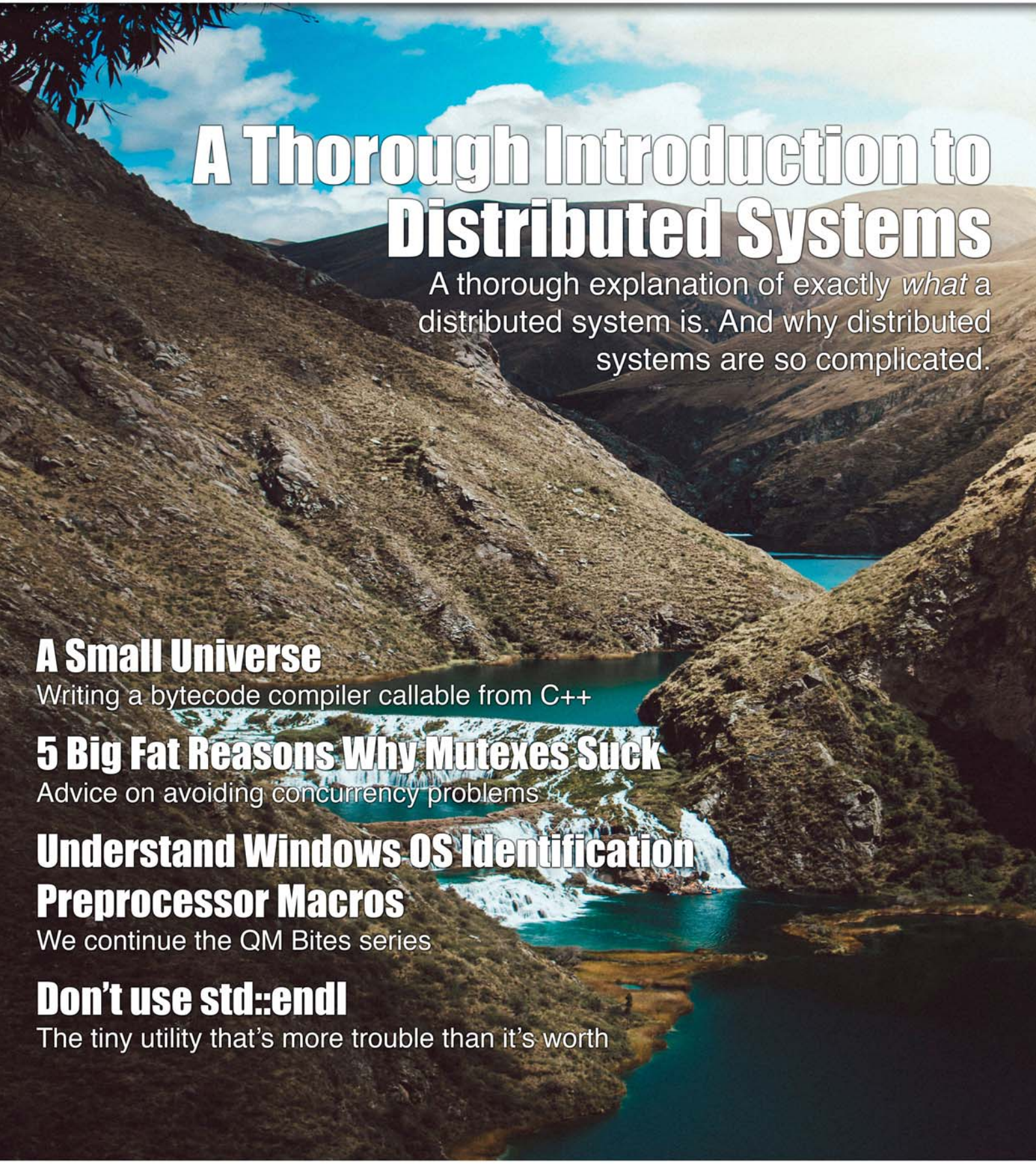
overload 149
FEBRUARY 2019 £4.50



A Thorough Introduction to Distributed Systems

A thorough explanation of exactly *what* a distributed system is. And why distributed systems are so complicated.

A Small Universe

Writing a bytecode compiler callable from C++

5 Big Fat Reasons Why Mutexes Suck

Advice on avoiding concurrency problems

Understand Windows OS Identification Preprocessor Macros

We continue the QM Bites series

Don't use std::endl

The tiny utility that's more trouble than it's worth

A magazine of ACCU
ISSN: 1354-3172

OVERLOAD 149

February 2019

ISSN 1354-3172

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 150 should be submitted by 1st March 2019 and those for Overload 151 by 1st May 2019.

Overload is a publication of the ACCU

For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

The ACCU

The ACCU is an organisation of programmers who care about professionalism in programming. That is, we care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

The articles in this magazine have all been written by ACCU members – by programmers, for programmers – and have been contributed free of charge.

Copyrights and Trade Marks

Rip It Up and Start Again

Some things can be resurrected, others cannot. Frances Buontempo wonders when we need to repent and start over.

I have failed to write an editorial. I have notes dotted all around the house, with potential titles or ideas scribbled down. Many are unreadable. Several are so old now, I've visited the topics from a different angle in previous excuses for not writing an editorial. Now, if I listened to my own musings in our last *Overload* ('Revolution, Restoration and Revival', [Buontempo18]), I could try to revive some of these. I didn't. I decided to tidy the house, starting with making a new TODO list, subsequently running out of paper because I remembered several other things that needed to get done. I should throw all my old notes out to give myself a fresh start for a new year. Restoring or reviving something old is one way to go. Other times, the best thing to do is rip it up and start again.

Take a previous talk or an old code base: sometimes you can refactor it slightly, sometimes you can't. A complete rewrite might be the way to go. In order to refactor, a code base requires some tests, otherwise you are reworking or changing the code instead and potentially changing its behaviour. There might not be conventional unit tests. Sometimes the best you can do is produce some output in a file and use that as a ground truth to ensure any changes introduced don't cause regressions. Emily Bache ran a 90-minute workshop at the ACCU conference in 2013 on refactoring [Bache13]. The code was a snaky mess of ifs and elses, with no tests. However, it did write lots of log messages, so you could tell if it still was functionally identical after making changes. This made it possible to massage the code into a state where you could add a new feature. Watching the number of lines decrease, then the lines, scattered with ands and ors, get shorter was wonderful. Refactoring is a skill I need to work on. I suspect many people don't really understand what it means. I've heard people say they don't have time to refactor and know they are leaving lots of technical debt behind. Sometimes the same people have gatekeeper-style code reviews. These can work as a place to discuss potential refactoring, and often do. I've not often seen a code review out and out reject the proffered solution and demand a complete rewrite. I would hope a team would talk to each other earlier, and realise they are about to walk into a brick wall a long time before a huge patch file needs reviewing. By that point, you have not only lost several days of work, but it's hard to let go of something you've spent a long time on and start over.

Circumstances can force a new start on a project. Sometimes it is a deliberate choice, such as time constraints pushing you into adopting a Plan B. Once or twice, a machine crashed and I lost my work. Another reason to do small commits often. However, I have found that when I do something again from scratch, it is often quicker than my first attempt. I remembered some dead ends I had explored before, or which order to install dependencies in, or an edge case or two that needed a test. I'm not suggesting that destroying your computer is the best way to discover if you are improving. Nonetheless, circumstances sometimes push you to rework or rebuild something. Trying to develop a sense of when you are digging a deep hole for yourself and need to form a Plan B, regardless of outside pressures, is an important skill. Keeping your eye on the prize, by holding in mind what you are trying to achieve, can pull you back from the edge. Being aware of options, so you can unwind your current attempts and try a different method matters. The same goes for trying to mentor or teach others. If one way of explaining things doesn't work, just saying it louder, more emphatically and waving your hands around probably won't work. Try diagrams instead. Or some code. Or going for a coffee and moving cups and sugar bowls around. Or get them to explain you what they understand. Shut up and listen.

Now, you might be faced with a challenge that is similar to an algorithm or method you have previously implemented. The promise of object oriented programming suggested code reuse. However, we all know that adding a monster code base as a library to a new project in order to use one or two utility functions is not a sensible plan. At the expense of contradicting my previous cry for a 5p copy-and-paste tax [Buontempo15], it might be wise to copy and paste the functions you need into a new project. You could carve them out into a shared library, of course. But if your needs differ slightly, don't be shy to look at something you have previously built, and start over, using that for inspiration. In fact, "Evidence suggests that cut 'n' paste is a nearly essential, and nearly universal, practice when new framework applications are constructed, be they for GUI code," [Coplien05]. The article goes on to point out that a copy provides a "safe sandbox" which cannot disrupt production code "for a time". In some circumstances, this will work like branching and need merging back. For a new project, however, you may have no intention of merging back; instead you are revisiting old code, and re-writing it is a potential learning experience. Any copy can be shallow or deep. Copy-and-paste code is a deep copy or a clone. Two, initially identical, functions or classes exist afterwards. Cut-and-paste, in contrast, simply moves the code. This may happen in order to refer to it from different places, either in a code base, or across code bases. Can you shallow-copy code? I suspect some meta-programming, or something functional would work, but I can't dream up a sensible use case here. Imagine if you could reference count all your attempts at Fizz Buzz and so on, and garbage collect once in a while, without your computer dying to enforce the code deletion. Various tools for automatic code duplication detection exist. I've never used one… well, that's not strictly speaking true. I think an expensive enterprise tool was being run on a CI box where I worked a while ago that claimed to spot code duplication but I couldn't make sense of the output it gave. Their docs say, "A developer has everything at hand to take ownership of the

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD technically in Chemical Engineering, but mainly programming and learning about AI and data mining – and she's written a a book on machine learning (https://pragprog.com/book/fbmach/genetic-algorithms-and-machine-learning-for-programmers). She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

quality of his code." Glad to see careless use of pronouns is alive and well. I digress.

Of course, if you try to use old code you might run into issues. An older code base might use a library. Only newer versions may be available, so you can run into compatibility issues. Or in my case, you can't figure out which version you were using because your machine died and your code repo wasn't entirely helpful. Beyond libraries, languages themselves evolve and change. C++ has a veritable history of trying to avoid breaking backwards compatibility. Nonetheless, the keyword has been repurposed, and many features, such as `auto_ptr` have now been deprecated. The new meaning of auto has theoretically stopped some older code working, leaving it needing a rewrite. To be fair, a reviewer pointed out that a couple of the larger companies checked for use in their code bases when the word was originally repurposed and found almost no instances outside compiler test suite. If you are using this, you need to rip it up and start again. Why are interfaces and methods deprecated? "The deprecation period gives people a chance to change their code before the compiler removes it outright", according to one stack exchange answer [StackExchange]. Where features have been deprecated, you have been warned, but can limp along keeping things as they are. For now. What happens if you are trying to migrate a Python 2 codebase to Python 3? Tools exist to help with the upgrade [Python], but they might not catch changes in libraries. Feed our esteemed *CVu* editor a pint and ask him about numpy and csv files. But only if you have a whole evening free. The next step after deprecation is obsolescence. Whether the old API or technology ever really goes away is another matter.

OK, so we can muse on why deprecation happens. How do you deprecate a method? It is possible to support two APIs in tandem, but that requires careful thought. It can be argued this is better than slapping in a Boolean flag and a new set of inputs to switch the behaviour via the calling code. If you try to switch out a method, you need to be careful not to break existing code. This, in essence, is the Liskov substitution principle. In 'Kissing SOLID goodbye', [Oldwood14] Chris Oldwood notes, "The Liskov Substitution Principle (LSP) is usually 'explained' by quoting directly from it. What it boils down to is the proposition that two types are related if one can be used in place of the other and the code still works exactly as before." A square is not the same as a rectangle. A stack is not the same as a queue. In contrast, version 2 of a function should still behave the same way as version 1, if called with the same inputs. A naïve and misguided explanation of LSP talks about inheritance. Oldwood's article points out switching between containers, supporting the same iterator interface, allows a variation point, leaving the behaviour identical. Common interfaces are one way to introduce changes. Feature toggles are another way to migrate to new behaviour. Be warned: these can get out of hand if you don't retire them quickly.

Various different forces take you to the place of no return, where you do start over. Trying to develop a good sense of when something isn't working is invaluable. It's all too easy to keep on digging. That might work out in the end, but might lead to despair. I notice Wikipedia has a disambiguation page for 'Metanoia' [Wikipedia]. The psychology entry defines it as a psychotic "breakdown" followed by a rebuilding or "healing". Sometimes coding feels that way. I know the word from theology, and Wikipedia describes this sense as "a transformative change of heart", carrying the sense of repentance, atonement and conversion. Perhaps you've written code you feel the need to repent for? I have. What would you do differently if you revisited this code? The 'Breakthrough Ideas' article [Coplien05], referred to earlier, started with a list of questions, one of which was, "If you could delete any idea, technology, or event from computing history, what would it be?" What a splendid question. I'd consider voting for an editorial in each *Overload*, but that might just be me. I'm not sure you can delete an idea. What technology would you delete? OO? VBA? JavaScript? Wizards? Dependency injection frameworks? Anything with the word Enterprise in? What about an event? Recent news about the US joint enterprise defense initiative (JEDI) military cloud request for proposals [Wired]. "This program is truly about increasing the lethality of our department." Skynet anyone? It aims to "inject" AI into data analysis and provide real-time data during operations. What could possibly go wrong? Some things seem like a bad idea from the outset. Others start well but their time comes. Knowing when that time has come is important. Pick something you don't like about your approach to programming and stop it. Find a new way forward.

References

[Bache13] Emily Bache, 'Coding Dojo Challenge-Refactoring', ACCU conference 2013, available at: https://accu.org/index.php/conferences/accu_conference_2013/accu2013_sessions#coding_dojo_challenge-refactoring

[Buontempo15] Frances Buontempo, 'Reduce, reuse, recycle', *Overload* 130, December 2015: https://accu.org/index.php/articles/2179

[Buontempo18] Frances Buontempo, 'Revolution, Restoration and Revival', *Overload* 148, December 2018, https://accu.org/index.php/journals/2591

[Coplien05] James O. Coplien, Brian Foote, Richard P. Gabriel, Dave Thomas, Cristina Lopes, Brian Marick, Bonnie Nardi, Rob Tow, Andrew Hunt and Glen Vanderburg (2005) 'Breakthrough Ideas' in *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ISBN 1-59593-193-7

[Oldwood14] Chris Oldwood, 'Kissing SOLID goodbye', *Overload* 122, August 2014, https://accu.org/index.php/journals/1957

[Python] 2to3: https://docs.python.org/2/library/2to3.html

[StackExchange] 'Deprecation considered harmful?': https://softwareengineering.stackexchange.com/questions/83263/deprecation-considered-harmful

[Wikipedia] Metanoia: https://en.wikipedia.org/wiki/Metanoia

[Wired] https://www.wired.com/story/how-pentagons-move-to-cloud-landed-in-mud/

5 Big Fat Reasons Why Mutexes Suck Big Time

Mutable shared state in multithreaded code is often protected by mutexes. Sergey Ignatchenko reminds us that Re-Actors can avoid many of the problems.

Disclaimer: as usual, the opinions within this article are those of 'No Bugs' Hare, and do not necessarily coincide with the opinions of the translators and *Overload* editors; also, please keep in mind that translation difficulties from Lapine (like those described in [Loganberry04]) might have prevented an exact translation. In addition, the translator and *Overload* expressly disclaim all responsibility from any action or inaction resulting from reading this article.

Amicus Plato, sed magis amica veritas
Plato is my friend, but truth is a better friend
~ Aristotle

Of course, I am perfectly aware that an all-powerful multithreaded inquisition will try to burn me at the stake for publishing this kind of heresy. Still, I do believe in the truth (at least as I understand it), so – trying to keep in line with Aristotle, mentioned in the epigraph – I have no choice other than to try to bring these things (that have been known for ages but conveniently forgotten way too often) to the forefront.

And yet ~~it moves~~ they suck!
~ ~~Galileo Galilei~~ 'No Bugs' Hare

On terminology

Just to be clear: in this article I am not attacking mutexes in a narrow sense, so don't think that a critical section or an OpenMP lock is in any way better; rather, *all* techniques which provide mutual exclusion functionality are equally bad for the purposes of this article. These include, but are not limited to:

- All kinds of mutexes (whether recursive or not, single-process or multi-process, etc. etc.)
- Critical sections (including both Windows ones and OpenMP ones)
- Semaphores used for mutual exclusion
- Java *synchronized* objects.

A bit of history

...in the Beginning there was only The Darkest Darkness, and in the Darkness – Magnetics, which unwound the atoms; spinning, the atom struck the atom, and Pra-Current arose, and with it the First Light... and the stars lit up, the planets cooled, and mini-micro-Pramachines emerged, and from them Pramachines developed, and then, by the Providence of Holy Statistics – Protomachines. They still could not count, they only knew what is two plus two, and nothing more, but then, thanks to the Natural Evolution, somehow they got the hang of it, and the Multi-Stats and Omnistats originated from them, and then there was Pithecanthrobotus and from him our forefather, Automatus Sapiens...
~ Stanislaw Lem, The Cyberiad

In the very beginning, Stone Age programmers were programming directly on the hardware, without any intermediaries. However, by 1956, it was realized that app-level programs need some kind of a helper program between them and hardware, and the first operating system (intended for one single computer!) was written [Wikipedia]; it was the beginning of the Bronze Age of programming.

With the advent of a common OS for multiple computers (OS/360 in the mid-60s), computing progressed into the Iron Age. It was during the Iron Age that threads were first introduced (as 'tasks' in OS/360 in 1967 or so). Still, the dominant approach to programming at the time was via *processes* rather than via *threads*. From our current perspective, the main difference between processes and threads is that with processes we don't normally have shared memory, and have to communicate using more structured IPC means.

By the mid-70s, computing got microprocessors (including my personal favorite, i8080A) and has reached Ancient Times. By the early 80s, microprocessors have become ubiquitous and have started to make inroads into our homes.

By the late 80s, POSIX threads had been developed [McCracken02], and by the early 90s, (originally thread-oriented) WinNT was released. However, threads were *still* seriously unpopular with developers; just as one example, Linux didn't get even a half-decent implementation of POSIX threads until LinuxThreads in 1996 (and a *really good* implementation wasn't there until 2003, *after* the CPUs hit the gigahertz wall – more on this below).

Hardware-wise, in the 90s, there was an explosive growth in the performance of microprocessors; in 1989, the i486 was running at 25MHz and by 2000, Willamette P4 was running at 1.5GHz – this is a whopping 60x increase in frequency in about ten years (and more like a 100x overall performance increase if we count in other optimizations). Some dinosaur hares like me can even remember an interpretation of Moore's Law saying that not only the number of transistors but also CPU frequencies will double every two years; in 2000, Intel promised 10GHz CPUs by 2005 [Shimpi00].

However, after approximately 2002, all this explosive growth in CPU frequencies (and in per-core performance) abruptly slowed down; frequencies even dropped (reaching 4GHz again only after ~10 years with the 4GHz but hugely inefficient Netburst), and per-core performance improvements have slowed down very significantly [Edwards12].

As a result of this sudden stop in GHz growth, Intel gave up on GHz-based marketing, and switched to marketing based on the number of cores. But there was an obstacle to this – the lack of programs able to use multi-core processors. And Intel has started to promote multi-cored programs. There is nothing wrong with multi-coring *per se*, but apparently, the most efficient tactics for going this way turned out to be to say "Hey! You can

'No Bugs' Hare Translated from Lapine by Sergey Ignatchenko using the classic dictionary collated by Richard Adams.

Sergey Ignatchenko has 15+ years of industry experience, including being a co-architect of a stock exchange, and the sole architect of a game with 400K simultaneous players. He currently holds the position of Security Researcher. Sergey can be contacted at sergey@ignatchenko.com

simply take your existing program, and just make a few minor changes (often even called 'annotations'): just insert this magical mutex (critical section, etc.) whenever you have a data race – and your program will be able to utilize all our cores!"

I remember attending an Intel workshop on OpenMP during one of the ACCU conferences in the early 2000s… Marketing-wise, it was probably a success, but OMG, technically it was a disaster. Just two observations from the workshop *supposed to teach us proper multithreading*; first, there was a multithreading bug even in the few hundreds of lines of code they gave us; and if the Intel folks teaching us multithreading cannot write 200 LoC without a bug, what can an Joe Average developer with expertise elsewhere hope for?! However, while the bug was pretty telling *per se*, IMNSHO much more annoying was the following occurrence. When we did run their code, it *was* able to utilize all the cores ("hey, look at taskman – see, we ARE utilizing ALL the four cores!"); however, while all four cores were working like crazy, the wall-clock time of the calculation actually *increased* after we introduced multi-threading (!! – actually, it is quite common when MT is too fine-grained, more on this below). Of course, Intel's job at selling us their CPUs *does* stop as soon as 100% utilization is reached (!), but as a developer who cares about my customers and not about Intel's profits, I have *absolutely zero* reason to use this kind of multithreading (and I contend that the whole premise of *utilizing more cores* is deadly wrong – instead, the whole multi-coring thing is *only* about decreasing *response time* to an incoming request, and there is *absolutely no other reason* to use more than one CPU core to perform a given task).

That's where we were standing during the 2000s – pretty much everybody was writing multithreaded mutex-synced programs, and "hey, our program is multithreaded" started to be used as a selling point(!); as a result, lots of devs with no idea about multithreading started to add threads just for the sake of it. These were the Dark Ages of Multithreaded Inquisition in programming, which caused programs to crash often due to data races, and to perform poorly too. In my personal experience, I'd estimate that at least half of the crashes I personally ran into were due to some kind of a multithreading race; this is not to mention that our esteemed translator has personally found MT bugs in such projects as OpenSSL, most C++ `std::` implementations [Ignatchenko98] of the time, and in a WG21 working paper [NoBugs17]. Worse than that, one of those bugs was reported to manifest itself only once a month on a client site (!). As for performance, one of those faulty `std::` implementations in [Ignatchenko98] exhibited up to a 100x performance degradation compared to a mutex-less fix – even in those use cases when it didn't crash.

To be perfectly honest, even in the Dark Ages, there were people and whole technologies that avoided mutexes (such as Erlang), but sadly, such projects were very few and far between. Fortunately, starting around 2010, a more widespread understanding started to form that mutex-based programs suck, and more and more material started to emerge arguing that mutex-based sync must die – or at least that better alternatives are viable [NoBugs10] [TheGoBlog] [Henney16] [Fowler11b]. I would even say that by 2017, we had already passed the point of no return with regards to mutexes at the app level – and that mutexes will follow the lead of that once ubiquitous and now not really used operator, `goto` (sure, we can still use `goto` in quite a few languages – but we should NOT).

So, why do mutexes suck?

Now, after all the historical ranting, let's start looking at those promised Big Five Reasons for mutexes being, ahem, sub-optimal for the vast majority of tasks out there.

Big fat reason #1: Error-prone beyond belief

The very first reason why mutex-based programs suck big time is that it is extremely easy to write a seemingly-working multithreaded program which actually contains a bad data race. Looking from 50,000 feet, there are two types of data-race-related bugs in mutex-based multithreaded programs: (a) forgotten mutex lock, and (b) deadlock. And while it is possible to write correct mutex-based programs (for (a) you just need to be ultra-hyper-careful, and for (b) meticulously following of some kind of global order of acquiring locks will do the trick), in practice it happens to be an insurmountable task waaaay too often. If even such supposedly expert developers as those of Intel (teaching multithreaded programming, no less!), half a dozen different implementors of the std:: library, and several WG21 members, cannot do it right even for a *relatively small and static piece of code* – what can the rest of us realistically hope for, *especially when working on ever-changing app-level code*?!

