network stack needs to maintain more state than simply the set of active connections, but it also may need to generate new packets if the state machine so requires.*

Example 21.5. *TCP is a good example of protocol state processing, and in* UNIX *(as in most other OSes) it is performed in the kernel to prevent abuse of the protocol by unprivileged applications.*

TCP state for a given TCP connection in the kernel, generally held in a data structure called the TCP Control Block, is much more than simply the state you are familiar with from the networking course. It includes the congestion and flow control windows, a buffer of all unacknowledged data it might need to resend, plus a collection of __timers__ that trigger the state transitions that are not initiated by packets arriving.

Definition 21.6 (Buffering and data movement)**.** *The network stack also needs to move packet data through the system, both* ***buffering*** *it (holding the data until it is ready to be consumed) and* ***transferring*** *it between different protocol processing elements.*

Remarks:

- Copying packets between every protocol processing stage in the networking stack is clearly inefficient, and so implementations try to avoid copying wherever possible. However, encapsulation and de-encapsulation require care in data representation: adding a header to the front of a packet without copy any data is challenging.

In addition, most network stacks also perform *routing* and *forwarding*, which are discussed below.

21.2 Header space

Definition 21.7 (Header space)**.** *The* ***header space*** *is an abstract vector space which represents the set of all possible headers of a packet.*

Remarks:

- This definition is a bit under-specified: the dimensions of the header space might be each bit in a packet header, or something more abstract (port numbers, source/dest addresses, etc.).
- Even so, any network packet occupies a point in header space.
- Each node in the protocol graph corresponds to a region of header space.
- The set of packets than can be received at a NIC potentially occupies the whole header space, but each demultiplexing step in the protocol stack reduces the sub-volume of the header space that a packet must lie in.
- Each network connection, such as a socket, corresponds to two sub-volumes of header space: which we might call the receive set and the transmit set. Any packet to be transmitted has to lie inside the transmit set when it leaves the NIC, and any packet that is received on the socket must lie in the receive set.
- Packet demultiplexing therefore is the operation of identifying the set of receive sets each received packet lies within, and delivering the packet payload to the socket corresponding to each of these receive sets.

21.3 Protocol graphs

Definition 21.8 (Protocol graph)**.** *The **protocol graph** of a network stack is a directed-graph representation of the forwarding and multiplexing rules at any point in time in the OS. Nodes in the protocol graph represent a protocol acting on a communications channel, and perform encapsulation or decapsulation, and possibly multiplexing or demultiplexing.*

Example 21.9. *Nodes in the protocol graph might include:*

- *Demultiplexing an LLC/SNAP packet based on whether the protocol field is IP, or Appletalk, or ...*
- *Demultiplexing UDP packets based on the destination IP address and port*
- *Demultiplexing TCP packets based on the four-tuple (destination address, destination port, source address, source port).*
- *Processing a single, bidirectional TCP connection*
- *Responding to ICMP echo requests*
- *Encapsulating messages sent via a UDP socket with the correct UDP header*

And so on.

Remarks:

- If a node has multiple outgoing edges, it's probably demultiplexing packets. If it has multiple inbound edges, it's multiplexing.
- People have constructed network stacks explicitly this way, notably the x-kernel [HP88], but it's also an appropriate model of any network stack in the abstract.
- One advantage of the protocol graph over a traditional layered model is that it handles multiple connections at the same layer.
- More importantly, it can be *cyclic*, for example: an IP packet might be received, demultiplexed into a TCP flow, decrypted using TLS, decapsulated via a tunneling protocol like PPTP, and result in a new set of IP packets to be demultiplexed.

21.4 Network I/O

Let's work up the network stack, starting at the hardware.

We've already seen how a NIC operates at the lowest level in Chapter 10: packets that are received are copied into buffers supplied by the OS and enqueued onto a descriptor queue, and these filled buffers are then returned to the OS using the same, or possibly a different, queue.

Similarly, packets to be sent are enqueued on a descriptor queue, and the NIC notified. When a packet has been sent over the network, its buffer is returned

to the OS by the NIC. Synchronization over the queues is handled using device registers, flags in memory, and interrupts.

The first-level interrupt handler for packet receive therefore looks like this (somewhat simplified):

Algorithm 21.10 First-level interrupt handler for receiving packets

1: **inputs**
2: *rxq*: the receive descriptor queue

Device interrupt handler:

3: Acknowledge interrupt
4: **while** not(rxq.empty()) **do**
5: buf ← rxq.dequeue()
6: sk_buf ← sk_buf_allocate(buf)
7: enqueue(sk_buf) for processing
8: post a DPC (software interrupt)
9: **end while**
10: **return**

Remarks:

- The receive queue of the NIC is a way of doing *buffering*: the OS doesn't have to react immediately when a packet arrives.
- The packet is removed from the NIC queue (which is defined by the hardware) and wrapped in what I've called an "`sk_buf`". This is a data structure the OS uses for passing packets (and other data) around internally.

21.5 Data movement inside the network stack

Definition 21.11 (Packet descriptors)**.** ***packet descriptors****, known as* ***sk_bufs*** *in Linux, and* ***m_bufs*** *in BSD* Unix*, are data structures which describe an area of memory holding network packet data.*

Remarks:

- Packet descriptors in modern operating systems (particularly Linux) are really quite complex things (take a look at `/include/linux/skbuff.h`), but what they are doing is conceptually quite simple.
- A packet descriptor holds metadata about a packet, but also a reference to multiple areas of memory that hold the actual packet data.
- A packet descriptor can also identify a subset of a buffer in memory that is of interest.

Example 21.12 (BSD `mbuf` structures)**.** *To simplify somewhat, a* Unix *mbuf contains the following fields:*

```
struct mbuf {
  struct mbuf *m_next;      /* next mbuf in chain */
  struct mbuf *m_nextpkt;   /* next packet in list */
  char        *m_data;      /* location of data */
  int32_t      m_len;       /* amount of data in this mbuf */
  uint32_t     m_type:8,    /* type of data in this mbuf */
               m_flags:24;  /* flags; see below */
  char         m_dat[0];
};
```

How is this used?

A single mbuf `b` *describes* `b.m_len` *bytes of data, starting in memory at address* `b.m_data`*. This data might be stored in the mbuf itself, in the array* `b.m_dat`*, or alternatively somewhere else in memory; the* `b.m_type` *field specifies which case this is.*

An ***mbuf chain*** *represents a single contiguous packet, using non-contiguous areas of memory. An mbuf chain is a singly-linked list of mbufs chained with the* `m_next` *field.*

Moreover, packets themselves can be hooked together in lists or queues using the `m_nextpkt` *field.*

There are a whole host of utility functions in the BSD kernel for creating, filling, manipulating, duplicating, coalescing, freeing, and doing all kinds of other things to mbufs.

Remarks:

- There's a reason for this level of complexity, and it's to do with avoiding copying data, and the need for encapsulation. If a user program wants to send a packet, a packet descriptor is created to wrap the payload the user has supplied. To put a header on the packet, it's easier (in the BSD case above) to add a new mbuf to the front of the list which holds the new header, rather than copying the entire packet to a bigger buffer with room for the new header.
- Similarly, on receive, it's easier to strip the header from a packet when de-encapsulating it by simply bumping the `m_data` field.
- The main goal here is to avoid excessive *copying* of packet data in the kernel – it's expensive and inefficient.

21.6 Protocol state processing

Protocol state processing generally also happens in the "bottom half" of the network stack, even though it is generally independent of the particular NIC device driver. To understand why, consider TCP.

Example 21.13 (TCP protocol state processing)**.** *The code implementing a single TCP connection (managing a single* ***TCP control block****) has to run in response to many different external events: the user sending a segment, or the network receiving data on the connection, but also timers expiring, or acknowledgments received by the network, etc. Many of these events don't involve the user program at all (such as acknowledging data as soon as it is received).*

What's more, these events also require TCP to send packets (such as acknowledgments or retransmissions) that the user will never see. These can be time-critical: TCP estimates the size of its windows based on measuring round-trip time, which includes the time taken for TCP code to execute when a packet is received.

For this reason, if TCP is to run in the "top half" (i.e in code executed by the user program, or in the kernel but invoked by the user program via a syscall), it has to be scheduled to run at all kinds of events ***not*** *scheduled by the user program.* UNIX*-like operating systems avoid this problem by running most of TCP in the bottom half, as a set of DPCs.*

21.7 Top-half handling

Moving up the stack further, we come to the top half: that invoked by user programs. You're already familiar with the common "sockets" interface: `bind()`, `listen()`, `accept()`, `connect()`, `send()`, `recv()`, etc.

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 (if the user requests it) between queues of protocol descriptors in the kernel (attached to sockets) and user-space buffers.

21.8 Performance issues

So far, so good. But much time does the OS have to process a packet with a modern network?

Example 21.14 (10 Gb/s Ethernet performance)**.** *At full line rate for a single 10 Gb/s port, we receive about 1GB (gigabyte) per second of packet data. This corresponds to about 700,000 full-size Ethernet frames per second, rather more if the frames are smaller.*

At 2GHz, this means the OS has to process a packet in under 3000 cycles. This includes IP and TCP checksums, TCP window calculations and flow control, and copying the packet to user space. Note that this is a 1500-byte packet, and we can copy about 4 bytes per cycle through CPU registers. That's 400 cycles gone right there, and we need some left over for the user program to run.

It gets worse. An L3 cache miss (64-byte lines) is about 200 cycles, which means we can afford at most 10 or so cache misses per packet. However, on most machines, DMA transfers from the NIC mean that the processor cache is cold for the packet.

Furthermore, interrupt latency in a typical PC is 500 cycles, so we're going to have trouble reacting fast enough if we take an interrupt on each packet.

This is without considering the cost in latency for kernel entry and exit, hardware register access (often hundreds of cycles), context switch overhead, the cost of enqueueing and dispatching a DPC, etc.

We also have to send packets as well as receive them.

This example is a single-port 10Gb/s Ethernet card, but vendors are selling dual-port cards that run at 200Gb/s today.

Clearly, the network stack design we have described so far will be unable to handle this traffic.

Definition 21.15 (Polling)**.** *Instead of the conventional interrupt-driven descriptor queues, a network receive queue can serviced by a processor* ***polling*** *it continuously.*

Remarks:

- Polling eliminates the overhead of interrupts, context switches, and even kernel entry and exit if there is a way to access the queues from user space.
- This is an old idea, but it has recently come into vogue through schemes like NetMap [Riz12] and Intel's Data Plane Development Kit [dpd18]. There is now a direct polling interface to the network in Linux.
- This, of course, requires a dedicated processor core to spin forever waiting for network packets. At low load, it is possible to transition back to the interrupt-driven model, and resort to polling at high-load – for example, microkernel-based drivers have done this for some time.
- Even polling, however, is insufficient to handle modern high-speed networks. For one thing, it is not clear how to scale this to multiple cores. If the network stack for a given NIC has to go through a single core, Amdahl's Law will fundamentally limit the performance of networked programs.

21.9 Network hardware acceleration

The solution is to put more functionality into hardware. As with much of networking, there are few agreed-upon terms here but a huge variety of hardware technology features, so treat this list as a broad overview.

Definition 21.16 (Multiple queues)**.** *Modern NICs support* ***multiple send and receive queues*** *per port. Received packets are demultiplexed in hardware based on a set of* ***flow tables*** *which determine which descriptor queue to put each packet onto. Similarly, multiple transmit queues are multiplexed onto the network physical port.*

Remarks:

- The number of queues supported these days ranges from 2 (on cheap cards), to 64 (on fairly cheap cards), to several thousand (on fancy hardware).
- There are plenty of criteria for demultiplexing flows. A typical table maps IP 5-tuples in the packet (sometimes with "wildcard" entries) to the queues. Other pattern matching is possible, for example the Intel 82599 has a "SYN filter", which can redirect TCP connection setup packets to a different queue.
- On transmit, the card typically has a configurable *scheduling policy* for picking which non-empty transmit queue to pick a packet from next.

Definition 21.17 (Flow steering)**.** *Sending received packets to the right receive queue is only part of the solution.* ***Flow steering*** *not only picks a receive queue based on the network flow (e.g. TCP connection) that the packet is part of, but can send an interrupt to a specific core that is waiting for that packet.*

Remarks:

- Given the overhead of moving packet data from one core's cache to another, it's quite important in many cases that the first core to find out about a new packet is the one running the thread that is waiting to receive it.

Definition 21.18 (Receive-side scaling)**.** ***Receive-side scaling*** *(RSS) uses a deterministic hash function on the packet header to balance flows across receive queues and interrupts.*

Remarks:

- Flow steering specifies for each flow (up to the size of the hardware table. . .) where to send it. RSS doesn't need a table, but attempts to balance lots of flows across lots of cores.

- The assumption here is that one core is just as good as any other at handling a flow, and as long as it's the same core for all packets in a flow, cache lines will migrate to that core.

- RSS allows the network stack performance to scale with the number of cores (assuming it is written correctly . . .) by removing the multiplexing function – *which is a serialization point* – from software.

Definition 21.19 (TCP Chimney Offload)**.** ***TCP chimney offload****, sometimes called* ***partial TCP offload****, allows the entire state machine for a TCP connection – once it has been established – to be pushed down the hardware, which will then handle timers, acknowledgments, retries, etc. and simply deliver in-order TCP segments to the kernel.*

Remarks:

- Chimney offload is a limited case of a full TCP Offload Engine (TOE), which moves even more of the TCP stack onto the NIC.

Definition 21.20 (Remote DMA)**.** ***Remote Direct Memory Access*** *or RDMA is a completely different set of network protocols and hardware implementations. RDMA supports Ethernet-style descriptor rings for messages, but also supports (hence the name) so-called* ***one-sided operations*** *which allow main memory on a machine to be written and read directly over the network without involving the host CPU: the NIC receives packets requesting such an operation, executes it itself, and returns the results.*

Remarks:

- RDMA is a big, and controversial, topic in itself which we do not talk much about here. It comes from the High-Performance Computing community, and hence cares much less about security, resource sharing, robustness, integration with existing systems, and performance for irregular workloads than other networking technologies.
- Nevertheless, it is gaining some traction in rack-scale appliances and some datacenter-scale applications.
- Most of the complexity of RDMA in practice, and the reason that it is harder than one might expect to exploit one-sided operations for performance, is the overhead of setting up the permissions in a distributed system to allow one machine to safely access another's memory.
- Stepping back, an alternative way to view RDMA (and some other of the more complex network acceleration hardware features) is seeing the NIC as another, rather limited, processor on the host which has its own network connections, and can execute very limited forms of user code (e.g. copy, and atomic memory operations).

21.10 Routing and Forwarding

Typically, the network stack in an OS not only sends and receives packets, but also forwards them between its network interfaces, much as a router or switch does.

Definition 21.21 (Forwarding)**.** *Packet **forwarding** is the process of deciding, based on a packet and the interface on which the packet was received, which interface to send the packet out on.*

Definition 21.22 (Routing)**.** *Packet **routing** is the process of calculating rules to determine how all possible packets are to be forwarded.*

Remarks:

- Forwarding needs to be fast, since any latency adds to the end-to-end delay of the packet concerned. Consequently, it's done at the lower levels of the stack, as close to the hardware as possible (unless the hardware itself performs forwarding).
- As in a hardware-based router, forwarding is performed according to a set of tables called the FIB

Definition 21.23 (Forwarding information base)**.** *The **forwarding information base** or FIB in a router is the set of data structures which are traversed for each packet received to determine what actions to perform on it, such as sending it out on a port or putting it into a memory buffer.*

Remarks:

- The FIB is the result of the *routing* calculation. Routing may be a purely local computation, but might involved a *routing protocol* which itself needs to send and receive network messages to exchange information with other routers.
- For this reason, routing in an OS stack usually happens in user space, either in a routing daemon (sometimes called `routed`) or by manual configuration utilities.

Example 21.24. *Routing in Linux can be done manually using the* `ip route` *command - type* `ip route show` *to show the machine's routing table. If you want to run a routing daemon to talk BGP, OSPF, or some other routing protocol, packages like* `quagga` *supply suitable daemons.*

Remarks:

- Forwarding a packet found on a receive queue essentially involves reading its header to *classify* it, then using this information to transfer the packet to one or more transmit queues to be sent.
- This is essentially the same operation as *demultiplexing* a received packet to an appropriate application. For this reason, forwarding code is generally the same as the lower-level protocol demultiplexing code.

