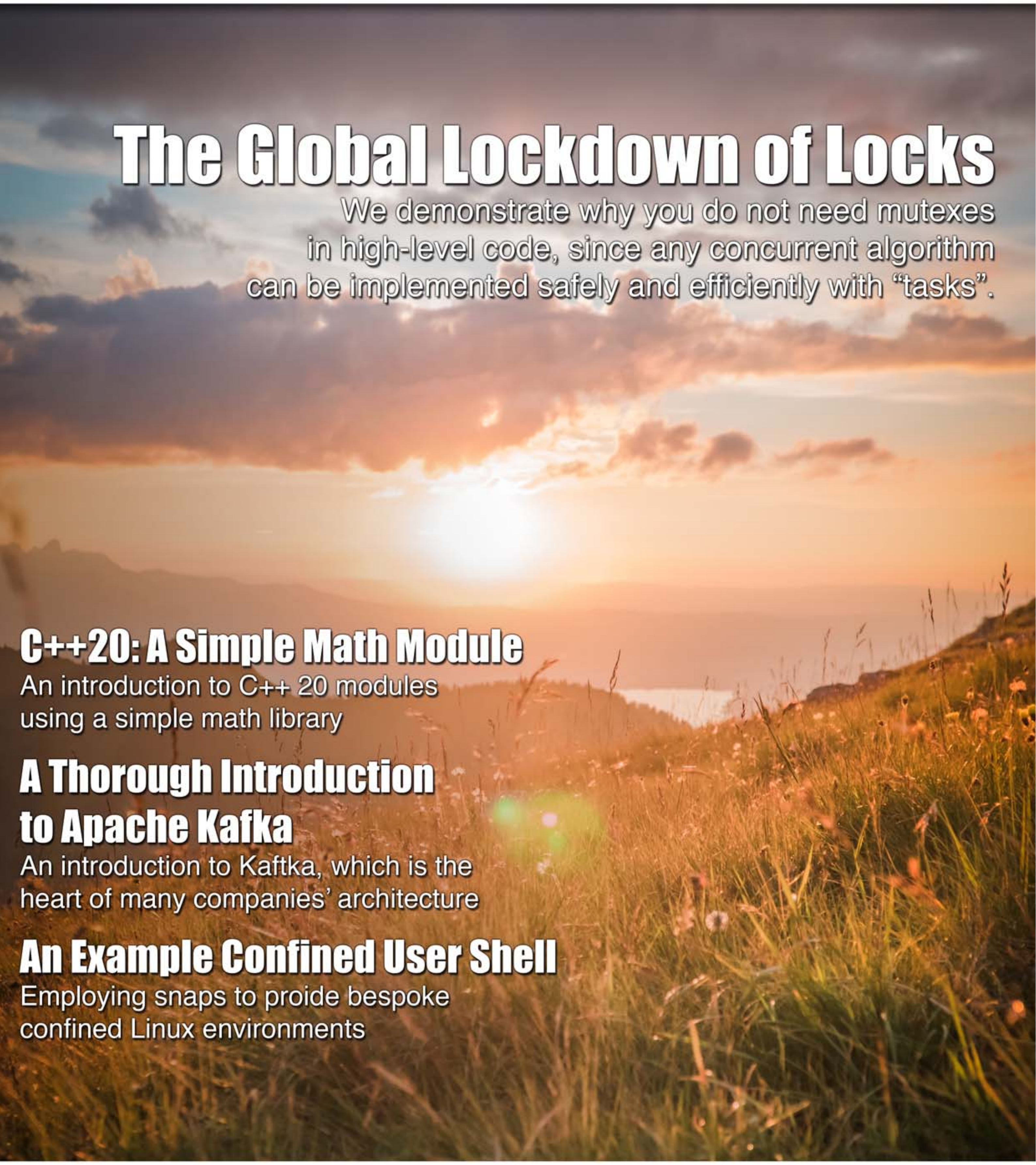
overload 158

AUGUST 2020 £4.50



The Global Lockdown of Locks

We demonstrate why you do not need mutexes in high-level code, since any concurrent algorithm can be implemented safely and efficiently with "tasks".

C++20: A Simple Math Module

An introduction to C++ 20 modules using a simple math library

A Thorough Introduction to Apache Kafka

An introduction to Kaftka, which is the heart of many companies' architecture

An Example Confined User Shell

Employing snaps to proide bespoke confined Linux environments

A magazine of ACCU

ISSN: 1354-3172

JET
BRAINS

A Power Language Needs Power Tools

Smart editor with full language support
Support for C++03/C++11, Boost and libc++, C++ templates and macros.

Reliable refactorings
Rename, Extract Function / Constant / Variable, Change Signature, & more

Code generation and navigation
Generate menu, Find context usages, Go to Symbol, and more

Profound code analysis
On-the-fly analysis with Quick-fixes & dozens of smart checks

GET A C++ DEVELOPMENT TOOL THAT YOU DESERVE

ReSharper C++
Visual Studio Extension for C++ developers

AppCode
IDE for iOS and OS X development

CLion
Cross-platform IDE for C and C++ developers

Start a free 30-day trial
jb.gg/cpp-accu

Find out more at **www.qbssoftware.com**

QBS
SOFTWARE

OVERLOAD 158

August 2020

ISSN 1354-3172

Editor

Frances Buontempo
overload@accu.org

Advisors

Ben Curry
b.d.curry@gmail.com

Mikael Kilpeläinen
mikael.kilpelainen@kolumbus.fi

Steve Love
steve@arventech.com

Chris Oldwood
gort@cix.co.uk

Roger Orr
rogero@howzatt.co.uk

Balog Pal
pasa@lib.hu

Tor Arve Stangeland
tor.arve.stangeland@gmail.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Printing and distribution

Parchment (Oxford) Ltd

Cover art and design

Pete Goodliffe
pete@goodliffe.net

Copy deadlines

All articles intended for publication in Overload 159 should be submitted by 1st September 2020 and those for Overload 160 by 1st November 2020.

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.

RE:Purpose FWD:Thinking

The pandemic could be a 'Once in a Lifetime' opportunity to stop and think. Frances Buontempo misses this chance and muses on lyrics by Talking Heads instead.

Though I should write an editorial, I might take a break this time, and think about the purpose of an *Overload* editorial instead. As I muse, I am looking out of the window watching rain dripping off trees, shoots and leaves. Without a strong sense of purpose, a compelling reason behind a chore, distractions creep in. Not that an editorial is a chore – I've never written one, so wouldn't know. Even with a sense of purpose, fear can creep in if you suspect your opinion may be in the minority, or controversial. I could write about diversity and inclusion. I could call into question excuses as to why there are fewer women in programming, and science or mathematics, in some parts of the world. I could discuss why some people claim women don't enjoy mathematics or their brains aren't wired up properly for it, or I could consider racial prejudice, and its preconceptions about innate 'racial' ability. Such biases tend to become self-fulfilling prophecies and even feedback loops. If most people in prison are black, police are more likely to stop and search black people, so in all probability more black people end up in prison, reinforcing the conception that 'they are criminals'. I've been reading geneticist Adam Rutherford's *How to argue with a racist* [Rutherford20], which shows why the idea of race is ill-formed to begin with. Highly recommended.

One very striking point made by Rutherford is that race, either in terms of genetics or other measures, is based on clustering. This is a statistical technique, often also used in machine learning. Data points, say people, are described by a tuple of features, say genes, skin colour, IQ, whatever. The clustering algorithm is then run, and reports back who belongs in which cluster. There are various ways to assign tuples to different groups, though all rely on a distance measure and almost all need to know how many groups you want. So, how many races are there? Two. Great you'll get the tuples of people split between two groups. Five. Ditto. What is the point of clustering data? Well, sometimes you get a clear 'decision boundary', helping you disambiguate between labels. If I plot the height of a random collection of mice and elephants, against specimen number, I can draw a straight line between the two groups. Above the line means elephant; below means mouse. If I asked for three clusters, I'd get three. The data would be labelled into one of three groups, but that wouldn't help spot elephants, in the room or otherwise. Why try clustering? To fish around and see if you do have some 'different' potential labels going on. If you do, you can then ask which features correlate best with the labels. Even if you discover something via clustering, you still need to think about what you have found out. If you discover AI can work out someone's 'criminality' by their skin colour, you may have discovered systemic racism. Just saying. Statistics often forms a null hypothesis; nothing to see here, and an alternate hypothesis; that's significant. The purpose of subsequence analysis is to decide which hypothesis is more likely correct. Some machine learning is applied without an up-front purpose. That's ok. A data pipeline producing visualisations for people to consider can be useful or interesting. Data science often does this – it's an initial step, but hasn't fully covered the 'science' part to my mind. It can be the start of some science though.

That's enough of controversial subjects. As I was saying, without a sense of purpose, random distractions creep in. If you've ever worked on a legacy code base, you may have found yourself deep in a call stack wondering how you got there. '*And you may ask yourself, "How do I work this?"*' [Talking Heads]. Furthermore,

> And you may ask yourself, 'What is that beautiful house?'
> And you may ask yourself, 'Where does that highway go to?'
> And you may ask yourself, 'Am I right? Am I wrong?'
> And you may say to yourself, 'My God! What have I done?'

Actually, listening to music can help you concentrate and not get too distracted by the highways in the code. If you get to the end of a playlist or album and haven't got anywhere, it's time to get up and walk, grab a drink, do something else for a bit to get your focus back. Legacy code can be hard work. If you understand what part of it is trying to achieve, for example if you have some kind of code test round a bit of it, you can experiment with the code and try to refactor to something clearer. Without tests, you're probably trying to dig yourself out of a hole though:

> Water dissolving and water removing
> There is water at the bottom of the ocean
> Under the water, carry the water
> Remove the water at the bottom of the ocean

If you're trying to bail out the ocean, you need to give up and learn to swim. Don't drown in a confusing code base. There are many resources to help with such code, for example *Working Effectively with Legacy code* by Mike Feathers, or *The Legacy Programmer's Toolbox* By Jonathan Boccara. Make sure you are using version control, then you won't be afraid to try changes to see what happens. Don't trust the comments, though they may show the original purpose of some code. Think about what you need to achieve, and don't get lost in a maze of twisty passages, all alike. Use a heuristic – always go left, always delete the comments, never change code without a test. Don't get put off – keep your eyes on the prize. Above all, don't drown.

Our garden looks slightly drowned at the moment, which is a distraction. Several of the bushes look a bit overgrown, and a few weeds are trying to take over. Looking at it, I can see how it possibly was, a while ago. I can sense what sort of shape might be trying to happen – where to prune, and

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. 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.

cut back. However, I'll get soaked if I go out there so these refactors will have to wait. If you look at legacy code you can sometimes see an older, simpler code base in the undergrowth and behind the weeds. I often see branches spring up to bolt-on new behaviour. Rather than grafting new growth on to established root-stocks, a cutting in a pot has been balanced in a tree, fallen over, smashed and turned into some weird looking triffid. OK, perhaps the analogy isn't perfect, but you get the idea. Why does this happen? Because the simplest ways to shoe-horn new features into code is often by slapping an `if`/`else` in. If code has been designed with potential future variations in mind, there may be a place to use a new strategy, send in a lambda, or similar. The Open/Closed principle (OCP) might be driving at this idea. Jon Skeet blogged about the OCP a while ago [Skeet13]. He questions what 'open' and 'closed' mean here, and quotes Wikipedia's summary of Bertrand Meyer's version:

> The idea was that once completed, the implementation of a class could only be modified to correct errors; new or changed features would require that a different class be created. That class could reuse coding from the original class through inheritance. The derived subclass might or might not have the same interface as the original class.

Kinda like repurposing the original, via inheritance, or as Jon puts it, "*A ghastly abuse of inheritance*". He is having more of a dig at Uncle Bob's annotation than Meyer's original quote though. Jon talks about 'Protected variation' as a clearer idea:

> Identify points of predicted variation and create a stable interface around them.

In order to achieve this, some forward thinking is required. On the face of it, this seems to be in conflict with the YAGNI principle – 'You aren't going to need it' – a mantra from Extreme Programming. The words, 'on the face of it' are important. There is a difference between building something now, that you might need in the future, versus architecting your code so that you can add new features in the future. Otherwise the twisty maze of `if`s/`else`s will happen. Martin Fowler talks about YAGNI in a blog [Fowler15]. He asks devs to consider what refactoring would be needed to introduce a new feature later on, and as a side effect, to:

> …add something that's easy to do now, adds minimal complexity, yet significantly reduces the later cost. Using lookup tables for error messages rather than inline literals are an example that are simple yet make later translations easier to support.

A little bit of forward thinking and a sense of purpose can make your life easier.

One advantage the pandemic has brought is many local Meetups now take place online. You can therefore join people who are miles away, and virtually attend talks you would otherwise have not been able to get to. A case in point, for me, being the Norfolk developers [Nor(Dev)]. On the 2nd July, Jez Higgins gave a talk entitled, 'Journey into space' [Higgins20]. He started by reminiscing about computers from the 1980s and how he learnt to code. He talked about good code, agile, and how he '*read an article by one of the original signatories of the manifesto for agile software development, and accidentally ended up writing a version of Asteroids for [his] phone.*' The talk was excellent and is available on YouTube. By talking about previous work on a long project, which had been designed up front, Jez reminded us to ask why the agile manifesto had been written. I suspect many scrum masters and agile managers haven't worked on an old-school Waterfall project, so don't fully appreciate the context on this. I've heard people talk about being 'rigidly agile', or insisting you need specified meetings for prescribed lengths of time. Four-hour back-log grooming meetings etc. The first of the four better ways of developing software stated in the manifesto says, '*Individuals and interactions over processes and tools.*' [Agile]. Ron Jeffries, the signatory to whom Jez referred, blogged about 'Dark Scrum' [Jeffries16]. He talks about how Scrum can end up being a tool of oppression. If you understand why you're using scrum, and how to keep communication open things will improve. If the goal is working software, then being able to show new features, no matter how small/lean, that do something useful is a win for everyone. Getting to that point can be hard work though.

Jefferies talks about testing quite a bit in the blog. Who doesn't? Well, I know I do, and it annoys some people. He also talks about continuous integration. I notice build/test/deploy pipelines referred to frequently, for example on Jenkins' 'Continuous Delivery' articles [Jenkins]. Recently, the UK prime minister announced, "*We're going to build, build, build, and deploy jobs, jobs, jobs.*" To build and deploy without testing is asking for trouble. When we test, we check what effect our construction has, and ask if it does what we expected. This inspired me to write a dreadful little ditty:

> We're going to build, build, build
> And deliver jobs, jobs, jobs
> sudo crontab -e e e
> Gee
> Didn't work.
> Didn't test first. What a berk.

Apologies. I'll stick to the day job. So, what's the purpose of an *Overload* editorial? I'll go have a think about that for next time, while you read this issues' articles. Do feel free to contact our writers and encourage them. It gives a sense of purpose, and might spark new ideas to think about, and hopefully more articles in the future.