BTW, while it is theoretically possible to use tools to enforce safe mutex-locking programming practices, even the best thing I know in this regard (the Rust programming language) only solves the problem partially: even after jumping through all the Rust hoops (and there are lots of them ☹), and even if we manage to avoid any *unsafe{}* code, Rust can only guarantee safety against forgotten mutex locks; as for deadlocks, "Rust considers it 'safe' to… deadlock" [Rustonomicon], so deadlocks are still perfectly possible even within 'safe' Rust code ☹.

Big fat reason #2: Fundamentally untestable

The second Big Fat Problem™ with mutex-based programs is that these data races (both forgotten locks and deadlocks) are fundamentally untestable (well, at least on all the existing hardware). How it works in the real world:

- You write your mutex-based program.
- You test it, and it passes all of your unit tests.

  Some of unit tests may occasionally fail – but when you re-run them, they're fine, so you ignore such occasional failures (which is BTW a Mortgage-Crisis Size Mistake™).
- You deploy it to your servers.
- You run it for months without a hitch.
- You start saying, "Hey, see – it is easy to write perfectly correct mutex-based programs, I did it myself!"
- Then your Big Day comes (Christmas sale, Tournament of the Year, Black Thursday, etc.) – and the bug which was sitting there all this time manifests itself, so your system crashes (and being on your Big Day, this crash comes at the worst possible moment too).
- Bummer.

The root of this problem is that *inter-thread interaction is inherently non-deterministic* (at least with modern hardware). *Each and every run of a multithreaded program is potentially substantially different from a previous one, with thread context switches – as well as inter-core interactions – happening at random times on each run.* As a result, it is perfectly possible that the same program with the same input data which was running ok 99 times in a row, will crash on the 100th run; even worse, crash probabilities are *highly* dependent on the environment (including, but not limited to, specific hardware, load at the time of test, other programs running at the same time, and Acrux rising interrupts from your NIC coming at a specific moment).

This, in turn, means that any test we can possibly run is inherently meaningless ☹ ☹ (as Fowler wrote: "Non-deterministic tests have two problems, firstly they are useless…") [Fowler11a]; in other words – regardless of the amount of testing, we cannot possibly say that we have tested all the possible data-race-related scenarios.

From this very observation, it follows that

We have to prove *the correctness of our mutex-based programs.*

However, as mentioned above, even after jumping through all the hoops of Rust, we cannot really *prove* the correctness of our mutex-based programs (as the potential for deadlocks is still there); in general, automated tools for such *proofs* are not really here (and given the rest of the problems with mutex-based stuff, chances are they will never be written).

In the real world, this problem happens to be *soooo* bad that once upon a time I was even involved in writing a tool which used fibers to *simulate* a preemptive scheduler to ensure that if a bug in a program under test is found during such testing, the bug is at least reproducible (also, it was possible to run an exhaustive test, though this was feasible only for very small programs).

Big fat reason #3: Mental overload → development slow down

In addition to the two first Big Fat Problems discussed above, the third one becomes apparent – *being ultra-careful* and *meticulously following* prerequisites to writing correct code (exacerbated by an inherent inability to test our mutex-based programs) tends to take a toll on the mind of the app-level developer who tries to do it. The cognitive capabilities of our brains are actually very limited, so being occupied with mutexes and their order – which are things *light years away* from the app-level task we're trying to solve – inevitably causes much fewer brain resources to be available for that app-level task we're *really* programming right now.

As discussed in [NoBugs15], thinking *both* about thread sync *and* about the business-level task *at the same time* quickly leads to an explosive growth in number of entities the developer has to keep in mind while programming; this, in turn, leads to exhaustion/exceeding the cognitive capacity of our brains (violating the so-called 7±2 rule, and leading to cognitive overload), which means that either development speed has to suffer *a lot*, or our program will have more bugs (including non-multithreaded ones!), or both. To make things even worse, maintenance of mutex-based code (and we do have to *re-prove* MT correctness *after each and every potentially relevant change*!) is not really feasible for anything more complicated than a 'Hello, mutex!' program.

Big fat reason #4: Poor performance

The fourth reason for mutexes sucking Really Badly™ follows from two observations: (a) whenever we're trying to obtain a lock on an already-locked mutex, we're about to cause an inter-thread context switch, and (b) the cost of an inter-thread context switch is *huge*.

Let's discuss the cost of an inter-thread context switch. In [Li07], it was observed that as soon as we account for cache invalidation costs, the cost of an inter-thread context switch can easily reach a *million* CPU clock cycles; my personal calculations show that on a modern Xeon-class CPU, theoretically cache invalidation can cost up to 30M CPU clock cycles. In practice, my personal observations are much milder, but even the 20–50K CPU clock cycles I observed are bad enough to want to avoid mutexes, at least in seriously contentious scenarios. BTW, IMNSHO my observations are consistent with the practice of spin-locks; just think of it: how bad is the cost of a thread context switch that it's better to be *busy-waiting* for several thousand iterations (with each taking at least 3 CPU cycles to read from L1) just *in the blind hope that we will avoid the thread context switch*? That is, if we're lucky, with a spinlock we'll just incur the penalty of the spinning though not of the context switch, but if we're NOT lucky – we'll incur *both* the cost of the spinning *and* the cost of the context switch; and even with all of this in mind, spinning still makes sense for quite a few use cases…

Yet another observation, which will lead us to about the same numbers, is that (as mentioned above) it is *easy* to utilize more cores while *increasing* the total time of solving a certain problem; as mentioned above, this problem often manifests itself if our calculations are split into slices which are way too fine-grained (and this often causes context switches to kick in, eating up all the resources). As a Big Fat Rule of Thumb™, if we're running for ~100ms, we can realistically consider a/the? thread context switch negligible (actually, 100–200ms is a typical time slice of a modern OS, and for this very reason too). On a modern CPU, 100ms corresponds to ~300M CPU cycles, so the decision of OS developers to have a time slice of 100–200ms to ensure that thread context switch costs are negligible looks rather consistent with our worst-case estimates of millions CPU cycles per thread context switch.

There is one more way to think about it: if we take a look at a CPU, we'll realize that it is inherently an event-driven system; all we have at the hardware level are cores and hardware interrupts routed to those cores, there is nothing more – and in this case, each core is actually a (Re)Actor with its state being core registers + associated memory, allowing the (Re)Actor to handle interrupts (including timer interrupts which cause preemption) as its input events. From this point of view, the very concept of a thread is an artificial entity (~= 'abstraction with a non-zero cost') – and using threads, and especially inter-thread synchronization between those threads, the non-zero cost of this abstraction starts to hurt performance.

Big fat reason #5: Lack of scalability

By design, shared-memory approaches (which BTW include both mutexes and atomics) cannot be scaled beyond one single computer (well, strictly speaking, it is possible to simulate shared memory across boxes, but it is going to be excruciatingly slow [StackOverflow] – that's like 1000x slower, which makes it impractical).

Ok, mutexes are bad – but is there anything better?

By now, I hope that I have made a case against mutexes (and, more generally, against shared-memory architectures). But this line of argument makes sense only if something better exists (otherwise we're stuck with "Hey, it is the worst thing in the universe – except that it is the only one, so we have to use it anyway").

Message-passing, including (Re)Actors a.k.a. Actors a.k.a. Reactors a.k.a. event-driven a.k.a. ad hoc FSMs

Fortunately, there is a well-known approach that solves *all* the issues raised above (though not without some cost): it is using message-passing shared-nothing architectures. They have been known for ages; in modern computing, the oldest living relative of the shared-nothing message-passing stuff is probably Erlang; however, recently many more technologies have emerged which are operating more or less along the same lines (though they're mostly event-driven which is a subset of more generic message-passing): Akka Actors, Python Twisted, Go (at least as its idiomatic version), and of course, Node.js; also there is an ongoing development in which I am involved too [Node.cpp].

Let's take a look at the Five Big Deficiencies of mutexes we discussed above, and observe how shared-nothing message-passing and (Re)Actor-like programming models fare in this regard:

1. **MT-error prone**. As message-passing and event-driven programs behave 'as if' they're single-threaded, they're naturally mostly free from shared-memory artifacts such as forgotten mutexes and deadlocks (well, in theory it is possible to have a deadlock between different message-passing programs, but during all my years with them I didn't see one).

   More formally, while message-passing programs may still use some multithreading primitives deep inside (say, to implement inter-thread queues), these primitives are extremely small and never-changing, and therefore can be *proven* to be correct. And as soon as we're inside a message-passing program which doesn't use shared memory, all such programs can be *proven* just once to be MT-correct regardless of their specific logic.

2. **Untestability**. Unlike mutex-based stuff, message-passing programs are testable; in other words, if we feed the same inputs to a message-passing program, it will produce the same results (='our tests become reproducible')

   Even better, message-passing/(Re)Actor programs can be made deterministic in a practical sense, all the way up to recording-replay logic, which can be used in production (!!). While such things are known, I don't know of a single generic framework providing this functionality (except for WIP [Node.cpp]).

3. **Complexity**. As with message-passing programs, there are no mutexes to worry about, which eliminates related mental costs. In practice, the development of non-mutex programs is known to be significantly faster than that of mutex-based ones.

   As discussed in [DDMoGv2], while asynchronous processing has its own complexity costs, as soon as we're speaking about the *interaction* between different requests, this complexity happens to be *much lower* than that of thread-synced approaches; this becomes even more true if we can use modern async-oriented constructs such as `futures`/`promises` and the `await` operator in C#, Node.js, and C++.

4. **Performance**. No mutexes and blocking calls → no forced thread context switches → no millions of CPU cycles spent on those thread context switches. In practice, it means the best possible performance for our message-passing/(Re)Actor program (hey, there is a reason why **nginx** performs significantly better than **Apache**). One way to think about comparing a mutex-based system with a message-passing one is an analogy with traffic: each mutex is a traffic light, slowing things down; and message-passing doesn't have mutexes/traffic lights, which allows to move like on a highway – much faster.
5. **Scalability**. Message-passing programs can scale to multiple boxes easily; two common examples are Erlang and MPI, but I've seen much more examples in the real-world than that.

As we can see, *none* of the Five Big Fat Problems of mutex-based programs apply to message-passing and event-driven programs.

Conclusion

We discussed Five Big Fat Reasons why mutexes are bad… Moreover, we took a look at the alternative message-passing/event-driven programs and found that *none* of the Five Big Reasons apply.

Therefore:

> *If you care about any of the {correctness | MTBF | complexity | performance | scalability} of your programs, do you and your customers a favour and get rid of mutex-based shared-memory abominations.*

Note that this does NOT exclude multi-coring as such – but moves thread sync out of sight of the app-level and very severely limits its nature, so it becomes manageable (and its correctness, provable). ■

Bibliography

[DDMoGv2] 'No Bugs' Hare, 'Development and Deployment of Multiplayer Online Games', vol. II

[Edwards12] Stephen A. Edwards (2012) 'History of Processor Performance' http://www.cs.columbia.edu/~sedwards/classes/2012/3827-spring/advanced-arch-2011.pdf

[Fowler11a] Martin Fowler (2011) 'Eradicating Non-Determinism in Tests', https://martinfowler.com/articles/nonDeterminism.html

[Fowler11b] Martin Fowler (2011) 'The LMAX Architecture', https://martinfowler.com/articles/lmax.html

[Henney16] Kevlin Henney, (2016) 'Thinking Outside the Synchronisation Quadrant', presented at code::dive 2016 and published 27 June 2017 at https://www.slideshare.net/Kevlin/thinking-outside-the-synchronisation-quadrant

[Ignatchenko98] Sergey Ignatchenko (1998) 'STL Implementations and Thread Safety', *C++ Report*, July/Aug 1998

[Li07] Chuanpeng Li, Chen Ding, Kai Shen, 'Quantifying The Cost of Context Switch', https://www.usenix.org/legacy/events/expcs07/papers/2-li.pdf

[Loganberry04] David 'Loganberry', Frithaes! - 'An Introduction to Colloquial Lapine!', http://bitsnbobstones.watershipdown.org/lapine/overview.html

[McCracken02] Dave McCracken (2002) 'POSIX Threads and the Linux Kernel' in *Proceedings of the Ottawa Linux Symposium* https://landley.net/kdocs/ols/2002/ols2002-pages-330-337.pdf

[NoBugs10] 'No Bugs' Hare, 'Single-Threading: Back to the Future?' in *Overload* 97, June 2010, https://accu.org/index.php/journals/1634

[NoBugs15] 'No Bugs' Hare, Multi-threading at Business-logic Level is Considered Harmful, *Overload* 128, Aug 2015, https://accu.org/index.php/journals/2134

[NoBugs17] 'No Bugs' Hare, 'Eight Ways to Handle Non-Blocking Returns in Message-Passing Programs', CPPCON17, http://ithare.com/eight-ways-to-handle-non-blocking-returns-in-message-passing-programs-with-script/

[Node.cpp] Node.cpp, https://github.com/node-dot-cpp

[Rustonomicon] The Rustonomicon, 'What Unsafe Rust Can Do', https://doc.rust-lang.org/beta/nomicon/what-unsafe-does.html

[Shimpi00] Anand Lal Shimpi (2000), 'The future of Intel's manufacturing processes'on *Anandtech* at https://www.anandtech.com/show/680/6

[StackOverflow] Sharing memory across multiple computers?, *Stack Overflow*, https://stackoverflow.com/a/11903065/4947867

[TheGoBlog] The Go Blog 'Share Memory By Communicating', July 2010, https://blog.golang.org/share-memory-by-communicating

[Wikipedia] History of operating systems, *Wikipedia*, https://en.wikipedia.org/wiki/History_of_operating_systems

A Small Universe

Writing a programming language is a hot topic. Deák Ferenc shows us how he wrote a compiler for bytecode callable from C++.

Let's take a deep dive in the world of programming languages, compilers, virtual machines and embeddable execution environments in this article, since we are not setting a lesser goal than creating a tiny programming language, then writing a compiler for it which will create bytecode for a virtual machine – which, of course, we have the luxury of imagining from scratch – and last but not least, making all this run while being embedded inside a C++ application.

But before we embark on this epic journey, we also need to know what we are dealing with by providing an overview of how compilers work, what the steps are they perform in order to transform human-readable (or better said, human-engineered) source code into a series of bytes incomprehensible for aforementioned humans, but which makes perfect sense for computers. So, our article will start with an overview of the inner working of a generic compiler, on a level more theoretical than practical. From there, we will move to one of the main goals of this article, which is the creation of a very simple programming language and providing a fully functional compiler for it.

Since creating a simple compiler is a task which usually targets the architecture of a computer, we also will introduce a virtual machine, with a set of unique instructions that will be executing our compiled language, so let's get started.

How a compiler works

The widely used mainstream compilers tend to appertain to a family of applications characterized by complexity, with elaborately written source code, most typically engineered over several years by a few experts in the field. When an end user uses a compiler in order to compile a piece of source code, the compiler has to go through a series of intricate steps in order to produce the required executable, or to provide valuable error messages if errors are found during compilation.

On a very high level, Figure 1 shows the steps a compiler does with a source file, in order to generate executable binary code.

First, it takes the source code and reads it. This step provides input for the next, usually in forms of 'tokens'. The output of step 1 is run through an analyze phase (step 2), which generates a so-called 'Abstract Syntax Tree', which in turn is used as input for the last stage. The last stage provides the binary code that is runnable on the target platform.

Between these steps, a compiler often translates the data into an Intermediate Representation that is equivalent to the input source code but that the compiler can work with more easily (the data are algorithmically easier to process), such as graphs, trees or other data structures – or even a different (programming) language.

Deák Ferenc Ferenc has wanted to be a better programmer for the last 15 years. Right now he tries to accomplish this goal by working at Maritime Robotics as a system programmer, and in his free time, by exploring the hidden corners of the C++ language in search of new quests. fritzone@gmail.com

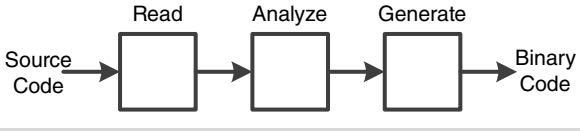


Figure 1

And at a high level, this explanation is more than enough for someone to understand what happens inside a very simple and naïve compiler.

Types of compilers

When it comes regarding the approach of a compiler to source files, we can differentiate two types of compilers:

- One-pass compilers
- Multi-pass compilers

One-pass compiler

As the name suggests, the one-pass compiler is a compiler which passes through a compilation unit exactly once, and it generates corresponding machine code instantly. 'One-pass' does not mean that the source file is read only once but that there is just one logical step affecting the various (compiler internal) data of the compilation steps. The compiler does not go back to run more steps on the data once done; instead the compiler passes from parsing to analyzing and generating the code then going back for the next read.

A compilation unit can be also imagined as just a fancy technical term for a source file, but in practice it can be a bit more. For example, it might contain information provided by the pre-processor, if the target language has this notion, or it might contain all the imported source code of the modules this file needs in order to compile (again if there is support for this feature in the language this compiler targets).

One-pass compilers have a few advantages, and several disadvantages, in comparison to multi-pass compilers. Some advantages are that they are much simpler to write, they are faster and much smaller. For some (most popular) programming languages, it is impossible to write a one-step compiler due to the syntax the languages require. A notable exception is the programming language Pascal, which is a perfectly good candidate for a one-pass compiler because it requires definitions be available before use (by requiring syntactically that the variables are declared at the beginning of the current procedure/function or that the procedure is forward declared). The application provided as educational material within this article is a simple one-pass compiler.

Multi-pass compilers

In contrast, multi-pass compilers [Wikipedia-3] convert the source into several intermediate representations between various steps in the long path from source code to the corresponding machine code. These intermediate representations are passed back and forth between various stages of the

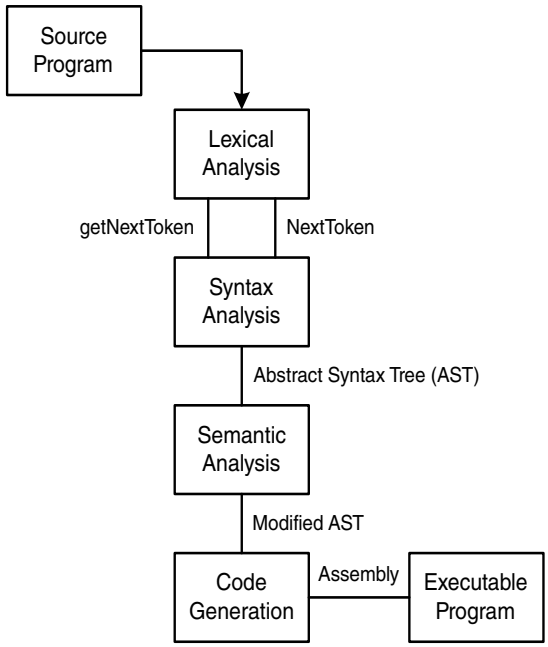


Figure 2

compile process between the compilation phases. At all stages, the compiler sees the entire program being compiled in various shapes.

Multi-pass compilers – because they have access to the entire program in each pass (in various internal representations) – can perform optimizations, such as removing irrelevant pieces of binary code to favor smaller code, or excluding redundant code, and they can apply a variety of other gains at the cost of a slower compile time, and higher memory usage.

The typical stages of a multi-pass compiler can be seen in Figure 2[1].

Lexical analysis

The lexical analysis stage is the first stage in the front end of a compiler's implementation. When performing lexical analysis, the compiler considers the source as a big chunk of text, which needs to be broken up into small entities (called tokens).

This stage is responsible for building up the internal representation of the source code using tokens, after removing irrelevant information from it (such as comments, or contiguous space, unless they are relevant to the syntax of the source code, just like in case of **Python**). Each token is assigned a type (for example, identifiers, constants, keywords) among other relevant information, and the information from this point on is passed on to the next stage.

Now follows a very simple example of tokenization.

The following expression $x = a + 2;$ can be tokenized into the resulting sequence (partially using the example from [Wikipedia-2])

```
[(identifier, x), (operator, =), (identifier, a),
(operator, +), (literal, 2), (separator, ;)]
```

One of the methods of constructing an efficient lexer for a language is to build the language based on a regular grammar for which there are already well defined, efficient and optimal algorithms. A regular expression [Wikipedia-5] can parse the language generated by the regular grammar and, most importantly, there is a finite state automaton that can interpret the given grammar.

Finite state automatons

A finite state automaton can be viewed as an abstract machine that is in exactly one specific state at any given moment in time. The automaton changes its state because of an external impact, and this process is called a transition. It can be modelled from a mathematical point of view as a quintuple:

$$M = (Q, \Sigma, \delta, q_0, F)$$

where:

- Q is a finite set of states
- Σ is a finite alphabet
- δ is the transition function:

$$\delta : Q \times E \to Q$$

- F represents the set of finite states, and
- $F \subseteq Q$, F might be empty.

For example, the finite state machine in Figure 3 is designed to recognize integral numbers (without sign).

Working out the mathematical definitions and the regular expression for the given automaton from the diagram is left as a fun homework for the reader.

Syntax analysis

In the next phase of the compilation, the tokens (the output of the lexical analysis stage) are checked in order to validate their conformity with the rules of the programming language. Regardless of the success of the lexical analysis, not all randomly gathered tokens can be considered a valid application, and this step ensures that these tokens are indeed a valid embodiment of a program written in this language. Due to the recursive nature of a programming language's grammar and syntax, at this stage the application is best represented by a context-free grammar, and the responsible corresponding push-down automaton. Once the rules are satisfied, an internal representation of the source is generated, most probably an abstract syntax tree.

Context-free grammars

A context-free grammar is just another way of describing a language and mathematically it can be represented with the following structure:

$$G = (N, \Sigma, P, S)$$

where:

- N is a finite set of non-terminals (**variables**), which represent the syntactic constructs of the language
- Σ contains the symbols (**terminals**) of the alphabet on which the automaton is working, ie. the symbols of the alphabet of the defined language (and not to be confused with Σ from the previous definition of the automaton: it's not the same)
- P is a set of **rules** used to create the syntactic constructs which operate on the variables and result in a string of variables and terminals (rules for different derivation of the same non-terminal are often separated by `"|"`).
- S is the start symbol

Consider the following context-free grammar:

$$G = (\{S\}, \{0,1\}, \{S \to 01 | 0S1\}, S)$$

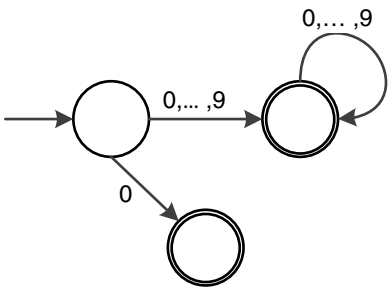


Figure 3

1. Original diagram by Kenstruys (own work) and placed in the public domain. Gratefully downloaded from Wikimedia Commons. https://commons.wikimedia.org/w/index.php?curid=6020058.

which was designed to be able to generate the language *L*. *L(G)* is the notation for the language generated by the given grammar *G*:

$$L(G) = \{0^n 1^n \,|\, n \geq 1\}$$

For example, obtaining 000111 can be done by starting with the Start symbol and consecutively applying the rules (2), (2) and (1) resulting in the following **derivation**:

$$S \to 0S1 \quad (2)$$
$$S \to 00S11 \quad (2)$$
$$S \to 000111 \quad (1)$$

A derivation in CFGs, which have rules in their rulesets involving more than one non-terminal (for example, $(\{S\},\{+,1\},\{S \to S+S|S|1\},S)$ is called a left-most (right-most) derivation. At each step, we replace the left-most (right-most) non terminal:

$$S \to S+S \to 1+S \to 1+1$$

thus obtaining the expression 1+1 and the parse tree (shown in Figure 4), which is an ordered tree representing the structure of the string according to the context-free grammar used to derive it.

