

# acm **QUEUE**



Association for  
Computing Machinery

## Why Reactivity Matters

Idle-Time  
Garbage-Collection  
Scheduling

**Cluster-Level  
Logging  
of Containers  
with Containers**

The Hidden Dividends of  
**MICROSERVICES**

*Complete table of contents on the following two pages*

# CONTENTS

MAY-JUNE

# 2016

VOLUME 14 ISSUE 3

acmqueue

**RFP**

## **Distributed Consensus and Implications of NVM on Database Management Systems** 53

This second installment of Research for Practice looks at research in two critical areas in storage and large-scale services. First, Camille Fournier has selected papers that address distributed consensus systems, including two on implementing Paxos and one on the easier-to-understand Raft. Then Joy Arulraj and Andrew Pavlo share three papers detailing the future impact of nonvolatile memory on DBMS architectures.

BY PETER BAILIS, CAMILLE FOURNIER AND ANDREW PAVLO

## **FEATURES**

### **The Hidden Dividends of Microservices** 25

Though the effort is considerable, the benefits of adopting a microservices-based approach to building distributed systems may be well worth it. In addition to autonomy, agility, resilience, and developer productivity, there are some other unexpected dividends to be gained.

BY TOM KILLALEA

### **Idle-Time Garbage-Collection Scheduling** 35

Google's Chrome serves as an example of how to use the V8 JavaScript engine to reduce user-visible *jank* on real-world web pages by scheduling garbage-collection pauses during downtime.

BY ULAN DEGENBAEV  
JOCHEN EISINGER  
MANFRED ERNST  
ROSS MCILROY  
HANNES PAYER

### **Dynamics of Change: Why Reactivity Matters** 68

As software grows more complex, managing change in the code base becomes more of a challenge. Structuring the code into a system of modules and arrows helps make changes easier to maintain.

BY ANDRE MEDEIROS



# CONTENTS

## FEATURES

### **Cluster-Level Logging of Containers with Containers** 83

Implementing cluster-level log aggregation and inspection on the Kubernetes framework can overcome the challenges imposed by collecting the logs of containers running in an orchestrated cluster.

BY SATNAM SINGH

## COLUMNS / DEPARTMENTS

### **ESCAPING THE SINGULARITY**

*The Singular Success of SQL* 5

SQL may be aging, but it still has a brilliant future as a major figure in the growing pantheon of data representations.

BY PAT HELLAND

### **THE SOFT SIDE OF SOFTWARE**

*Bad Software Architecture is a People Problem* 13

Poor communication leads to bugs.

BY KATE MATSUDAIRA

### **KODE VICIOUS**

*What Are You Trying to Pull?* 19

Optimizing away a few instructions isn't worth it if it leads to a lack of code clarity.

BY GEORGE NEVILLE-NEIL

VOLUME 14 ISSUE 3

acmqueue

MARCH-APRIL

2016



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.

### QUEUE STAFF

EXECUTIVE EDITOR  
*James Maurer*

MANAGING EDITOR  
WEB EDITOR  
*Matt Slaybaugh*

COPY EDITORS  
*Susie Holly*  
*Vicki Rosenzweig*

ART DIRECTION  
*Reuter & Associates*

### CONTACT POINTS

feedback@  
queue.acm.org

editor@  
queue.acm.org

acmhelp@acm.org

http://queue.acm.org

**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

### EXECUTIVE DIRECTOR / CEO

*Bobby Schnabel*

### DEPUTY EXECUTIVE DIRECTOR / COO

*Patricia Ryan*

### ACM EXECUTIVE COMMITTEE

#### PRESIDENT

*Alexander L. Wolf*

#### VICE-PRESIDENT

*Vicki L. Hanson*

#### SECRETARY/TREASURER

*Erik Altman*

#### PAST PRESIDENT

*Vinton G. Cerf*

### PRACTITIONER BOARD

#### BOARD CHAIR

*George Neville-Neil*

#### BOARD MEMBERS

*Samy Al Bahra*

*Eve Andersson*

*Steve Bourne*

*Karin Breitman*

*Alain Chesnais*

*Terry Coatta*

*Ben Fried*

*Stephen Ibaraki*

*Erik Meijer*

*Theo Schlossnagle*

*Jim Waldo*

### QUEUE EDITORIAL BOARD

#### BOARD CHAIR / EDITOR-IN-CHIEF

*Stephen Bourne*

#### BOARD MEMBERS

*Eric Allman*

*Peter Bailis*

*Terry Coatta*

*Stuart Feldman*

*Camille Fournier*

*Benjamin Fried*

*Pat Hanrahan*

*Tom Killalea*

*Tom Limoncelli*

*Kate Matsudaira*

*Marshall Kirk McKusick*

*Erik Meijer*

*George Neville-Neil*

*Theo Schlossnagle*

*Jim Waldo*

*Meredith Whittaker*

### 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].

# The Singular Success of SQL

PAT HELLAND

**SQL HAS A  
BRILLIANT FUTURE  
AS A MAJOR  
FIGURE IN THE  
PANTHEON OF DATA  
REPRESENTATIONS**

**S**QL has been singularly successful in its impact on the database industry. Nothing has come remotely close to its ubiquity. Its success comes from its high-level use of relational algebra allowing set-oriented operations on data shaped as rows, columns, cells, and tables.

SQL's impact can be seen in two broad areas. First, the programmer can accomplish a lot very easily with set-oriented operations. Second, the high-level expression of the programmer's intent has empowered huge performance gains.

This column discusses how these features are dependent on SQL creating a notion of stillness through transactions and a notion of a tight group of tables with schema fixed at the moment of the transaction. These characteristics are what make SQL different from the increasingly pervasive distributed systems.

SQL has a brilliant past and a brilliant future. That future is not as the singular and ubiquitous holder of data but rather as a major figure in the pantheon of data representations. What the heck happens when data is not kept in SQL?

SQL: THE MIRACLE OF THE AGE, OF THE AGES,  
AND OF THE AGED

I launched my career in database implementation when

\*It's Not Your Grandmother's  
Database Anymore

Jimmy Carter was president. At the time, there were a couple of well-accepted representations for data storage: the *network* model was expressed in the CODASYL (Conference/Committee on Data Systems Languages) standard with data organized in sets having one set owner (parent) and multiple members (children); the *hierarchical* model ensured that all data was captured in a tree structure with records having a parent-child-grandchild relationship. Both of these models required the programmer to navigate from record to record.

Then along came these new-fangled relational things. INGRES (and its language QUEL) came from UC Berkeley. System-R (and its language SQL) came from IBM Research. Both leveraged relational algebra to support set-oriented abstractions allowing powerful access to data.

At first, they were really, really, really slow. I remember lively debates with database administrators who fervently believed they must be able to know the cylinder on disk holding their records! They most certainly did not want to change from their hierarchical and network databases. As time went on, SQL became inexorably faster and more powerful. Soon, SQL meant database and database meant SQL.

A funny thing happened by the early 2000s, though. People started putting data in places other than “the database.” The old-time database people (including yours truly) predicted their demise. Boy, were we wrong!

Of course, for most of us who had worked so hard to build transactional, relational, and strongly consistent systems, these new representations of data in HTML, XML, JSON, and other formats didn’t fit into our worldview. The

**T**he data and metadata (schema) must remain still while SQL does its thing.

radicals of the '70s and '80s became the fuddy-duddies of the '00s. A new schism had emerged.

## SQL, VALUES, AND RELATIONAL ALGEBRA

Relational databases have tables with rows and columns. Each column in a row provides a cell that is of a well-known type. DDL (Data Definition Language) specifies the tables, rows, and columns and can be dynamically changed at any time, transforming the shape of the data.

The fundamental principle in the relational model is that all interrelating is achieved by means of comparisons of values, whether these values identify objects in the real world or indicate properties of those objects. A pair of values may be meaningfully compared, however, if and only if these values are drawn from a common domain.

The stuff being compared in a query must have matching DDL or it doesn't make sense. SQL depends on its DDL being rigid for the duration of the query.

There's not really a notion of some portion of the SQL data having extensible metadata that arrives with the data. All of the metadata is defined before the query is issued. Extensible data is, by definition, not defined (at least at the receiver's system).

SQL's strength depends on a well-defined schema. Its set-oriented nature uses the well-defined schema for the duration of the operations. The data and metadata (schema) must remain still while SQL does its thing.

## THE STILLNESS AND ISOLATION OF TRANSACTIONS

SQL is set oriented. Bad stuff happens when the set of data

1

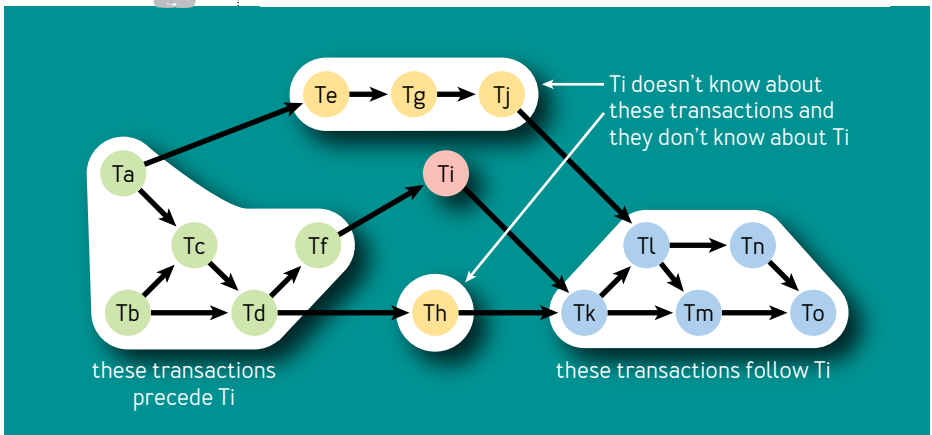
slides around during the computation. SQL is supposed to produce consistent results. Those consistent results are dependent on input data that *appears to be unchanging*.

Transactions and, specifically, transactional isolation provide the sense that nothing else is happening in the world.

The Holy Grail of transaction isolation is *serializability*. The idea is to make transactions *appear* as if they happened in a serial order. They don't actually have to occur in a serial order; it just has to seem like they do.

In figure 1, the red transaction  $T_i$  depends upon changes made by the green transactions ( $T_a$ ,  $T_b$ ,  $T_c$ ,  $T_d$ , and  $T_f$ ). The blue transactions ( $T_k$ ,  $T_l$ ,  $T_m$ ,  $T_n$ , and  $T_o$ ) depend on the changes made by  $T_i$ .  $T_i$  definitely is ordered after the green transactions and before the blue ones. It doesn't matter if any of the yellow transactions ( $T_e$ ,  $T_g$ ,  $T_j$ , and  $T_h$ ) occur

FIGURE 1: TRANSACTION SERIALIZABILITY





before or after  $T_i$ . There are many correct serial orders. What matters is that the concurrency implemented in the system provides a view that is serializable.

Suddenly, the world is still and set orientation can smile on it.

### A SENSE OF PLACE

SQL and its relational data are almost always kept inside a single system or a few systems close to each other. Each SQL database is almost always contained within a trust boundary and protected by surrounding application code.

I don't know of any systems that allow untrusted third parties to access their back-end databases. My bank's ATM, for example, has never let me directly access its back-end database with JDBC (Java Database Connectivity). So far, the bank has constrained me to a handful of operations such as deposit, withdrawal, or transfer. It's really annoying! In fact, I can't think of any enterprise databases that allow untrusted third parties to "party" on their databases. All of them insist on using application code to mitigate the foreigners' access to the system.

Interactions occur across these systems, but they are implemented with some messages or other data exchange that is loosely coupled to the underlying databases on each side. The messages hit the application code and not the database.

Each of these databases appears to be an island unto itself. Now, that island may have a ferry or even a four-lane bridge connecting it to other islands. Still, you always know the island upon which you stand when you access a database.

**D**istributed transactions across different SQL databases are rare and challenging.

## DIFFERENT PLACES MEANS DIFFERENT TIMES

Multiple databases sharing a transactional scope is extremely rare. When a transaction executes on a database, there is no clear and crisp notion of its time when compared with the time on another system's database. Distributed transactions across different SQL databases are rare and challenging.

If you assume two databases do not share a transactional scope, then the simple act of spreading work across space (the databases) implies spreading the work across time (multiple transactions). This transition from one database to more than one database is a seminal semantic shift. Space and time are intrinsically tied to each other.

When you pop from one to many, SQL and its relational algebra cannot function without some form of restriction. The most common form is to cast some of the data into immutable views that can be meaningfully queried over time. The system projecting these views won't change them. When they don't change, you can use them across space.

Freezing data in time allows its use across spatial boundaries. Typically, you also project the data to strip out the private stuff as you project across trust boundaries. To span spatial boundaries, time must freeze, at least for the data being shared. When you freeze data, it's immutable.

## IMMUTABILITY: THE ONE CONSTANT OF DISTRIBUTED SYSTEMS

Immutable data can be immortal and omnipresent. Think of it as the Gideon Bible, which seems to be in every hotel room; I suspect there will be Bibles there for a long time. If you want to do a query leveraging the Gideon Bible as an

input, you will not struggle with challenges of concurrency or isolation. It's relatively straightforward to cache a copy close to where you need it, too.

SQL's relational operations can be applied to immutable data at a massive scale because the metadata is immutable and the data is immutable. This empowers MapReduce, Hadoop, and the other big-data computation. By being immutable, the contents are still and the set-oriented computations make sense.

Immutable data can be everywhere at any time. That allows it to be both inside the singularity and outside of it. No big deal. Immutability truly is one of the unifying forces of distributed systems.

Classic centralized databases force their data to appear immutable using transactions. When distribution impedes the use of transactions, you snapshot a subset of your data so it can be cast across the boundaries with predictable behavior.

## ESCAPING THE SINGULARITY

SQL databases are phenomenally powerful and have enjoyed singular success in providing access to and control over data. They allow the combination and analysis of data by leveraging relational algebra. Relational algebra relates values contained in the rows and the columns of its tables. This has provided incredible power in programming and huge performance gains in accessing relational data.

To do this, relational algebra requires a static set of tables unmolested by concurrent changes. Both the data and the schema for the data must be static while operations are performed. This is achieved with

transactional serializability or other slightly weaker isolation policies. Serializability provides the illusion that each user of the database is alone at a *single point in time*.

In a relational database, it is hard to provide full functionality when distributed except, perhaps, across a handful of machines in close proximity. Even more profoundly, SQL works well within a single trust boundary such as a department or a company. SQL databases provide the illusion that they exist at a *single point in space*.

Providing a single point in space and time yields both stillness and isolated location. This empowers the value-based comparisons of relational algebra. It looks just like a *singularity*.

The industry has leapt headlong toward data representations that are neither bound to a single point in time nor to a single point in space with distributed, heterogeneous, and loosely coupled systems. Nowadays, far more data is being generated outside the SQL environment than within it. This trend is accelerating.

This column explores various consequences of *escaping the singularity* and relaxing the constraints of both space and time. No, it ain't your grandmother's database anymore.

**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.*

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.



# Bad Software Architecture is a People Problem

**WHEN PEOPLE  
DON'T WORK  
WELL TOGETHER  
THEY MAKE BAD  
DECISIONS**

KATE MATSUDAIRA

**I**t all started with a bug.

Customers were complaining that their information was out of date on the website. They would make an update and for some reason their changes weren't being reflected. Caching seemed like the obvious problem, but once we started diving into the details, we realized it was a much bigger issue.

What we discovered was the back-end team managing the APIs and data didn't see eye-to-eye with the front-end team consuming the data. The back-end team designed the APIs the way they thought the data should be queried—one that was optimized for the way they had designed the schema. The challenge was that when the front-end team wrote the interface, the API seemed clunky to them—there were too many parameters, and they had to make too many calls. This negatively impacted the mobile experience, where browsers can't handle as many concurrent requests, so the front-end team made the decision to cache part of the data locally.