References

[Agile] Manifesto for Agile Software Development at: https://agilemanifesto.org/

[Fowler15] Martin Fowler (2015) 'Yagni' posted 26 May 2015 at: https://www.martinfowler.com/bliki/Yagni.html

[Higgins20] Jez Higgins (2020) 'Journey Into Space', talk on 2 July 2020. Abstract: https://www.meetup.com/Norfolk-Developers-NorDev/events/271181133/
Recording: https://www.youtube.com/watch?v=8BOnppFZo6s

[Jeffries16] Ron Jeffries (2016) 'Dark Scrum', posted 8 September 2015 at https://ronjeffries.com/articles/016-09ff/defense/

[Jenkins] 'Continuous Deliver Articles' at https://www.jenkins.io/solutions/pipeline/

[Nor(DEV)] Norfolk Developers meetups: https://www.meetup.com/Norfolk-Developers-NorDev/

[Rutherford20] Adam Rutherford (2020) *How to argue with a racist* Orion Publishing Co, Feb 2020, ISBN 978-1474611244

[Skeet13] Jon Skeet (2013) 'The Open–Closed Principle, in Review' on Jon Skeet's coding blog, posted 15 March 2013 at: https://codeblog.jonskeet.uk/2013/03/15/the-open-closed-principle-in-review/

[Talking Heads] Talking Heads: Lyrics from *Once in a lifetime*

The Global Lockdown of Locks

Locks can be replaced with tasks.
Lucian Radu Teodorescu shows us how.

In the previous article [Teodorescu20], we showed how one can achieve great speedups using tasks as the foundation for concurrent programming instead of locks. The one thing that is not clear is whether one can use tasks for all the concurrent algorithms[1]. Tasks may be applicable to certain types of problem, but not to all problems. This article tries to complete the picture by showing that all algorithms can use tasks and there is no need for locks.

We prove that one can find a general algorithm for implementing all concurrent algorithms just by using a task system, without the need for locks or other synchronisation primitives in user code.

Hopefully, by the end of the article, the reader will be convinced that locks should not be used in day-to-day problems but instead should be confined to use by experts when building concurrent systems.

Task systems and task graphs

Let us recap from the previous article a few notions about what it means to have a task system:

- The work of the algorithm is represented by tasks; a task is an **independent unit of work**.
- We add constraints between tasks, modelling dependencies and possible race conditions.
- There is an execution engine that can take tasks and execute them dynamically on the various cores available on the running machine. The execution engine is typically greedy, meaning that it executes tasks as soon as it has hardware resources.

One key point here is that the tasks are independent. Two *active* tasks should never block each other. (We'll discuss the meaning of active tasks below.) As shown below, all the constraints between tasks can be implemented on top of such a task system. The task system framework may provide facilities to ease this work, but this is not essential.

As examples of task systems, we have Concore [concore] or the more popular Intel Threading Building Blocks [tbb]. The few examples in this article will be written using Concore, as it is the library I'm developing while trying to make sense of the concurrency world.

It is obvious that all algorithms can be expressed in terms of non-blocking tasks, possibly with some threads that don't have tasks to be executed (or colloquially waiting for them to be unblocked). At the bare minimum, one can wrap any non-blocking instruction in a task (very inefficient, but possible); however, in general, a task will be a set of instructions that make sense to be considered as one unit, both from the semantic point of view and from the constraints point of view.

Lucian Radu Teodorescu has a PhD in programming languages and is a Software Architect at Garmin. As hobbies, he is working on his own programming language and he is improving his Chuck Norris debugging skills: staring at the code until all the bugs flee in horror. You can contact him at lucteo@lucteo.ro

Tasks may have *dependencies* or *restrictions* between them. If two tasks A and B can be safely run in parallel, then there is no dependency or restriction between the two of them. If task A needs to finish before executing B we say that there is a dependency from A to B; we denote that by $A \rightarrow B$. If the two tasks cannot be safely run in parallel, but the order in which we run them is irrelevant, then we say that there is a restriction between them; we denote that by $A \sim B$. A restriction between two tasks can always be turned into dependency by arbitrarily choosing which task needs to be executed first. For example, if two tasks write the same data, we typically want to create a restriction between them; not doing so will probably result in race condition bugs.

For efficiency reasons, we consider the tasks to be as small as possible, with respect to the constraints between them. That is, if only a small part of task A has constraints with other tasks, then we would break up task A into smaller parts, to minimize the scope of the constraint.

The dependency is transitive, but the restriction relation is not. If we have three tasks A, B and C, with $A \rightarrow B$ and $B \rightarrow C$, then $A \rightarrow C$; for a given task we can always find the transitive closure for the dependency relation. On the other hand, if $A \sim B$ and $B \sim C$, it doesn't imply that $A \sim C$: tasks A and C could be run in parallel. Also, it's important to note that the restriction relation is commutative (while the dependency is not).

Let us denote by $succ(A)$ the set of tasks given by the transitive closure of the dependency relation. Also, let us denote by $pred(A)$ the set of tasks X for which $A \in succ(X)$ (i.e., all the predecessors of task A).

With these notions set, a task A can be executed in isolation if $pred(A)=\emptyset$. Also, a set of tasks S can be executed in parallel, if for each task X in that set $pred(X)=\emptyset$, and $\nexists X,Y \in S$ such as $X \sim Y$; we call S to be a *parallel set of tasks*. These are the conditions that we need for executing the tasks.

The reader should pause a bit and digest the information above. Basically, we have expressed in algebraic form the safety conditions for executing tasks. We defined the constraints in such a way to eliminate possible data race bugs.

A more intuitive way is to represent all these with graphs of tasks. Figure 1 (overleaf) shows an example; most of the links between tasks are dependencies, but we also have restrictions $T_9 \sim T_{13}$ and $T_{14} \sim T_{15}$.

Representing the concurrent algorithm as a graph is highly encouraged. Software developers tend to reason better on the algorithm if it is expressed visually as a graph than if it's expressed as a set of rules. Knowledge of graphs is deeply rooted in the software engineering mindset. This can help both at algorithm design and at understanding the performance consequences of the algorithm.

For example, one can easily check what are the tasks that can be run in parallel. In our case, task T_6, T_7, T_8, T_9, T_{10}, T_{11} and T_{12} can safely be run in parallel, and so are the set of tasks T_2, T_8, T_{14}, T_{11} and T_{16}.

1. As in the previous article, we use the term *algorithm* to denote a general problem that we need to solve; should not be mistaken for well-known algorithms in computer science.

a task will be a set of instructions that make sense to be considered as one unit, both from the semantic point of view and from the constraints point of view

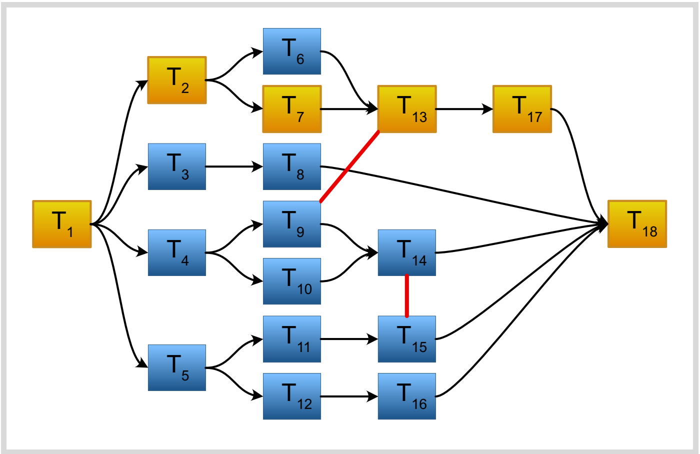


Figure 1

A general method for executing algorithms concurrently

We need to distinguish some time-points relevant to the algorithm execution:

- The point at which a task can be created (we have all the data needed; we know about its constraints with existing tasks) – this is typically different from the time the task can start executing
- The point at which the task can start executing
- The point at which the task finishes executing

We will construct a general method that will have logic at the creation of the task and at the point at which the task finishes executing to determine the point of execution start for all the tasks. We are building a scheduling algorithm.

To do so, we need to divide the tasks into three groups: *finished*, *active* and *pending* tasks. Finished tasks are, as the name implies, tasks that were completely executed. Active tasks are either tasks that are in progress or tasks that can be started at any given point. Because we have a limited number of cores, we cannot guarantee that all the active tasks start immediately; for example, on an 8-core machine, we will start executing 8 tasks, even if there are 1000 active tasks. Finally, pending tasks are tasks that cannot be executed right away, as some constraints are not satisfied; i.e., not all the predecessors are finished, or there is a restriction task with some task that is currently active.

The key point of this division is that all the active tasks might be executed in parallel given enough hardware resources. That is, all the predecessor tasks for them are completed, and there is no restriction relation between any two active tasks. That is, the set of active tasks at any given time is a parallel set of tasks as described above.

A task can be in any of the three groups at different times. The valid transitions are *pending* to *active*, and *active* to *finished*. No other transition is allowed.

It is obvious that we transition an *active* task to *finished* after the task execution is complete. Maybe less obvious, but still straightforward is the transition between *pending* and *active*; we make this transition as soon as possible, as soon as the constraints of the tasks disappear. The only caveat here is that we need to have a holistic approach. It may be the case that two tasks *A* and *B* can be both independently transitioned from *pending* to *active*, but not together at the same time, as $A \sim B$. In this case, we can choose one task and add it to the *active* group, while keeping the other to the *pending* group. In other words, transitioning from *pending* to *active* needs to be done one task at a time.

With this, we can finally describe the general method. It amounts to 4 steps:

1. Ensure we have at least one active task at the start of the algorithm
2. Keep track of the state of all the tasks and constraints between them
3. Add some special logic when a task needs to be created and when a task finishes executing; this is presented in pseudo-code in Listing 1.
4. Whenever a task becomes active, add it to the task system so that it can execute it.

At the end of the **executeTask()** function, we decide which of the *pending* tasks can be moved to *active*. For simplicity, we consider here that we will iterate over all the *pending* tasks. But that is overkill; in practice, we should look only at the tasks that are either successor of the current task or that have a restriction relation with it.

The key to this algorithm is the decision of whether the task can be active or not. It needs to be done whenever a task can be created, and at the end of running the tasks, when *pending* tasks can be made *active*. This is a greedy algorithm as it tries to move to *active* as many tasks as possible,

```
void executeTask(task& t) {
  assert(isActive(t));
  // the body of the task 't'
  {
    …
    // at any point that a task 'new_task' can be
    // created
    if (canTaskBeActive(new_task))
      spawnActive(new_task);
    else
      spawnPending(new_task);
    …
  }
  moveTaskToFinished(t);
  for ( auto t2: pendingTasks() )
    if (canTaskBeActive(t2))
      moveTaskToActive(t2);
}
```

Listing 1

if two tasks can cause a race condition bug, then there must be restriction between them or a dependency chain

```
bool canTaskBeActive(task& t) {
  for ( auto t2: directPredecessorOf(t) )
    if (!isFinished(t2))
      return false;
  for ( auto t2: restrictionsOf(t) )
    if (isActive(t2))
      return false;
  return true;
}
```

Listing 2

without considering that there may be combinations in which holding up tasks may lead to better parallelism. In practice, this tends to work really well due to the scheduling non-determinism.

A possible implementation of the test to check if a task can be active is given in Listing 2.

Let us now prove that this scheduling algorithm can safely and efficiently be used to implement any concurrent algorithm that is properly divided into tasks.

Lemma 1 (termination). If the concurrent algorithm needs a finite set of finite tasks that do not have dependency cycles, then the scheduling algorithm will eventually execute all the tasks, leading to the completion of the algorithm.

We start with at least one *active* task, and by the problem description, we are always going to finish executing the current set of active tasks.

If there are *pending* tasks, by executing all the currently *active* tasks at least one *pending* task will be transitioned to *active* (as seen above this transition happens greedily, as soon as possible). By the fact that the tasks don't have dependency cycles, and as the number of tasks is finite, there must be at least one task *A* in the *pending* group that doesn't have dependent tasks in the *pending* group. By construction, we put task A in the *pending* group only if there is an *active* task *B* such as *B*→*A* or *B*~*A*. But *B* is guaranteed to finish at some point; at that point, task *A* can become *active*. And, by the construction of our scheduling algorithm, we always check what tasks can be transitioned to *active* whenever a task completes. Therefore, *A* will be transitioned to *active* whenever *B* completes. That shows that we are always making progress.

As we start with a non-empty set of active tasks and we are always making progress, we eventually will execute all the tasks in the algorithm (remember, the algorithm has a finite set of tasks).

Q.E.D.

Lemma 2 (soundness). If the dependencies and restrictions between the tasks are set correctly (i.e., if two tasks can cause a race condition bug, then there must be restriction between them or a dependency chain), the scheduling algorithm will never schedule two tasks at the same time that can cause race condition bugs.

By the construction of the algorithm, we cannot execute in parallel tasks that are not present in the *active* group at the same time.

We add to the *active* group one task at a time. Let's say that *A* and *B* are *active* tasks, with *A* added before *B*. Both tasks could not have non-finished predecessors, as otherwise they would not have been added to the *active* group; so there can't be a dependency between them. Also, it can't be that *A*~*B* as *B* would not have been added to the *active* group. Thus, there can be no incompatibility between the tasks in the *active* group. If there can't be any incompatibility between two *active* tasks, and we never have in execution tasks that are not in the *active* group at that time, we can't have two incompatible tasks in execution at the same time.

Q.E.D.

Lemma 3 (efficiency). On a machine with infinite task parallelism, at any given point in time (except points in which we are running the scheduling logic) one cannot schedule for execution another task while maintaining the soundness of the algorithm as expressed just by the constraints between the tasks.

Please note that this aims for the efficiency of a greedy algorithm. In other words, this scheduling algorithm ensures a local maximal capacity, and not necessarily a global optimum. Again, due to timing non-determinism, a greedy scheduling works well in practice.

A key point of this lemma is to ignore the time spent by the scheduling algorithm. Of course, there is a small window of time, while we are deciding which task to make *active*, in which that task can be *active* and it isn't yet. We are just focusing on the times in which all the queued tasks are still executing.

Also, it's important to note that as we are looking at the incompatibility between tasks as expressed just by the constraints between the tasks, the constraints between tasks apply for the whole duration of the tasks. In other words, we cannot say that a constraint for a task vanishes after a task executes half-way through. The tasks are atomic with respect to constraints.

Let us assume that *S* is the current set of *active* tasks and that *B* is a task still in *pending* state, but that can be run without being unsound. But, if *B* can be safely run, it means that it can be an *active* task. So, if *B* could be an *active* task and it isn't, it means that there was a point in time at which *B* could have been categorised as *active* (but the scheduling algorithm didn't). In other words, there must be a time when the active constraints for task *B* change. As the tasks are atomic, these constraints can only happen when *B* is created, or whenever a task *A* for which *A*→*B* or *A*~*B* is completed. But the algorithm covers both cases: both when *B* is created and whenever another task finishes, we re-evaluate whether *B* can be moved to the *active* group. So, there cannot be another time in which *B* could be scheduled, and *B* is added to the *active* group as soon as it possibly can. This means that our scheduling algorithm always guarantees that the set *S* of active tasks is always at its maximal capacity (assuming a greedy strategy of creating this set).