Bibliography

[dpd18] Data plane development kit. `https://www.dpdk.org/`, November 2018.

[HP88] N. Hutchinson and L. Peterson. Design of the x-kernel. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 65–75, New York, NY, USA, 1988. ACM.

[Riz12] Luigi Rizzo. Revisiting network i/o apis: The netmap framework. *Commun. ACM*, 55(3):45–51, March 2012.

Chapter 22

Virtualization

We've seen lots of examples of virtualization.

This is another: a virtual machine monitor. A VMM virtualizes an entire hardware machine.

Remarks:

- We can contrast this OS processes and address spaces. In a sense, processes and address spaces present user programs with a virtual machine, but one with a rather different execution environment to raw hardware: system calls instead of hardware devices, signals instead of interrupts, etc.
- The term "virtual machine" is also used to describe highly abstract execution environments, such as the byte code executed by the Java Virtual Machine. In practice, there is a continuum: what these definitions all have in common is that they are *interpreters*.
- The virtual machines we deal with here are at one end of this spectrum: the execution environment they provide is a simulation of raw machine hardware. This means that, in most cases, the applications than run in this environment are, themselves, operating systems.

Definition 22.1 (Guest operating system)**.** *A* ***guest operating system*** *is an OS, plus associated applications, etc. which is running inside a virtual machine.*

Definition 22.2 (Hypervisor)**.** *Many people draw a distinction between a VMM, and a* ***hypervisor****. A VMM is the functionality required to create the illusion of real hardware for a single guest OS – that is, it creates a single virtual machine. A hypervisor is the software than runs on real, physical hardware and supports multiple virtual machines (each with its associated virtual machine monitor).*

Remarks:

- In this distinction (which we'll use), the hypervisor is, for all intents and purposes, an operating system itself.
- In fact, an OS can be extended to act as a hypervisor.

Definition 22.3 (Type 1 and type 2 hypervisors)**.** *A **type 1 hypervisor** runs "on the metal", that is to say, it functions as an OS kernel. In contrast, a **type 2 hypervisor** runs on top of, or as a part of, a conventional OS like Linux or Windows.*

Remarks:

- IBM VM/CMS, VMware ESX, and Xen are all type 1 hypervisors.
- VMware Workstation, kvm, and VirtualBox are all type 2 hypervisors.

These days, there is considerable interest in *containers*, which share some characteristics with virtual machines, but are actually an example of something different, sometimes known as *OS-level virtualization.*

Definition 22.4 (OS-level virtualization)**.** ***Operating system-level virtualization*** *uses a single OS to provide the illusion of multiple instances or **containers** of that OS. Code running in a container have the same system call interface as the underlying OS, but cannot access any devices.*

Remarks:

- System-level virtualization 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 addition, the OS may provide more sophisticated scheduling and memory allocation policies to allocate to containers rather than simply processes.
- Containers are somewhat limited in functionality compared with virtual machines, but are more efficient (at least at present; this claim is disputed by some researchers who have produced extremely efficient hypervisors). We don't cover them further in this course.

22.1 The uses of virtual machines

Virtual machines are an old idea, dating back to IBM's VM/CMS for System/370 in the 1960s. They were revived about 15 years ago, by systems (and companies) like VMware, Xen, Hyper-V, kvm, etc. Why would you want one?

Definition 22.5 (Server Consolidation)**.** *The industry (marketing) term **server consolidation** refers to taking a set of services, each running on a dedicated server, and **consolidating** them onto a single (probably larger) physical machine so that each one runs in a virtual machine.*

Remarks:

- You might ask why these were all running on dedicated machines in the first place. It's typically for reasons of security and performance isolation, and sometimes because the applications can only run one-per-machine (for a variety of reasons), or can only run on an old hardware configuration.
- Server consolidation is not the first use for VMMs that many people think about, but it was this that really drive the market in the early 2000's.

Definition 22.6 (Backward compatibility)**.** ***Backward compatibility*** *is the ability of a new machine to run programs (including operating systems) written for an old machine.*

Example 22.7. *Programs written for Windows XP or earlier versions of Windows can still run on Windows 10, but frequently this is achieved by Windows 10 creating a new virtual machine running XP behind the scenes to run the application.*

Definition 22.8 (Cloud computing)**.** ***Cloud computing*** *is, broadly speaking, the business of renting computing resources as a utility to paying customers, rather than selling hardware.*

Remarks:

- Hypervisors decouple allocation of resources (VMs), from provisioning of infrastructure (physical machines)

Example 22.9. *Amazon Web Services (AWS) is, probably, the world's largest cloud computing business, and it is primarily based on renting computing resources in the form of virtual machines running customers' code.*

Definition 22.10 (Resource isolation)**.** *When multiple applications contend for resources (CPU time, physical memory, etc.), the performance of one or more may degrade in ways outside the control of the OS.* ***Resource isolation*** *is the property of an OS guaranteeing to one application that its performance will not be impacted by others.*

Remarks:

- One might think that resource isolation would be a fundamental function of any OS. Performance isolation can be critical in many enterprises (e.g. in cloud computing where real money is at stake).
- In some ways, this is true, but OS mechanism miss the target: Unix, for example, can sometimes perform resource isolation between processes, but the problem is that an "application" is not a process: many applications are made up of multiple processes, and many processes are servers shared by multiple applications.
- This problem has been recognized for some time (one comment is that Unix lacks *resource containers* [BDM99]; however, it was virtual machine monitors that first provided this (in the form of virtual machines).

The above uses are the most common, and the most commercially important, cases where virtual machines are used. There are many more, it turns out:

- OS development and testing
- Recording and replaying the entire state of a machine, for debugging, auditing, etc.
- Sandboxing for security
- Lock-step replication of arbitrary code
- Speculative execution and rollback.

– and others.

22.2 Virtualizing the CPU

How can we build an efficient virtual machine monitor? Let's go through all the resources that need to be virtualized, starting with the CPU.

In a sense, threads or processes virtualize the processor, but only in "user mode". To run an OS inside a VM, we need completely virtualize the processor *including kernel mode.*

What happens when the processor executing code in a virtual machine executes a privileged instruction? Obviously it can't execute it "for real", but it has to do something.

By default, if the processor tries to execute a privileged operation in user space, the result is a trap or fault. We can use this to catch the attempt to do something privileged and simulate its effect.

Definition 22.11 (Trap-and-emulate)**.** ***Trap-and-emulate*** *is a technique for virtualization which runs privileged code (such as the guest OS kernel) 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.*

Remarks:

- This might be enough to fully virtualize the CPU over a conventional OS (i.e. the host OS is functioning as the hypervisor): we can run the guest kernel in a regular process in user mode, and use trap-and-emulate to allow the host OS to emulate the effects of privileged operations in the VMM.
- This *is* enough, if the instruction set is *strictly virtualizable.*

Definition 22.12 (Strict virtualizability)**.** *An instruction set architecture (ISA) is* ***strictly virtualizable*** *iff it can be perfectly emulated over itself, with all non-privileged instructions executed natively, and all privileged instructions emulated via traps*

Remarks:

- IBM S/390, DEC Alpha, and IBM PowerPC are all strictly virtualizable ISAs.
- Basic Intel x86 and ARM are not.

What goes wrong? Instructions which do the same thing in kernel and user space will work, and instructions which can only be executed in kernel mode will trap and be emulated. The problem is instructions which work in both user space and kernel mode, but do something *different* depending on the mode.

Example 22.13. *The `PUSHF` and `POPF` instructions are among 20 or so in the x86 ISA that cannot be virtualized. The push and pop the condition code register, which includes the Includes interrupt enable flag (`IF`). In kernel mode, this really can enable and disable interrupts, but not in user space. In this case, the VMM can't determine if Guest OS wants interrupts disabled. We can't cause a trap on a (privileged) `POPF`.*

What can we do? There are several solutions, including:

1. Full software emulation
2. Paravirtualization
3. Binary rewriting
4. Change the hardware architecture

Definition 22.14 (Software emulation)**.** *A **software emulator** creates a virtual machine by interpreting all kernel-mode code in software.*

Remarks:

- This is, unsurprisingly, very slow - particularly for I/O intensive workloads.
- It is used by, e.g. SoftPC, DosBox, MAME, and other emulators.

Definition 22.15 (Paravirtualization)**.** *. A **paravitualized** guest OS is one which has been specially modified to run inside a virtual machine. Critical calls are replaced with explicit trap instruction to VMM.*

Remarks:

- This was used for the first version of Xen, for example [**?**]: the authors modified any problematic bits of Linux to explicitly talk to Xen.
- It prevents you from running arbitrary OS binaries in your VMM, because the VMM simply can't copy with, e.g., non-virtualizable instructions.
- However, it's fast: almost all commercial VMMs use paravirtualizing to punch holes in the strict VM/VMM interface for performance reasons, usually in special kernel modules loaded at boot time, or custom device drivers.

Definition 22.16 (Hypercall)**.** *A **hypercall** is the virtual machine equivalent of a system call: it explicitly causes the VM to trap into the hypervisor. Paravirtualized VMs use this to ask the hypervisor to do something for them; see below.*

Definition 22.17 (Binary rewriting)**.** *Virtualization using **binary rewriting** scans kernel code for unvirtualizable instructions, and rewrites them – essentially patching the kernel on the fly.*

Remarks:

- This is generally done on demand: all guest kernel instruction pages are first protected (no-execute). When the VMM takes a trap on the first instruction fetch to the page, it is scanned for all possible instructions that might need rewriting. After patching the code, the page is marked executable and the faulting instruction restarted in the Guest OS.
- VMware uses this approach for x86 virtualization [BDR$^+$12] quite effectively.

Definition 22.18 (Virtualization extensions)**.** *An instruction set architecture which cannot be strictly virtualized can be converted into one that is by adding **virtualization extensions**. This typically takes the form of a new processor mode.*

Remarks:

- Both x86 and ARM processor architectures now have additional processor modes which change the behavior of non-virtualizable instructions so that they all trap in guest kernel mode.
- x86 actually has two different ways of doing this, depending on whether you are on an Intel (VT-x) or AMD (AMD-V) processor.
- Hardware support for virtualization often goes beyond merely making the instruction set virtualizable, as we show see below.

22.3 Virtualizing the MMU

So much for the processor, what about the MMU?

The guest OS kernel is going to create page tables and install them in the MMU. How do we virtualize this, that is to say, how does the VMM let the guest OS do this and create a result which is, from the point of view of the guest kernel, correct, given that we only have one MMU per core? First, we need some (revised) definitions of addresses.

Definition 22.19 (Virtual address (virtualized))**.** *We define **virtual address** now to mean an address in a virtual address space created by the Guest OS.*

Definition 22.20 (Physical address (virtualized))**.** *We define **physical address** to mean an address that the guest OS thinks is a physical address. In practice, this is likely to be in virtual memory as seen by the VMM. We let the guest OS create arbitrary mappings between virtual and physical addresses inside its virtual machine.*

Definition 22.21 (Machine address)**.** *We define a* ***machine address*** *to be a "real" physical address, that is, a physical address as seen by the hypervisor. Guest physical addresses are translated into machine addresses, but the guest OS is typically unaware of this extra layer of translation.*

What's happening under the cover is that the hypervisor is allocating machine memory to the VM, and somehow ensuring that the MMU translates a guest virtual address not to a guest physical address but instead to a machine address. The efficiency of this is critical for VM performance.

There are basically three ways to achieve this:

1. Direct writable page tables
2. Shadow page tables
3. Hardware-assisted nested paging

Definition 22.22 (Directly writeable page tables)**.** *In the first approach, the guest OS creates the page tables that the hardware uses to directly translate guest virtual to machine addresses.*

Remarks:

- Clearly, this requires paravirtualization: the guest must be modified to do this.
- The VM has to enforce two conditions on each update to a PTE:
  1. The guest may only map pages that it owns
  2. Page table pages may only be mapped RO
- The VMM needs to validate all updates to page tables, to ensure that the guest is not trying to "escape" its VM by installing a rogue mapping.
- In fact, we need more than that: the VMM needs to check *all* writes to *any* PTE in the system.
- In practice, all page table pages are marked read-only to the guest kernel, and a hypercall is used to modify them.
- We also need to watch for race conditions: the VMM may have to "unhook" whole chunks of the page table in order to be able to apply a set of updates.
- A further hypercall is needed to change the page table base.
- This is naturally expensive (hypercalls can be slow), but many updates to the page tables can be batched up and requested in a single hypercall, which helps somewhat.

Definition 22.23 (Shadow mage tables)**.** *A* ***shadow page table*** *is a page table maintained by the hypervisor which contains the result of translating virtual addresses through first the guest OS's page tables, and then the VMM's physical-to-machine page table.*

Remarks:

- With shadow page tables, the guest OS sets up its own page tables but these are never used by the hardware.
- Instead, the VMM maintains shadow page tables which map directly from guest VAs to machine addresses.
- A new shadow page table is installed whenever the guest OS attempts to reload the Page Table Base Register (which does cause a trap to the VMM).
- The VMM must keep the shadow table consistent with both the guest's page tables and the hypervisors own physical-to-machine table. It does this by write-protecting all the guest OS page tables, and trapping writes to them. When this happens, it applies the update to the shadow table as well.
- As with direct page tables, this can incur significant overhead, but many clever optimizations can be applied [BDF+03].

Definition 22.24 (Nested paging)**.** ***Nested paging****, also known as* ***second level page translation*** *or (on Intel processors)* ***extended page tables****, 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 end result (guest-virtual to machine) in the TLB.*

Remarks:

- Most modern processors which support virtualization offer nested paging. It can be significantly faster than shadow page tables (but was not always such).
- Nested paging reduces TLB coverage, the TLB tends to hold both guest-virtual to machine and host-virtual to machine translations (in particular, the ones needed for the guest OS page tables).
- TLB miss costs are correspondingly higher, and a TLB fill (and table talk) can itself miss in the TLB.
- Prior to adding nested paging to x86 processors, neither AMD nor Intel provided tagged TLBs (with address space identifiers). However, the context switch overhead became so high (since the TLB had to be completely flushed even on an intra-guest context switch) that they finally introduced TLB tags to the architecture.
- Whatever its performance trade offs, nested paging gives hardware designers a performance target, to is likely to improve in the future.
- Nested paging is also much, much easier to write the VMM for. It is the main reason that *kvm* is so small.

22.4 Virtualizing physical memory

That takes care of the page tables and MMU, but what about allocating memory to a virtual machine? A VM guest OS is, typically, expecting a fixed area of physical memory (since that's what a real computer looks like). It is certainly not expecting it's allocation of "physical" memory to change dynamically.

In practice, of course, the VM's "physical" memory is not actually real, but "virtual" memory as seen by the underlying VMM. Not only that, as we saw with virtual memory and processes, the amount of physical memory allocated to a VM should be able to change over time. This leads to two problems:

1. How can the hypervisor "overcommit" RAM, as an OS does with regular processes, and obtain the same dramatic increase in efficiency as a result?
2. How can the hypervisor reallocate (machine) memory between VMs without them crashing?

In theory, this is just demand paging: if the hypervisor demand pages guest-physical memory to disk, it can reallocate machine memory between VMs exactly how an OS reallocates physical memory among processes. The problem is:

Definition 22.25 (Double Paging)**.** ***Double paging*** *is the following sequence of events:*

1. *The hypervisor pages out a guest physical 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, which pages P back into memory*
4. *The page is immediate written out to disk and discarded by the guest OS.*

Again, this might be fixable using paravirtualization to move paging code out of the guest and into the hypervisor, at the cost of considerable complexity in both the (modified) guest OS and hypervisor. A more elegant solution was created by VMware: ballooning [Wal02].

Definition 22.26 (Memory ballooning)**.** ***Memory ballooning*** *is a technique to allow hypervisors to reallocate machine memory between VMs without incurring the overhead of double paging. A loadable device driver (the* ***balloon driver*** *is installed in the guest OS kernel. This driver is "VM-aware": it can make hypercalls, and also receive messages from the underlying VMM.*

Ballooning allows memory to be reclaimed from a guest OS thus (this is called "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 uses the guest OS memory allocator to allocate n pages of kernel memory for its own private use.

3. It then communicates the guest-physical addresses of these frames to the VMM using a hypercall.

4. The VMM then unmaps these pages from the guest OS kernel, and reallocates them elsewhere.

Deflating the balloon, i.e. reallocating machine memory back to a VM, is similar:

1. The VMM maps the newly-allocate machine pages into guest-physical pages inside the balloon, i.e. page numbers previous handed by the balloon driver to the VMM.

