Volume 14 | Issue 4

July | August 2016

acmqueue

Association for Computing Machinery

The Power of Babble

FUNCTIONALITY AT SCALE

Scaling Synchronization in Multicore Programs

Facebook's React JavaScript Library

Complete table of contents on the following two pages

CONTENTS

JULY-AUGUST 2016

VOLUME 14 ISSUE 4
acmqueue

VOLUME 14 ISSUE 4
acmqueue

CONTENTS

JULY-AUGUST
2016

COLUMNS / DEPARTMENTS

Association for Computing Machinery

ACM, the world's largest educational and scientific computing society, delivers resources that advance computing as a science and profession. ACM provides the computing field's premier Digital Library and serves its members and the computing profession with leading-edge publications, conferences, and career resources.

acmqueue

(ISSN 1542-7730) is published bi-monthly by ACM, 2 Penn Plaza, Suite 701, New York, NY 10121-0701. USA
T (212) 869-7440
F (212) 869-0481

SUBSCRIPTIONS

A one-year subscription (six bi-monthly issues) to the digital magazine is $19.99 (free to ACM Professional Members) and available through your favorite merchant store (Mac App Store/Google play). A browser-based version is available through the acmqueue website (queue.acm.org)

SINGLE COPIES

Single copies are available through the same venues and are $6.99 each (free to ACM Professional Members).

TEXT ONLY

The Power of Babble

EXPECT TO BE CONSTANTLY AND PLEASANTLY BEFUDDLED

PAT HELLAND

Metadata defines the shape, the form, and how to understand our data. It is following the trend taken by natural languages in our increasingly interconnected world. While many concepts can be communicated using shared metadata, no one can keep up with the number of disparate new concepts needed to have a common understanding.

English is the lingua franca of the world, yet there are many facets of humanity and the concepts held by different people that simply cannot be captured in English no matter how pervasive the language. In fact, English itself has nooks, crannies, dialects, meetups, and teenager slang that innovate and extend its permutations with usages that usually do not converge. My personal idiolect shifts depending on whether I am speaking to a computer science audience, my team at work with its contextual usages, my wife, my grandkids, or the waiter at a local restaurant. Different communities of people extend English in different ways.

Computer systems have an emerging and increasing common metadata for interoperability. XML and now JSON fill similar roles by making the parsing of messages easy and common. It's great we are no longer arguing over ASCII versus EBCDIC, but that's hardly the most challenging problem of understanding.

As we move up the stack of understanding, new

It's Not Your Grandmother's Database Anymore subtleties constantly emerge. Just when we think we understand, the other guy has some crazy new ideas!

As much as we would like to have complete understanding of each other, independent innovation is far more important than crisp and clear communication. Our economic future depends on the "power of babble".

THE APOCALYPSE OF TWO ELEPHANTS

To facilitate communications, the computing industry, various companies, and other organizations try to establish standard forms of communication. We see TCP, IP, Ethernet and other communication standards as well as XML, JSON, and even ASCII making it easier to communicate. Above this, there are vertical specific standards (e.g. health care and manufacturing standards). Many companies have internal communication standards as well.

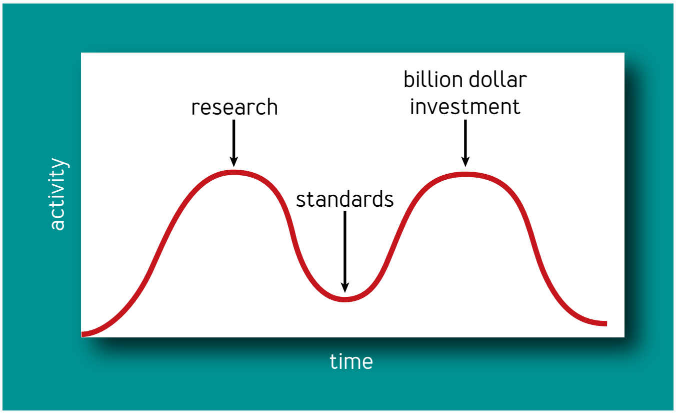
Dave Clark of MIT observed that successful standards happen only if they are lucky enough to slide into a trough of inactivity after a flurry of research and before huge investments in productization (figure 1). This observation is known as the Apocalypse of the Two Elephants (although Clark actually didn't name it that).[1]

Standards that happen in this trough are effective and experience little competition. If a standard doesn't emerge here or the trough is squished by the two humps overlapping, it's a much murkier road forward.

The best de jure standards are rubber stamps over de facto standards.

1

FIGURE 1: **THE APOCALYPSE OF TWO ELEPHANTS**



If there's no de facto standard to start from, then the de jure standard typically contains the union of all ideas discussed by the committee. Natural selection relegates these standards and their clutter to history books.

THE DIALECTS OF THE BUSINESS WORLD

Computer systems and applications tend to be developed independently to support the special needs of their users. In the past, each system would be bespoke and support detailed specifications. Increasingly, shared application platforms are leveraged, either on premises or in the cloud. In these common apps, there is common metadata—at least as far as the apps have a common heritage.

When applications are independently developed, they have disparate concepts and representations. Many of these purchased applications are designed for extensions. As the specific customer gloms extensions onto the side

Whenever there are two representations of data, either somebody adapts or the fidelity of the translation suffers.

of the app, this impacts the shape, form, and meaning of its internal and shared data.

When there's a common application lineage, there's a common understanding of its data. Popular ERP (enterprise resource planning), CRM (customer relationship management), and HRM (human resources management) applications have their ways of solving business problems, and different companies that have adopted these solutions may find it easier to interoperate.

INTERNECINE INTEROP

Still, challenges of understanding may exist even across departments or divisions of the same company. A large conglomerate may sell many products, including light bulbs, dishwashers, locomotives, and nuclear power plants. I would hazard a guess that it doesn't have a single canonical customer record type.

Of course, mergers and divestitures impact a company's metadata. I know from personal experience how hard it is to change my mailing address with a bank or insurance company. They can't seem to track down all the systems that record my address even over the course of a year. It's not a big surprise that they have a hard time managing their metadata.

WHAT'D YOU SAY?

Whenever there are two representations of data, either somebody adapts or the fidelity of the translation suffers. In many cases, the adaptation is driven by economic power. When a manufacturer wants to sell something to a huge retailer, it may be told exactly the shape, form, and semantics of the messaging between the companies. To get the business, the manufacturer will figure it out!

The dog wags the tail. In any communication partnership, the onus to adapt rests on the side that most needs the relationship to work.

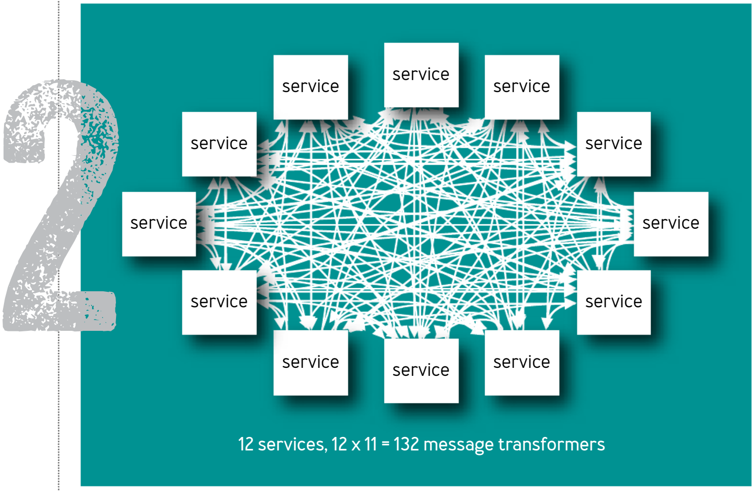
Translating between two data representations may very likely be *lossy*. Not all of the information in one form can be moved to the other form. It's highly likely that some stuff will be nulled out or possibly translated into a form that doesn't precisely map.

Each translation is lossy. By the time the translation occurs, a loss of knowledge has occurred. The best results will be from dedicated transformations designed to take exactly one source and translate it as best as possible to exactly one target. This is the least lossy form of translation. Unfortunately, this results in a boatload of translators. Creating a specific conversion for each source and destination pair results in great conversion fidelity but also results in N^2 converters (see figure 2).

What to do? Many times, we simply capture a canonical representation and do two data translations: first, a lossy translation into the canonical representation; then, a lossy translation from the canonical representation into the target representation. This is *double-lossy* and just doesn't supply as good a result.

Why do the translation to a canonical form? Because only 2N translators are needed for N sources, and that's a heck of a lot fewer than N^2, as N gets large. Using canonical metadata as a common translation reduces the number of

FIGURE 2: **LEAST-LOSSY CONVERSION**



converters but results in a double-lossy conversion (see figure 3).

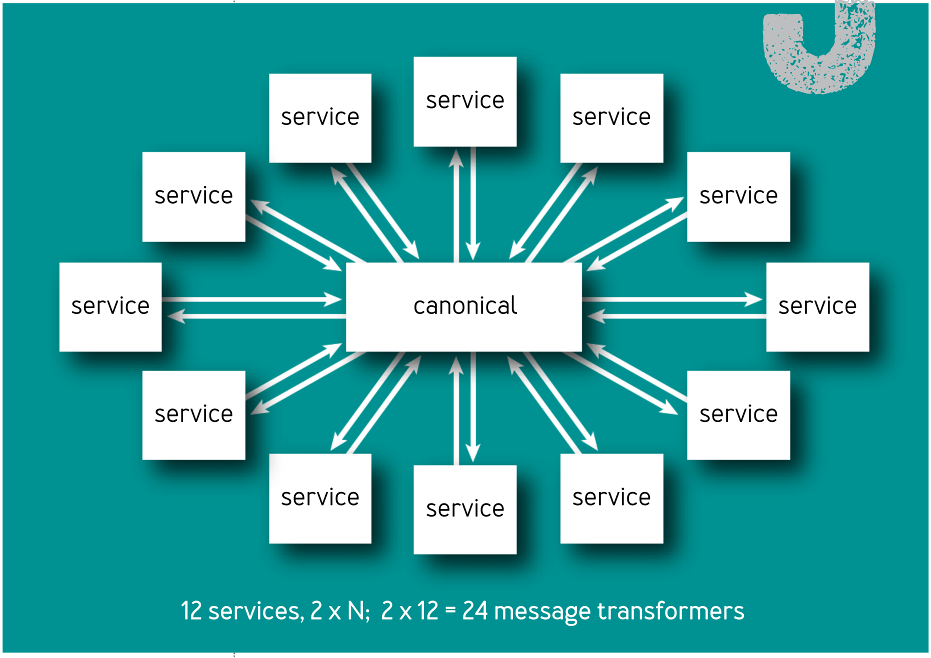
In most cases, people use canonical metadata to bound complexity but add specific source-to-target translators when the lossiness is too large.

WHAT COLOR ARE YOUR ROSE-COLORED GLASSES?

We all see stuff couched in terms of a set of assumptions. This is a worldview that allows us to interpret incoming information. This interpretation may be right or wrong, but, more importantly, it is right or wrong for our subjective usage.

Computer systems are invariably designed for a certain company, department, or group. The data is typically cast into a meaning and use that are appropriate for one side but lose their deeper meaning through the translation.

FIGURE 3: **“DOUBLE-LOSSY” CONVERSION**



Sometimes, the meaning and understanding of some data are deeply couched in cultural issues. Any translation to a new environment and culture simply loses all meaning. Reading about daily life in Medieval Europe doesn’t help much unless you study the relationships between serfs and lord as well as between men and women. Only then can you understand the actions described in the book. Similarly, in any discussion of privacy, cultural expectations must be addressed. In North America and Europe, protecting against the damage that may result by disclosing a medical challenge is paramount. In India, the essential need to vet a prospective spouse for your child is deemed more important than holding an illness private. Communication cannot take place without understanding the assumptions and interpreting through that lens.

The artificial language Esperanto was created in 1887 with the hope of achieving a common shared natural language for all people. Some folks grabbed hold and used it to write and share. Some say a few million speak it today.

The use of Esperanto has been waning, however. Each of the roughly 6,000 languages spoken by different communities in the world has its own flavor and nuance. You can say certain things in one language that you just can't say in another one.

DIVERSE AND HOMOGENEOUS

The words and phrases people use and the metadata that applications use follow a similar pattern. With a common codebase DNA and history, some meanings are the same. As time, evolution, and commingling occur, it's harder to understand one another.

New software applications either in the cloud or on premises sometimes offer enough business benefit that enterprises adapt their ways of doing business to fit the application. The new user adopts the canonical representation of data and business processes by sheer hard work. When the business value of the software is high enough, mapping to it is cost effective. Now the enterprise is much more closely aligned to the new approach and to interoperating with other enterprises sharing the new data and process.

Next, the enterprise will begin to extend the system using extensibility features. These extensions can then become a source of misunderstanding, but they bring business value to the enterprise.

The United States, Canada, and many other Western countries have tremendous diversity in their populations. New arrivals bring new customs. They work to understand the existing customs in their new home. While there are many differences at first, in a few short years the immigrants fit in. Their children are deeply ingrained in the new country, even though they still like some of that food their mom cooked at home. That food becomes as American (or English or German) as pizza, tacos, and falafel. Similarly, the base metadata continues to move and adjust as it assimilates those new messages and fields that made no sense at all a short time ago.

RELISHING DIVERSITY

While not understanding another party is a pain, it probably means that innovation and growth have occurred. Economic forces will drive when and where it's worth the bother to invest in deeper understanding.

Playing loose with understanding allows for better cohesion, as exemplified by Amazon's product catalog and the search results from Google or Bing. Remember that in many cases, cultural and contextual issues will drive how something is interpreted. Extensible data does not have a prearranged understanding. Translating between representations is lossy and frequently involves a painful tradeoff between expensive handcrafted translators and even lossier multiple translations.

Personally, as the years have gone by, I've gotten much more relaxed about the things I don't know and don't understand. A lot of stuff confuses me! As we interoperate across disparate boundaries, it would do us well to remember that the less stressed we are about perfect understanding and agreement, the better we will all get along. Moving forward, I expect to be constantly and pleasantly befuddled by the power of babble.

References

1. Clark, D. 2009. The Apocalypse of Two Elephants, or "what I really said." Advanced Network Architecture. MIT CSAIL; http://groups.csail.mit.edu/ana/People/DDC/.

Pat Helland *has been implementing transaction systems, databases, application platforms, distributed systems, fault-tolerant systems, and messaging systems since 1978. For recreation, he occasionally writes technical papers. He currently works at Salesforce.*

CONTENTS

Fresh Starts

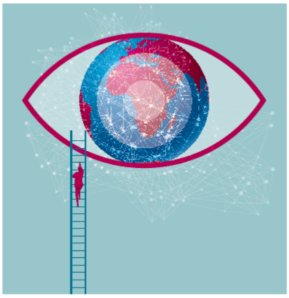
JUST BECAUSE YOU HAVE BEEN DOING IT THE SAME WAY DOESN'T MEAN YOU ARE DOING IT THE RIGHT WAY

KATE MATSUDAIRA

I love fresh starts. Growing up, one of my favorite things was starting a new school year. From the fresh school supplies (I am still a sucker for pen and paper) to the promise of a new class of students, teachers, and lessons, I couldn't wait for summer to be over and to go back to school.

The same thing happens with new jobs (and to some extent, new teams and new projects). They reinvigorate you, excite you, and get you going.

The trouble is that starting anew isn't something you get to do all the time. For some people it might happen once a year, once every two years, or once every four years. Furthermore, learning something new isn't always in the best interest of your employer. Of course, great managers want you constantly to be learning and advancing your career, but if you are doing your job well, they also probably like the idea of keeping you in that role where they can rely on you to get the work done. Putting you into a position where you will have to work hard to learn new skills isn't always best for your company—and so it probably doesn't happen often.



Wouldn't it be great if you frequently were in a position where you were pushed to grow outside of your comfort zone? Where you had to start new and fresh?

Well, the good news is that you can. In fact, you can make your current position one that focuses on your growth and extends the boundaries of your knowledge—and that is all up to you.

In technology and computer science, almost more than any other field, a growth mindset is mandatory for success. In this field the tools and best practices are constantly evolving—there is always something new to learn. For many people this high rate of change can be overwhelming, but for the right person this can mean opportunity. When you are willing to dive in and learn new skills, it puts you ahead of the game; and when you are strategic about what skills you learn, it can help you grow your career even faster.

No matter where you are in your career, there is more to learn. All of us can always use an excuse to get more invigorated and excited by our jobs. Here are three steps you can take to develop your current role and make tomorrow (or even the rest of today) a fresh start.