Push-down automaton

A push-down automaton can be imagined like a finite state-machine but with an extra stack attached to it. The following 7-tuple gives a more or less formal definition for it:

$$(Q,\Sigma,\Gamma,\delta,q_0,Z_0,F)$$

where:

- Q, Σ and F are defined likely as for a finite-state automaton
- Γ is the alphabet of the stack
- q_0 is the start state of the automaton
- Z_0 is the start symbol, $Z_0 \in \Gamma$

and the transition function is:

$$\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to P(Q \times \Gamma_\epsilon)$$

so it takes in a triplet of a state from Q, an input symbol (which can also be the empty string, called ϵ – epsilon) from Σ and a stack symbol from Γ and it gives result a set of zero or more actions of the form (p, α) where p is a state and α is a string of concatenated stack symbols.

Remembering the context-free grammar from our previous paragraph, we can construct a push-down automaton which will recognize the elements of $\{0^n\,1^n \,|\, n \geq 1\}$:

$$(\{q,p,f\},\{0,1\},\{Z_0,X\},\delta,q,Z_0,\{f\})$$

Concerning the states:

- q is the initial state and the automaton is in this state if it has not detected the symbol 1 so far but only 0s.
- p is the state where the automaton lies if it has detected at least one 1 and it can proceed forward only if there are 1s in the input.
- f is the final state

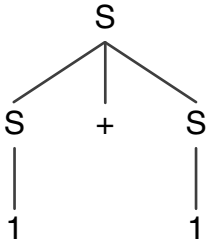


Figure 4

And where the transition function is defined as:

$$\delta(q,0,Z_0) = \{(q,XZ_0)\}$$
$$\delta(q,0,X) = \{(q,XX)\}$$

These two rules are handling the 0 from the input: both of them push one X onto the stack of the automaton for each 0 identified.

$$\delta(q,1,X) = \{(p,\epsilon)\}$$

This rule will change to state p and pop one X from the stack

$$\delta(p,1,X) = \{(p,\epsilon)\}$$

If in state p and we encounter a 1 in the input, pop one X from the stack

$$\delta(p,\epsilon,Z_0) = \{(f,Z_0)\}$$

And finally, at this stage we have supposedly successfully identified our string of 0s and 1s. Our stack needs to be empty, the input fully consumed and then we can advance to the final state.

So, if we have 000111, the automaton goes through the following steps in order to recognize it as a valid input for the given grammar:

$$(q,000111,Z_0) \vdash (q,00111,XZ_0) \vdash (q,0111,XXZ_0) \vdash$$
$$(q,111,XXXZ_0) \vdash (p,11,XXZ_0) \vdash (p,1,XZ_0) \vdash$$
$$(p,\epsilon,Z_0) \vdash (f,\epsilon,Z_0)$$

Now we can hopefully see the methodology involved in this step, we just need to create a push down automaton for our language which will successfully consume the series of tokens from the previous step thus resulting in a syntactically correct application.

In the paragraphs below, we frequently mention elements of a push-down automaton and refer to (context-free) grammars, hence this very short theoretical introduction. Unfortunately, a deeper study of this very exciting subject is beyond the scope of this article, being such a huge subject in the field of computer science that there are semester-long courses dedicated to the research of this domain. It is practically impossible for us to dig deeper into this field now. For everyone interested in this fascinating domain, I highly recommend *Introduction to Automata Theory, Languages, and Computation* by Hopcroft, Motwani and Ullman (2nd, 2001 edition).

The parser

The component responsible for this step in the compiler's architecture is called the parser. Depending on the approach chosen to build the parse tree upon encountering an expression, we differentiate between the following types of parsers:

- descending – the tree is built from the root towards the leaves. In this situation, the parsing starts by applying consecutive transformations to the start symbol till we obtain the required expression. This approach is also called top-down parsing.
- ascending – the tree is built from the leaves towards the root, upon applying reverse transformations to the expression in order to reach the start symbol. This is also called bottom-up parsing.

Recursive-descent parsing

Usually the top-down parsers, which process the input from left to right, and the parse tree, which is constructed from top to bottom, are also known as recursive-descent parsers due to the recursive nature of context-free grammars. In the construction of the tree, a backtracking mechanism might be involved, which results in a less effective performance due to the constant back-stepping of the algorithm. These parsers are usually avoided in case of large grammars.

The parsers which are able to 'predict' which production is to be used on the input string do not require any backtracking steps and are called predictive parsers. These parsers operate on specialized *LL*(k) grammars, which are a simplified subset of the context-free grammars. Due to some

applied restrictions and a precomputed state transition table introduced in the algorithm (which determines how to compute the next state from the current state and the lookahead symbol), they gain an extra simplicity when it comes to implementation.

LL(*k*) can be interpreted in the following manner:

- the first *L* means: **L**eft to right
- the second *L* means: **L**eft-most derivation
- and the *k* represents the number of lookaheads the parser can perform.

Parsers which operate on *LL*(*k*) grammars are also called *LL*(*k*) parsers.

Shift-reduce parsing

While reading the input, usually from left to right and trying to build up sequences on the stack which are recognized as the right side of a production, the parser can use two steps:

- Reduce: Let's consider the rule `A→w`. When the stack contains `qw`, its content can be reduced to `qA`, where `A` is a non-terminal from our grammar. When the entire content of the stack is reduced to the start symbol, we have successfully parsed the expression.
- Shift: If we can't perform a reduction (or 'if we can't reduce the stack') of the stack, this step advances the input stream by one symbol, which is pushed onto the stack. This shifted symbol will be treated as a node in the parse tree.

Parsers using these steps are called shift-reduce parsers.

LR-parsers

A more performant family of the shift-reduce parsers are the *LR* parsers (invented by Donald Knuth in 1965 [Knuth65]). They operate in linear time, producing correct results without backtracking on the input, using a **L**eft to right order and always use the **R**ight-most derivation in reverse. They are also known as *LR*(*k*) parsers, where *k* denotes the number of lookahead symbols the parser uses (which is mostly 1) in the decision-making step. Similarly to *LL* parsers, *LR* parsers use a state transition table that must be precomputed from the given grammar.

Several variants of the *LR* parsers are known:

- *SLR* parsers: these are the **S**imple **LR** parsers, categorized by a very small state transition table, working on the smallest grammars. They are easy to construct.
- *LR*(1)parser: these parsers work on large grammars. They have large generated tables, and they are theoretically enough to handle any reasonably constructed language – they are just slow to generate due to the huge state transition table.
- *LALR*(1) parsers: the **L**ook **A**head **LR** parsers are a simplified version of the *LR* parsers, able to handle intermediate size grammars, with the number of states being similar to *SLR* parsers.

LR parsers operate on *LR* grammars, whose pure theoretical definition we will skip for now. We will just mention that as per the [Dragon] book, page 242, the requirement of an *LR*(*k*) grammar states that "we must be able to recognize the occurrence of the right side of a production in a right-sentential form with *k* input symbols of lookahead" is less strict than the one for *LL*(*k*) grammars "where we must be able to recognize the use of a production seeing only the first *k* symbols of what its right side derives". This allows *LR* grammars to describe a wider family of languages than *LL* grammars with only one drawback: the allowed complexity of the grammar makes it difficult to hand-construct a proper parser for it, so for a context-free grammar with a decent complexity, a parser generator will be required to generate a parser.

With this, we have ended our long but still frivolous journey in the world of parsers and syntax analysis, but for anyone interested there is no better source to turn to, than Chapter 4 of the dragon book [Dragon] where all these concepts are discussed in great length.

Semantic analysis

The semantic analysis stage during compilation validates the output generated by the syntax analysis and applies semantic rules to it in order to confirm the requirements of the language, such as strict type checking or other language-specific rules.

The input to this stage is an intermediate representation, mostly in the form of an abstract syntax tree created by the previous step, and the following operations may be performed on it, resulting in an annotated abstract syntax tree (or, if our product is a source-to-source translator, even the final variant of the result). The following list contains a few of the rules that can be applied in this stage; however, since semantic analysis is a highly programming-language-specific step, it is worth mentioning that each of the items can be prepended with 'if the language has support for it then':

- Type checking for various entities (such as, does the language require that indexes to an array must be a positive integer – for example, Pascal allows also negative indexes in the syntax).
- Validation of operands of various operators (some programming languages may consider having a multiplication sign between a string and an integer to be ill-formed to have, but not Python).
- Verification of implicit conversions (what will be the result of an addition of a real type variable to an integer type variable).

Code generation

In the last step of the compilation process, a typical compiler will generate executable code for the application. This is done by converting the resulting intermediate representation into a set of instructions (usually target architecture assembly, if we are compiling through assembly, or direct binary code).

The code generation phase, depending on the compiler itself, might be through the introduction and use of intermediate code, or some compilers can generate code directly for the target architecture. The intermediate code is a sort of machine code, which without being architecture specific can still represent the operations of the real machine.

The intermediate code is very often represented in the form of a 3-address code. The 3-address code can be viewed as a set of instructions where each instruction has at most 3 operands, and they are usually a combination of an assignment operation and a binary operation. For example, the following expression:

$$y = a \cdot x + b$$

can be represented by the following sequence of 3-address code instructions:

$$t1 = a \cdot x$$
$$t2 = t1 + b$$
$$y = t2$$

The importance of a 3-address code is that a very complex expression is broken down into several separate instructions which are very close to assembly level and they are very easily translated into the required machine code instruction.

Transforming an abstract syntax tree into a series of 3-address instructions is not an extremely complex task; basically, it just means traversing the tree and correctly identifying the set of temporaries and the values assigned to them, so generating code for the simple expressions of the language is possible using this process.

Generating code for compound statements, such as representing `IF` or `FOR` loops, usually boils down to a combination of jumps depending on the conditions imposed on the statements and then again generating code for simple expressions.

Generating code for function calls requires the presence of a stack, where – in the currently used Von Neumann architecture – we store not only the ingoing parameters to a function (if any) but also the addresses of return after the completion of the function.

First, the parameters are calculated, and their value is pushed onto the stack, then we issue the call command to invoke the method. This will also push the address where the called function should return after completing its execution. After the function has returned, we are back to the first instruction after the call of the method; the only thing that remains is to adjust the stack properly and continue execution.

Architecture of a compiler

For complex compiler projects, we usually see a three-stage structure like in Figure 5.

This design has the advantage that several programming languages can be compiled using the same compiler (applying the same optimizations to all the source code written in different programming languages) to several machines and architectures (image courtesy of Wikipedia).

The front end

The front end section of the compiler is the one responsible for the lexical and syntax analysis of a dedicated programming language. It produces an intermediate representation of the program and should also manage the symbol table (which is a map of the symbols that were encountered in the program, such as variable names and function names, together with their location).

The lexical analysis, the syntax analysis and the semantic analysis steps are all part of the front end since these steps are all programming-language dependent.

If there is macro support in the language, the preprocessing phase is also part of the front end. The front end is responsible for creating an intermediate representation for further processing by the middle end.

The middle end

The middle end of the compiler is responsible for performing optimizations on the intermediary code. This is a platform and programming language independent operation, so all the programs compiled from languages that the compiler's front end supports will benefit from this optimization phase. Some of the easiest to implement optimizations that can be performed on the intermediary code are in the section following, but for those interested in this subject [CompOpt] and [Wikipedia-1] contain pretty exhaustive lists.

Constant folding and constant propagation

This optimization is responsible for finding values in code that occur in calculations that will never change during the lifetime of the application and for calculating them instead of doing the operations at runtime. for example:

```
const int SECONDS_IN_HOUR = 60 * 60;
const int SECONDS_IN_DAY = SECONDS_IN_HOUR * 24;
```

can be replaced with

```
const int SECONDS_IN_DAY = 86400;
```

where the value of **SECONDS_IN_HOUR** has been propagated to contain the correctly calculated (folding) value.

Dead code elimination

Sometimes there is a piece of code in the application which will not be reached. Dead code elimination will identify these cases and remove them, thus reducing the size of the code. For example:

```
bool some_fun(int a)
{
  if( a < 10)
  {
    return true;
  }
  else
  {
    return false;
  }
  std::cout << "some_fun exiting";
}
```

In this case, the **std::cout** will never be called, so the compiler is free to remove it.

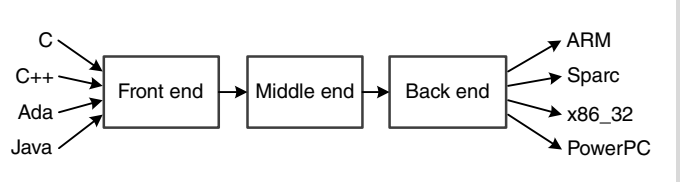


Figure 5

Common sub-expression identification

This optimization is responsible for identifying calculations which are performed more than once. It extracts them into a commonly used part which can be inserted into the location where the calculation was done.

```
int x = a + b + c;
int z = a + b - d;
```

This can be optimized into the following sequence of code by extracting the common part **(a+b)** and reducing the generated code to the following:

```
int temp = a + b;
int x = temp + c;
int z = temp - d;
```

These items are just a quick peek into the exhaustive list that an optimizing compiler does to your source code, so feel free to do more research in this field if you are interested in this fascinating domain.

The back end

The back end component of a compiler is the one which actually generates the code for the specified platform that this back end was written for. It receives the optimized intermediate representation (either an abstract syntax tree or a parse tree) from the middle end, and it may do some optimizations which are platform specific. In cases where is support for debugging the applications on the given platform, debug info is collected and inserted into the generated code.

As presented in the code generation paragraph, the abstract syntax tree is converted into a 3-address code representation, which the back end then re-works and re-organizes in order to represent the real computer architecture related concepts, such as registers, memory addresses, etc…

Modern compilers also have support for generating code for a specific platform (for example x86/64) but for different processor architecture, with the risk that the generated code will not run on anything else, but that specific processor, or it is possible to have the back end generate code for a totally different architecture (which process is called cross compilation).

Real-life compilers

In real life, it is a very frequent situation that a compiler is split up into several smaller applications, for example considering **gcc**, the following applications can be found on the system [GCCArchitecture]:

- **gcc** is a driver program which invokes the appropriate compilation application for the target language (for example **cc1** being the preprocessor and the compiler for the C language, or **gdc** is the front end for the D language)
- **as** being the assembler (the assembler is the application which creates object (machine) code for specific compilation units), actually it is a part of the binutils package shipped with various systems
- **collect2** is the linker (the application that creates a system specific binary from the various object files)

For **clang**, the overall system architecture and design is very similar to **GCC**, with the major difference being that clang was created with the aim of integration into various IDEs so there is a considerable set of libraries that make the project more flexible and easier to work with [Guntli11], and

also **clang** has the advantage of using the LLVM compiler infrastructure [LLVM].

For those not familiar with the terminology, LLVM used to stand for 'Low Level Virtual Machine', but right now it is just … LLVM, since the project outgrew the notion of a virtual machine, and evolved into a collection of various compiler-related artifacts that include a set of high quality components. This includes an implementation of the C++Standard Library to a debugger (**lldb**), and a set of front and back ends for various languages and target architectures via a powerful intermediate representation of the compiled language with a unique language (which is similar to assembly, and can be read by humans too) that the middle layer uses to perform strong optimizations on it before emitting architecture-specific code.

A script called primal

Now that we are done with the theoretical introduction of what a compiler is, how it works, and the algorithms and data structures behind the wall, it is time to sail into more practical waters, by introducing a small scripting language for which we create a compiler from scratch, and also a virtual machine to actually run the compiled code.

I mentioned that we will create the compiler from scratch although there is a plethora of tools available for this purpose. However, in my experience they have a tendency to generate code which is overly complex, difficult to read, and not suitable for our situation, where we want to explain the methodology of creating a compiler. Regardless, feel free to browse [Wikipedia-4] to find a list of these tools stacked up against each other and find your favourite in there, dare you endeavour to create a new programming language.

The basic syntax

The script's syntax is highly inspired by BASIC and for now it will support the following (very limited set of) operations:

- Mathematical operations: + (addition), - (subtraction), * (multiplication), / (division)
- Binary operations: & (and), | (or), ! (not), ^ (xor)
- Comparison operators: > (greater than), < (less than), == (equals), <= (less than or equals), >= (greater than or equals)
- Assignment operator: = (assignment)

And the following keywords will be implemented:

- `if` and `then` to verify the truthness of an expression (sadly, no `else` is implemented yet)
- `goto` to hijack the execution flow by jumping to a specified label
- `let` to define a variable (if not found) and assign a value to it
- `asm` to directly tell the compiler to compile assembly syntax as per the VM's opcodes (presented below)
- `var` to introduce a variable to the system

  As a side note, all variables need to be declared before they are used, à la Pascal. And for the moment, only numeric variables are supported.
- `end` to close the code blocks of commands which have blocks (such as `if`)
- `import` to include the content of another source file into the current one

In order to define a label we use the following syntax:

```
:label
```

And as an extra feature, we will implement a `write` function which simply writes out what is passed in as parameters.

And now, armed with this knowledge we can write the following short application to print out a few Fibonacci numbers (see Listing 1).

As you have correctly observed, each line contains exactly one instruction. And also as you have observed, each line starts with a keyword. Both of these play an important role in the design of an application and will be explained later in the article.

```
import write

var t1, t2, nextTerm, n

let n = 100
let t1 = 0
let t2 = 1

:again

let nextTerm = t1 + t2

let t1 = t2
let t2 = nextTerm
if nextTerm < n then
    write(nextTerm, "  ")
    goto again
end
```

Listing 1

In case you wish to compile scripts upon compilation, there are two executables in the build directory: **primc** is the compiler for the script language and **primv** is the virtual machine which runs the output of the compiler.

The Backus-Naur form of the primitive script

The Backus-Naur form is a notation scheme for context-free grammars, and it can be used to describe programming languages. Our own primitive script can be described (with a more complete version) of the following sequence of BNF (where `|` is used to indicate that only one can be chosen from the given set between `{` and `}`, the items between `[` and `]` are optional, `...` means a repetition of the previous group and `CR` means a newline).

I have omitted few of the supported keywords from Listing 2 (overleaf) in order to not to have the listing very long and boring; however, the example gives an overview of the complexity of how to start implementing the grammar required for a programming language, which in turn can be fed into a compiler compiler (for example yacc or bison) in order to generate a parser for the language.

Our compiler

The project is a simple one-pass compiler, without any implemented optimizations. It was written in C++ and uses CMake as the build system. After you have checked out the project from https://github.com/fritzone/primal, you will find a directory structure, where the names of the directories are hopefully self-explanatory:

- **compiler** is where the compiler files are to be found
- **vm** stands for the source code for the virtual machine
- **hal** stands for the 'Hardware Abstraction Layer', just a fancy name
- **opcodes** stands for the assembly opcodes supported by the virtual machine
- **tests** is the directory of the unit tests, right now we use Catch2

But before we dig deeper here, just a quick mention: this entire project was created as a homegrown fun project, mostly for research purposes, so don't expect production-quality speed (or code) neither from the compiler, nor from the compiled part. There are several excellent compilers in the market right now. I have tried to keep the primal compiler as simple and understandable as possible, and sometimes I had to cut a few corners to have it in a digestible size. For now, the entire project is released under an Affero GPL license, so feel free to modify, contribute and distribute in the spirit of true open source.

Banalities of the compiler

I had to create a few wrappers around common notions, such as 'source' in order to have an easier approach to them, but the entire project is packed

```
<identifier>              ::= <letter>[{<letter>|<digit>}...]
<label identifier>        ::= <identifier>
<letter>                  ::= <small letter> | <capital letter>
<small letter>            ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<capital letter>          ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<digit>                   ::= 0|1|2|3|4|5|6|7|8|9
<digit string>            ::= digit [{digit}...]
<label>                   ::= :<label identifier>
<immediate statement>     ::= <compound statement> CR
<compound statement>      ::= <statement> | <immediate statement>
<statement>               ::= let <identifier>=<expression>                    |
                              if <expression> then <compound statement> endif |
                              goto <label>                                    |
                              <label>                                         |
<comp op>                 ::= < | > | <= | >= | == | !=
<add op>                  ::= + | -
<mul op>                  ::= * | /
<expression>              ::= <term1> | <expression> \| <term1>
<term1>                   ::= <term2> | <term1> & <term2> | !<term2>
<term2>                   ::= <term3> | <term2> <comp op> <term3>
<term3>                   ::= <term4> | <term3> <add op> <term4>
<term4>                   ::= <term5> | <term4> <mul op> <term5>
<term5>                   ::= <term5> | <term5> ^ <term6>
<term6>                   ::= <expression> | <identifier> | <digit string>
```

Listing 2

in the primal namespace to not to pollute the global one. As mentioned above, the compiler does not follow the standard mechanisms of creating a compiler, as we have not created a context-free grammar for it that can be fed in one into one of the automated tools to generate the proper mechanisms to automatically deal with situations that you can (will) encounter while writing a compiler. Instead, we tried to 'guess' how to approach these situations and 'solved' what had to be solved in a more manual manner, and – with explanations – they are presented here.

- The compiler itself is located in the class **`compiler`**.
- The source code is wrapped into the class **`source`**. Very conveniently for us, this class uses **`std::stringstream`** to read in the required source code, from a string variable. As mentioned before, each line in the script lives its own life, and we have abused the string stream to serve us the text line by line by using **`std::getline`**. The text of the source is traversed only once.
- Each line of code is wrapped into a **`class sequence`** derived keyword object. As mentioned before, in this programming language each line starts with a keyword, and these keywords are stored in their separate classes too, in the `keywords` folder inside the compiler. This decision was mainly influenced by the following
  - To keep it simple and easy to implement.
  - To make it extensible without too much hassle – for example, introducing a new keyword should not be an extremely complex operation.

Parsing and tokenizing

For our little compiler, we have created a very simple tokenizer with a few lines of code, which suits very well our primitive script's syntax. It can be found spread across the classes **`lexer`**, **`parser`** and **`token`**.

The **`parser`** class is responsible for parsing the entire source object that it was assigned, and the method **`parser::parse`** does the actual work:

```
template<class CH> parse(source& input,
CH checker, std::string& last_read) { ... }
```

The **`checker`** parameter has a specific role: the caller of the parse method can specify the condition at which the current sequence can stop. For the main compiler, this method is called like (from inside**`compiler::compile`**):

```
std::string last;
auto seqs = p.parse(m_src, [&](std::string) {
  return false;}, last);
```

however the implementation of the **`if`** keyword does something else:

```
parser p;
std::string last;
auto seqs = p.parse(m_src,[&](std::string s)
  {
    return util::to_upper(s) == "END";
  },
  last);
m_if_body = std::get<0>(seqs);
```

The reasoning is the following: in the primal language's implementation, each keyword manages its own parsing and compilation, with some backup from the underlying architecture. So, the **`if`** keyword was responsible of extracting its own body (**`m_if_body`**) from the **`source`** object (**`m_src`**) and due to the syntax of the language, the body of the **`if`** is between the current position in the **`m_src`** and the corresponding **`endif`** keyword.

The inner workings of the **`parser::parse`** method are as following:

```
std::string next_seq = input.next();
```

which will extract the next sequence of code (for us: next line) from the input (remember, this was of type **`source`**), and at the same time advance the location in the input. We see the result of the checker on the current sequence: if it evaluates to true, we will not continue the parsing but will go back to the entity which requested the parsing of the source from its specific location.