The crux of the issue



**T**he challenge is that if you want your software to last, uniformity and predictability are good things—unique snowflakes are not.

was that the teams had not communicated well with each other. Neither team had taken the time to understand the needs of the other team. The result was a weird caching bug that affected the end user.

You might be thinking this could never happen on your team, but the reality is that when many different people are working on a problem, each could have a different idea about the best solution. And when you don't have a team that works well together, it can hurt your software design, along with its maintainability, scalability, and performance.

Most software systems consist of parts and pieces that come together to perform a larger function. Those parts and pieces can be thought out and planned, and work together in a beautiful orchestra. Or they can be designed by individuals, each one as unique as the person who created it. The challenge is that if you want your software to last, uniformity and predictability are good things—unique snowflakes are not.

One of the challenges of managing a software team is balancing the knowledge levels across your staff. In an ideal world, every employee would know enough to do his or her job well, but the truth is, in larger software teams there is always someone getting up to speed on something: a new technology, a way of building software, or even the way your systems work. When someone doesn't know something well enough to do a great job, there is a knowledge gap, and this is pretty common.

When building software and moving fast, people don't always have enough time to learn everything they need to

bridge their gaps. So each person will make assumptions or concessions that can impact the effectiveness of any software that individual works on.

For example, an employee may choose a new technology that hasn't been road tested enough in the wild, and that later falls apart under heavy production load. Another example is someone writing code for a particular function, without knowing that code already exists in a shared library written by another team—reinventing the wheel and making maintenance and updates more challenging in the future.

On larger teams, one of the common places these knowledge gaps exist is between teams or across disciplines: for example, when someone in operations creates a Band-Aid in one area of the system (like repetitively restarting a service to fix a memory leak), because the underlying issue is just too complex to diagnose and fix (the person doesn't have enough understanding of the running code to fix the leaky resources).

Every day, people are making decisions with imperfect knowledge. The real question is, how can you improve the knowledge gaps and enable your team to make better decisions?

Here are a few strategies that can help your team work better, and in turn help you create better software. While none of these strategies is a new idea, they are all great reminders of ways to make your teams and processes that much better.

**O**ne of the most important strategies is to think about how you will truly test the end-to-end functionality of a system.

### Define how you will work together

Whether you are creating an API or consuming someone else's data, having a clearly defined contract is the first step toward a good working relationship. When you work with another service it is important to understand the guardrails and best practices for consuming that service. For example, you should establish the payload maximums and discuss the frequency and usage guidelines. If for some reason the existing API doesn't meet your needs, instead of just working around it, talk about why it isn't working and collaboratively figure out the best way to solve the problem (whether that is updating the API or leveraging a caching strategy). The key here is communication.

### Decide how you will test the whole system

One of the most important strategies is to think about how you will truly test the end-to-end functionality of a system. Having tests that investigate only *your* parts of the system (like the back-end APIs) but not the end-customer experience can result in uncaught errors or issues (such as my opening example of caching). The challenge then becomes, who will own these tests? And who will run these tests and be responsible for handling failures? You may not want tests for every scenario, but certainly the most important ones are worth having.

### When bugs happen, work together to solve them

When problems arise, try to avoid solutions that only mask the underlying issue. Instead, work together to figure out what the real cause of the problem is, and then make a decision as a team on the best way of addressing it going forward. This way the entire team can learn more about



how the systems work, and everyone involved will be informed of any potential Band-Aids.

### Use versioning

When another team consumes something you created (an API, a library, a package), versioning is the smartest way of making updates and keeping everyone on the same page with those changes. There is nothing worse than relying on something and having it change underneath you. The author may think the changes are minor or innocuous, but sometimes those changes can have unintended consequences downstream. By starting with versions, it is easy to keep everyone in check and predictably manage their dependencies.

### Create coding standards

Following standards can be really helpful when it comes to code maintenance. When you depend on someone else and have access to that source code, being able to look at it—and know what you are looking at—can give you an edge in understanding, debugging, and integration. Similarly, in situations where styles are inherited and reused throughout the code, having tools like a style guide can help ensure that the user interfaces look consistent—even when different teams throughout the company develop them.

### Do code reviews

One of the best ways of bridging knowledge gaps on a team is to encourage sharing among team members. When other people review and give feedback, they learn

the code, too. This is a great way of spreading knowledge across the team.

Of course, the real key to great software architecture for a system developed by lots of different people is to have great communication. You want everyone to talk openly to everyone else, ask questions, and share ideas. This means creating a culture where people are open and have a sense of ownership—even for parts of the system they didn't write.

**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 (<https://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](http://katemats.com).

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.



# Chilling the Messenger

## KEEPING EGO OUT OF SOFTWARE-DESIGN REVIEW

by [@kylemcdonald](#)

who is  
**KV?**



click for video



Dear KV,

I was recently hired as a midlevel web developer working on version 2 of a highly successful but outdated web application. It will be implemented with ASP.Net WebAPI. Our architect designed a layered architecture, roughly like Web Service > Data Service > Data Access. He noted that data service should be agnostic to Entity Framework ORM (object-relational mapping), and it should use unit-of-work and repository patterns. I guess my problem sort of started there.

Our lead developer has created a solution to implement the architecture, but the implementation does not apply the unit-of-work and repository patterns correctly. Worse, the code is really hard to understand and it does not actually fit the architecture. So I see a lot of red flags coming up with this implementation. It took me almost an entire weekend to work through the code, and there are still gaps in my understanding.

This week our first sprint starts, and I feel a responsibility to speak up and try to address this issue. I know that I will face a lot of resistance, just based on the fact that the lead developer wrote that code and understands it more than the alternatives. He may not see the issue that I will try to convey. I need to convince him and the rest of the team that the code needs to be refactored or reworked. I feel apprehensive, because I am

like the new kid on the block trying to change the game. I also don't want to be perceived as Mr. Know-It-All, even though I might be a little more opinionated than I should be sometimes.

My question is, how can I convince the team that there is a real problem with the implementation without offending anyone?

~Opinionated

Dear ~Opinionated,

Let me work backwards through your letter from the end. You are asking me, Kode Vicious, how to point out problems without offending anyone? Have you read any of my previous columns? Let's just start out with the KV ground rules: it's only the law and other deleterious side effects that keep me on the "right" side of violence in some meetings. I'd like to think a jury of my peers would acquit me should I eventually cross to the wrong side, but I don't want to stake my freedom on that. I will try my best to give you solutions that do not land you in jail, but I will not guarantee them not to offend.

Trying to correct someone who has just done a lot of work, even if, ultimately, that work is not the right work, is a daunting task. The person in question no doubt believes that he has worked very hard to produce something of value to the rest of the team, and walking in and spitting on it, literally or metaphorically, probably crosses your "offense" line—at least I think it does. I'm a bit surprised that since this is the first sprint and there is already so

**In order to become more comfortable with the system, there are two things to call for: a design review and a code review.**

much code written, shouldn't the software have shown up after the sprints established what was needed, who the stakeholders were, etc.? Or was this a piece of previously existing code that was being brought in to solve a new problem? It probably doesn't matter, because the crux of your letter is the fact that you and your team do not sufficiently understand the software in question to be comfortable with fielding it.

In order to become more comfortable with the system, there are two things to call for: a design review and a code review. These are not actually the same things, and KV has already covered how to conduct a code review ["Kode Reviews 101." *Communications of the ACM* 52(10): 28-29. (October 2009)]. Let's talk now about a design review.

A software design review is intended to answer a basic set of questions:

1. How does the design take inputs and turn them into outputs?
2. What are the major components that make up the system?
3. How do the components work together to achieve the goals set out by the design?

That all sounds simple, but the devil is in the level of the details. Many software developers and systems architects would prefer that everyone but themselves see the systems they have built as black boxes, where data goes in and other data comes out, no questions asked. You clearly do not have the necessary level of trust with the software you're working with to allow the lead developer to get away with that, so you should call for a design review

where you take the lid off the box and poke around at the parts inside. In fact, questions 2 and 3 are going to be your main tools for figuring out what the software does and whether or not it is suitable for the task.

When I have to interview people for jobs, I always ask them questions about systems they have worked on while we draw out the block diagram on a whiteboard: What are the major components? How does component A talk to component B? What happens if C fails? I'm trying to transfer their mental images of their software into my own mind, of course without either going mad or having a nasty flashback. Some pieces of software are best left outside your mind, but hopefully that's not going to be the case with the system you're working with.

Remember that every box that this person draws can be opened if you think you're not getting sufficient detail. Much like the ancient game show, "Let's Make a Deal," it is always OK for you to ask, "What's behind door number 1, Monty?" Of course, you might find that it's a goat, but hopefully you find that it's a working set of components that are understandable to you and the team.

The one thing *not* to do in a design review is turn it into a code review. You are definitely not interested in the internals of any of the algorithms, at least not yet. The only code you might want to look at are the APIs that glue the components together, but even these are best left abstract, so that the amount of detail does not overwhelm you. Remember that the goal is always to get the big picture rather than the fine details, at least in a design review.

Coming back to the question of offense, I have found only one legal way to avoid giving offense, and that is always

**T**he Socratic method can be applied in an annoyingly pedantic way, but since you're trying not to give offense, I suggest that you play by a few useful rules.

to phrase things as questions. Often called the Socratic method, this can be a good way to get people to explain to you, and often to themselves, what they think they are doing. The Socratic method can be applied in an annoyingly pedantic way, but since you're trying not to give offense, I suggest that you play by a few useful rules. First, do not hammer the person with a relentless list of questions right off. Remember that you are trying to explore the design space in a collaborative way; this is not an interrogation. Second, leave spaces for the people you're working with to think. A pause doesn't mean they don't know; in fact, it might be that they're trying to adjust their mental model of the system in a way that will be beneficial to everyone when the review is done. Lastly, try to vary the questions you ask and the words you use. No one wants to be subjected to a lot of, "And then what happens?"

Finally, I find that when I'm in a design review and about to do something that might give offense, such as throwing a chair or a whiteboard marker, I try to do something less obvious. My personal style is to take off my glasses, put them on the table and speak in a very calm voice. That usually doesn't offend, but it does get people's attention, which leads them to concentrate harder on working to understand the problem we're all trying to solve.

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.*

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.

**SHAPE THE FUTURE OF COMPUTING!**

Join ACM today at [acm.org/join](http://acm.org/join)

**BE CREATIVE.  
STAY CONNECTED.  
KEEP INVENTING.**



Association for  
Computing Machinery

*Advancing Computing as a Science & Profession*



# The Hidden Dividends of Microservices

**MICROSERVICES  
AREN'T FOR  
EVERY COMPANY,  
AND THE JOURNEY  
ISN'T EASY**

TOM KILLALEA

**M**icroservices are an approach to building distributed systems in which services are exposed only through hardened APIs; the services themselves have a high degree of internal cohesion around a specific and well-bounded context or area of responsibility, and the coupling between them is loose. Such services are typically simple, yet they can be composed into very rich and elaborate applications. The effort required to adopt a microservices-based approach is considerable, particularly in cases that involve migration from more monolithic architectures. The explicit benefits of microservices are well known and numerous, however, and can include increased agility, resilience, scalability, and developer productivity. This article identifies some of the hidden dividends of microservices that implementers should make a conscious effort to reap.

The most fundamental of the benefits driving the momentum behind microservices is the clear separation of concerns, focusing the attention of each service upon

some well-defined aspect of the overall application. These services can be composed in novel ways with loose coupling between the services, and they can be deployed independently. Many implementers are drawn by the allure of being able to make changes more frequently and with less risk of negative impact. Robert C. Martin described the *single responsibility principle*: “Gather together those things that change for the same reason. Separate those things that change for different reasons.”<sup>5</sup> Clear separation of concerns, minimal coupling across domains of concern, and the potential for a higher rate of change lead to increased business agility and engineering velocity.

Martin Fowler argues that the adoption of continuous delivery and the treatment of infrastructure as code are more important than moving to microservices, and some implementers adopt these practices on the way to implementing microservices, with positive effects on resilience, agility, and productivity. An additional key benefit of microservices is that they can enable owners of different parts of an overall architecture to make very different decisions with respect to the hard problems of building large-scale distributed systems in the areas of persistence mechanism choices, consistency, and concurrency. This gives service owners greater autonomy, can lead to faster adoption of new technologies, and can allow them to pursue custom approaches that might be optimal for only a few or even for just one service.

## THE DIVIDENDS

While difficult to implement, a microservices-based approach can pay dividends to the organization that takes

**A**n organization that has embraced permissionless innovation should have a high rate of experimentation and a low rate of cross-team meetings

the trouble, though some of the benefits are not always obvious. What follows is a description of a few of the less obvious ones that may make the adoption of microservices worth the effort.

### Dividend #1: Permissionless Innovation

Permissionless innovation is about “the ability of others to create new things on top of the communications constructs that we create,”<sup>1</sup> as put forth by Jari Arkko, chair of IETF (Internet Engineering Task Force). When enabled, it can lead to innovations by consumers on a set of interfaces that the designers of those interfaces might find surprising and even bewildering. It contrasts with approaches where *gatekeepers* (a euphemism for *blockers*) have to be consulted before an integration can be considered.

To determine whether permissionless innovation has been unleashed to the degree possible, a simple test is to look at the prevalence of meetings *between* teams (as distinct from *within* teams). Cross-team meetings suggest coordination, coupling, and problems with the granularity or functionality of service interfaces. Engineers don’t seek out meetings if they can avoid them; such meetings could mean that a service’s APIs aren’t all that is needed to integrate. An organization that has embraced permissionless innovation should have a high rate of experimentation and a low rate of cross-team meetings.

### Dividend #2: Enable Failure

It should come as no surprise to hear that in computer science, we still don’t know how to build complex systems

that work reliably,<sup>6</sup> and the unreliability of systems increases with size and complexity. While opinions differ as to whether microservices allow a reduction in overall complexity, it's worth embracing the notion that microservices will typically increase the number of failures. Further, failures across service boundaries will be more difficult to troubleshoot since external call stacks are inherently more fragile than internal ones, and the debugging task is limited by poorer tooling and by more challenging ad hoc analysis characteristics. This tweet by @Honest\_Update can sometimes feel uncomfortably accurate: "We replaced our monolith with micro services so that every outage could be more like a murder mystery."<sup>4</sup>

Designing for the inevitability and indeed the routineness of failure can lead to healthy conversations about state persistence, resilience, dependency management, shared fate, and graceful degradation. Such conversations should lead to a reduction of the blast radius of any given failure by leveraging techniques such as caching, metering, traffic engineering, throttling, load shedding, and back-off. In a mature microservices-based architecture, failure of individual services should be expected, whereas the cascading failure of all services should be impossible.

### **Dividend #3: Disrupt Trust**

In small companies or in small code bases, some engineers may have a strong sense of trust in what's being deployed because they look over every shoulder and review every commit. As team size and aggregate velocity increase,

**“Any organization that designs a system will inevitably produce a design whose structure is a copy of the organization’s communication structure.”**

**—MELVIN CONWAY**

“Dunbar’s number” takes effect, leading to such trust becoming strained. As defined by British anthropologist Robin Dunbar, this is the maximum number of individuals with whom one can maintain social relationships by personal contact.

A move to microservices can force this expectation of trust to surface and be confronted. The boundary between one service and another becomes a set of APIs. The consumer gives up influence over the design of what lies behind those APIs, how that design evolves, and how its data persists, in return for a set of SLAs (service-level agreements) governing the stability of the APIs and their runtime characteristics. Trust can be replaced with a combination of autonomy and accountability.

As stated by Melvin Conway, who defined what is now known as Conway’s law: “Any organization that designs a system will inevitably produce a design whose structure is a copy of the organization’s communication structure.”<sup>2</sup>

Microservices can provide an effective model for evolving organizations that scale far beyond the limits of personal contact.