Create a learning plan

When you have been doing a job for a while, there isn't as much for you to learn in your day-to-day. Sure, there are always opportunities to improve little things, but your rate of knowledge acquisition slows down the longer you have been in a position. This makes it even more important to have a learning plan. You should have a list of things you plan to learn with some concrete tasks associated with each. If you need some inspiration on what should be on this list, here are some questions to ponder:

➡ To be promoted to the next level in your job, what do you need to accomplish? Are there any skills you need to acquire or improve?

➡ If you think 10 years into the future, what do you want to do? Do you know anyone doing that now? What do they know that you don't?

➡ Look back over your past performance reviews. Are there any areas where you could continue to develop and improve? If you ask others for feedback, what would they say and how can you do better?

When you have great relationships with the people you work with everyday, you tend to be happier—and you tend to be more productive and collaborative.

Build better relationships

Most of us spend more time with our coworkers than our families. When you have great relationships with the people you work with every day, you tend to be happier—and you tend to be more productive and collaborative. Also, when people like you and want to help you, then you are more likely to get promoted and discover opportunities. Here are two ideas for improving your working relationships:

➡ *Improve your communication skills.* When you get better at writing emails, or verbal presentation, you help share information, and this creates better decision-making across your whole team.

➡ *Take someone to lunch.* If you work with someone you don't know very well, or haven't had the best working relationship with, make the first move and ask this person to lunch or coffee. This is a great way to get to know people and understand their points of view. Working relationships are usually strained because two sides are making incorrect assumptions, and the first step is opening the lines of communication. Be open, practice your listening skills, and offer to foot the bill—for the cost of a lunch you would be amazed at how much that gesture can improve your work life.

Make better use of your down time

One of my favorite time-management tricks is using spare minutes to maximize your learning. When you can make the most of the small moments and learn things that help advance your career, then you will be one step ahead. This can be as simple as nixing social-media checks and replacing them with 10-15 minutes of reading articles or websites that help increase your knowledge. Here are some other ideas to get more out of those little moments:

➡ *Be on time.* When you can start on time and end on time, you make the most of meetings (plus it is a sign of respect when you show up when you say you will), and you will have more freedom to do what you want to do.

➡ *Keep a reading queue.* Whether you use bookmarks, notes, or some other tool, keep a list of items you want to read. These can be articles, whitepapers, or books—but when you have a list it is much easier just to go there to fill 15 minutes with useful learning than to spend those 15 minutes surfing the web looking for something interesting.

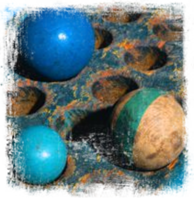
➡ *Listen to audiobooks or smart podcasts.* Whether it is on your commute or when you are working out, if you can't sit and read, try listening to your lessons. There are so many great options here, and it is a great way to maximize time and knowledge.

Of course, there are lots of other great ways to make your old career new again, but these little ideas could give you inspiration so that when you come to work tomorrow you can be excited.

If you have any other thoughts or suggestions, feel free to leave them in the comments on the website. And if there is a topic you would like to see covered in this column, let me know.

Kate Matsudaira *is an experienced technology leader. She worked in big companies such as Microsoft and Amazon and three successful startups (Decide acquired by eBay, Moz, and Delve Networks acquired by Limelight) before starting her own company, Popforms (http://popforms.com/), which was acquired by Safari Books. Having spent her early career as a software engineer, she is deeply technical and has done leading work on distributed systems, cloud computing, and mobile. She has experience managing entire product teams and research scientists, and has built her own profitable business. She is a published author, keynote speaker, and has been honored with awards such as Seattle's Top 40 under 40. She sits on the board of* acmqueue *and maintains a personal blog at katemats.com.*

CONTENTS



10 Optimizations on Linear Search

THE OPERATIONS SIDE OF THE STORY

THOMAS A. LIMONCELLI

A friend was asked the following question during a job interview: What is the fastest algorithm to find the largest number in an unsorted array?

The catch, of course, is that the data is unsorted. Because of that, each item must be examined; thus, the best algorithm would require `O(N)` comparisons, where N is the number of elements. Any computer scientist knows this. For that reason, the fastest algorithm will be a linear search through the list.

End of story.

All the computer scientists may leave the room now.

(looks around)

Are all the computer scientists gone? Good!

Now let's talk about the operational answer to this question.

System administrators (DevOps engineers or SREs or whatever your title) must deal with the operational aspects of computation, not just the theoretical aspects. Operations is where the rubber hits the road. As a result, operations people see things from a different perspective and can realize opportunities outside of the basic `O()` analysis.

Let's look at the operational aspects of the problem of trying to improve something that is theoretically optimal already.

1. DON'T OPTIMIZE CODE THAT IS FAST ENOUGH

The first optimization comes from deciding to optimize time and not the algorithm itself. First, ask whether the code is fast enough already. If it is, you can optimize your time by not optimizing this code at all. This requires a definition of *fast enough*.

Suppose 200 ms and under is *fast enough*. Anything that takes less than 200 ms is perceived to be instantaneous by the human brain. Therefore, any algorithm that can complete the task in less than 200 ms is usually good enough for interactive software.

Donald Knuth famously wrote that premature optimization is the root of all evil. Optimized solutions are usually more complex than the solutions they replace; therefore, you risk introducing bugs into the system. A bird in hand is worth two in the bush. Why add complexity when you don't have to?

My biggest concern with premature optimization is that it is a distraction from other, more important work. Your time is precious and finite. Time spent on a premature optimization is time that could be spent on more important work.

Prioritizing your work is not about deciding in what order you will do the items on your to-do list. Rather, it is deciding which items on your to-do list will be intentionally dropped on the floor. I have 100 things I would like to do this week. I am going to complete only about 10 of them. How I prioritize my work determines which 90 tasks won't get done. I repeat this process every week. One of the best time-management skills you can develop is to learn to let go of that 90 percent.

The ability to use rough estimates to decide whether or not an engineering task is worthwhile may be one of the most important tools in a system administrator's toolbox.

In the case of the interview question, whether optimizing is worthwhile relates to the number of data items. It isn't worth optimizing if only a small amount of data is involved. I imagine that if, during the interview, my friend had asked, "How many elements in the list?" the interviewer would have told him that it doesn't matter. From a theoretical point of view, it doesn't; from an operational point of view, however, it makes *all* the difference.

Deciding if an optimization is worth your time requires a quick back-of-the-envelope estimate to determine what kinds of improvements are possible, how long they might take to be achieved, and if the optimization will result in a return on investment. The ability to use rough estimates to decide whether or not an engineering task is worthwhile may be one of the most important tools in a system administrator's toolbox.

If *small* is defined to mean any amount of data that can be processed in under 200 ms, then you would be surprised at how big *small* can be.

I conducted some simple benchmarks in Go to find how much data can be processed in 200 ms. A linear search can scan 13 million elements in less than 200 ms on a three-year-old MacBook laptop, and 13 million is no small feat.

This linear search might be buggy, however. It is five lines long and, not to brag, but I can pack a lot of bugs into five lines. What if I were to leverage code that has been heavily tested instead? Most languages have a built-in sort function that has been tested far more than any code I've ever written. I could find the max by sorting the list and picking the last element. That would be lazy and execute more slowly than a linear search, but it would be very reliable. A few simple benchmarks found that on the same old laptop, this "lazy algorithm" could sort 700,000 elements and still be under the 200-ms mark.

What about smaller values of N?

If N=16,000, then the entire dataset fits in the L1 cache of the CPU, assuming the CPU was made in this decade. This means the CPU can scan the data so fast it will make your hair flip. If N=64,000, then the data will fit in a modern L2 cache, and your hair may still do interesting things. If the computer wasn't made in this decade, I would recommend that my friend reconsider working for this company.

If N is less than 100, then the lazy algorithm runs imperceptibly fast. In fact, you could repeat the search on demand rather than storing the value, and unless you were running the algorithm thousands of times, the perceived time would be negligible.

The algorithms mentioned so far are satisfactory until N=700,000 if we are lazy and N=13,000,000 if we aren't; 13 million 32-bit integers (about 52 MB) is hardly small by some standards. Yet, in terms of human perception, it can be searched instantly.

If my friend had known these benchmark numbers, he could have had some fun during the interview, asking the interviewer to suggest a large value of N, and replying, "What? I don't get out of bed for less than 13 million integers!" (Of course, this would probably have cost him the job.)

2. USE SIMD INSTRUCTIONS

Most modern CPUs have SIMD (single instruction, multiple data) instructions that let you repeat the same operation over a large swath of memory. They are able to do this very quickly because they benefit from more efficient memory access and parallel operations.

According to one simple benchmark (http://stackoverflow.com/a/2743040/71978), a 2.67-GHz Core i7 saw a 7-8x improvement by using SIMD instructions where N = 100,000. If the amount of data exceeded the CPU's cache size, the benefit dropped to 3.5x.

With SIMD, *small* becomes about 45 million elements, or about 180 MB.

3. WORK IN PARALLEL

Even if N is larger than the *small* quantity, you can keep within your 200-ms time budget by using multiple CPUs. Each CPU core can search a shard of the data. With four CPU cores, *small* becomes 4N, or nearly 200 million items.

When I was in college, the study of parallel programming was hypothetical because we didn't have access to computers with more than one CPU. In fact, I didn't think I would ever be lucky enough to access a machine with such a fancy architecture. Boy, was I wrong! Now I have a phone with eight CPU cores, one of which, I believe, is dedicated exclusively to crushing candy.

Parallel processing is now the norm, not the exception. Code should be written to take advantage of this.

4. HIDE CALCULATION IN ANOTHER FUNCTION

The search for the max value can be hidden in other work. For example, earlier in the process the data is loaded into memory. Why not have that code also track the max value as it iterates through the data? If the data is being loaded from disk, the time spent waiting for I/O will dominate, and the additional comparison will be, essentially, free.

If the data is being read from a text file, the work to convert ASCII digits to 32-bit integers is considerably more than tracking the largest value seen so far. Adding max-value tracking would be "error in the noise" of any benchmarks. Therefore, it is essentially free.

You might point out that this violates the SoC (separation of concerns) principle. The method that loads data from the file should just load data from a file. Nothing else. Having it also track the maximum value along the way adds complexity. True, but we've already decided that the added complexity is worth the benefit.

Where will this end? If the `LoadDataFromFile()` method also calculates the max value, what's to stop us from adding other calculations? Should it also calculate the min, count, total and average? Obviously not. If you have the count and total, then you can calculate the average yourself.

5. MAINTAIN THE MAX ALONG THE WAY

What if the max value cannot be tracked as part of loading the dataset? Perhaps you don't control the method that loads the data. If you are using an off-the-shelf JSON (JavaScript Object Notation) parser, adding the ability to track the max value would be very difficult. Perhaps the data is modified after being loaded, or it is generated in place.

In such situations I would ask why the data structure holding the data isn't doing the tracking itself. If data is only added, never removed or changed, the data structure can easily track the largest value seen so far. The need for a linear search has been avoided altogether.

If items are being removed and changed, more sophisticated data structures are required. A heap makes the highest value accessible in `O(1)` time. The data can be kept in the original order but in a heap or other index on the side. You will then always have fast access to the highest value, though you will suffer from additional overhead maintaining the indexes.

6. HIDE LONG CALCULATIONS FROM USERS

Maybe the process can't be made any faster, but the delay can be hidden from the user.

One good place to hide the calculation is when waiting for user input. You don't need the entire processing power of the computer to ask "Are you sure?" and then wait for a response. Instead, you can use that time to perform calculations, and no one will be the wiser.

One video-game console manufacturer requires games to have some kind of user interaction within a few seconds of starting. Sadly, most games need more time than that to load and initialize. To meet the vendor's requirement, most games first load and display a title screen, then ask users to click a button to start the game. What users don't realize is that while they are sitting in awe of the amazing title screen, the game is finishing its preparations.

Many optimizations come from end-to-end thinking. Rather than optimizing the code itself, we should look at the entire system for inspiration.

GET OUT OF YOUR SILO

Before discussing the remaining optimizations, let's discuss the value of thinking more globally about the problem. Many optimizations come from end-to-end thinking. Rather than optimizing the code itself, we should look at the entire system for inspiration.

To do this requires something scary: talking to people. Now, I understand that a lot of us go into this business because we like machines more than people, but the reality is that operations is a team sport.

Sadly, often the operations team is put in a silo, expected to work issues out on their own without the benefit of talking to the people who created the system. This stems from the days when one company created software and sold it on floppy disk. The operations people were in a different silo from the developers because they were literally in a different company. System administrators' only access to developers at the other company was through customer support, whose job it was to insulate developers from talking to customers directly. If that ever did happen, it was called an *escalation*, an industry term that means that a customer accidentally got the support he or she paid for. It is something that the software industry tries to prevent at all costs.

Most (or at least a growing proportion of) IT operations, however, deal with software that is developed in-house. In that situation there is very little excuse to have developers and operations in separate silos. In fact, they should talk to each other and collaborate. There should be a name for this kind of collaboration between developers and operations... and there is: DevOps.

If your developers and operations teams are still siloed away from each other, then your business model hasn't changed since software was sold on floppy disk. This is ironic since your company probably didn't exist when floppy disks were in use. What's wrong with this picture?

Get out of your silo and talk to people. Take a walk down the hallway and introduce yourself to the developers in your company. Have lunch with them. Indulge in your favorite after-work beverage together. If you are a manager who requires operations and developers to communicate only through "proper channels" involving committees and product management chains, get out of their way.

Once operations has forged a relationship with developers, it is easier to ask important questions, such as How is the data used? What is it needed for and why?

This kind of social collaboration is required to develop the end-to-end thinking that makes it possible to optimize code, processes, and organizations. Every system has a bottleneck. If you optimize upstream of the bottleneck, you are simply increasing the size of the backlog waiting at the bottleneck. If you optimize downstream of the bottleneck, you are adding capacity to part of a system that is starved for work. If you stay within your silo, you'll never know enough to identify the actual bottleneck.

Getting out of your silo opens the door to optimizations such as our last four examples.

7. USE A "GOOD ENOUGH" VALUE INSTEAD

Is the maximum value specifically needed, or is an estimate good enough?

Perhaps the calculation can be avoided entirely.

Often an estimate is sufficient, and there are many creative ways to calculate one. Perhaps the max value from the previous dataset is good enough.

Perhaps the max value is being used to preallocate memory or other resources. Does this process really need to be fine-tuned every time the program runs? Might it be sufficient to adjust the allocations only occasionally—perhaps in response to resource monitoring or performance statistics?

If you are dealing with a small amount of data (using the earlier definition of *small*), perhaps preallocating resources is overkill. If you are dealing with large amounts of data, perhaps preallocating resources is unsustainable and needs to be reengineered before it becomes dangerous.

8. SEEK INSPIRATION FROM THE UPSTREAM PROCESSES

Sometimes we can get a different perspective by examining the inputs.

Where is the data coming from?

I once observed a situation where a developer was complaining that an operation was very slow. His solution was to demand a faster machine. The sysadmin who investigated the issue found that the code was downloading millions of data points from a database on another continent. The network between the two hosts was very slow. A faster computer would not improve performance.

The solution, however, was not to build a faster network, either. Instead, we moved the calculation to be closer to the data. Rather than download the data and do the calculation, the sysadmin recommended changing the SQL query to perform the calculation at the database server. Instead of downloading millions of data points, now we were downloading the single answer.

This solution seems obvious but eluded the otherwise smart developer. How did that happen? Originally, the data was downloaded because it was processed and manipulated many different ways for many different purposes. Over time, however, these other purposes were eliminated until only one purpose remained. In this case the issue was not calculating the max value, but simply counting the number of data points, which SQL is very good at doing for you.

9. SEEK INSPIRATION FROM THE DOWNSTREAM PROCESSES

Another solution is to look at what is done with the data later in the process. Does some other processing step sort the data? If so, the max value doesn't need to be calculated. You can simply sort the data earlier in the process and take the last value.

You wouldn't know this was possible unless you took the time to talk with people and understand the end-to-end flow of the system.

Once I was on a project where data flowed through five different stages, controlled by five different teams. Each stage took the original data and sorted it. The data didn't change between stages, but each team made a private copy of the entire dataset so they could sort it. Because they hadn't looked outside their silos, they didn't realize how much wasted effort this entailed.

By sorting the data earlier in the flow, the entire process became much faster. One sort is faster than five.

10. QUESTION THE QUESTION

When preparing this column I walked around the New York office of stackoverflow.com and asked my coworkers if they had ever been in a situation where calculating the max value was a bottleneck worth optimizing.