Q.E.D.

Theorem. There is a scheduling algorithm that can be used to schedule the execution of any concurrent algorithm properly expressed with tasks smaller scheduling algorithms that can be used to solve various types of common problems, and then we can easily compose them to get an overall scheduling algorithm

and constraints between tasks; the scheduling algorithm has the maximum efficiency it can have with greedy scheduling.

The proof follows immediately from Lemma 1, Lemma 2 and Lemma 3.

Q.E.D.

In this section we constructed a general scheduling algorithm that can be used to solve any type of concurrent algorithms, we proved its viability and that its theoretical efficiency is maximal. However, in practice, we don't necessarily need a general/global algorithm for scheduling. We can instead have smaller scheduling algorithms that can be used to solve various types of common problems, and then we can easily compose them to get an overall scheduling algorithm. Let's look at a few such particular scheduling algorithms

Replacing locks

Most probably the reader is familiar with solving concurrency problems by utilizing locks. Thus, probably it will be easier to just have a direct translation of problems that can be solved with locks into problems that can be solved with tasks. We will present three specialised schedulers to cover the same effect as one would traditionally cover with mutexes, semaphores and read-write mutexes.

The serializer

Let us attempt to model the behaviour of a mutex with tasks. A mutex protects one of the multiple regions of code against access from multiple threads. For simplicity, let us assume that the entire region covered by a lock is atomic – there are no other locks in it, and the parallelisation constraints with other regions do not change midway. This assumption is an invitation to make that zone of code a task. We would actually have tasks for all the invocations of the zones protected by the mutex.

If the mutex M protects zones/tasks $A_1, A_2, \ldots, A_n$ then the run-time effect of the mutex would be similar to imposing restrictions between any of the tasks: $A_i \sim A_j, \forall i,j$.

Therefore, we can introduce the so-called *serializer*, as a local scheduler that can accept tasks and ensure that there are restriction relations between all the tasks enqueued in it. As any two tasks enqueued into it will have a restriction between them, a serializer will only schedule one task at a time for execution – therefore the name: serializer.

Figure 2 shows the restriction relationships between 4 tasks that can be enqueued into a serializer, while Figure 3 shows a possible execution order.

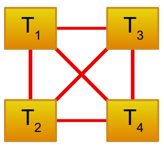


Figure 2

```
concore::serializer my_ser;
backup_engine my_backup_engine;

void trigger_backup(app_data data) {
  my_ser([=]{ my_backup_engine.save(data); });
}
```

Listing 3

The implementation of a serializer can be easily made in the following way:

- the abstraction should have a method of enqueueing a task into the serializer
- keep a list of pending tasks that are enqueued in the serializer but are not yet executed; also keep track of whether we are executing any task or not (i.e., by using an atomic variable)
- whenever a new task is enqueued, if there is no other task in execution, start executing the new task
- whenever a new task is enqueued, if there is another task in execution, add it to the pending list
- whenever a task finishes executing, check if there are other tasks in the pending list; if the answer is yes, start executing the first task in the pending list

The serializer can be implemented relatively easily in about 100 lines of C++ code. If the reader is curious, I would invite the reader to check the implementation inside the Concore library.

Listing 3 shows an example of using a serializer in Concore. We assume that the application needs to have backups of the main data at a certain point, two backups cannot run in parallel, and the way the application is constructed, one can trigger the backups from multiple threads/tasks, possible in parallel.

Regardless of how many threads will call **`trigger_backup()`**, the backup engine's **`save()`** method will not be called in parallel from multiple threads. Simple, right?

n_serializer

A semaphore is in one way a generalisation of a mutex. Instead of allowing only one region of code to be executed in parallel, it allows N such regions, based on how the semaphore is configured. Semaphores can be used in different ways too (for example, as notification primitives), but let us focus on protecting regions of code. From this perspective, if we translate this in terms of tasks, we can say that the effect would be to allow N tasks to run in parallel.

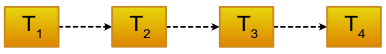


Figure 3

```
concore::n_serializer my_ser{10};
concore::concurrent_queue<backup_engine>
  my_backup_engines; // at least 10

void trigger_backup(app_data data) {
  my_ser([=]{
    // acquire a free backup engine
    backup_engine engine;
    bool res = my_backup_engines.try_pop(engine);
    assert(res);
    // do the backup
    engine.save(data);
    // release the backup engine to the system
    my_backup_engines.push(std::move(engine));
  });
}
```

Listing 4

To provide such an abstraction, Concore implements the **n_serializer** class. It is similar to **serializer** but it allows a configurable number of tasks to be run in parallel. The implementation follows the same pattern as for the **serializer**, so we won't describe it here.

Let us also give an example of how to use an **n_serializer** with Concore. To continue the previous example, let's assume that saving a backup can be an expensive operation, so that it can queue up in certain cases, producing delays in saving the backups; for performance reasons, we want to create multiple backup engines that save the backups at different locations in parallel. Listing 4 shows how this can be implemented with Concore.

The example shows a bit more than just the serializer; it also showcases the use of a concurrent queue to operate on multiple data objects from different tasks in parallel. Again, simple enough.

rw_serializer

Another generalisation of a mutex is a read-write mutex. This type of mutex allows multiple "read" operations to execute in parallel but does not allow any "write" operations to be executed in parallel with any other operations.

Just like in the previous two cases, we can express read-write mutexes with tasks. We create two types of tasks: *read* tasks and *write* tasks. The *read* tasks can be executed in parallel, but *write* tasks cannot be executed in parallel with *read* tasks or other *write* tasks. The runtime constraints between them are relatively straight-forward: just create restriction constraints between *write* tasks and everything else. Figure 4 shows an example of how we can encode 5 read tasks and 2 write tasks.

The implementation of such an abstraction is slightly more involved than the implementation of a serializer. For this, we need to have two queues of tasks: one for *read* tasks and one for *write* tasks. Then we need to decide which tasks take priority: *write* tasks or *read* tasks. An implementation that favours *write* tasks will not execute any *read* tasks while there are *write* tasks in the pending queue. Conversely, an implementation that favours *read* tasks will not execute any *write* tasks while there are *read* tasks; continuously adding *read* tasks may make the *write* tasks starve. After deciding which type of task has priority, the implementation follows the same pattern, with the caveat that it's more difficult to ensure atomicity for various operation in a multi-threaded environment. Slightly more complex, but not necessarily hard.

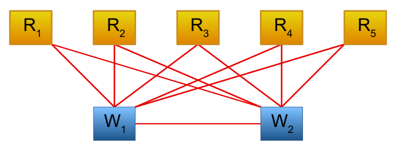


Figure 4

```
concore::rw_serializer my_ser;
backup_engine my_backup_engine;

void get_latest_backup_info(backup_info& dest,
    concore::task& next) {
  my_ser.reader()([&dest, &next] {
    // query backup data
    dest = std::move(my_backup_engine.get_info());
   // query is done; continue with a new task after
    // the data is ready
    concore::global_executor(std::move(next));
  });
}
```

Listing 5

Concore implements an abstraction for this type of problems called **rw_serializer**. Its implementation favours *write* tasks over *read* tasks.

To exemplify this type of abstraction, let's consider the case in which we want to allow easy querying of information from the backup engine, but doing a backup cannot run in parallel with any querying operation. As opposed to our previous example for **n_serializer**, we consider that the save operations are relatively easy to perform, and they occur less frequently. Listing 5 shows how this can be implemented with Concore.

The above example also showcases how task continuations can be implemented by hand, in the absence of more elaborate scheduler. Again, simple.

Replacing locks, discussion

In this section, we showed that one can build local schedulers to replace locks with tasks. They provide convenient ways to attack the problem and, in terms of implementation, they are more efficient than a general scheduling algorithm. If one's application can be relatively easily translated into tasks, using these serializers should be straight-forward.

However, there are a few complications. I cannot properly end this article in a very positive manner without touching on the main problems that one would face whenever one would try to put this into practice. So, here they are:

- How can we add a section of code/task in two serializers at once (i.e., like taking two locks for a region)?
- How can we break down the code to protect a zone of code in the middle of a task?

The first question discusses the composability of two task schedulers. Without entering in too many details we will just state that tasks are more composable than locks. Locks are known for not being composable [Lee06]. On the other hand, given two task schedulers, we can compose them, even without knowing their inner structure. There are multiple ways of doing this, but probably the most prominent method is the fork-join model [McCool12, Robison14]; this is the default programming model for both Cilk [cilk] and its successor CilkPlus [cilkplus]. The main idea is that, while executing a task, one can fork (spawn) a new task, and later on wait for that task to complete.

In our case, one would enqueue one task in a serializer that would enqueue-and-wait another task in the second serializer, and this second task will actually do the work. Listing 6 provides a quick example of *composability*. It's not complicated to do such a thing. The logic is a bit wired, but that's inherent to the problem that we are trying to solve; one if far better to avoid mixing multiple scheduling systems for the same task (in terms of locks; avoid using multiple locks to protect certain regions).

The second problem may not be apparent when looking from a theoretical perspective, but it's immediately visible when trying to change an existing system that currently uses locks to use tasks instead. Let's assume that one has an application that can be broken down in tasks relatively easily. But, at some point, deep down in the execution of a task one would need a lock. Ideally one would break the task in 3 parts: everything that goes before the lock, the protected zone, and everything that goes after. Still, easier said

```
concore::serializer my_ser1, my_ser2;

void double_serialize(std::function<void()> work)
{
  my_ser1([work = std::move(work)]{
    concore::spawn_and_wait([work =
       std::move(work)]{
      my_ser2(work);
    });
  });
}
```

Listing 6

```
concore::serializer my_ser;

void deep_fun() {
    some_work();
    concore::spawn_and_wait([=]{
        my_ser([=]{ my_protected_work(); });
    });
    some_other_work();
}
void fun_with_depth(int level) {
    do_pre_work(level);
    if (level > 0) fun_with_depth(level-1);
    else deep_fun();
    do_post_work(level);
}

fun_with_depth(20);
```

Listing 7

than done; this can be hard if one is 20 layers of function-calls – it's not easy to break all these 20 layers of functions into 3 tasks and keep the data dependencies correct.

If breaking the task into multiple parts is not easily doable, then one can also use the fork-join model to easily get out of the mess. A sample code is shown in Listing 7.

This technique is very useful not just for transforming locks into tasks. It can be used also to break applications into tasks that are used in other ways – for example, to model some asynchronous flows.

With that, we covered the two most prominent problems that people might have. Oh, wait… we still need to have *that* discussion on performance.

Performance

The sceptic reader might argue that using serializers is no better than using a mutex. After all, all the tasks are executed one after the other, without exercising parallelism. And yes, the last statement is true.

The key point of the serializers is that they do not block any threads. Threads can still be free to execute other tasks while certain tasks need serial execution.

Let's try an analogy. Mutexes are like intersections (Kevlin Henney has nicely explained this in [Henney18]). While a car goes through an intersection, all other cars from the adjacent roads are blocked. If a city has a lot of crowded intersections, then traffic congestions can easily spread across large parts of the city – severe losses in the performance of the traffic system. Serializers are like (isolated) drive-throughs – the cars are waiting in line for the cars in front of them to be served (n_serializers can be associated with drive-ins). But the key point here is that these drive-throughs have sufficiently lengthy lanes – the cars waiting at the drive-through are not blocking the nearby intersections. In this case, the number of cars that can circulate compensates for a few cars that are waiting in the queue.

Hope that clears the performance worries for most of the readers. However, some astute readers might still wonder about the performance of the fork-join model, since we invoked it twice as a solution to common problems. Of course, they would argue, the performance of the application is affected when we are waiting for tasks to complete. But this is not the case. Most libraries these days (and Concore is one of them) can implement busy-waiting – that is, execute other tasks while waiting on a task to complete. With such a strategy, we are not blocking the worker threads, so we are still fully utilising the CPU to do real work.

Conclusions

The previous article [Teodorescu20] showed that if one can make a concurrent algorithm with tasks, it can obtain good parallel speedups (under certain conditions), especially when compared with using mutexes. This article proves that any concurrent algorithm can be implemented safely and efficiently with tasks and completely avoid mutexes. Thus, there is no reason to use mutexes anymore in high-level code.

Besides the proof, the article showed that one can use simpler structures (serializers) to move away from lock-based synchronisation primitives towards tasks, and that it is not a complicated endeavour. Another argument that we need to stop using mutexes.

So, please, please, please, unless you are building low-level concurrency primitives, avoid using mutexes, just like you avoid `goto`s (which I hope you are already avoiding).

The article also started to explore a few patterns that can be used for concurrent programming. The point was not to showcase the most important patterns, but to somehow exemplify the concepts discussed here and show that it isn't hard to put them in practice. However, these hints oblige me to provide a follow-up article that explores patterns for designing concurrent applications; after all, software engineering is about design issues far more than it is about coding. ■

References

[cilk] MIT, The Cilk Project, http://supertech.csail.mit.edu/cilk/

[cilkplus] Intel, Cilk Plus, https://www.cilkplus.org

[concore] Lucian Radu Teodorescu, Concore library, https://github.com/lucteo/concore

[Henney18] Kevlin Henney, Concurrency Versus Locking, https://www.youtube.com/watch?v=mEtoXwB9HFk

[Lee06] Edward A. Lee, The Problem with Threads, *Technical Report UCB/EECS-2006-1*, 2006, https://www2.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf

[McCool12] Michael McCool, Arch D. Robison, James Reinders, *Structured Parallel Programming: Patterns for Efficient Computation*, Morgan Kaufmann, 2012

[Robison14] Arch Robison, A Primer on Scheduling Fork-Join Parallelism with Work Stealing, *Technical Report N3872*, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3872.pdf

[tbb] Intel, Threading Building Blocks library, https://github.com/oneapi-src/oneTBB

[Teodorescu20] Lucian Radu Teodorescu, 'Refocusing Amdahl's Law', *Overload* 157, June 2020

C++20: A Simple Math Module

Modules are one big change introduced by C++20. Rainer Grimm describes how to build a mathematics module.

Modules are one of the four prominent features of C++20. They overcome the restrictions of header files and promise a lot: faster build-times, fewer violations of the One-Definition-Rule, less usage of the preprocessor.

The long history of modules in C++

Modules may be older than you think. My short historic detour should give you an idea how long it takes to get something so valuable into the C++ standard.

In 2004, Daveed Vandevoorde wrote proposal N1736.pdf [N1736], which described the idea of modules for the first time. It took until 2012 to get a dedicated Study Group (SG2, Modules) for modules. In 2017, Clang 5.0 and MSVC 19.1 provided the first implementation. One year later, the Modules TS (technical specification) was finalized. Around the same time, Google proposed the so-called ATOM (Another Take On Modules) proposal [P0947] for modules. In 2019, the Modules TS and the ATOM proposal was merged into the C++20 committee draft [N4842], which is the syntax I use when writing about modules.

The C++ standardization process is democratic. The section Standardization [ISO] gives you more information about the standard and the standardization process. Figure 1 shows the various study groups.