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, which can now use them for any purpose.

22.5 Virtualizing devices

How do we virtualize devices? That is to say, how do we give each guest OS a set of devices to access?

Recall that, to software, a device is something that the kernel (or the driver at least) communicates with using:

- Memory-mapped I/O register access from the CPU
- Interrupts from the device to the CPU
- DMA access by the device to and from main memory

Definition 22.27 (Device model). *A **device model** is a software model of a device that can be used to emulate a hardware device inside a virtual machine, using trap-and-emulate to catch CPU writes to device registers.*

Remarks:

- Device models emulate real, commonly-found hardware devices, so that the guest OS is already likely to have a driver for.
- “Interrupts” from the emulated device are simulated using upcalls from the hypervisor into the guest OS kernel at its interrupt vector.
- There’s a minimal number of device models that a virtual machine has to support, and it’s not trivial: you need device models for all the interrupt controllers, memory controllers, PCIe bridges, basic console, in short, everything the OS needs in order to minimally boot.
- The best hardware devices to emulate in software are not always the ones you would want “in real life”: a highly sophisticated network card, for instance, would be difficult and slow to emulate, while a very basic card (such as the notorious RTL8139, or the much-loved DECchip 21140 Tulip, emulated in Microsoft Hyper-V) can transfer data to and from the virtual machine more efficiently.

The next logical step after emulated "real" devices, is to create fictitious devices which are only designed to be emulated in a VMM, and write optimized drivers for them for the most popular OSes.

Definition 22.28 (Paravirtualized devices)**.** *A **paravirtualized device** is a hardware device design which only exists as an emulated piece of hardware. The driver of the device in the guest OS is aware that it's running in a virtual machine, and can communicate efficiently with the hypervisor using shared memory buffers and hypercalls instead of trap-and-emulate.*

Remarks:

- When using a desktop hypervisor (such as VMware workstation or VirtualBox), it's common to install the guest OS (particularly if it's an old one like Windows XP), and then subsequently install a set of "tools" which make its performance improve dramatically. Most of these "tools" are actually paravirtualized drivers.
- It sounds like a good idea to standardize the interface to paravirtualized drivers to that they can be supported by as many hypervisors and guest OSes as possible. It's such a good idea that almost every hypervisor has standardized them, unfortunately in different ways. The Linux/kvm way of doing this is called `virtio`.

What about the real device drivers, that talk to the real devices?

One option is to put them in the hypervisor kernel (as in, say, VMware ESX). This is fast, but requires the hypervisor itself to natively provide drivers for any device that it supports (much like any other OS).

Alternatively, they can be in a virtual machine, using "device passthrough."

Definition 22.29 (Device passthrough)**.** ***Device passthrough*** *maps a real hardware device into the physical address space of a guest OS, allowing it exclusive access to the hardware device as if it were running on real hardware.*

Remarks:

- Device passthrough is rather more difficult than simply mapping the memory-mapped I/O regions of the device into the virtual machine's physical address space. For one thing, you need to make sure that the physical addresses the driver hands to its DMA engine are translated into machine addresses before it actually transfers data.
- If you know something about PCI configuration, you'll also have realized that you need to emulate a whole PCI or PCIe tree just to have a single device direct-mapped into it.
- Nevertheless, it can be done (in the past, three ETH students did this for Barrelfish's VMM as a semester lab project!).
- Modern hardware has made this a lot easier: IOMMUs (mostly) solve the memory translation problem, and processor support for virtual machines now includes the ability to deliver selected hardware interrupts directly to the virtual machine.

- Device passthrough doesn't solve the problem of how to share a real device among multiple virtualized guest OSes – all it does is allow a *single* guest to use the device directly.

Definition 22.30 (Driver domain)**.** *A **driver domain** is a virtual machine whose purpose is not to run user applications, but instead to provide drivers for devices mapped into its physical address space using device passthrough.*

Remarks:

- The solution to device sharing used by, say, Xen, uses driver domains. A driver domain runs some version of a popular, well-supported kernel (such as Linux), which has drivers for all the devices you might need to support on the real machine.

- The driver domain then runs user-space code which talks to the kernel device drivers, and re-exports a different interface to these devices using inter-VM communication channels.

- For example, the driver domain might export a physical disk by running a file server which other virtual machines might access over "the network", or alternatively it could export a block interface to virtual disk volumes over another channel. In the client virtual machines, this channel appears as a paravirtualized disk driver.

- Driver domains are great for compatibility, but they are slow for the same reason that poorly-implemented microkernels can be slow: communication between a guest OS kernel and the physical device involves a lot of boundary crossings: from the guest kernel to the hypervisor, into the driver domain kernel, up into user space in the driver domain, and all the way back again.

The commercial importance of virtual machines is demonstrated by the fact that, fairly early on, hardware vendors started working on how to make devices which solved these problems on their own.

Definition 22.31 (Self-virtualizing devices)**.** *A **self-virtualizing device** is a hardware device which is designed to be shared between different virtual machines on the same physical machine by having different parts of the device mapped into each virtual machine's physical address space*

The most common form of self-virtualizing devices today is SR-IOV.

Definition 22.32 (Single-Root I/O Virtualization)**.** ***Single-Root I/O Virtualization** (SR-IOV) is a extension to the PCI Express standard which is designed to give virtual machines fast, direct, but safe access to real hardware.*

*An SR-IOV-capable device appears initially as a single PCI device (or "function", in PCI jargon), known as the **physical function** or PF. This device, however, can be configured to make further emphvirtual functions (VFs) to appear in the PCI device space: each of this is restricted version of the PF, but otherwise looks like a complete different, new device.*

Remarks:

- The way this works is that you run a driver for the PF in a driver domain. When a new VM is created, the system asks the driver domain to create a new VF for the VM, and this is then direct-mapped in the new guest's physical address space.
- You can have quite a few of these. It's not unusual for a SR-IOV device to support to 4096 VF on a single PF (4096 is the architectural limit). That's a lot of ethernet cards.

22.6 Virtualizing the network

Networking is a particularly interesting case for virtualization, as it is often the main interface between the virtual machine and the "real" world.

Definition 22.33 (Soft switch)**.** *A **soft switch** is a network switch implemented inside a hypervisor, which switches network packets sent from paravirtualized network interfaces in virtual machines to other VMs and/or one or more physical network interfaces.*

Remarks:

- In effect, a soft switch extends the network deep into the edge system.
- The soft switch can be quite powerful (including switching on IP headers, etc.) but it needs to be fast: implementations like OpenVSwitch are highly optimized at parsing packet headers.
- There are many different ways of addressing network interfaces inside virtual machines. The most common is to give each virtual network interface a new MAC address, and let DHCP do the rest. However, it's also common for packets to and from a particular VM to go through a GRE tunnel, so that if the virtual machine is *migrated* between physical machines the network traffic can follow it.
- With SR-IOV-capable network interfaces, this gets a bit complicated. Many SR-IOV NICs actually have their own hardware switch on board, which switches packets between all the VFs, the PF, and the physical network ports.

Bibliography

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In

Proceedings of the Third Symposium on Operating Systems Design and Implementation, OSDI '99, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.

[BDR+12] Edouard Bugnion, Scott Devine, Mendel Rosenblum, Jeremy Sugerman, and Edward Y. Wang. Bringing virtualization to the x86 architecture with the original vmware workstation. *ACM Trans. Comput. Syst.*, 30(4):12:1–12:51, November 2012.

[Wal02] Carl A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, December 2002.

Chapter 23

Game Theory

"Game theory is a sort of umbrella or 'unified field' theory for the rational side of social science, where 'social' is interpreted broadly, to include human as well as non-human players (computers, animals, plants)."

– Robert Aumann, 1987

23.1 Introduction

In this chapter we look at a distributed system from a different perspective. Nodes no longer have a common goal, but are *selfish*. The nodes are not byzantine (actively malicious), instead they try to benefit from a distributed system – possibly without contributing.

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

Remarks:

- Examples of potentially selfish behavior are file sharing or TCP. If a packet is dropped, then most TCP implementations interpret this as a congested network and alleviate the problem by reducing the speed at which packets are sent. What if a selfish TCP implementation will not reduce its speed, but instead transmit each packet twice?
- We start with one of the most famous games to introduce some definitions and concepts of game theory.

23.2 Prisoner's Dilemma

A team of two prisoners (players u and v) are being 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 / v		Player u	
		Cooperate	Defect
Player v	Cooperate	1 1	0 3
	Defect	3 0	2 2

Table 23.1: The prisoner's dilemma game as a matrix.

- If both of them stay silent (*cooperate*), both will be sentenced to one year of prison on a lesser charge.
- If both of them testify against their fellow prisoner (*defect*), the police has a stronger case and they will be sentenced to two years each.
- If player u defects and the player v cooperates, then player u will go free (snitching pays off) and player v will have to go to jail for three years; and vice versa.
- This two player game can be represented as a matrix, see Table 23.1.

Definition 23.2 (game)**.** *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.*

Definition 23.3 (social optimum)**.** *A strategy profile is called social optimum (SO) if and only if it minimizes the sum of all costs (or maximizes payoff).*

Remarks:

- The social optimum for the prisoner's dilemma is when both players cooperate – the corresponding cost sum is 2.

Definition 23.4 (dominant)**.** *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.*

Remarks:

- The dominant strategy profile in the prisoner's dilemma is when both players defect – the corresponding cost sum is 4.

Definition 23.5 (Nash Equilibrium)**.** *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.*

Remarks:

- A game can have multiple Nash Equilibria.
- In the prisoner's dilemma both players defecting is the only Nash Equilibrium.
- 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. There are more solution concepts, e.g. correlated equilibria or best response.
- The best response is the best strategy given a belief about the strategy of the other players. In this game the best response to both strategies of the other player is to defect. If one strategy is the best response to any strategy of the other players, it is a dominant strategy.
- If two players play the prisoner's dilemma repeatedly, it is called iterated prisoner's dilemma. It is a dominant strategy to always defect. To see this, consider the final game. Defecting is a dominant strategy. Thus, it is fixed what both players do in the last game. Now the penultimate game is the last game and by induction always defecting is a dominant strategy.
- Game theorists were invited to come up with a strategy for 200 iterations of the prisoner's dilemma to compete in a tournament. Each strategy had to play against every other strategy and accumulated points throughout the tournament. The simple Tit4Tat strategy (cooperate in the first game, then copy whatever the other player did in the previous game) won. One year later, after analyzing each strategy, another tournament (with new strategies) was held. Tit4Tat won again.
- We now look at a distributed system game.

23.3 Selfish Caching

Computers in a network want to access a file regularly. Each node $v \in V$, with V being the set of nodes and $n = |V|$, has a demand d_v for the file and wants to minimize the cost for accessing it. In order to access the file, node v can either cache the file locally which costs 1 or request the file from another node u which costs $c_{v \leftarrow u}$. If a node does not cache the file, the cost it incurs is the minimal cost to access the file remotely. Note that if no node caches the file, then every node incurs cost ∞. There is an example in Figure 23.6.

Remarks:

- We will sometimes depict this game as a graph. The cost $c_{v \leftarrow u}$ for node v to access the file from node u is equivalent to the length of the shortest path times the demand d_v.

- Note that in undirected graphs $c_{u \leftarrow v} > c_{v \leftarrow u}$ if and only if $d_u > d_v$. We assume that the graphs are undirected for the rest of the chapter.

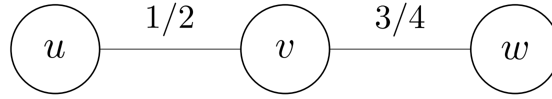


Figure 23.6: In this example we assume $d_u = d_v = d_w = 1$. Either the nodes u and w cache the file. Then neither of the three nodes has an incentive to change its behavior. The costs are 1, 1/2, and 1 for the nodes u, v, w, respectively. Alternatively, only node v caches the file. Again, neither of the three nodes has an incentive to change its behavior. The costs are 1/2, 1, and 3/4 for the nodes u, v, w, respectively.

Algorithm 23.7 Nash Equilibrium for Selfish Caching

```
1: S = {}                                   //set of nodes that cache the file
2: repeat
3:    Let v be a node with maximum demand d_v in set V
4:    S = S ∪ {v}, V = V \ {v}
5:    Remove every node u from V with c_{u←v} ≤ 1
6: until V = {}
```

Theorem 23.8. *Algorithm 23.7 computes a Nash Equilibrium for Selfish Caching.*

Proof. Let u be a node that is not caching the file. Then there exists a node v for which $c_{u \leftarrow v} \leq 1$. Hence, node u has no incentive to cache.

Let u be a node that is caching the file. We now consider any other node v that is also caching the file. First, we consider the case where v cached the file before u did. Then it holds that $c_{u \leftarrow v} > 1$ by construction.

It could also be that v started caching the file after u did. Then it holds that $d_u \geq d_v$ and therefore $c_{u \leftarrow v} \geq c_{v \leftarrow u}$. Furthermore, we have $c_{v \leftarrow u} > 1$ by construction. Combining these implies that $c_{u \leftarrow v} \geq c_{v \leftarrow u} > 1$.

In either case, node u has no incentive to stop caching. □

Definition 23.9 (Price of Anarchy)**.** *Let NE_- denote the Nash Equilibrium with the highest cost (smallest payoff). The **Price of Anarchy** (PoA) is defined as*

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

Definition 23.10 (Optimistic Price of Anarchy)**.** *Let NE_+ denote the Nash Equilibrium with the smallest cost (highest payoff). The **Optimistic Price of Anarchy** (OPoA) is defined as*

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

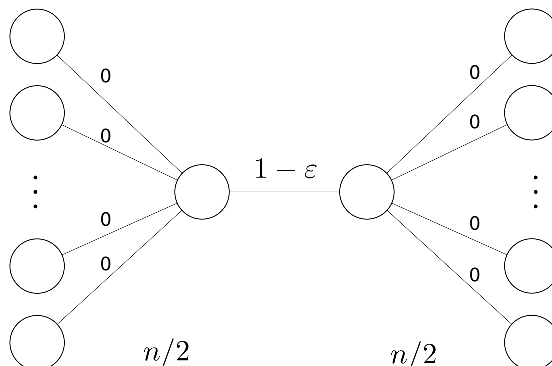


Figure 23.12: A network with a Price of Anarchy of $\Theta(n)$.

Remarks:

- The Price of Anarchy measures how much a distributed system degrades because of selfish nodes.
- We have $PoA \geq OPoA \geq 1$.

Theorem 23.11. *The (Optimistic) Price of Anarchy of Selfish Caching can be $\Theta(n)$.*

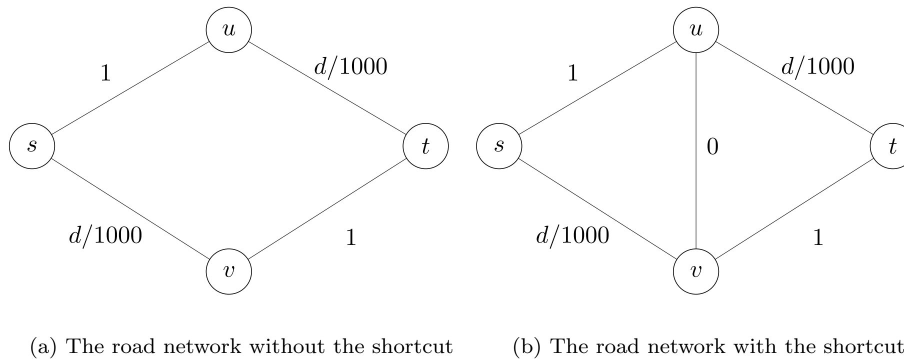
Proof. Consider a network as depicted in Figure 23.12. Every node v has demand $d_v = 1$. Note that if any node caches the file, no other node has an incentive to cache the file as well since the cost to access the file is at most $1-\varepsilon$. Without loss of generality, let us assume that a node v on the left caches the file, then it is cheaper for every node on the right to access the file remotely. Hence, the total cost of this solution is $1 + \frac{n}{2} \cdot (1-\varepsilon)$. In the social optimum one node from the left and one node from the right cache the file. This reduces the cost to 2. Hence, the Price of Anarchy is $\frac{1+\frac{n}{2}\cdot(1-\varepsilon)}{2} \underset{\varepsilon\to 0}{=} \frac{1}{2} + \frac{n}{4} = \Theta(n)$. □

23.4 Braess' Paradox

Consider the graph in Figure 23.13, it models a road network. Let us assume that there are 1000 drivers (each in their own car) that want to travel from node s to node t. Traveling along the road from s to u (or v to t) always takes 1 hour. The travel time from s to v (or u to t) depends on the traffic and increases by 1/1000 of an hour per car, i.e., when there are 500 cars driving, it takes 30 minutes to use this road.