The answer I got was a resounding no.

One developer pointed out that calculating the max is usually something done infrequently, often once per program run. Optimization effort should be spent on tasks done many times.

A developer with a statistics background stated that the max is useless. For most datasets it is an outlier and should be ignored. What *are* useful to him are the top N items, which presents an entirely different algorithmic challenge.

Another developer pointed out that anyone dealing with large amounts of data usually stores it in a database, and databases can find the max value very efficiently. In fact, he asserted, maintaining such data in a homegrown system is a waste of effort at best and negligent at worst. Thinking you can maintain a large dataset safely with homegrown databases is hubris.

Most database systems can determine the max value very quickly because of the indexes they maintain. If the system cannot, it isn't the system administrator's responsibility to rewrite the database software, but to understand the situation well enough to facilitate a discussion among the developers, vendors, and whoever else is required to find a better solution.

A better question would be to ask job candidates to describe a situation where they optimized an algorithm.

CONCLUSION: FIND ANOTHER QUESTION

This brings me to my final point. Maybe the interview question posed at the beginning of this column should be retired. It might be a good logic problem for a beginning programmer, but it is not a good question to use when interviewing system administrators because it is not a realistic situation.

A better question would be to ask job candidates to describe a situation where they optimized an algorithm. You can then listen to their story for signs of operational brilliance.

I would like to know that the candidates determined ahead of time what would be considered good enough. Did they talk with stakeholders to determine whether the improvement was needed, how much improvement was needed, and how they would know if the optimization was achieved? Did they determine how much time and money were worth expending on the optimization? Optimizations that require an infinite budget are not nearly as useful as one would think.

I would look to see if they benchmarked the system before and after, not just one or the other or not at all. I would like to see that they identified a specific problem, rather than just randomly tuning parts until they got better results. I would like to see that they determined the theoretical optimum as a yardstick against which all results were measured.

I would pay careful attention to the size of the improvement. Was the improvement measured, or did it simply "feel faster"? Did the candidates enhance performance greatly or just squeeze a few additional percentage points out of the existing system? I would be impressed if they researched academic papers to find better algorithms.

I would be most impressed, however, if they looked at the bigger picture and found a way to avoid doing the calculation entirely. In operations, often the best improvements come not from adding complexity, but by eliminating processes altogether.

Related articles

You're Doing It Wrong
Poul-Henning Kamp
http://queue.acm.org/detail.cfm?id=1814327

You Don't Know Jack about Network Performance
Kevin Fall and Steve McCanne
http://queue.acm.org/detail.cfm?id=1066069

Why Writing Your Own Search Engine is Hard
Anna Patterson
http://queue.acm.org/detail.cfm?id=988407

Thomas A. Limoncelli *is a site reliability engineer at Stack Overflow Inc. in New York City. His books include* The Practice of Cloud Administration *(http://the-cloud-book.com),* The Practice of System and Network Administration *(http://the-sysadmin-book.com), and* Time Management for System Administrators. *He blogs at EverythingSysadmin.com and tweets at @YesThatTom. He holds a B.A. in computer science from Drew University.*

CONTENTS

TEXT ONLY

Cloud Calipers

NAMING THE NEXT GENERATION AND REMEMBERING THAT THE CLOUD IS JUST OTHER PEOPLE'S COMPUTERS

who is KV?

click for video

Dear KV,

Why do so many programmers insist on numbering APIs when they version them? Is there really no better way to upgrade an API than adding a number on the end? And why are so many systems named "NG" when they're clearly just upgraded versions?

API2NG

Dear API2NG,

While software versioning has come a long way since the days when source code control was implemented by taping file names to hacky sacks in a bowl in the manager's office, and file locking was carried out by digging through said bowl looking for the file to edit, programmers' inventiveness with API names has not advanced very much. There are languages such as C++ that can handle multiple functions—wait, methods with the same names but different arguments—but these present their own problems, because now instead of a descriptive name, programmers have to look at the function arguments to know which API they're calling.

Perhaps the largest sources of numbered APIs are the base systems to which everyone programs, such as operating systems and their libraries. These are written in C, a lovely, fancy assembler that has no truck with such fancy notions as variant function signatures. Because of this limitation of the language that actually does most of the work on all of our collective behalves, C programmers add whole new APIs when they only want to create a library function or system call with different arguments.

Take, for example, the creation of a pipe, a very common operation. Once upon a time, pipes were simple and returned a new pipe to the program, but then someone wanted new features in pipes, such as making them nonblocking and making the pipe close when a new sub-program is executed. Since `pipe()` is a system call defined both by the operating system and in the Posix standard, the meaning of `pipe()` was already set in stone. In order to add a flags argument, a new pipe-like API was required, and so we got `pipe2()`. I would say something like "Ta-da!" but it's more like the sad trombone sound. Given that the system call interface is written in C, there was nothing to do but add a new call so that we could have some flags. The utter lack of naming creativity is shocking. So now there are two system calls, `pipe()` and `pipe2()`, but it could have been worse: we could have had `pipeng()`.

Perhaps the worst thing that Paramount ever did was name its *Star Trek* reboot *The Next Generation*, as this seems to have encouraged a generation of developers to name their shiny new thing, no matter what that thing is, ThingNG. Somehow, no one thinks about what the next, next version might be. Will the third version of something be ThingNGNG? If your software lasts a decade, will it eventually be a string of NGs preceded by a name? The use of "next generation" is probably the only thing more

Programmers are a notoriously lazy lot and making them type an extra character annoys them, which is another reason that versioned APIs often end in a single digit to save typing time.

aggravating than numeric indicators of versioned APIs.

The right answer to these versioning dilemmas is to create a descriptive name for the newer interface. After all, you created the new version for a good reason, didn't you? Instead of `pipe2()`, perhaps it might have made sense to name it `pipef()` for "pipe with a flags argument." Programmers are a notoriously lazy lot and making them type an extra character annoys them, which is another reason that versioned APIs often end in a single digit to save typing time.

For the time being, we are likely to continue to have programmers who version their functions as a result of the limitations of their languages, but let's hope we can stop them naming their next generations after the next generation.

KV

Dear KV,

My team has been given the responsibility of moving some of our systems into a cloud service as a way of reducing costs. While the cloud looks cheaper, it has also turned out to be more difficult to manage and measure because many of our former performance-measuring systems depended on having more knowledge about how the hardware was performing as well as the operating system and other components. Now that all of our devices are virtual, we find that we're not quite sure we're getting what we paid for.

Cloudy with a Chance

Dear Cloudy,

Remember the cloud is just other people's computers. Virtualized systems have existed for quite a while now and are deployed for an assortment of reasons, most of which have to do with lower costs and ease of management. Of course, the question is whose management is easier. For services that are not performance-critical, it often makes good sense to move them off dedicated hardware to virtualized systems, since such systems can be easily paused and restarted without the applications knowing that they have been moved within or between data centers.

The problems with virtualized architectures appear when the applications have high demands in terms of storage or network. A virtualized disk might try to report the number of IOS (I/O operations per second), but since the underlying hardware is shared, it is difficult to determine if that number is real, consistent, and will be the same from day to day. Sizing a system for a virtualized environment runs the risk of the underlying system changing performance from day to day. While it's possible to select a virtual system of a particular size and power, there is always the risk that the underlying system will change its performance characteristics if other virtualized systems are added or if nascent services suddenly spin up in other containers. The best one can do in many of these situations is to measure operations in a more abstract way that can hopefully be measured with wall-clock time. Timestamping operations in log files ought to give some reasonable set of measures, but even here, virtualized systems can trip you up because virtual systems are pretty poor at tracking the time of day.

Working backward toward the beginning, if you want to know about performance in a virtualized system, you'll have to establish a reliable time base, probably using NTP (Network Time Protocol) or the like, and on top of that, you'll have to establish the performance of your system via logging the time that your operations require. Other tools may be available on various virtualized environments, but would you trust them? How much do you trust other people's computers?

KV

Kode Vicious, *known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. Neville-Neil is the co-author with Marshall Kirk McKusick and Robert N. M. Watson of* The Design and Implementation of the FreeBSD Operating System *(second edition). He is an avid bicyclist and traveler who currently lives in New York City.*

CONTENTS

Functional at Scale

APPLYING FUNCTIONAL PROGRAMMING PRINCIPLES TO DISTRIBUTED COMPUTING PROJECTS

MARIUS ERIKSEN, TWITTER

Modern server software is demanding to develop and operate: it must be available at all times and in all locations; it must reply within milliseconds to user requests; it must respond quickly to capacity demands; it must process a lot of data and even more traffic; it must adapt quickly to changing product needs; and in many cases it must accommodate a large engineering organization, its many engineers the proverbial cooks in a big, messy kitchen.

What's more, the best computers for these applications—whether you call them clouds, data centers, or warehouse computers—are really bad. They are complex and unreliable, and prone to partial failures. They have asynchronous interconnects and deep memory hierarchies, and leave a lot of room for operator error.[1]

Cloud computing thus forces us to confront the full complexity of distributed computing, where seemingly simple problems require complicated solutions. While much of this complexity is inherent—the very nature of the problem precludes simpler solutions—much of it is also incidental, a simple consequence of using tools unfit for the purpose.

At Twitter, we use ideas from functional programming to tackle many of the complexities of modern server software, primarily through the use of higher-order functions and effects. *Higher-order functions*, or those that return other functions, let us combine simpler functions to define more complex ones, building up application functionality in a piecemeal fashion. *Effects* are values that represent some side-effecting operation; they are used in conjunction with higher-order functions to build complex effects from simpler ones.[7]

Higher-order functions and effects help build scalable software in two ways: first, building complex software from simple parts makes it easy to understand, test, reuse, and replace individual components; second, effects make side-effecting operations tractable, promoting modularity and good separation of concerns.

This article explores three specific abstractions that follow this style of functional programming: *futures* are effects that represent asynchronous operations; *services* are functions that represent service boundaries; and *filters* are functions that encapsulate application-independent behavior. In turn, higher-order functions provide the glue used to combine these to create complex systems from simple parts.

Futures, services, and filters are thus combined to build server software in a piecemeal fashion. They let programmers build up complex software while preserving their ability to reason about the correctness of its constituent parts.

By consistently applying these principles, programmers can construct systems that are at once simpler, more flexible, and performant.

"Concurrent programs wait faster."
—Tony Hoare

CONCURRENT PROGRAMMING WITH FUTURES

Concurrency is a central topic in server-software development.[12] Two sources of concurrency prevail in this type of software. First, scale implies concurrency. For example, a search engine may split its index into many small pieces (shards) so that the entire corpus can fit in main memory. To satisfy queries efficiently, all shards must be queried concurrently. Second, communication between servers is asynchronous and must be handled concurrently for efficiency and safety.

Concurrent programming is traditionally approached by employing threads and locks.[3] Threads furnish the programmer with concurrent threads of execution, while locks coordinate the sharing of (mutable) data across multiple threads.

In practice, threads and locks are notoriously difficult to get right.[9] They are hard to reason about, and they are a stubborn cause of nasty bugs. What's more, they are difficult to compose: you cannot safely and arbitrarily combine a set of threads and locks to construct new functionality. Their semantics of computation are wrapped up in the mechanics of managing concurrency.

At Twitter, we instead structure concurrent programs around futures. A future is an effect that represents the result of an asynchronous operation. It's a type of reference cell that can be in one of three states: *incomplete*, when the future has not yet taken on a value; *completed with a value*, when the future holds the result of a successful operation; and *completed with a failure,* when the operation has failed. Futures can undergo at most one state transition: from the incomplete state to either the success or failure state

In the following example, using Scala, the future `count` represents the result of an integer-valued operation. We *respond* to the future's completion directly: the block of code after `respond` is a callback that is invoked when the future has completed. (As you'll see shortly, we rarely respond directly to future completions in this way.)

```
val count: Future[Int] = getCount()
count.respond {
 case Return(value) =>
  println(s"The count was $value")
 case Throw(exc) =>
  println(s"getCount failed with $exc")
}
```

Futures represent just about every asynchronous operation in Twitter systems: RPC (remote procedure call), timeout, reading a file from disk, receiving the next event from an event stream.

With the help of a set of higher-order functions (called *combinators*), futures can be combined freely to express more complex operations. These combinations usually fall into one of two composition categories: sequential or concurrent.

Sequential composition permits defining a future as a function of another, such that the two are executed sequentially. This is useful where data dependency exists between two operations: the result of the first future is needed to compute the second future. For example, when a user sends a Tweet, we first need to see if that user is within the hourly rate limits before writing the Tweet to a database. In the following example, the future `done` represents this composite operation. (For historical reasons, the sequencing combinator is called flatMap.)

```
def isRateLimited(user: Long): Future[Boolean] = ...
def writeTweet(user: Long, tweet: String): Future[Unit] = ...

val user = 12L
val tweet: String = “just setting up my twitter”

val done: Future[Unit] =
 isRateLimited(user).flatMap {
  case true => Future.exception(new RateLimitingError)
  case false => writeTweet(user, tweet)
 }
```

This also shows how failures are expressed in futures: `Future.exception` returns a future that has already completed in a failure state. In the case of rate limiting, `done` becomes a failed future (with the exception `RateLimitingError`). Failure short-circuits any further composition: if, in the previous example, the future returned by `isRateLimited(user)` fails, then `done` is immediately failed; the closure passed to flatMap is not run.

Another set of combinators defines concurrent composition, allowing multiple futures to be combined when no data dependencies exist among them. Concurrent combinators turn a list of futures into a future of a list of values. For example, you may have a list of futures representing RPCs to all the shards of a search index. The concurrent combinator `collect` turns this list of futures into a future of the list of results.

```
val results: List[Future[String]] = …
val all: Future[List[String]] =
 Future.collect(results)
```

Independent futures are executed concurrently by default; execution is sequenced only where data dependencies exist.

Future combinators never modify the underlying future; instead, they return a *new* future that represents the composite operation. This is an important tool for reasoning: composite operations do not change the behavior of their constituent parts. Indeed, a single future can be used and reused in many different compositions.

Let's modify the earlier `getCount` example to see how composition allows building complex behavior piecemeal. In distributed systems, it is often preferable to degrade gracefully (e.g., where a default value or guess may exist) than to fail an entire operation.[8] This can be implemented by using a timeout for the `getCount` operation, and, upon failure, returning a default value of 0. This behavior can be expressed as a compound operation among different futures. Specifically, you want the future that represents the winner of a timeout-with-default and the `getCount` operation.

```
val count: Future[Int] = getCount()
// Future.sleep(x) returns a future which completes
// after the given amount of time.
val timeout: Future[Unit] = Future.sleep(5.seconds)

// The map method on Future constructs a new Future
// which, after successful completion, applies the given
// function to the result. It returns a new Future
// representing this composite operation. In this case,
// we simply return a default value of 0 after the timeout.
val default: Future[Int] = timeout.map(unit => 0)

// Select composes two Futures, returning a new
// Future which represents the first future to complete.
val finalCount: Future[Int] = Future.select(count, default)
```

The future `finalCount` now represents the composite operation as described. This example has a number of notable features. First, we have created a composite operation using simple, underlying parts—futures and functions. Second, the constituent futures preserve their semantics under composition—their behavior does not change, and they may be used in multiple different compositions. Third, nothing has been said about the mechanics of execution; instead, the composite computation is expressed as the combination of a number of underlying parts. No threads were explicitly created, nor was there any explicit communication between them—it is all implied by data dependencies.

This neatly illustrates how futures liberate the application's semantics (what is computed) from its mechanics (how it is computed). The programmer composes concurrent operations but needn't specify how they are scheduled or how values are communicated. This is a good separation of concerns: application logic is not entangled with the minutiae of runtime concerns.

Futures can sometimes free programmers from having to use locks. Where data dependencies are witnessed by composition of futures, the implementation is responsible for concurrency control. Put another way, futures can be composed into a dependency graph that is executed in the manner of data-flow programming.[11] While we need to resort to explicit concurrency control from time to time, a large set of common use cases are handled directly by the use of futures.

At Twitter, we implement the machinery required to make concurrent execution with futures work in our open-source Finagle[4,5] and Util[6] libraries. These take care of mapping execution onto OS threads through a pluggable scheduler mechanism. Some teams at Twitter have used this capacity to construct scheduling strategies that better match their problem domain and its attendant tradeoffs. We have also used this capability to add features such as maintaining runtime statistics and Dapper-style RPC tracing to our systems, without changing any existing APIs or modifying existing user code.