#### **Dividend #4: You Build It, You Own It**

Microservices encourage the “you build it, you own it” model. Amazon CTO Werner Vogels described this model in a 2006 conversation with Jim Gray that appeared in *ACM Queue*: “Each service has a team associated with it, and that team is completely responsible for the service—from scoping out the functionality, to architecting it, to building it, and operating it. You build it, you run it. This brings developers into contact with the day-to-day

operation of their software. It also brings them into day-to-day contact with the customer. The customer feedback loop is essential for improving the quality of the service.”<sup>3</sup>

In the decade since that conversation, as more software engineers have followed this model and taken on responsibility for the operation as well as the development of microservices, they’ve driven broad adoption of a number of practices that enable greater automation and that lower operational overhead. Among these are continuous deployment, virtualized or containerized capacity, automated elasticity, and a variety of self-healing techniques.

### **Dividend #5: Accelerate Deprecations**

In a monolith, it’s difficult to deprecate anything safely. With microservices, it’s easy to get a clear view of a service’s call volume, to stand up different and potentially competing versions of a service, or to build a new service that shares nothing with the old service other than backwards compatibility with those interfaces that consumers care about the most.

In a world of permissionless innovation, services can and should routinely come and go. It’s worth investing some effort to make it easier to deprecate services that haven’t meaningfully caught on. One approach to doing this is to have a sufficiently high degree of competition for resources so that any resource-constrained team that is responsible for a languishing service is drawn to spending most of their time on other services that matter more to customers. As this occurs, responsibility for the unsuccessful service should be transferred to the

consumer who cares about it the most. This team may rightfully consider themselves to have been left “holding the can,” although the deprecation decision also passes into their hands. Other teams that wish not to be left holding the can have an added incentive to migrate or terminate their dependencies. This may sound brutal, but it’s an important part of “failing fast.”

#### **Dividend #6: End Centralized Metadata**

In Amazon’s early years, a small number of relational databases were used for all of the company’s critical transactional data. In the interest of data integrity and performance, any proposed schema change had to be reviewed and approved by the DB Cabal, a gatekeeping group of well-meaning enterprise modelers, database administrators, and software engineers. With microservices, consumers shouldn’t know or care about how data persists behind a set of APIs on which they depend, and indeed it should be possible to swap out one persistence mechanism for another without consumers noticing or needing to be notified.

#### **Dividend #7: Concentrate the Pain**

A move to microservices should enable an organization to take on very different approaches to the governance expectations that it has of different services. This will start with a consistent companywide model for data classification and with the classification of the criticality of the integrity of different business processes. This will typically lead to threat modeling for the services that handle the most important data and processes,

and the implementation of the controls necessary to serve the company's security and compliance needs. As microservices proliferate, it can be possible to ensure that the most severe burden of compliance is concentrated in a very small number of services, releasing the remaining services to have a higher rate of innovation, comparatively unburdened by such concerns.

### Dividend #8: Test Differently

Engineering teams often view the move to microservices as an opportunity to think differently about testing. Frequently, they'll start thinking about how to test earlier in the design phase, before they start to build their service. A clearer definition of ownership and scope can provide an incentive to achieve greater coverage. As stated by Yelp in setting forth its service principles, "Your interface is the most vital component to test. Your interface tests will tell you what your client actually sees, while your remaining tests will inform you on how to ensure your clients see those results."<sup>7</sup>

The adoption of practices such as continuous deployment, smoke tests, and phased deployment can lead to tests with higher fidelity and lower time-to-repair when a problem is discovered in production. The effectiveness of a set of tests can be measured less by their rate of problem detection and more by the rate of change that they enable.

### WARNING SIGNS

The following indicators are helpful in determining that the



journey to microservices is incomplete. You're probably not doing microservices if:

- Different services do coordinated deployments.
- You ship client libraries.
- A change in one service has unexpected consequences or requires a change in other services.
- Services share a persistence store.
- You cannot change your service's persistence tier without anyone caring.
- Engineers need intimate knowledge of the designs and schemas of other teams' services.
- You have compliance controls that apply uniformly to all services.
- Your infrastructure isn't programmable.
- You can't do one-click deployments and rollbacks.

## CONCLUSION

Microservices aren't for every company, and the journey isn't easy. At times the discussion about their adoption has been effusive, focusing on autonomy, agility, resilience, and developer productivity. The benefits don't end there, however, and to make the journey worthwhile, it's important to reap the additional dividends.

## References

1. Arkko, J. 2013. Permissionless innovation. IETF; <https://www.ietf.org/blog/2013/05/permissionless-innovation/>.
2. Conway, M. E. 1968. How do committees invent? *Datamation Magazine*; [http://www.melconway.com/Home/Committees\\_Paper.html](http://www.melconway.com/Home/Committees_Paper.html).

3. Gray, J. 2006. A conversation with Werner Vogels. *ACM Queue* 4(4); <http://queue.acm.org/detail.cfm?id=1142065>.
4. Honest Status Page. 2015. @honest\_ update; [https://twitter.com/honest\\_update/status/651897353889259520](https://twitter.com/honest_update/status/651897353889259520).
5. Martin, R. C. 2014. The single responsibility principle; <https://blog.8thlight.com/uncle-bob/2014/05/08/SingleResponsibilityPrinciple.html>.
6. Perera, D. 2015. The crypto warrior. Politico; <http://www.politico.com/agenda/story/2015/12/crypto-war-cyber-security-encryption-000334>.
7. Yelp service principles; <https://github.com/Yelp/service-principles>.

**Tom Killalea** was with Amazon for 16 years and now consults and sits on several company boards, including those of Capital One, ORRECO, and MongoDB.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.

# Idle-Time Garbage-Collection Scheduling

ULAN DEGENBAEV  
JOCHEN EISINGER  
MANFRED ERNST  
ROSS MCILROY  
HANNES PAYER

## TAKING ADVANTAGE OF IDLENESS TO REDUCE DROPPED FRAMES AND MEMORY CONSUMPTION

Google's Chrome web browser strives to deliver a smooth user experience. An animation will update the screen at 60 FPS (frames per second), giving Chrome around 16.6 milliseconds to perform the update. Within these 16.6 ms, all input events have to be processed, all animations have to be performed, and finally the frame has to be rendered. A missed deadline will result in dropped frames. Such are visible to the user and degrade the user experience. These sporadic animation artifacts are referred to here as *jank*.<sup>3</sup>

JavaScript, the lingua franca of the web, is typically used to animate web pages. It is a garbage-collected programming language where the application developer does not have to worry about memory management. The garbage collector interrupts the application to pass over the memory allocated by the application, determine live memory, free dead memory, and compact memory by moving objects closer together. While some of these garbage-collection phases can be performed in parallel

or concurrently to the application, others cannot, and as a result they may cause application pauses at unpredictable times. Such pauses may result in user-visible jank or dropped frames; therefore, we go to great lengths to avoid such pauses when animating web pages in Chrome.

This article describes an approach implemented in the JavaScript engine V8, used by Chrome, to schedule garbage-collection pauses during times when Chrome is idle.<sup>1</sup> This approach can reduce user-visible jank on real-world web pages and results in fewer dropped frames.

## GARBAGE COLLECTION IN V8

Garbage-collector implementations typically optimize for the *weak generational hypothesis*,<sup>6</sup> which states that most of the allocated objects in applications die young. If the hypothesis holds, garbage collection is efficient and pause times are low. If it does not hold, pause times may lengthen.

V8 uses a generational garbage collector, with the JavaScript heap split into a small young generation for newly allocated objects and a large old generation for long-living objects. Since most objects typically die young, this generational strategy enables the garbage collector to perform regular, short garbage collections in the small young generation, without having to trace objects in the large old generation.

The young generation uses a semi-space allocation strategy, where new objects are initially allocated in the young generation's active semi-space. Once a semi-space becomes full, a scavenge operation will trace through the live objects and move them to the other semi-space.

**A**fter the live objects have been moved, the new semi-space becomes active and any remaining dead objects in the old semi-space are discarded without iterating over them.

Such a semi-space scavenge is a *minor garbage collection*. Objects that have already been moved in the young generation are promoted to the old generation. After the live objects have been moved, the new semi-space becomes active and any remaining dead objects in the old semi-space are discarded without iterating over them.

The duration of a minor garbage collection therefore depends on the size of the live objects in the young generation. A minor garbage collection is typically fast, taking no longer than one millisecond when most of the objects become unreachable in the young generation. If most objects survive, however, the duration of a minor garbage collection may be significantly longer.

A major garbage collection of the whole heap is performed when the size of live objects in the old generation grows beyond a heuristically derived memory limit of allocated objects. The old generation uses a mark-and-sweep collector with compaction. Marking work depends on the number of live objects that have to be marked, with marking of the whole heap potentially taking more than 100 ms for large web pages with many live objects.

To avoid such long pauses, V8 marks live objects incrementally in many small steps, pausing only the main thread during these marking steps. When incremental marking is completed the main thread is paused to finalize this major collection. First, free memory is made available for the application again by sweeping the whole old-generation memory, which is performed concurrently by dedicated sweeper threads. Afterwards, the young generation is evacuated, since we mark through the young generation and have liveness information. Then memory

compaction is performed to reduce memory fragmentation in old-generation pages. Young-generation evacuation and old-generation compaction are performed by parallel compaction threads. After that, the object pointers to moved objects in the remembered sets are updated in parallel. All these finalization tasks occur in a single atomic pause that can easily take several milliseconds.

#### THE TWO DEADLY SINS OF GARBAGE COLLECTION

**T**he garbage-collection phases outlined here can occur at unpredictable times, potentially leading to application pauses that impact the user experience. Hence, developers often become creative in attempting to sidestep these interruptions if the performance of their application suffers. This section looks at two controversial approaches that are often proposed and outlines their potential problems. These are the two deadly sins of garbage collection.

**Sin one: turning off the garbage collector.** Developers often ask for an API to turn off the garbage collector during a time-critical application phase where a garbage-collection pause could result in missed frames. Using such an API, however, complicates application logic and leads to it becoming more difficult to maintain. Forgetting to turn on the garbage collector on a single branch in the program may result in out-of-memory errors. Furthermore, this also complicates the garbage-collector implementation, since it has to support a never-fail allocation mode and must tailor its heuristics to take into account these non-garbage-collecting time periods.

**Sin two: explicit garbage-collection invocation.** JavaScript does not have a Java-style `System.gc()` API, but some developers would like to have that. Their motivation is proactively to invoke garbage collection during a non-time-critical phase in order to avoid it later when timing is critical. The application, however, has no idea how long such a garbage collection will take and therefore may by itself introduce jank. Moreover, garbage-collection heuristics may get confused if developers invoke the garbage collector at arbitrary points in time.

Given the potential for developers to trigger unexpected side effects with these approaches, they should not interfere with garbage collection. Instead, the runtime system should endeavor to avoid the need for such tricks by providing high-performance application throughput and low-latency pauses during mainline application execution, while scheduling longer-running work during periods of idleness such that it does not impact application performance.

## IDLE-TASK SCHEDULING

To schedule long-running garbage collection tasks while Chrome is idle, V8 uses Chrome's task scheduler. This scheduler dynamically reprioritizes tasks based on signals it receives from a variety of other components of Chrome and various heuristics aimed at estimating user intent. For example, if the user touches the screen, the scheduler will prioritize screen rendering and input tasks for a period of 100 ms to ensure that the user interface remains responsive while the user interacts with the web page.

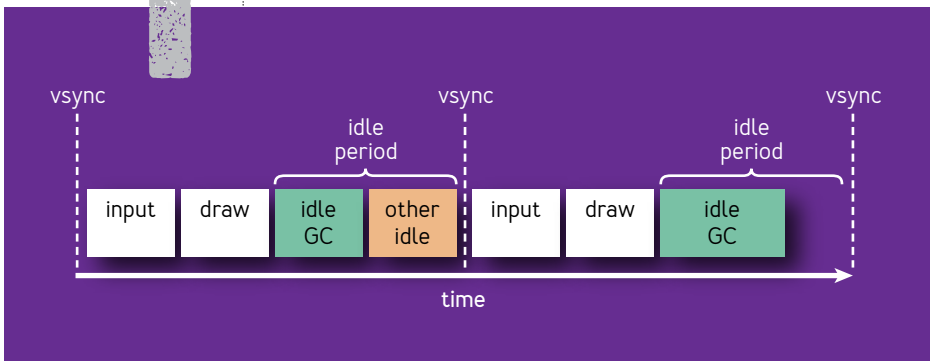
The scheduler's combined knowledge of task queue

occupancy, as well as signals it receives from other components of Chrome, enables it to estimate when Chrome is idle and how long it is likely to remain so. This knowledge is used to schedule low-priority tasks, hereafter called *idle tasks*, which are run only when there is nothing more important to do.

To ensure that these idle tasks don't cause jank, they are eligible to run only in the time periods between the current frame having been drawn to screen and the time when the next frame is expected to start being drawn. For example, during active animations or scrolling (see figure 1), the scheduler uses signals from Chrome's compositor subsystem to estimate when work has been completed for the current frame and what the estimated start time for the next frame is, based on the expected interframe interval (e.g., if rendering at 60 FPS, the interframe interval is 16.6 ms). If no active updates are being made to the screen, the scheduler will initiate a longer idle period, which lasts until the time of the next pending delayed

1

FIGURE 1: IDLE PERIOD EXAMPLE





task, with a cap of 50 ms to ensure that Chrome remains responsive to unexpected user input.

To ensure that idle tasks do not overrun an idle period, the scheduler passes a deadline to the idle task when it starts, specifying the end of the current idle period. Idle tasks are expected to finish before this deadline, either by adapting the amount of work they do to fit within this deadline or, if they cannot complete any useful work within the deadline, by reposting themselves to be executed during a future idle period. As long as idle tasks finish before the deadline, they do not cause jank in web-page rendering.

#### IDLE-TIME GARBAGE-COLLECTION SCHEDULING IN V8

Chrome's task scheduler allows V8 to reduce both jank and memory usage by scheduling garbage-collection work as idle tasks. To do so, however, the garbage collector needs to estimate both when to trigger idle-time garbage-collection tasks and how long those tasks are expected to take. This allows the garbage collector to make the best use of the available idle time without going past an idle-tasks deadline. This section describes implementation details of idle-time scheduling for minor and major garbage collections.

#### Minor garbage-collection idle-time scheduling

Minor garbage collection cannot be divided into smaller work chunks and must be performed either completely or not at all. Performing minor garbage collections during idle time can reduce jank; however, being too proactive in scheduling a minor garbage collection can result

in promotion of objects that could otherwise die in a subsequent non-idle minor garbage collection. This could increase the old-generation size and the latency of future major garbage collections. Thus, the heuristic for scheduling minor garbage collections during idle time should balance between starting a garbage collection early enough that the young-generation size is small enough to be collectable during regular idle time, and deferring it long enough to avoid false promotion of objects.

Whenever Chrome's task scheduler schedules a minor garbage-collection task during idle time, V8 estimates if the time to perform the minor garbage collection will fit within the idle-task deadline. The time estimate is computed using the average garbage-collection speed and the current size of the young generation. It also estimates the young-generation growth rate and performs an idle-time minor garbage collection only if the estimate is that at the next idle period the size of the young generation is expected to exceed the size that could be collected within an average idle period.

### Major garbage-collection idle-time scheduling

A major garbage collection consists of three parts: initiation of incremental marking, several incremental marking steps, and finalization. Incremental marking starts when the size of the heap reaches a certain limit, configured by a heap-growing strategy. This limit is set at the end of the previous major garbage collection, based on the heap-growing factor  $f$  and the total size of live objects in the old generation:  $\text{limit} = f \cdot \text{size}$ .