Lemma 23.14. *Adding a super fast road (delay is 0) between u and v can increase the travel time from s to t.*

Proof. Since the drivers act rationally, they want to minimize the travel time. In the Nash Equilibrium, 500 drivers first drive to node u and then to t and 500 drivers first to node v and then to t. The travel time for each driver is 1 + 500 / 1000 = 1.5.



(a) The road network without the shortcut (b) The road network with the shortcut

Figure 23.13: Braess' Paradox, where d denotes the number of drivers using an edge.

To reduce congestion, a super fast road (delay is 0) is built between nodes u and v. This results in the following Nash Equilibrium: every driver now drives from s to v to u to t. The total cost is now $2 > 1.5$. □

Remarks:

- There are physical systems which exhibit similar properties. Some famous ones employ a spring. YouTube has some fascinating videos about this. Simply search for "Braess Paradox Spring".
- We will now look at another famous game that will allow us to deepen our understanding of game theory.

23.5 Rock-Paper-Scissors

There are two players, u and v. Each player simultaneously chooses one of three options: rock, paper, or scissors. The rules are simple: paper beats rock, rock beats scissors, and scissors beat paper. A matrix representation of this game is in Table 23.15.

u / v		Player u		
		Rock	Paper	Scissors
Player v	Rock	0 0	1 -1	-1 1
	Paper	-1 1	0 0	1 -1
	Scissors	1 -1	-1 1	0 0

Table 23.15: Rock-Paper-Scissors as a matrix.

Remarks:

- None of the three strategies is a Nash Equilibrium. Whatever player u chooses, player v can always switch her strategy such that she wins.
- This is highlighted in the best response concept. The best response to e.g. scissors is to play rock. The other player switches to paper. And so on.
- Is this a game without a Nash Equilibrium? John Nash answered this question in 1950. By choosing each strategy with a certain probability, we can obtain a so called mixed Nash Equilibrium. Indeed:

Theorem 23.16. *Every game has a mixed Nash Equilibrium.*

Remarks:

- The Nash Equilibrium of this game is if both players choose each strategy with probability 1/3. The expected payoff is 0.
- Any strategy (or mix of them) is a best response to a player choosing each strategy with probability 1/3.
- In a pure Nash Equilibrium, the strategies are chosen deterministically. Rock-Paper-Scissors does not have a pure Nash Equilibrium.
- Even though every game has a mixed Nash Equilibrium. Sometimes such an equilibrium is computationally difficult to compute. One should be cautious about economic assumptions such as "the market will always find the equilibrium".
- Unfortunately, game theory does not always model problems accurately. Many real world problems are too complex to be captured by a game. And as you may know, humans (not only politicians) are often not rational.
- In distributed systems, players can be servers, routers, etc. Game theory can tell us whether systems and protocols are prone to selfish behavior.

23.6 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".

Definition 23.17 (auction)**.** *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.*

Remarks:

- For simplicity, we assume that no two bids are the same, and that $b_1 > b_2 > b_3 > \ldots$

Definition 23.19 (truthful)**.** *An auction is truthful if no player v_i can gain anything by not stating the truth, i.e., $b_i = z_i$.*

Algorithm 23.18 First Price Auction

1: every bidder v_i submits his bid b_i
2: the good is allocated to the highest bidder v_1 for the price $p = b_1$

Theorem 23.20. *A First Price Auction (Algorithm 23.18) is not truthful.*

Proof. Consider an auction with two bidders, with bids b_1 and b_2. By not stating the truth and decreasing his bid to $b_1 - \varepsilon > b_2$, player one could pay less and thus gain more. Thus, the first price auction is not truthful. □

Algorithm 23.21 Second Price Auction

1: every bidder v_i submits his bid b_i
2: the good is allocated to the highest bidder v_1 for $p = b_2$

Theorem 23.22. *Truthful bidding is a dominant strategy in a Second Price Auction.*

Proof. Let z_i be the truthful value of node v_i and b_i his bid. Let $b_{\max} = \max_{j \neq i} b_j$ is the largest bid from other nodes but v_i. The payoff for node v_i is $z_i - b_{\max}$ if $b_i > b_{\max}$ and 0 else. Let us consider overbidding first, i.e., $b_i > z_i$:

- If $b_{\max} < z_i < b_i$, then both strategies win and yield the same payoff $(z_i - b_{\max})$.
- If $z_i < b_i < b_{\max}$, then both strategies lose and yield a payoff of 0.
- If $z_i < b_{\max} < b_i$, then overbidding wins the auction, but the payoff $(z_i - b_{\max})$ is negative. Truthful bidding loses and yields a payoff of 0.

Likewise underbidding, i.e. $b_i < z_i$:

- If $b_{\max} < b_i < z_i$, then both strategies win and yield the same payoff $(z_i - b_{\max})$.
- If $b_i < z_i < b_{\max}$, then both strategies lose and yield a payoff of 0.
- If $b_i < b_{\max} < z_i$, then truthful bidding wins and yields a positive payoff $(z_i - b_{\max})$. Underbidding loses and yields a payoff of 0.

Hence, truthful bidding is a dominant strategy for each node v_i. □

Remarks:

- Let us use this for Selfish Caching. We need to choose a node that is the first to cache the file. But how? By holding an auction. Every node says for which price it is willing to cache the file. We pay the node with the lowest offer and pay it the second lowest offer to ensure truthful offers.
- Since a mechanism designer can manipulate incentives, she can implement a strategy profile by making all the strategies in this profile dominant.

Theorem 23.23. *Any Nash Equilibrium of Selfish Caching can be implemented for free.*

Proof. If the mechanism designer wants the nodes from the caching set S of the Nash Equilibrium to cache, then she can offer the following deal to every node not in S: "If any node from set S does not cache the file, then I will ensure a positive payoff for you." Thus, all nodes not in S prefer not to cache since this is a dominant strategy for them. Consider now a node $v \in S$. Since S is a Nash Equilibrium, node v incurs cost of at least 1 if it does not cache the file. For nodes that incur cost of exactly 1, the mechanism designer can even issue a penalty if the node does not cache the file. Thus, every node $v \in S$ caches the file. □

Remarks:

- Mechanism design assumes that the players act rationally and want to maximize their payoff. In real-world distributed systems some players may be not selfish, but actively malicious (byzantine).

- What about P2P file sharing? To increase the overall experience, BitTorrent suggests that peers offer better upload speed to peers who upload more. This idea can be exploited. By always claiming to have nothing to trade yet, the BitThief client downloads without uploading. In addition to that, it connects to more peers than the standard client to increase its download speed.

- Many techniques have been proposed to limit such free riding behavior, e.g., tit-for-tat trading: I will only share something with you if you share something with me. To solve the bootstrap problem ("I don't have anything yet"), nodes receive files or pieces of files whose hash match their own hash for free. One can also imagine indirect trading. Peer u uploads to peer v, who uploads to peer w, who uploads to peer u. Finally, one could imagine using virtual currencies or a reputation system (a history of who uploaded what). Reputation systems suffer from collusion and Sybil attacks. If one node pretends to be many nodes who rate each other well, it will have a good reputation.

Chapter Notes

Game theory was started by a proof for mixed-strategy equilibria in two-person zero-sum games by John von Neumann [Neu28]. Later, von Neumann and Morgenstern introduced game theory to a wider audience [NM44]. In 1950 John Nash proved that every game has a mixed Nash Equilibrium [Nas50]. The Prisoner's Dilemma was first formalized by Flood and Dresher [Flo52]. The iterated prisoner's dilemma tournament was organized by Robert Axelrod [AH81]. The Price of Anarchy definition is from Koutsoupias and Papadimitriou [KP99]. This allowed the creation of the Selfish Caching Game [CCW$^+$04], which we used as a running example in this chapter. Braess' paradox was discovered by Dietrich Braess in 1968 [Bra68]. A generalized version of the second-price auction is the VCG auction, named after three successive papers from first Vickrey,

then Clarke, and finally Groves [Vic61, Cla71, Gro73]. One popular example of selfishness in practice is BitThief – a BitTorrent client that successfully downloads without uploading [LMSW06]. Using game theory economists try to understand markets and predict crashes. Apart from John Nash, the Sveriges Riksbank Prize (Nobel Prize) in Economics has been awarded many times to game theorists. For example in 2007 Hurwicz, Maskin, and Myerson received the prize for "for having laid the foundations of mechanism design theory". There is a considerable amount of work on mixed adversarial models with byzantine, altruistic, and rational ("BAR") players, e.g., [AAC+05, ADGH06, MSW06]. Daskalakis et al. [DGP09] showed that computing a Nash Equilibrium may not be trivial.

This chapter was written in collaboration with Philipp Brandes.

Bibliography

[AAC+05] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 45–58, 2005.

[ADGH06] Ittai Abraham, Danny Dolev, Rica Gonen, and Joseph Y. Halpern. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 2006, Denver, CO, USA, July 23-26, 2006*, pages 53–62, 2006.

[AH81] Robert Axelrod and William Donald Hamilton. The evolution of cooperation. *Science*, 211(4489):1390–1396, 1981.

[Bra68] Dietrich Braess. Über ein paradoxon aus der verkehrsplanung. *Unternehmensforschung*, 12(1):258–268, 1968.

[CCW+04] Byung-Gon Chun, Kamalika Chaudhuri, Hoeteck Wee, Marco Barreno, Christos H Papadimitriou, and John Kubiatowicz. Selfish caching in distributed systems: a game-theoretic analysis. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 21–30. ACM, 2004.

[Cla71] Edward H Clarke. Multipart pricing of public goods. *Public choice*, 11(1):17–33, 1971.

[DGP09] Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a nash equilibrium. *SIAM J. Comput.*, 39(1):195–259, 2009.

[Flo52] Merrill M Flood. Some experimental games. *Management Science*, 5(1):5–26, 1952.

[Gro73] Theodore Groves. Incentives in teams. *Econometrica: Journal of the Econometric Society*, pages 617–631, 1973.

[KP99] Elias Koutsoupias and Christos Papadimitriou. Worst-case equilibria. In *STACS 99*, pages 404–413. Springer, 1999.

[LMSW06] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. In *5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA*, November 2006.

[MSW06] Thomas Moscibroda, Stefan Schmid, and Roger Wattenhofer. When selfish meets evil: byzantine players in a virus inoculation game. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 2006, Denver, CO, USA, July 23-26, 2006*, pages 35–44, 2006.

[Nas50] John F. Nash. Equilibrium points in n-person games. *Proc. Nat. Acad. Sci. USA*, 36(1):48–49, 1950.

[Neu28] John von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[NM44] John von Neumann and Oskar Morgenstern. *Theory of games and economic behavior*. Princeton university press, 1944.

[Vic61] William Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of finance*, 16(1):8–37, 1961.

Chapter 24

Distributed Storage

How do you store 1M movies, each with a size of about 1GB, on 1M nodes, each equipped with a 1TB disk? Simply store the movies on the nodes, arbitrarily, and memorize (with a global index) which movie is stored on which node. What if the set of movies or nodes changes over time, and you do not want to change your global index too often?

24.1 Consistent Hashing

Several variants of hashing will do the job, e.g. consistent hashing:

Algorithm 24.1 Consistent Hashing

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

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

Theorem 24.2 (Consistent Hashing)**.** *In expectation, each node in Algorithm 24.1 stores km/n movies, where k is the number of hash functions, m the number of different movies and n the number of nodes.*

Proof. For a specific movie (out of m) and a specific hash function (out of k), all n nodes have the same probability $1/n$ to hash closest to the movie hash. By linearity of expectation, each node stores km/n movies in expectation if we also count duplicates of movies on a node. □

Remarks:

- Let us do a back-of-the-envelope calculation. We have $m = 1\text{M}$ movies, $n = 1\text{M}$ nodes, each node has storage for 1TB/1GB = 1K movies, i.e., we use $k = 1\text{K}$ hash functions. Theorem 24.2 shows each node stores about 1K movies.

- Using the Chernoff bound below with $\mu = km/n = 1\text{K}$, the probability that a node uses 10% more memory than expected is less than 1%.

Facts 24.3. *A version of a **Chernoff bound** states the following:*
Let $x_1, \ldots, x_n$ be independent Bernoulli-distributed random variables with $Pr[x_i = 1] = p_i$ and $Pr[x_i = 0] = 1 - p_i = q_i$, then for $X := \sum_{i=1}^{n} x_i$ and $\mu := \mathbb{E}[X] = \sum_{i=1}^{n} p_i$ the following holds:

$$\text{for any } \delta > 0\text{: } Pr[X \geq (1+\delta)\mu] < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

Remarks:

- Instead of storing movies directly on nodes as in Algorithm 24.1, we can also store the movies on any nodes we like. The nodes of Algorithm 24.1 then simply store forward pointers to the actual movie locations.

- For better load balancing, we might also hash nodes multiple times.

- In this chapter we want to push unreliability to the extreme. What if the nodes are so unreliable that on average a node is only available for 1 hour? In other words, nodes exhibit a high *churn*, they constantly join and leave the distributed system.

- With such a high churn, hundreds or thousands of nodes will change every second. No single node can have an accurate picture of what other nodes are currently in the system. This is remarkably different to classic distributed systems, where a single unavailable node may already be a minor disaster: all the other nodes have to get a consistent view (Definition 25.5) of the system again. In high churn systems it is impossible to have a consistent view at any time.

- Instead, each node will just know about a small subset of 100 or less other nodes ("neighbors"). This way, nodes can withstand high churn situations.

- On the downside, nodes will not directly know which node is responsible for what movie. Instead, a node searching for a movie might have to ask a neighbor node, which in turn will recursively ask another neighbor node, until the correct node storing the movie (or a forward pointer to the movie) is found. The nodes of our distributed storage system form a virtual network, also called an *overlay network*.

24.2 Hypercubic Networks

In this section we present a few overlay topologies of general interest.

Definition 24.4 (Topology Properties)**.** *Our virtual network should have the following properties:*

- *The network should be (somewhat)* ***homogeneous****: no node should play a dominant role, no node should be a single point of failure.*
- *The nodes should have* ***IDs****, and the IDs should span the universe* $[0, 1)$*, such that we can store data with hashing, as in Algorithm 24.1.*
- *Every node should have a small* ***degree****, if possible polylogarithmic in* n*, the number of nodes. This will allow every node to maintain a persistent connection with each neighbor, which will help us to deal with churn.*
- *The network should have a small* ***diameter****, and routing should be easy. If a node does not have the information about a data item, then it should know which neighbor to ask. Within a few (polylogarithmic in* n*) hops, one should find the node that has the correct information.*

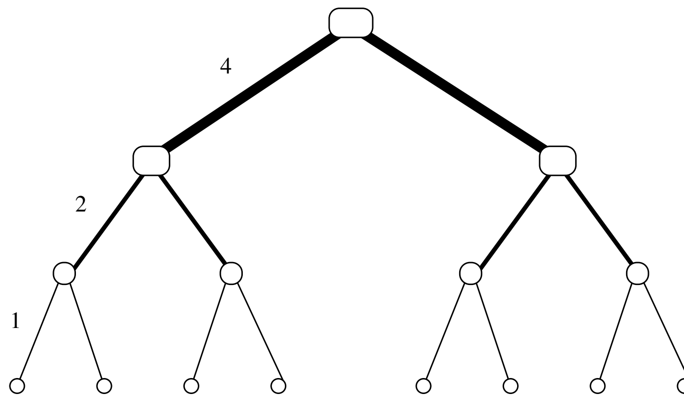


Figure 24.5: The structure of a fat tree.

Remarks:

- Some basic network topologies used in practice are trees, rings, grids or tori. Many other suggested networks are simply combinations or derivatives of these.
- The advantage of trees is that the routing is very easy: for every source-destination pair there is only one path. However, since the root of a tree is a bottleneck, trees are not homogeneous. Instead, so-called *fat trees* should be used. Fat trees have the property that every edge connecting a node v to its parent u has a capacity that is proportional to the number of leaves of the subtree rooted at v. See Figure 24.5 for a picture.