"We should have some ways of coupling programs like garden hose—screw in another segment when it becomes necessary to massage data in another way. This is the way of I/O also."
—Doug McIlroy

PROGRAMMING WITH SERVICES AND FILTERS

Modern server software abounds with essential complexity—that which is inherent to the problem and cannot be avoided—as well as complexity of the more self-inflicted variety (did we really need that product feature)? Anyhow, since we can't seem to keep our applications simple, we must instead cope with their complexities.

Thus, the modern software engineering practice is centered on how to contain and manage complexity—to package it up in ways that allow us to reason about our application's behavior. In this endeavor the goal is to balance simplicity and clarity with reusability and modularity.

What's more, server software must account for the realities of distributed systems. For example, a search engine might simply omit results from a failing index shard so that it can return a partial result rather than failing the query in its entirety (as seen in the `getCount` example). In this case, the user would not usually be able to tell the difference—the search engine is still useful without a complete set of results. Applying application-level knowledge in this way is often essential to creating a truly resilient application.

Distributed applications are therefore organized around *services* and *filters*. Services represent RPC service endpoints. They are a function of type `A => Future[R]` (i.e., a function that takes a single argument of type `A` and returns an `R`-typed future). These functions are asynchronous: they return immediately; deferred actions are captured by the returned future. Services are symmetric: RPC clients call services to dispatch RPC calls;

servers implement services to handle them.

The following example defines a simple dictionary service that looks up a key in a map, which is stored in memory. The service takes a `String` argument and returns a `String` value, or else null if the key is not found.

```
// This is the dictionary data, encoded in a
// string-to-string map.
val map: Map[String, String] = Map(...)

val dictionary: Service[String, String] =
 // Construct a new service from an
 // an anonymous function, invoked
 // for every request issued.
 Service(key: String => {
  // Return the lookup results. (Map.get
  // returns an optional string.) Future.value
  // creates a Future that is already
  // satisfied with the given value.
  if (map.contains(key))
   Future.value(map(key))
  else
   Future.value(null)
 })
```

Note that the service, `dictionary`, is again a value like any other. Once defined, it can be made available for dispatch over the network through a favorite RPC protocol. On the server, services are exported as

```
Rpc.serve(dictionary, “:8080”)
```

while clients can bind and call remote instances of the service:

```
val service: Service[String, String] =
 Rpc.bind("server:8080")

val result: Future[String] =
 service("key")
```

Observe the symmetry between client and server: both are conversing in terms of services, which are location-transparent; it is an implementation detail that one is implemented locally while the other is bound remotely.

Services are easily composed using ordinary functional composition. For example, the following service performs a scatter-gather lookup across multiple dictionary services. The dictionary can therefore be distributed over multiple shards.

```
def multipleLookup(services: Seq[Service[String, String]])
  : Service[String, String] =
 Service(key => {
  val results: Seq[Future[String]] =
   services.map(_.apply(key))
  val all: Future[Seq[String]] =
   Future.collect(results)
  all.map { results: Seq[String] =>
   results.indexWhere(_ != null) match {
    case -1 => null
    case i => results(i)
   }
  }
 })
```

This example has a lot going on. First, each underlying service is called with the desired key, obtaining a sequence of results—all futures. `Future.collect` composes futures concurrently, turning a sequence of futures into a single one containing a sequence of the values of the (successful) completion of the constituent futures. The code then looks for the first non-null result and returns it.

Filters are also asynchronous functions that are combined with services to modify their behavior. Their type is `(A, Service[A, R]) => Future[R]` (i.e., a function with two arguments: a value of type `A` and a service `Service[A, R]`); the filter then returns a future of that service's return type, `R`. The filter's type indicates that it is responsible for satisfying a request, given a service. It can thus modify both how the request is dispatched to the service (e.g., it can modify it) and how it is returned (e.g., add a timeout to the returned future).

Filters are used to implement functionality such as timeouts and retries, failure-handling strategies, and authentication. Filters may be combined to form compound filters—for example, a filter that implements retry logic can be combined with a filter implementing timeouts. Finally, filters are combined with a service to create a new service with modified behavior.

Building on the previous example, the following is a filter that downgrades failures from lookup services to null. This kind of behavior is often useful when constructing resilient services—it's sometimes better to return partial results than to fail altogether.

```
val downgradeToNull: Filter[String, String] =
 Filter((key, service) => {
  service.apply(key).transform {
   case Return(value) => value
   case Throw(exception) => null
  }
 })
```

Another filter short-circuits requests for useless keys (these are known as *stopwords* in search engines), so that they don't inflict needless load on the underlying service:

```
val stopwords: Set[String] …
val stopwordFilter: Filter[String, String] =
 Filter((key, service) => {
  if (stopwords.contains(key))
   Future.value(null)
  else
   service.apply(key)
 })
```

Finally, the two filters are combined and then applied to all of the services.

```
val resiliencyFilter: Filter[String, String] =
 stopwordFilter.andThen(downgradeToNull)

def resilientMultipleLookup(services: Service[String, String]) = {
 val resilientServices =
  services.map { service => resiliencyFilter.andThen(service) }
 multipleLookup(resilientServices)
}

val resilientService: Service[String, String] =
 resilientMultipleLookup(
  Rpc.bind("server1:8080"),
  Rpc.bind("server2:8080"),
  ...
 )
```

The dictionary lookup service `resilientService` implements scatter-gather lookup in a resilient fashion. The functionality was built from individual well-defined components that are composed together to create a service that behaves in a desirable way.

As with futures, combining filters and services does not change the underlying, constituent components. It creates a new service with new behavior but does not change the meaning of either underlying filter or service. This again enhances reasoning since the constituent components stand on their own; we need not reason about their interactions after they are combined.

Futures, services, and filters form the foundation upon which server software is built at Twitter. Together they support both modularity and reuse: we define applications and behaviors independently, composing them together as required. While their application is pervasive, two examples nicely illustrate their power. First, we implemented an RPC tracing system à la Dapper[10] as a set of filters, requiring no changes to application code. Second, we implemented backup requests[2] as a small, self-contained filter.

"Don't tie the hands of the implementer."
—Martin Rinard

CONCLUSION

Functional programming promotes thinking about building complex behavior out of simple parts, using higher-order functions and effects to glue them together. At Twitter we have applied this line of thinking to distributed computing, structuring systems around a set of core abstractions that express asynchrony through effects and that are composable. This allows building complex systems from components with simple semantics that, preserved under composition, makes it easier to reason about the system as a whole.

This approach leads to simpler and more modular systems. These systems promote a good separation of concerns and enhance flexibility, while at the same time permitting efficient implementations.

Functional programming has thus furnished essential tools for managing the complexity that is inherent in modern software—untying the hands of the implementer.

Acknowledgments

Jake Donham, Erik Meijer, Mark Compton, Terry Coatta, Steve Jenson, Kevin Oliver, Ruben Oanta, and Oliver Gould provided excellent feedback and guidance on earlier drafts of this article. Early versions of the abstractions discussed here were designed together with Nick Kallen; numerous people at Twitter and in the open-source community have since worked to improve, expand, and solidify them.

References

1. Barroso, L. A., Clidaras, J., Hölzle, U. 2013. The datacenter as a computer: an introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture* 8(3): 1-154.
2. Dean, J., Barroso, L. A. 2013. The tail at scale. *Communications of the ACM* 56(2): 74-80.
3. Dijkstra, E. W. 1965. Solution of a problem in concurrent programming control. *Communications of the ACM* 8(9): 569.
4. Eriksen, M. 2013. Your server as a function. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, ACM (November): 5.
5. Eriksen, M., Kallen, N. 2010. Finagle; http://twitter.github.com/finagle.
6. Eriksen, M., Kallen, N. 2010. Util; http://twitter.github.com/util/.
7. Hughes, J. 1989. Why functional programming matters. *The Computer Journal* 32(2): 98-107.
8. Netflix. Hystrix; https://github.com/Netflix/Hystrix.
9. Ousterhout, J. 1996. Why threads are a bad idea (for most purposes). In presentation given at the Usenix Annual Technical Conference, vol. 5.
10. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C., 2010.

Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google: 36.

11. Smolka, G. 1995. The Oz programming model. In *Computer Science Today*, ed. Jan van Leeuwen, Lecture Notes in Computer Science, Volume 1000: 324-343. Berlin: Springer-Verlag.
12. Sutter, H. 2005. The free lunch is over: a fundamental turn toward concurrency in software. *Dr. Dobb's Journal* 30(3): 202-210.

Marius Eriksen *is a principal engineer in Twitter's systems infrastructure group. He works on all aspects of distributed systems and server software and is currently working on data management and integration systems; he also chairs Twitter's architecture group. You can reach him at marius@twitter.com or @marius on Twitter.*

CONTENTS

TEXT ONLY

Scaling Synchronization in Multicore Programs

ADVANCED SYNCHRONIZATION METHODS CAN BOOST THE PERFORMANCE OF MULTICORE SOFTWARE.

ADAM MORRISON, TEL AVIV UNIVERSITY

Designing software for modern multicore processors poses a dilemma. Traditional software designs, in which threads manipulate shared data, have limited scalability because synchronization of updates to shared data serializes threads and limits parallelism. Alternative distributed software designs, in which threads do not share mutable data, eliminate synchronization and offer better scalability. But distributed designs make it challenging to implement features that shared data structures naturally provide, such as dynamic load balancing and strong consistency guarantees, and are simply not a good fit for every program.

Often, however, the performance of shared mutable data structures is limited by the synchronization methods in use today, whether lock-based or lock-free. To help readers make informed design decisions, this article describes advanced (and practical) synchronization methods that can push the performance of designs using shared mutable data to levels that are acceptable to many applications.

PROS AND CONS OF SHARED MUTABLE DATA

To get a taste of the dilemmas involved in designing multicore software, let us consider a concrete problem: implementing a *work queue*, which allows threads to enqueue and dequeue work items—events to handle, packets to process, and so on. Issues similar to those discussed here apply in general to multicore software design.[14]

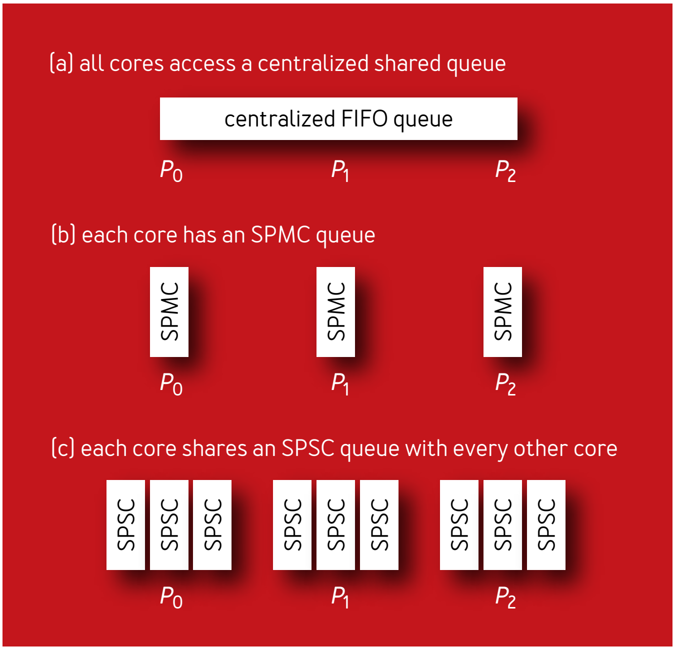
Centralized shared queue

One natural work queue design (depicted in figure 1a) is to implement a centralized shared (thread-safe) version of the familiar FIFO (first in, first out) queue data structure—say, based on a linked list. This data structure supports enqueuing and dequeuing with a constant number of memory operations. It also easily facilitates dynamic load balancing: because all pending work is stored in the data structure, idle threads can easily acquire work to perform. To make the data structure thread-safe, however, updates to the head and tail of the queue must be synchronized, and this inevitably limits scalability.

Using locks to protect the queue serializes its operations: only one core at a time can update the queue, and the others must wait for their turns. This ends up creating a sequential bottleneck and destroying performance very quickly. One possibility to increase scalability is by replacing locks with *lock-free*

1

FIGURE 1: **POSSIBLE DESIGNS FOR A WORK QUEUE**


(a) all cores access a centralized shared queue
centralized FIFO queue
P_0 P_1 P_2
(b) each core has an SPMC queue
SPMC SPMC SPMC
P_0 P_1 P_2
(c) each core shares an SPSC queue with every other core
SPSC SPSC SPSC SPSC SPSC SPSC SPSC SPSC SPSC
P_0 P_1 P_2

synchronization, which directly manipulates the queue using atomic instructions,[1,11] thereby reducing the amount of serialization. (Serialization is still a problem because the hardware cache coherence mechanism[1] serializes atomic instructions updating the same memory location.) In practice, however, lock-free synchronization often does not outperform lock-based synchronization, for reasons to be discussed later.

Partially distributed queue

Alternative work-queue designs seek scalability by distributing the data structure, which allows for more parallelism but gives up some of the properties of the centralized shared queue. For example, figure 1b shows a design that uses one SPMC (single-producer/multiple-consumer) queue per core. Each core enqueues work into its queue. Dequeues can be implemented in various ways—say, by iterating over all the queues (with the starting point selected at random) until finding one containing work.

This design should scale much better than the centralized shared queue: enqueues by different cores run in parallel, as they update different queues, and (assuming all queues contain work) dequeues by different cores are expected to pick different queues to dequeue from, so they will also run in parallel.

What this design trades off, though, is the data structure's *consistency guarantee*. In particular, unlike the centralized shared queue, the distributed design does not maintain the cause and effect relation in the program. Even if core P_1 enqueues x_1 to its queue after core P_0 enqueues x_0 to its queue, x_1 may be dequeued before x_0. The design *weakens* the consistency guarantees provided by the data structure.

The fundamental reason for this weakening is that in a distributed design, it is hard (and slow) to combine the per-core data into a *consistent* view of the data structure—one that would have been produced by the simple centralized implementation. Instead, as in this case, distributed designs usually weaken the data structure's consistency guarantees.[5,8,14] Whether the weaker guarantees are acceptable or not depends on the application, but figuring this out—reasoning about the acceptable behaviors—complicates the task of using the data structure.

Despite its more distributed nature, this per-core SPMC queue design can still create a bottleneck when load is not balanced. If only one core generates work, for example, then dequeuing cores all swoop on its queue and their operations become serialized.

Distributed queue

To eliminate many-thread synchronization altogether, you can turn to a design such as the one depicted in figure 1c, with each core maintaining one SPSC (single-producer/single-consumer) queue for each other core in the system, into which it enqueues items that it wishes its peer to dequeue. As before, this design weakens the consistency guarantee of the queue. It also makes dynamic load balancing more difficult because it chooses which core will dequeue an item in advance.

Motivation for improving synchronization

The crux of this discussion is that obtaining scalability by distributing the queue data structure trades off some useful properties that a centralized shared queue provides. Usually, however, these tradeoffs are clouded by the unacceptable performance of centralized data structures, which obviates any benefit they might offer. This article makes the point that much of this poor performance is a result of inefficient synchronization methods. It surveys advanced synchronization methods that boost the performance of centralized designs and make them acceptable to more applications. With these methods at hand, designers can make more informed choices when architecting their systems.

SCALING LOCKING WITH DELEGATION

Locking inherently serializes executions of the critical sections it protects. Locks therefore limit scaling: the more cores there are, the longer each core has to wait for its turn to execute the critical section, and so beyond some number of cores, these waiting times dominate the computation. This scalability limit, however, can be pushed quite high—in some cases, beyond the scales of current systems—by *serializing more efficiently*. Locks that serialize more efficiently support more operations per second and therefore can handle workloads with more cores.

More precisely, the goal is to minimize the computation's *critical path*: the length of the longest series of operations that have to be performed sequentially because of data dependencies. When using locks, the critical path contains successful lock acquisitions, execution of the critical sections, and lock releases.