As soon as an incremental major garbage collection is

started, V8 posts an idle task to Chrome's task scheduler, which will perform incremental marking steps. These steps can be linearly scaled by the number of bytes that should be marked. Based on the average measured marking speed, the idle task tries to fit as much marking work as possible into the given idle time. The idle task keeps reposting itself until all live objects are marked. V8 then posts an idle task for finalizing the major garbage collection. Since finalization is an atomic operation, it is performed only if it is estimated to fit within the allotted idle time of the task; otherwise, V8 reposts that task to be run at a future idle time with a longer deadline.

### Memory Reducer

Scheduling a major garbage collection based on the allocation limit works well when the web page shows a steady allocation rate. If the web page becomes inactive and stops allocating just before hitting the allocation limit, however, there will be no major garbage collection for the whole period while the page is inactive. Interestingly, this is an execution pattern that can be observed in the wild. Many web pages exhibit a high allocation rate during page load as they initialize their internal data structures. Shortly after loading (a few seconds or minutes), the web page often becomes inactive, resulting in a decreased allocation rate and decreased execution of JavaScript code. Thus, the web page will retain more memory than it actually needs while it is inactive.

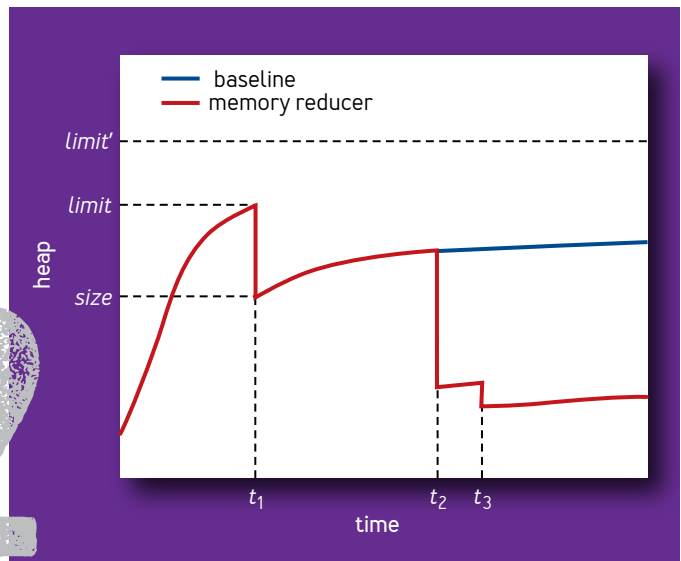
A controller, called a *memory reducer*, tries to detect when the web page becomes inactive and proactively schedules a major garbage collection even if the allocation

limit is not reached. Figure 2 shows an example of major garbage-collection scheduling.

The first garbage collection happens at time  $t_1$  because the allocation limit is reached. V8 sets the next allocation limit based on the heap size. The subsequent garbage collections at times  $t_2$  and  $t_3$  are triggered by the memory reducer before the limit is reached. The dotted line shows what the heap size would be without the memory reducer.

Since this can increase latency, we developed heuristics that rely not only on the idle time provided by Chrome's task scheduler, but also on whether the web page is now inactive. The memory reducer uses the JavaScript invocation and allocation rate as a signal for whether the web page is active or not. When the rate drops below a predefined threshold,

FIGURE 2: EFFECT OF MEMORY REDUCER ON HEAP SIZE



the web page is considered to be inactive and major garbage collection is performed in idle time.

### SILKY SMOOTH PERFORMANCE

Our aim with this work was to improve the quality of user experience for animation-based applications by reducing jank caused by garbage collection. The quality of the user experience for animation-based applications depends not only on the average frame rate, but also on its regularity. A variety of metrics have been proposed in the past to quantify the phenomenon of jank—for example, measuring how often the frame rate has changed, calculating the variance of the frame durations, or simply using the largest frame duration. Although these metrics provide useful information, they all fail to measure certain types of irregularities. Metrics that are based on the distribution of frame durations, such as variance or largest frame duration, cannot take the temporal order of frames into account. For example, they cannot distinguish between the case where two dropped frames are close together and the case where they are further apart. The former case is arguably worse.

We propose a new metric to overcome these limitations. It is based on the discrepancy of the sequence of frame durations. Discrepancy is traditionally used to measure the quality of samples for Monte Carlo integration. It quantifies how much a sequence of numbers deviates from a uniformly distributed sequence. Intuitively, it measures the duration of the worst jank. If only a single frame is dropped, the discrepancy metric is equal to the size of the gap between the drawn frames. If multiple frames are dropped in a row—with some good frames in between—the

discrepancy will report the duration of the entire region of bad performance, adjusted by the good frames.

Discrepancy is a great metric for quantifying the worst-case performance of animated content. Given the timestamps when frames were drawn, the discrepancy can be computed in  $O(N)$  time using a variant of Kadane's algorithm for the maximum subarray problem.

The online WebGL (Web Graphics Library) benchmark OortOnline (<http://oortonline.gl/#run>) demonstrates jank improvements of idle-time garbage-collection scheduling. Figure 3 shows these improvements: frame-time discrepancy, frame time, number of frames missed because of garbage collection, and total garbage-collection time compared with the baseline on the oortonline.gl benchmark.

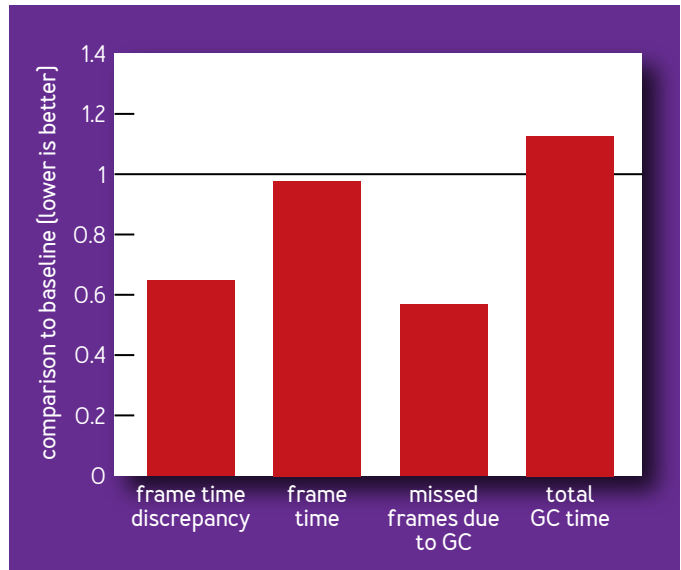
Frame-time discrepancy is reduced on average from 212 ms to 138 ms. The average frame-time improvement is from 17.92 ms to 17.6 ms. We observed that 85 percent of garbage-collection work was scheduled during idle time, which significantly reduced the amount of garbage-collection work performed during time-critical phases. Idle-time garbage-collection scheduling increased the total garbage-collection time by 13 percent to 780 ms. This is because scheduling garbage collection proactively and making faster incremental marking progress with idle tasks resulted in more garbage collections.

Idle-time garbage collection also improves regular web browsing. While scrolling popular web pages such as Facebook and Twitter, we observed that about 70 percent of the total garbage-collection work is performed during idle time.

The memory reducer kicks in when web pages become

## 3

FIGURE 3: IMPROVEMENTS TO THE OORTONLINE.GL BENCHMARK

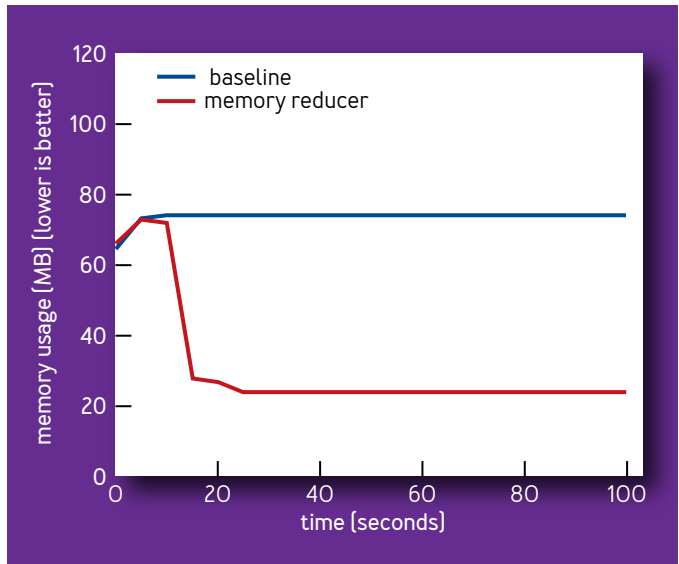


inactive. Figure 4 shows an example run of Chrome with and without the memory reducer on the Google Web Search page. In the first few seconds both versions use the same amount of memory as the web page loads and allocation rate is high. After a while the web page becomes inactive since the page has loaded and there is no user interaction. Once the memory reducer detects that the page is inactive, it starts a major garbage collection. At that point the graphs for the baseline and the memory reducer diverge. After the web page becomes inactive, the memory usage of Chrome with the memory reducer decreases to 34 percent of the baseline.

A detailed description of how to run the experiments presented here to reproduce these results can be found

## 4

FIGURE 4: MEMORY USAGE COMPARISON



in the 2016 PLDI (Programming Language Design and Implementation) artifact evaluation document.<sup>2</sup>

#### OTHER IDLE-TIME GARBAGE-COLLECTED SYSTEMS

A comprehensive overview of garbage collectors taking advantage of idle times is available in a previous article.<sup>4</sup> The authors classify different approaches in three categories: slack-based systems where the garbage collector is run when no other task in the system is active; periodic systems where the garbage collector is run at predefined time intervals for a given duration; and hybrid systems taking advantage of both ideas. The authors found that, on average, hybrid systems provide the best performance, but some applications favor a slack-based or periodic system.



Our approach of idle-time garbage-collection scheduling is different. Its main contribution is that it profiles the application and garbage-collection components to predict how long garbage-collection operations will take and when the next minor or major collection will occur as a result of application allocation throughput. That information allows efficient scheduling of garbage-collection operations during idle times to reduce jank while providing high throughput.

#### CONCURRENT, PARALLEL, INCREMENTAL GARBAGE COLLECTION

An orthogonal approach to avoid garbage-collection pauses while executing an application is achieved by making garbage-collection operations concurrent, parallel, or incremental. Making the marking phase or the compaction phase concurrent or incremental typically requires read or write barriers to ensure a consistent heap state. Application throughput may degrade because of expensive barrier overhead and code complexity of the virtual machine.

Idle-time garbage-collection scheduling can be combined with concurrent, parallel, and incremental garbage-collection implementations. For example, V8 implements incremental marking and concurrent sweeping, which may also be performed during idle time to ensure fast progress. Most importantly, costly memory-compaction phases such as young-generation evacuation or old-generation compaction can be efficiently hidden during idle times without introducing costly read or write barrier overheads.

For a best-effort system, where hard realtime deadlines do not have to be met, idle-time garbage-collection scheduling may be a simple approach to provide both high throughput and low jank.

## BEYOND GARBAGE COLLECTION AND CONCLUSIONS

Idle-time garbage-collection scheduling focuses on the user's expectation that a system that renders at 60 frames per second appears silky smooth. As such, our definition of idleness is tightly coupled to on-screen rendering signals. Other applications can also benefit from idle-time garbage-collection scheduling when an appropriate definition of idle time is applied. For example, a node.js-based server that is built on V8 could forward idle-time periods to the V8 garbage collector while it waits for a network connection.

The use of idle time is not limited to garbage collection. It has been exposed to the web platform in the form of the `requestIdleCallback` API,<sup>5</sup> enabling web pages to schedule their own callbacks to be run during idle time. As future work, other management tasks of the JavaScript engine could be executed during idle time (e.g., compiling code with the optimizing just-in-time compiler that would otherwise be performed during JavaScript execution).

## References

1. Degenbaev, U., Eisinger, J., Ernst, M., McIlroy, R., Payer, H. 2016. Idle time garbage collection scheduling. *In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.*
2. Degenbaev, U., Eisinger, J., Ernst, M., McIlroy, R., Payer, H.

2016. Idle time garbage collection scheduling; PLDI'16, June 13–17, 2016, Santa Barbara, CA, USA ACM. 978-1-4503-4261-2/16/06, pages 570-583
3. Google Inc. The RAIL performance model; <http://developers.google.com/web/tools/chrome-devtools/profile/evaluate-performance/rail>.
  4. Kalibera, T., Pizlo, F., Hosking, A. L., Vitek, J. 2011. Scheduling real-time garbage collection on uniprocessors. *ACM Transactions on Computer Systems* 29(3): 8:1–8:29.
  5. McIlroy. R. 2016. Cooperative scheduling of background tasks. W3C editor's draft; <https://w3c.github.io/requestidlecallback/>.
  6. Ungar, D. 1984. Generation scavenging: a nondisruptive high-performance storage reclamation algorithm. In *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*.

**Ulan Degenbaev** is a software engineer at Google, working on the garbage collector of the V8 JavaScript engine. He received his master's degree and a Ph.D in computer science from Saarland University, where he worked on formal specifications and software verification.

**Jochen Eisinger** is a software engineer at Google, working on the V8 JavaScript engine and Chrome security. Prior to that, he worked on various other parts of Chrome. Before joining Google, he was a postdoc fellow at the University of British Columbia Vancouver. He received his Diplom degree and a Ph.D in computer science from the University of Freiburg, Germany.

**Manfred Ernst** is a software engineer at Google, where he works on virtual reality. Prior to that, he integrated a GPU rasterization engine into the Chrome web browser. Before joining Google, Ernst was a research scientist at Intel Labs, where he led the development of the Embree ray tracing kernels. He was also a cofounder and the CEO of Bytes+Lights, a start-up company that developed visualization tools for the automotive industry. He received his Diplom degree and a Ph.D in computer science from the University of Erlangen-Nuremberg.

**Ross McIlroy** is a software engineer at Google and tech lead of V8's interpreter effort. He previously worked on Chrome's scheduling subsystem and mobile optimization efforts. Before joining Google, McIlroy worked on various operating-system and virtual-machine research projects, including Singularity, Helios, Barrelfish, and HeraJVM. He received his Ph.D in computing science from the University of Glasgow, where he worked on heterogeneous multicore virtual machines.

**Hannes Payer** is a software engineer at Google, tech lead of the V8 JavaScript garbage collection effort, and a virtual-machine enthusiast. Prior to V8, Hannes worked on Google's Dart virtual machine and various Java virtual machines. He received a Ph.D from the University of Salzburg, where he worked on multicore scalability of concurrent objects and was the principal investigator of the Scal project.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.

# Distributed Consensus and Implications of NVM on Database Management Systems

**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 two experts share a short curated selection of papers on a concentrated, practically oriented topic.*

I am thrilled to introduce our second installment of Research for Practice, which provides highlights from two critical areas in storage and large-scale services: distributed consensus and nonvolatile memory.

First, how do large-scale distributed systems mediate access to shared resources, coordinate updates to mutable state, and reliably make decisions in the presence of failures? Camille Fournier, a seasoned and experienced distributed-systems builder (and ZooKeeper PMC), has curated a fantastic selection on distributed consensus in practice. The history of consensus echoes many of the goals of RfP: for decades the study and use of consensus protocols were considered notoriously difficult to understand and remained primarily academic concerns. As a result, these protocols were largely ignored by industry. The rise of Internet-scale services and demands for automated solutions to cluster management, failover, and sharding in the 2000s finally led to the widespread practical adoption of these techniques. Adoption proved difficult, however, and the process in turn led to new (and ongoing) research on the subject. The papers in this selection highlight the challenges and the rewards of

making the theory of consensus practical—both in theory and in practice.

