Issue January 2014 | presented by jaxenter www.jaxenter.com

jaxmagazine

#35

Into the wormhole

A fresh new look at big data

Parallel processing on the Grid

Wrangling traditional cache semantics

Spinning a YARN

Developing Distributed Applications with Apache Twill

Get started with GitHub

Guru Brent Beer shares his wisdom

Image Licensed by Ingram Image

Java and beyond in 2014

First of all, a very warm welcome to the very first JAX Magazine of 2014! This year, we can expect the Java resurgence to carry on apace – especially with Java 8 (at long last) just around the corner – and we cannot to hear what you all have to say about the finished product this spring. Elsewhere in the amorphous, fuzzy area we at JAX fondly like to think of as the "Javasphere and all the floaty intertwined bits relevant to Java and JVM devs", there's also plenty happening.

You're certainly going to be hearing a lot more about reactive programming this year. In fact, according to Typesafe's Jamie Allen, the programming model defined in the Reactive Manifesto will be **the** key software development paradigm of the year.

In the big data space, the ongoing trickle to NoSQL will likely continue, much to the chagrin of traditional relational database leaders. There's definitely something of an ongoing paradigm shift in terms of how people are thinking about how the store their information – with one study showing that graph databases are gaining popularity faster than any other category in the field.

NoSQL providers have made steady inroads in preparing markets around the world, especially in Europe, for this new movement in data storage. And, with companies expected to collect more data than they could ever analyze in 2014, the time is right for a revolution. Finally, expect to waves of IoT hype continue to build. Malware has already trumped rancid milk as the thing we least want to find in our fridges this year – and that's just the start. Everyone is scrambling to climb on the bandwagon at the moment, and, for now, a slice of the M2M pie is everyone's for the taking.

But enough of this vague crystal ball gazing. As ever, we've got a host of dev friendly, insightful features lined up for you. Andreas Neumann and Terence Yim have put together an expert guide to developing Distributed Applications with Apache Twill, and Steve Millidge will be showing us how to wrangle cache semantics into submission. We've also got a chat with GitHub trainer Brent Beer, and JAX Editor Elliot Bentley goes tunnelling for small worms in big data.

Lucy Carey, Editor

Index

Let a GitHub trainer knock down your barriers to collaboration 4
Time to douse those internal firewalls and open-source your internal projects – it'll make things so much simpler.

Small worms and big data 7
How Java helped OpenWorm wriggle to life
Elliot Bentley

Developing Distributed Applications with Apache Twill 9
Analytics worth spinning
Andreas Neumann and Terence Yim

Processing on the Grid 16
Why do traditional cache semantics sometimes struggle to scale?
Steve Millidge

Dart on the mark

Since the release of a production-ready Dart 1.0 last November, the race has been on to convince developers of its appeal. In recent months, dart2js compiler has begun to outshine hand-written JavaScript. Of the four key benchmarks being tracked by the Dart team, the Dart-generated JavaScript is considerably faster in the "Tracer" test, and JavaScript and dart2js are currently neck-and-neck in "FluidMotion". It remains to be seen if such increases are enough to lure users of existing preprocessing language users. The truth though? It doesn't really matter: Dart is designed for teams building Google-sized front-end projects, and could become popular within this niche without being widely adopted.

Sticking to deadlines

It's been a long (much longer than anticipated), and bumpy road, but it appears that Java 8 is really coming in March – albeit with a few rough edges. In an update published this January, Mathias Axelsson, Oracle JDK 8 Release Manager, wrote that for the past few months, the devs team had been furiously beavering away in Oracle Towers to get the upgrade pushed out this spring. Don't expect perfection though. At this stage of development, only "showstopper bugs" are being considered for fixing in the initial JDK 8 release. But, moth-eaten or not, at least it'll be here.

Java skills = Cash/money

There may be flashier, easier languages out there, but, for now, according to Stack Overflow's calculations at least, "heritage" programming lingo is still where the big bucks lie. In 2013, Java was the most searched for skill keyword by recruiters, and almost a quarter of all employers' searches were for Java developers. Of all keywords searched for on Careers 2.0, Java was far and away the most in-demand, contained within 22.26 percent of all search queries. Compare this to the second most searched-for skill, PHP, which had almost half that number. Proof that there's still gold in the old school – as any Rolling Stone would attest.

Malware misadventures

Oh dear. Clearly someone at Oracle didn't get the memo about, "Out with the old, in with the new." as January 2014 broke that, thanks to an exploit in Yahoo.com's advertising network (Ads.yahoo.com), thousands of visitors had been served up Java-based malware. So, the new year started on exactly the same note as the last one ended – with the platform mired in contempt from security experts, and exasperation from users. The attack appears to have been largely confined to European PCs – although this probably comes of scant comfort to the thousands of potentially compromised users.

Rough patches

A small security fix in Java update 51, released this January, appears to be incompatible with the popular Google Guava library. However, Oracle have so far refused to revert the change, which according to the changelogs was meant to "enhance generic classes". Uanfortunately, to Google Guava and the many projects that rely on it, it is indeed an issue, breaking an essential part of the widely-used library. Regardless of who is in the right, until the issue is resolved Guava users might want to think twice before updating to the newest Java release.

NOT

Time to douse those internal firewalls and open-source your internal projects – it'll make things so much simpler.

Let a GitHub trainer knock down your barriers to collaboration

Wouldn't life be so much easier if you could treat internal projects at your company as if they were open source? Well, you can! Just like a business, open source works through similarly aligned teams and individuals – you just need to know a few fundamentals to make it all come together. In this article, Brent Beer explains how, by using patterns implemented at GitHub, you can make your in-house development more enjoyable, and easier to get involved with and contribute to.