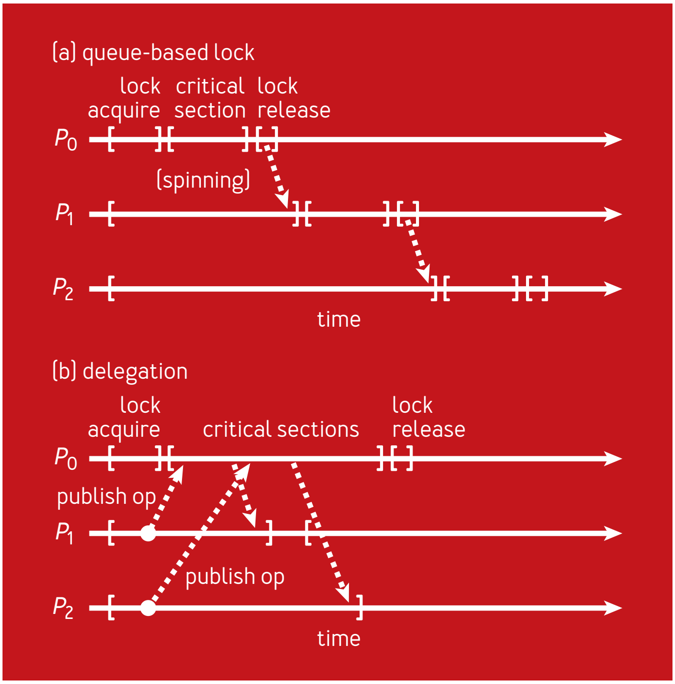
As an example of inefficient serialization that limits scalability, consider the *lock contention* that infamously occurs in simple spin locks. When many cores simultaneously try to acquire a spin lock, they cause its cache line to bounce among them, which slows down lock acquisitions/releases. This increases the length of the critical path and leads to a performance "meltdown" as core counts increase.[2]

Lock contention has a known solution in the form of scalable queue-based locks.[2,10] Instead of having all waiting threads compete to be the next one to acquire the lock, queue-based locks line up the waiting threads, enabling a lock release to hand the lock to the next waiting thread. These hand-offs, which require only a constant number of cache misses per acquisition/release, speed up lock acquisition/release and decrease the length of the critical path, as depicted in figure 2a: the critical path of a queue-based lock contains only a single transfer of the lock's cache line (dotted arrow).

Can lock-based serialization be made even more efficient? The *delegation* synchronization method described in this section does so: it eliminates most lock acquisitions and releases from the critical path, and it speeds up execution of the critical sections themselves.

Delegation. In a delegation lock, the core holding the

FIGURE 2: **CRITICAL PATH OF LOCK-BASED CODE**



lock acts as a server and executes the operations that the cores waiting to acquire the lock wish to perform. Delegation improves scalability in several ways (figure 2b). First, it eliminates the lock acquisitions and releases that would otherwise have been performed by waiting threads. Second, it speeds up the execution of operations (critical sections), because the data structure is hot in the server's cache and does not have to be transferred from a remote cache or from memory. Delegation also enables new optimizations that exploit the semantics of the data structure to speed up critical-section execution even further, as will be described shortly.

Implementing delegation. The idea of serializing faster by having a single thread execute the operations of the waiting threads dates to the 1999 work of Oyama et al.,[13] but the overheads of their implementation overshadow its benefits. Hendler et al.,[6] in their *flat combining* work, were first to implement this idea efficiently and to observe that it facilitates optimizations based on the semantics of the executed operations.

In the flat combining algorithm, every thread about to acquire the lock posts the operation it intends to perform (e.g., `dequeue` or `enqueue(x)`) in a shared *publication list*. The thread that acquires the lock becomes the server; the remaining threads spin, waiting for their operations to be applied. The server scans the publication list, applies pending operations, and releases the lock when done. To amortize the synchronization cost of adding records to the publication list, a thread leaves its publication record in the list and reuses it in future operations. Later work explored piggybacking the publication list on top of a queue lock's queue,[4] and boosting cache locality of the operations by dedicating a core for the server role instead of having threads opportunistically become servers.[9]

Semantics-based optimizations. The server thread has a global view of concurrently pending operations, which it can leverage to optimize their execution in two ways:

- *Combining.* The server can combine multiple operations into one and thereby save repeated accesses to the data structure. For example, multiple counter-increment operations can be converted into one addition.
- *Elimination.* Mutually canceling operations, such as a counter increment and decrement, or an insertion and removal of the same item from a set, can be executed without modifying the data structure at all.

Deferring delegation. For operations that only update the data structure but do not return a value, such as `enqueue()`, delegation facilitates an optimization that can sometimes eliminate serialization altogether. Since these operations do not return a response to the invoking core, the core does not have to wait for the server to execute them; it can just log the requested operation in the publication list and keep running. If the core later invokes an operation whose return value depends on the state of the data structure, such as a `dequeue()`, it must then wait for the server to apply all its prior operations. But until such a time—which in update-heavy workloads can be rare—all of its operations execute asynchronously.

The original implementation of this optimization still required cores to synchronize when executing these deferred operations, since it logged the operations in a centralized (lock-free) queue.[7] Boyd-Wickizer et al.,[3] however, implemented deferred delegation without any synchronization on updates by leveraging systemwide synchronized clocks. Their OpLog library logs invocations of responseless update operations in a per-core log, along with their invocation times. Operations that read the data structure become servers: they acquire the locks of all per-core logs, apply the operations in timestamp order, and then read the updated data-structure state. OpLog thus creates scalable implementations of data structures that are updated heavily but read rarely, such as LRU (least recently used) caches.

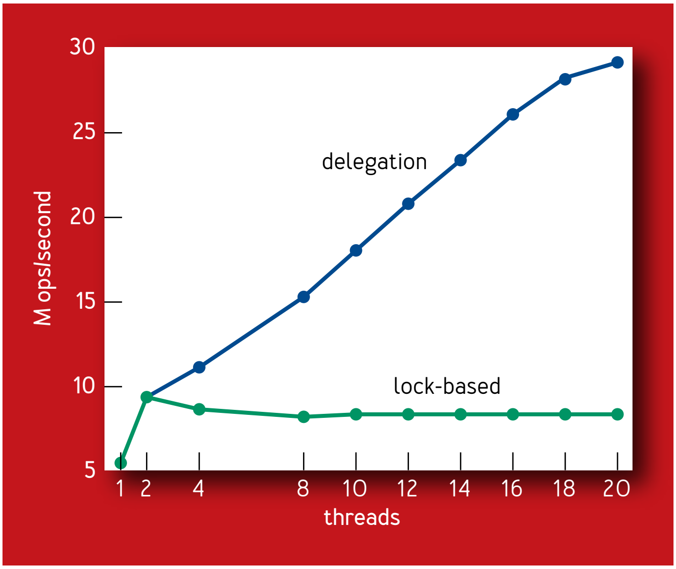
Performance

To demonstrate the benefits of delegation, let's compare a lock-based work queue to a queue implemented using delegation. The lock-based algorithm is Michael and Scott's *two-lock queue*.[11] This algorithm protects updates to the queue's head and tail with different locks, serializing operations of the same type but allowing enqueues and dequeues to run in parallel. Queue-based CLH (Craig, Landin, and Hagerstein) locks are used in the evaluated implementation of the lock-based algorithm. The delegation-based queue is Fatourou and Kallimanis' *CC-Queue*,[4] which adds delegation to each of the two locks in the lock-based algorithm. (It thus has two servers running: one for dequeues and one for enqueues.)

Figure 3 shows enqueue/dequeue throughput comparison (higher is better) of the lock-based queue and its delegation-based version. The benchmark models a generic application. Each core repeatedly accesses the data structure, performing pairs of enqueue and

3

FIGURE 3: **ENQUEUE/DEQUEUE THROUGHPUT COMPARISON**



dequeue operations, reporting the throughput of queue operations (i.e., the total number of queue operations completed per second). To model the work done in a real application, a period of "think time" is inserted after each queue operation. Think times are chosen uniformly at random from 1 to 100 nanoseconds to model challenging workloads in which queues are heavily exercised. The C implementations of the algorithms from Fatourou and Kallimanis's benchmark framework (https://github.com/nkallima/sim-universal-construction) are used, along with a scalable memory allocation library to avoid `malloc` bottlenecks. No semantics-based optimization is implemented.

This benchmark (and all other experiments reported in this article) was run on an Intel Xeon E7-4870 (Westmere EX) processor. The processor has ten 2.40 GHz cores, each of which multiplexes two hardware threads, for a total of twenty hardware threads.

Figure 3 shows the benchmark throughput results, averaged over ten runs. The lock-based algorithm scales to two threads, because it uses two locks, but fails to scale beyond that amount of concurrency because of serialization. In contrast, the delegation-based algorithm scales and ultimately performs almost 30 million operations per second, which is more than 3.5 times that of the lock-based algorithm's throughput.

AVOIDING CAS FAILURES IN LOCK-FREE SYNCHRONIZATION

Lock-free synchronization (also referred to as nonblocking synchronization) directly manipulates shared data using atomic instructions instead of locks. Most lock-free algorithms use the CAS (compare-and-swap) instruction (or equivalent) available on all multicore processors. A CAS takes three operands: a memory address `addr`, an `old` value, and a `new` value. It atomically updates the value stored in `addr` from `old` to `new`; if the value stored in `addr` is not `old`, the CAS fails without updating memory.

CAS-based lock-free algorithms synchronize with a CAS loop pattern: a core reads the shared state, computes a new value, and uses CAS to update a shared variable to the new value. If the CAS succeeds, this read-compute-update sequence appears to be atomic; otherwise, the core must retry. Figure 4 shows an example of such a CAS loop for linking a node to the head of a linked list, taken from

FIGURE 4: **LOCK-FREE LINKING OF A NODE TO THE HEAD OF A LINKED LIST**

```
struct Node {
    struct Node* next;
    void* value;
}
// Pointer to head of the list
Node* head = NULL;

void enqueue (void* v) {
     Node *old, *new = malloc();
     new->value = v;
     while (true) {
          old = head;
          new->next = old;
          if  ( CAS(&head, old, new) )
              return;
}   }
```

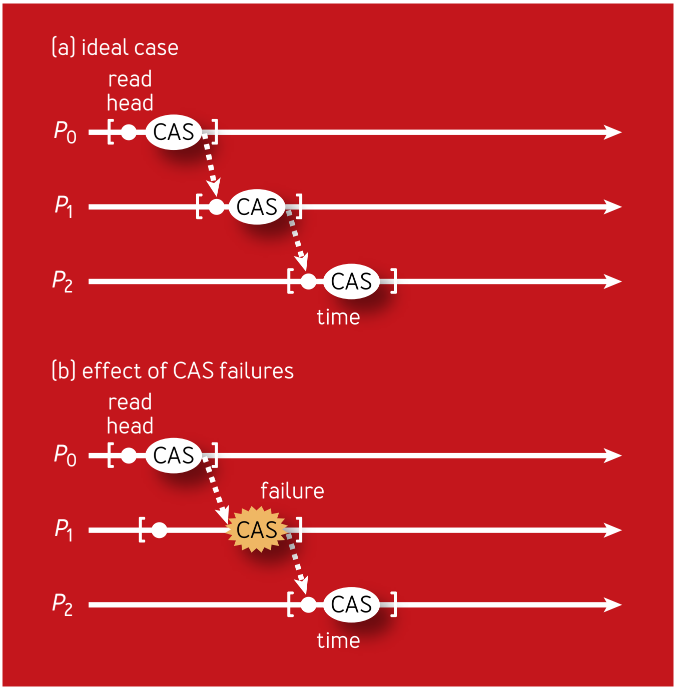
Treiber's classic LIFO (last in, first out) stack algorithm.[15] Similar ideas underlie the lock-free implementations of many basic data structures such as queues, stacks, and priority queues, all of which essentially perform an entire data-structure update with a single atomic instruction.

The use of (sometimes multiple) atomic instructions can make lock-free synchronization slower than a lock-based solution when there is no (or only light) contention. Under high contention, however, lock-free synchronization has the potential to be much more efficient than lock-based synchronization, as it eliminates lock acquire and release operations from the critical path, leaving only the data structure operations on it (figure 5a).

In addition, lock-free algorithms guarantee that some operation can always complete and thus behave gracefully under high load, whereas a lock-based algorithm can grind to a halt if the operating system preempts a thread that holds a lock.

In practice, however, lock-free algorithms may not live up to these performance expectations. Consider, for example, Michael and Scott's lock-free queue algorithm.[11] This algorithm implements a queue using a linked list, with

FIGURE 5: **CRITICAL PATH OF LOCK-FREE UPDATING HEAD OF LINKED LIST**



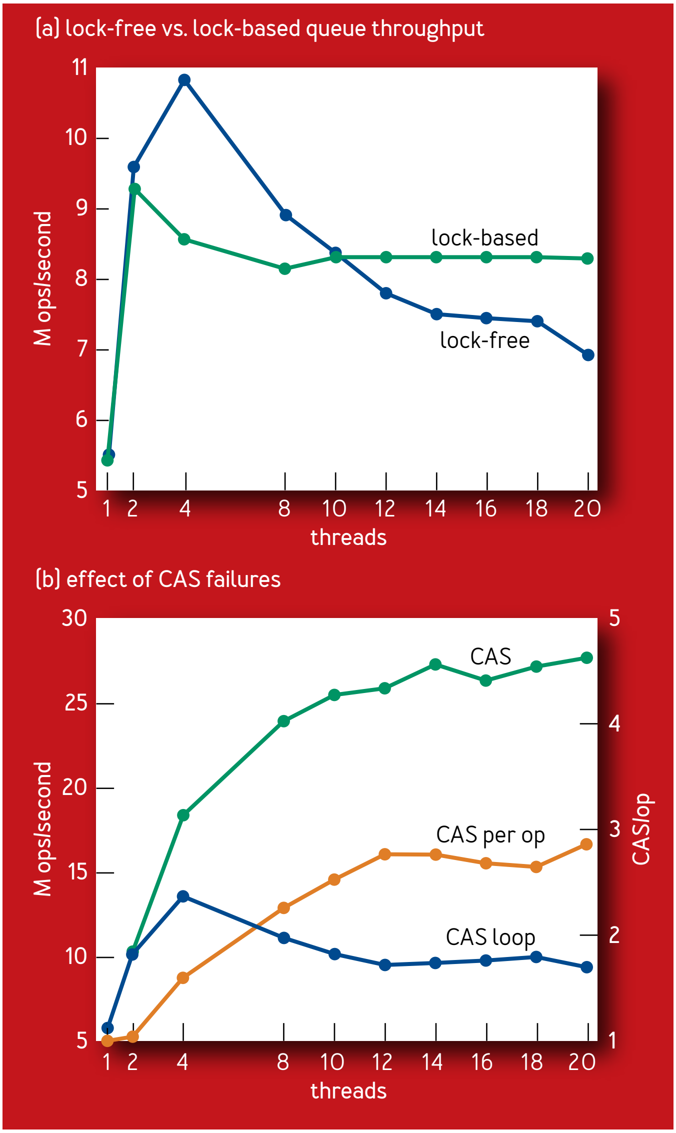
items enqueued to the tail and removed from the head using CAS loops. (The exact details are not as important as the basic idea, which is similar in spirit to the example in figure 4.) Despite this, as figure 6a shows, the lock-free algorithm fails to scale beyond four threads and eventually performs worse than the two-lock queue algorithm.

The reason for this poor performance is *CAS failure*: as the amount of concurrency increases, so does the chance that a conflicting CAS gets interleaved in the middle of a core's read-compute-update CAS region, causing its CAS to fail. CAS operations that fail in this way *pile useless work on the critical path*. Although these failing CASes do not modify memory, executing them still requires obtaining exclusive access to the variable's cache line. This delays the time at which later operations obtain the cache line and complete successfully (see figure 5b, in which only two operations complete in the same time that three operations completed in figure 5a).

To estimate the amount of performance wasted because of CAS failures, figure 6b compares the throughput of successful CASes executed in a CAS loop (as in figure 4) to the total CAS throughput (including failed CASes). Observe that the system executes contending atomic instructions at almost three times the rate ultimately observed in the data structure. If there were a way to make every atomic instruction useful toward completing an operation, you would significantly improve performance. But how can this be achieved, given that CAS failures are inherent?

The key observation to make is that the x86 architecture supports several atomic instructions that always succeed. One such instruction is FAA (fetch-and-add), which

FIGURE 6: **LOCK-FREE SYNCHRONIZATION CAS FAILURE PROBLEM**



atomically adds an integer to a variable and returns the previous value stored in that variable. The following section describes the design of a lock-free queue based on FAA instead of CAS. The algorithm, named LCRQ (for linked concurrent ring queue),[12] uses FAA instructions to

FIGURE 7: **INFINITE ARRAY QUEUE**

```
// The following defines a node
struct Cell {
   void* value;
}
// Queue is infinite array of nodes,
// with head and tail pointers.
Cell Q [] = { ⊥, ⊥, ...};
int head = 0;
int tail = 0;

void enqueue(void* x) {
  while (true) {
   t = FAA(&tail, 1)
   if ( CAS(&Q[t], ⊥, x) ) return
} }
void *dequeue() {
  while (true) {
   h = FAA(&head, 1)
   if ( !CAS(&Q[h], ⊥, ⊤) ) return Q[h]
   if ( tail ≤ h+1 ) return NULL
} }
```

spread threads among items in the queue, allowing them to enqueue and dequeue quickly and in parallel. LCRQ operations typically perform one FAA to obtain their position in the queue, providing exactly the desired behavior.