Second, while consensus concerns *distributed* shared state, our second selection concerns the impact of hardware trends on *single-node* shared state. Joy Arulaj and Andy Pavlo provide a whirlwind tour of the implications of NVM (nonvolatile memory) technologies on modern storage systems. NVM promises to overhaul the traditional paradigm that stable storage (i.e., storage that persists despite failures) be block-addressable (i.e., requires reading and writing in large chunks). In addition, NVM's performance characteristics lead to entirely different design trade-offs than conventional storage media such as spinning disks.

As a result, there is an arms race to rethink software storage-systems architectures to accommodate these new characteristics. This selection highlights projected implications for recovery subsystems, data-structure design, and data layout. While the first NVM devices have yet to make it to market, these pragmatically oriented citations from the literature hint at the volatile effects of nonvolatile media on future storage systems.

I believe these two excellent contributions fulfill RfP's goal of allowing you, the reader, to become an expert in a weekend afternoon's worth of reading. To facilitate this process, as always, we have provided open access to the ACM Digital Library for the relevant citations from these selections so you can enjoy these research results in their full glory. Keep on the lookout for our next installment, and please enjoy! —*Peter Bailis*

*“A distributed system is one in which the failure of a computer you didn’t know existed can render your own computer unusable.”*  
—Leslie Lamport

## DISTRIBUTED CONSENSUS

BY CAMILLE FOURNIER

**A**s Lamport predicted in this quote, the real challenges of distributed computing—not just communicating via a network, but communicating to unknown nodes in a network—have greatly intensified in the past 15 years. With the incredible scaling of modern systems, “we have found ourselves in a world where answering the question, what is running where?” is increasingly difficult. Yet, we continue to have requirements that certain data never be lost and that certain actions behave in a consistent and predictable fashion, even when some nodes of the system may fail. To that end, there has been a rapid adoption of systems that rely on consensus protocols to guarantee this consistency in a widely distributed world.

The three papers included in this selection address the real world of consensus systems: Why are they needed? Why are they difficult to understand? What happens when you try to implement them? Is there an easier way, something that more developers can understand and therefore implement?

The first two papers discuss the reality of implementing Paxos-based consensus systems at Google, focusing first on the challenges of correctly implementing Paxos itself, and second on the challenges of creating a system based on a consensus algorithm that provides useful functionality for developers. The final paper attempts to answer the question, *is there an easier way?* by introducing Raft, a consensus algorithm designed to be easier for developers to understand.

## Theory Meets Reality

*Chandra, T. D., Griesemer, R., Redstone, J. et al. 2007. Paxos made live—an engineering perspective. Proceedings of the 26<sup>th</sup> Annual ACM Symposium on Principles of Distributed Computing: 398-407.*

<http://queue.acm.org/rfp/vol14iss3.html>

Paxos as originally stated is a page of pseudocode. The complete implementation of Paxos inside of Google’s Chubby lock service is several thousand lines of C++. What happened? “Paxos Made Live” documents the evolution of the Paxos algorithm from theory into practice.

The basic idea of Paxos is to use voting by replicas with consistent storage to ensure that, even in the presence of failures, there can be a unilateral consensus. This requires a coordinator be chosen, proposals sent and voted upon, and finally a commit recorded. Generally, systems record a series of these consensus values to a sequence log. This log-based variant is called multi-Paxos, which is less formally specified.

In creating a real system, durable logs are written to disks, which have finite capacity and are prone to corruption that must be detected and taken into account. The algorithm must be run on machines that can fail, and to make it operable at scale you need to be able to change group membership dynamically. While the system was expected to be fault-tolerant, it also needed to perform quickly enough to be useful; otherwise, developers would work around it and create incorrect abstractions. The team details their efforts to make sure the core algorithm is expressed correctly and is testable, but even with these



conscious efforts, the need for performance optimizations, concurrency, and multiple developers working on the project still means that the final system is ultimately an extended version of Paxos, which is difficult to prove correct.

### Hell is Other Programmers

*Burrows, M. 2006. The Chubby lock service for loosely coupled distributed systems. Proceedings of the Seventh Symposium on Operating Systems Design and Implementation: 335-350.*

<http://queue.acm.org/rfp/vol14iss3.html>

While “Paxos Made Live” discusses the implementation of the consensus algorithm in detail, this paper about the Chubby lock service examines the overall system built around this algorithm. As research papers go, this one is a true delight for the practitioner. In particular, it describes designing a system and then evolving that design after it comes into contact with real-world usage. This paper should be required reading for anyone interested in designing and developing core infrastructure software that is to be offered as a service.

Burrows begins with a discussion of the design principles chosen as the basis for Chubby. Why make it a centralized service instead of a library? Why is it a lock service, and what kind of locking is it used for? Chubby not only provides locks, but also serves small files to facilitate sharing of metadata about distributed system state for its clients. Given that it is serving files, how many clients

should Chubby expect to support, and what will that mean for the caching and change notification needs?

After discussing the details of the design, system structure, and API, Burrows gets into the nitty-gritty of the implementation. Building a highly sensitive centralized service for critical operations such as distributed locking and name resolution turns out to be quite difficult. Scaling the system to tens of thousands of clients meant being smart about caching and deploying proxies to handle some of the load. The developers misused and abused the system by accident, using features in unpredictable ways, attempting to use the system for large data storage or messaging. The Chubby maintainers resorted to reviewing other teams' planned uses of Chubby and denying access until review was satisfied. Through all of this we can see that the challenge in building a consensus system goes far beyond implementing a correct algorithm. We are still building a system and must think as carefully about its design and the users we will be supporting.

### Can We Make This Easier?

*Ongaro, D., Ousterhout, J. 2014. In search of an understandable consensus algorithm. Proceedings of the Usenix Annual Technical Conference: 305-320.*  
<https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro>

Finally we come to the question, have we built ourselves into unnecessary complexity by taking it on faith that Paxos and its close cousins are the only way to implement

consensus? What if there were an algorithm that we could also show to be correct but was designed to be easier for people to comprehend and implement correctly?

Raft is a consensus algorithm written for managing a replicated log but designed with the goal of making the algorithm itself more understandable than Paxos. This is done both by decomposing the problem into pieces that can be implemented and understood independently and by reducing the number of states that are valid for the system to hold.

Consensus is decomposed into issues of leader election, log replication, and safety. Leader election uses randomized election timeouts to reduce the likelihood of two candidates for leader splitting the vote and requiring a new round of elections. It allows candidates for leader to be elected only if they have the most up-to-date logs. This prevents the need for transferring data from follower to leader upon election. If a follower's log does not match the expected state for a new entry, the leader will replay entries from earlier in its log until it reaches a point at which the logs match, thus correcting the follower. This also means that a history of changes is stored in the logs, providing a side value of letting clients read (some) historical entries, should they desire.

The authors then show that after teaching a set of students both Paxos and Raft, the students were quizzed on their understanding of each and scored meaningfully higher on the Raft quiz. Looking around the current state of consensus systems in industry, we can see this play out in another way: namely, several new consensus systems have

been created since 2014 based on Raft, where previously there were very few reliable and successful open-source systems based on Paxos.

**Bottlenecks, Single Points of Failure, and Consensus**  
Developers are often tempted to use a centralized consensus system to serve as the system of record for distributed coordination. Explicit coordination can make certain problems much easier to reason about and correct for; however, that puts the consensus system in the position of the bottleneck or critical point of failure for the other systems that rely on it to make progress. As we can see from these papers, making a centralized consensus system production-ready can come at the cost of adding optimizations and recovery mechanisms that were not dreamed of in the original Paxos literature.

What is the way forward? Arguably, writing systems that do not rely on centralized consensus brokers to operate safely would be the best option, but we are still in the early days of coordination-avoidance research and development. While we wait for more evolution on that front, Raft provides an interesting alternative, an algorithm designed for readability and general understanding. The impact of having an easier algorithm to implement is already being felt, as far more developers are embedding Raft within distributed systems and building specifically tailored Raft-based coordination brokers. Consensus remains a tricky problem—but one that is finally seeing a diversity of approaches to reaching a solution.

## IMPLICATIONS OF NVM ON DATABASE MANAGEMENT SYSTEMS

BY JOY ARULRAJ AND ANDREW PAVLO

**T**he advent of NVM (nonvolatile memory) will fundamentally change the dichotomy between memory and durable storage in a DBMS (database management system). NVM is a broad class of technologies—including phase-change memory, memristors, and STT-MRAM (spin-transfer torque-magnetoresistive random-access memory)—that provide low-latency reads and writes on the same order of magnitude as DRAM (dynamic random-access memory), but with persistent writes and large storage capacity like an SSD (solid-state drive). Unlike DRAM, writes to NVM are expected to be more expensive than reads. These devices also have limited write endurance, which necessitates fewer writes and wear-leveling to increase their lifetimes.

The first NVM devices released will have the same form factor and block-oriented access as today's SSDs. Thus, today's DBMSes will use this type of NVM as a faster drop-in replacement for their current storage hardware.

By the end of this decade, however, NVM devices will support byte-addressable access akin to DRAM. This will require additional CPU architecture and operating-system support for persistent memory. This also means that existing DBMSes are unable to take full advantage of NVM because their internal architectures are predicated on the assumption that memory is volatile. With NVM, many of the components of legacy DBMSes are unnecessary

and will degrade the performance of data-intensive applications.

We have selected three papers that focus on how the emergence of byte-addressable NVM technologies will impact the design of DBMS architectures. The first two present new abstractions for performing durable atomic updates on an NVM-resident database and recovery protocols for an NVM DBMS. The third paper addresses the write-endurance limitations of NVM by introducing a collection of write-limited query-processing algorithms. Thus, this selection contains novel ideas that can help leverage the unique set of attributes of NVM devices for delivering the features required by modern data-management applications. The common theme for these papers is that you cannot just run an existing DBMS on NVM and expect it to leverage its unique set of properties. The only way to achieve that is to come up with novel architectures, protocols, and algorithms that are tailor-made for NVM.

### ARIES Redesigned for NVM

*Coburn, J., et al. 2013. From ARIES to MARS: transaction support for next-generation, solid-state drives. Proceedings of the 24th ACM Symposium on Operating Systems Principles. 197-212.*

<http://queue.acm.org/rfp/vol14iss3.html>

ARIES is considered the standard for recovery protocols in a transactional DBMS. It has two key goals: first, it provides an interface for supporting scalable ACID

(atomicity, consistency, isolation, durability) transactions; second, it maximizes performance on disk-based storage systems. In this paper, the authors focus on how ARIES should be adapted for NVM-based storage.

Since random writes to the disk whenever a transaction updates the database obviously decrease performance, ARIES requires that the DBMS first record a log entry in the write-ahead log (a sequential write) before updating the database itself (a random write). It adopts a *no-force* policy wherein the updates are written to the database lazily after the transaction commits. Such a policy assumes that sequential writes to nonvolatile storage are significantly faster than random writes. The authors, however, demonstrate that this is no longer the case with NVM.

The MARS protocol proposes a new hardware-assisted logging primitive that combines multiple writes to arbitrary storage locations into a single atomic operation. By leveraging this primitive, MARS eliminates the need for an ARIES-style undo log and relies on the NVM device to apply the redo log at commit time. We are particularly fond of this paper because it helps in better appreciating the intricacies involved in designing the recovery protocol in a DBMS for guarding against data loss.

### Near-Instantaneous Recovery Protocols

*Arulraj, J., Pavlo, A., Dulloor, S. R. 2015. Let's talk about storage and recovery methods for nonvolatile memory database systems. Proceedings of the ACM SIGMOD International Conference on Management of Data: 707-722. <http://queue.acm.org/rfp/vol14iss3.html>*

This paper takes a different approach to performing durable atomic updates on an NVM-resident database than the previous paper. In ARIES, during recovery the DBMS first loads the most recent snapshot. It then replays the redo log to ensure that all the updates made by committed transactions are recovered. Finally, it uses the undo log to ensure that the changes made by incomplete transactions are not present in the database. This recovery process can take a lot of time, depending on the load on the system and the frequency with which snapshots are taken. Thus, this paper explores whether it is possible to leverage NVM's properties to speed up recovery from system failures.

The authors present a software-based primitive called a *nonvolatile pointer*. When a pointer points to data residing on NVM, and is itself stored on NVM, then it will remain valid even after the system recovers from a power failure. Using this primitive, the authors design a library of nonvolatile data structures that support durable atomic updates. They propose a recovery protocol that, in contrast to MARS, obviates the need for an ARIES-style redo log. This enables the system to skip replaying the redo log, and thereby allows the NVM DBMS to recover the database almost instantaneously.

Both papers propose recovery protocols that target an NVM-only storage hierarchy. The generalization of these protocols to a multitier storage hierarchy with both DRAM and NVM is a hot topic in research today.



### Trading Expensive Writes for Cheaper Reads

Viglas, S. D. 2014. *Write-limited sorts and joins for persistent memory*. *Proceedings of the VLDB Endowment* 7(5): 413-424. <http://www.vldb.org/pvldb/vol7/p413-viglas.pdf>

The third paper focuses on the higher write costs and limited write-endurance problems of NVM. For several decades algorithms have been designed for the random-access machine model where reads and writes have the same cost. The emergence of NVM devices, where writes are more expensive than reads, opens up the design space for new write-limiting algorithms. It will be fascinating to see researchers derive new bounds on the number of writes that different kinds of query-processing algorithms must perform.

Viglas presents a collection of novel query-processing algorithms that minimize I/O by trading off expensive NVM writes for cheaper reads. One such algorithm is the *segment sort*. The basic idea is to use a combination of two sorting algorithms—external merge sort and selection sort—that splits the input into two segments that are then processed using a different algorithm. The selection-sort algorithm uses extra reads, and writes out each element in the input only once at its final location. By using a combination of these two algorithms, the DBMS can optimize both the performance and the number of NVM writes.

### Game Changer for DBMS Architectures

NVM is a definite game changer for future DBMS architectures. It will require system designers to rethink

many of the core algorithms and techniques developed over the past 40 years. Using these new storage devices in the manner prescribed by these papers will allow DBMSes to achieve better performance than what is possible with today's hardware for write-heavy database applications. This is because these techniques are designed to exploit the low-latency read/writes of NVM to enable a DBMS to store less redundant data and incur fewer writes. Furthermore, we contend that existing in-memory DBMSes are better positioned to use NVM when it is finally available. This is because these systems are already designed for byte-addressable access methods, whereas legacy disk-oriented DBMSes will require laborious and costly overhauls in order to use NVM correctly, as described in these papers.

**Peter Bailis** is an assistant professor of computer science at Stanford University. His research in the Future Data Systems group (<http://futuredata.stanford.edu/>) focuses on the design and implementation of next-generation data-intensive systems. He received a Ph.D. from UC Berkeley in 2015 and an A.B. from Harvard in 2011, both in computer science.

**Camille Fournier** is a writer, speaker, and entrepreneur. Formerly the CTO of Rent the Runway, she serves on the technical oversight committee for the Cloud Native Computing Foundation, as a Project Management Committee member of the Apache ZooKeeper project, and a project overseer of the Dropwizard web framework. She has an M.S. from the University of Wisconsin-Madison and a B.S. from

Carnegie Mellon University. You can find more of her writing at [elidedbranches.com](http://elidedbranches.com).

**Joy Arulraj** is a Ph.D. candidate at Carnegie Mellon University. He is interested in the design and implementation of next-generation database management systems, particularly in the areas of realtime data analytics, self-driving modules, and adoption of nonvolatile memory technologies. He earned an M.S. in computer sciences from the University of Wisconsin, Madison, and received a B.E. in computer science and engineering from the College of Engineering, Guindy. He is a recipient of the 2016 Samsung Ph.D. Fellowship.

**Andy Pavlo** is an assistant professor of databaseology in the computer science department at Carnegie Mellon University. Copyright © 2016 held by owner/author. Publication rights licensed to ACM.