If we are still in the parsing phase, the tokenizer jumps in:

```
lexer l(next_seq);
std::vector<token> tokens = l.tokenize();
```

where the tokenize method is just a very rudimentary implementation of splitting a string into several components separated by space, comma or other characters. The result of the method is a vector of tokens, where each token has its own type that you can check out in `token.h`:

```
enum class type
{
  TT_OPERATOR  = 1, // omitting the rest
                    // to save space
```

The very first token, as per the definition of the language, must be a keyword. After tokenizing the sequence, we try to identify the keyword object and pass the control to it in order to prepare the sequence in the requested manner (for example, the way it was done for the **`if`** keyword is presented a few lines above). Different keywords do different preparations and checks, in order to validate the sequence.

```
// to compile the expression on which the IF takes
// its decision
sequence::compile(c);

// and set up the jumps depending on the trueness
// of the expression
label lbl_after_if =
  label::create(c->get_source());
label lbl_if_body =
  label::create(c->get_source());

// the comparator which evaluates the expression
comp* comparator =
  dynamic_cast<comp*>
  (operators[m_root->data.data()].get());

// generate code for the true case of it
(*c->generator()) << comparator->jump
  << lbl_if_body;
(*c->generator()) << DJMP() << lbl_after_if;

(*c->generator()) << declare_label(lbl_if_body);
for(const auto& seq : m_if_body)
{
  seq->compile(c);
}
(*c->generator()) << declare_label(lbl_after_if);
```

Listing 3

When the keyword has successfully prepared and validated the sequence of tokens, it is time for us to run Edsger Dijkstra's shunt-yard algorithm [Dijkstra61] on it in order to obtain a reverse Polish notation from which we will construct the abstract syntax tree of the expression.

Building the ast is done in a recursive manner in

```
static void build_ast(std::vector<token>& output,
std::shared_ptr<ast>& croot);
```

where the parameter output is the actual output of the shuntyard algorithm:

```
// create the RPN for the expression
std::vector<token> output = shuntyard(tokens);
// build the abstract syntax tree for the result
// of the shuntyard
ast::build_ast(output, seq->root());
```

Compiling

When the parsing is done, we get back the list of sequences – and by continuing our journey in **`compiler::compile`** we have reached the stage where we need to actually compile the sequences into the corresponding machine code. This is achieved by:

```
// now compile the sequences coming from the parser
for(const auto& seq : std::get<0>(seqs))
{
  seq->compile(this);
}
```

don't let **`std::get<0>(seqs)`** scare you: as you can see in the source code, **`parser::parse`** actually returns a tuple of vector of sequences, where the first one means the instructions from the global namespace. Also, now you might want to jump down to the virtual machine section and read a bit about the architecture of it, to understand the instructions below, and come back after that.

As mentioned before, each keyword is required to compile its own code. Let's present (in Listing 3), as an example, how the **`if`** keyword is actually doing its work (**`kw_if::compile`**).

In the very first step: we compile this sequence via **`sequence::compile(c);`** to obtain the expression, the trueness of which is used by the if to decide whether to execute its body or not. Then we create two labels, one for the body of the **`if`**, the other one for the immediate location after the **`if`**.

Now, we obtain the comparator object from the root of the abstract syntax tree of this **`if`** object. At the current stage of the script (when writing this article), that must be a check for equality or comparison as no other operation is supported. The comparators are stored in the global map of **`primal::operators`** found in `operators.h`. This just lists all the operators the scripting language supports at the moment, together with the assembly opcode (more about this at a later stage, when we discuss the virtual machine) and a jump operation depending on the trueness of the operation.

The expression **`*c->generator()`** will yield a **`generate`** object which is actually a convenience class for seamlessly accessing (read: appending to) the **`compiled_code`** class, which is the part of the compiler where the compiled code actually resides.

Now that we have a **`generate`** object, first we output into it the **`JUMP`** operation (which is currently a jump depending on the state of a flag in the machine where this will run, which will be set depending on the trueness of the expression of the **`if`**) and the label of the body of the **`if`**.

After that, we output a direct jump to the first instruction after the body of the if (just in case the if was actually false).

Now it is time to compile the body of the **`if`**, and we are done.

Implementing the jumps

There is just one issue with compiling the jumps. Some of the labels might refer to locations/labels that at the current moment are not known because they come way after the current location. How this is solved currently is that the compiler inserts a bogus value at the location of the jump label, and in an internal map marks the spot of the label and its reference point. This happens in **`generate::operator<<(const label& l)`**.

When a label declaration happens, the compiler will finally know the location of the label, so it can update its map with the location in **`generate::operator<<(declare_label &&dl)`**.

When the compilation is done and all the locations of the labels are known, **`compiled_code::finalize()`** will patch the locations of the labels with the correct value.

The sequence compiler

At the very lowest level of the compiler is the sequence compiler found in **`sequence::traverse_ast(uint8_t level, const std::shared_ptr<ast>& croot, compiler* c)`**. This is responsible for compiling basic arithmetic expressions, dealing with numbers and variables, etc… The result – the compiled code – is actually a 3-address representation of the code, and the virtual machine interpreting the compiled bytecode was written to execute this representation.

This level tracks the variable index of the 3-address code. Initially, this starts at 0, is very conveniently mapped to registers in the virtual machine, and each descent in the abstract syntax tree will increment this. At the end of the traversal of the tree, the Reg(0) of the virtual machine will contain the value of the expression.

Depending on the type of the expression found in the current node of the tree, a different set of instructions is created. For example, the arithmetic operations generate the sequence of code shown in Listing 4 (overleaf).

To understand this piece of code, let's consider that we are trying to compile the expression **`2+3`**.

This has yielded the following tokens, where **`TT`** stands for **T**oken **T**ype: **`(2, TT_NUMBER)`**, **`(+, TT_OPERATOR)`** and **`(3, TT_NUMBER)`**.

After shuntyarding the expression, we have the following reverse Polish notation: **`(+, TT_OPERATOR)`**, **`(3, TT_NUMBER)`**, **`(2, TT_NUMBER)`**.

This was transformed into the abstract syntax tree in Figure 6.

Now we enter the **`sequence::traverse_ast`** method with the current root pointing to the root of the above tree, on level 0. The code sees that we actually have to deal with an operator: **`+`**.

```
if (croot->data.get_type() ==
  token::type::TT_OPERATOR)
{
  traverse_ast(level + 1, croot->left, c);
  (*c->generator()) << MOV() << reg(level)
    << reg(level + 1);
  traverse_ast(level + 1, croot->right, c);
  (*c->generator())
    << operators.at(croot->data.data())->opcode
    << reg(level) << reg(level + 1) ;
}
```

Listing 4

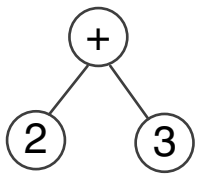


Figure 6

It enters the **if** above, and it will immediately start descending into the tree on the next level, towards the left side of the tree. Now the current node points towards the **2**. The code sees that the data is actually a number, so it enters here:

```
if (tt == token::type::TT_NUMBER)
{
  (*c->generator()) << MOV() << reg(level)
    << croot->data;
}
```

Now the following piece of code will be generated (which has the significance of initializing the register number 1 to 2):

```
MOV $r1 2
```

Since there is nothing more to be done here, we go back to the previous step (we were inside the **TT_OPERATOR if** branch), where the following instruction waits for us:

```
(*c->generator()) << MOV() << reg(level)
  << reg(level + 1);
```

which will generate:

```
MOV $r0 $r1
```

ie. move the content of register 1 into register 0. Don't forget that register 1 was initialized to 2 a few lines above. Now, we descend the tree to the right branch, where we see the **3** thus giving us the following:

```
MOV $r1 3
```

and we are done with that part too. Back to the inside of the **if** for **TT_OPERATOR** where the next operation is:

```
(*c->generator())
  << operators.at(croot->data.data())->opcode
  << reg(level) << reg(level + 1)
```

This will go searching in the global **operators** table for the operator which is at the current level in the tree, and it will find the following row:

```
("+",  util::make_unique<ops> ("+", PRIO_10,
  new opcodes::ADD))
```

where the **opcodes::ADD** is the opcode responsible for executing addition in the system. So it will generate the following assembly code sequence:

```
ADD $r0 $r1
```

Since we are done with the tree, we are also done with the compilation, and we just exit. And at a very high level, this is the logic based on which the compiler is built.

There is just one drawback for this entire mechanism: since the virtual machine was built to have 256 register (see below) at some point in very complex expressions, we simply might run out of registers. A resolution for future versions would be to implement the inclusion of a temporary variable in the code generation where the partial results are accumulated, but I reserve this for future enhancements of the compiler.

The assembly compiler

The script provides the programmer with the possibility of writing virtual-machine-specific assembly instructions. The keyword responsible for this is **asm** and the implementation is found in `kw_asm.cpp`. The build system generates a set of compiler files for the registered opcodes (please see below on how to register new opcodes) and a special component called the **asm_compiler** found in `asm_compiler.h` will generate the required code for the assembly instruction.

How to deal with the write function calls

As mentioned above, the function **write** is added to the system (see its source code in Appendix 1) as a way to output data onto the screen. The **write** function is a dynamic function, meaning it can take in a various number and type of arguments (strings, variables and numbers) so I had to come up with a way to make it flexible enough to be usable.

One way to deal with it would have been the C way with a format string where various format specifiers are representing various interpretations of the data, but personally I'm not a great fan of it. Another way of dealing with this situation is presented by the Pascal compiler: while printing out, each type printed out is represented by a different function being called, and the compiler generates a list of method calls for all the parameters. Simple and effective. But unfortunately, at this moment I don't have the infrastructure to support this.

So, I went in a third direction with this. With the help of the stack, I specify the number of parameters the function is having, then their type (string constant/number for now) and then the actual parameters. Now all is needed is just a clever write function which interprets the data from the stack, and all is done.

The output of the compiler

The compiled application has a very simple format. The bytecode starts with **.P10** representing the version of the script, 4 bytes follow as reserved, 4 more bytes representing the number of global variables and 4 more bytes to indicate the start of the string table of the application (or, depending on how we view it, the size of the compiled bytecode).

The constant string table, which is populated by the compiler upon encountering a string in the source, is to be found in the generated code after the application bytes. The string representation in this part always starts with an 8-bit value, representing the length of the string followed by the ASCII characters themselves.

This output is then read and interpreted by the virtual machine, which sets up its structures and starts executing the application code from the first application bytecode.

Adding a new keyword to the language

In its current incarnation, the script is quite … primitive, and lacks support for most of the keywords and provided functionality we are used to from other scripting languages. But extending the script is actually quite easy. When introducing a new keyword, we will need to provide the parsing implementation and the compilation ourselves. Let's consider implementing the **while** keyword as an exercise.

We can get some inspiration from the **if** keyword, since the difference between **while** and **if** is just a jump back to testing again the trueness of an expression, so let's start and declare the header file for **while** in the `keywords` directory of the compiler, with the content in Listing 5 (I have omitted things like include guard and base class includes).

And some explanation:

- A keyword must inherit from sequence in order to have access to the **prepare** and **compile** methods that actually perform the parsing of the keyword and the compilation of the keyword.

```
namespace primal
{
  class kw_while : public sequence, public keyword
  {
  public:
    static constexpr const char* N= "WHILE";
    explicit kw_while(source& src) : sequence(src)
    {}
    sequence::prepared_type
      prepare(std::vector<token>& tokens) override;
    bool compile(compiler* c) override;
    std::string name() override { return N; }
  private:
    std::vector<std::shared_ptr<sequence>>
      m_while_body;
  };
}
```

Listing 5

- A keyword must inherit from the **keyword** class in order to be included in the keyword store of the application.
- The **kw_while::N** must have the name of the keyword as present in the scripting language's syntax in uppercase. At some point it is used to compare in a cases insensitive comparison, which means the scripting language itself is case insensitive. Just like Pascal.
- Since this keyword will have a body, we need a list of sequences that will represent the body of it, hence the **m_while_body**.

Now the implementation of the **while** keyword can commence with the parsing of the new syntax (see Listing 6).

The **prepare** of the **while** keyword is very similar to the **prepare** of the **if** keyword, with the difference being that since we have no **then** in the **while**, that part is missing. Parsing out the body of the **while** is identical to the **if**, ie. read and parse lines till we encounter the **END** keyword. At this level, you don't have to worry about the expression of the **while** (like in: **while expression**) as parsing it is taken care of by the layer calling it; however, if you wish to inspect it, feel free to use and consume the **tokens** parameter. If you wish to notify the parsing layer not to try to parse the expressions after your keyword has done its work on it, simply return **sequence::prepared_type::PT_CONSUMED** instead of **PT_NORMAL** as we do here.

And the compilation of the **while** keyword is shown in Listing 7.

The compilation begins with declaring a label to the location where the evaluation of the **while**'s expression will be performed. Then we do the actual compilation of the expression via **sequence::compile**. The rest till **(*c->generator()) << DJMP() << lbl_while;** is actually the looping in the **while** loop itself, and is identical to the **if** keyword, as presented above.

```
sequence::prepared_type
kw_while::prepare(std::vector<token> &tokens)
{
  if(tokens.empty())
  {
    return sequence::prepared_type::PT_INVALID;
  }
  parser p;
  std::string last;
  auto seqs = p.parse(m_src,[&](std::string s)
  {
    return util::to_upper(s) == kw_end::N;
  },
  last);
  m_while_body = std::get<0>(seqs);
  return sequence::prepared_type::PT_NORMAL;
}
```

Listing 6

```
bool kw_while::compile(compiler* c)
{
  label lbl_while = label::create(c->get_source());
  (*c->generator()) << declare_label(lbl_while);
  sequence::compile(c);
  label lbl_after_while =
    label::create(c->get_source());
  label lbl_while_body =
    label::create(c->get_source());
  comp* comparator =
    dynamic_cast<comp*>
    (operators[m_root->data.data()].get());
  if(!comparator)
  {
    throw syntax_error("Invalid WHILE statement
      condition. Nothing to compare");
  }
  (*c->generator()) << comparator->jump
    << lbl_while_body;
  (*c->generator()) << DJMP() << lbl_after_while;
  (*c->generator())
    << declare_label(lbl_while_body);
  for(const auto& seq : m_while_body)
  {
    seq->compile(c);
  }
  (*c->generator()) << DJMP() << lbl_while;
  (*c->generator())
    << declare_label(lbl_after_while);
  return false;
}
```

Listing 7

Now, only one step remains: writing a unit test for it. The project uses Catch2 as its unit test framework, so into the `tests.cpp` found in the `tests` directory add the contents of Listing 8.

The virtual machine

The general design and architecture of the virtual machine is loosely based on the workings of a Turing machine: it relentlessly behaves in a similar manner to a computer adhering to the von Neumann architecture, it feels more like a real mode x86 machine thus suffering from all its deficiencies and efficiencies, it has a dedicated assembly level programming language and it contains the following important components:

- A set of 256 registers, integers, 32 bit by default represented in the source code by the type **word_t** found in `numeric_decl.h`.
  - Reg 255 is reserved for the stack pointer

```
TEST_CASE("Compiler compiles, while test",
"[compiler]")
{
  auto c = primal::compiler::create();
  c->compile(R"code(
              var a,b
              let a = 5
              let b = 0
              while a > 0
              let a = a - 1
              let b = b + 1
              end
            )code"
          );
  auto vm = primal::vm::create();
  REQUIRE(vm->run(c->bytecode()));
  REQUIRE(vm->get_mem(0) == 0);
  REQUIRE(vm->get_mem(word_size) == 5);
}
```

Listing 8

```
MOV $r1 12       # initialize register 1 to 12
MOV [0] $r1      # load into memory, at location 0 the value of register 1
MOV $r6@0 12     # load 12 into register 6's lowest byte
MOV $r1@1 [@$r2] # load into byte 1 of reg 1 the value in memory at register 2's value
MOV $r3@0 [@0]   # load into byte 0 of reg 3 the byte value in memory at addr 0
```

Listing 9

- Reg 254 is frequently used by the compiler
- Reg 253 holds the value of the LOF flag (see below)
- Reg 252 is initialized to the start of the stack segment
- Reg 251 is initialized to the entry point of the application upon the start of the application
- Reg 250 is reserved for future endeavours

All other registers are freely available for you to play with.

- A flag (called LOF – **L**ast **O**peration **F**lag) which is set to a non-zero value by the last operation if the value after the operation evaluated to non-zero and is cleared to zero by the first conditional jump command.
- A code segment which is the application's compiled byte-code. The size of this is flexible and depends on the application compiled. This is a read-only section and its content populated upon start-up by the virtual machine 'loader'. For the sake of completeness, we have to mention that the virtual machine places the code segment in memory after the memory segments 1MB limit, continuously.

  The machine has an Instruction Pointer to track the next instruction that needs to be executed. Initially, this is placed on the very first byte-code, and on stepping through the code, this can increase or decrease. When the virtual machine encounters errors in byte-code being executed, it issues a PANIC call and exits.
- A free-memory segment, which is used by the virtual machine and the code of the programmer. Usually it's 1MB in size; however, it can be modified by specifying the size in the `hal.h` header. At this moment, only assembly level access (r/w) is granted to this area via the **MOV** and **COPY** instructions, since the compiled script makes no use of it. The virtual machine will allocate this memory segment on the heap of the real machine.
  - The memory can be accessed by loading the desired address value into a register and then performing the required memory access operation: either a full 32 bit value will be loaded from the memory into the required register either a byte value will be moved in the required register target.
  - A memory area (usually beginning after the end of the global variables) is designated to be functioning as a stack which can be controlled via the **PUSH**/**POP**/**MOV** commands. The assembly instructions **CALL** and **RET** are also utilizing the stack, **CALL** pushes the address where the code should continue when a **RET** was issued. **POP** or **RET** in case of the stack pointer being 0 will result in **PANIC**.

    There is a Stack Pointer (same variable type as the registers, referenced in register with index 255) that is used to access the stack. Upon a push, the stack pointer increases its value, and upon pop, it decreases it. The assembler syntax for the stack pointer is: **$sp**
  - An important compatibility note: due to the way the direct jump addresses are calculated, both the compiler and vm need to be compiled using the same memory size.
- A set of 256 interrupts, but only one of them is used at the moment to handle writing to the screen, the other ones are available for adventurous programmers to experiment with.

The opcodes of the virtual machine

In order to perform the lowest possible level of operations, the primitive virtual machine has a set of opcodes that are acted upon by the VM. Following is a brief presentation of the opcodes which for the technically versed should be immediately familiar due to the heavy influence of the Intel family of opcodes.

Target specifiers

Please note that in the binary stream (some of) the values (the parameters to the assembly commands) are prepended by a type specifier which indicates how to interpret the bytes following the specifier. There is a specific assembly syntax for the various specifiers.

The specifier is always 1 byte, and it can be:

- **0x00** – the following 4 bytes are to be interpreted as an immediate number, negative or positive. Assembler syntax: **12345** (ie: just a normal number)
- **0x01** – the following 1 byte is the index of a given register. Assembler syntax: $**r123** for register 123.
- **0x02** – this represents the value of the memory at the register's value in the following byte. The syntax: **[$r123]** will give the byte value stored at the value of register 123 in the memory.
- **0x4X** – the following 1 byte is the index of a register, and the nibble marked with X is the index of the byte in the given register. Assembler syntax: **$r12@1** meaning byte 1 from register 12. For 32 bit virtual machines **@0**, **@1**, **@2**, **@3** are valid byte indexes.
- **0x05** – the coming 4 bytes represent a number, and its value represents an address in the memory. Syntax: **[1234]**. This is mostly used by the compiler to index in variables from the memory.
- **0x06** – the coming 4 bytes represent a number, and its value represents the address of 1 byte in the memory. Syntax: **[@1234]**
- **0x07** – this represents the value of the memory at the register's value in the following byte + the added offset following that. The syntax: **[$r123+4]** will give the byte value stored at the value of register 123 + 4 in the memory and **[$r123-4]** will give the byte value stored at the value of register 123 - 4 in the memory. The four basic operations (addition, multiplication, division, subtraction) are possible in this context.
- **0x08** represents the value of the memory at the register's value in the following byte + the following register's value. The syntax: **[$r123+$r245]** will give the byte value stored at the value of register 123 + value of register 245. The four basic operations (addition, multiplication, division, subtraction) are possible in this context.

MOV

The **MOV** opcode moves data from the given source to the given destination.

The following is the syntax of the command:

```
MOV <DST> <SRC>
```

where the syntax of **<DST>** and **<SRC>** correspond to specific syntax in the target specifier list. **MOV** makes sure that only sane operations are permitted, and the machine will stop execution with a **PANIC** command if an invalid operation is attempted for execution (for example trying to load a value into an immediate number). One of **<DST>** or **<SRC>** must be a register. For example, see Listing 9 (overleaf).

ADD, SUB, MUL, DIV, MOD, OR, XOR, AND and NOT

The assembly opcodes with these very descriptive names perform the requested operation on two parameters following them. The syntax of the

parameters is identical to the parameters of **MOV**, so you can just refer to that section when writing your assembly code should you require it.

NOT requires only one parameter and it will do a binary negation of it, which cannot be an immediate value. If any of the operations results in a zero value, the **LOF** flag will be set to false, otherwise to true.

COPY

COPY is used to copy data between two addresses in the memory segment. **COPY** can be imagined as multiple **MOV** commands where both the destination and source are at increasing locations.

The syntax of the command is:

```
COPY <DEST> <SRC> <COUNT>
```

where **DEST** is interpreted as a numerical value representing the address of the area in the memory, **SRC** represents the source address and **COUNT** is the number of bytes to copy. If **DEST** + **COUNT** > **MEM_SIZE**, the virtual machine will issue a **PANIC** call and not perform the operation. At the implementation level, **std::memmove** is used so it is possible to have overlapping memory areas for this command. This operation does not modify the content of the **LOF** flag.

EQ, NEQ, LT, GT, LTE and GTE

The comparison operators perform two operations. The first one is to compare the value of the two parameters they get, and the second one is to set the value of the **LOF** flag to be either **true** or **false**.

JMP, JT and JNT

The direct jump commands will set the IP of the virtual machine to the specified parameter and will continue the execution of the code from the new location.

- The **JMP** command will jump to the given address regardless of any surrounding conditions.
- The **JT** will jump only if the VM's **LOF** flag was set to **true** by the previous comparison operation, and it will clear the flag simultaneously.
- The **JNT** command will jump to the given location only if the flag was set to **false**, and it also will clear the flag to **false**.

The syntax of the commands is:

```
JMP | JT | JNT <ADDRESS>
```

where **ADDRESS** will be interpreted as the address where the execution will be continued. The programmer needs to take into consideration that **ADDRESS** is actually a valid destination, where valid, executable code is to be found.

DJMP, DJT and DJNT

The delta jump commands, such as the 'direct jump', 'jump if last comparison was true' and 'jump if last comparison was not true' commands, jump from the current location in various directions and the new address determined as being at a specific bytes away from the current location. The syntax is:

```
DJMP | DJT | DJNT <DELTA>
```

where **DELTA** is to be interpreted as the *difference* between the current IP (instruction pointer) and the upcoming new location. At this point of execution (when **DST** was evaluated by the jump command) the IP points to the next executable operation, this needs to be taken into consideration when manually calculating jump addresses.

As a side note, the jump commands cannot jump to labels since they all expect a number. In order to jump to a specific label, you will have to use **goto**.

PUSH and POP

As guessed from the name, **PUSH** and **POP** work with the stack. **PUSH** always pushes an immediate value onto it, with the type modifier 0 for numeric data or 7 for string indexes, and **POP** pops the value into the parameter it was required to pop into. The syntax is: **PUSH <SOMETHING>** and **POP <SOMETHING>**.

Internals of the VM

Like every other code interpreter out there, our VM is performing very similar operations:

1. Load the next instruction
2. The code for the instruction loads its parameters
3. The code for the instruction executes the instruction
4. Check for failure
5. Repeat from step 1.