The LCRQ algorithm

This section presents an overview of the LCRQ algorithm; for a detailed description and evaluation, see the paper.[12] Conceptually, LCRQ can be viewed as a practical realization of the following simple but unrealistic queue algorithm (figure 7). The unrealistic algorithm implements the queue using an *infinite* array, `Q`, with (unbounded) `head` and `tail` indices that identify the part of `Q` that may contain items. Initially, each cell `Q[i]` is empty and contains a reserved value ⊥ that may not be enqueued. The `head` and `tail` indices are manipulated using FAA and are used to spread threads around the cells of the array, where they synchronize using (uncontended) CAS.

An `enqueue(x)` operation obtains a cell index `t` via a FAA on `tail`. The enqueue then atomically places `x` in `Q[t]` using a CAS to update `Q[t]` from ⊥ to `x`. If the CAS succeeds, the enqueue operation completes; otherwise, it repeats this process.

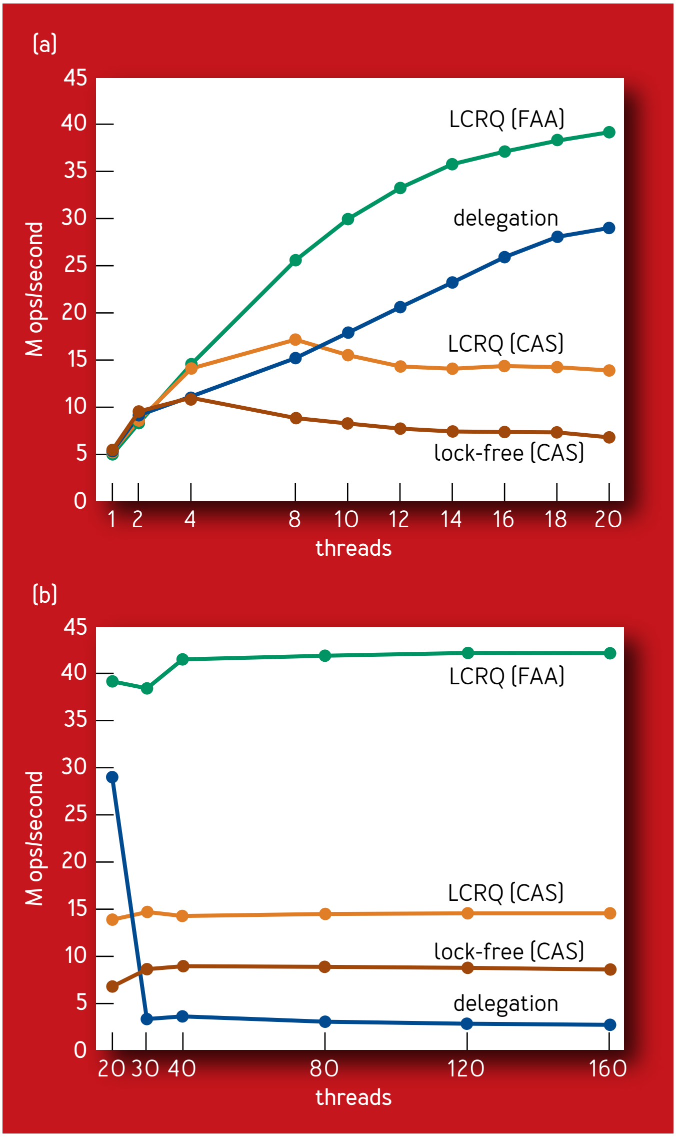
A dequeue, `D`, obtains a cell index `h` using FAA on `head`. It tries to atomically CAS the contents of `Q[h]` from ⊥ to another reserved value ⊤. This CAS fails if `Q[h]` contained some `x ≠` ⊥, in which case `D` returns `x`. Otherwise, the fact that `D` stored ⊤ in the cell guarantees that an enqueue operation that later tries to store an item in `Q[h]` will not succeed. `D` then returns NULL (indicating the queue is empty) if `tail ≤ h + 1` (the value of `head` following `D`'s FAA is `h + 1`). If `D` cannot return NULL, it repeats this process.

This algorithm can be shown to implement a FIFO queue correctly, but it has two major flaws that prevent it from being relevant in practice: using an infinite array and susceptibility to livelock (when a dequeuer continuously writes `T` into the cell an enqueuer is about to access). The practical LCRQ algorithm addresses these flaws.

The infinite array is first collapsed to a concurrent ring (cyclic array) queue—CRQ for short—of `R` cells. The `head` and `tail` indices still strictly increase, but now the value of an index modulo `R` specifies the ring cell to which it points. Because now more than one enqueuer and dequeuer can concurrently access a cell, the CRQ uses a more involved CAS-based protocol for synchronizing within each cell. This protocol enables an operation to avoid waiting for the completion of operations whose FAA returns smaller indices that also point to the same ring cell.

The CRQ's crucial performance property is that in the common fast path, an operation executes only one FAA instruction. The LCRQ algorithm then builds on the CRQ to prevent the livelock problem and handle the case of the CRQ filling up. The LCRQ is essentially a Michael and Scott linked list queue[11] in which each node is a CRQ. A CRQ that fills up or experiences livelock becomes closed to further enqueues, which instead append a new CRQ to the list and begin working in it. Most of the activity in the LCRQ therefore occurs in the individual CRQs, making contention (and CAS failures) on the list's head and tail a nonissue.

FIGURE 8: ENQUEUE/DEQUEUE THROUGHPUT COMPARISON OF ALL QUEUES



Performance

This section compares the LCRQ to Michael and Scott's classic lock-free queue,[11] as well as to the delegation-based variant presented in the previous section. The impact of CAS failures is explored by testing LCRQ-CAS, a version of LCRQ in which FAA is implemented with a CAS loop.

Figure 8a shows the results. LCRQ outperforms all other queues beyond two threads, achieving peak throughput of ≈ 40 million operations per second, or about 1,000 cycles per queue operation. From eight threads onward, LCRQ outperforms the delegation-based queue by 1.4 to 1.5 times and the MS (Michael and Scott) queue by more than three times. LCRQ-CAS matches LCRQ's performance up to four threads, but at that point its performance levels off. Subsequently, LCRQ-CAS exhibits the throughput "meltdown" associated with CAS failures. Similarly, the MS queue's performance peaks at two threads and degrades as concurrency increases.

Oversubscribed workloads can demonstrate the graceful behavior of lock-free algorithms under high load. In these workloads the number of software threads exceeds the hardware-supported level, forcing the operating system to context-switch between threads. If a thread holding a lock is preempted, a lock-based algorithm cannot make progress until it runs again. Indeed, as figure 8b shows, when the number of threads exceeds 20, the throughput of the lock-based delegation algorithm plummets by 15 times, whereas both LCRQ and the MS queue maintain their peak throughput.

CONCLUSION

Advanced synchronization methods can boost the performance of shared mutable data structures. Synchronization still has its price, and when performance demands are extreme (or if the properties of centralized data structures are not needed), then distributed data structures are probably the right choice. For the many remaining cases, however, the methods described in this article can help build high-performance software. Awareness of these methods can assist those designing software for multicore machines.

References

1. Al Bahra, S. 2013. Nonblocking algorithms and scalable multicore programming. *Communications of the ACM* 56(7): 50–61.
2. Boyd-Wickizer, S., Frans Kaashoek, M., Morris, R., Zeldovich, N. 2012. Non-scalable locks are dangerous. In *Proceedings of the Ottawa Linux Symposium*: 121–132.
3. Boyd-Wickizer, S., Frans Kaashoek, M., Morris, R., Zeldovich, N. 2014. OpLog: a library for scaling update-heavy data structures. Technical Report MIT-CSAIL-TR2014-019.
4. Fatourou, P., Kallimanis, N. D. 2012. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*: 257– 266.
5. Haas, A., Lippautz, M., Henzinger, T. A., Payer, H., Sokolova, A., Kirsch, C. M., Sezgin, A. 2013. Distributed queues in shared memory: multicore performance and scalability through quantitative relaxation. In

Proceedings of the ACM International Conference on Computing Frontiers: 17:1–17:9.

6. Hendler, D., Incze, I., Shavit, N., Tzafrir, M. 2010. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*: 355–364.
7. Klaftenegger, D., Sagonas, K., Winblad, K. 2014. Delegation locking libraries for improved performance of multithreaded programs. In *Proceedings of the 20th International European Conference on Parallel and Distributed Computing*: 572–583.
8. Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., Chew, L. P. 2007. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*: 211–222.
9. Lozi, J.-P., David, F., Thomas, G., Lawall, J., Muller, G. 2012. Remote core locking: migrating critical section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Annual Technical Conference*: 65–76.
10. Mellor-Crummey, J. M., Scott, M. L. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems* 9(1): 21–65 .
11. Michael, M. M., Scott, M. L. 1996. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*: 267–275.
12. Morrison, A., Afek, Y. 2013. Fast concurrent queues

for x86 processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*: 103–112.

13. Oyama, Y., Taura, K., Yonezawa, A. 1999. Executing parallel programs with synchronization bottlenecks efficiently. In *Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*: 182–204.
14. Shavit, N. Data structures in the multicore age. 2011. *Communications of the ACM* 54(3): 76–84.
15. Treiber, R. K. 2006. Systems programming: coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center.

Adam Morrison *works on making parallel and distributed systems simpler to use without compromising their performance; his research interests span computer architecture, systems software and theory of distributed computing. He is an assistant professor at the Blavatnik School of Computer Science, Tel Aviv University, Israel. He received a PhD in computer science from Tel Aviv University and then spent three years as a postdoctoral fellow at the Technion—Israel Institute of Technology. He has been awarded an IBM PhD Fellowship as well as Intel and Deutsch prizes.*

CONTENTS

Web Security and Mobile Web Computing

EXPERT-CURATED GUIDES TO THE BEST OF CS RESEARCH

Research for Practice combines the resources of the ACM Digital Library, the largest collection of computer science research in the world, with the expertise of the ACM membership. In every RfP column, experts share a short curated selection of papers on a concentrated, practically oriented topic.

Our third installment of Research for Practice brings readings spanning programming languages, compilers, privacy, and the mobile web.

First, Jean Yang provides an overview of how to use information flow techniques to build programs that are secure by construction. As Yang writes, information flow is a conceptually simple "clean idea": the flow of sensitive information across program variables and control statements can be tracked to determine whether information may in fact leak. Making information flow practical is a major challenge, however. Instead of relying on programmers to track information flow, how can compilers and language runtimes be made to do the heavy lifting? How can application writers easily express their privacy policies and understand the implications of a given policy for the set of values that an application user may see? Yang's set of papers directly addresses these questions via a clever mix of techniques from compilers, systems, and language design. This focus on theory made practical is an excellent topic for RfP.

Second, Vijay Janapa Reddi and Yuhao Zhu provide an overview of the challenges for the future of the mobile web. Mobile represents a major frontier in personal computing, with extreme growth in adoption and data volume. Accordingly, Reddi and Zhu outline three major ongoing challenges in mobile web computing: responsiveness of resource loading, energy efficiency of computing devices, and making efficient use of data. In their citations, Reddi and Zhu draw on a set of techniques spanning browsers, programming languages, and data proxying to illustrate the opportunity for "cross-layer optimization" in addressing these challenges. Specifically, by redesigning core components of the web stack, such as caches and resource-fetching logic, systems operators can improve users' mobile web experience. This opportunity for co-design is not simply theoretical: Reddi and Zhu's third citation describes a mobile-optimized compression proxy that is already running in production at Google.

As always, our goal in RfP is to allow readers to become experts in the latest, practically oriented topics in computer science research in a weekend afternoon's worth of reading time. I am grateful to this installment's experts for generously contributing such a strong set of contributions, and, as always, we welcome your feedback!
—Peter Bailis

PRACTICAL INFORMATION FLOW FOR WEB SECURITY

BY JEAN YANG

Information leaks have become so common that many have given up hope when it comes to information security.[3] Data breaches are inevitable anyway, some say.[1] I don't even go on the Internet anymore, other (computers) say.[6]

This despair has led yet others to the Last Resort: reasoning about what our programs actually do. For years, bugs didn't matter as long as your robot could sing. If your program can go twice the speed it did yesterday, who cares what outputs it gives you? But we are starting to learn the hard way that no amount of razzle-dazzle can make up for Facebook leaking your phone number to the people you didn't invite to the party.[4]

This realization is leading us to a new age, one in which reasoning techniques that previously seemed unnecessarily baroque are coming into fashion. Growing pressure from regulators is finally making it increasingly popular to use precise program analysis to ensure software security.[5] Growing demand for producing web applications quickly makes it relevant to develop new paradigms—well-specified ones, at that—for creating secure-by-construction software.

The construction of secure software means solving the important problem of *information flow*. Most of us have heard of trapdoor ways to access information we should not see. For example, one researcher showed that it is possible to discover the phone numbers of thousands of Facebook users simply by searching for random phone numbers.[2] Many such leaks occur not because a system shows sensitive values directly, but because it shows the results of computations—such as search—on sensitive values. Preventing these leaks requires implementing policies not only on sensitive values themselves, but also whenever computations may be affected by sensitive values.

Enforcing policies correctly with respect to information flow means reasoning about sensitive values and policies as they flow through increasingly complex programs, making sure to reveal only information consistent with the privileges associated with each user. There is a body of work dedicated to compile-time and runtime techniques for tracking values through programs for ensuring correct information flow.

While information flow is a clean idea, getting it to work on real programs and systems requires solving many hard problems. The three papers presented here focus on solving the problem of secure information flow for web applications. The first one describes an approach for taking trust out of web applications and shifting it instead to the framework and compiler. The second describes a fully dynamic enforcement technique implemented in a web framework that requires programmers to specify each policy only once. The third describes a web framework that customizes program behavior based on the policies and viewing context.

Shifting trust to the framework and compiler through language-based enforcement

Chong, S., Vikram, K., Myers, A. C. 2007. SIF: enforcing confidentiality and integrity in web applications. In Proceedings of the 16th Usenix Security Symposium; https://www.usenix.org/conference/16th-usenix-security-symposium/sif-enforcing-confidentiality-and-integrity-web.

In securing web applications, a major source of the burden on programmers involves reasoning about how information may be leaked through computations across different parts of an application and across requests. Without additional checks and balances, the programmer must be fully trusted to do this correctly.

This first selection presents a framework that shifts trust from the application to the framework and compiler. The SIF (Servlet Information Flow) framework follows a line of work in language-based information flow focused on checking programs against specifications of security policies. Built using the Java servlet framework, SIF prevents many common sources of information flow—for example, those across multiple requests. SIF applications are written in Jif, a language that extends Java with programmer-provided labels specifying policies for information flow. SIF uses a combination of compile-time and runtime enforcement to ensure that security policies are enforced from the time a request is submitted to when it is returned, with modest enforcement overhead. The major contribution of the SIF work is in showing how to provide assurance (much of it at compile time) about information flow guarantees in complex, dynamic web applications.

Mitigating annotation burden through principled containment

Giffin, D. B., et al. 2012. Hails: protecting data privacy in untrusted web applications. 10th Usenix Symposium on Operating Systems Design and Implementation; https://www.usenix.org/node/170829.

While compile-time checking approaches are great for providing assurance about program security, they often require nontrivial programmer effort. The programmer must not only correctly construct programs with respect to information flow, but also annotate the program with the desired policies.

An alternative approach is confinement: running untrusted code in a restricted way to prevent the code from exhibiting undesired behavior. For information flow, confinement takes the form of tagging sensitive values, tracking them through computations, and checking tags at application endpoints. Such dynamic approaches are often more popular because they require little input from the programmer.

This paper presents Hails, a web framework for principled containment. Hails extends the standard MVC (model-view-controller) paradigm to include policies, implementing the MPVC (model-*policy*-view-controller) paradigm where the programmer may specify label-based policies separately from the rest of the program. Built in Haskell, Hails uses the LIO (labeled IO) library to enforce security policies at the thread/context level and MAC (mandatory access control) to mediate access to resources such as the database. It has good performance for an information flow control framework, handling approximately 47.8 K requests per second.

Hails has been used to build several web applications, and the startup Intrinsic is using a commercial version of Hails. The Hails work shows that it is possible to enforce information flow in web applications with negligible overhead, without requiring programmers to change how they have been programming.