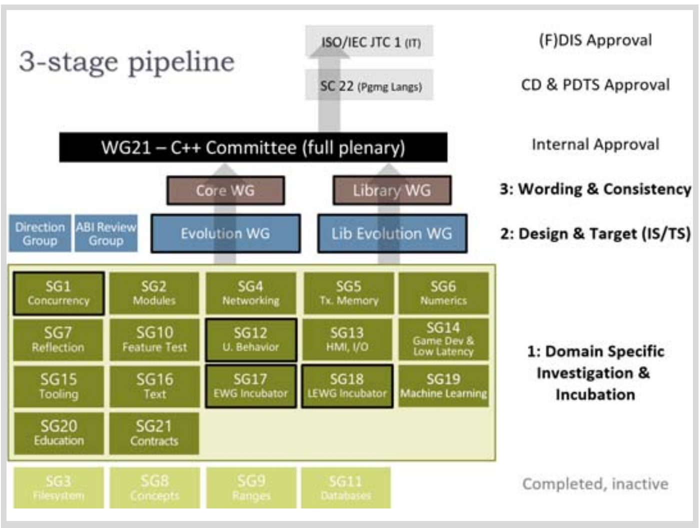


Figure 1

Rainer Grimm Rainer has 20 years of experience as a software developer and software architect, a good 10 years as a training manager and seminar leader, and 3 years as a team leader in software. He has published several books on C ++, 70 articles on C ++, Python and Haskell for Linux-Magazin and iX Magazin and presents at specialist conferences. He can be contacted through https://www.modernescpp.net/

Explaining modules from a user's perspective is quite easy, but this does not hold true for the implementer's perspective. My plan for this article is to start with a simple modules math and add more features to it as we go.

The math module

First, here is my first module:

```
// math.ixx
export module math;
export int add(int fir, int sec){
  return fir + sec;
}
```

The expression **`export module math`** is the module declaration. By putting **`export`** before the function **`add`**, **`add`** is exported and can, therefore, be used by a consumer of my module.

```
// client.cpp
import math;
int main() {
  add(2000, 20);
}
```

`import math` imports the module **`math`** and makes the exported names in the module visible to the `client.cpp`. Let me say a few words about module declaration files before I build the module.

Module declaration files

Did you noticed the strange name of the module: `math.ixx`.

- `cl.exe` (Microsoft) uses the required extension `ixx`. The `ixx` stands for a module interface source.
- Clang uses the extension `cppm`. `cppm` presumably stands for a `cpp` module declaration. Wrong!!! The documentation to Clang is misleading. Stop using the `cppm` extension until have you read my post about my attempt [Rainer20]. Use the extension `cpp`. I assume you don't want to make the same Odyssey as me.
- I don't know of a GCC extension.

Compile the module math

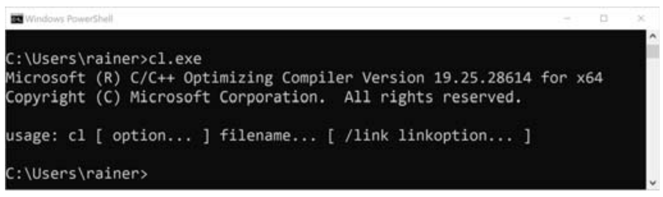
To compile the module, you have to use a very current Clang, GCC, or `cl.exe` compiler. In this article, I'm using `cl.exe` on Windows. The Microsoft blog provides two excellent introductions to modules: Overview of modules in C++ [Microsoft-1] and C++ Modules conformance improvements with MSVC in Visual Studio 2019 16.5 [Microsoft-2]. In contrast, the lack of introductions to the Clang and GCC compilers makes it quite difficult to use modules.

Figure 2 shows more details of the Microsoft compiler I used.

These are the steps to compile and use the module with the Microsoft compiler. I only show the minimal command line. With an older Microsoft compiler, you have to use at least **`/std:cpplatest`**.

```
cl.exe /experimental:module /c math.ixx ①
cl.exe /experimental:module client.cpp math.obj ②
```

Explaining modules from a user's perspective is quite easy, but this does not hold true for the implementer's perspective.



```
C:\Users\rainer>cl.exe
Microsoft (R) C/C++ Optimizing Compiler Version 19.25.28614 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]

C:\Users\rainer>
```

Figure 2

① Creates an obj file `math.obj` and an IFC file `math.ifc`. The IFC file contains the metadata description of the module interface. The binary format of the IFC is modeled after the Internal Program Representation by Gabriel Dos Reis and Bjarne Stroustrup (2004/2005).

② Creates the executable `client.exe`. Without the implicitly used `math.ifc` file from the first step, the linker can not find the module (see Figure 3).

For obvious reasons, I am not showing you the output of the program execution. Let me change this.

Global module fragment

The global module fragment is meant to compose module interfaces. It's a place to use preprocessor directives such as **`#include`** so that the module interface can compile. The code in the global module fragment is not exported by the module interface.

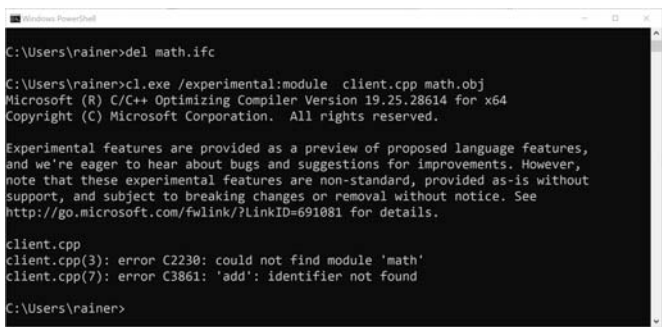
The second version of the module **`math`** supports the two functions **`add`** and **`getProduct`** (see Listing 1).

I included the necessary headers between the global module fragment (line 1) and the module declaration (line 2).

The client imports the module math and uses its functionality (see Listing 2 and Figure 4).

Maybe, you don't like using a Standard Library Header anymore. Microsoft supports modules for all STL headers. Here is what I found out from the Microsoft C++ team blog [Microsoft-3]:

- **`std.regex`** provides the content of the header **`<regex>`**



```
C:\Users\rainer>del math.ifc

C:\Users\rainer>cl.exe /experimental:module  client.cpp math.obj
Microsoft (R) C/C++ Optimizing Compiler Version 19.25.28614 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

Experimental features are provided as a preview of proposed language features,
and we're eager to hear about bugs and suggestions for improvements. However,
note that these experimental features are non-standard, provided as-is without
support, and subject to breaking changes or removal without notice. See
http://go.microsoft.com/fwlink/?LinkID=691081 for details.

client.cpp
client.cpp(3): error C2230: could not find module 'math'
client.cpp(7): error C3861: 'add': identifier not found

C:\Users\rainer>
```

Figure 3

```
module;                // global module fragment (1)

#include <numeric>
#include <vector>

export module math;  // module declaration (2)

export int add(int fir, int sec)
{
  return fir + sec;
}
export int getProduct(const std::vector<int>& vec)
{
  return std::accumulate(vec.begin(), vec.end(),
  1, std::multiplies<int>());
}
```

Listing 1

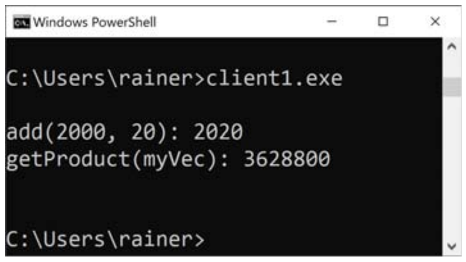
```
#include <iostream>
#include <vector>

import math;

int main() {
  std::cout << std::endl;
  std::cout << "add(2000, 20): "
    << add(2000, 20) << std::endl;
  std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8,
    9, 10};
  std::cout << "getProduct(myVec): "
    << getProduct(myVec) << std::endl;
  std::cout << std::endl;
}
```

Listing 2

- **`std.filesystem`** provides the content of the header **`<experimental/filesystem>`**
- **`std.memory`** provides the content of the header **`<memory>`**



```
C:\Users\rainer>client1.exe

add(2000, 20): 2020
getProduct(myVec): 3628800


C:\Users\rainer>
```

Figure 4

The global module fragment is meant to compose module interfaces ... The code in the global module fragment is not exported by the module interface

```
module;

import std.core;          // (1)

export module math;

export int add(int fir, int sec)
{
    return fir + sec;
}

export int getProduct(const std::vector<int>& vec)
{
  return std::accumulate(vec.begin(),
                         vec.end(),
                         1,
                         std::multiplies<int>());
}
```

Listing 3

```
import std.core;          // (1)

import math;

int main() {
  std::cout << std::endl;

  std::cout << "add(2000, 20): " << add(2000, 20)
    << std::endl;

  std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8,
    9, 10};

  std::cout << "getProduct(myVec): "
    << getProduct(myVec) << std::endl;

 std::cout << std::endl;
}
```

Listing 4

- `std.threading` provides the contents of headers:
  - `<atomic>`
  - `<condition_variable>`
  - `<future>`
  - `<mutex>`
  - `<shared_mutex>`
  - `<thread>`
- `std.core` provides everything else in the C++ Standard Library

To use the Microsoft Standard Library modules, you have to specify the exception handling model (`/EHsc`) and the multithreading library (`/MD`). Additionally, you have to use the flag `/std:c++latest`.

Listing 3 and Listing 4 are the modified versions of the interface file `math2.ixx` and the source file `client2.cpp` respectively.

Both files use – in line (1) – the module `std.core`. ■

This article was first published on Rainer's blog on 12 May 2020: https://www.modernescpp.com/index.php/cpp20-a-first-module

References

[ISO] 'Standardization': https://isocpp.org/std/

[Microsoft-1] 'Overview of modules in C++': https://docs.microsoft.com/en-us/cpp/cpp/modules-cpp?view=vs-2019

[Microsoft-2] 'C++ Modules conformance improvements with MSVC in Visual Studio 2019 16.5': https://devblogs.microsoft.com/cppblog/c-modules-conformance-improvements-with-msvc-in-visual-studio-2019-16-5/

[Microsoft-3] 'Using C++ Modules in Visual Studio 2017': https://devblogs.microsoft.com/cppblog/cpp-modules-in-visual-studio-2017/

[N1736] 'Modules in C++': http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1736.pdf

[N4842] 'N4842 Post-Belfast 2019 C++ working draft': https://github.com/cplusplus/draft/releases/tag/n4842

[P0947] 'Another take on Modules': http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0947r1.html

[Rainer20] 'C++20: Module Interface Unit and Module Implementation Unit' posted 25 May 2020 at: https://www.modernescpp.com/index.php/c-20-module-interface-unit-and-module-implementation-unit

A Thorough Introduction to Apache Kafka

Kafka serves as the heart of many companies' architecture. Stanislav Kozlovski takes a deep dive into the system.

Kafka is a word that gets heard a lot nowadays. A lot of leading digital companies seem to use it. But what is it actually?

Kafka was originally developed at LinkedIn in 2011 and has improved a lot since then. Nowadays, it's a whole platform, allowing you to redundantly store absurd amounts of data, have a message bus with huge throughput (millions/sec), and use real-time stream processing on the data that goes through it all at once.

This is all well and great, but stripped down to its core, Kafka is a distributed, horizontally scalable, fault-tolerant commit log.

Those were some fancy words – let's go at them one by one and see what they mean. Afterwards, we'll dive deep into how it works.

Distributed

A *distributed* system is one which is split into multiple running machines, all of which work together in a cluster to appear as one single node to the end user. Kafka is distributed in the sense that it stores, receives, and sends messages on different nodes (called *brokers*).

The benefits to this approach are high scalability and fault tolerance.

Horizontally scalable

Let's define the term *vertical scalability* first. Say, for instance, you have a traditional database server that's starting to get overloaded. The way to get this solved is to simply increase the resources (CPU, RAM, SSD) on the server. This is called *vertical scaling* – where you add more resources to the machine. There are two big disadvantages to scaling upwards:

- There are limits defined by the hardware. You cannot scale upwards indefinitely.
- It usually requires downtime, something which big corporations can't afford.

Horizontal scalability is solving the same problem by throwing more machines at it. Adding a new machine doesn't require downtime, nor are there any limits to the amount of machines you can have in your cluster. The catch is not all systems support horizontal scalability, as they're not designed to work in a cluster, and those that *are* are usually more complex to work with.

Figure 1 shows that horizontal scaling becomes much cheaper after a certain threshold.

Fault-tolerant

Something that emerges in nondistributed systems is they have a single point of failure (SPoF). If your single database server fails (as machines do) for whatever reason, you're screwed.

Distributed systems are designed in such a way to accommodate failures in a configurable way. In a 5-node Kafka cluster, you can have it continue working even if two of the nodes are down. It's worth noting that fault tolerance is at a direct trade-off with performance, as in the more fault-tolerant your system is, the less performant it is.

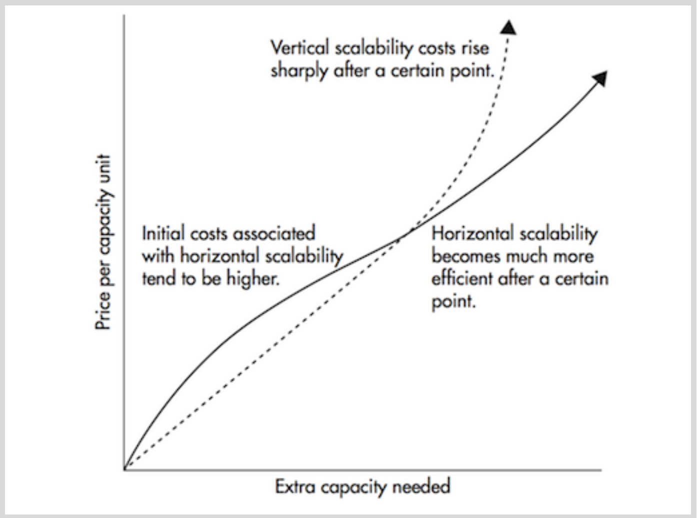


Figure 1

Commit log

A *commit* log (also referred to as *write-ahead* log or a *transaction* log) is a persistent-ordered data structure that only supports appends. You can't modify or delete records from it. It's read from left to right and guarantees item ordering. Figure 2 is an illustration of a commit log [Kreps13].