- Fat trees belong to a family of networks that require edges of non-uniform capacity to be efficient. Networks with edges of uniform capacity are easier to build. This is usually the case for grids and tori. Unless explicitly mentioned, we will treat all edges in the following to be of capacity 1.

Definition 24.6 (Torus, Mesh)**.** *Let $m, d \in \mathbb{N}$. The (m,d)-**mesh** $M(m,d)$ is a graph with node set $V = [m]^d$ and edge set*

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

*where $[m]$ means the set $\{0,\ldots,m-1\}$. The (m,d)-**torus** $T(m,d)$ is a graph that consists of an (m,d)-mesh and additionally wrap-around edges from nodes $(a_1,\ldots,a_{i-1},m-1,a_{i+1},\ldots,a_d)$ to nodes $(a_1,\ldots,a_{i-1},0,a_{i+1},\ldots,a_d)$ for all $i \in \{1,\ldots,d\}$ and all $a_j \in [m]$ with $j \neq i$. In other words, we take the expression $a_i - b_i$ in the sum modulo m prior to computing the absolute value. $M(m,1)$ is also called a **path**, $T(m,1)$ a **cycle**, and $M(2,d) = T(2,d)$ a d-**dimensional hypercube**. Figure 24.7 presents a linear array, a torus, and a hypercube.*

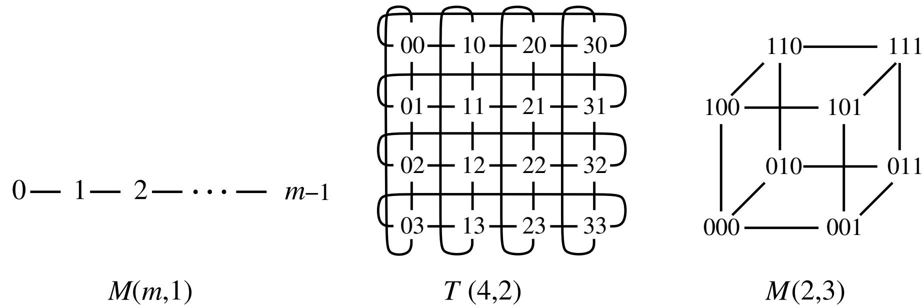


Figure 24.7: The structure of $M(m,1)$, $T(4,2)$, and $M(2,3)$.

Remarks:

- 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; in other words, if the source and the target differ by k bits, there are $k!$ routes with k hops.

- As required by Definition 24.4, the d-bit IDs of the nodes need to be mapped to the universe $[0,1)$. One way to do this is by interpreting an ID as the binary representation of the fractional part of a decimal number. For example, the ID **101** is mapped to $\mathbf{0.101_2}$ which has a decimal value of $\mathbf{0} \cdot 2^0 + \mathbf{1} \cdot 2^{-1} + \mathbf{0} \cdot 2^{-2} + \mathbf{1} \cdot 2^{-3} = \frac{5}{8}$.

- The Chord architecture is a close relative of the hypercube, basically a less rigid hypercube. The hypercube connects every node with an ID in $[0,1)$ with every node in *exactly* distance 2^{-i}, $i = 1,2,\ldots,d$ in $[0,1)$. Chord instead connect nodes with *approximately* distance 2^{-i}.

- The hypercube has many derivatives, the so-called *hypercubic networks*. Among these are the butterfly, cube-connected-cycles, shuffle-exchange, and de Bruijn graph. We start with the butterfly, which is basically a "rolled out" hypercube.

Definition 24.8 (Butterfly)**.** *Let $d \in \mathbb{N}$. The* ***d-dimensional butterfly*** *$BF(d)$ is a graph with node set $V = [d+1] \times [2]^d$ and an edge set $E = E_1 \cup E_2$ with*

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

and

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

A node set $\{(i, \alpha) \mid \alpha \in [2]^d\}$ is said to form ***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 $\alpha \in [2]^d$.*

Remarks:

- Figure 24.9 shows the 3-dimensional butterfly $BF(3)$. The $BF(d)$ has $(d+1)2^d$ nodes, $2d \cdot 2^d$ edges and degree 4. It is not difficult to check that combining the node sets $\{(i, \alpha) \mid i \in [d]\}$ for all $\alpha \in [2]^d$ into a single node results in the hypercube.

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

- You may have seen butterfly-like structures before, e.g. sorting networks, communication switches, data center networks, fast fourier transform (FFT). The Benes network (telecommunication) is nothing but two back-to-back butterflies. The Clos network (data centers) is a close relative to Butterflies too. Actually, merging the 2^i nodes on level i that share the first $d - i$ bits into a single node, the Butterfly becomes a fat tree. Every year there are new applications for which hypercubic networks are the perfect solution!

- Next we define the cube-connected-cycles network. It only has a degree of 3 and it results from the hypercube by replacing the corners by cycles.

Definition 24.10 (Cube-Connected-Cycles)**.** *Let $d \in \mathbb{N}$. 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*

$$\begin{aligned} E = & \big\{\{(a, p), (a, (p+1) \bmod d)\} \mid a \in [2]^d, p \in [d]\big\} \\ & \cup \big\{\{(a, p), (b, p)\} \mid a, b \in [2]^d, p \in [d], |a - b| = 2^p\big\} \end{aligned}$$

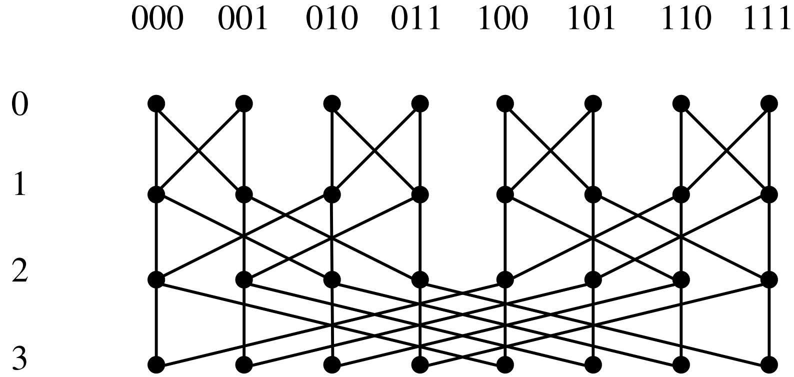


Figure 24.9: The structure of BF(3).

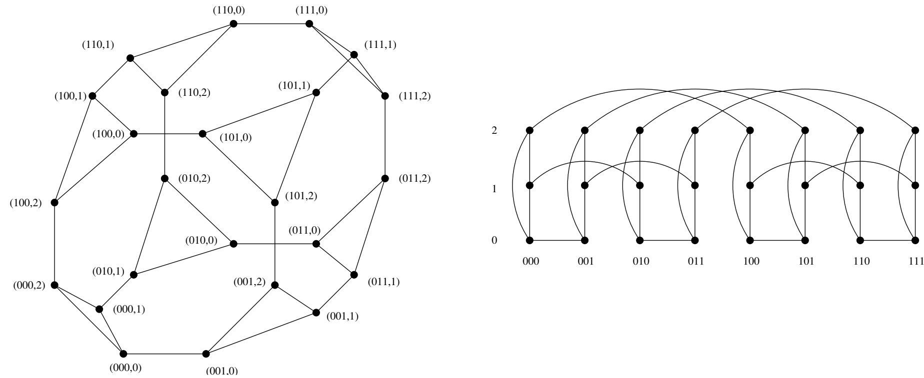


Figure 24.11: The structure of CCC(3).

Remarks:

- Two possible representations of a CCC can be found in Figure 24.11.
- The shuffle-exchange is yet another way of transforming the hypercubic interconnection structure into a constant degree network.

Definition 24.12 (Shuffle-Exchange)**.** *Let $d \in \mathbb{N}$. The* ***d-dimensional shuffle-exchange*** *$SE(d)$ is defined as an undirected graph with node set $V = [2]^d$ and an edge set $E = E_1 \cup E_2$ with*

$$E_1 = \{\{(a_1, \ldots, a_d), (a_1, \ldots, \bar{a}_d)\} \mid (a_1, \ldots, a_d) \in [2]^d,\ \bar{a}_d = 1 - a_d\}$$

and

$$E_2 = \{\{(a_1, \ldots, a_d), (a_d, a_1, \ldots, a_{d-1})\} \mid (a_1, \ldots, a_d) \in [2]^d\} \ .$$

Figure 24.13 shows the 3- and 4-dimensional shuffle-exchange graph.

Definition 24.14 (DeBruijn)**.** *The* ***b-ary DeBruijn graph of dimension*** *d $DB(b, d)$ is an undirected graph $G = (V, E)$ with node set $V = \{v \in [b]^d\}$*

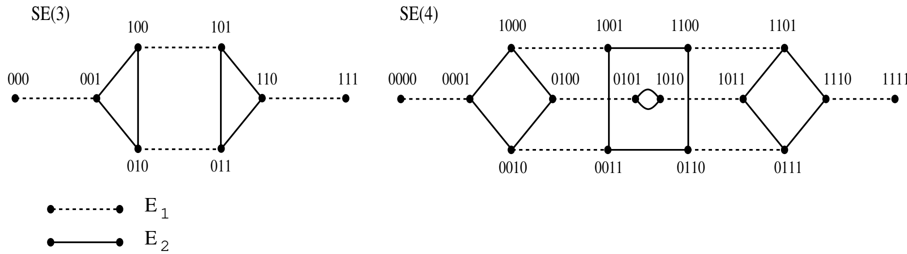


Figure 24.13: The structure of SE(3) and SE(4).

and edge set E that contains all edges $\{v, w\}$ with the property that $w \in \{(x, v_1, \ldots, v_{d-1}) : x \in [b]\}$, where $v = (v_1, \ldots, v_d)$.

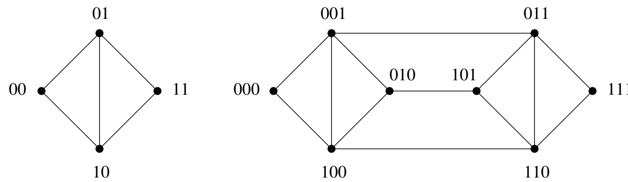


Figure 24.15: The structure of $DB(2, 2)$ and $DB(2, 3)$.

Remarks:

- Two examples of a DeBruijn graph can be found in Figure 24.15.
- There are some data structures which also qualify as hypercubic networks. An example of a hypercubic network is the skip list, the balanced binary search tree for the lazy programmer:

Definition 24.16 (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.*

Remarks:

- Search, insert, and delete can be implemented in $\mathcal{O}(\log n)$ expected time in a skip list, simply by jumping from higher levels to lower ones when overshooting the searched position. Also, the amortized memory cost of each object is constant, as on average an object only has two forward links.
- The randomization can easily be discarded, by deterministically promoting a constant fraction of objects of level i to level $i + 1$, for all

i. When inserting or deleting, object o simply checks whether its left and right level i neighbors are being promoted to level $i+1$. If none of them is, promote object o itself. Essentially we establish a maximal independent set (MIS) on each level, hence at least every third and at most every second object is promoted.

- There are obvious variants of the skip list, e.g., the skip graph. Instead of promoting only half of the nodes to the next level, we always promote all the nodes, similarly to a balanced binary tree: All nodes are part of the root level of the binary tree. Half the nodes are promoted left, and half the nodes are promoted right, on each level. Hence on level i we have have 2^i lists (or, if we connect the last element again with the first: rings) of about $n/2^i$ objects. The skip graph features all the properties of Definition 24.4.
- More generally, how are degree and diameter of Definition 24.4 related? The following theorem gives a general lower bound.

Theorem 24.17. *Every graph of maximum degree $d > 2$ and size n must have a diameter of at least $\lceil (\log n)/(\log(d-1)) \rceil - 2$.*

Proof. Suppose we have a graph $G = (V, E)$ of maximum degree d and size n. Start from any node $v \in V$. In a first step at most d other nodes can be reached. In two steps at most $d \cdot (d-1)$ additional nodes can be reached. Thus, in general, in at most r steps at most

$$1 + \sum_{i=0}^{r-1} d \cdot (d-1)^i = 1 + d \cdot \frac{(d-1)^r - 1}{(d-1) - 1} \leq \frac{d \cdot (d-1)^r}{d-2}$$

nodes (including v) can be reached. This has to be at least n to ensure that v can reach all other nodes in V within r steps. Hence,

$$(d-1)^r \geq \frac{(d-2) \cdot n}{d} \quad \Leftrightarrow \quad r \geq \log_{d-1}((d-2) \cdot n/d) .$$

Since $\log_{d-1}((d-2)/d) > -2$ for all $d > 2$, this is true only if $r \geq \lceil (\log n)/(\log(d-1)) \rceil - 2$. □

Remarks:

- 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. The pancake graph, for instance, minimizes the maximum of these with $\max(d, D) = \Theta(\log n / \log\log n)$. The ID of a node u in the pancake graph of dimension d is an arbitrary permutation of the numbers $1, 2, \ldots, d$. Two nodes u, v are connected by an edge if one can get the ID of node v by taking the ID of node u, and reversing (flipping) the first k (for $k = 1, \ldots, d$) numbers of u's ID. For example, in dimension $d = 4$, nodes $u = 2314$ and $v = 1324$ are neighbors.

- There are a few other interesting graph classes which are not hypercubic networks, but nevertheless seem to relate to the properties of Definition 24.4. Small-world graphs (a popular representations for social networks) also have small diameter, however, in contrast to hypercubic networks, they are not homogeneous and feature nodes with large degrees.
- Expander graphs (an expander graph is a sparse graph which has good connectivity properties, that is, from every not too large subset of nodes you are connected to an even larger set of nodes) are homogeneous, have a low degree and small diameter. However, expanders are often not routable.

24.3 DHT & Churn

Definition 24.18 (Distributed Hash Table (DHT))**.** *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.*

Remarks:

- A DHT has many applications beyond storing movies, e.g., the Internet domain name system (DNS) is essentially a DHT.
- A DHT can be implemented as a hypercubic overlay network with nodes having identifiers such that they span the ID space $[0, 1)$.
- A hypercube can directly be used for a DHT. Just use a globally known set of hash functions h_i, mapping movies to bit strings with d bits.
- Other hypercubic structures may be a bit more intricate when using it as a DHT: The butterfly network, for instance, may directly use the $d+1$ layers for replication, i.e., all the $d+1$ nodes are responsible for the same ID.
- Other hypercubic networks, e.g. the pancake graph, might need a bit of twisting to find appropriate IDs.
- We assume that a joining node knows a node which already belongs to the system. This is known as the bootstrap problem. Typical solutions are: If a node has been connected with the DHT previously, just try some of these previous nodes. Or the node may ask some authority for a list of IP addresses (and ports) of nodes that are regularly part of the DHT.
- Many DHTs in the literature are analyzed against an adversary that can crash a fraction of random nodes. After crashing a few nodes the system is given sufficient time to recover again. However, this seems unrealistic. The scheme sketched in this section significantly differs from this in two major aspects.

- First, we assume that joins and leaves occur in a worst-case manner. We think of an adversary that can remove and add a bounded number of nodes; the adversary can choose which nodes to crash and how nodes join.
- Second, the adversary does not have to wait until the system is recovered before it crashes the next batch of nodes. Instead, the adversary can constantly crash nodes, while the system is trying to stay alive. Indeed, the system is *never fully repaired* but *always fully functional*. In particular, the system is resilient against an adversary that continuously attacks the "weakest part" of the system. The adversary could for example insert a crawler into the DHT, learn the topology of the system, and then repeatedly crash selected nodes, in an attempt to partition the DHT. The system counters such an adversary by continuously moving the remaining or newly joining nodes towards the areas under attack.
- Clearly, we cannot allow the adversary to have unbounded capabilities. In particular, in any constant time interval, the adversary can at most add and/or remove $O(\log n)$ nodes, n being the total number of nodes currently in the system. This model covers an adversary which repeatedly takes down nodes by a distributed denial of service attack, however only a logarithmic number of nodes at each point in time. The algorithm relies on messages being delivered timely, in at most constant time between any pair of operational nodes, i.e., the synchronous model. Using the trivial synchronizer this is not a problem. We only need bounded message delays in order to have a notion of time which is needed for the adversarial model. The duration of a round is then proportional to the propagation delay of the slowest message.