**SHAPE THE FUTURE OF COMPUTING!**

Join ACM today at [acm.org/join](http://acm.org/join)

**BE CREATIVE. STAY CONNECTED.  
KEEP INVENTING.**



Association for  
Computing Machinery

*Advancing Computing as a Science & Profession*

DYNAMICS  
OF CHANGE:

# Why Reactivity Matters

**TAME THE  
DYNAMICS OF  
CHANGE BY  
CENTRALIZING  
EACH CONCERN IN  
ITS OWN MODULE**

ANDRE MEDEIROS

**P**rofessional programming is about dealing with software at scale. Everything is trivial when the problem is small and contained: it can be elegantly solved with imperative programming or functional programming or any other paradigm. Real-world challenges arise when programmers have to deal with large amounts of data, network requests, or intertwined entities, as in UI (user interface) programming.

Of these different types of challenges, managing the dynamics of change in a code base is a common one that may be encountered in either UI programming or the back end. How to structure the flow of control and concurrency among multiple parties that need to update one another with new information is referred to as *managing change*. In both UI programs and servers, concurrency is typically present and is responsible for most of the challenges and complexity.

Some complexity is accidental and can be removed. Managing concurrent complexity becomes difficult when the amount of essential complexity is large. In those cases, the interrelation between the entities is complex—and cannot be made less so. For example, the requirements themselves may already represent essential complexity. In an online text editor, the requirements alone may determine that a keyboard input needs to change the view, update text formatting, perhaps also change the table of contents, word count and paragraph count, request the document to be saved, and take other actions.

Because essential complexity cannot be eliminated, the alternative is to make it as understandable as possible, which leads to making it maintainable. When it comes to complexity of change around some entity Foo, you want to understand what Foo changes, what can change Foo, and which part is responsible for the change.

#### HOW CHANGE PROPAGATES FROM ONE MODULE TO ANOTHER

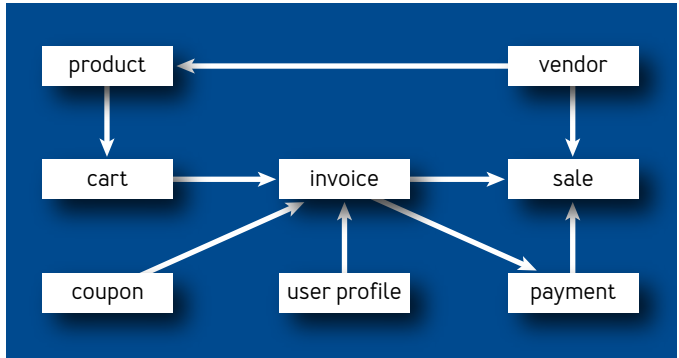
Figure 1 is a data flow chart for a code base of e-commerce software, where rectangles represent modules and arrows represent communication. These modules are interconnected as requirements, not as architectural decisions. Each module may be an object, an object-oriented class, an actor, or perhaps a thread, depending on the programming language and framework used.

An arrow from the `Cart` module to the `Invoice` module (figure 2a) means that the cart changes or affects the state in the invoice in a meaningful way. A practical example of this situation is a feature that recalculates the

# 1

**M**anaging concurrent complexity becomes difficult when the amount of essential complexity is large.

FIGURE 1: DATA FLOW FOR A CODEBASE OF AN E-COMMERCE SOFTWARE

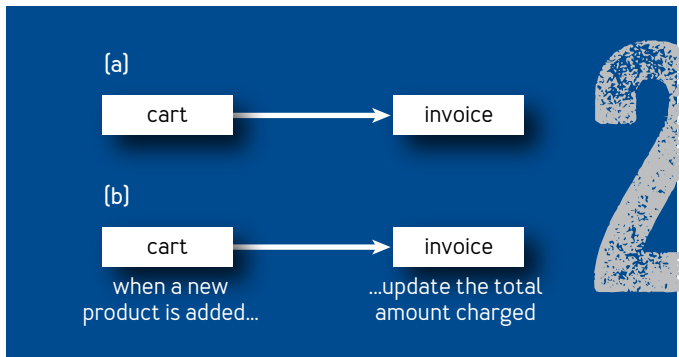


total invoicing amount whenever a new product is added to the cart (figure 2b).

The arrow starts in the `Cart` and ends in the `Invoice` because an operation internal to the `Cart` may cause the state of the `Invoice` to change. The arrow represents the dynamics of change between the `Cart` and the `Invoice`.

Assuming that all code lives in some module, the arrow

FIGURE 2: THE CART CHANGES THE INVOICE



cannot live in the space between; it must live in a module, too. Is the arrow defined in the `Cart` or in the `Invoice`? It is up to the programmer to decide that.

### Passive Programming

It is common to place the arrow definition in the arrow tail: the `Cart`. Code in the `Cart` that handles the addition of a new product is typically responsible for triggering the `Invoice` to update its invoicing data, as demonstrated in the chart and the Kotlin (<https://kotlinlang.org/>) code snippet in figure 3.

The `Cart` assumes a proactive role, and the `Invoice`

FIGURE 3: **PASSIVE PROGRAMMING WITH CODE IN TAIL**

3



takes a passive role. While the `Cart` is responsible for the change and keeping the `Invoice` state up to date, the `Invoice` has no code indicating that the update is coming from the `Cart`. Instead, it must expose `updateInvoicing` as a public method. On the other hand, the cart has no access restrictions; it is free to choose whether the `ProductAdded` event should be private or public.

Let's call this programming style *passive programming*, characterized by remote imperative changes and delegated responsibility over state management.

### Reactive Programming

The other way of defining the arrow's ownership is *reactive programming*, where the arrow is defined at the arrow head: the `Invoice`, as shown in figure 4. In this setting, the `Invoice` listens to a `ProductAdded` event happening in the cart and determines that it should change its own internal invoicing state.

The `Cart` now assumes a broadcasting role, and the `Invoice` takes a reactive role. The `Cart`'s responsibility is to carry out its management of purchased products, while providing notification that a product has been added or removed.

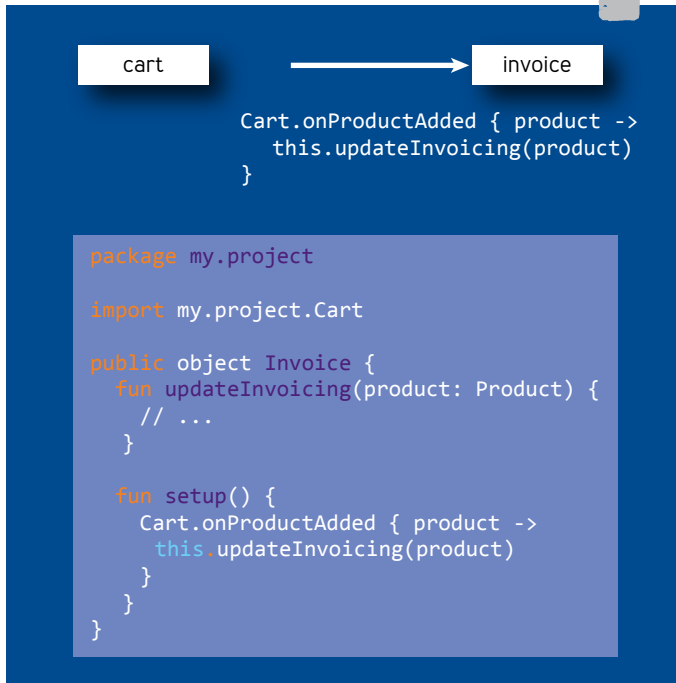
Therefore, the `Cart` has no code that explicitly indicates that its events may affect the state in the `Invoice`. On the other hand, the `Invoice` is responsible for keeping its own invoicing state up to date and has the `Cart` as a dependency.

The responsibilities are now inverted, and the `Invoice` may choose to have its `updateInvoicing` method private or public, but the `Cart` must make the `ProductAdded`



# 4

FIGURE 4: REACTIVE PROGRAMMING WITH CODE IN HEAD



# 5

event public. Figure 5 illustrates this duality.

The term *reactive* was vaguely defined in 1989 by Gérard Berry.<sup>1</sup> The definition given here is broad enough to cover existing notions of reactive systems such as spreadsheets, the actor model, Reactive Extensions (Rx), event streams, and others.

FIGURE 5: PUBLIC VS. PRIVATE

PROGRAMMING	"PRODUCT ADDED" EVENT IN THE CART	UPDATE INVOICING DATA METHOD IN THE INVOICE
Passive	private or public	public
Reactive	public	private or public

## PASSIVE VERSUS REACTIVE FOR MANAGING ESSENTIAL COMPLEXITY

In the network of modules and arrows for communication of change, where should the arrows be defined? When should reactive programming be used and when is the passive pattern more suitable?

There are usually two questions to ask when trying to understand a complex network of modules:

- ➔ Which modules does module X change?
- ➔ Which modules can change module X?

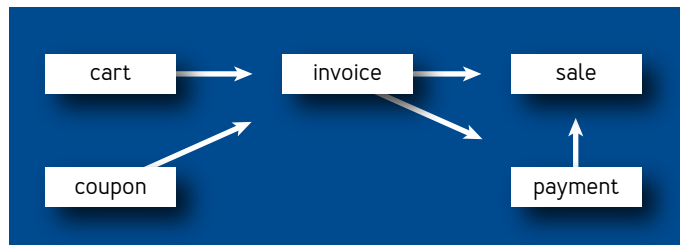
The answers depend on which approach is used: reactive or passive, or both. Let's assume, for simplicity, that whichever approach is chosen, it is applied uniformly across the architecture.

For example, consider the network of e-commerce modules shown in figure 6, where the passive pattern is used everywhere.

To answer the first question for the `Invoice` module (Which modules does the invoice change?), you need only to look at the code in the `Invoice` module, because it owns the arrows and defines how other modules are remotely changed from within the `Invoice` as a proactive component.

FIGURE 6: FREQUENT PASSIVE PATTERN

6



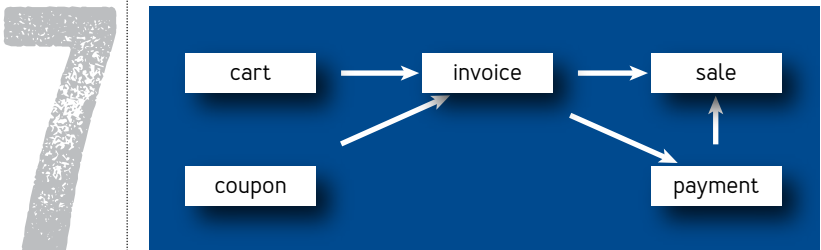
To discover which modules can change the state of the `Invoice`, however, you need to look for all the usages of public methods of the `Invoice` throughout the code base.

In practice, this becomes hard to maintain when multiple other modules may change the `Invoice`, which is the case in essentially complex software. It may lead to situations where the programmer has to build a mental model of how multiple modules concurrently modify a piece of state in the module in question. The opposite alternative is to apply the reactive pattern everywhere, illustrated in figure 7.

To discover which modules can change the state of the `Invoice`, you can just look at the code in the `Invoice` module, because it contains all “arrows” that define dependencies and dynamics of change. Building the mental model of concurrent changes is easier when all relevant entities are co-located.

On the other hand, the dual concern of discovering which other modules the `Invoice` affects can be answered only by searching for all usages of the `Invoice` module’s public broadcast events.

FIGURE 7: **FREQUENT REACTIVE PATTERN**



When arranged in a table, as in figure 8, these described properties for passive and reactive are dual to each other.

The pattern you choose depends on which of these two questions is more commonly on a programmer’s mind when dealing with a specific code base. Then you can pick the pattern whose answer to the most common question is “look inside,” because you want to be able to find the answer quickly. A centralized answer is better than a distributed one.

While both questions are important in an average code base, a more common need may be knowing how a particular module works. This is why reactivity matters: you usually need to know how a module works before looking at what the module affects.

Because a passive-only approach generates *irresponsible* modules (they delegate their state management to other modules), a reactive-only approach is a more sensible default choice. That said, the passive pattern is suitable for data structures and for creating a hierarchy of ownership. Any common data structure (such as a hash map) in object-oriented programming is a passive module, because it exposes methods that allow changing its internal state. Because it delegates the responsibility of answering the question “When does it change?” to whichever module contains the data-structure object, it

# 8

FIGURE 8: **DUAL PROPERTIES**

	PASSIVE	REACTIVE
How does it work?	Find usages	Look inside
What does it affect?	Look inside	Find usages


creates a hierarchy: the containing module as the parent and the data structure as the child.

### MANAGING DEPENDENCIES AND OWNERSHIP

With the reactive-only approach, every module must statically define its dependencies to other modules. In the `Cart` and `Invoice` example, `Invoice` would need to statically import `Cart`. Because this applies everywhere, all modules would have to be singletons. In fact, Kotlin's `object` keyword is used (in Scala as well) to create singletons.

In the reactive example in figure 9, there are two concerns regarding dependencies:

FIGURE 9: **REACTIVE-ONLY APPROACH**



```
package my.project

import my.project.Cart // This is a singleton

public object Invoice { // This is a singleton too
    fun updateInvoicing(product: Product) {
        // ...
    }

    fun setup() {
        Cart.onProductAdded { product ->
            this.updateInvoicing(product)
        }
    }
}
```

- ➔ What the dependency is: defined by the `import` statement.
- ➔ How to depend: defined by the event listener.

The problem with singletons as dependencies relates only to the *what* concern in the reactive pattern. You would still like to keep the reactive style of *how* dependencies are put together, because it appropriately answers the question, “How does the module work?”

While reactive, the module being changed is statically aware of its dependencies through imports; while passive, the module being changed is unaware of its dependencies.

So far, this article has analyzed the passive-only and reactive-only approaches, but in between lies the opportunity for mixing both paradigms: keeping only the *how* benefit from reactive, while using passive programming to implement the *what* concern.

The `Invoice` module can be made passive with regard to its dependencies: it exposes a public method to allow another module to set or inject a dependency. Simultaneously, `Invoice` can be made reactive with regard to how it works. This is shown in the example code in figure 10, which yields a hybrid passively reactive solution:

- ➔ How does it work? Look inside [reactive].
- ➔ What does it depend on? Injected via a public method [passive].

This would help make modules more reusable, because they are not singletons anymore. Let’s look at another example where a typical passive setting is converted to a passively reactive one.

## 10

FIGURE 10: A HYBRID PASSIVELY REACTIVE SOLUTION

```
package my.project

public object Invoice {
    fun updateInvoicing(product: Product) {
        // ...
    }

    private var cart: Cart? = null

    public fun setCart(cart: Cart) {
        this.cart = cart
        cart.onProductAdded { product ->
            this.updateInvoicing(product)
        }
    }
}
```

## EXAMPLE: ANALYTICS EVENTS

It is common to write the code for a UI program in passive-only style, where each different screen or page of the program uses the public methods of an `Analytics` module to send events to an `Analytics` back end. The example code in figure 11 illustrates this.

The problem with building a passive-only solution for analytics events is that every single page needs to have code related to analytics. Also, to understand the behavior of analytics, you must study it scattered throughout the code. It's desirable to separate the analytics aspect from the core features and business logic concerning a page

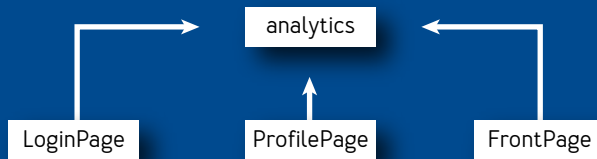
## 11

FIGURE 11: PASSIVE-ONLY APPROACH

```
// In the LoginPage module
package my.project

import my.project.Analytics

val loginButton = //...
loginButton.addClickListener { clickEvent ->
    Analytics.sendEvent('User clicked the login button')
}
```



such as the `LoginPage`. Aspect-oriented programming<sup>2</sup> is one attempt at solving this, but it is also possible to separate aspects through reactive programming with events.