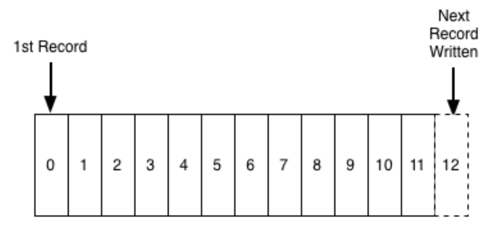


Figure 2

"*Are you telling me that Kafka is such a simple data structure?*"

Stanislav Kozlovski Stanislav began his programming career racing through some coding academies and bootcamps, where he aced all of his courses and began work at SumUp, a German fintech company aiming to become the first global card acceptance brand. He was later recruited into Confluent, a company offering a hosted solution and enterprise products around Apache Kafka. Contact him on Twitter, where he's @BdKozlovski or at Stanislav_Kozlovski@outlook.com

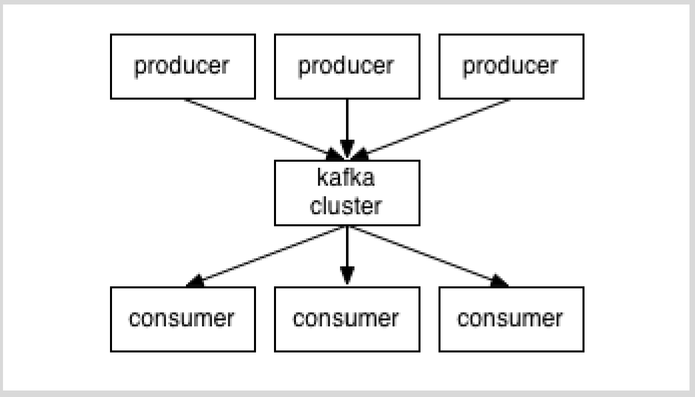


Figure 3

In many ways, yes. This structure is at the heart of Kafka and is invaluable, as it provides ordering, which in turn provides deterministic processing. Both of which are nontrivial problems in distributed systems.

Kafka actually stores all of its messages to disk (more on that later), and having them ordered in the structure lets it take advantage of sequential disk reads.

- Reads and writes are a constant time O(1) (*knowing the record ID*), which compared to other structure's O(log N) operations on disk is a huge advantage, as each disk seek is expensive
- Reads and writes don't affect each other. Writing wouldn't lock reading and vice versa (as opposed to balanced trees).

These two points have huge performance benefits, since the data size is completely decoupled from performance. Kafka has the same performance whether you have 100 KB or 100 TB of data on your server.

How does it work?

Applications (*producers*) send messages (*records*) to a Kafka node (*broker*), and said messages are processed by other applications called *consumers*. Said messages get stored in a *topic* and consumers subscribe to the topic to receive new messages (see Figure 3).

As topics can get quite big, they get split into *partitions* of a smaller size for better performance and scalability. (For example, say you were storing user login requests. You could split them by the first character of the user's username.)

Kafka guarantees that all messages inside a partition are ordered in the sequence they came in. The way you distinct a specific message is through its *offset*, which you could look at as a normal array index, a sequence number which is incremented for each new message in a partition (see Figure 4).

Kafka follows the principle of a dumb broker and smart consumer. This means that Kafka doesn't keep track of what records are read by the consumer, then deleting them. Rather, it stores them for a set amount of

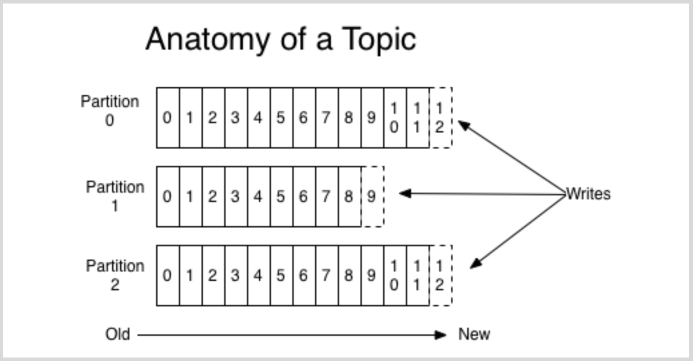


Figure 4

time (e.g., one day) or until some size threshold is met. Consumers, themselves, poll Kafka for new messages and say what records they want to read. This allows them to increment/decrement the offset they're at as they wish, thus being able to replay and reprocess events.

It's worth noting consumers are actually consumer groups that have one or more consumer processes inside. In order to avoid two processes reading the same message twice, each partition is tied to only one consumer process per group. See Figure 5 for a representation of the data flow.

Persistence to disk

As I mentioned earlier, Kafka actually stores all of its records to disk and doesn't keep anything in RAM. You might be wondering how this is in the slightest way a sane choice. There are numerous optimizations behind this that make it feasible:

- Kafka has a protocol that groups messages together. This allows network requests to group messages together and reduce network overhead; the server, in turn, persists chunk of messages in one go, and consumers fetch large linear chunks at once.
- Linear reads/writes on a disk are fast. The concept that modern disks are slow is because of numerous disk seeks, something that's not an issue in big linear operations.
- Said linear operations are heavily optimized by the OS, via *read-ahead* (prefetch large block multiples) and *write-behind* (group small logical writes into big physical writes) techniques.
- Modern OSes cache the disk in free RAM. This is called *pagecache*.
- Since Kafka stores messages in a standardized binary format unmodified throughout the whole flow (producer → broker → consumer), it can make use of the *zero-copy* optimization. That's when the OS copies data from the pagecache directly to a socket, effectively bypassing the Kafka broker application entirely.

All of these optimizations allow Kafka to deliver messages at near network speed.

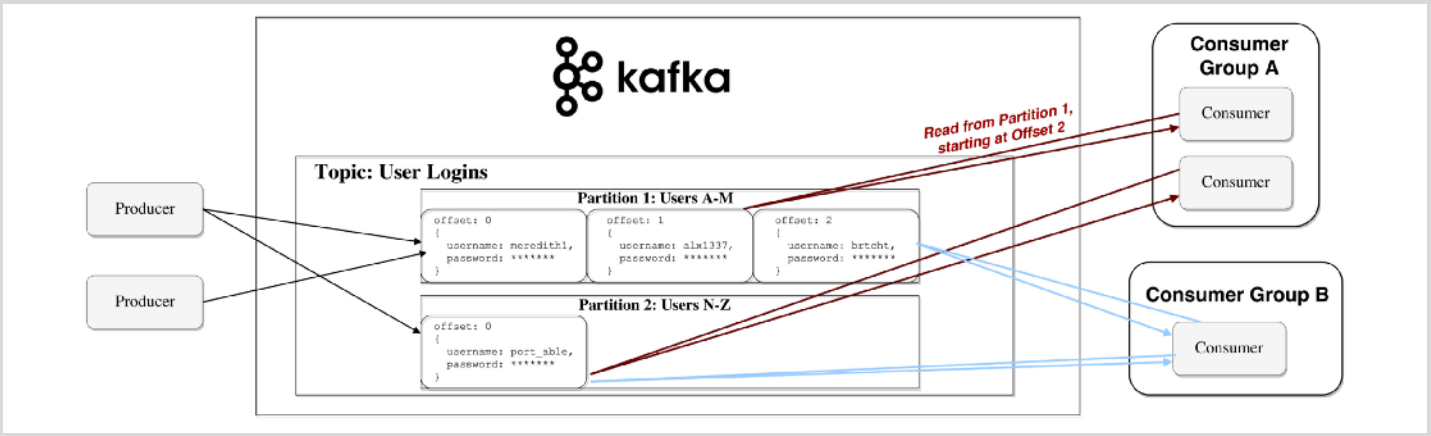

kafka
Topic: User Logins
Partition 1: Users A-M
offset: 0 { username: meredith1, password: ******* }
offset: 1 { username: alx1337, password: ******* }
offset: 2 { username: brtcht, password: ******* }
Partition 2: Users N-Z
offset: 0 { username: port_able, password: ******* }
Producer
Producer
Read from Partition 1, starting at Offset 2
Consumer Group A
Consumer
Consumer
Consumer Group B
Consumer

Figure 5

Data distribution and replication

Let's talk about how Kafka achieves fault tolerance and how it distributes data between nodes.

Data replication

Partition data is replicated across multiple brokers in order to preserve the data in case one broker dies.

At all times, one broker owns a partition and is the node through which applications write/read from the partition. This is called a *partition leader*. It replicates the data it receives to *n* other brokers, called *followers*. They store the data as well and are ready to be elected as leader in case the leader node dies.

This helps you configure the guarantee that any successfully published message won't be lost. Having the option to change the replication factor lets you trade performance for stronger durability guarantees, depending on the criticality of the data. Figure 6 shows 4 Kafka brokers with a replication factor of 3.

In this way, if one leader ever fails, a follower can take his place.

You may be asking, though:

"*How does a producer/consumer know who the leader of a partition is?*"

For a producer/consumer to write/read from a partition, they need to know its leader, right? This information needs to be available from somewhere.

Kafka stores such metadata in a service called ZooKeeper.

What's ZooKeeper?

ZooKeeper is a distributed key-value store. It's highly optimized for reads, but writes are slower. It's most commonly used to store metadata and handle the mechanics of clustering (heartbeats, distributing updates/configurations, etc).

It allows clients of the service (the Kafka brokers) to subscribe and have changes sent to them once they happen. This is how brokers know when to switch partition leaders. ZooKeeper is also extremely fault tolerant, and it ought to be, as Kafka heavily depends on it.

It's used for storing all sort of metadata, to mention some:

- Consumer groups offset per partition (although modern clients store offsets in a separate Kafka topic)
- Access control lists (ACLs) – used for limiting access/authorization
- Producer and consumer quotas –maximum message/sec boundaries
- Partition leaders and their health

How does a producer/consumer know who the leader of a partition is?

Producers and consumers used to directly connect and talk to ZooKeeper to get this (and other) information. Kafka has been moving away from this coupling, and since versions 0.8 and 0.9 (released 5 years ago) clients have been fetching metadata information from Kafka brokers directly, who themselves talk to ZooKeeper.

Kafka needs no keeper

Recently, the Kafka community has started moving away from ZooKeeper with a concrete proposal to use a self-managed metadata quorum based on the Raft algorithm [Apache-2]. Figure 7 shows the metadata flow.

Streaming

In Kafka, a stream processor is anything that takes continual streams of data from input topics, performs some processing on this input, and produces a stream of data to output topics (or external services, databases, the trash bin – wherever really).

It's possible to do simple processing directly with the producer/consumer APIs; however, for more complex transformations like joining streams together, Kafka provides a integrated Streams API library [KafkaAPI].

This API is intended to be used within your own codebase – it's not running on a broker. It works similar to the consumer API and helps you scale out the stream processing work over multiple applications (similar to consumer groups).