A big part of the virtual machine's code (specifically automatic handling of the registered opcodes) is generated by the build system (see below at the extending the VM) and the remaining of the operations are distributed between the data types representing the VM's data structures and the actual operations the VM can perform.

The virtual machine's basic data structures are declared in the header `registers.h`. This header has a basic data type **primal::valued**. The class acts as a common aggregate for the entities that can be handled by the virtual machine (see Figure 7, overleaf).

Upon each execution step, the instruction (opcode) needs to fetch its parameters from the memory of the virtual machine (the opcode can have zero, 1, 2 or 3 parameters for now) and perform the most basic of operations on the data types. The overloaded operators in the valued class more act as commodities, the most important operations are the **word_t value()** and the **void set_value(word_t)**. This class hierarchy makes it very easy to implement basic operations, for example this is the implementation of the **ADD** opcode (found in `impl_ADD.h`):

```
bool primal::impl_ADD(primal::vm* v)
{
  primal::valued* dest = v->fetch();
  primal::valued* src  = v->fetch();

  *dest += *src;

  v->set_flag(dest->value() != 0);

  return true;
}
```

Extending the virtual machine

The project was created with extendibility in mind, and nothing is easier than adding a new binary opcode and the corresponding implementation, or providing your own interrupt to deal with tasks.

Adding a new opcode

As you might have observed, we have not introduced the INC operation when designing the virtual machine (along with other missing operations, which hopefully will be implemented at a later stage), but no worries – let's read through the following paragraph in order to see how you can implement your own opcodes to extend the virtual machine.

We will need to do modifications to the build system and also add a new file to it in order to have the new opcode up and running. You have to open the `CMakeLists.txt` from the `opcodes` directory around line 35 to find the registration of the opcodes section.

Go to the end of the section, and after the last **register_opcode** call insert yours:

```
register_opcode("INC" 0xEE 1 OF_ARITH)
```

With this you have told the build system the following facts:

- It will have an opcode called **INC**
- When in the binary stream, it will be represented by the value **0xEE**
- It expects one parameter, ie. the one which will be incremented.
- It is of type arithmetic

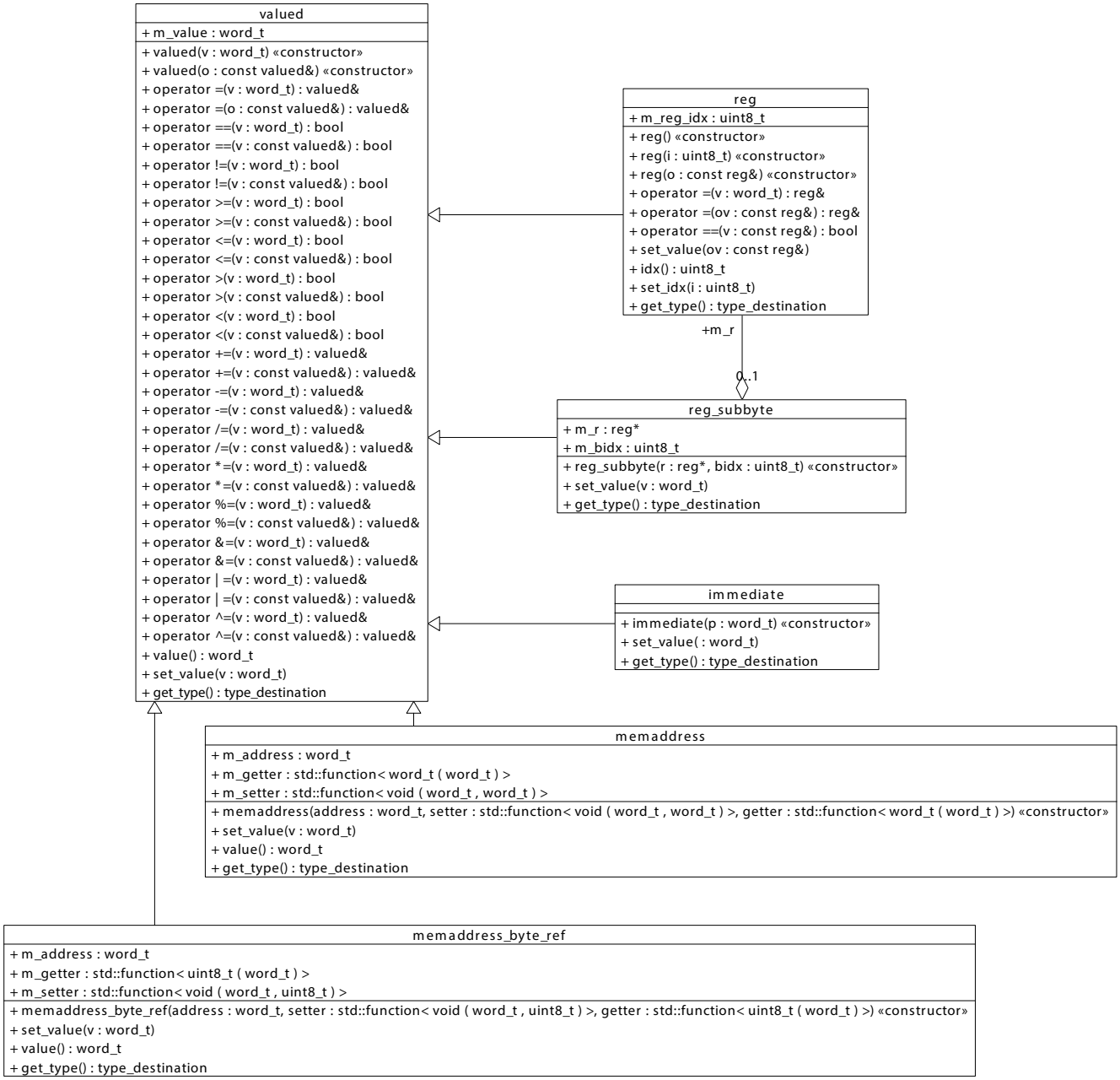


Figure 7

What happens behind the scenes is that **CMake** generates two files for you:

- One is the opcodes header file `INC.h` in the `build/opcodes` directory (see Listing 10), which has the declaration of the class INC and also two functions.

```
struct INC final : public opcode
{
  uint8_t bin() const override {return 0xEE; }
  std::string name() const override {return "INC";}
  size_t paramcount() const override { return 1; }
  virtual opcode_family family() const override
  { return
    primal::opcodes::opcode_family::OF_ARITH; };
};

bool impl_INC(vm*);
std::vector<uint8_t>
  compile_INC(std::vector<token>&);
```

Listing 10

- The other file is `compile_INC.cpp`, generated in the same directory, which will be used by the compiler when encountering the **INC** opcode in the assembly code of your source file.

The two functions in the header file are:

- **compile_INC**, which is responsible for compiling the assembly statements into corresponding (virtual) machine code and it will be provided by the build system in the `compile_INC.cpp` file
- **impl_INC**, which actually tells the virtual machine what to do upon encountering the **INC** in the binary stream, needs to be provided by us in the `opcodes/impl/impl_INC.cpp` file, because this is where the build system will look for it.

  So create that file and paste in the following:

```
#include <INC.h>
#include <vm.h>
bool impl_INC(vm* v)
{
  valued* dest = v->fetch();
  dest->set_value(dest->value() + 1);
  return true;
}
```

```
TEST_CASE("ASM compiler - INC opcode", "[asm-
compiler]")
{
  auto c = compiler::initalize();
  c->compile(R"code(
              asm MOV $r1 10
              asm INC $r1
            )code"
  );
  auto vm = vm::create();
  REQUIRE(vm->run(c->bytecode()));
  REQUIRE(vm->r(1).value() == 11);
}
```

Listing 11

After you regenerate your CMake cache, and once the build of the system is ready, everything should be up and running, and you can use **INC** in your assembly commands. So maybe it's time to write a unit test for this purpose too (Listing 11).

Now if you run **make test**, your newly created opcode should be up, running and incrementing values you want to.

How the opcodes are registered into the VM

When you have declared a new opcode, with the **register_opcode** function in the build system in the background a CPP file (`opcode-impl.cpp`) is generated with the contents similar to Listing 12 for each of the opcodes. In the VM's implementation class (`vm_impl.h`) this boils down Listing 13.

Now in its turn, the VM's opcode runner has the code in Listing 14.

And in short, this sums up how the VM is handling the execution of the various opcodes.

Writing a new interrupt

The interrupts are pieces of code that are compiled C++ code, and they can be used by the VM to communicate with the world outside the sandbox.

```
namespace primal {
  void register_opcodes() {
    vm_impl::register_opcode
      (primal::opcodes::MOV(),
        [&](primal::vm* v) -> bool
        { return primal::impl_MOV(v); }
      );
    vm_impl::register_opcode
      (primal::opcodes::ADD(),
        [&](primal::vm* v) -> bool
        { return primal::impl_ADD(v); }
       );
```

Listing 12

```
struct executor
{
  std::function<bool(vm*)> runner;
};
template<class OPC, class EXECUTOR>
  static void register_opcode(OPC&& o,
  EXECUTOR&& ex)
{
  auto f = [&](vm* machina) -> bool {
    return ex(machina);};
  executor t;
  t.runner = std::function<bool(vm*)>(f);
  opcode_runners[o.bin()] = t;
};
static std::map<uint8_t, executor> opcode_runners;
```

Listing 13

```
while(opcode_runners.count(ms[static_cast<size_t>
  (m_ip)]))
{
  // read in an opcode
  uint8_t opc = ms[static_cast<size_t>(m_ip++)];
  try
  {
    // is there a registered opcode runner for
    // the given opcode?
    if(!opcode_runners[ opc ].runner(v))
    {
      panic();
    }
  }
  catch(...)
  {
    panic();
  }
  if(ms[static_cast<size_t>(m_ip)] == 0xFF)
  {
    return true;
  }
}
```

Listing 14

For the moment, only one interrupt is in use, which is **INT 1** used to write to the screen. In order to implement a new interrupt, you will have to modify the `CMakeLists.txt` of the virtual machine in the section where it states 'Register the interrupts' add the interrupt with a new call to **enable_interrupt**, for example: **enable_interrupt(2)**.

Now, create the implementation file for the interrupt in `vm/intr` and call it `intr_2.cp` As a start, the content of it can be:

```
#include <vm.h>
namespace primal
{
  bool intr_2(vm* v)
  {
    return true;
  }
}
```

And from this point on, you can provide the implementation for your interrupt. You can use the interface exposed by `vm.h` in order to get access to the VM internals and memory.

Binding it together

As I mentioned in the introduction, our ultimate goal in this article is to have it all in one place, ie: in one CPP file we compile a piece of primal script code and let it run in the primal VM.

Using the compiler to compile and the vm in your own source files is pretty easy, you just need a few instructions.

To properly use it in your own code, you will need a sequence of instructions like:

```
std::shared_ptr<primal::compiler> c
  = primal::compiler::create();
```

This will create for you the object **c** which is a primal script compiler that you can use to compile code in the following manner:

```
c->compile(R"code(
    let x = 40
  )code"
);
```

The next step you need to do is to create a virtual machine:

```
std::shared_ptr<primal::vm> vm
  = primal::vm::create();
```

It is possible to use **auto** for the return type: I just wanted to show the actual return type. Now that you have the compiled code in the compiler and the virtual machine up and running, nothing is easier than to run it:

```
vm->run(c->bytecode())
```

After this point, you have full access to the memory and registers of the virtual machine with the proper function calls found in class **primal::vm** located in `vm.h`:

- **word_t get_mem(word_t address)** – to get the 32/64 bit numeric value from the memory of the virtual machine from the specified address.
- **uint8_t get_mem_byte(word_t address)** – to get the 8 bit value from the memory of the virtual machine from the specified address.
- **const reg& r(uint8_t i) const** – to get the object representing the ith. register of the VM.

Conclusion

Working on this project was really fun, and I have learned a lot during the implementation. However, progress does not stop here. There are a lot of features in the compiler to be implemented, new keywords, proper register handling, and – why not – even some optimizations that can be introduced. The VM needs good profiling in order to pinpoint the bits that can be optimized, and a lot of other features can be included too in the entire ecosystem. While designing this whole universe, I made it as modular as possible with the thought that some day, someone might have time to work on it, and include the features he/she considers nice to have. So, if you feel that you could contribute in any way, feel free to get in touch and start coding on it in the spirit of open source. ■

References

[CompOpt] Compiler optimizations: http://compileroptimizations.com/

[Dijkstra61] EdsgerW.Dijkstra (1961) *ALGOL Bulletin Supplement no. 10*: http://www.cs.utexas.edu/~EWD/MCReps/MR35.PDF

[Dragon] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman (2006) *Compilers: Principles, Techniques, and Tools*

[GCCArchitecture] GNU C Compiler Architecture: https://en.wikibooks.org/wiki/GNU_C_Compiler_Internals/GNU_C_Compiler_Architecture

[Guntli11] Christopher Guntli (2011) 'Architecture of clang', https://wiki.ifs.hsr.ch/SemProgAnTr/files/Clang_Architecture_-_ChristopherGuntli.pdf

[Knuth65] Donald E. Knuth (July 1965). 'On the translation of languages from left to right'. *Information and Control* 8(6): 607 - 639. doi: 10.1016/S0019 - 9958(65)90426-2

[LLVM] The LLVM Compiler Infrastructure: https://llvm.org/

[Wikipedia-1] Compiler optimizations: https://en.wikipedia.org/wiki/Category:Compiler_optimizations

[Wikipedia-2] Lexical analysis: https://en.wikipedia.org/wiki/Lexical_analysis

[Wikipedia-3] Multi-pass compiler: https://en.wikipedia.org/wiki/Multi-pass_compiler

[Wikipedia-4] Parser generators: https://en.wikipedia.org/wiki/Comparisonofparser_generators

[Wikipedia-5] Regular expression: https://en.wikipedia.org/wiki/Regular_expression

Appendix – The write function

As promised, Listing 15 is the write function you can use to display data. Save it under the name `write.prim` and place it in the directory where your compiler and source file is, for the moment. Don't forget to include it in your project.

```
fun write(...)
  asm MOV $r249 $r255
  # Decrease the stack pointer to skip the pushed
  # R254 and the return address.
  # This is for 32 bit builds. For 64 bit builds,
  # use 16
  asm SUB $r255 8

  # First: the number of parameters that came in
  asm POP $r10

:next_var
  # fetch the value that needs to be printed
  asm POP $r2

  # This $r1 will contain the type of the variable:
  # 1 for string, 0 for number
  asm POP $r1

  # Is this a numeric value we want to print?
  asm EQ $r1 0

  # If yes, goto the print number location
  asm JT print_number

  # else goto the print string location
  asm JMP print_string

:print_number
  # print it out
  asm INTR 1

  # Move to the next variable
  asm SUB $r10 1

  # JT is logically equivalent to JNZ
  asm JT next_var

  # Done here, just return
  asm MOV $r255 $r249
  asm JMP leave

:print_string
  # Here $r2 contains the address of the string,
  #first character is the length

  # Initialize $r1 with the length
  asm MOV $r1 0
  asm MOV $r1@0 [$r2+$r251]

  # Get the address of the actual character data
  asm ADD $r2 1

  # Print it
  asm INTR 1

  # Move to the next variable
  asm SUB $r10 1

  # JT is logically equivalent to JNZ
  asm JT next_var

  # Done here, just return
  asm MOV $r255 $r249
:leave
end
```

Listing 15

QM Bites: Understand Windows Operating-System Identification Preprocessor Macros

Quality matters and bite sized articles help. Matthew Wilson returns with a QM Bites.

TL;DR

Compiler defines **_WIN32** and **_WIN64**; you define **WIN32** or **WIN64**. Carefully discriminate.

Bite

When compiling for Windows 32 and 64-bit architectures, there are four main preprocessor object-like macro definitions for discriminating operating system (not architecture) that one may encounter:

- **_WIN32**
- **_WIN64**
- **WIN32**
- **WIN64**

You must take care that you understand the origins and meanings of these.

_WIN32 and _WIN64

The symbol **_WIN32** is defined *by the compiler* to indicate that this is a (32-bit) Windows compilation. Unfortunately, for historical reasons, it is also defined for 64-bit compilation.

The symbol **_WIN64** is defined *by the compiler* to indicate that this is a 64-bit Windows compilation.

Thus:

To identify unambiguously whether the compilation is 64-bit Windows one tests only **_WIN64** as in:

```
#if defined(_WIN64)
/* Is Windows 64-bit */
#else
/* Is not Windows 64-bit */
#endif
```

To identify unambiguously whether the compilation is 32-bit Windows one tests both **_WIN32** and **_WIN64** as in:

```
#if defined(_WIN32) && \
   !defined(_WIN64)
/* Is Windows 32-bit */
#else
/* Is not Windows 32-bit */
#endif
```

To identify unambiguously whether the compilation is one or the other form of Windows one tests both **_WIN32** and **_WIN64** as in:

```
#if defined(_WIN64)
/* Is Windows 64-bit */
#elif defined(_WIN32)
/* Is Windows 32-bit */
#else
/* Not Windows */
#endif
```

WIN32 and WIN64

The symbol **WIN32** is defined by the user to indicate whatever the user chooses it to indicate. By convention, the definition of this symbol indicates a 32-bit Windows compilation, and nothing else! Microsoft (and other) tools generate projects with this symbol defined.

The symbol **WIN64** is defined by the user to indicate whatever the user choose it to indicate. By convention, the definition of this symbol indicates a 64-bit Windows compilation, and nothing else!

When properly defined, these symbols can be used to indicate unambiguously the 32- and 64-bit Windows compilation contexts.

Caution with WIN32 / WIN64

Unfortunately, when duplicating a **Win32** project to **x64**, the Microsoft Visual Studio wizards do not translate **WIN32** to **WIN64**. You must remember to do this yourself, in order for the inferences given above to hold. Do not add a separate **WIN64** to the x64 configuration settings – replace the existing **WIN32** with **WIN64**. (All of this can be dealt with much better by use of props files, but that's a long article …)

Why bother with WIN32 / WIN64 (and not simply rely on _WIN32 / _WIN64)?

There are doubtless many reasons. The reasons I adhere strictly to this are:

- it is a widely adopted and meaningful convention, so adheres to the principle of least surprise [Raymond03];
- it facilitates the emulation of (parts of) other operating systems (e.g. UNIX [UNIXem]) while on Windows, which can be tremendously helpful when porting code. ■

References

[Raymond03] Eric S. Raymond (2003) *The Art of UNIX Programming* Addison-Wesley, 2003

[UNIXem] UNIXem is a simple, limited UNIX-API emulation library for Windows. See http://synesis.com.au/software/unixem.html

Matthew Wilson Matthew is a software development consultant and trainer for Synesis Software who helps clients to build high-performance software that does not break, and an author of articles and books that attempt to do the same. He can be contacted at matthew@synesis.com.au.

A Thorough Introduction to Distributed Systems

What is a distributed system, and why is it so complicated? Stanislav Kozlovski explains.

With the ever-growing technological expansion of the world, distributed systems are becoming more and more widespread. They are a vast and complex field of study in computer science.

This article aims to introduce you to distributed systems in a basic manner, showing you a glimpse of the different categories of such systems while not diving deep into the details.

What is a distributed system?

A distributed system in its most simplest definition is a group of computers working together as to appear as a single computer to the end-user.

These machines have a shared state, operate concurrently and can fail independently without affecting the whole system's uptime.

I propose we incrementally work through an example of distributing a system so that you can get a better sense of it all.

Let's go with a database! Traditional databases are stored on the filesystem of one single machine, whenever you want to fetch/insert information in it – you talk to that machine directly (see Figure 1, a traditional stack).

For us to distribute this database system, we'd need to have this database run on multiple machines at the same time. Users must be able to talk to whichever machine they choose and should not be able to tell that they are not talking to a single machine – if they insert a record into node#1, node #3 must be able to return that record. Figure 2 (overleaf) shows an architecture that can be considered distributed.

Why distribute a system?

Systems are always distributed by necessity. The truth of the matter is – managing distributed systems is a complex topic, chock-full of pitfalls and landmines. It is a headache to deploy, maintain and debug distributed systems, so why go there at all?

What a distributed system enables you to do is *scale horizontally*. Going back to our previous example of the single database server, the only way to handle more traffic would be to upgrade the hardware the database is running on. This is called *scaling vertically*.

Scaling vertically is all well and good while you can, but after a certain point you will see that even the best hardware is not sufficient for enough traffic, not to mention impractical to host.

Scaling horizontally simply means adding more computers rather than upgrading the hardware of a single one. Figure 3 (overleaf) shows that horizontal scaling becomes much cheaper after a certain threshold.

It is significantly cheaper than vertical scaling after a certain threshold but that is not its main case for preference.

Stanislav Kozlovski Stanislav has been programming since the age of 19 –and is now 22. He spent a year racing through some coding academies and bootcamps, where he aced all of his courses, and took a job at a Berlin company aiming to become the first ever global card acceptance brand. Contact him on github (where he's enether) or at Stanislav_Kozlovski@outlook.com

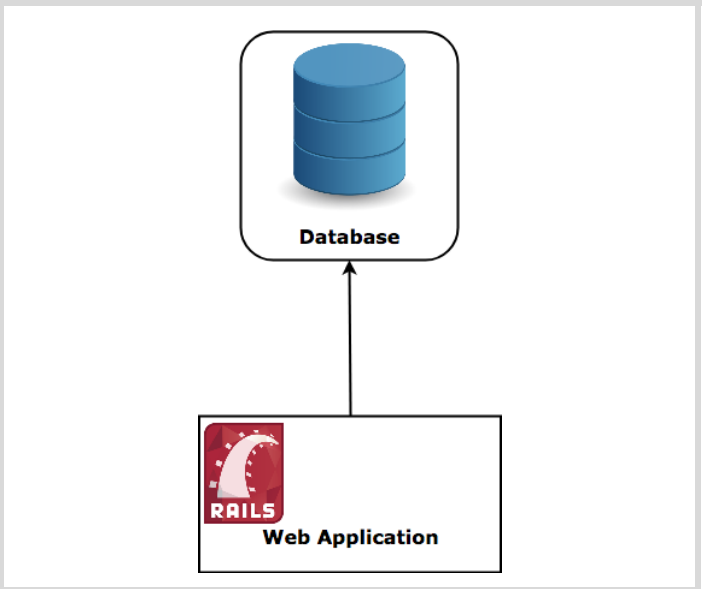


Figure 1

Vertical scaling can only bump your performance up to the latest hardware's capabilities. These capabilities prove to be *insufficient* for technological companies with moderate to big workloads.

The best thing about horizontal scaling is that you have no cap on how much you can scale – whenever performance degrades you simply add another machine, up to infinity potentially.

Easy scaling is not the only benefit you get from distributed systems. F*ault tolerance* and *low latency* are also equally as important.

- **Fault Tolerance** – a cluster of ten machines across two data centers is inherently more fault-tolerant than a single machine. Even if one data center catches on fire, your application would still work.
- **Low Latency** – The time for a network packet to travel the world is physically bounded by the speed of light. For example, the shortest possible time for a request's round-trip time (that is, go back and forth) in a fiber-optic cable between New York to Sydney is 160ms. Distributed systems allow you to have a node in both cities, allowing traffic to hit the node that is closest to it.

For a distributed system to work, though, you need the software running on those machines to be specifically designed for running on multiple computers at the same time and handling the problems that come along with it. This turns out to be no easy feat.

Scaling our database

Imagine that our web application got insanely popular. Imagine also that our database started getting twice as much queries per second as it can handle. Your application would immediately start to decline in performance and this would get noticed by your users.