In order to make the code base reactive only, the `Analytics` module would need to statically depend on all the pages in the program. Instead, you can use the passively reactive solution to make the `Analytics` module receive its dependencies through a public injection method. This way, a parent module that controls routing of pages



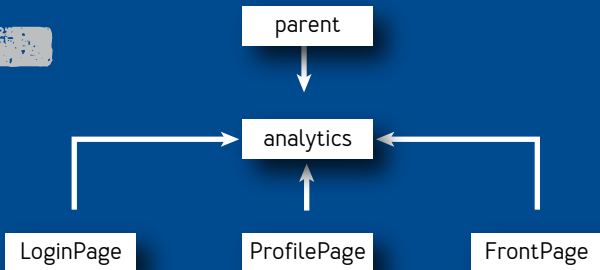
## 12

can also bootstrap the analytics with information on those pages. See the example in figure 12.

### MIND THE ARROWS

Introducing reactive patterns in an architecture can help better define which module *owns* a relationship of change between two modules. Software architectures

FIGURE 12: **PUBLIC INJECTION METHOD**



```
// In the Analytics module
package my.project

public object Analytics {
    public fun inject(loginPage: Page) {
        loginPage.loginButton.addClickListener { clickEvent ->
            this.sendEvent('User clicked the login button')
        }
    }

    private fun sendEvent(eventMessage: String) {
        // ...
    }
}
```

for essential complex requirements are often about structuring the code in modules, but do not forget that the arrows between modules also live in modules. Some degree of reactivity matters because it creates separation of concerns. A particular module should be responsible for its own state. This is easily achievable in an event-driven architecture, where modules do not invasively change each other. Tame the dynamics of change by centralizing each concern in its own module.

## References

1. Berry, G. 1989. Real time programming: special purpose or general purpose languages. [Research Report] RR-1065. INRIA (French Institute for Research in Computer Science and Automation); <https://hal.inria.fr/inria-00075494/document>.
2. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., Irwin, J. 1997. Aspect-oriented programming. *Proceedings of the 11th European Conference on Object-Oriented Programming*: 220–242.

**Andre Medeiros** is a web and mobile developer at Futurece. He is known for his involvement with reactive programming for user interfaces, particularly with the *ReactiveX* libraries. Medeiros has built JavaScript libraries and tools such as *Cycle.js* and *RxMarbles*. He has an M.Sc. in theoretical computer science.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.

# Cluster-Level Logging *of* Containers *with* Containers

SATNAM SINGH

**LOGGING  
CHALLENGES  
OF CONTAINER  
BASED CLOUD  
DEPLOYMENTS**

**T**his article shows how cluster-level logging infrastructure can be implemented using open source tools and deployed using the very same abstractions that are used to compose and manage the software systems being logged.

Collecting and analyzing log information is an essential aspect of running production systems to ensure their reliability and to provide important auditing information. Many tools have been developed to help with the aggregation and collection of logs for specific software components (e.g., an Apache web server) running on specific servers (e.g., Fluentd<sup>4</sup> and Logstash.<sup>9</sup>) They are accompanied by tools such as Elasticsearch<sup>3</sup> for ingesting log information into persistent storage and tools such as Kibana<sup>7</sup> for querying log information.

Collecting the logs of components realized using containers such as those from Docker<sup>2</sup> and orchestrated by systems such as Kubernetes,<sup>8</sup> however, is more challenging because there is no longer a specific program and a specific sever. This is because a component consists of many anonymous instances (replicas) that are scaled up and down in number depending on the system load.

Furthermore, there is no specific server because each replica is run on a server chosen by the orchestrator.

This article looks at how to overcome these challenges by describing how cluster-level log aggregation and inspection can be implemented on the Kubernetes orchestration framework. A key aspect of the approach described here is its exploitation of the same abstractions that are used to compose and manage the system to be logged to also build the logging infrastructure itself. This approach makes use of using existing open source tools such as Fluentd, Elasticsearch, and Kibana, which are deployed inside containers and orchestrated to collect the logs of the other containers running in a cluster.

#### A BRIEF INTRODUCTION TO KUBERNETES

This article describes just enough of the Kubernetes system to help motivate a log collection and aggregation scenario for a simple application. A comprehensive description of the Kubernetes container orchestrator can be found on its website,<sup>8</sup> and an overview article on Borg, Omega and Kubernetes is available on *acmqueue*.<sup>1</sup>

The Kubernetes system can orchestrate components of applications on a variety of public clouds as well as private clusters. In this article an application is deployed on a Kubernetes cluster created on a collection of VMs (virtual machines) running on the public cloud Google Compute Engine.<sup>5</sup> A cluster could have been created using Google Container Engine (GKE),<sup>6</sup> which automates many aspects of cluster creation and management. To emphasize the provider-agnostic nature of the approach, we illustrate the explicit creation of a Kubernetes cluster that performs

log collection and aggregation with open-source tools. Either explicit cluster creation or creation using a cluster management system like GKE allows us to perform log collection and aggregation with open-source tools, although GKE allows for tighter integration with Google’s proprietary cloud logging system.

Figure 1 shows the deployment of a four-node Kubernetes cluster that is used for the example application described in this article. This cluster has four worker VMs called `kubernetes-minion-08on`, `kubernetes-minion-7i2t`, `kubernetes-minion-917k`, and `kubernetes-minion-ei9f`. A fifth Kubernetes master VM orchestrates work onto the other VMs. For work scheduled on this cluster by Kubernetes, however, you should remain oblivious to the name or IP address of the particular node that is used to run the applications since this is one of the details that is abstracted by Kubernetes. *You don’t know the name of the machine running our program.* Furthermore, the components of the application will scale up and down in size as the system evolves and deals with failure, so one logical component may execute across many different machines. *The name of the machine[s] running your program may change.*

# 1

FIGURE 1: KUBERNETES CLUSTER RUNNING ON GOOGLE COMPUTE PLATFORM

Name	Zone	Disk	Network	is scale by	External IP
kubernetes-master	us-central1-f	kubernetes-master	default		104.197.216.233
kubernetes-minion-08on	us-central1-f	kubernetes-minion-08on	default		104.197.75.50
kubernetes-minion-7i2t	us-central1-f	kubernetes-minion-7i2t	default		104.197.32.209
kubernetes-minion-917k	us-central1-f	kubernetes-minion-917k	default		104.197.240.237
kubernetes-minion-ei9f	us-central1-f	kubernetes-minion-ei9f	default		104.154.55.190

Consequently, in the Kubernetes model it does not make sense to think of a specific program  $P$  running on a specific machine  $M$ . It is far more idiomatic to identify parts of the system by making queries over labels that are attached to anonymous entities created by the Kubernetes orchestrator, which will return the currently running entities that match the query. This allows us to talk about a dynamically evolving infrastructure without mentioning the names of specific resources.

### A Music Store Application

A Kubernetes deployment of a hypothetical music store application is used to help describe how cluster-level container logs can be collected. The application has several front-end microservices that accept HTTP requests to a web interface for browsing and buying music. These front-end services work by communicating with a back-end MySQL instance and a Redis cluster that provide the persistent storage needed by the application. A persistent disk hosted on Google Compute Engine also provides the storage needed by the MySQL database.

### LOGGING PODS

The basic unit of deployment in Kubernetes is a pod. A pod is the specification of resources that should always be allocated together as an atomic unit onto the same node along with other information that a cluster orchestrator can use to manage the pod's behavior. The music store application uses one pod to describe the deployment of the MySQL instance as shown in the YAML file [albums-db-pod.

yaml [<https://github.com/satnam6502/logging-acm-queue/blob/master/albums-db-pod.yaml>]:

```
apiVersion: v1
kind: Pod
metadata:
  name: albums-db
  labels:
    app: music1983
    role: db
    tier: backend
spec:
  containers:
  - name: mysql
    image: mysql:5.6
    env:
      - name: MYSQL_ROOT_PASSWORD
        value: REDACTED
    ports:
      - containerPort: 3306
    volumeMounts:
      - name: mysql-persistent-storage
        mountPath: /var/lib/mysql
  volumes:
  - name: mysql-persistent-storage
    gcePersistentDisk:
      pdName: albums-disk
      fsType: ext4
```

This specification can be used to create a deployment of the music store database:

## 2

FIGURE 2: A DEPLOYMENT OF THE MYSQL ALBUMS DATABASE



```
$ kubectl create -f albums-db-pod.yaml
```

Figure 2 illustrates a deployment of this pod, which has the name `albums-db` and a pod IP address of `10.240.0.5`. It runs on the Google Compute Engine VM called `kubernetes-minion-917k`, contains a Docker image of a MySQL instance, and uses a persistent disk on Google Compute Engine called `albums-disk`. Three labels identify the application `music1983`, the role `db`, and the tier `backend`. The pod exposes the port `3306` serviced by the MySQL Docker instance for use by other components in the same cluster through the address `10.240.0.5:3306`.

Inside the Kubernetes cluster, you can connect to this database, populate it, and make queries. For example:



```
$ mysql --host=10.240.0.5 --user=NAME --password=REDACTED albums
mysql> select * from pop where artist = 'Pink Floyd';
+-----+-----+-----+-----+
| artist      | album                      | inventory | released |
+-----+-----+-----+-----+
| Pink Floyd | Dark Side of the Moon      | 57       | 1973    |
| Pink Floyd | The Wall                    | 103      | 1983    |
+-----+-----+-----+-----+
2 rows in set (0.08 sec)
```

The logs for a pod can be extracted using the Kubernetes command-line tool:

```
$ kubectl logs albums-db
...
2016-03-01 00:43:20 1 [Note] InnoDB: 5.6.29 started; log sequence
number 1710893
2016-03-01 00:43:20 1 [Note] Server hostname (bind-address): '*';
port: 3306
2016-03-01 00:43:20 1 [Note] IPv6 is available.
...
```

This command fetches the logs for the currently running MySQL Docker image. You can ask Kubernetes to report the Docker container ID for the running MySQL instance:

```
$ kubectl describe pod albums-db | grep "Container ID"
Container ID:   docker://38ab5c9e9aa8004e9b61f19885...
```

Does this solve the problem of collecting logs from an application deployed on a Kubernetes cluster? One problem is that during the lifetime of a pod the underlying Docker container (or containers) that is deployed may

terminate and new replacement containers created (e.g., to deal with a container that has failed in some way). The following induces a failure by sabotaging the MySQL container and seeing how Kubernetes responds. This is done by SSH'ing to the Google Compute Engine VM that is running the container in order to kill the MySQL Docker container.

```
$ gcloud compute --project "kubernetes-6502"
ssh --zone "us-central1-b" "kubernetes-minion-
917k"
$ sudo -s
# docker ps
CONTAINER ID          IMAGE
38ab5c9e9aa8         mysql:5.6
# docker kill 38ab5c9e9aa8
38ab5c9e9aa8
# docker ps
CONTAINER ID
abdfca342daa         mysql:5.6
```

Soon after, an agent running on the node that is part of the Kubernetes system noticed the container was no longer running. In order to drive the current state of the system to the desired state, a new Docker instance of MySQL is created with a container ID that starts with `abdfca342daa`. Checking the logs of `albums-db` now reveals:

```
$ kubectl logs albums-db
2016-03-01 01:33:25 0 [Note] mysqld (mysqld 5.6.29) starting as
process 1 ...
...
2016-03-01 01:33:25 1 [Note] InnoDB: The log sequence numbers
1710893 and 1710893 in ibdata files do not match the log se-
quence number 1710903 in the ib_logfiles!
2016-03-01 01:33:25 1 [Note] InnoDB: Database was not shutdown
normally!
2016-03-01 01:33:25 1 [Note] InnoDB: Starting crash recovery.
...
2016-03-01 01:33:25 1 [Note] InnoDB: 5.6.29 started; log sequence
number 1710903
2016-03-01 01:33:25 1 [Note] Server hostname (bind-address): '*';
port: 3306
```

The logs are now for the currently running container [abdfca342daa], and the logs for the previous instance of the MySQL container [38ab5c9e9aa8] have been lost. The lifetime of these logs is determined from the lifetime of the underlying Docker container rather than the lifetime of the pod. What is really needed is a mechanism for collecting and storing all the log information that was generated by every container instance that runs as part of this pod's execution lifecycle.

### Logging Pods Managed by Replication Controllers

Although a single pod in a Kubernetes cluster can be specified and deployed, it is far more idiomatic to specify a replication controller that creates many replicas of a pod. Here is an example of a replication controller that

specifies the deployment of two Redis slave pods (redis-slave-controller.yaml [https://github.com/satnam6502/logging-acm-queue/blob/master/redis-slave-controller.yaml]):

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: redis-slave
  labels:
    app: music1983
    role: slave
    tier: backend
spec:
  replicas: 2
  template:
    metadata:
      labels:
        app: music1983
        role: slave
        tier: backend
    spec:
      containers:
      - name: slave
        image: redis
        resources:
          requests:
            cpu: 300m
            memory: 250Mi
        ports:
        - containerPort: 6379
```

This specification declares a replication controller called `redis-slave`, which has three user-defined labels of metadata that are attached to each pod it creates. The labels identify the name of the overall application `music1983`, the role of the Redis instance `slave`, and this pod as being a member of the `backend` tier. The initial number of replica pods is set to two, although this number may be dialed up or down later. Each pod to be replicated consists of a Redis Docker container, an exposed port `6379` over which the Redis protocol operates, and some resource requests for CPU and memory utilization that are communicated to the scheduler. This specification can be given to the Kubernetes command-line tool to bring the Redis slave pods to life:

```
$ kubectl create -f redis-slave-controller.yaml
```

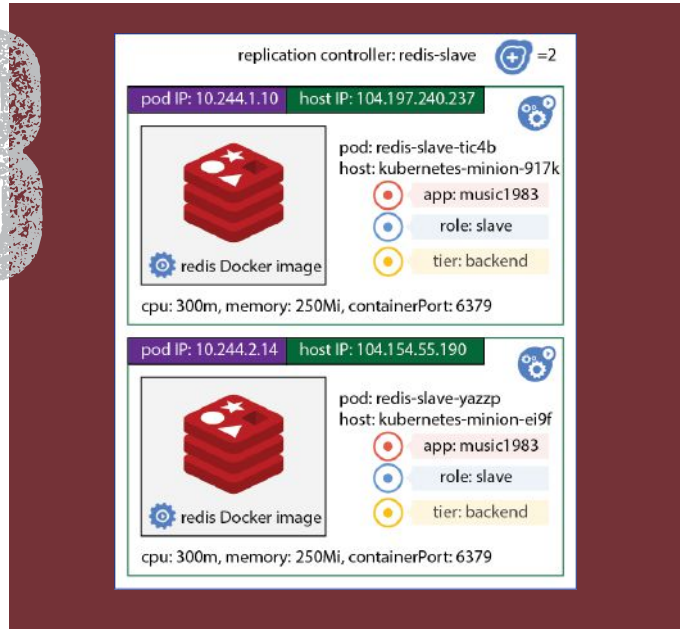
Figure 3 shows a sample deployment of such a Redis slave controller with two pods running on two different Google Compute Engine VMs. The pods have automatically generated names: `redis-slave-tic4b` and `redis-slave-yazzp`. Do not get too attached to the name of any specific pod, since pods may come and go as a result of failure or changes in the cardinality of the replication controller.

### Logging Pods Captured by a Service Specification

Each pod has its own IP address, and the IP address of the host VM is also shown, although this address is never of any interest to the Kubernetes application running on the cluster. If you can't utter the name of a specific pod, then how can you interact with it? Label selectors can define an

# 3

FIGURE 3: DEPLOYMENT OF THE REDIS SLAVE REPLICATED PODS



entity called a *service*, which introduces a *stable name* for a collection of resources. Requests sent to the stable name provided by the service are automatically routed to a pod that matches the net cast by the service label selectors. Here is the definition of a service identifying pods that provide the Redis slave functionality (redis-slave-service.yaml [<https://github.com/satnam6502/logging-acm-queue/blob/master/redis-slave-service.yaml>]):