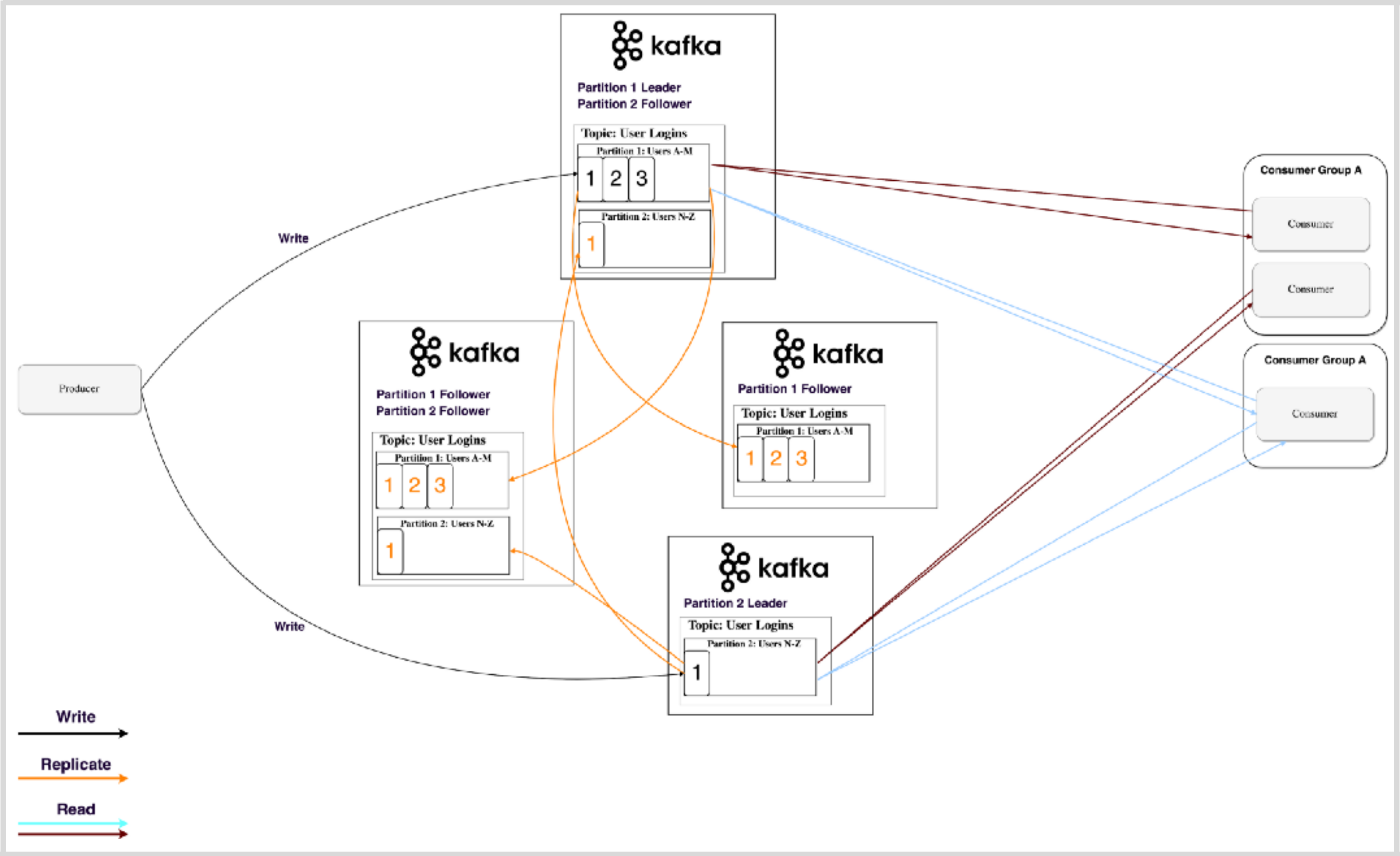


Figure 6

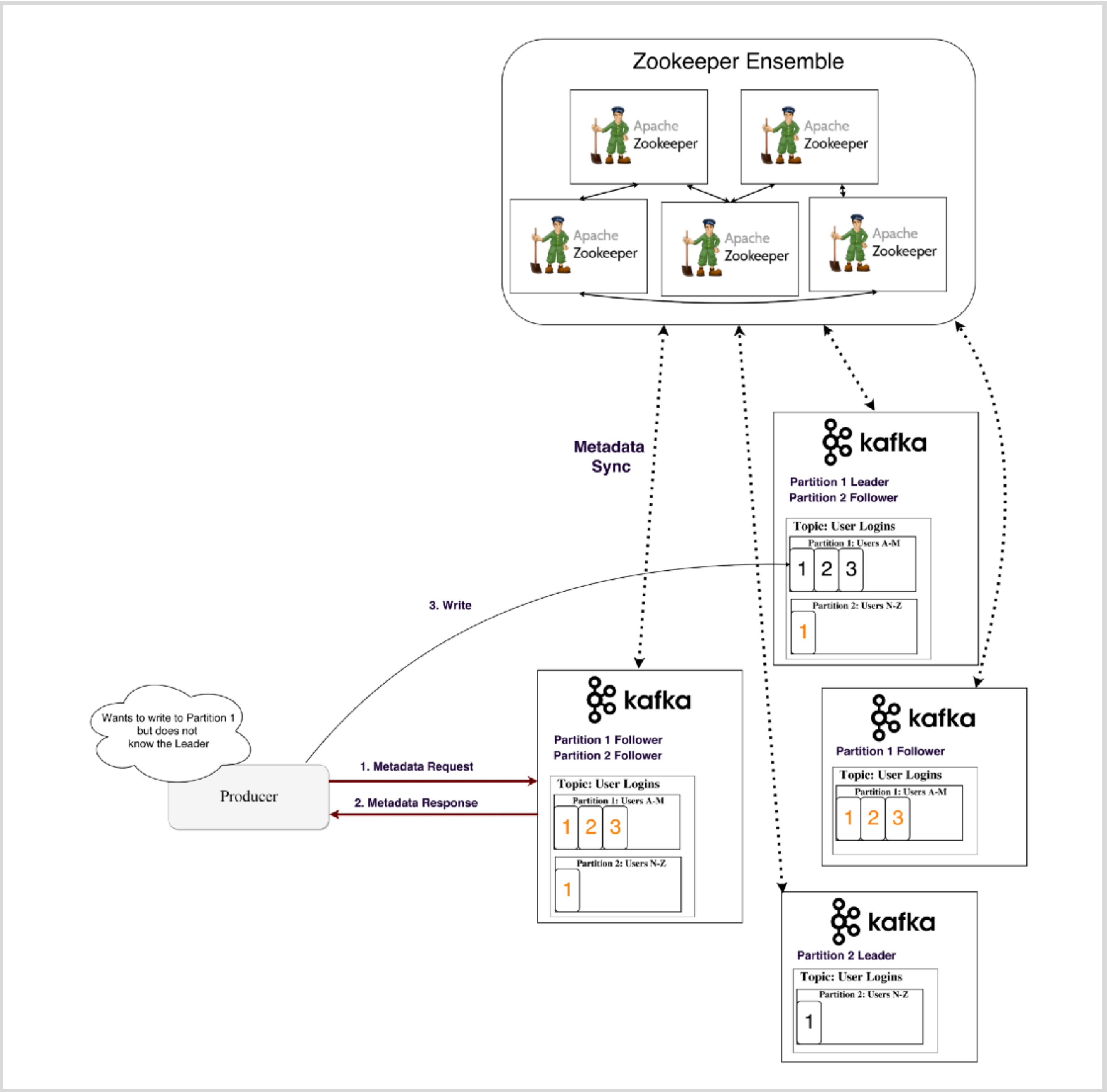


Figure 7

Stateless processing

A stateless processing of a stream is deterministic processing that doesn't depend on anything external. You know that for any given data, you'll always produce the same output independent of anything else. An example for that would be simple data transformation – appending something to a string `"Hello"` → `"Hello, World!"` (see Figure 8).

Stream-table duality

It's important to recognize that streams and tables are essentially the same. A stream can be interpreted as a table, and a table can be interpreted as a stream.

Stream as a table

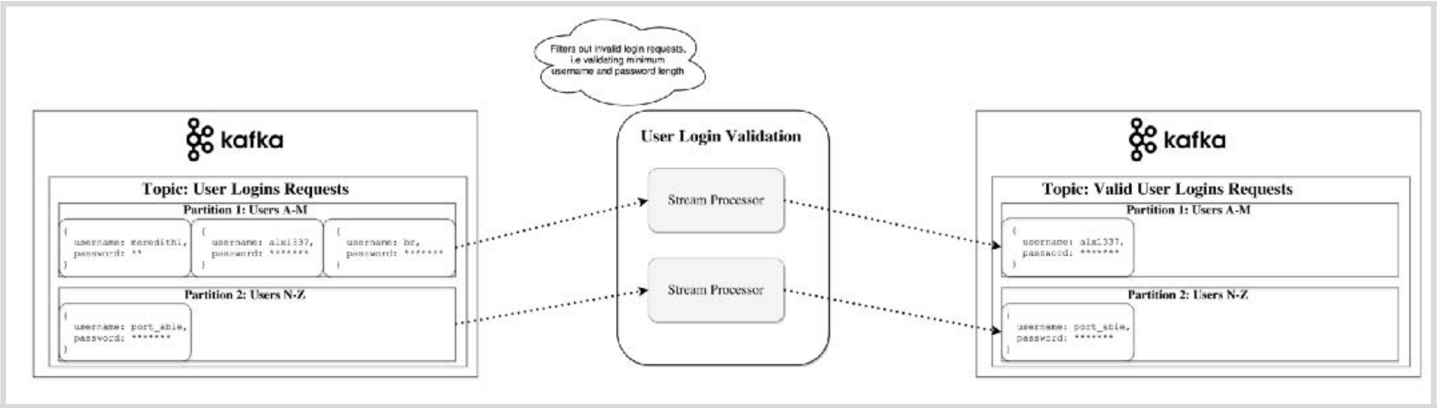
A stream can be interpreted as a series of updates for data, in which the aggregate is the final result of the table. This technique is called *event sourcing* [Fowler05].

If you look at how synchronous database replication is achieved, you'll see it's through the so-called *streaming replication*, where each change in a table is sent to a replica server. Another example of event sourcing are blockchain ledgers – a ledger is a series of changes as well.

A Kafka stream can be interpreted in the same way – events which, when accumulated, form the final state. Such stream aggregations get saved in a local RocksDB [GitHub] (by default) and are called a *KTable*. Figure 9 illustrates that each record increments the aggregated count.

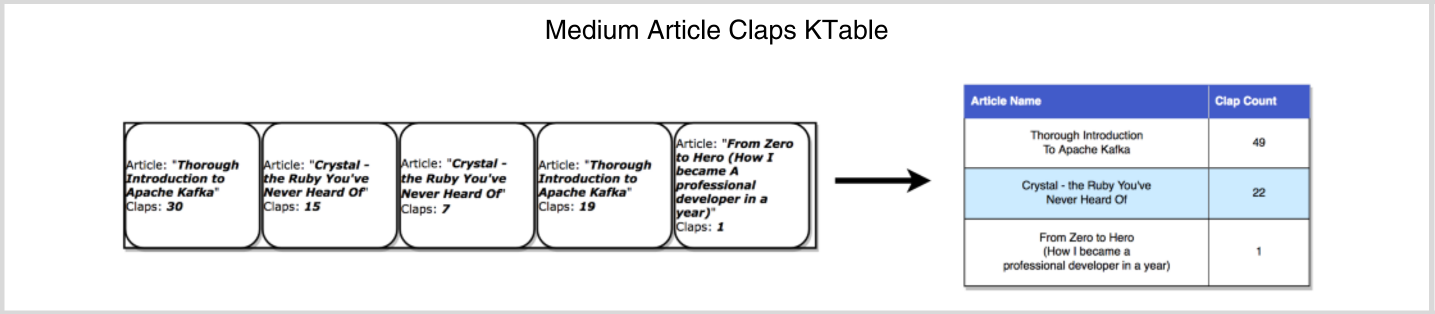
Table as a stream

A table can be looked at as a snapshot of the latest value for each key in a stream. In the same way that stream records can produce a table, table updates can produce a changelog stream (see Figure 10).



```
{
  username: meredithl,
  password: **
}
{
  username: alxl337,
  password: ******
}
{
  username: br,
  password: ******
}
{
  username: port_able,
  password: ******
}
```

Figure 8


Medium Article Claps KTable
Article: "Thorough Introduction to Apache Kafka" Claps: 30
Article: "Crystal - the Ruby You've Never Heard Of" Claps: 15
Article: "Crystal - the Ruby You've Never Heard Of" Claps: 7
Article: "Thorough Introduction to Apache Kafka" Claps: 19
Article: "From Zero to Hero (How I became A professional developer in a year)" Claps: 1

Article Name	Clap Count
Thorough Introduction To Apache Kafka	49
Crystal - the Ruby You've Never Heard Of	22
From Zero to Hero (How I became a professional developer in a year)	1

Figure 9

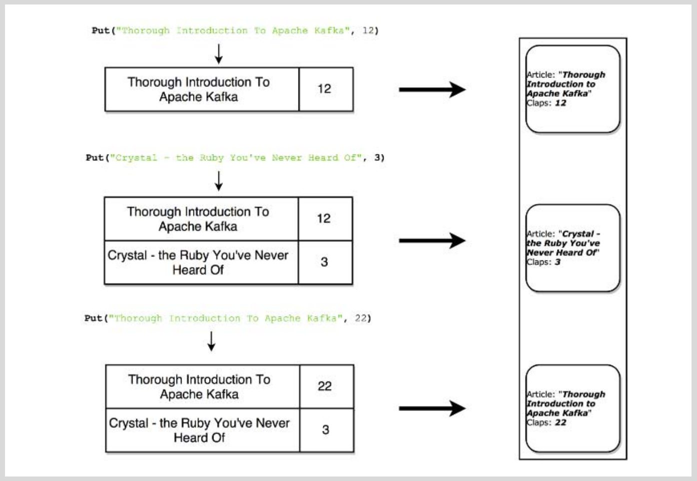

Put("Thorough Introduction To Apache Kafka", 12)
Thorough Introduction To Apache Kafka | 12
Put("Crystal - the Ruby You've Never Heard Of", 3)
Thorough Introduction To Apache Kafka | 12
Crystal - the Ruby You've Never Heard Of | 3
Put("Thorough Introduction To Apache Kafka", 22)
Thorough Introduction To Apache Kafka | 22
Crystal - the Ruby You've Never Heard Of | 3
Article: "Thorough Introduction to Apache Kafka" Claps: 12
Article: "Crystal - the Ruby You've Never Heard Of" Claps: 3
Article: "Thorough Introduction to Apache Kafka" Claps: 22

Figure 10

Stateful processing

Some simple operations, like `map()` or `filter()`, are stateless and don't require you to keep any data regarding the processing. However, in real life, most operations you'll do will be stateful (e.g., `count()`), and as such, will require you to store the currently accumulated state.

The problem with maintaining the state on stream processors is the stream processors can fail! Where would you need to keep this state in order to make it fault-tolerant?

A naive approach is to simply store all states in a remote database and join over the network to that store. The problem with this is there's no locality of data and lots of network round trips, both of which will significantly slow down your application.

A more subtle but important problem is your stream processing job's uptime would be tightly coupled to the remote database and the job won't be self-contained (a change in the database from another team might break your processing).

So what's a better approach?

Recall the duality of tables and streams. This allows us to convert streams into tables that are colocated with our processing. It also provides us with a mechanism for handling fault tolerance – by storing the streams in a Kafka broker.

A stream processor can keep its state in a local table (e.g., RocksDB), which will be updated from an input stream (after perhaps some arbitrary transformation). When the process fails, it can restore its data by replaying the stream.

You could even have a remote database be the producer of the stream, effectively broadcasting a changelog with which you rebuild the table locally. Figure 11 (overleaf) illustrates stateful processing – joining a KStream with a KTable.

ksqlDB

Normally, you'd be forced to write your stream processing in a JVM language, as that's where the only official Kafka Streams API client is. Figure 12 shows a sample ksqlDB setup.

Released in April 2018 [Confluent-1], ksqlDB [Apache-1] is a feature that allows you to write your simple streaming jobs in a familiar SQL-like language.

You set up a ksqlDB server and interactively query it through a CLI [Wikipedia] to manage the processing. It works with the same abstractions (KStream and KTable), guarantees the same benefits of the Streams API (scalability, fault tolerance), and greatly simplifies work with streams.

This might not sound like a lot, but in practice, it's way more useful for testing out stuff and even allows people outside of development (e.g., product owners) to play around with stream processing. I encourage you to take a look at the quick-start video and see how simple it is [Confluent-2].

Streaming alternatives

Kafka streams are a perfect mix of power and simplicity. They arguably have the best capabilities for stream jobs on the market, and they integrate with Kafka way easier than other stream-processing alternatives (Storm, Samza, Spark, Wallaroo).

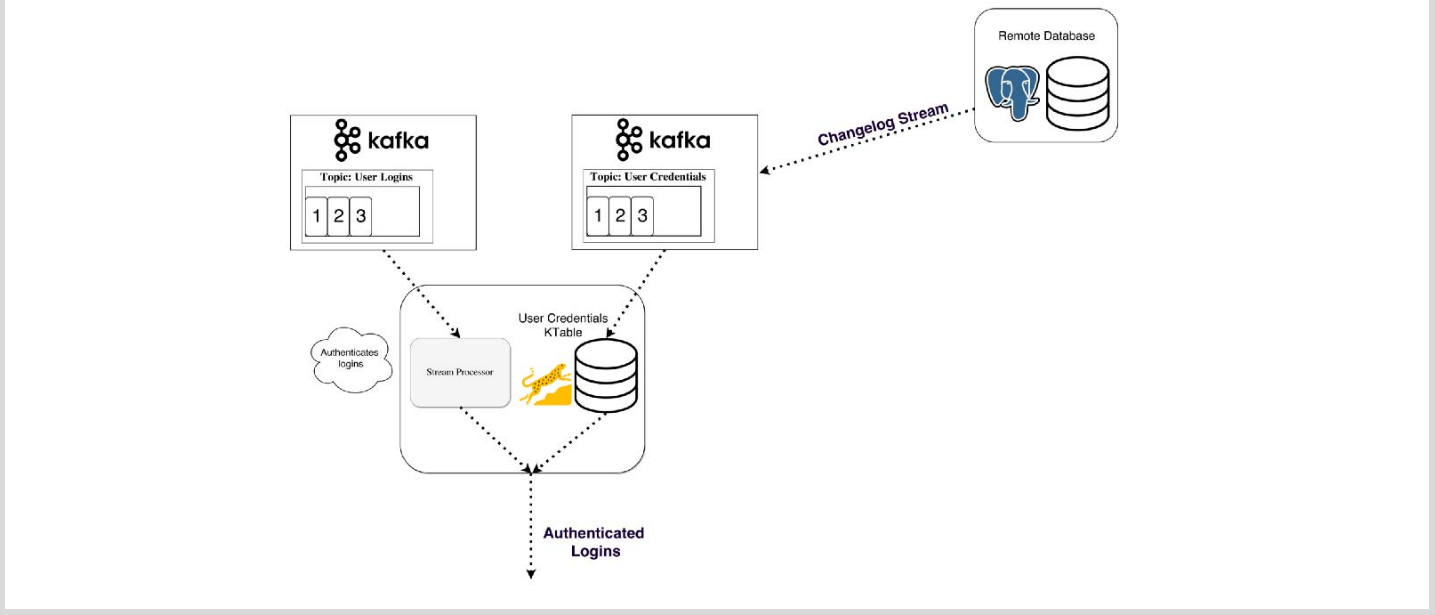


Figure 11

The problem with most other stream-processing frameworks is they're complex to work with and deploy. A batch-processing framework like Spark needs to:

- Control a large number of jobs over a pool of machines and efficiently distribute them across the cluster
- To achieve this, it has to dynamically package up your code and physically deploy it to the nodes that'll execute it (along with the configuration, libraries, etc.)