The truth of the matter is – managing distributed systems is a complex topic, chock-full of pitfalls and landmines

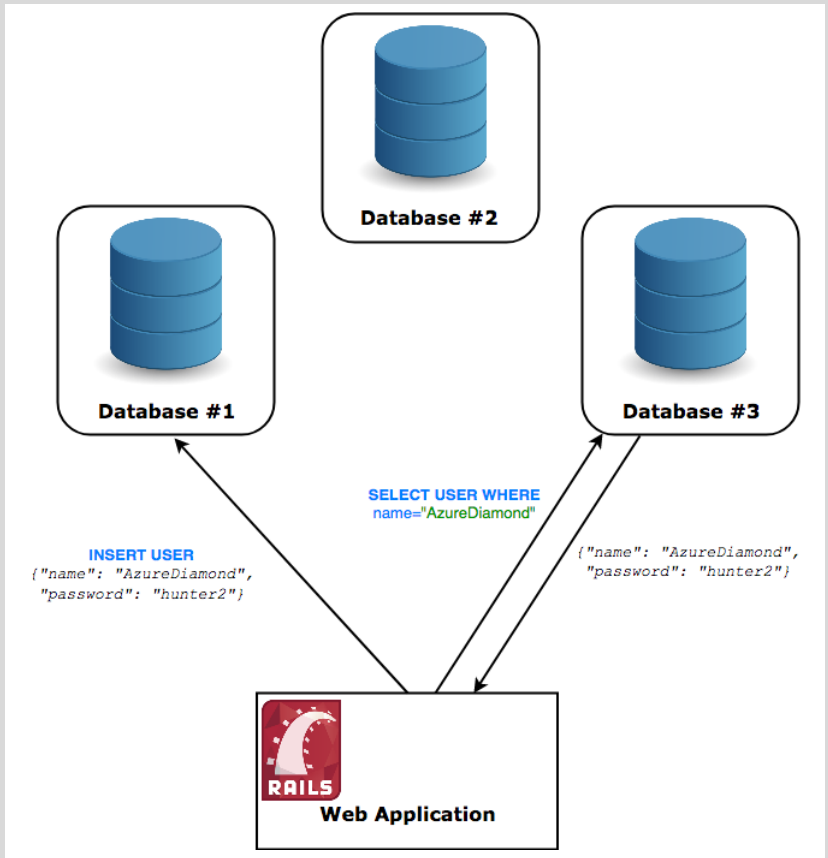


Figure 2

Let's work together and make our database scale to meet our high demands.

In a typical web application you normally read information much more frequently than you insert new information or modify old one.

There is a way to increase read performance and that is by the so-called *Master-Slave Replication* strategy. Here, you create two new database servers which sync up with the main one. The catch is that you can *only read* from these new instances. (See Figure 4, overleaf.)

Whenever you insert or modify information – you talk to the master database. It, in turn, asynchronously informs the slaves of the change and they save it as well.

Congratulations, you can now execute 3x as much read queries! Isn't this great?

Pitfall

Gotcha! We immediately lost the C in our relational database's *ACID* guarantees, which stands for *Consistency*.

You see, there now exists a possibility in which we insert a new record into the database, immediately afterwards issue a read query for it and get nothing back, as if it didn't exist!

Propagating the new information from the master to the slave does not happen instantaneously. There actually exists a time window in which you can fetch stale information. If this were not the case, your write performance would suffer, as it would have to synchronously wait for the data to be propagated.

Distributed systems come with a handful of trade-offs. This particular issue is one you will have to live with if you want to adequately scale.

Continuing to scale

Using the slave database approach, we can horizontally scale our read traffic up to some extent. That's great but we've hit a wall in regards to our write traffic – it's still all in one server!

We're not left with much options here. We simply need to split our write traffic into multiple servers as one is not able to handle it.

One way is to go with a multi-master replication strategy. There, instead of slaves that you can only read from, you have multiple master nodes which support reads and writes. Unfortunately, this gets complicated real quick as you now have the ability to create conflicts (e.g insert two records with same ID).

Let's go with another technique called *sharding* (also called partitioning).

With sharding you split your server into multiple smaller servers, called *shards*. These shards all hold different records – you create a rule as to what kind of records go into which

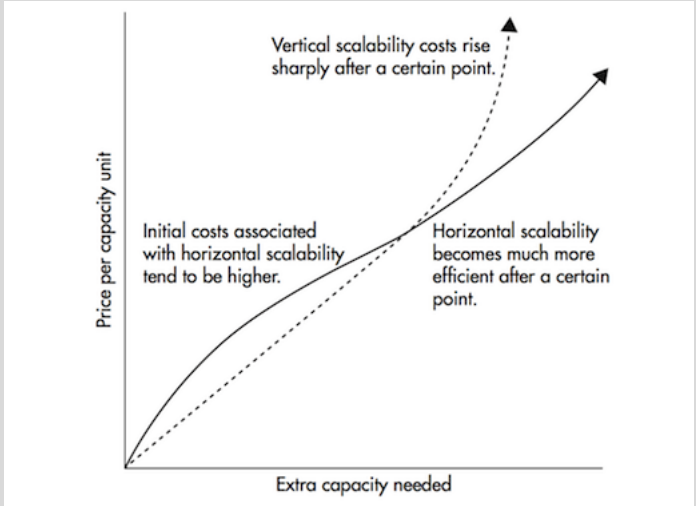


Figure 3

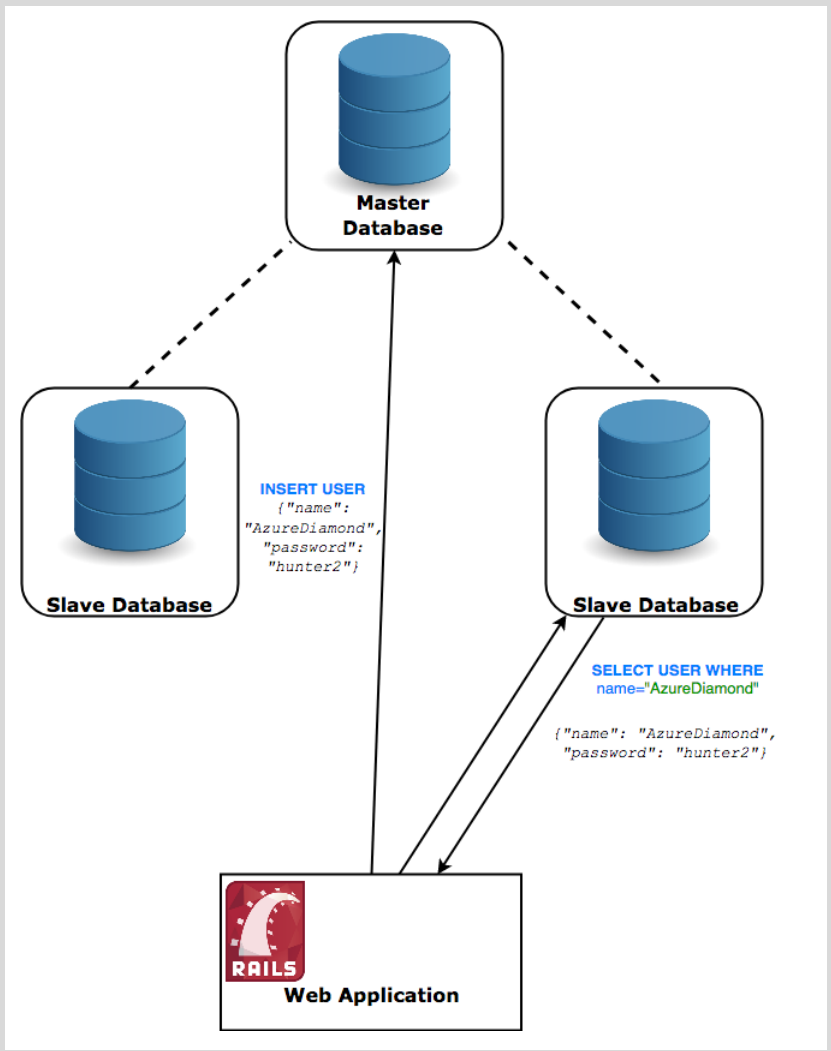


Figure 4

shard. It is very important to create the rule such that the data gets spread in a *uniform way*.

A possible approach to this is to define ranges according to some information about a record (e.g users with name A-D, as illustrated in Figure 5).

This sharding key should be chosen very carefully, as the load is not always equal based on arbitrary columns. (e.g more people have a name starting with C rather than Z). A single shard that receives more requests than others is called a *hot spot* and must be avoided. Once split up, re-sharding data becomes incredibly expensive and can cause significant downtime, as was the case with FourSquare's infamous 11 hour outage.

To keep our example simple, assume our client (the Rails app) knows which database to use for each record. It is also worth noting that there are many strategies for sharding and this is a simple example to illustrate the concept.

We have won quite a lot right now – we can increase our write traffic N times where *N* is the number of shards. This practically gives us almost no limit – imagine how finely-grained we can get with this partitioning.

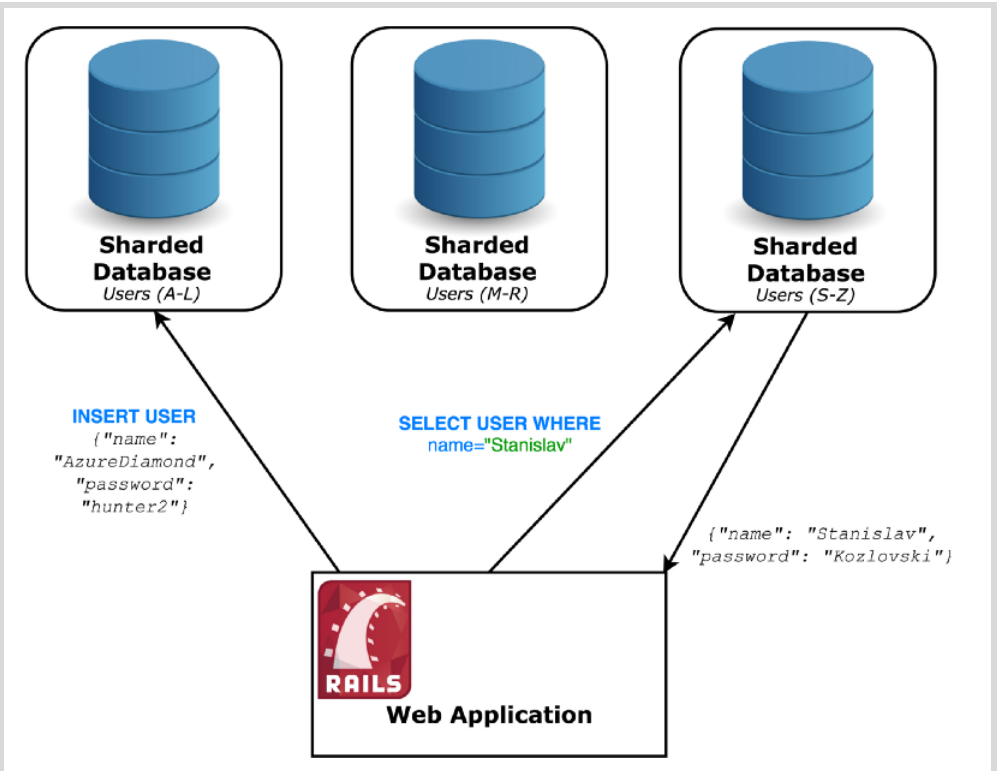


Figure 5

Pitfall

Everything in Software Engineering is more or less a trade-off and this is no exception. Sharding is no simple feat and is best avoided until really needed.

We have now made queries by keys *other than the partitioned key* incredibly inefficient (they need to go through all of the shards). SQL JOIN queries are even worse and complex ones become practically unusable.

Decentralized vs distributed

Before we go any further I'd like to make a distinction between the two terms.

Even though the words sound similar and can be concluded to mean the same logically, their difference makes a significant technological and political impact.

Decentralized is still *distributed* in the technical sense, but the whole decentralized systems is not owned by one actor. No one company can own a decentralized system, otherwise it wouldn't be decentralized anymore.

This means that most systems we will go over today can be thought of as *distributed centralized systems* – and that is what they're made to be.

If you think about it, it is harder to create a decentralized system because then you need to handle the case where some of the participants are malicious. This is not the case with normal distributed systems, as you know you own all the nodes.

Note: This definition has been debated a lot and can be confused with others (peer-to-peer, federated). In early literature, it's been defined differently as well. Regardless, what I gave you as a definition is what I feel is the most widely used now that blockchain and cryptocurrencies popularized the term.

Distributed system categories

We are now going to go through a couple of distributed system categories and list their largest publicly-known production usage.

Bear in mind that most such numbers shown are outdated and are most probably significantly bigger as of the time you are reading this.

Distributed data stores

Distributed data stores are most widely used and recognized as distributed databases. Most distributed databases are NoSQL non-relational databases, limited to key-value semantics. They provide incredible performance and scalability at the cost of consistency or availability.

> Known Scale – Apple is known to use 75,000 Apache Cassandra nodes storing over 10 petabytes of data, back in 2015

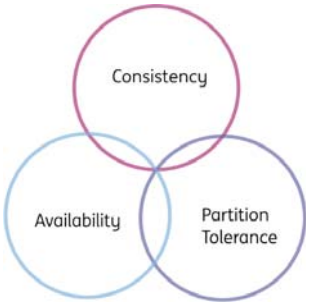
We cannot go into discussions of distributed data stores without first introducing the CAP Theorem.

CAP theorem

Proven way back in 2002, the CAP theorem states that a distributed data store cannot simultaneously be consistent, available and partition tolerant.

Choose 2 out of 3 (but not consistency *and* availability).

Some quick definitions:



- **Consistency** – What you read and write sequentially is what is expected (remember the gotcha with the database replication a few paragraphs ago?)
- **Availability** – the whole system does not die – every non-failing node always returns a response.
- **Partition Tolerant** – The system continues to function and uphold its consistency/availability guarantees in spite of network partitions

In reality, partition tolerance must be a given for any distributed data store. As mentioned in many places, one of which this great article, you cannot have consistency and availability without partition tolerance.

Think about it: if you have two nodes which accept information and their connection dies – how are they both going to be available and simultaneously provide you with consistency? They have no way of knowing what the other node is doing and as such have can either become offline (*unavailable*) or work with stale information (*inconsistent*).

What do we do? In the end, you're left to choose if you want your system to be strongly consistent or highly available *under a network partition* (see Figure 6).

Practice shows that most applications value availability more. You do not necessarily always need strong consistency. Even then, that trade-off is not necessarily made because you need the 100% availability guarantee, but rather because network latency can be an issue when having to synchronize machines to achieve strong consistency. These and more factors make applications typically opt for solutions which offer high availability.

Such databases settle with the weakest consistency model – *eventual consistency* (strong vs eventual consistency explanation). This model guarantees that if no new updates are made to a given item, *eventually* all accesses to that item will return the latest updated value.

Those systems provide BASE properties (as opposed to traditional databases' ACID)

- **B**asically **A**vailable – The system always returns a response
- **S**oft state – The system could change over time, even during times of no input (due to eventual consistency)
- **E**ventual consistency – In the absence of input, the data will spread to every node sooner or later – thus becoming consistent

Examples of such available distributed databases – Cassandra, Riak, Voldemort

Of course, there are other data stores which prefer stronger consistency – HBase, Couchbase, Redis, Zookeeper

The CAP theorem is worthy of multiple articles on its own – some regarding how you can tweak a system's CAP properties depending on how the client behaves and others on how it is not understood properly.

Cassandra

Cassandra, as mentioned above, is a distributed No-SQL database which prefers the AP properties out of the CAP, settling with eventual consistency. I must admit this may be a bit misleading, as Cassandra is highly configurable – you can make it provide strong consistency at the expense of availability as well, but that is not its common use case.

Cassandra uses consistent hashing to determine which nodes out of your cluster must manage the data you are passing in. You set a replication factor, which basically states to how many nodes you want to replicate your data. (Figure 7, overleaf, shows a sample write.)

When reading, you will read from those nodes only.

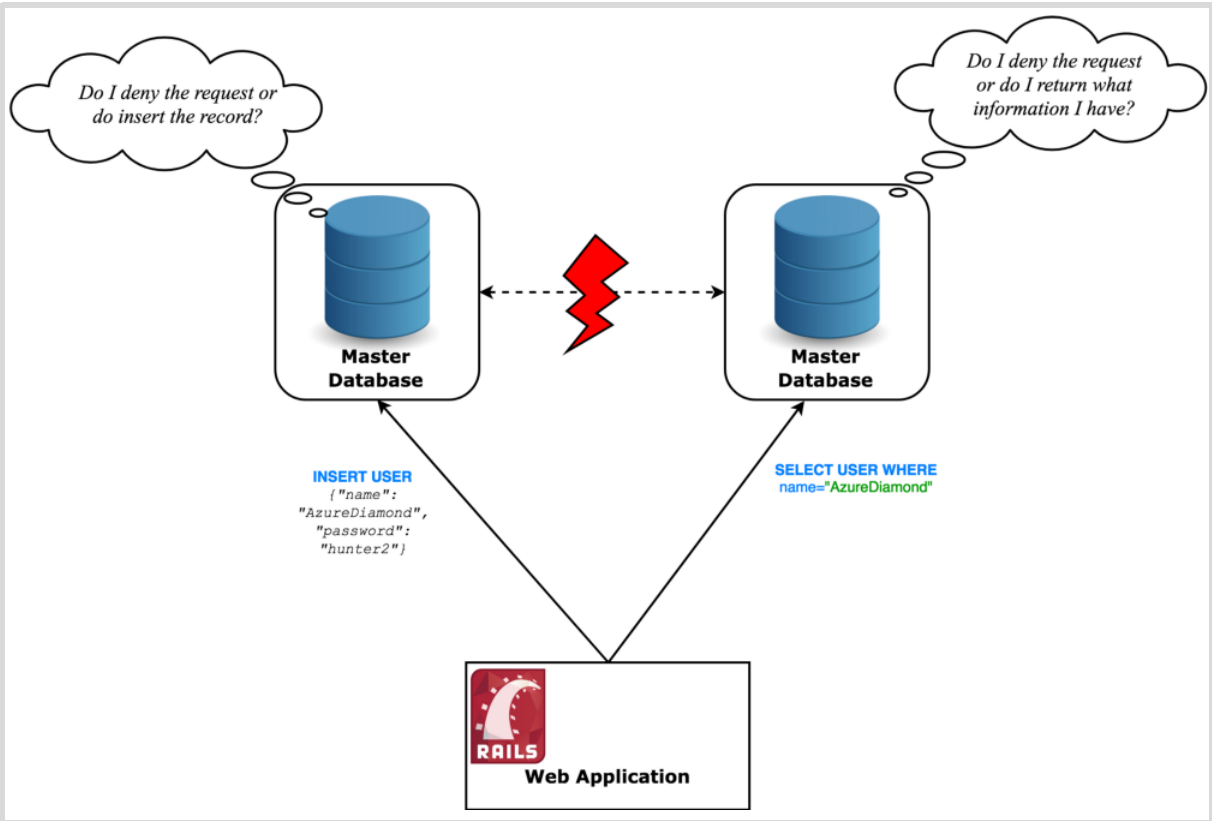


Figure 6

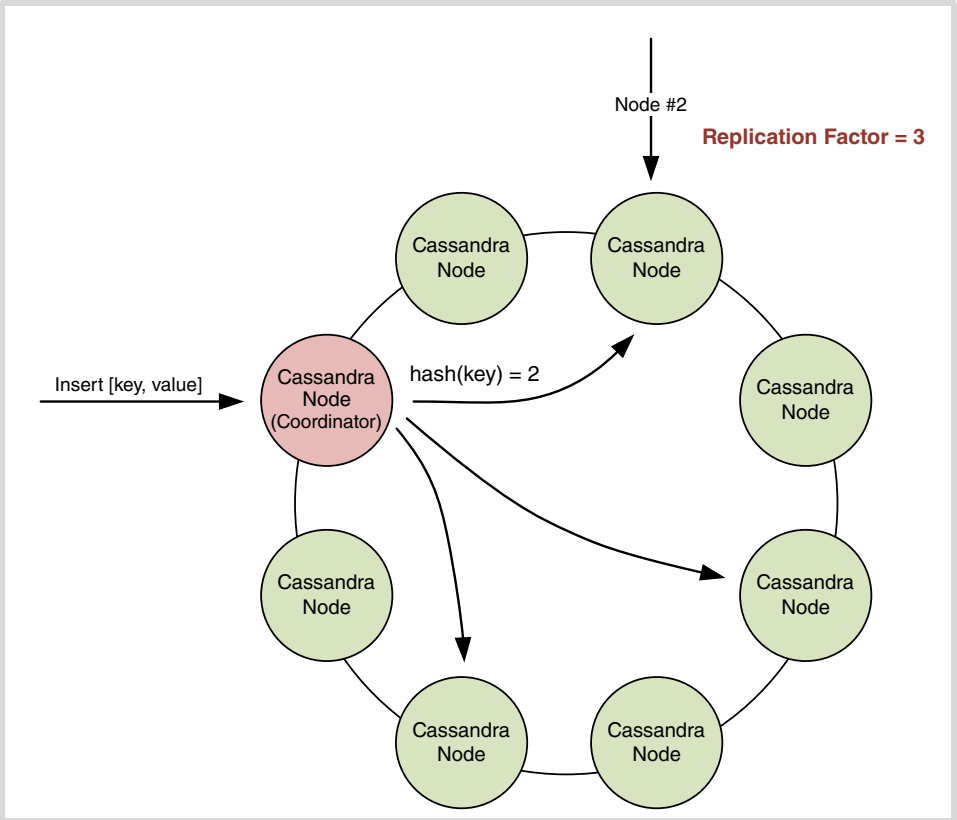


Figure 7

Cassandra is massively scalable, providing absurdly high write throughput. Figure 8 is a possibly biased diagram, showing writes per second benchmarks. [Datastax].

Even though this diagram might be biased and it looks like it compares Cassandra to databases set to provide strong consistency (otherwise I can't see why MongoDB would drop performance when upgraded from 4 to 8 nodes), this should still show what a properly set up Cassandra cluster is capable of.

Regardless, in the distributed systems trade-off which enables horizontal scaling and incredibly high throughput, Cassandra does not provide some fundamental features of ACID databases – namely, transactions.

Consensus

Database transactions are tricky to implement in distributed systems as they require each node to agree on the right action to take (abort or commit). This is known as **consensus** and it is a fundamental problem in distributed systems.

Reaching the type of agreement needed for the "transaction commit" problem is straightforward if the participating processes and the network are completely reliable. However, real systems are subject to a number of possible faults, such as process crashes, network partitioning, and lost, distorted, or duplicated messages.

This poses an issue – it has been proven impossible to guarantee that a correct consensus is reached within a bounded time frame on a non-reliable network.

In practice, though, there are algorithms that reach consensus on a non-reliable network pretty quickly. Cassandra actually provides lightweight transactions through the use of the Paxos algorithm for distributed consensus.

Distributed computing

Distributed computing is the key to the influx of Big Data processing we've seen in recent years. It is the technique of splitting an enormous task (e.g aggregate 100 billion records), of which no single computer is capable of practically executing on its own, into many smaller tasks, each of which can fit into a single commodity machine. You split your huge task into many smaller ones, have them execute on many machines in parallel, aggregate the data appropriately and you have solved your initial problem. This approach again enables you to scale horizontally – when you have a bigger task, simply include more nodes in the calculation.