Shifting implementation burden to the framework

Yang, J., et al. 2016. Precise, dynamic information flow for database-backed applications. In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation: 631-647; http://queue.acm.org/rfp/vol14iss4.html.

In the previous two approaches, the programmer remains burdened by constructing programs correctly with respect to information flow. Without a change in the underlying execution model, the most any framework can do is raise exceptions or silently fail when policies are violated.

This paper looks at what the web programming model might look like if information flow policies could be factored out of programs the way memory-managed languages factor out allocation and deallocation. The paper presents Jacqueline, an MPVC framework that allows programmers to specify: (1) how to compute an alternative default for each data value; and (2) high-level policies about when to show each value that may contain database queries and/or depend on sensitive values.

A plausible default for a sensitive location value is the corresponding city. A valid policy is allowing a viewer to see the location only if the viewer is within some radius of the location. This paper presents an implementation strategy for Jacqueline that works with existing SQL databases. While the paper focuses more on demonstrating feasibility than on the nuts and bolts of web security, it de-risks the approach for practitioners who may want to adopt it.

Final Thoughts

The past few years have seen a gradual movement toward the adoption of practical information flow: first with containment, then with microcontainers and microsegmentation. These techniques control which devices and services can interact with policies for software-defined infrastructures such as iptables and software-defined networking. Illumio, vArmour, and GuardiCore are three among the many startups in the microsegmentation space. This evolution toward finer-grained approaches shows that people are becoming more open to the system re-architecting and runtime overheads that come with information flow control approaches. As security becomes even more important and information flow techniques become more practical, the shift toward more adoption will continue.

Acknowledgments

With thanks to Aliza Aufrichtig, Stephen Chong, Vincenzo Iozzo, Leo Meyerovich, and Deian Stefan for comments.

References

1. Balluck, K. 2014. Corporate data breaches "inevitable," expert says. *The Hill* (November 30); http://thehill.com/policy/cybersecurity/225550-cybersecurity-expert-data-breaches-inevitable.
2. Cunningham, M. 2015. Facebook security flaw could leak your personal info to criminals. Komando.com (August 10); http://www.komando.com/happening-now/320275/facebook-security-flaw-could-leak-your-personal-info-to-criminals/all.
3. Information is beautiful. 2016. World's biggest data

breaches; http://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/.

4. Gellman, B., Poitras, L. 2013. U.S., British intelligence mining data from nine U.S. Internet companies in broad, secret program. *Washington Post* (June 7); http://wpo.st/rwJq1
5. OWASP (Open Web Application Security Project. 2016. Static code analysis; https://www.owasp.org/index.php/Static_Code_Analysis.
6. Zetter, K. 2014. Hacker lexicon: What is an air gap? *Wired* (December 8); http://www.wired.com/2014/12/hacker-lexicon-air-gap/.

THE RED FUTURE OF MOBILE WEB COMPUTING

BY VIJAY JANAPA REDDI AND YUHAO ZHU

The web is on the cusp of a new evolution, driven by today's most pervasive personal computing platform—mobile devices. At present, there are more than 3 billion web-connected mobile devices. By 2020, there will be 50 billion such devices. In many markets around the world mobile web traffic volume exceeds desktop web traffic, and it continues to grow in double digits.

Three significant challenges stand in the way of the future mobile Web. The papers selected here focus on carefully addressing these challenges. The first major challenge is the *responsiveness* of Web applications. It is estimated that a one-second delay in web page load time costs Amazon $1.6 billion in annual sales lost, since mobile users abandon a web service altogether if the web page takes too long to load. Google loses 8 million searches from a four-tenths-of-a-second slowdown in search-results generation. A key bottleneck of mobile web responsiveness is resource loading. The number of objects in today's web pages is already on the order of hundreds, and it continues to grow steadily. Future mobile web computing systems must improve resource-loading performance, which is the focus of the first paper.

The second major challenge is *energy efficiency*. Mobile devices are severely constrained by the battery. While computing capability driven by Moore's Law advances approximately every two years, battery capacity doubles every 10 years—creating a widening gap between computational horsepower and the energy needed to power the device. Therefore, future mobile web computing must be energy efficient. The second paper in our selection proposes web programming language support for energy efficiency.

The third major challenge is *data usage*. A significant amount of future mobile web usage will come from emerging markets in developing countries where the cost of mobile data is prohibitively large. To accelerate the web's growth in emerging markets, future mobile web computing infrastructure must serve data consciously. The final paper discusses how to design a practical and efficient HTTP data compression proxy service that operates at Google's scale.

Developers and system architects must optimize for RED (responsiveness, energy efficiency, and data usage), ideally together, to usher in a new generation of mobile web computing.

Intelligent Resource Loading for Responsiveness

Netravali et al. 2016. Polaris: faster page loads using fine-grained dependency tracking. 13th Usenix Symposium on Networked Systems Design and Implementation; https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/netravali.

A key bottleneck for mobile web responsiveness is resource loading. The bottleneck stems from the increasing number of objects (e.g., images and Cascading Style Sheets files) on a web page. According to the HTTP Archive, over the past three years alone, web pages have doubled in size. Therefore, improving resource-loading performance is crucial for improving the overall mobile web experience.

Resource loading is largely determined by the critical path of the resources that web browsers load to render a page. This critical path, in the form of a resource-dependency graph, is not revealed to web browsers statically. Therefore, today's browsers make conservative decisions during resource loading. To avoid resource-dependency violations, a web browser typically constrains its resource-loading concurrency, which results in reduced performance.

Polaris is a system for speeding up the loading of web page resources, an important step in coping with the surge in mobile web resources. Polaris constructs a precise resource-dependency graph offline, and it uses the graph at runtime to determine an optimal resource-loading schedule. The resulting schedule maximizes concurrency and, therefore, drastically improves mobile web performance. Polaris also stands out because of its transparent design. It runs on top of unmodified web browsers without the intervention of either web application or browser developers. Such a design minimizes the deployment inconvenience and increases its chances of adoption, two factors that are essential for deploying the web effectively.

Web Language Support for Energy Efficiency

Zhu, Y., Reddi, J. 2016. GreenWeb: language extensions for energy-efficient mobile web computing. Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation: 145-160; http://queue.acm.org/rfp/vol14iss4.html.

Energy efficiency is the single most critical constraint on mobile devices that lack an external power supply. Web runtimes (typically the browser engine) must start to budget web application energy usage wisely, informed by user QoS (quality-of-service) constraints. End-user QoS information, however, is largely unaccounted for in today's web programming languages.

The philosophy behind GreenWeb is that application developers provide minimal yet vital QoS information to guide the browser's runtime energy optimizations. Empowering a new generation of energy-conscious web application developers necessitates new programming abstractions at the language level. GreenWeb proposes two new language constructs, *QoS type* and *QoS target*, to capture the critical aspects of user QoS experience. With the developer-assisted QoS information, a GreenWeb browser determines how to deliver the specified user QoS expectation while minimizing the device's energy consumption.

GreenWeb does not enforce any particular runtime implementation. As an example, the authors demonstrate one implementation using ACMP (asymmetric chip-multiprocessor) hardware. ACMP is an energy-efficient heterogeneous architecture that mobile hardware vendors such as ARM, Samsung, and Qualcomm have widely adopted—you probably have one in your pocket. Leveraging the language annotations as hints, the GreenWeb browser dynamically schedules execution on the ACMP hardware to achieve energy savings and prolong battery life.

Data Consciousness in Emerging Markets

Agababov, V., et al. 2015. Flywheel: Google's data compression proxy for the mobile web. Proceedings of the 12th Usenix Symposium on Networked Systems Design and Implementation; *http://research.google.com/pubs/pub43447.html.*

The mobile web is crucial in emerging markets. The first order of impedance for the mobile web in emerging markets is the high cost of data, more so than performance or energy efficiency. It is not uncommon for spending on mobile data to be more than half of an individual's income in developing countries. Therefore, reducing the amount of data transmitted is essential.

Flywheel from Google is a compression proxy system to make the mobile web conscious of data usage. Compression proxies to reduce data usage (and to improve latency) are not a new idea. Flywheel, however, demonstrates that while the core of the proxy server is compression, there are many design concerns to consider that demand a significant amount of engineering effort, especially to make such a system practical at Google scale. Examples of the design concerns include fault tolerance and availability upon request anomalies, safe browsing, robustness against middlebox optimizations, etc. Moreover, drawing from large-scale measurement results, the authors present interesting performance results that might not have been observable from small-scale experiments. For example, the impact of data compression on latency reduction is highly dependent on the user population, metric of interest, and web page characteristics.

Conclusion

We advocate addressing the RED challenge holistically. This will entail optimizations that span the different system layers synergistically. The three papers in our selection are a first step toward such cross-layer optimization efforts. With additional synergy we will likely uncover more room for optimization than if each of the layers worked in isolation. It is time that we as a community make the Web great again in the emerging era.

Jean Yang *is an assistant professor in the computer science department at Carnegie Mellon University. Her research interests are in programming language design and software*

verification applied to security, privacy, and biological modeling. She has interned at Google, Facebook, and Microsoft Research. In 2015 she cofounded the Cybersecurity Factory accelerator to bridge the gap between research and practice in cybersecurity.

Vijay Janapa Reddi *is an assistant professor in the department of electrical and computer engineering at the University of Texas at Austin. His research interests span the definition of computer architecture, including software design and optimization, to enhance the quality of mobile experience and improve the energy efficiency of high-performance computing systems. Reddi is a recipient of the National Academy of Engineering Gilbreth Lectureship honor (2016), IEEE Computer Society TCCA Young Computer Architect Award (2016), Intel Early Career Award (2013), and multiple Google Faculty Research Awards (2012, 2013, 2015). He is also the recipient of the Best Paper at the 2005 International Symposium on Microarchitecture, Best Paper at the 2009 International Symposium on High-Performance Computer Architecture, and IEEE's Top Picks in Computer Architecture awards (2006, 2010, 2011).*

Yuhao Zhu *is a Ph.D. candidate at the University of Texas at Austin. He likes building better software and hardware to make next-generation client and cloud computing fast and energy efficient, and deliver high quality of experience. His dissertation focuses on improving the energy efficiency of mobile web computing through a holistic approach* *spanning the processor architecture, web-browser runtime, programming language, and application layers. He received an M.S. from UT Austin in 2015 and a B.S. from Beihang University, China, in 2010. He is a Google Ph.D. Fellow (2016). His papers have been awarded Best of Computer Architecture Letters (2014) and IEEE MICRO Top Picks in Computer Architecture (Honorable Mention in 2016).*

CONTENTS

TEXT ONLY

React: Facebook's Functional Turn on Writing JavaScript

case study

A DISCUSSION WITH PETE HUNT, PAUL O'SHANNESSY, DAVE SMITH AND TERRY COATTA

One of the long-standing ironies of user-friendly JavaScript front ends is that building them typically involved trudging through the DOM (Document Object Model), hardly known for its friendliness to developers. But now developers have a way to avoid directly interacting with the DOM, thanks to Facebook's decision to open-source its React library for the construction of user interface components.

React essentially manages to abstract away the DOM, thus simplifying the programming model while also—in a somewhat surprising turn—improving performance. The key to both advances is that components built from standard JavaScript objects serve as the fundamental building blocks for React's internal framework, thus allowing for greatly simplified composability. Once developers manage to get comfortable with building front ends in this way, they typically find they can more readily see what's going on while also enjoying greater flexibility in terms of how they structure and display data.

All of which caused us to wonder about what led to the creation of React in the first place and what some of its most important guiding principles were. Fortunately for us, Pete Hunt, who at the time was an engineering manager at Instagram as well as one of the more prominent members of

Facebook's React core team, is willing to shed some light on React's beginnings. Hunt has since gone on to cofound Smyte, a San Francisco startup focused on security for marketplaces and social networks.

Also helping to tell the story is Paul O'Shannessy, one of the first engineers at Facebook to be dedicated to React full time. He came to that role from Mozilla, where he had previously worked on the Firefox front end.

The job of asking the probing questions that drive the discussion forward falls to Dave Smith and Terry Coatta. Smith is an engineering director at HireVue, a Salt Lake City company focused on team-building software, where he has had an opportunity to make extensive use of both Angular and React. Coatta is the CTO of Marine Learning Systems, where he is building a learning management system targeted at the maritime industry. He is also a member of the acmqueue *editorial board.*

DAVE SMITH What is it exactly that led to the creation of React?

PETE HUNT Of all the web apps at Facebook, one of the most complex is what we use to create ads and manage ad accounts. One of the biggest problems is keeping the UI in sync with both the business logic and the state of the application. Traditionally, we've done that by manually manipulating the DOM using a centralized event bus, whether by putting events into the queue or by having listeners for the event and then letting them do their thing.

That proved to be really cumbersome, so a few years ago we implemented what we then considered to be a state-of-the-art DOM-monitoring system called Bolt. It was kind of like Backbone with observables, where you would register for computed properties that would eventually get flushed to the DOM. But then we found that also was pretty hard to manage since you could never be sure when your properties were going to be updated—meaning that if you changed a value, you couldn't be sure whether it was going to cause a single update, cascading updates, or no updates at all. Figuring out when those updates might actually occur also proved to be a really hard problem.

The whole idea behind React initially was just to find some way to wire up those change handlers such that engineers could actually wrap their heads around them. That hadn't been the case with Bolt, and as a consequence we ended up with lots of bugs nobody could solve. So the engineers who started working on a way to remedy that ended up going wild for a couple of months and came out with this weird-looking thing nobody thought had any chance whatsoever of working. If you're even vaguely familiar with React, you already know that whenever there's a change in your underlying data model, it essentially re-renders the whole application and then does a diff to see what actually changed in the rendered result. Then it's only those parts of the page that get updated.

Some people here had some performance concerns about that, so an early version of React ended up being run through a serious gauntlet of engineering tests where it got benchmarked against pretty much everything that could be thrown at it. As part of that, of course, we looked at how this new programming model fared against both the Bolt model and our old event model. React ended up really surprising a lot of people—enough so, in fact, that it was shipped almost immediately as part of our "liking and commenting" interface on News Feed. That was the first big test for React, and that came a few years ago.

Then we tried it out on Instagram.com, which is where I entered the picture since I was the person at Instagram responsible for building a few things using React. We were really happy with it since it proved to be capable of running our whole page instead of just one small widget here or there. That gave us a pretty good indication it was actually going to work. Since then, it has essentially become the de facto way people write JavaScript at Facebook.

TERRY COATTA I've heard React takes a different approach to data binding. What sets React apart there?

PH The way I think about data binding in a web context is that you've got some sort of observable data structure down to the DOM nodes. The challenge is that when you're implementing some sort of observable system, you're obliged to observe this data structure wherever your application touches the data model.

For example, if you use something like Ember, everything you do is going to use getters and setters, meaning you're going to need to remain aware of this observable abstraction throughout the entire application. So if you want to compute a value, you're not going to use a function only; you're going to use a computed property number, which is a domain-specific thing for Ember.

Angular, I think, does a much better job of this since it uses dirty checking, which means you can actually take advantage of regular JavaScript objects. The problem with

Angular, though, is that it makes it difficult to compose your application. That's because, instead of using regular functions or objects to build up abstractions (as you would do with JavaScript), you have to pass everything through a scope in order to observe those changes. Then you end up with this data binding that couples different parts of your program in ways that aren't necessarily all that clear or obvious.

For example, let's say we're looking to sort a list of your top friends—which is the kind of thing we do all of the time here. In order for us to do that with an observable system, we would have to set up an observer for every one of the thousand friends you've listed, even if all we're really looking to do is to render the top ten. So, as you can imagine, it's going to take a good chunk of memory to maintain that whole representation.

Obviously, there are ways to get around that, but people typically just break out of the data binding abstraction altogether at that point so they can proceed manually. Now, I generally hate to say something isn't going to scale, but it's fairly obvious this is going to present some scaling issues. It's clear that the bigger your application gets, the more you're going to run into this sort of edge case.

TC I agree completely about the Angular situation since I also find composition there to be tricky for just the reason you mentioned—that is, you end up having different parts of your application essentially coupled silently via two-way data binding. But I see that React also has data binding, so I'm curious about how you've managed to provide for better composability despite that coupling.

PH Let me zoom out a little here to observe that, at a very high level, React essentially treats your user code as a black box while also taking in whatever data you tell it to accept. That basically allows for any structure. It could be something like Backbone. It could be plain JSON. It could be whatever you want. Then your code will just go ahead and do whatever it's supposed to do, backed by the full power of JavaScript.