Unfortunately, tackling these problems makes the frameworks pretty invasive. They want to control many aspects of how code is deployed, configured, monitored, and packaged.

Kafka Streams let you roll out your own deployment strategy when you need it, be it Kubernetes, Mesos, Nomad, Docker Swarm, or others.

The underlying motivation of Kafka Streams is to enable all your applications to do stream processing without the operational complexity of running and maintaining yet another cluster. The only potential downside is that it's tightly coupled with Kafka, but in the modern world, where most if not all real-time processing is powered by Kafka, that may not be a big disadvantage.

When would you use Kafka?

As we already covered, Kafka allows you to have a huge amount of messages go through a centralized medium and to store them without worrying about things like performance or data loss.

This means it's perfect for use as the heart of your system's architecture, acting as a centralized medium that connects different applications. Kafka can be the center piece of an event-driven architecture and allows you to truly decouple applications from one another. See Figure 13.

Kafka allows you to easily decouple communication between different (micro)services. With the Streams API, it's now easier than ever to write business logic that enriches Kafka topic data for service consumption. The possibilities are huge, and I urge you to explore how companies are using Kafka.

Why has it seen so much use?

High performance, availability, and scalability alone aren't strong enough reasons for a company to adopt a new technology. There are other systems that boast similar properties, but none have become so widely used. Why is that?

The reason Kafka has grown in popularity (and continues to do so) is because of one key thing – businesses nowadays benefit greatly from event-driven architecture. This is because the world has changed – an enormous (and ever-growing) amount of data is being produced and

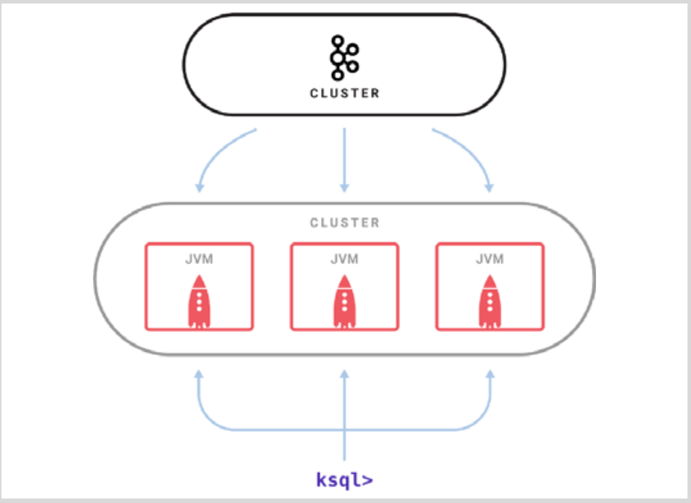


Figure 12

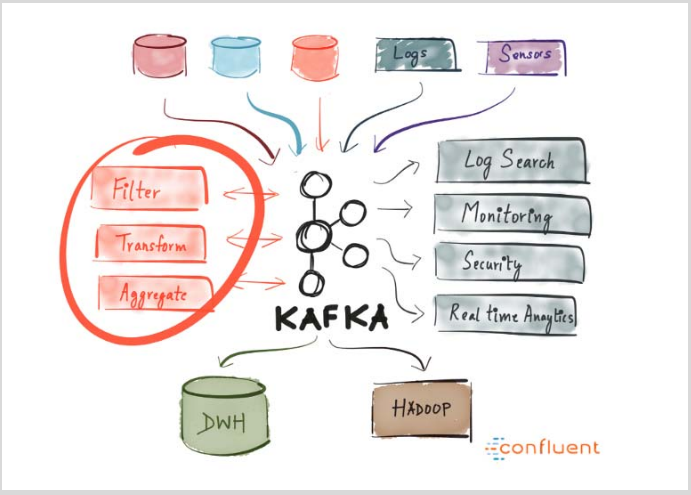


Figure 13

consumed by many different services (internet of things, machine learning, mobile, microservices).

A single real-time event-broadcasting platform with durable storage is the cleanest way to achieve such an architecture. Imagine what kind of a mess it'd be if streaming data to/from each service used a different technology specifically catered to it.

This, paired with the fact that Kafka provides the appropriate characteristics for such a generalized system (durable storage, event broadcast, table and stream primitives, abstraction via ksqlDB, open source, actively developed) make it an obvious choice for companies.

Summary

Apache Kafka is a distributed streaming platform capable of handling trillions of events a day. Kafka provides low-latency, high-throughput, fault-tolerant publish and subscribe pipelines and is able to process streams of events.

We went over its basic semantics (producer, broker, consumer, topic), learned about some of its optimizations (pagecache), learned how it's fault-tolerant by replicating data, and were introduced to its ever-growing powerful streaming abilities.

Kafka has seen large adoption at thousands of companies worldwide, including a third of the Fortune 500. With the active development of Kafka and the recently released first major version 1.0 [Confluent-3], there are predictions that this streaming platform is going to be as big and central of a data platform as relational databases are.

I hope this introduction helped familiarize you with Apache Kafka and its potential. ■

Further reading, resources, references and things I didn't mention

The rabbit hole goes deeper than this article was able to cover. Here are some features I didn't get the chance to mention but are nevertheless important to know:

- Controller broker, in-sync replicas: The way in which Kafka keeps the cluster healthy and ensures adequate consistency and durability [Kozlovski18]
- Connector API: API helping you connect various services to Kafka as a source or sink (PostgreSQL, Redis, Elasticsearch) [Confluent-4]
- Log compaction: An optimization which reduces log size. Extremely useful in changelog streams [Cloudurable].
- Exactly once message semantics: Guarantee that messages are received exactly once. This is a big deal, as it's difficult to achieve [Confluent-5].

Resources

'Kafka Acks Explained': A short article of mine explaining the commonly confused **`acks`** and **`min.isr`** settings, at https://medium.com/better-programming/kafka-acks-explained-c0515b3b707e

'Kafka Needs No Keeper': A great talk by Colin McCabe about how Apache Kafka is implementing its own consensus algorithm for metadata based on Raft at https://www.infoq.com/presentations/kafka-zookeeper/

Confluent blog: A wealth of information regarding Apache Kafka at https://www.confluent.io/blog/

Kafka documentation: Great, extensive, high-quality documentation at https://kafka.apache.org/documentation/

Kafka Summit 2017 videos at https://www.confluent.io/resources/kafka-summit-san-francisco-2017/

References

[Apache-1] ksqlDB: https://ksqldb.io

[Apache-2] 'KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum': https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum

[Cloudurable] 'Kafka Architecture: Log Compaction' posted 18 May 2017 at http://cloudurable.com/blog/kafka-architecture-log-compaction/index.html

[Confluent-1] 'Confluent Platform 4.1 with Production-Ready KSQL Now Available' posted 17 April 2018 at https://www.confluent.io/blog/confluent-platform-4-1-with-production-ready-ksql-now-available/

[Confluent-2] 'Developer Preview: KSQL from Confluent', uploaded 28 August 2017 at https://www.confluent.io/blog/confluent-platform-4-1-with-production-ready-ksql-now-available/

[Confluent-3] 'Apache Kafka Goes 1.0', uploaded 1 November 2017 at https://www.confluent.io/blog/apache-kafka-goes-1-0/

[Confluent-4] 'Announcing Kafka Connect: Building large-scale low-latency data pipelines' posted 18 February 2016 at https://www.confluent.io/blog/announcing-kafka-connect-building-large-scale-low-latency-data-pipelines/

[Confluent-5] 'Exactly-Once Semantics Are Possible: Here's How Kafka Does It', posted 30 June 2017 at https://www.confluent.io/blog/exactly-once-semantics-are-possible-heres-how-apache-kafka-does-it/

[Fowler05] Martin Fowler, 'Event Sourcing', posted 12 December 2005 at https://martinfowler.com/eaaDev/EventSourcing.html

[GitHub] 'RocksDB Basics' at https://github.com/facebook/rocksdb/wiki/rocksdb-basics

[KafkaAPI] 'Kafka Streams' at https://kafka.apache.org/documentation/streams/

[Kozlovski18] 'Apache Kafka's Distributed Systems Firefighter – the Controller Broker' (covers how coordination between the brokers works and much more) at https://medium.com/@stanislavkozlovski/apache-kafkas-distributed-system-firefighter-the-controller-broker-1afca1eae302

[Wikipedia] 'Command-line interface' at https://en.wikipedia.org/wiki/Command-line_interface

Acknowledgements

Some of the images in this article are from the Kafka blog.

Figure 13 is reproduced courtesy of Confluent.

This article was first published on Stanislav's blog on 15 December 2017: https://medium.com/better-programming/thorough-introduction-to-apache-kafka-6fbf2989bbc1

It has been lightly edited to incorporate new advancements in this fast-moving technology.

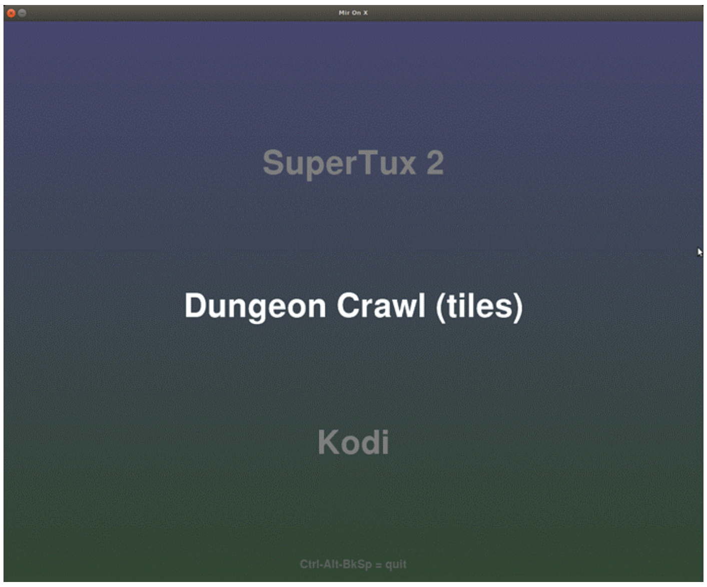
An Example Confined User Shell

Snap confinement is not just for individual apps but can be applied to a complete GUI environment. Alan Griffiths demonstrates with the Mircade shell.

In *Overload* #155 [Griffiths20], I wrote about snap confinement and why confinement is important in the current computing landscape. Having the operating system enforce limits on the things programs can do allows them to be installed and used with confidence.

This time I'm going to describe another way of employing snaps that provides a bespoke confined environment that can be deployed a range of target platforms from embedded IoT devices to all the major Linux distros.

There are various scenarios and reasons for packaging a Snap confined shell and a selection of applications together in a confined environment. You might have applications that work well together for a particular task; or, you may want to offer a number of alternative applications and have them available on a wide range of target platforms. The Mircade snap illustrates this approach.



Contents of the Mircade snap

The contents of the Mircade snap are determined by a `snapcraft.yaml` packaging script. I won't go into detail about how the packaging tools work, as there's plenty of online material that covers this well. But I will show a few extracts that illustrate how things are put together.

The user shell

A user shell is a program that allows the user to interact with the computer. It could be as simple as a command-line shell or as complex as a full desktop environment.

Alan Griffiths Alan has delivered working software and development processes to a range of organizations, written for a number of magazines, spoken at several conferences, and made many friends. He can be contacted at alan@octopull.co.uk

```
egmde:
  source: https://github.com/AlanGriffiths/
egmde.git
  source-branch: mircade
  plugin: cmake-with-ppa
  ppa: mir-team/release
  build-packages:
    - pkg-config
    - libmiral-dev
    - libboost-filesystem-dev
    - libfreetype6-dev
    - libwayland-dev
    - libxkbcommon-dev
    - g++
  stage-packages:
    - try: [libmiral4]
    - else: [libmiral3]
    - mir-graphics-drivers-desktop
    - fonts-freefont-ttf
  stage:
    - -usr/share/wayland-sessions/egmde.desktop
    - -bin/egmde-launch
```

Listing 1

For Mircade, I use a modified example Mir shell (egmde) I've presented in my writing for *C Vu* [Griffiths]. This 'mircade' branch of egmde allows the user to select one of a number of programs and run it all within the Snap confined environment (see Listing 1).

If you're emulating this approach you don't have to use egmde, or even a Mir based shell, but doing so ensures there are no unexpected issues to resolve.

The applications

A successful 'bundled' snap is really down to choosing a compelling set of applications.

I've taken a bunch of games from the Ubuntu archive and bundled them into the snap. That choice is only an illustration, there's no need to choose games, or programs from the archive.

```
neverball:
  plugin: nil
  stage-packages:
    - neverball
```

In this example, most of the applications use SDL2 and all use Wayland.

```
sdl2:
  plugin: nil
  stage-packages:
    - libsdl2-2.0-0
    - libsdl2-image-2.0-0
    - libsdl2-mixer-2.0-0
    - libsdl2-net-2.0-0
```

it is possible to take some applications, a user shell and Snap technology and use them deliver a portable, secure package to multiple Linux platforms

I've not covered other toolkits in the Mirade example. In spite of this, applications based on GTK, Qt and X11 can also be packaged. (X11 support does require Mir 2.0 which had not been released at the time of writing.)

The target platforms

Running on Ubuntu Core

There are a lot of advantages to running Ubuntu Core on IoT devices, and Mircade shows how a bundle of applications can be delivered for this. When installed on Ubuntu Core, Mircade connects to a Wayland server (such as mir-kiosk).

Running on Classic Linux

On Ubuntu Classic, there are four ways that Mircade can run, the first three are:

1. Connecting to an X11 compositor as a window on a traditional desktop
2. Connecting to a Wayland compositor as a full-screen window on a traditional desktop
3. Running directly on the hardware as a graphical login session

For each of these the corresponding interface needs to be connected:

- Connecting to an X11 compositor:
  `snap connect mircade:x11`
- Connecting to a Wayland compositor:
  `snap connect mircade:wayland`
- Running directly on the hardware:
  `snap connect mircade:login-session-control`

The fourth option, typically on an Ubuntu Server installation, is to run in the same way as on Ubuntu Core using a **`mir-kiosk`** daemon as to access the hardware.

Conclusion

The Mircade snap confined shell demonstrates how it is possible to take some applications, a user shell and Snap technology and use them deliver a portable, secure package to multiple Linux platforms including Ubuntu Core, Ubuntu Desktop and many other distros.

Targeting multiple platforms is important to the developers of snaps and confinement is important as users of a snap can ensure that it has limited access to their computer and what they are doing with it.