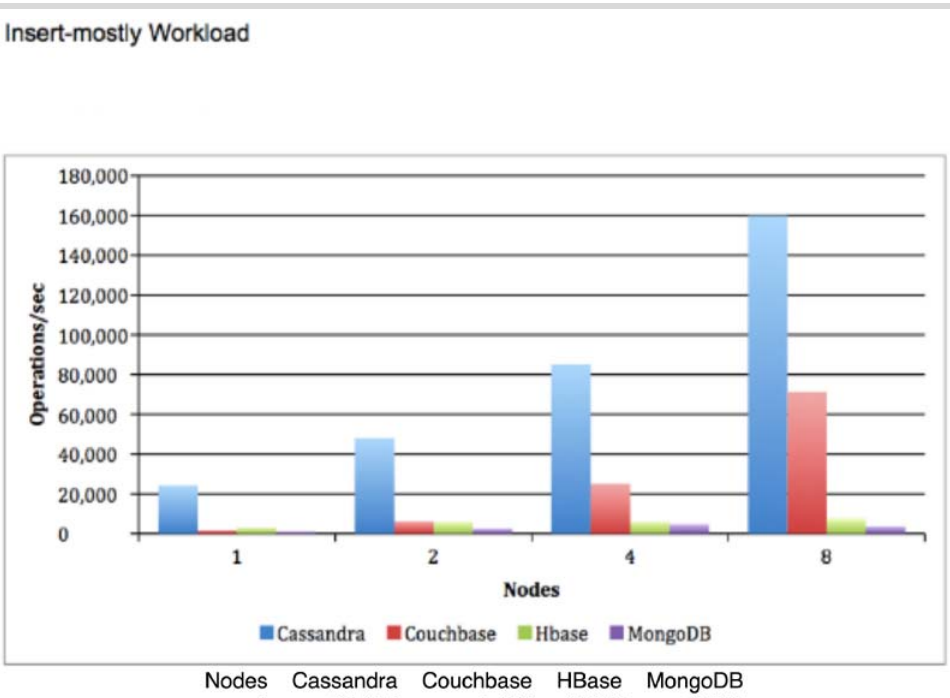
> Known Scale – Folding@Home had 160k active machines in 2012

An early innovator in this space was Google, which by necessity of their large amounts of data had to invent a new paradigm for distributed computation – MapReduce. They published a paper on it in 2004 and the open source community later created Apache Hadoop based on it.

MapReduce

MapReduce can be simply defined as two steps – mapping the data and reducing it to something meaningful.

Let's get at it with an example again (see Figure 9, overleaf).



Nodes	Cassandra	Couchbase	HBase	MongoDB
1	~24,000	~1,600	~2,900	~1,000
2	~48,000	~6,200	~5,800	~2,400
4	~85,300	~25,200	~5,900	~4,800
8	~160,000	~71,300	~7,600	~3,600

Figure 8

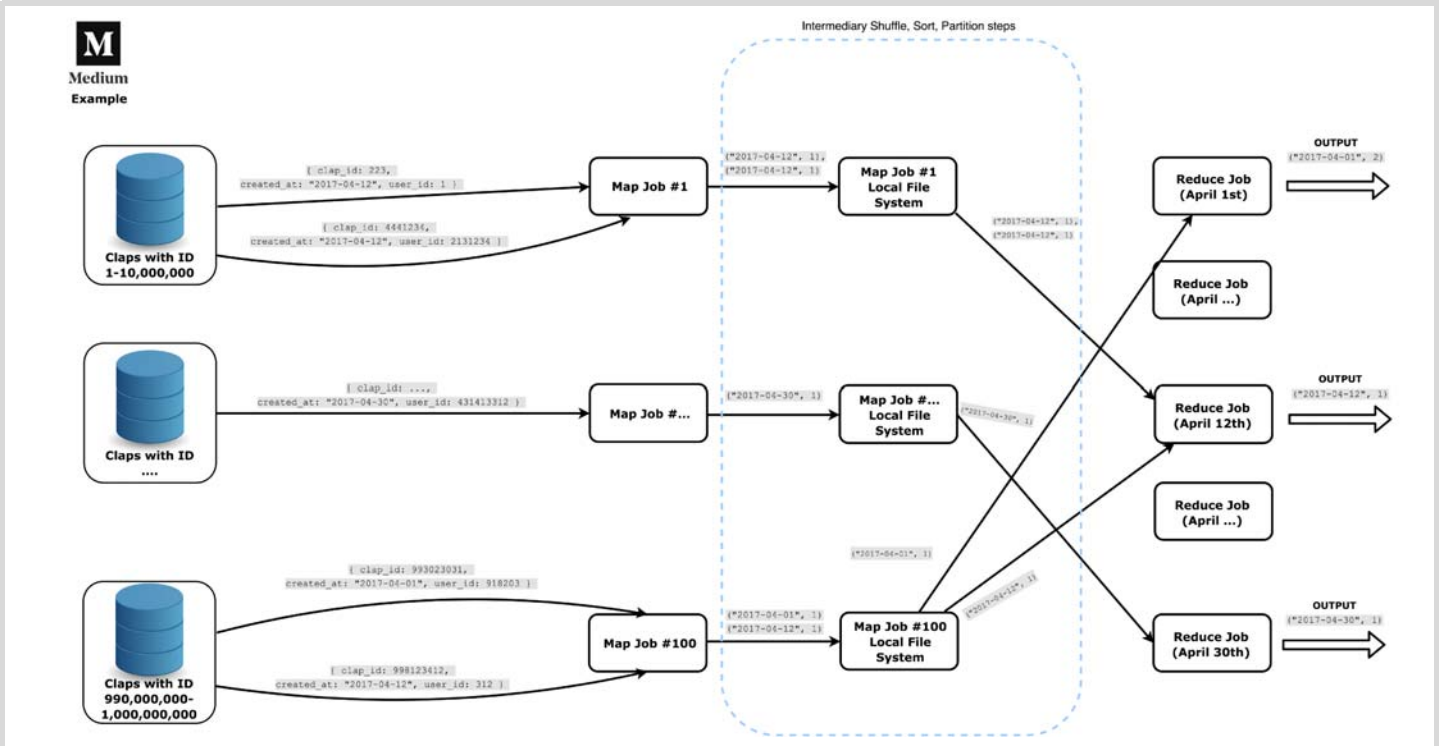


Figure 9

Say we are Medium and we stored our enormous information in a secondary distributed database for warehousing purposes. We want to fetch data representing the number of claps issued each day throughout April 2017 (a year ago).

This example is kept as short, clear and simple as possible, but imagine we are working with loads of data (e.g analyzing billions of claps). We won't be storing all of this information on one machine obviously and we won't be analyzing all of this with one machine only. We also won't be querying the production database but rather some "warehouse" database built specifically for low-priority offline jobs.

Each Map job is a separate node transforming as much data as it can. Each job traverses all of the data in the given storage node and maps it to a simple tuple of the date and the number one. Then, three intermediary steps (which nobody talks about) are done – Shuffle, Sort and Partition. They basically further arrange the data and delete it to the appropriate reduce job. As we're dealing with big data, we have each Reduce job separated to work on a single date only.

This is a good paradigm and surprisingly enables you to do a lot with it – you can chain multiple MapReduce jobs for example.

Better techniques

MapReduce is somewhat legacy nowadays and brings some problems with it. Because it works in batches (jobs) a problem arises where if your job fails – you need to restart the whole thing. A 2-hour job failing can really slow down your whole data processing pipeline and you do not want that in the very least, especially in peak hours.

Another issue is the time you wait until you receive results. In real-time analytic systems (which all have big data and thus use distributed computing) it is important to have your latest crunched data be as fresh as possible and certainly not from a few hours ago.

As such, other architectures have emerged that address these issues. Namely Lambda Architecture (mix of batch processing and stream processing) and Kappa Architecture (only stream processing). These advances in the field have brought new tools enabling them – Kafka Streams, Apache Spark, Apache Storm, Apache Samza.

Distributed file systems

Distributed file systems can be thought of as distributed data stores. They're the same thing as a concept – storing and accessing a large amount of data across a cluster of machines all appearing as one. They typically go hand in hand with Distributed Computing.

> Known Scale – Yahoo is known for running HDFS on over 42,000 nodes for storage of 600 Petabytes of data, way back in 2011

Wikipedia defines the difference being that distributed file systems allow files to be accessed using the same interfaces and semantics as local files, not through a custom API like the Cassandra Query Language (CQL).

HDFS

Hadoop Distributed File System (HDFS) is the distributed file system used for distributed computing via the Hadoop framework (see Figure 10, overleaf). Boasting widespread adoption, it is used to store and replicate large files (GB or TB in size) across many machines.

Its architecture consists mainly of NameNodes and DataNodes. NameNodes are responsible for keeping metadata about the cluster, like which node contains which file blocks. They act as coordinators for the network by figuring out where best to store and replicate files, tracking the system's health. DataNodes simply store files and execute commands like replicating a file, writing a new one and others.

Unsurprisingly, HDFS is best used with Hadoop for computation as it provides data awareness to the computation jobs. Said jobs then get ran on the nodes storing the data. This leverages data locality – optimizes computations and reduces the amount of traffic over the network.

IPFS

Interplanetary File System (IPFS) is an exciting new peer-to-peer protocol/network for a distributed file system. Leveraging Blockchain technology, it boasts a completely decentralized architecture with no single owner nor point of failure.

IPFS offers a naming system (similar to DNS) called IPNS and lets users easily access information. It stores file via historic versioning, similar to how Git does. This allows for accessing all of a file's previous states.

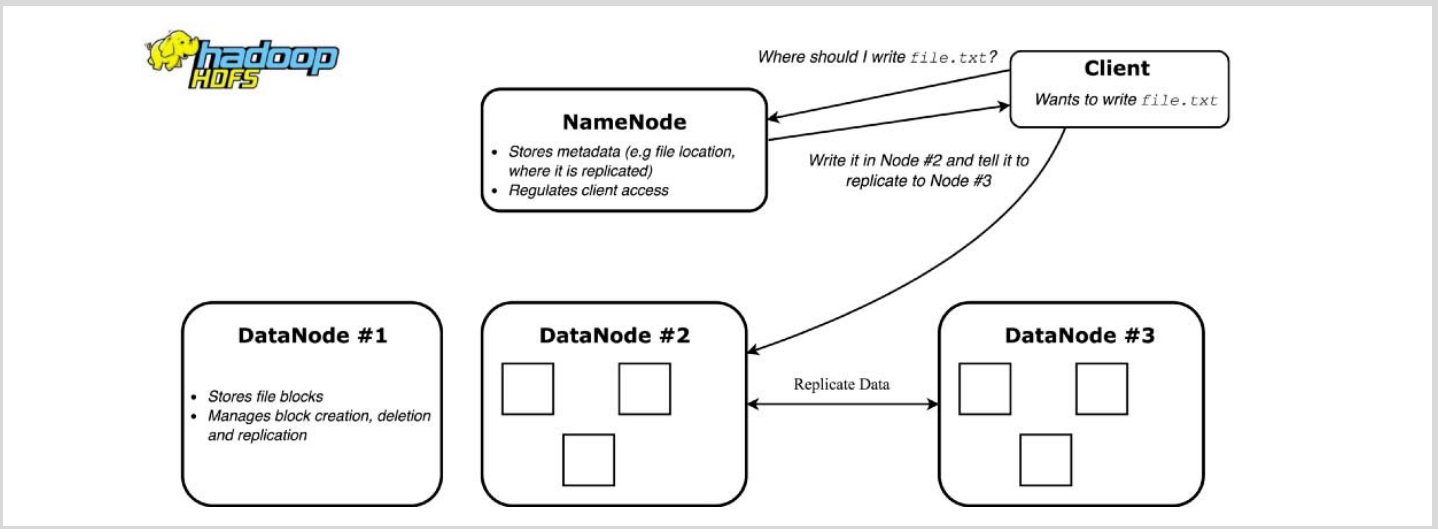


Figure 10

It is still undergoing heavy development (v0.4 as of time of writing) but has already seen projects interested in building over it (FileCoin).

Distributed messaging

Messaging systems (see Figure 11) provide a central place for storage and propagation of messages/events inside your overall system. They allow you to decouple your application logic from directly talking with your other systems.

> Known Scale – LinkedIn's Kafka cluster processed 1 trillion messages a day with peaks of 4.5 millions messages a second.

Simply put, a messaging platform works in the following way:

A message is broadcast from the application which potentially create it (called a *producer*), goes into the platform and is read by potentially multiple applications which are interested in it (called *consumers*).

If you need to save a certain event to a few places (e.g user creation to database, warehouse, email sending service and whatever else you can come up with) a messaging platform is the cleanest way to spread that message.

Consumers can either pull information out of the brokers (pull model) or have the brokers push information directly into the consumers (push model).

There are a couple of popular top-notch messaging platforms:

- **RabbitMQ** – Message broker which allows you finer-grained control of message trajectories via routing rules and other easily configurable settings. Can be called a smart broker, as it has a lot of logic in it and tightly keeps track of messages that pass through it. Provides settings for both AP and CP from CAP.

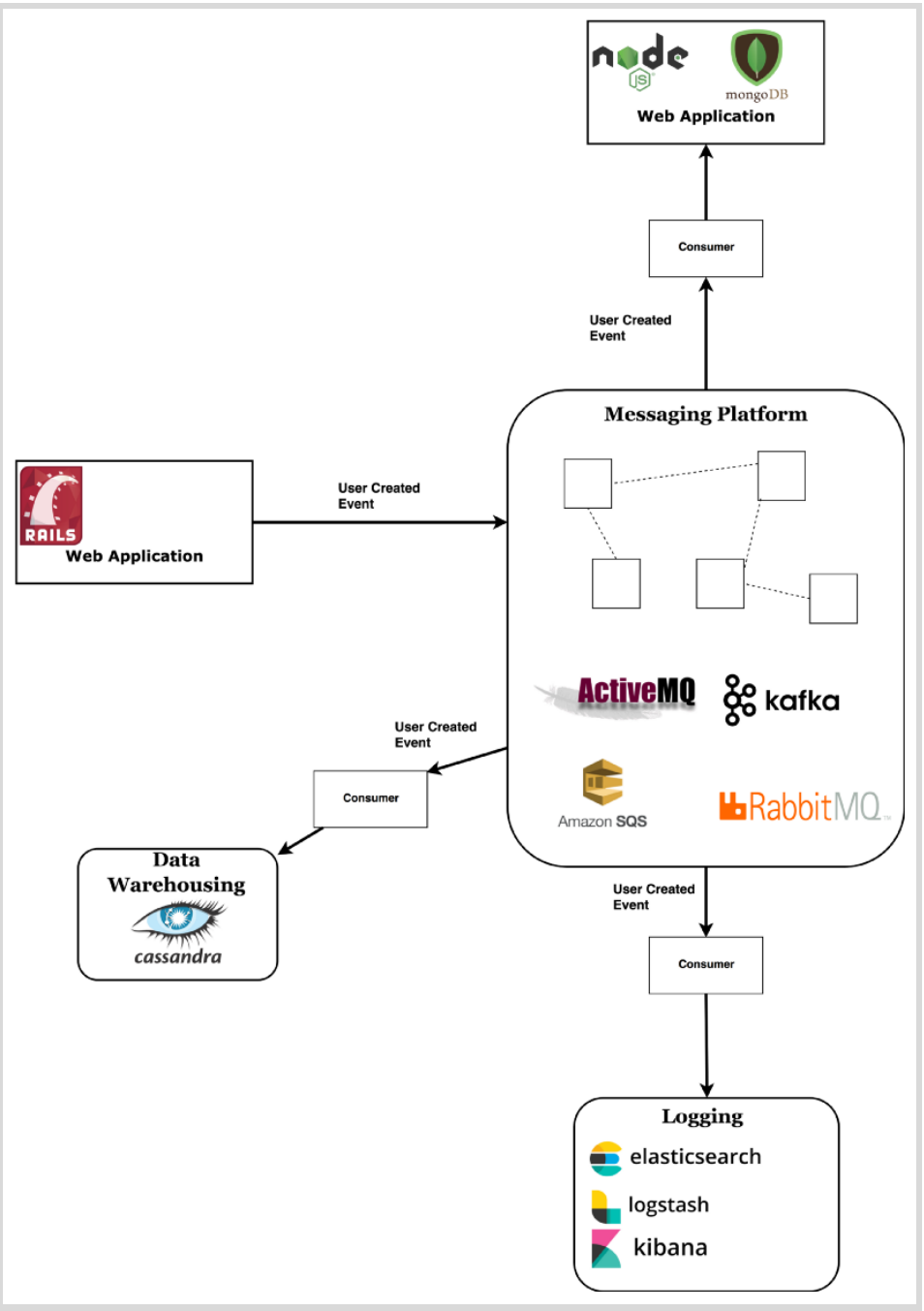


Figure 11

Uses a push model for notifying the consumers.

- **Kafka** – Message broker (and all out platform) which is a bit lower level, as in it does not keep track of which messages have been read and does not allow for complex routing logic. This helps it achieve amazing performance. In my opinion, this is the biggest prospect in this space with active development from the open-source community and support from the Confluent team. Kafka arguably has the most widespread use from top tech companies. I wrote a thorough introduction to this, where I go into detail about all of its goodness.
- **Apache ActiveMQ** – The oldest of the bunch, dating from 2004. Uses the JMS API, meaning it is geared towards Java EE applications. It got rewritten as ActiveMQ Artemis, which provides outstanding performance on par with Kafka.
- **Amazon SQS** – A messaging service provided by AWS. Lets you quickly integrate it with existing applications and eliminates the need to handle your own infrastructure, which might be a big benefit, as systems like Kafka are notoriously tricky to set up. Amazon also offers two similar services – SNS and MQ, the latter of which is basically ActiveMQ but managed by Amazon.

Distributed applications

If you roll up 5 Rails servers behind a single load balancer all connected to one database, could you call that a distributed application? Recall my definition from up above:

A distributed system is a group of computers working together as to appear as a single computer to the end-user. These machines have a shared state, operate concurrently and can fail independently without affecting the whole system's uptime.

If you count the database as a shared state, you could argue that this can be classified as a distributed system – but you'd be wrong, as you've missed the "working together" part of the definition.

A system is distributed only if the nodes communicate with each other to coordinate their actions.

Therefore something like an application running its back-end code on a peer-to-peer network can better be classified as a distributed application. Regardless, this is all needless classification that serves no purpose but illustrate how fussy we are about grouping things together.

Known Scale – BitTorrent swarm of 193,000 nodes for an episode of Game of Thrones, April, 2014

Erlang Virtual Machine

Erlang is a functional language that has great semantics for concurrency, distribution and fault-tolerance. The Erlang Virtual Machine itself handles the distribution of an Erlang application.

Its model works by having many *isolated* lightweight processes all with the ability to talk to each other via a built-in system of message passing. This is called the *Actor Model* and the Erlang OTP libraries can be thought of as a distributed actor framework (along the lines of Akka for the JVM).

The model is what helps it achieve great concurrency rather simply – the processes are spread across the available cores of the system running them. Since this is indistinguishable from a network setting (apart from the ability to drop messages), Erlang's VM can connect to other Erlang VMs running in the same data center or even in another continent. This swarm of virtual machines run one single application and handle machine failures via takeover (another node gets scheduled to run).

In fact, the distributed layer of the language was added in order to provide fault tolerance. Software running on a single machine is always at risk of having that single machine dying and taking your application offline. Software running on many nodes allows easier hardware failure handling, provided the application was built with that in mind.

BitTorrent

BitTorrent is one of the most widely used protocol for transferring large files across the web via torrents. The main idea is to facilitate file transfer between different peers in the network without having to go through a main server.

Using a BitTorrent client, you connect to multiple computers across the world to download a file. When you open a .torrent file, you connect to a so-called *tracker*, which is a machine that acts as a coordinator. It helps with peer discovery, showing you the nodes in the network which have the file you want. Figure 12 shows a sample network.

You have the notions of two types of user, a *leecher* and a *seeder*. A leecher is the user who is downloading a file and a seeder is the user who is uploading said file.

The funny thing about peer-to-peer networks is that you, as an ordinary user, have the ability to join and contribute to the network.

BitTorrent and its precursors (Gnutella, Napster) allow you to voluntarily host files and upload to other users who want them. The reason BitTorrent is so popular is that it was the first of its kind to provide incentives for contributing to the network. *Freeriding*, where a user would only download files, was an issue with the previous file sharing protocols.

BitTorrent solved freeriding to an extent by making seeders upload more to those who provide the best download rates. It works by incentivizing you to upload while downloading a file. Unfortunately, after you're done, nothing is making you stay active in the network. This causes a lack of seeders in the network who have the full file and as the protocol relies heavily on such users, solutions like private trackers came into fruition. Private trackers require you to be a member of a community (often invite-only) in order to participate in the distributed network.

After advancements in the field, trackerless torrents were invented. This was an upgrade to the BitTorrent protocol that did not rely on centralized

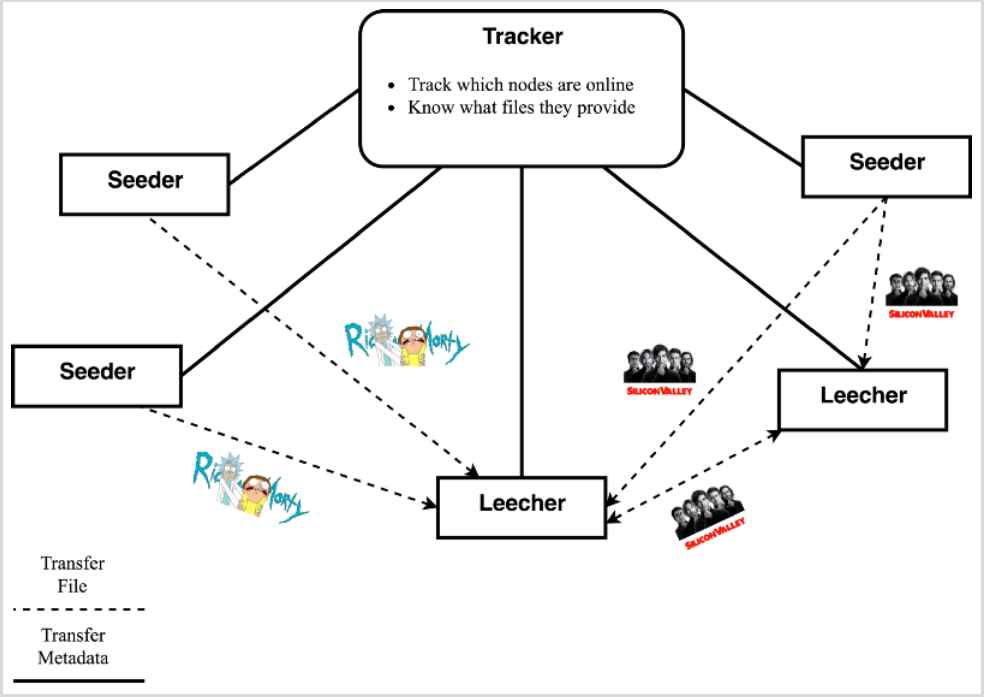


Figure 12

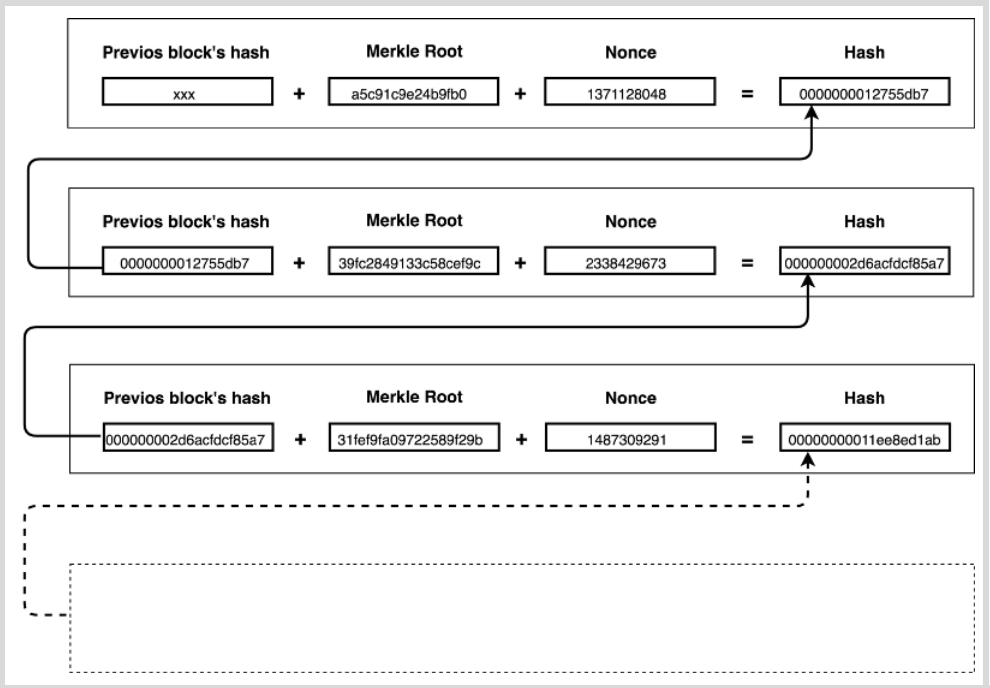


Figure 13

trackers for gathering metadata and finding peers but instead use new algorithms. One such instance is Kademlia (Mainline DHT), a distributed hash table (DHT) which allows you to find peers through other peers. In effect, each user performs a tracker's duties.