At the end of that, however, it will return a value, which we call a virtual DOM data structure. That's basically just a fancy handle for JavaScript objects that tell you which kinds of elements they are and what their attributes are. So if you think of data binding as a way to keep your UI up to date with your underlying model, you can accomplish that with React just by signaling, "Hey, something in my data model may have just changed." That will prompt React to call the black-box user code, which in turn will emit a new virtual DOM representation. Then, having kept the previous representation, React will look at the new version and the old version and do a diff of the two. Based on that, it might conclude, "Oh, we need to build a className attribute at this node."

The advantage of this approach is that it involves no actual tracking of your underlying data model.

The advantage of this approach is that it involves no actual tracking of your underlying data model. You don't have to pay a data-binding cost up front. Most systems that require you to track changes within the data model and then keep your UI up to date with that are faced with a data-binding cost driven by the size of the underlying data model. React, on the other hand, pays that cost relative only to what actually gets rendered.

TC If I understand you correctly, you're saying React is in some sense a highly functional environment that takes some arbitrary input, renders an output, and then computes the difference between the two to determine what it ought to be displaying on the screen.

PH Exactly. I like to describe this as "referentially transparent UI." Which is to say your user interface is generally a pure function of some set of inputs, and it emits the same kind of virtual DOM structure every single time for some given data input.

TC So the data bindings that have caused us grief in Angular run in the other direction here in the sense that they reflect the value of DOM elements that are bound to underlying model objects or scope variables. Any changes there effectively become visible at multiple locations throughout your code at much the same time, meaning the composability issues surface since different locations in your code are made aware almost simultaneously of changes that propagate backwards from the UI.

PH Another problem is that you might have multiple bindings to the same data source. So then which piece of code is going to be treated as the authoritative source for determining what the value ought to be?

This is why, with React, we emphasize one-way data flow. As I said earlier, data in our model first goes into this application black box, which in turn emits a virtual DOM representation. Then we close the loop with simple browser events. We'll capture a KeyUp event and command, "Update the data model in this place based on that KeyUp event." We've also architected the system in such a way as to encourage you to keep the least possible mutable state in your application. In fact, because React is such a functional system, rather than computing a value and then storing it somewhere, we just recompute the value on demand with each new render.

The problem is that people sometimes want to have a big form that includes something like 20,000 fields that then bind to some simple keys and data objects. The good news is that it's actually very easy for us to build an abstraction on top of a simple event loop that basically captures all the events that might possibly update the value of this field, and then set up an automatic handler to pass the value down from the data model into the form field. The form and the data model essentially get updated at the same time. This means you end up with a system that looks a lot like data binding, but if you were to peel it back, you would see that it's actually only simple syntactic sugar on top of an event loop.

TC One of the things I've observed about React is that it seems to be what people would call fairly opinionated. That is, there's a certain way of doing things with React. This is in contrast to Angular, which I'd say is not opinionated since it generally lets you do things in several different ways. Do you think that is an accurate portrayal?

PH It depends. There are certain places where React is very opinionated and others where it's quite unopinionated. For example, React is unopinionated in terms of how you express your view logic since it treats your UI as a black box and looks only at the output. But it's opinionated in the sense that we really encourage idempotent functions, a minimal set of mutable state, and very clear state transitions.

I've built a lot of stuff with React, and I have a team that's run a lot of stuff with it. From all that experience, I can tell you that whenever you run into a bug in a React application, nine times out of ten you're going to find it's because you have too much state in there. We try to push as much mutable state as possible out of applications to get to what I like to call a fully normalized application state. In that respect, yes, we're very opinionated, but that's just because a lot of React abstractions don't work as well if you have too much mutable state.

I think Angular is actually less opinionated in that regard, but it certainly has opinions about how you need to compose your application. It's very much a model-view-presenter type of architecture. If you want to create reasonable widgets, you're going to have to use directives, which are very opinionated.

TC Another thing I noticed right away about React is that it's very component oriented. What was the reason for going in that direction?

PH We actually think of a component as being quite similar to a JavaScript function. In fact, the only difference between a function and a component is that components need to be aware of a couple of lifecycle hooks about themselves, since it's important they know when they get added to or removed from the DOM as well as when they're going to be able to get their own DOM node. The component is a fundamental building block on top of which we've built our own internal framework. Now a lot of other people out in the open-source world are also building on top of it.

We emphasize it because it's composable, which is the one thing that most separates React components from Angular directives and web components like partials and templates. This focus on composability—which I see as the ability to build nested components on multiple layers—not only makes it easier to see what's actually going on, but also gives you flexibility in terms of how to structure and display data, while also letting you override behaviors and pass data around in a more scalable and sensible way.

PAUL O'SHANNESSY This also has a lot to do with how we build applications on the server, where we have a core library of components that any product team can use as the basis for building their own components. This idea of using components is really just a natural extension of the core way we build things in PHP and XHP, with the idea simply being to compose larger and larger components out of smaller components.

PH Those product teams tend to be made up of generalists who work in all kinds of different languages, which is to say they're not necessarily experts in JavaScript or CSS (Cascading Style Sheets). We strongly discourage the average product engineer from writing much CSS. Instead, we suggest that they take these components off the shelf, drop them into whatever it is they're doing, and then maybe tweak the layout a little. That has worked really well for us.

PO That way we end up writing good code pretty much across the board since there are fewer people going off into crazy land writing CSS. Basically, this just gives us a way at the top level to control all that.

For all the ways in which React simplifies the creation of user interfaces, it also poses a learning curve for developers new to the environment. In particular, those who have worked primarily on monolithic systems in the past may find it challenging to adopt *more of a component-oriented mindset. They will also soon find that React is opinionated about how state should be handled, which can lead to some hard lessons and harsh reminders whenever people stray.*

Most of the pain points are almost certain to be about state. State is the hardest part of building applications anyway, and React just makes that super-explicit.

TC There's a lot about React that's appealing, but where are the sharp edges that people ought to look out for before diving in? What kinds of mistakes are likely to make their lives more painful?

PH Most of the pain points are almost certain to be about state. State is the hardest part of building applications anyway, and React just makes that super-explicit. If you don't think about state in the right way, React is going to highlight those bugs for you very early in the process.

TC Give me a concrete example of how people might think about state in the wrong way.

PH OK, I'm looking at a site powered by React that was launched earlier today. It looks like the page has four main components: side navigation, a search-results list, a search bar, and a content area containing both the search bar and the search-results list.

When you type in the search bar, it filters the results to be shown in the results grid. If I were to ask you where that filter state should live, there's a good chance you would think, "Well, the search-results list is what's doing the filtering, so the state probably ought to live there." That's what intuitively makes sense.

But actually the state should live in the common ancestor between the search box and the search-results list, sort of like a view controller. That's because the search box has the state of the search filter as well as the search results. Still, the search-results list needs access to that data as well. React will quickly let you know, “Hey, you actually need to put that in a common ancestor.”

PO If you were building that same UI with Angular and used a directive for the search box and then another directive for the search results, you would be encouraged in that case as well to put your state in a common ancestor. This would mean having a controller hold the scope variable, since that’s where you’ll find the search text to which both of those directives would then bind. I think you’re actually looking at a pretty similar paradigm there.

PH Good catch. But I think there’s still a distinction to be made in that React components are building blocks that can be used to construct a number of conceptually different components or objects. You could use a React component to implement a view controller or some pure view-only thing—whereas with Angular, the controller is distinct from a directive, which in turn is distinct from the “service,” which is how Angular describes those things you shove all the other logic into. Sometimes it makes sense just to make all those things React components.

DS In this case, if you were building the UI with React, what would be the common ancestor? A React component?

PH Yes. I would use React components for everything.

DS When I was starting out with React, I think one of the hardest things for me to grasp was this idea that everything is a component. Even when I walked through an example on the React website that included a comment box and a comment list, I was surprised to learn that even those were treated as components. I also found myself getting lost in the relationships between those components. I wonder if you find that to be a common problem for other new React developers.

PO For people who are used to building more monolithic things, that often proves to be a problem. At Facebook, where we've always coded in PHP, we're accustomed to building microcomponents and then composing them, so that hasn't proved to be such a huge problem here. Anyway, what I think we've always encouraged is that, whenever you're thinking about reusing something, break it down into its smallest elements. That's why, in the example you cited, you would want to separate the comment box from the comment list, since you can reuse both of those things in other parts of your application. We really do encourage people to think that way.

PH We also encourage that you make stuff stateless. Basically, I like to think people are going to feel really bad about using state. I know there are times when it's a necessary evil, but you should still feel dirty whenever you have to resort to doing that. That's because then you'll start thinking, "OK, so I really want to put this search state in only one place in my app." Generally, that means you'll find the right spot for it since you're not going to want to deal with having to synchronize states throughout your application. And you won't have to if it lives in only one canonical place.

DS What other major differentiators set React apart from other JavaScript frameworks?

PH We haven't yet talked about the idea that React, as a general way of expressing user interface or view hierarchies, treats the DOM as just one of many potential rendering back ends. It also renders to Canvas and SVG (Scalable Vector Graphics), for example. Among other things, this means React can render on the server without booting up like a full-browser DOM. It doesn't work like it's just some other domain-specific language on top of the DOM. Basically, React pretty much hates the DOM and wants to live outside a browser as much as possible. I definitely see that as a huge differentiator between React and the other JavaScript frameworks.

PO We've basically seen the same thing happen with WebGL or any other generic rendering platform. It just goes back to the question of immediate vs. retained mode, where you soon discover that as long as you can output something, it really doesn't matter. You just blow away whatever was there before.

DS I'm also curious about the functional programming aspects of React. In particular, I'm interested in knowing more about which specific functional principles you've adopted.

PH The truth is, we're actually a bunch of functional programming geeks. In part, that's because if you truly subscribe to the Church of Functional Programming, you can get a lot of performance benefits for free. For example, if your data model is serializable and you treat your render method as a pure function of your properties, you get server-side rendering and client-side rendering for free since both of those end up being pure functions of the same data on both sides of the wire. That way, you can guarantee that when your application initializes, it will get into the same state on both sides automatically. That

The truth is, we're actually a bunch of functional programming geeks. In part, that's because if you truly subscribe to the Church of Functional Programming, you can get a lot of performance benefits for free.

can be really important if you have a very stateful kind of object-oriented mutative system, since then it becomes much, much harder to synchronize those two states otherwise.

The other advantage has to do with optimizing your apps. We have a hook called Chute Component Update, where you can replace React's diff algorithm with a faster custom one. Also, many functional programmers really like to use immutable data structures since they let them quickly figure out whether something has changed—just another example of how you can get free performance benefits this way.

TC In the immutable data structures vein, one really powerful library I've heard about is David Nolen's Om.

PH That's a very cool piece of technology. It's for ClojureScript, the version of Clojure that compiles to JavaScript. What makes Clojure really cool is its persistent data structures, which basically are really fast and easy-to-use immutable data structures.

What that means for us is that if you have a post on Facebook and somebody likes it, that gives you a new *like* event that should be reflected on the like count appearing on that post. Normally, you would just mutate that, but then you would have no way of detecting whether the change actually happened or not, which means you would basically need to re-render the whole thing and then diff it. From that diff, you would learn that only that particular part of the UI actually changed. But if you were using immutable persistent data structures, instead of mutating the like count, you could just copy the story object and, within that copy, update the like count.

Normally, that would be a very expensive way to go, but in Clojure the copy isn't expensive since it has a way of doing it where it shares the pointers with all the other parts of that data structure and then allocates new objects only for whatever actually changed. That's a good example of an abstraction that's quite complicated under the hood and yet manages to present a very, very simple user interface—something that's extremely easy for people to reason about.

TC I assume that could also help with undo/redo capabilities.

PH Right. When everything is immutable, everything gets simpler. Om undos and redos basically just keep around pointers to the previous state and the next state. When you want to undo, you just pass the old object into React, and it will update the whole UI accordingly.

TC The whole thing?

PO When your state is serialized into one object at the top level, all you do is pass that through and re-render it—and you're done. With some of the Om examples I've seen, it just snapshots the state at every point and then gives you a UI that indicates how many states you have. Then you can just drag back and forth on that. Or you could start doing some fancier things with the help of trees to produce a really advanced undo system.

PO I should also point out that React clearly is not purely functional. We also have some very imperative steps and hooks that let you break out of the functional paradigm. But in an ideal world, you don't have any other sources of data, so everything is at the top and just flows through—

meaning everything ends up being a very pure output of these render functions.

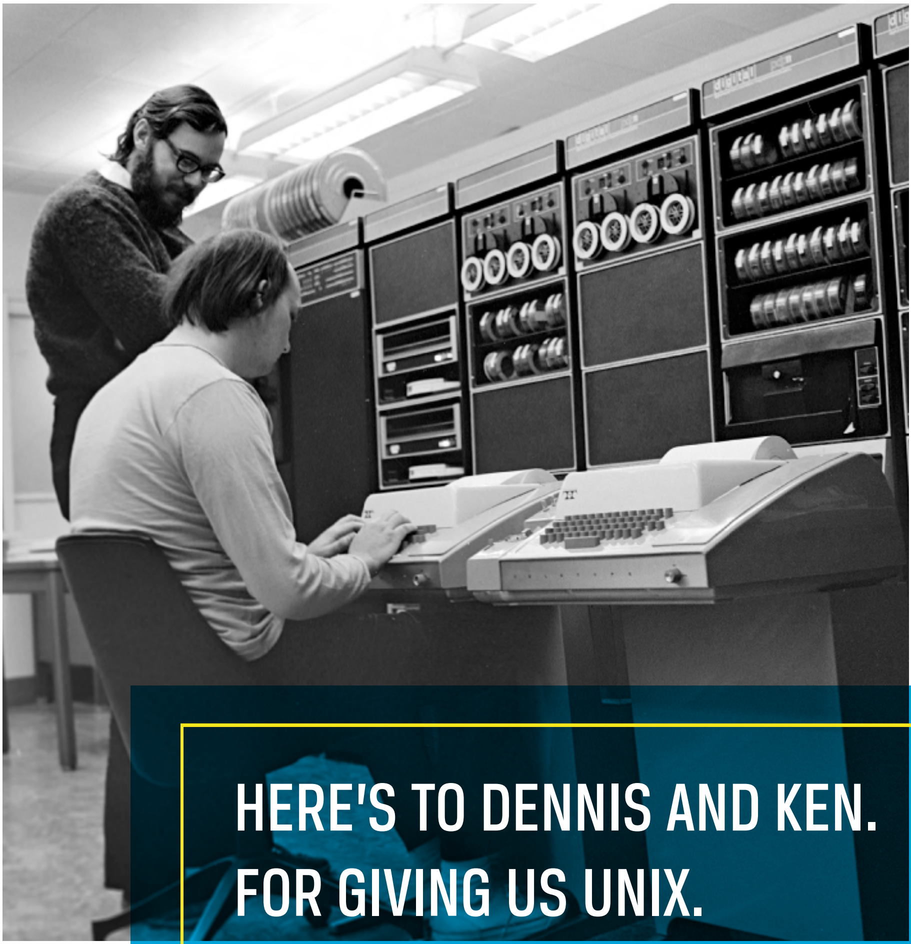
DS A bit earlier, you used the term "referential transparency" to describe the way React renders UI. Can you explain what that means?

PH Basically, React components have props, parameters that can be used to instantiate those components. You might think of them as function parameters. In order to say, "I want to create a type-ahead with these options," you can just pass in the options list as a prop.

The idea is that if you render a component using the same props and states, you'll always render the same user interface. This can get a little bit tricky, though. For example, you can't read from the random-number generator because that would change the output. Still, if you handle this as a pure function of props and state and make sure you don't read from anything else, you can probably see that this is going to make testing really fast and easy. You basically say, "I just want to make sure my component looks this certain way when it gets this data." Then, since you don't have to take the Web-driver approach of clicking on every single button to get the app into the right state before double-checking to make sure you've got everything right... well, it becomes pretty obvious how this makes testing a whole lot easier—which, of course, makes debugging easier as well.

CONTENTS

Credit: Reprinted with permission of Alcatel-Lucent USA Inc.



HERE'S TO DENNIS AND KEN.
FOR GIVING US UNIX.

We're more than computational theorists, database managers, UX mavens, coders and developers. We're on a mission to solve tomorrow. ACM gives us the resources, the access and the tools to invent the future. Join ACM today and receive 25% off your first year of membership.

BE CREATIVE. STAY CONNECTED. KEEP INVENTING.

ACM.org/KeepInventing

Association for Computing Machinery