Algorithm 24.19 DHT

1: Given: a globally known set of hash functions h_i, and a hypercube (or any other hypercubic network)
2: Each hypercube virtual node ("hypernode") consists 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.

Remarks:

- Having a logarithmic number of hypercube neighbors, each with a logarithmic number of nodes, means that each node has $\Theta(\log^2 n)$ neighbors. However, with some additional bells and whistles one can achieve $\Theta(\log n)$ neighbor nodes.

- The balancing of nodes among the hypernodes can be seen as a dynamic token distribution problem on the hypercube. Each hypernode has a certain number of tokens, the goal is to distribute the tokens along the edges of the graph such that all hypernodes end up with the same or almost the same number of tokens. While tokens are moved around, an adversary constantly inserts and deletes tokens. See also Figure 24.20.

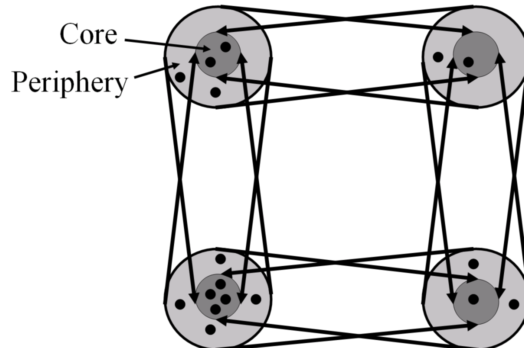


Figure 24.20: A simulated 2-dimensional hypercube with four hypernodes, each consisting of several nodes. Also, all the nodes are either in the core or in the periphery of a node. All nodes within the same hypernode are completely connected to each other, and additionally, all nodes of a hypernode are connected to the core nodes of the neighboring nodes. Only the core nodes store data items, while the peripheral nodes move between the nodes to balance biased adversarial churn.

- In summary, the storage system builds on two basic components: (i) an algorithm which performs the described dynamic token distribution and (ii) an information aggregation algorithm which is used to estimate the number of nodes in the system and to adapt the dimension of the hypercube accordingly:

Theorem 24.21 (DHT with Churn). *We have a fully scalable, efficient distributed storage system which tolerates* $O(\log n)$ *worst-case joins and/or crashes per constant time interval. As in other storage systems, nodes have* $O(\log n)$ *overlay neighbors, and the usual operations (e.g., search, insert) take time* $O(\log n)$.

Remarks:

- Indeed, handling churn is only a minimal requirement to make a distributed storage system work. Advanced studies proposed more elaborate architectures which can also handle other security issues, e.g., privacy or Byzantine attacks.

Chapter Notes

The ideas behind distributed storage were laid during the peer-to-peer (P2P) file sharing hype around the year 2000, so a lot of the seminal research

in this area is labeled P2P. The paper of Plaxton, Rajaraman, and Richa [PRR97] laid out a blueprint for many so-called structured P2P architecture proposals, such as Chord [SMK+01], CAN [RFH+01], Pastry [RD01], Viceroy [MNR02], Kademlia [MM02], Koorde [KK03], SkipGraph [AS03], SkipNet [HJS+03], or Tapestry [ZHS+04]. Also the paper of Plaxton et. al. was standing on the shoulders of giants. Some of its eminent precursors are: linear and consistent hashing [KLL+97], locating shared objects [AP90, AP91], compact routing [SK85, PU88], and even earlier: hypercubic networks, e.g. [AJ75, Wit81, GS81, BA84].

Furthermore, the techniques in use for prefix-based overlay structures are related to a proposal called LAND, a locality-aware distributed hash table proposed by Abraham et al. [AMD04].

More recently, a lot of P2P research focussed on security aspects, describing for instance attacks [LMSW06, SENB07, Lar07], and provable countermeasures [KSW05, AS09, BSS09]. Another topic currently garnering interest is using P2P to help distribute live streams of video content on a large scale [LMSW07]. There are several recommendable introductory books on P2P computing, e.g. [SW05, SG05, MS07, KW08, BYL08].

Some of the figures in this chapter have been provided by Christian Scheideler.

Bibliography

[AJ75] George A. Anderson and E. Douglas Jensen. Computer Interconnection Structures: Taxonomy, Characteristics, and Examples. *ACM Comput. Surv.*, 7(4):197–213, December 1975.

[AMD04] Ittai Abraham, Dahlia Malkhi, and Oren Dobzinski. LAND: stretch (1 + epsilon) locality-aware networks for DHTs. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '04, pages 550–559, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[AP90] Baruch Awerbuch and David Peleg. Efficient Distributed Construction of Sparse Covers. Technical report, The Weizmann Institute of Science, 1990.

[AP91] Baruch Awerbuch and David Peleg. Concurrent Online Tracking of Mobile Users. In *SIGCOMM*, pages 221–233, 1991.

[AS03] James Aspnes and Gauri Shah. Skip Graphs. In *SODA*, pages 384–393. ACM/SIAM, 2003.

[AS09] Baruch Awerbuch and Christian Scheideler. Towards a Scalable and Robust DHT. *Theory Comput. Syst.*, 45(2):234–260, 2009.

[BA84] L. N. Bhuyan and D. P. Agrawal. Generalized Hypercube and Hyperbus Structures for a Computer Network. *IEEE Trans. Comput.*, 33(4):323–333, April 1984.

[BSS09] Matthias Baumgart, Christian Scheideler, and Stefan Schmid. A DoS-resilient information system for dynamic data management. In

Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures, SPAA '09, pages 300–309, New York, NY, USA, 2009. ACM.

[BYL08] John Buford, Heather Yu, and Eng Keong Lua. *P2P Networking and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[GS81] J.R. Goodman and C.H. Sequin. Hypertree: A Multiprocessor Interconnection Topology. *Computers, IEEE Transactions on*, C-30(12):923–933, dec. 1981.

[HJS+03] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. SkipNet: a scalable overlay network with practical locality properties. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[KK03] M. Frans Kaashoek and David R. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In M. Frans Kaashoek and Ion Stoica, editors, *IPTPS*, volume 2735 of *Lecture Notes in Computer Science*, pages 98–107. Springer, 2003.

[KLL+97] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In Frank Thomson Leighton and Peter W. Shor, editors, *STOC*, pages 654–663. ACM, 1997.

[KSW05] Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. A Self-Repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn. In *4th International Workshop on Peer-To-Peer Systems (IPTPS), Cornell University, Ithaca, New York, USA, Springer LNCS 3640*, February 2005.

[KW08] Javed I. Khan and Adam Wierzbicki. Introduction: Guest editors' introduction: Foundation of peer-to-peer computing. *Comput. Commun.*, 31(2):187–189, February 2008.

[Lar07] Erik Larkin. Storm Worm's virulence may change tactics. http://www.networkworld.com/news/2007/080207-black-hat-storm-worms-virulence.html, Agust 2007. Last accessed on June 11, 2012.

[LMSW06] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. In *5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA*, November 2006.

[LMSW07] Thomas Locher, Remo Meier, Stefan Schmid, and Roger Wattenhofer. Push-to-Pull Peer-to-Peer Live Streaming. In *21st International Symposium on Distributed Computing (DISC), Lemesos, Cyprus*, September 2007.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 183–192, New York, NY, USA, 2002. ACM.

[MS07] Peter Mahlmann and Christian Schindelhauer. *Peer-to-Peer Networks.* Springer, 2007.

[PRR97] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *SPAA*, pages 311–320, 1997.

[PU88] David Peleg and Eli Upfal. A tradeoff between space and efficiency for routing tables. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 43–52, New York, NY, USA, 1988. ACM.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, August 2001.

[SENB07] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. Exploiting KAD: possible uses and misuses. *SIGCOMM Comput. Commun. Rev.*, 37(5):65–70, October 2007.

[SG05] Ramesh Subramanian and Brian D. Goodman. *Peer to Peer Computing: The Evolution of a Disruptive Technology.* IGI Publishing, Hershey, PA, USA, 2005.

[SK85] Nicola Santoro and Ramez Khatib. Labelling and Implicit Routing in Networks. *Comput. J.*, 28(1):5–8, 1985.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.

[SW05] Ralf Steinmetz and Klaus Wehrle, editors. *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science.* Springer, 2005.

[Wit81] L. D. Wittie. Communication Structures for Large Networks of Microcomputers. *IEEE Trans. Comput.*, 30(4):264–273, April 1981.

[ZHS+04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

Chapter 25

Authenticated Agreement

In Section 12.5 we have already had a glimpse into the power of cryptography. In this Chapter we want to build a *practical* byzantine fault-tolerant system using cryptography. With cryptography, Byzantine lies may be detected easily.

25.1 Agreement with Authentication

Definition 25.1 (Signature)**.** *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* $\texttt{msg}(x)$ *signed by node* u *with* $\texttt{msg}(x)_u$.

Remarks:

- Algorithm 25.2 shows a synchronous agreement protocol for binary inputs relying on signatures. We assume there is a designated "primary" node p that all other nodes know. The goal is to decide on p's value.

Theorem 25.3. *Algorithm 25.2 can tolerate* $f < n$ *byzantine failures while terminating in* $f + 1$ *rounds.*

Proof. Assuming that the primary p is not byzantine and its input is 1, then p broadcasts $\texttt{value}(1)_p$ in the first round, which will trigger all correct nodes to decide on 1. If p's input is 0, there is no signed message $\texttt{value}(1)_p$, and no node can decide on 1.

If primary p is byzantine, we need all correct nodes to decide on the same value for the algorithm to be correct.

Assume $i < f + 1$ is minimal among all rounds in which any correct node u decides on 1. In this case, u has a set S of at least i messages from other nodes for value 1 in round i, including one of p. Therefore, in round $i + 1 \leq f + 1$, all other correct nodes will receive S and u's message for value 1 and thus decide on 1 too.

Now assume that $i = f + 1$ is minimal among all rounds in which a correct node u decides for 1. Thus u must have received $f + 1$ messages for value 1, one

Algorithm 25.2 Byzantine Agreement with Authentication

Code for primary p:

```
1: if input is 1 then
2:    broadcast value(1)_p
3:    decide 1 and terminate
4: else
5:    decide 0 and terminate
6: end if
```

Code for all other nodes v:

```
7: for all rounds i ∈ {1, ..., f + 1} do
8:    S is the set of accepted messages value(1)_u.
9:    if |S| ≥ i and value(1)_p ∈ S then
10:      broadcast S ∪ {value(1)_v}
11:      decide 1 and terminate
12:   end if
13: end for
14: decide 0 and terminate
```

of which must be from a correct node since there are only f byzantine nodes. In this case some other correct node u' must have decided on 1 in some round $j < i$, which contradicts i's minimality; hence this case cannot happen.

Finally, if no correct node decides on 1 by the end of round $f + 1$, then all correct nodes will decide on 0. □

Remarks:

- The algorithm only takes $f + 1$ rounds, which is optimal as described in Theorem 11.20.
- Using signatures, Algorithm 25.2 solves consensus for any number of failures! Does this contradict Theorem 11.12? Recall that in the proof of Theorem 11.12 we assumed that a byzantine node can distribute contradictory information about its own input. If messages are signed, correct nodes can detect such behavior – a node u signing two contradicting messages proves to all nodes that node u is byzantine.
- Does Algorithm 25.2 satisfy any of the validity conditions introduced in Section 11.1? No! A byzantine primary can dictate the decision value. Can we modify the algorithm such that the correct-input validity condition is satisfied? Yes! We can run the algorithm in parallel for $2f + 1$ primary nodes. Either 0 or 1 will occur at least $f + 1$ times, which means that one correct process had to have this value in the first place. In this case, we can only handle $f < \frac{n}{2}$ byzantine nodes.
- If the primary is a correct node, Algorithm 25.2 only needs two rounds! Can we make it work with arbitrary inputs? Also, relying on synchrony limits the practicality of the protocol. What if messages can be lost or the system is asynchronous?

25.2 Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (PBFT) is one of the first and perhaps the most instructive protocol for achieving state replication among nodes as in Definition 7.8 with byzantine nodes in an asynchronous network. We present a very simple version of it without any optimizations.

Definition 25.4 (System Model)**.** *There are $n = 3f+1$ nodes and an unbounded number of clients. There are at most f byzantine nodes, and clients can be byzantine as well. The network is asynchronous, and messages have variable delay and can get lost. Clients send requests that correct nodes have to order to achieve state replication.*

The ideas behind PBFT can roughly be summarized as follows:

- Signatures guarantee that every node can determine which node/client generated any given message.
- At any given time, every node will consider one designated node to be the *primary* and the other nodes to be *backups*. Since we are in the variable delay model, requests can arrive at the nodes in different orders. While a primary remains in charge (this timespan corresponds to what is called a *view*), it thus has the function of a serializer (cf. Algorithm 7.9).
- If backups detect faulty behavior in the primary, they start a new view and the next node in round-robin order becomes primary. This is called a *view change*.
- After a view change, a correct new primary makes sure that no two correct nodes execute requests in different orders. Exchanging information will enable backups to determine if the new primary acts in a byzantine fashion.

Definition 25.5 (View)**.** *A* ***view*** *is represented locally at each node i by a non-negative integer v (we say i* ***is in view*** *v) that is incremented by one whenever the node changes to a different view.*

Definition 25.6 (Primary; Backups)**.** *A node that is in view v considers node $v \bmod n$ to be the* ***primary*** *and all other nodes to be* ***backups****.*

Definition 25.7 (Sequence Number)**.** *During a view, a node relies on the primary to pick consecutive integers as* ***sequence numbers*** *that function as indices in the global order (cf. Definition 7.8) for the requests that clients send.*

Remarks:

- All nodes start out in view 0 and can potentially be in different views (i.e. have different local values for v) at any given time.
- The protocol will guarantee that once a correct node has executed a request r with sequence number s, then no correct node will execute any $r' \neq r$ with sequence number s, not unlike Lemma 7.14.
- Correct primaries choose sequence numbers such that they are *dense*, i.e. if a correct primary proposed s as the sequence number for the last request, then it will use $s+1$ for the next request that it proposes.

- Before a node can safely execute a request r with a sequence number s, it will wait until it knows that the decision to execute r with s has been reached and is widely known.
- Informally, nodes will collect confirmation messages by sets of at least $2f + 1$ nodes to guarantee that that information is sufficiently widely distributed.

Definition 25.8 (Accepted Messages)**.** *A correct node that is in view v will only* ***accept messages*** *that it can authenticate, that follow the specification of the protocol, whose components can be validated in the same way, and that also belong to view v.*

Lemma 25.9 (2f+1 Quorum Intersection)**.** *Let S_1 with $|S_1| \geq 2f + 1$ and S_2 with $|S_2| \geq 2f + 1$ each be sets of nodes. Then there exists a correct node in $S_1 \cap S_2$.*

Proof. Let S_1, S_2 each be sets of at least $2f + 1$ nodes. There are $3f + 1$ nodes in total, thus due to the pigeonhole principle the intersection $S_1 \cap S_2$ contains at least $f + 1$ nodes. Since there are at most f faulty nodes, $S_1 \cap S_2$ contains at least 1 correct node. □

25.3 PBFT: Agreement Protocol

First we describe how PBFT achieves agreement on a unique order of requests within a view.

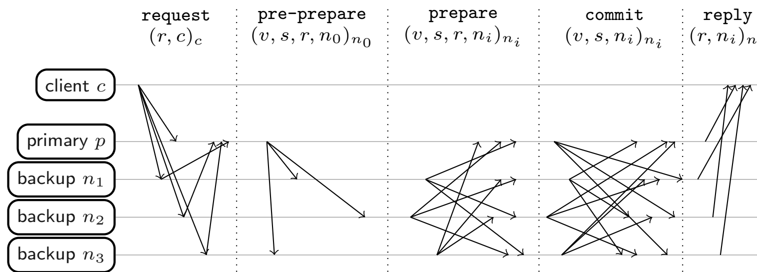


Figure 25.10: The agreement protocol used in PBFT for processing a client request, exemplified for a system with 4 nodes. Node n_0 is the primary in current view v. Time runs from left to right. Messages sent at the same time need not arrive at the same time.

Remarks:

- Figure 25.10 shows how the nodes come to an agreement on a sequence number for a client request. Informally, the protocal has these three steps:

  1. The primary sends a `pre-prepare`-message to all backups, informing them that he wants to execute that request with the sequence number specified in the message.