Distributed ledgers

A distributed ledger can be thought of as an immutable, append-only database that is replicated, synchronized and shared across all nodes in the distributed network.

> Known Scale – Ethereum Network had a peak of 1.3 million transactions a day on January 4th, 2018.

They leverage the Event Sourcing pattern, allowing you to rebuild the ledger's state at any time in its history.

Blockchain

Blockchain is the current underlying technology used for distributed ledgers and in fact marked their start. This latest and greatest innovation in the distributed space enabled the creation of the first ever truly distributed payment protocol – Bitcoin.

Blockchain is a distributed ledger carrying an ordered list of all transactions that ever occurred in its network. Transactions are grouped and stored in blocks. The whole blockchain is essentially a linked-list of blocks (hence the name). Said blocks are computationally expensive to create and are tightly linked to each other through cryptography.

Simply said, each block contains a special hash (that starts with X amount of zeroes) of the current block's contents (in the form of a Merkle Tree) plus the previous block's hash. This hash requires a lot of CPU power to be produced because the only way to come up with it is through brute-force. Figure 13 shows a simplified blockchain.

Miners are the nodes who try to compute the hash (via bruteforce). The miners all compete with each other for who can come up with a random string (called a nonce) which, when combine with the contents, produces the aforementioned hash. Once somebody finds the correct nonce – he broadcasts it to the whole network. Said string is then verified by each node on its own and accepted into their chain.

This translates into a system where it is absurdly costly to modify the blockchain and absurdly easy to verify that it is not tampered with.

It is costly to change a block's contents because that would produce a different hash. Remember that each subsequent block's hash is dependent on it. If you were to change a transaction in the first block of the picture above – you would change the Merkle Root. This would in turn change the block's hash (most likely without the needed leading zeroes) – that would change block #2's hash and so on and so on. This means you'd need to brute-force a new nonce for every block after the one you just modified.

The network always trusts and replicates the longest valid chain. In order to cheat the system and **eventually** produce a longer chain you'd need more than 50% of the total CPU power used by all the nodes.

Blockchain can be thought of as a distributed mechanism for *emergent consensus*. Consensus is not achieved explicitly – there is no election or fixed moment when consensus occurs. Instead, consensus is an *emergent* product of the asynchronous interaction of thousands of independent nodes, all following protocol rules.

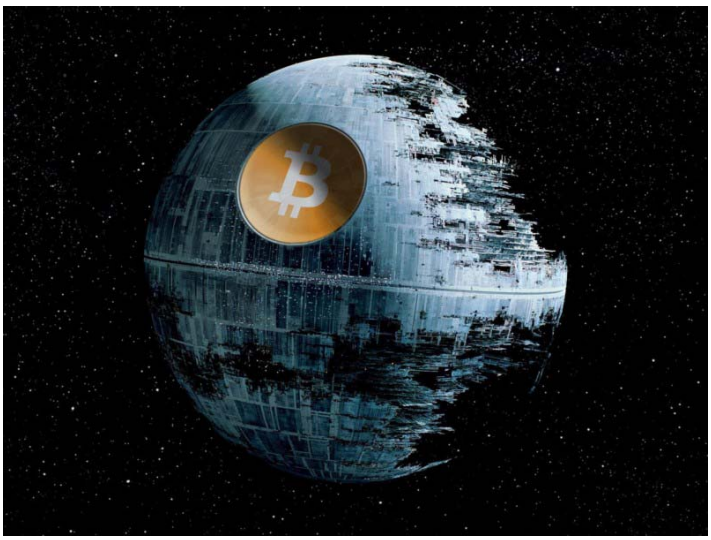
This unprecedented innovation has recently become a boom in the tech space with people predicting it will mark the creation of the Web 3.0. It is definitely the most exciting space in the software engineering world right now, filled with extremely challenging and interesting problems waiting to be solved.

Bitcoin

What previous distributed payment protocols lacked was a way to practically prevent the double-spending problem in real time, in a distributed manner. Research has produced interesting propositions ([Osipkov07] and [Szabo05]) but Bitcoin was the first to implement a practical solution with clear advantages over others.

The double spending problem states that an actor (e.g Bob) cannot spend his single resource in two places. If Bob has $1, he should not be able to give it to both Alice and Zack – it is only one asset, it cannot be duplicated. It turns out it is really hard to truly achieve this guarantee in a distributed system. There are some interesting mitigation approaches predating blockchain, but they do not completely solve the problem in a practical way.

Double-spending is solved easily by Bitcoin, as only one block is added to the chain at a time. Double-spending is impossible within a single block,



therefore even if two blocks are created at the same time – only one will come to be on the eventual longest chain.

Bitcoin relies on the difficulty of accumulating CPU power.

While in a voting system an attacker need only add nodes to the network (which is easy, as free access to the network is a design target), in a CPU power based scheme an attacker faces a physical limitation: getting access to more and more powerful hardware.

This is also the reason malicious groups of nodes need to control over 50% of the computational power of the network to actually carry any successful attack. Less than that, and the rest of the network will create a longer blockchain faster.

Ethereum

Ethereum can be thought of as a programmable blockchain-based software platform. It has its own cryptocurrency (Ether) which fuels the deployment of *smart contracts* on its blockchain.

Smart contracts are a piece of code stored as a single transaction in the Ethereum blockchain. To run the code, all you have to do is issue a transaction with a smart contract as its destination. This in turn makes the miner nodes execute the code and whatever changes it incurs. The code is executed inside the Ethereum Virtual Machine.

Solidity, Ethereum's native programming language, is what's used to write smart contracts. It is a turing-complete programming language which directly interfaces with the Ethereum blockchain, allowing you to query state like balances or other smart contract results. To prevent infinite loops, running the code requires some amount of Ether.

As the blockchain can be interpreted as a series of *state changes*, a lot of distributed applications (DApps) have been built on top of Ethereum and similar platforms.

Further usages of distributed ledgers

- **Proof of existence** – A service to anonymously and securely store proof that a certain digital document existed at some point of time. Useful for ensuring document integrity, ownership and timestamping.
- **Decentralized autonomous organizations (DAO)** – organizations which use blockchain as a means of reaching consensus on the organization's improvement propositions. Examples are Dash's governance system, the SmartCash project
- **Decentralized authentication** – Store your identity on the blockchain, enabling you to use single sign-on (SSO) everywhere. Sovrin, Civic

And many, many more. The distributed ledger technology really did open up endless possibilities. Some are most probably being invented as we speak!

Summary

In the short span of this article, we managed define what a distributed system is, why you'd use one and go over each category a little. Some important things to remember are:

- Distributed systems are complex
- They are chosen by necessity of scale and price
- They are harder to work with
- CAP Theorem – Consistency/Availability trade-off
- They have 6 categories – data stores, computing, file systems, messaging systems, ledgers, applications

To be frank, we have barely touched the surface on distributed systems. I did not have the chance to thoroughly tackle and explain core problems like consensus, replication strategies, event ordering & time, failure tolerance, broadcasting a message across the network and others.

Caution

Let me leave you with a parting forewarning:

> *Don't get addicted to the buzz that comes with solving hard problems. If you're solving the wrong problems, your effort will be wasted. If you miss a chance to turn a hard problem into an easy one, your effort will be wasted. Find inspiration in progress, not problem solving.*
>
> ~ @practicingdev

You must stray away from distributed systems as much as you can. The complexity overhead they incur with themselves is not worth the effort if you can avoid the problem by either solving it in a different way or some other out-of-the-box solution. ■

References

[Datastax] 'Apache Cassandra NoSQL Performance Benchmarks' at https://academy.datastax.com/planet-cassandra/nosql-performance-benchmarks

[Osipkov07] Ivan Osipkov, Eugene Vasserman, Nicholas Hopper and Yondae Kim (2007) 'Combating Double-Spending Using Cooperative P2P Systems' 25–27 June 2007, published in *27th International Conference on Distributed Computing Systems (ICDCS '07)*. A proposed solution in which each 'coin' can expire and is assigned a witness (validator) to it being spent. https://ieeexplore.ieee.org/document/4268195

[Szabo05] Nick Szabo (2005) 'Bit gold'Bitgold, published December 2005. A high-level overview of a protocol extremely similar to Bitcoin's. It is said this is the precursor to Bitcoin. http://web.archive.org/web/20060329122942/http://unenumerated.blogspot.com/2005/12/bit-gold.html

Further distributed systems reading

Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems by Martin Kleppmann, published by O'Reilly (January 2016). – A great book that goes over everything in distributed systems and more.

Cloud Computing Specialization, University of Illinois, Coursera – A long series of courses (6) going over distributed system concepts, applications. https://www.coursera.org/specializations/cloud-computing

Jepsen – Blog explaining a lot of distributed technologies (ElasticSearch, Redis, MongoDB, etc). https://aphyr.com/tags/Jepsen

This article was first published on freeCodeCamp on 27 April 2018 at https://medium.freecodecamp.org/a-thorough-introduction-to-distributed-systems-3b91562c9b3c

Don't Use std::endl

How do you add a new line in C++? Chris Sharpe suggests std::endl is a tiny utility that's more trouble than it's worth.

Just what am I complaining about?

I wanted to write about a particular bad habit I always see from students and C++ beginners, especially in questions on *Stack Overflow*, that seems to be taught in a lot of books, online tutorials, classes, etc.

Do you use `std::endl` to end lines when streaming text? Do you know what it does?

```
std::cout << "Foo" << std::endl;
```

`std::endl` does two things:

- writes `'\n'` to the stream
- flushes the stream [cppreference-1].

And nothing else. I've seen people mention that it is the right thing to use to get cross platform line endings. This is just wrong; streaming `std::endl` is guaranteed to do the same thing as streaming `'\n'`, and platforms make their own guarantees about expanding this into their canonical line endings (for instance, it becomes `<CR><LF>` on Windows). `std::endl` exists in the Standard Library only for those situations where you want to both write a newline character and flush the stream. I think, with the benefit of hindsight, it is wrong for such a utility to exist.

Why use std::endl?

In my mind, these are two entirely unrelated operations. The first is simply writing a character to the target underlying the stream, no more special than any other character. The second is an administrative action on the stream object itself, and one that I have rarely seen a good reason to carry out manually. Why would you want to do both at once? Some possibilities:

- You have some urgent output you want the user to see immediately. Sounds like a perfect case for `std::cerr`, which has `unitbuf` [cppreference-2] set so will display all output immediately and never needs to be manually flushed, entirely for this purpose.
- You want to display a prompt and make sure it appears before asking for input. `std::cout` and `std::cin` are `tie()`'d together [cppreference-3], so this will happen automatically. (Note they are also synced with the C equivalents [cppreference-4]).
- You want some sort of live updating output that is not urgent *per se*, and is not part of user interaction (interleaved with reading).

Well it may be the case that for a live updating UI (e.g. `top` [linux-1]), basic console output is not the best thing, and you should use a toolkit such as ncurses [linux-2]. But let's say you are just writing a basic example, like the pendulum simulator in Listing 1. What should you use in that case?

Chris Sharpe In between ~~herding cats~~ leading a team, juggling, and being a domestic servant to two huskies, Chris Sharpe occasionally finds time to write C++ at Bloomberg L.P. He can be reached at chris.sharpe.99@gmail.com

```
while (true) {
  std::cout << "Tick" << std::endl;
  sleep(1);
  std::cout << "Tock" << std::endl;
  sleep(1);
}
```

Listing 1

Why NOT use std::endl?

Ok, now I've painted myself into a corner where you might legitimately want to flush the stream each time. But the delimiter is still irrelevant to the flushing. What if you decided to separate the ticks with spaces or tabs instead? And anyway, many implementations of `std::cout` are line-buffered when writing to an actual terminal, for instance **libstdc++** (default with gcc) [GNU], but of course you can't 100% rely on that and remain completely cross-platform.

So I've argued that writing a newline and flushing a stream are unrelated operations, and you rarely want to do the latter anyway. But maybe you do occasionally want to do both at the same time. Isn't `std::endl` ideal for that?

I'd still argue no. It's very important to clearly express your intent in code. Comments are important of course, but it's even better when they simply aren't necessary. Comments can be out of date. The code can't. Now many beginners (and some more experienced programmers, sadly) simply don't know that `std::endl` flushes. So when I see it used, I simply have no idea if the original author really intended to flush or not. I see many uses of `std::endl` where flushing makes absolutely no sense whatsoever, and plenty of uses where it is certainly not clear that flushing is useful.

What should I do instead?

So what do I recommend? Use `'\n'`, and `std::flush` if you really do mean it. You may as well put the `'\n'` into the preceding string literal while you are at it.

```
std::cout << "foo\n";
std::cout << "Some int: " << i << '\n';
std::cout << "bar\n" << std::flush;
```

If your printing is a bit convoluted and you really do want to make it clear where you are printing a newline, you can separate it from the preceding string literal, and even give it a name if you like (Listing 2) or you can model it on more closely on `std::endl` (Listing 3).

```
namespace cds {
  char const nl = '\n';
}
// ...
std::cout << "Tick" << cds::nl << std::flush;
```

Listing 2

I've argued that writing a newline and flushing a stream are unrelated operations, and you rarely want to do the latter anyway

```
namespace cds {
  std::ostream& nl(std::ostream& os) {
    return os << '\n';
  }
}
// ...
std::cout << "Tick" << cds::nl << std::flush;
```

Listing 3

If you stream a function that takes and returns an **std::ostream&**, the function is called on the stream.

My argument is simply about writing expressive code – code that says what you mean and means what you say. If you don't find that convincing, many other people have also raised the genuine performance problem all the extra flushing can cause [Kuhl12], [Stroustrup] and [Turner16]. (The first of those links also provides another **nl** manipulator, that will work on streams with a character type other than **char**.) ■

References

[cppreference-1] endl: https://en.cppreference.com/w/cpp/io/manip/endl

[cppreference-2] unitbuf: http://en.cppreference.com/w/cpp/io/manip/unitbuf

[cppreference-3] tie: https://en.cppreference.com/w/cpp/io/basic_ios/tie

[cppreference-4] stdio: https://en.cppreference.com/w/cpp/io/ios_base/sync_with_stdio

[GNU] Buffering: https://gcc.gnu.org/onlinedocs/libstdc++/manual/streambufs.html#io.streambuf.buffering

[Kuhl12] Dietmar Kuhl 'stop excessive use of std::endl', 14 January 2012, https://kuhllib.com/2012/01/14/stop-excessive-use-of-stdendl/

[linux-1] top: https://linux.die.net/man/1/top

[linux-2] ncurses: https://linux.die.net/man/3/ncurses

[Stroustrup] Bjorn Stroustrup and Herb Sutter (editors), *C++ Core Guidelines*, https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#Rio-endl

[Turner16] Jason Turner (2016) 'C++ Weekly – Ep 7 Stop Using std::endl', at https://www.youtube.com/watch?v=GMqQOEZYVJQ

And the winners are...

In the last *Overload* (and *CVu*) we invited our readers to vote for their favourite articles of 2018 in *CVu*, which is our sibling magazine for members, and in *Overload*.

For CVu:

1st place: Chris Oldwood for 'To Mob, Pair or Fly Solo', in *CVu* 30.5 (November 2018)

2nd place: Frances Buontempo for 'Visualization of Multidimensional Data' in *CVu* 29.6 (January 2018)

For Overload:

1st place: Sergey Ignatchenko for '5 Reasons NOT to Use std::ostream for Human-Readable Output', in *Overload* 144 (April 2018) *and*

Andy Balaam for 'How to Write a Programming Language: Part 1, The Lexer', in *Overload* 145 (June 2018)

2nd place: Daniel James for 'Algol 68 – A Retrospective', in *Overload* 148 (December 2018)

Thank you to everyone who took time to vote, and for those who wrote. We can't offer a prize to these winners, just the mention here. A number of other writers got a vote – so be assured if you wrote for us someone probably thoroughly enjoyed what you had to say. Keep up the good work.

The article titles above link to the articles if you are reading this as a PDF. *Overload* articles are publicly available, but you must be a member (and logged in) to access the *CVu* ones. If you're not a member yet, why not join?

Afterwood

Good workers tidy up after themselves – it avoids accidents and makes them more productive. Chris Oldwood argues that good software developers should do the same.

It's interesting to watch other professions at work. Unless you work on the ground floor of a glass building or sit on a train with a monster-sized laptop, it's highly unlikely that Joe Public is ever going to be able to watch you writing software without being highly intrusive. Even if someone is visiting the office much of what happens is on a screen that is too small and inappropriately angled to see what's really going on; an impromptu huddle or conversation around a whiteboard is about the only glimpse they might get of us in action.

Sitting in the queue at the hairdressers – a traditionally open-plan workspace – you can watch the staff at work. While plenty of time is spent working directly with each client to make their hair look special, there are other activities which take place before, during and after which are not directly related to styling the client's hair, but nevertheless still contribute to the experience for both the client and stylist. For example, it's quite common for a member of staff to be wandering around with a broom constantly sweeping the floor. This task is not solely performed by some intern or graduate who's beginning to learn the ropes but may be done by senior staff too, if the need arises, rather than let the pile of discarded hair accumulate. Similarly, their core tools and products are readily accessible – you rarely see a hairdresser wandering around trying to find a pair of scissors or some styling wax.

I find the same can be true of building sites, whether it be large scale construction or a new extension to the house. Decent builders will constantly sweep up and clean around themselves whilst they are working, not just at the end of the day before knocking off. Likewise, people working in a kitchen can be seen wiping the sides, returning products to the store and washing up crockery and utensils.

While there is undoubtedly an element of 'health & safety' in all these professions that acts as a strong driver to try and keep the workspace clean and tidy, that can't be the only reason. The open plan nature of many hairdressing salons no doubt acts as another driver for them to keep the place presentable, as the window is literally a shop window showing passers-by what's on offer. Open kitchens are 'a thing' too, so that customers can sit and watch the chefs at work and be part of the entire cooking experience.

I find it hard to believe anybody truly enjoys working in a pig sty, except maybe some pig farmers, and yet those of us who work in offices (of the virtual kind where our 'files' and 'folders' exist merely as electrical phenomena and our 'desktop' is just a visual metaphor) often seem oblivious to the virtual detritus surrounding us. Unless you're working in an industry where lives are at stake, there is no risk to anyone's health from leaving around commented out code, ignored tests, unused source files, stale 3rd party packages, old build configurations, redundant source repositories, out of date documentation, uncommitted deployment hacks, build warnings, temporary log messages, etc.

While there are essentially no health risks, undue stress notwithstanding, there is surely an effect on productivity as people keep discovering the same remnants of days gone past and then wasting time on questioning what they're failing to comprehend until somebody fills in the blanks. It takes time before the muscle memory kicks in so that the same discovery again and again can quickly be discarded. Even if we deal with the problem the first time we encounter it, we are unlikely to have all the same context the original 'perpetrator' had and therefore it will take us longer to correct than simply doing it properly first time around.

While commented out code is trivial to deal with, dead code is often much harder to isolate even with modern refactoring tools unless it literally has no call sites and is not an obvious target for invocation by reflection either. Writing unit tests is great for live code but it hides dead code from our tools. Even when you don't have any tests keeping it alive, finding the roots and checking if *they* are still live can require a fair bit of legwork.

Ancillary code such as the tools and scripts used to stitch together build, test, deployment and support processes can often accumulate a large amount of cruft over time as their 'non-production' status frequently grants them only 3rd class citizenship. Their very nature can be highly environment-specific too, meaning that any change is going to be tricky to test and therefore *proving* redundancy could be non-trivial.

Not all changes start with a clear entry path that leads to an obvious test plan either. Often, we find something that needs changing and then go looking sideways to find similar points of interest, such as the same type of mistake made elsewhere or where the same code could have been copy-pasted rather than factored out. Once we find them, unless we are going to simply 'hit and hope', we have to navigate back out again to work out how we're going to test those changes too. Choosing not to join up the dots and blindly changing redundant code or configuration only adds to the appearance of liveliness.

Remember when you were a student or first shared a house with other people you didn't really know? Don't be *that* person who puts empty cereal boxes back in the cupboard, leaves a useless amount of milk in the carton and only washes up their own dishes. Please, for the sake of your ~~flat~~ team-mates, clean the sides and sweep the floor. ■

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it's enterprise grade technology in plush corporate offices. He also commentates on the Godmanchester duck race and can be easily distracted via gort@cix.co.uk or @chrisoldwood

JOIN THE ACCU!

You've read the magazine, now join the association dedicated to improving your coding skills.

The ACCU is a worldwide non-profit organisation run by programmers for programmers.

With full ACCU membership you get:

- 6 copies of *C Vu* a year
- 6 copies of *Overload* a year
- The ACCU handbook
- Reduced rates at our acclaimed annual developers' conference
- Access to back issues of ACCU periodicals via our web site
- Access to the *mentored developers projects*: a chance for developers at all levels to improve their skills
- Mailing lists ranging from general developer discussion, through programming language use, to job posting information
- The chance to participate: write articles, comment on what you read, ask questions, and learn from your peers.

Basic membership entitles you to the above benefits, but without Overload.

Corporate members receive five copies of each journal, and reduced conference rates for all employees.

How to join

You can join the ACCU using our online registration form. Go to **www.accu.org** and follow the instructions there.

Also available

You can now also purchase exclusive ACCU T-shirts and polo shirts. See the web site for details.

Design: Pete Goodliffe

PERSONAL MEMBERSHIP
CORPORATE MEMBERSHIP
STUDENT MEMBERSHIP

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

intel Software

CODE MAXIMIZED

from £510

#HighPerformance

Develop high performance parallel applications from enterprise to cloud, and HPC to AI using Intel® Parallel Studio XE. Deliver fast, scalable and reliable, parallel code.

QBS Software Ltd is an award-winning software reseller and Intel Elite Partner

To find out more about Intel products please contact us:
020 8733 7101 | sales@qbs.co.uk | www.qbssoftware.com/parallelstudio

qbs SOFTWARE