Do you have, or know of, a set of applications that would benefit from this approach? ■

References

[Griffiths] Some of the relevant articles (available online to members) are:

- 'Making a Linux Desktop' in *CVu* 31.4, available at: https://accu.org/index.php/journals/2696
- 'Making a Linux Desktop: Painting Some Wallpaper' in *CVu* 31.5, available at: https://accu.org/index.php/journals/2714
- 'Making a Linux Desktop – Launching Applications' in *CVu* 32.1, available at: https://accu.org/index.php/journals/2761

[Griffiths20] Alan Griffiths (2020) 'What does 'app confinement' mean?' in *Overload* 155, available at: https://accu.org/index.php/journals/2747

Resources

Mircade on GitHub: https://github.com/MirServer/mircade

Egmde on GitHub: https://github.com/AlanGriffiths/egmde/

The Mir display server: https://mir-server.io/

Agile Manchester 2020: Testers Edition

Many tech conferences are still going ahead online. Jit Gosai reports on Agile Manchester from a tester's perspective.

I've always found tester representation at agile conferences to be lacking. It's a bit like it doesn't have the word test in the titles, so it's not for me. Personally, I've always found a treasure trove of information from talks that are directly or indirectly related to software testing. Remember, testing doesn't always look like testing: you sometimes need to change the frame with which you look at things to get the best out of them.

Below are the talks that I attended with a brief summary of the talk and what it could mean for testers. If you want the full unadulterated notes then see my personal notes (follow the link on the original blog).

Fighting Code Rot with Continuous Improvement

by @garyfleming Slides http://bit.ly/fight-code-rot

FIGHT CODE ROT USING CONTINUOUS IMPROVEMENT
@GARYFLEMING

@garyfleming

Summary of the talk

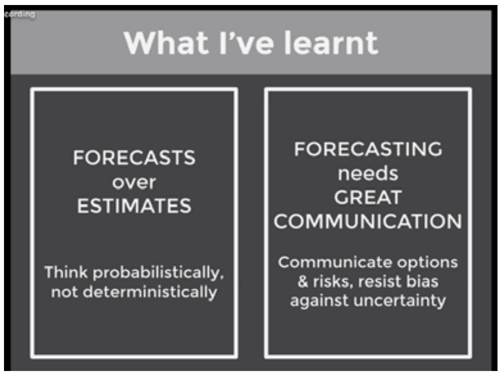
Good talk covering all the basic with keeping your system update and why. Well delivered and really useful for less experienced team members and a good recap for 'they should know better' members.

For testers

For testers, understanding what needs to be updated when can help them understand how that change could affect end users. Be proactive what do the release notes say for X, how do we use system Y. Building this knowledge takes time but can be really valuable in the long run. Start small and work your way up. Developers can help you but try and help by having specific questions for them.

Agile metrics for predicting the future

by Mattia Battiston Slides: https://www.slideshare.net/mattiabattiston/agile-metrics-for-predicting-the-future


What I've learnt

FORECASTS over ESTIMATES

Think probabilistically, not deterministically

FORECASTING needs GREAT COMMUNICATION

Communicate options & risks, resist bias against uncertainty

Summary of the talk

Forecasts will always beat estimates for non-deterministic projects (think all software projects). As they help you understand what could happen with a confidence rating. You've probably already got most of this data but knowing your lead times and throughput can help with this and some spreadsheets.

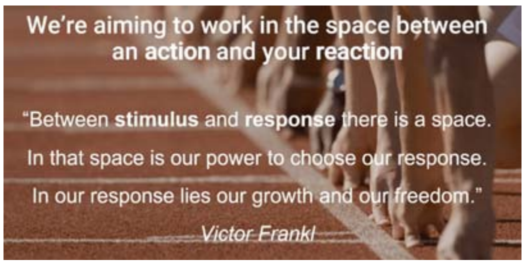
The key thing to remember is you need to talk to your stakeholders and make sure they understand what the numbers mean and how it affects them. Don't just give them the spreadsheets and expect them to understand.

For testers

If quality means value to someone then value to people working in delivery roles is greater predictability with delivering our software systems. Understanding what these metrics are, how they are used and what affects them will not only enable you to have more productive conversation with delivery but also understand why they are important to that group. This will help you articulate risk better within your teams as you will be able to tailor the message to that specific audience. This will not only help you help them understand risk better but increase your value from just being the person that finds all the issues. This elevates you from being just a tester towards a test analysts.

Crucial conversations in agile teams

How making it safe to talk about almost anything unlocks continuous improvement by Chris Smith. Slides: https://www.slideshare.net/chris_smith1976/crucial-conversations-for-agile-teams-agile-manchester-virtual-may-2020


We're aiming to work in the space between an action and your reaction

"Between stimulus and response there is a space.

In that space is our power to choose our response.

In our response lies our growth and our freedom."

Victor Frankl

Jitesh Gosai (or Jit, as he's better known) has over 15 years' test experience, working with a variety of companies from mobile manufactures to OS builders and app developers. He currently works with the BBC's Mobile Platforms team. His career started in development, and he moved towards Development in Test to integrate both his passions. He can be contacted at jiteshgosai@gmail.com and can be found on Twitter (@jitgo).

Agile Manchester

Agile Manchester is a practical, hands-on Agile software development conference, usually held in the north of England. This year it was online. More details are available at https://2020.agilemanchester.net/

Summary of the talk

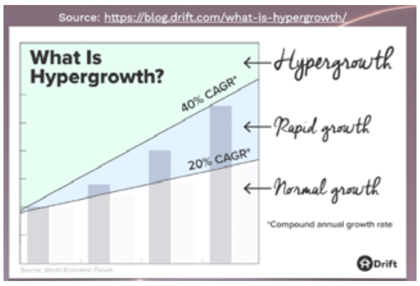
Really interesting talk by @cj_smithy at #agilemanc focusing around the book *Crucial Conversations* but bringing in ideas from the Chimp paradox, 5 dysfunctions of team, Radical Candor and generative cultures by Ron Westrum. Conversations within agile software teams are incredibly important (remember individuals and interactions over process and tools) so being better skilled at them is a great thing.

For testers

We have conversation with team members all the time. Whether that is to find out information or to inform others about what's going on, its a core part of our skill set. Getting better at communicating verbally should be part of our personally development as testers. The resources mentioned in this talk would go a long way to help you build up that skill and help keep it sharp.

Leading an agile organisation through hyper growth

By Patrick Kua



Summary of the talk

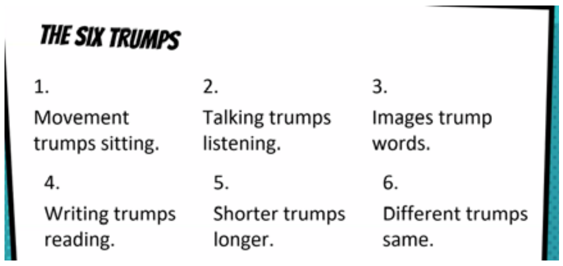
The company Patrick was a CEO of went through some really fast growth over a very short period. The model he used was simple and acknowledged that it wouldn't work forever, so they kept iterating and scaling the organisation. Useful to benchmark your company against to see where you are in the growth of your organisation.

For testers

Understanding how a company develops from startup to enterprise is really helpful in seeing what types of problems you're likely to face. This can help you help your team understand how quality is likely to be affected when scaling and what they can do to mitigate it.

Improve your agile coaching skills with Training from the BACK of the Room

By Sabine Khan



THE SIX TRUMPS

1. Movement trumps sitting.
2. Talking trumps listening.
3. Images trump words.
4. Writing trumps reading.
5. Shorter trumps longer.
6. Different trumps same.

Summary of the talk

Its called back of the room as you're not using slide decks and presenting but are getting the participants to stand and present instead. Hence coaching from the back of the room.

Really interesting approach to learning and using coaching skills in teaching others. The 6 learning principles are really quite easy to apply and can make almost any session interactive. This combined with the 4C's gives you a framework to turn any learning topics into something more than just sit and listen.

For testers

Need to help team members understand what exploratory testing is then why not do something interactive instead of just another slide deck. Teach them through doing and the key points might actually stick.

Culture + Code ≠ Delivery

By Vimla Appadoo @thatgirlvim

Summary of the talk

Vim makes a great point in that just delivering through code and a good culture isn't enough as this misses how that delivery affects users. Especially if the system has biases built in unintentionally. She says we need communications as well. Communication to be able to link together all the parts of the culture of the organisation, systems, processes and people + the code can help us deliver the right things.

For testers

Testers help raise the awareness of quality within systems, but for them to be able to do that affectively they need to take into account the team, the business and users as well. The key skill to be able to do this is communication and helping to link together all the parts to understand the other. This is not to say that testers are the key to culture but on a smaller team scale they can have huge influence over what direction that culture moves in. Do the teams care about how their systems affect users or wait to see what happens? ■

This article was first published on Jit's blog on 19 May 2020: https://www.jitgo.uk/agile-manchester-2020-testers-edition/

For further resources for testers, see: https://www.jitgo.uk/blog/

Afterwood

Mind your language! Chris Oldwood recounts a Stack Overflow comment that got him thinking.

When Stack Overflow sprang up way back in mid-2008 it seemed like a game changer for the industry. Even for someone like me who had already earned a few stripes as a programmer by struggling with the craft through the use of product documentation, journals, books, web sites, CIS & CiX forums, etc. I found much to like about the innovative format as it promised to raise the signal-to-noise ratio to much higher levels. At that time I was also making the switch from a career spent entirely in the native world to the managed one of .NET, so felt I would be in a good position to provide help to those new to the place I was leaving behind while drawing on the expertise of those already ahead of me on the path to my new destination.

Of course, I didn't reckon on the volume of people out there who also appeared eager to help out but apparently didn't need to either eat or sleep! I was never expecting to reach the dizzying heights of 'reputation' achieved by the likes of Stack Overflow's most famous respondent – Jon Skeet – but I felt chalking up a few answers would at least give me something else to raise my profile and give any potential future employers one more reason to prefer me over someone else.

It's funny how things work out. It's now over 10 years since I joined and yet I'm still to reach even the 1,000 mark. Consequently, it's rare I visit the site and get a notification saying I've earned some more points or received a comment to one of my answers. Hence when I visited recently and found a little badge on my inbox, I was expecting good news; I never for a moment expected a comment from someone suggesting that one of my answers was incredibly condescending and that I was blaming the questioner for their own predicament.

Being a programmer with a bit of mileage, I probably don't google things much less frequently than anyone else but years of skim reading articles, blog posts, and forum messages has meant that I've found it easy to skip over the 'noise' and focus on whether or not the answer is likely to be of interest. As a consequence, I've become ignorant of the kinds of problems that have caused a number of people to suggest that Q&A sites like Stack Overflow are not as welcoming or as helpful as they purport to be. Given their reliance on contributions from the general public, such as myself, that means some portion of the blame almost certainly lies with the attitudes of us programmers who have chosen to offer our time and expertise to help others out. In more recent years, I've made a conscious effort to try and learn more about the subtle ways that people denigrate others, so I was unprepared to discover that I might have ended up on the wrong side of the tracks.

It's all too easy when challenged to go on the defensive and find ways to brush off the accuser as someone who's clearly made no effort to understand what you've said and just needs to 'get over themselves'. The somewhat passive-aggressive nature of the comment didn't really encourage me to immediately take the issue seriously whereas the apparent 'injustice' did mean I felt the need to investigate further. I followed the notification, which (curiously) had taken 9 months to deliver, and re-read whatever it was I said back in 2010 that had caused offence [Oldwood10]:

> You don't mention what language you're using but there is no hidden setting that hides errors per se, other than the compiler giving up after it has encountered a gazillion issues (Visual C++). It has decided that there is so much wrong with the code that it's not going to waste even more time telling you stuff that you're just going to ignore. Classic examples of this are caused by missing braces and parenthesis, or botched #include guards etc.

My gut reaction was that '2010 me' was probably trying to be light-hearted or sarcastic but '2020 me' was somewhat appalled that I had not spotted the obvious potential for misunderstanding. But it was just a misunderstanding, right? I want to believe there was no intention of malice on my part – that I was suggesting the tool was at fault, not the operator – but I also can't deny that I haven't overused sarcasm in the past and stepped over the mark once or twice. My hope is that while '2010 me' might have been better at judging the mood in person, I clearly still had a lot to learn back then about the written word and personal responsibility.

Hemingway said that "*the only kind of writing is rewriting*" and '2020 me' is beginning to understand that more and more. While my younger self may have been satisfied to simply avoid spelling and grammatical mistakes, my older self is slowly becoming more aware of the ways in which language can be used by a writer to insult certain kinds of reader. I realize it falls on my shoulders to be wise to this and ensure that if my intent is not to be misinterpreted then I should steer clear of those devices.

I toyed briefly with deleting my answer as it turned out in the end to be totally irrelevant to the question and I'm not entirely sure what I said was even correct. Instead I decided not to hide my embarrassment and so I rewrote the answer using '2020 me' to keep the gist but hopefully place the emphasis on the limitations of the tool rather than the programmer using it. I also added a short apology too, to make it clear I was not hiding anything by rewriting it after the fact. This, I hope, also ensures the comment calling my answer out as inappropriate does not look out of place either. And, while a tiny part of me feels their tone was somewhat inappropriate too, '2020 me' accepts that negativity is not best handled with yet more negativity; instead I can thank them for giving me the catalyst to review my past attempt and revise it so that no one else has to feel uncomfortable in the future.

The question I now find myself asking is if this was an isolated moment of naivety or do I need to go back and check what else I've written in public, and if so, how far back? The answer, for me, I feel is 'yes', although I know I'm lucky enough not to have that much to look over specifically in Stack Overflow. I have since found a few of my earlier blog posts that use terms which my current self would never consider appropriate, not because they are in any way nasty, more that I think I've found better terms that I hope has made my subsequent writing more accessible.

I'm in no doubt that feedback from both friends and reviewers at ACCU over the last 10 years has had a significant impact in changing my view on the importance of the written word and this incident continues to confirm my suspicion that whether it be code or prose, it always suffers when I'm left to my own devices. ■

Reference

[Oldwood10] https://stackoverflow.com/a/3000064/106119

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80's writing assembler on 8-bit micros. These days it's enterprise grade technology from the lounge below his bedroom. With no Godmanchester duck race to commentate on this year, he's been even more easily distracted by messages to gort@cix.co.uk or @chrisoldwood

“The magazines”

The ACCU's *C Vu* and *Overload* magazines are published every two months, and contain relevant, high quality articles written by programmers for programmers.

“The conferences”

Our respected annual developers' conference is an excellent way to learn from the industry experts, and a great opportunity to meet other programmers who care about writing good code.

“The community”

The ACCU is a unique organisation, run by members for members. There are *many* ways to get involved. Active forums flow with programmer discussion. Mentored developers projects provide a place for you to learn new skills from other programmers.

“The online forums”

Our online forums provide an excellent place for discussion, to ask questions, and to meet like minded programmers. There are job posting forums, and special interest groups.

Members also have online access to the back issue library of ACCU magazines, through the ACCU web site.

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Invest in your skills. Improve your code. Share your knowledge.

Join a community of people who care about code. Join the ACCU.

Use our online registration form at **www.accu.org**.

Design: Pete Goodliffe

intel Software

PURE CODE ADRENALINE

Accelerate applications for enterprise, cloud, HPC, and AI.

Develop high-performance parallel code and accelerate workloads across enterprise, cloud, high-performance computing (HPC), and AI applications.

Amp up your code: www.qbssoftware.com