2. Backups send `prepare`-messages to all nodes, informing them that they agree with that suggestion.
3. All nodes send `commit`-messages to all nodes, informing everyone that they have committed to execute the request with that sequence number. They execute the request and inform the client.

- Figure 25.10 shows that all nodes can start each phase at different times.
- To make sure byzantine nodes cannot force the execution of a request, every node waits for a certain number of `prepare`- and `commit`-messages with the correct content before executing the request.
- Definitions 25.11, 25.14, 25.16 specify the agreement protocol formally. Backups run Phases 1 and 2 concurrently.

Definition 25.11 (PBFT Agreement Protocol Phase 1; Pre-Prepared Primary)**.** *In **phase 1** of the agreement protocol, the nodes execute Algorithm 25.12.*

Algorithm 25.12 PBFT Agreement Protocol: Phase 1

Code for primary p in view v:

1: accept $\texttt{request}(r, c)_c$ that originated from client c
2: pick next sequence number s
3: send $\texttt{pre-prepare}(v, s, r, p)_p$ to all backups

Code for backup b:

4: accept $\texttt{request}(r, c)_c$ from client c
5: relay $\texttt{request}(r, c)_c$ to primary p

Definition 25.13 (Faulty-Timer)**.** *When backup b accepts request r in Algorithm 25.12 Line 4, b starts a local **faulty-timer** (if the timer is not already running) that will only stop once b executes r.*

Remarks:

- If the faulty-timer expires, the backup considers the primary faulty and triggers a view change. We explain the view change protocol in Section 25.4.
- We leave out the details regarding for what timespan to set the faulty-timer as they are an optimization with several trade-offs to consider; the interested reader is advised to consult [CL+99].

Definition 25.14 (PBFT Agreement Protocol Phase 2; Pre-prepared Backups)**.** *In phase 2 of the agreement protocol, every backup b executes Algorithm 25.15. Once it has sent the* `prepare`*-message, b has **pre-prepared** r for (v, s).*

Algorithm 25.15 PBFT Agreement Protocol: Phase 2

Code for backup b in view v:

1: accept `pre-prepare`$(v, s, r, p)_p$
2: **if** p is primary of view v and b has not yet accepted a `pre-prepare`-message for (v, s) and some $r' \neq r$ **then**
3: send `prepare`$(v, s, r, b)_b$ to all nodes
4: **end if**

Definition 25.16 (PBFT Agreement Protocol Phase 3; Prepared-Certificate)**.** *A node n_i that has pre-prepared a request executes Algorithm 25.17. It waits until it has collected $2f$* `prepare`*-messages (including n_i's own, if it is a backup) in Line 1. Together with the* `pre-prepare`*-message for (v, s, r), they form a **prepared-certificate**.*

Algorithm 25.17 PBFT Agreement Protocol: Phase 3

Code for node n_i that has pre-prepared r for (v, s):

1: wait until $2f$ `prepare`-messages matching (v, s, r) have been accepted (including n_i's own message, if it is a backup)
2: send `commit`$(v, s, n_i)_{n_i}$ to all nodes
3: wait until $2f + 1$ `commit`-messages (including n_i's own) matching (v, s) have been accepted
4: execute request r once all requests with lower sequence numbers have been executed
5: send `reply`$(r, n_i)_{n_i}$ to client

Remarks:

- Note that the agreement protocol can run for multiple requests in parallel. Since we are in the variable delay model and messages can arrive out of order, we thus have to wait in Algorithm 25.17 Line 4 until a request has been executed for all previous sequence numbers.

- The client only considers the request to have been processed once it received $f + 1$ `reply`-messages sent by the nodes in Algorithm 25.17 Line 5. Since a correct node only sends a `reply`-message once it executed the request, with $f + 1$ `reply`-messages the client can be certain that the request was executed by a correct node.

- We will see in Section 25.4 that PBFT guarantees that once a single correct node executed the request, then all correct nodes will never execute a different request with the same sequence number. Thus, knowing that a single correct node executed a request is enough for the client.

- If the client does not receive at least $f+1$ `reply`-messages fast enough, it can start over by resending the request to initiate Algorithm 25.12 again. To prevent correct nodes that already executed the request

from executing it a second time, clients can mark their requests with some kind of unique identifiers like a local timestamp. Correct nodes can then react to each request that is resent by a client as required by PBFT, and they can decide if they still need to execute a given request or have already done so before.

Lemma 25.18 (PBFT: Unique Sequence Numbers within View)**.** *If a node gathers a prepared-certificate for (v, s, r), then no node can gather a prepared-certificate for (v, s, r') with $r' \neq r$.*

Proof. Assume two (not necessarily distinct) nodes gather prepared-certificates for (v, s, r) and (v, s, r'). Since a prepared-certificate contains $2f + 1$ messages, a correct node sent a `pre-prepare`- or `prepare`-message for each of (v, s, r) and (v, s, r') due to Lemma 25.9. A correct primary only sends a single `pre-prepare`-message for each (v, s), see Algorithm 25.12 Lines 2 and 3. A correct backup only sends a single `prepare`-message for each (v, s), see Algorithm 25.15 Lines 2 and 3. Thus, $r' = r$. □

Remarks:

- Due to Lemma 25.18, once a node has a prepared-certificate for (v, s, r), no correct node will execute some $r' \neq r$ with sequence number s during view v because correct nodes wait for a prepared-certificate before executing a request (cf. Algorithm 25.17).
- However, that is not yet enough to make sure that no $r' \neq r$ will be executed by a correct node with sequence number s during some later view $v' > v$. How can we make sure that that does not happen?

25.4 PBFT: View Change Protocol

If the primary is faulty, the system has to perform a view change to move to the next primary so the system can make progress. Nodes use their faulty-timer (and only that!) to decide whether they consider the primary to be faulty (cf. Definition 25.13).

Remarks:

- During a view change, the protocol has to guarantee that requests that have already been executed by some correct nodes will not be executed with the different sequence numbers by other correct nodes.
- How can we guarantee that this happens?

Definition 25.19 (PBFT: View Change Protocol)**.** *In the view change protocol, a node whose faulty-timer has expired enters the* ***view change phase*** *by running Algorithm 25.22. During the* ***new view phase*** *(which all nodes continually listen for), the primary of the next view runs Algorithm 25.23 while all other nodes run Algorithm 25.24.*

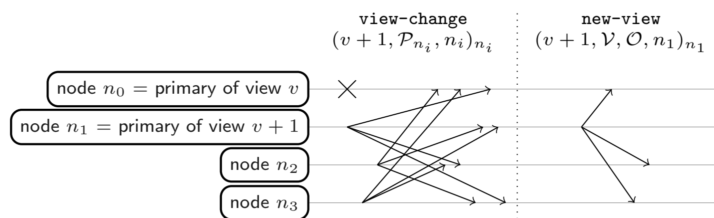


Figure 25.20: The view change protocol used in PBFT. Node n_0 is the primary of current view v, node n_1 the primary of view $v+1$. Once backups consider n_0 to be faulty, they start the view change protocol (cf. Algorithms 25.22, 25.23, 25.24). The X signifies that n_0 is faulty.

Remarks:

- The idea behind the view change protocol is this: during the view change protocol, the new primary gathers prepared-certificates from $2f+1$ nodes, so for every request that some correct node executed, the new primary will have at least one prepared-certificate.
- After gathering that information, the primary distributes it and tells all backups which requests need to be to executed with which sequence numbers.
- Backups can check whether the new primary makes the decisions required by the protocol, and if it does not, then the new primary must be byzantine and the backups can directly move to the next view change.

Definition 25.21 (New-View-Certificate)**.** $2f+1$ `view-change`-*messages for the same view* v *form a* ***new-view-certificate****.*

Algorithm 25.22 PBFT View Change Protocol: View Change Phase

Code for backup b *in view* v *whose faulty-timer has expired:*

1: stop accepting `pre-prepare`/`prepare`/`commit`-messages for v
2: let $\mathcal{P}_b$ be the set of all prepared-certificates that b has collected since the system was started
3: send `view-change`$(v+1, \mathcal{P}_b, b)_b$ to all nodes

Remarks:

- It is possible that $\mathcal{V}$ contains a prepared-certificate for a sequence number s while it does not contain one for some sequence number $s' < s$. For each such sequence number s', we fill up $\mathcal{O}$ in Algorithm 25.23 Line 4 with `null`-requests, i.e. requests that backups understand to mean “do not do anything here”.

Algorithm 25.23 PBFT View Change Protocol: New View Phase - Primary

Code for primary p of view $v+1$:

1: accept $2f+1$ `view-change`-messages (including possibly p's own) in a set $\mathcal{V}$ (this is the *new-view-certificate*)
2: let $\mathcal{O}$ be a set of `pre-prepare`$(v+1, s, r, p)_p$ for all pairs (s, r) where at least one prepared-certificate for (s, r) exists in $\mathcal{V}$
3: let $s^{\mathcal{V}}_{max}$ be the highest sequence number for which $\mathcal{O}$ contains a `pre-prepare`-message
4: add to $\mathcal{O}$ a message `pre-prepare`$(v+1, s', \texttt{null}, p)_p$ for every sequence number $s' < s^{\mathcal{V}}_{max}$ for which $\mathcal{O}$ does not contain a `pre-prepare`-message
5: send `new-view`$(v+1, \mathcal{V}, \mathcal{O}, p)_p$ to all nodes
6: start processing requests for view $v+1$ according to Algorithm 25.12 starting from sequence number $s^{\mathcal{V}}_{max}+1$

Algorithm 25.24 PBFT View Change Protocol: New View Phase - Backup

Code for backup b of view $v+1$ if b's local view is $v' < v+1$:

1: accept `new-view`$(v+1, \mathcal{V}, \mathcal{O}, p)_p$
2: stop accepting `pre-prepare-`/`prepare-`/`commit`-messages for v`// in case` b `has not run Algorithm 25.22 for` $v+1$ `yet`
3: set local view to $v+1$
4: **if** p is primary of $v+1$ **then**
5:   **if** $\mathcal{O}$ was correctly constructed from $\mathcal{V}$ according to Algorithm 25.23 Lines 2 and 4 **then**
6:     respond to all `pre-prepare`-messages in $\mathcal{O}$ as in the agreement protocol, starting from Algorithm 25.15
7:     start accepting messages for view $v+1$
8:   **else**
9:     trigger view change to $v+2$ using Algorithm 25.22
10:   **end if**
11: **end if**

Theorem 25.25 (PBFT:Unique Sequence Numbers Across Views)**.** *Together, the PBFT agreement protocol and the PBFT view change protocol guarantee that if a correct node executes a request r in view v with sequence number s, then no correct node will execute any $r' \neq r$ with sequence number s in any view $v' \geq v$.*

Proof. If no view change takes place, then Lemma 25.18 proves the statement. Therefore, assume that a view change takes place, and consider view $v' > v$.

We will show that if some correct node executed a request r with sequence number s during v, then a correct primary will send a `pre-prepare`-message matching (v', s, r) in the $\mathcal{O}$-component of the `new-view`$(v', \mathcal{V}, \mathcal{O}, p)$-message. This guarantees that no correct node will be able to collect a prepared-certificate for s and a different $r' \neq r$.

Consider the new-view-certificate $\mathcal{V}$ (see Algorithm 25.23 Line 1). If any correct node executed request r with sequence number s, then due to Algo-

rithm 25.17 Line 3, there is a set R_1 of at least $2f + 1$ nodes that sent a `commit`-message matching (s, r), and thus the correct nodes in R_1 all collected a prepared-certificate in Algorithm 25.17 Line 1.

The new-view-certificate contains `view-change`-messages from a set R_2 of $2f + 1$ nodes. Thus according to Lemma 25.9, there is at least one correct node $c_r \in R_1 \cap R_2$ that both collected a prepared-certificate matching (s, r) and whose `view-change`-message is contained in $\mathcal{V}$.

Therefore, if some correct node executed r with sequence number s, then $\mathcal{V}$ contains a prepared-certificate matching (s, r) from c_r. Thus, if some correct node executed r with sequence number s, then due to Algorithm 25.23 Line 2, a correct primary p sends a `new-view`$(v', \mathcal{V}, \mathcal{O}, p)$-message where $\mathcal{O}$ contains a `pre-prepare`(v', s, r, p)-message.

Correct backups will enter view v' only if the `new-view`-message for v' contains a valid new-view-certificate $\mathcal{V}$ and if $\mathcal{O}$ was constructed correctly from $\mathcal{V}$, see Algorithm 25.24 Line 5. They will then respond to the messages in $\mathcal{O}$ before they start accepting other `pre-prepare`-messages for v' due to the order of Algorithm 25.24 Lines 6 and 7. Therefore, for the sequence numbers that appear in $\mathcal{O}$, correct backups will only send `prepare`-messages responding to the `pre-prepare`-messages found in $\mathcal{O}$ due to Algorithm 25.15 Lines 2 and 3. This guarantees that in v', for every sequence number s that appears in $\mathcal{O}$, backups can only collect prepared-certificates for the triple (v', s, r) that appears in $\mathcal{O}$.

Together with the above, this proves that if some correct node executed request r with sequence number s in v, then no node will be able to collect a prepared-certificate for some $r' \neq r$ with sequence number s in any view $v' \geq v$, and thus no correct node will execute r' with sequence number s. □

Remarks:

- We have shown that PBFT protocol guarantees safety or nothing bad ever happens, i.e., the correct nodes never disagree on requests that were commited with the same sequence numbers. But, does PBFT also guarantee liveness, i.e., a legitimate client request is eventually committed and receives a reply.

- To prove liveness, we make an additional assumption that message delays are finite and bounded. With infinite message delays in an asynchronous system and even one faulty (byzantine) process, it is impossible to solve consensus with guaranteed termination [FLP85].

- A faulty new primary could delay the system indefinitely by never sending a `new-view`-message. To prevent this, as soon as a node sends its `view-change`-message for $v + 1$, it starts its faulty-timer and stops it once it accepts a `new-view`-message for $v + 1$. If the timer runs out before being stopped, the node triggers another view change.

- However, the timer doubles to trigger the next view change because the message delays might be larger. Eventually, the timer values are larger than the message delays and the messages are received before the timer expires.

- Since at most f consecutive primaries can be faulty, the system makes progress after at most $f+1$ view changes.

- We described a simplified version of PBFT; any practically relevant variant makes adjustments to what we presented. The references found in the chapter notes can be consulted for details that we did not include.

Chapter Notes

PBFT is perhaps the central protocol for asynchronous byzantine state replication. The seminal first publication about it, of which we presented a simplified version, can be found in [CL+99]. The canonical work about most versions of PBFT is Miguel Castro's PhD dissertation [Cas01].

Notice that the sets $\mathcal{P}_b$ in Algorithm 25.22 grow with each view change as the system keeps running since they contain all prepared-certificates that nodes have collected so far. All variants of the protocol found in the literature introduce regular *checkpoints* where nodes agree that enough nodes executed all requests up to a certain sequence number so they can continuously garbage-collect prepared-certificates. We left this out for conciseness.

Remember that all messages are signed. Generating signatures is somewhat pricy, and variants of PBFT exist that use the cheaper, but less powerful Message Authentication Codes (MACs). These variants are more complicated because MACs only provide authentication between the two endpoints of a message and cannot prove to a third party who created a message. An extensive treatment of a variant that uses MACs can be found in [CL02].

Before PBFT, byzantine fault-tolerance was considered impractical, just something academics would be interested in. PBFT changed that as it showed that byzantine fault-tolerance can be practically feasible. As a result, numerous asynchronous byzantine state replication protocols were developed. Other well-known protocols are Q/U [AEMGG+05], HQ [CML+06], and Zyzzyva [KAD+07]. An overview over the relevant literature can be found in [AGK+15].

This chapter was written in collaboration with Georg Bachmeier.

Bibliography

[AEMGG+05] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 59–74. ACM, 2005.

[AGK+15] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4):12, 2015.

[Cas01] Miguel Castro. *Practical Byzantine Fault Tolerance*. Ph.d., MIT, January 2001. Also as Technical Report MIT-LCS-TR-817.

[CL+99] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[CML+06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190. USENIX Association, 2006.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

Chapter 26

Advanced Blockchain

In this chapter we study various advanced blockchain concepts, which are popular in research.

26.1 Selfish Mining

Satoshi Nakamoto suggested that it is rational to be altruistic, e.g., by always attaching newly found block to the longest chain. But is it true?