```
apiVersion: v1
kind: Service
metadata:
```

```

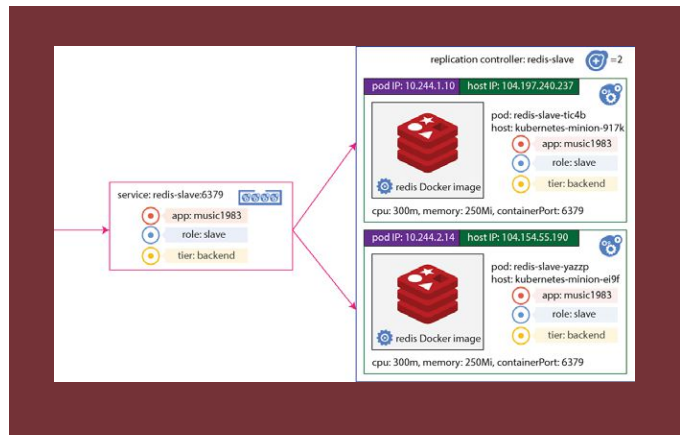
name: redis-slave
labels:
  app: music1983
  role: slave
  tier: backend
spec:
  ports:
  - port: 6379
  selector:
    app: music1983
    role: slave
    tier: backend

```

The deployment of this service is illustrated in figure 4. The service defines a DNS (Domain Name System)-resolvable name within the cluster `redis-slave`, which accepts requests on port 6379 and then forwards them to

**FIGURE 4: SERVICE MAPPING REQUESTS TO REDIS READ SLAVES**

# 4

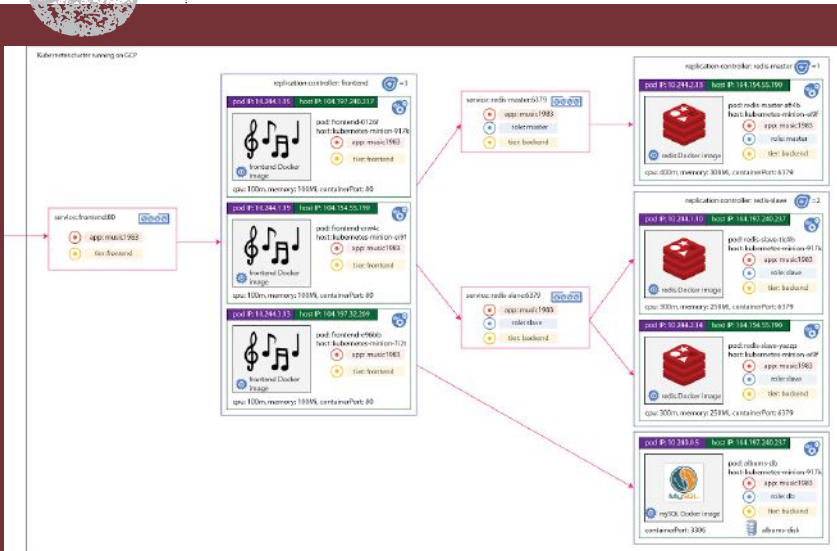


# 5

any pod that matches its label selectors (i.e., any pod that has the app label set to `music1983`, the role label set to `slave`, and the tier label set to `backend`). Now consumers of the `redis-slave-read-replicated` pods are insulated from the names of the specific pods that are used to service their requests as well as the names of the specific nodes on which these pods are running.

Figure 5 shows the deployment of a music store website made up of several front-end microservices that accept external requests and render a web user interface. These front-end services store information in a key/value store implemented by several instances of the Redis key/

FIGURE 5: A KUBERNETES DEPLOYMENT OF A MUSIC STORE SERVICE





value store. The system is designed to make it easy to independently scale up the capacity for (a) serving web traffic; (b) reading from the key/value storage system; (c) writing to the key/value storage system. As more users connect to the music store website, the number of front-end microservices can be dialed up. Typically, you expect many more relatively cheap read operations than expensive write operations to the key/value store. To process read operations as quickly as possible, reads from Redis slave instances (two in this case) are serviced and a separate pool of Redis microservices deployed as masters that perform write operations (initially just one in this case).

Collecting the logs of the front-end service pods brings up another life-cycle issue. It is not enough to just collect the logs from each of the three currently running pods (even when collecting the logs of multiple invocations of the front-end Docker image), because pods themselves may be terminated and then reborn (possibly on a different host machine). In certain situations, there may briefly be more than three front-end pods or perhaps fewer than three. If this occurs, the Kubernetes orchestration system will notice and create or kill pods to drive the system to the declared state of having just three front-end pods. As front-end pods come and go, you want to collect all of their logs, so the log-collection activity has a lifetime that is associated with the front-end replication controller rather than the lifetime of a specific pod.

### Using Fluentd to collect node-level logs

The open-source log aggregator Fluentd is used to collect

the logs of the Docker containers running on a given node. Trying to run an instance of a Fluentd collector process directly on each node (i.e., GCE VM) generates the same deployment problems that pods were created to solve (e.g., dealing with failure and performing updates). Consequently, node-level log aggregation of Docker containers is actually implemented from a Docker container that runs as part of a pod specification. This meta-approach allows the logging layer to benefit from the same advantages afforded to the application layers by the Kubernetes model for managing deployment and life cycle events. For example, the rolling update mechanism of Kubernetes can update the pods running on each node so they use an updated version of the log-aggregation software while the cluster is still running.

The Fluentd collectors do not store the logs themselves. Instead they send their logs to an Elasticsearch cluster that stores the log information in a replicated set of nodes. Again, rather than running this Elasticsearch cluster directly “on the metal,” you can define pods that specify the behavior of a single Elasticsearch replica, then define a replication controller to specify a collection of Elasticsearch nodes that contain the replicated log information and provide a query interface, and finally define a service that provides a stable name for balancing queries to the Elasticsearch cluster.

The complete specification of the Fluentd node-level collector pods is shown here (`fluentd-es.yaml` [<https://github.com/kubernetes/kubernetes/blob/master/cluster/saltbase/salt/fluentd-es/>]):

```
apiVersion: v1
kind: Pod
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
spec:
  containers:
  - name: fluentd-elasticsearch
    image: gcr.io/google_containers/fluentd-elasticsearch:1.11
    resources:
      limits:
        cpu: 100m
    args:
    - -q
    volumeMounts:
    - name: varlog
      mountPath: /var/log
    - name: varlibdockercontainers
      mountPath: /var/lib/docker/containers
      readOnly: true
  terminationGracePeriodSeconds: 30
  volumes:
  - name: varlog
    hostPath:
      path: /var/log
  - name: varlibdockercontainers
    hostPath:
      path: /var/lib/docker/containers
```

This specifies a node-level collector that runs a

specially built Fluentd image configured to send logs to an Elasticsearch cluster using the DNS name and port `elasticsearch-logging:9200` (which is itself implemented as a Kubernetes service). The specification also describes how the location of the Docker logs on the node-level file system are mapped to the file system inside the Docker container run by the pod. This allows the logs of all the Docker containers on the node to be collected by this Fluentd instance running inside this container.

When a Kubernetes cluster is configured to use logging with Elasticsearch as the data store, the cluster creation process instantiates a log-collector pod on each node. These pods can be observed in the `kube-system` namespace:

```
$ kubectl get pods --namespace=kube-system
NAME                                READY STATUS  RESTARTS AGE
fluentd-elasticsearch-kubernetes-minion-08on 1/1   Running 0      16d
fluentd-elasticsearch-kubernetes-minion-7i2t 1/1   Running 0      16d
fluentd-elasticsearch-kubernetes-minion-917k 1/1   Running 0      16d
fluentd-elasticsearch-kubernetes-minion-ei9f 1/1   Running 0      16d
...
```

A special process on each node makes sure that one of these log-collection pods is running on each node. If a log-collector pod fails for any reason, a new one is created in its place. These pods collect the logs of the locally running Docker containers and ingest them into an Elasticsearch Kubernetes service running in the `kube-system` namespace.

## Using Elasticsearch to Store and Query Cluster Logs

A cluster created using Elasticsearch for the storage of logs will by default instantiate two Elasticsearch instances. The specification for these Elasticsearch logging pods can be found at `es-controller.yaml` [<https://github.com/kubernetes/kubernetes/blob/master/cluster/addons/fluentd-elasticsearch/es-controller.yaml>], which describes a replication controller for the Elasticsearch instances as well as the actual configuration of the Elasticsearch logging pods. These can be observed in the `kube-system` namespace:

```
$ kubectl get pods --namespace=kube-system
NAME                                READY   STATUS    RESTARTS   AGE
elasticsearch-logging-v1-7rmo3      1/1    Running   0           16d
elasticsearch-logging-v1-v7lmv      1/1    Running   0           16d
...
```

The node-level log-collection Fluentd pods do not speak directly to these Elasticsearch pods. Instead, they connect to the DNS name `elasticsearch-logging:9200`, which is implemented by an Elasticsearch Kubernetes service `es-service.yaml` [<https://github.com/kubernetes/kubernetes/blob/master/cluster/addons/fluentd-elasticsearch/es-service.yaml>]:

```
apiVersion: v1
kind: Service
metadata:
  name: elasticsearch-logging
  namespace: kube-system
```

```
labels:
  k8s-app: elasticsearch-logging
  kubernetes.io/cluster-service: "true"
  kubernetes.io/name: "Elasticsearch"
spec:
  ports:
  - port: 9200
    protocol: TCP
    targetPort: db
  selector:
    k8s-app: elasticsearch-logging
```

You can observe this service running in the kube-system namespace:

```
$ kubectl get services --namespace=kube-system
NAME                                CLUSTER_IP      EXTERNAL_IP      PORT(S)
elasticsearch-logging              10.0.8.117      <none>            9200/TCP
...
```

Elasticsearch can be queried for the logs of all pods that are captured by the label selectors for the front-end service. A local proxy allows you to connect to the cluster with administrator privileges, which are required to retrieve the logs of running containers. You query for just the logs of containers that are marked with a `container_name` field of `frontend-server`.

```
$ kubectl proxy
<elsewhere>
$ curl -XGET "http://localhost:8001/api/v1/proxy/namespaces/
kube-system/services/elasticsearch-logging/_search?q=container_
name:frontend-server&_source=false&fields=log&pretty=true"
...
  }, {
    "_index" : "logstash-2016.02.26",
    "_type" : "fluentd",
    "_id" : "AVMa-C0pcuStSsThK0M4",
    "_score" : 2.8861463,
    "fields" : {
      "log" : [ "Slow read for key k103: 192 ms" ]
    }
  }
...
  }, {
    "_index" : "logstash-2016.02.26",
    "_type" : "fluentd",
    "_id" : "AVMa-C0pcuStSsThK0NE",
    "_score" : 2.8861463,
    "fields" : {
      "log" : [ "[negroni] Started GET /!range/k336" ]
    }
  }
...
  }, {
    "_index" : "logstash-2016.02.26",
    "_type" : "fluentd",
    "_id" : "AVMa-C_fcuStSsThK00p",
    "_score" : 2.8861463,
    "fields" : {
      "log" : [ "Slow write for key k970: 187 ms" ]
    }
  }, {
...

```

Since the Elasticsearch cluster is a collection of pods managed by a replication controller, it can deal with an increased query load to the logging system by simply increasing the number of replica nodes for the Elasticsearch logging instances. Each pod contains a replica of the ingested logs so if one pod dies for some reason (e.g., the machine it is running on fails), then a new pod will be created to replace it, and it will synchronize with the running pods to replicate the ingested logs.

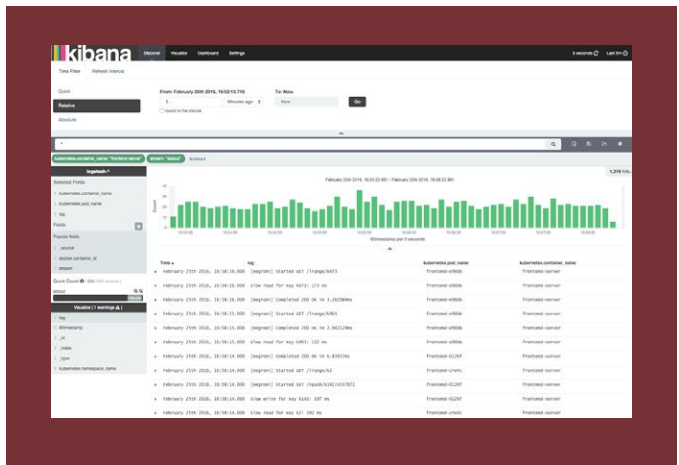
### VIEWING LOGS WITH KIBANA

The aggregated logs in the Elasticsearch cluster can be viewed using Kibana. This presents a web interface, which provides a more convenient interactive method for querying the ingested logs, as illustrated in figure 6.

The Kibana pods are also monitored by the Kubernetes

# 6

FIGURE 6: **QUERYING INGESTED LOGS USING KIBANA**





system to ensure they are running healthily and the expected number of replicas are present. The life cycle of these pods is controlled by a replication-controller specification similar in nature to how the Elasticsearch cluster was configured. The following output shows the cluster configured to maintain two Elasticsearch instances and one Kibana instance. If system load increases, a simple command can be issued to dial up the number of Elasticsearch and Kibana replicas. Furthermore, the number of Elasticsearch replicas can be scaled up independently of the number of Kibana instances, allowing you to respond to increases in different kinds of loads by scaling up only the subcomponents needed to meet that demand.

#### SUMMARY

Collecting the logs of containers running in an orchestrated cluster presents some challenges that are not faced by manually deployed software components. In particular, we cannot explicitly identify by name a particular container (or the name of the pod in which it is contained), nor the node that container is running on, because both of these may change during the lifetime of the deployed application. As application components (microservices) come and go, we need to gather and aggregate all the logs of the containers that work as part of the application during its life cycle. This challenge is addressed by the use of label-selector queries to identify which running activities belong to the application of interest at any given moment. Then these queries can be used (by way of a Kubernetes service) to query the logs of a dynamically evolving application.

The basic infrastructure needed to implement log aggregation and collection can itself be implemented using the same abstractions used to compose and manage the applications which need to be logged: pods, replication controllers, and services. This allows for adapting the capacity of the logging system and updating it while it is running as well as robustly dealing with failure. This also provides a model for developing other cloud computing system infrastructure components in a modular, flexible, reliable, and scalable manner.

## References

1. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J. 2016. Borg, Omega, and Kubernetes. *Acmqueue* 14(1); <http://queue.acm.org/detail.cfm?id=2898444>.
2. Docker; [www.docker.com](http://www.docker.com).
3. Elasticsearch. Elastic; <https://www.elastic.co/products/elasticsearch>.
4. Fluentd; <http://www.fluentd.org/>.
5. Google Compute Engine; <https://cloud.google.com/compute/>.
6. Google Container Engine; <https://cloud.google.com/container-engine/>.
7. Kibana. Elastic; <https://www.elastic.co/products/kibana>.
8. Kubernetes; <http://kubernetes.io/>.
9. Logstash. Elastic; <https://www.elastic.co/products/logstash>.

**Satnam Singh** [[s.singh@acm.org](mailto:s.singh@acm.org)] is a software engineer at Facebook working on mobile performance. Previously he worked at Google on the Kubernetes project.

Copyright © 2016 held by owner/author. Publication rights licensed to ACM.



**HERE'S TO ADI, RON AND LEN.  
FOR GIVING US RSA PUBLIC-KEY  
CRYPTOGRAPHY.**

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.**