Definition 26.1 (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 26.2 Selfish Mining

1: Idea: Mine secretly, without immediately publishing newly found blocks
2: Let d_p be the depth of the public blockchain
3: Let d_s be the depth of the secretly mined blockchain
4: **if** a new block b_p is published, i.e., d_p has increased by 1 **then**
5:    **if** $d_p > d_s$ **then**
6:       Start mining on that newly published block b_p
7:    **else if** $d_p = d_s$ **then**
8:       Publish secretly mined block b_s
9:       Mine on b_s and publish newly found block immediately
10:    **else if** $d_p = d_s - 1$ **then**
11:       Publish both secretly mined blocks
12:    **end if**
13: **end if**

Remarks:

- If the selfish miner is more than two blocks ahead, the original research suggested to always answer a newly published block by releasing the oldest unpublished block. The idea is that honest miners will then split their mining power between these two blocks. However, what matters is how long it takes the honest miners to find the next block,

to extend the public blockchain. This time does not change whether the honest miners split their efforts or not. Hence the case $d_p < d_s - 1$ is not needed in Algorithm 26.2.

Theorem 26.3 (Selfish Mining). *It may be rational to mine selfishly, depending on two parameters α and γ, where α is the ratio of the mining power of the selfish miner, and γ is the share of the altruistic mining power the selfish miner can reach in the network if the selfish miner publishes a block right after seeing a newly published block. Precisely, the selfish miner share is*

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

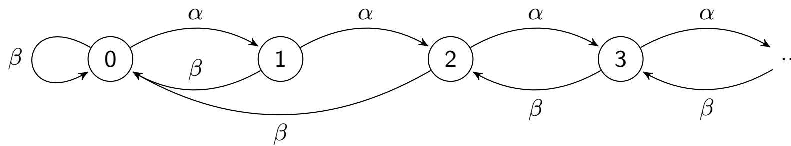


Figure 26.4: Each state of the Markov chain represents how many blocks the selfish miner is ahead, i.e., $d_s - d_p$. In each state, the selfish miner finds a block with probability α, and the honest miners find a block with probability $\beta = 1 - \alpha$. The interesting cases are the "irregular" β arrow from state 2 to state 0, and the β arrow from state 1 to state 0 as it will include three subcases.

Proof. We model the current state of the system with a Markov chain, see Figure 26.4.

We can solve the following Markov chain equations to figure out the probability of each state in the stationary distribution:

$$\begin{aligned} p_1 &= \alpha p_0 \\ \beta p_{i+1} &= \alpha p_i, \text{ for all } i > 1 \\ \text{and } 1 &= \sum_i p_i. \end{aligned}$$

Using $\rho = \alpha/\beta$, we express all terms of above sum with p_1:

$$1 = \frac{p_1}{\alpha} + p_1 \sum_{i \geq 0} \rho^i = \frac{p_1}{\alpha} + \frac{p_1}{1-\rho}, \text{ hence } p_1 = \frac{2\alpha^2 - \alpha}{\alpha^2 + \alpha - 1}.$$

Each state has an outgoing arrow with probability β. If this arrow is taken, one or two blocks (depending on the state) are attached that will eventually end up in the main chain of the blockchain. In state 0 (if arrow β is taken), the honest miners attach a block. In all states i with $i > 2$, the selfish miner eventually attaches a block. In state 2, the selfish miner directly attaches 2 blocks because of Line 11 in Algorithm 26.2.

State 1 in Line 8 is interesting. The selfish miner secretly was 1 block ahead, but now (after taking the β arrow) the honest miners are attaching a competing block. We have a race who attaches the next block, and where. There are three possibilities:

- Either the selfish miner manages to attach another block to its own block, giving 2 blocks to the selfish miner. This happens with probability α.
- Or the honest miners attach a block (with probability β) to their previous honest block (with probability $1-\gamma$). This gives 2 blocks to the honest miners, with total probability $\beta(1-\gamma)$.
- Or the honest miners attach a block to the selfish block, giving 1 block to each side, with probability $\beta\gamma$.

The blockchain process is just a biased random walk through these states. Since blocks are attached whenever we have an outgoing β arrow, the total number of blocks being attached per state is simply $1+p_1+p_2$ (all states attach a single block, except states 1 and 2 which attach 2 blocks each).

As argued above, of these blocks, $1-p_0+p_2+\alpha p_1-\beta(1-\gamma)p_1$ are blocks by the selfish miner, i.e., the ratio of selfish blocks in the blockchain is

$$\frac{1-p_0+p_2+\alpha p_1-\beta(1-\gamma)p_1}{1+p_1+p_2}.$$

□

Remarks:

- If the miner is honest (altruistic), then a miner with computational share α should expect to find an α fraction of the blocks. For some values of α and γ the ratio of Theorem 26.3 is higher than α.
- In particular, if $\gamma=0$ (the selfish miner only wins a race in Line 8 if it manages to mine 2 blocks in a row), the break even of selfish mining happens at $\alpha=1/3$.
- If $\gamma=1/2$ (the selfish miner learns about honest blocks very quickly and manages to convince half of the honest miners to mine on the selfish block instead of the slightly earlier published honest block), already $\alpha=1/4$ is enough to have a higher share in expectation.
- And if $\gamma=1$ (the selfish miner controls the network, and can hide any honest block until the selfish block is published) any $\alpha>0$ justifies selfish mining.

26.2 DAG-Blockchain

Traditional Bitcoin-like blockchains require mining blocks sequentially. Sometimes effort is wasted if two blocks happen to be mined at roughly the same time, as one of these two blocks is going to become obsolete. DAG-blockchains (where DAG stands for directed acyclic graph) try to prevent such wasted blocks. They allow for faster block production, as forks are less of a problem.

Definition 26.5 (DAG-blockchain). *In a DAG-blockchain the genesis block does not reference other blocks. Every other block has at least one (and possibly multiple references) to previous blocks.*

Definition 26.6 (DAG-Relations)**.** *Block p is a dag-parent of block b if block b references (includes a hash) to p. Likewise b is a dag-child of p. Block a is a dag-ancestor of block b, if a is b's dag-parent, dag-grandparent (dag-parent of dag-parent), dag-grandgrandparent, and so on. Likewise b is a's dag-descendant.*

Theorem 26.7. *There are no cycles in a DAG-blockchain.*

Proof. A block b includes its dag-parents' hashes. These dag-parents themselves include the hashes of their dag-parents, etc. To get a cycle of references, some of b's dag-ancestors must include b's hash, which is cryptographically infeasible. □

Definition 26.8 (Tree-Relations)**.** *We are going to implicitly mark some of the references in the DAG of blocks, such that these marked references form a tree, directed towards the genesis block. For every non-genesis block one edge to one of its dag-parents is marked. We use the prefix "tree" to denote these special relations. The marked edge is between tree-parent and tree-child. The tree also defines tree-ancestors and tree-descendants.*

Remarks:

- In other words, every tree-something is also a dag-something, but not necessarily vice versa.
- Blocks do not specify who is their tree-parent, or the order of their dag-parents. Instead, tree-parents are implicitly defined as follows.

Definition 26.9 (DAG Weight)**.** *The weight of a dag-ancestor block a with respect to a block b is defined as the number of tree-descendants of a in the set of dag-ancestors of b. If two blocks a and a' have the same weight, we use the hashes of a and a' to break ties.*

Definition 26.10 (Parent Order)**.** *Let x and y be any pair of dag-parents of b, and z be the lowest common tree-ancestor of x and y. x' and y' are the tree-children of z that are tree-ancestors of x and y respectively. If x' has a higher weight than y', then block b orders dag-parent x before y.*

Definition 26.11 (Tree-Parent)**.** *The tree-parent of b is the first dag-parent in b's parent order.*

Remarks:

- Now we can totally order all the blocks in the DAG-Blockchain.

Theorem 26.13. *Let p be the tree-parent of b. The order of blocks $<_b$ computed by Algorithm 26.12 extends the order $<_p$ by appending some blocks.*

Proof. Block p is the first dag-parent of b, so in the first iteration of the loop, we have $<_b = <_p$. Further modifications of $<_b$ consist only of appending more blocks to $<_b$, ending with block b itself. □

Algorithm 26.12 DAG-Blockchain Ordering

1: We totally order all dag-ancestors of block b as $<_b$ as follows:
2: Initialize $<_b$ as empty
3: **for** all dag-parents p of b, in their parent order **do**
4: Compute $<_p$ (recursively)
5: Remove from $<_p$ any blocks already included in $<_b$
6: Append $<_p$ at the end of $<_b$
7: **end for**
8: Append block b at the end of $<_b$

Remarks:

- Note that b is appended to the order only after ordering all its dag-ancestors. The genesis block is the only block where the recursion will stop, so the genesis block is always first in the total order.
- By Theorem 26.13 tree-children extend the order of their tree-parent, so appending blocks to the DAG preserves the previous order and new blocks are appended at the end.

Definition 26.14 (Transaction Order)**.** *Transactions in each block are ordered by the miner of the block. Since blocks themselves are ordered, all transactions are ordered. If two transactions contradict each other (e.g. they try to spend the same money twice), the first transaction in the total order is considered executed, while the second transaction is simply ignored (or possibly punished).*

Remarks:

- Ethereum allows blocks to not only have a parent, but also up to two "uncles" (childless blocks). In contrast to above description, blocks must specify the main parent.
- In Ethereum, new blocks are mined approximately every 15 seconds (as opposed to 10 minutes in Bitcoin). New blocks being generated in such rapid succession leads to a lot of childless blocks. Uncles have been introduced to not "waste" those blocks.
- In Ethereum, the original uncle-miners get 7/8 of the block reward. The miner who references these uncle blocks also gets a small reward. This reward depends on the height-difference of the uncle and the included parent. Also, to be included, the uncle and the current block should have a common ancestor not too far in the past.

26.3 Smart Contracts

Definition 26.15 (Ethereum)**.** *Ethereum is a distributed state machine. Unlike Bitcoin, Ethereum promises to run arbitrary computer programs in a blockchain.*

Remarks:

- 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.
- Like in Bitcoin, 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.

Definition 26.16 (Smart Contract)**.** *Smart contracts are programs deployed on the Ethereum blockchain that have associated storage and can execute arbitrarily complex logic.*

Remarks:

- Smart Contracts are written in higher level programming languages like Solidity, Vyper, etc. and are compiled down to EVM (Ethereum Virtual Machine) bytecode, which is a Turing complete low level programming language.
- Smart contracts 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. With this, many smart contracts can update to a new version.

Definition 26.17 (Account)**.** *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.*

Definition 26.18 (Ethereum Transaction)**.** *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 from the sender to the recipient.*
- *Data: Optional data field, 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, so a high GasPrice will make sure that the transaction is executed more quickly.*

Remarks:

- There are three types of transactions.

Definition 26.19 (Simple Transaction)**.** *A simple transaction in Ethereum transfers some of the native currency, called Wei, from one EOA to another. Higher units of curency are called Szabo, Finney, and Ether, with* 10^{18} *Wei =* 10^6 *Szabo =* 10^3 *Finney = 1 Ether. The data field in a simple transaction is empty.*

Definition 26.20 (Smart Contract Creation Transaction)**.** *A 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.*

Definition 26.21 (Smart Contract Execution Transaction)**.** *A transaction that has a smart contract address in its recipient field and code to execute a specific function of that contract in its data field.*

Remarks:

- Smart Contracts can execute computations, store data, send Ether to other accounts or smart contracts, and invoke other smart contracts.

- Smart contracts can 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. Storage is a key-value store of 256 bit words to 256 bit words. The storage data is persisted in the Ethereum blockchain, like the hard disk of a traditional computer. Memory and stack are for intermediate storage required while running a specific function, similar to RAM and registers of a traditional computer. The read/write gas costs of persistent storage is significantly higher than those of memory and stack.

Definition 26.22 (Gas)**.** *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.*

Remarks:

- As Ethereum contracts are programs (with loops, function calls, and recursions), end users need to pay more gas for more computations. In particular, smart contracts might call another smart contract as a subroutine, and StartGas must include enough gas to pay for all these function calls invoked by the transaction.

- The product of StartGas and GasPrice is the maximum cost of the entire transaction.

- 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.

Definition 26.23 (Block)**.** *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)*

26.4 Payment Hubs

How to we enable many parties to send payments to each other efficiently?

Definition 26.24 (Payment Hub)**.** *Multiple parties can send payments to each other by means of a payment hub.*

Remarks:

- While we could always call the smart contract to transfer money between users that joined the hub, every smart contract call costs as it involves the blockchain. Rather, we want a frugal system with just few blockchain transactions.

Definition 26.25 (Smart Contract Hub)**.** *A smart contract hub is a payment hub that is realized by a smart contract on a blockchain and an off-chain server. The smart contract and the server together enable off-chain payments between users that joined the hub.*

Algorithm 26.26 Smart Contract Hub

1: Users join the hub by depositing some native currency of the blockchain into the smart contract
2: Funds of all participants are maintained together as a fungible pool in the smart contract
3: Time is divided into epochs: in each epoch users can send each other payment transactions through the server
4: The server does the bookkeeping of who has paid how much to whom during the epoch
5: At the end of the epoch, the server aggregates all balances into a commitment, which is sent to the smart contract
6: Also at the end of the epoch, the server sends a proof to each user, informing about the current account balance
7: Each user can verify that its balance is correct; if not the user can call the smart contract with its proof to get its money back

Remarks:

- The smart contract lives forever, but the server can disappear anytime. If it does, nodes can show their recent balance proofs to the smart contract and withdraw their balances.
- The server can be scaled to in terms of latency and number of users. The smart contract does not need to scale as it only needs to just accept one commitment per epoch.
- In case the server disappears, the smart contract will be flooded with withdrawal requests, and could be subject to delays based on the delays of the underlying blockchain.

26.5 Proof-of-Stake

Almost all of the energy consumption of permissionless (everybody can participate) blockchains is wasted because of proof-of-work. Proof-of-stake avoids these wasteful computations, without going all the way to permissioned (the participating nodes are known a priori) systems such as Paxos or PBFT.

Definition 26.27 (Proof-of-stake). *Proof-of-work awards block rewards to the lucky miner that solved a cryptopuzzle. In contrast, proof-of-stake awards block rewards proportionally to the economic stake in the system.*

Remarks:

- Literally, "the rich get richer".
- Ethereum is expected to move to proof-of-stake eventually.
- There are multiple flavors of proof-of-stake algorithms.

Definition 26.28 (Chain based proof-of-stake). *Accounts hold lottery tickets according to their stake. The lottery is pseudo-random, in the sense that hash functions computed on the state of the blockchain will select which account is winning. The winning account can extend the longest chain by a block, and earn the block reward.*

Remarks:

- It gets tricky if the actual winner of the lottery does not produce a block in time, or some nodes do not see this block in time. This is why some suggested proof-of-stake systems add a voting phase.

Definition 26.29 (BFT based proof-of-stake). *The lottery winner only gets to propose a block to be added to the blockchain. A committee then votes (yes, byzantine fault tolerance) whether to accept that block into the blockchain. If no agreement is reached, this process is repeated.*

Remarks:

- Proof-of-stake can be attacked in various ways. Let us discuss the two most prominent attacks.

- Most importantly, there is the “nothing at stake” attack: In blockchains, forks occur naturally. In proof-of-work, a fork is resolved because every miner has to choose which blockchain fork to extend, as it does not pay off to mine on a hopeless fork. Eventually, some chain will end up with more miners, and that chain is considered to be the real blockchain, whereas other (childless) blocks are just not being extended. In a proof-of-stake system, a user can trivially extend all prongs of a fork. As generating a block costs nothing, the miner has no incentive to not extend all the prongs of the fork. This results in a situation with more and more forks, and no canonical order of transactions. If there is a double-spend attack, there is no way to tell which blockchain is valid, as all blockchains are the same length (all miners are extending all forks). It can be argued that honest miners, who want to preserve the value of the network, will extend the first prong of the fork that they see. But that leaves room for a dishonest miner to double spend by moving their mining opportunity to the appropriate fork at the appropriate time.

- Long range attack: As there are no physical resources being used to produce blocks in a proof-of-stake system, nothing prevents a bad player from creating an alternate blockchain starting at the genesis block, and make it longer than the canonical blockchain. New nodes may have difficulties to determine which blockchain is the real established blockchain. In proof-of-work, long range attacks takes an enormous amount of computing power. In proof-of-stake systems, a new node has to check with trusted sources to know what the canonical blockchain is.