JAX Magazine: How did you became involved with GitHub?

Brent Beer: I first became really interested in GitHub because a lot of my friends were using it. That got me initially signed up, then, once I started becoming more involved in different programming communities – whether it be web application frameworks, or experimenting building stuff myself – I started looking at how other people were building tools, and what they were using to build applications. I found out that everything they were using was also on GitHub, and any tutorials, or anything like that – anything that I would go through to find out, OK, well, I want to try out this new thing that just came out with this language, where's that code hosted? Oh, that's on GitHub as well. So everything just kept pointing me to GitHub, and I became immersed in the environment, I guess, to use GitHub.

Now I'm on board and in the company, seeing firsthand the amount of information that goes into GitHub, the rapid amount of coding that happens, is awesome.

JAXmag: What keeps you motivated to keep using GitHub now that you are not so involved in coding?

Beer: At this point, though I see things on Twitter, essentially I rely more on my co-workers. We heavily use our chat service – we use Campfire to all stay well connected as we're an incredibly distributed team, in an incredibly distributed company. So, when I see other people within the company talking about projects and tools and stuff that they use, that gets me interested on things that are happening on GitHub.

I'm also pretty closely connected with the team that's working on the "Explore" features of GitHub, so Jon Rohan and Andrew Nesbitt. They're working a lot with the features that will allow people to find things easier on GitHub. They find things and they show them off, or talk about them, and they're just building features to make that stuff easier to find. So that helps as well.

We also have a daily or a weekly newsletter. That actually comes out from people. Trending repositories on GitHub is also really helpful for me, just to see what the community at

Portrait

Brent Beer has been a passionate user of Git and GitHub for a number of years beginning in university where he constantly tried to adopt the way students collaborated. After graduation, Brent moved out to work in San Francisco to work as a web developer and later became a member of the GitHub Training Team.

large is up to. As you start following more and more people on GitHub, that newsletter also uses your friends, and you can see what they're messing with too. So that also really helps me.

"Seeing the amount of information that goes into GitHub…is awesome"

JAXmag: Three main pieces of advice you would give to anyone looking to enhance team collaboration within their company?

Beer: My first takeaway would be, if you're starting out a new company, or if you already have an existing company, one of the first things you have to do is have many, many, many channels of communication. So you have to be able to talk about things. Now, email works for this sometimes, but we built our entire company around a chat service, essentially. When we started out we didn't have an office – we had a chat room.

We had people in other parts of the country, and obviously we can't see them face to face, but we see the way they talk, and the way they work, come out through a chat. So Campfire, HitChat, Skype has a group chat service as well ... just so long as you have everyone able to share information, and collaborate and talk to each other, through some service. I find that helps a lot – so that's my first point.

Secondly, having a way for people to be able to explore different projects which are going on within your company is super helpful as well. If there are different projects that other people are working on in a different team to you, it's great to be able to explore them. And when you do, having that open channel of communication let's you tell your existing team if you're going to help out on that project for a little while, or to work on that new idea, or come back to work on that team later.

We've seen this with some people at GitHub specifically, like Sean Bryant, who works on the enterprise project itself. But he also worked with our 3D rendering team to work on 3D disks within GitHub and stuff like that. He took a short, you could call it vacation, from what he normally does, worked on something else, and then later came back to working on enterprise again. He was able to do that because he kept the channels of communication open with his team, he let us know what he was doing, and everything was fine from there.

Lastly, when you do have these projects within your company that you want people to collaborate with you on, whether they're on your team or not, you should make it easy for them to get started. So, if I'm going to jump on to a project, the first thing I'm going to want to do is read the install instructions. I want to get started as easily as possible.

Having a good readme is kind of the first step there, talking about how to get started on something, how to use it, and, once people are using something, making sure that their development environment is also very easy to set up helps as well.

We have a tool called BoxIn which is kind of a tool that allows your new employees to get started on getting up to date and started with their entire laptop, or if I myself completely lost my laptop, I could just get a new one, and with a few short commands, everything would reinstall the exact same way. All my development environments, and everything I would need to actually code on any projects, would already be set up for me. So BoxIn and those readmes make it nice.

Just to tie everything together; if it's really easy for people to jump into projects, you can talk to other people to bring them into those projects with very little overhead, and they can start helping you out and bringing in their own diverse experience to work with you.

JAXmag: What's the GitHub way of programming?

Beer: We have this way of programming at GitHub called the GitHub Flow – and it's really closely tied to our features of using pull requests and keeping those channels of communication open. It's the idea of, if you're going to work on some feature, to get it out in front of everyone, for them to see and talk about and collaborate with you, as soon as possible.

If I'm going to work on some feature, I don't want to work on it in some isolated way, on my laptop not talking to anyone for two or three months – or even two to three days. As soon as I start with the idea, I want to present that idea to my colleagues, and starting showing them the first bits of code as the conversation is starting.

This is how our pull requests are set up – there's an idea presented with a little bit of code attached, and people start talking about it, and then you update the code, and then they keep talking about it ... and you keep updating the code. That's kind of an essential way of how we get any amount of work done.

"Make your own little community for yourself, and keep exploring."

JAXmag: What do you think are the biggest obstacles to collaboration?

Beer: The hardest part about collaboration, I think, is when you don't necessarily really know the other person that well yet. So that first time understanding how they work. And tone too – that might come through with writing. From time to time, I see people who might be less familiar with typing having conversations over text, and I might not know the context of what they're talking about by the tone that they're using.

We have this unwritten rule of "assume no malice", so anyone I'm working with, especially in GitHub, I assume

they're not being mean to me. That's a good rule to have when meeting people for the first time! Nobody is going to try to be mean to you – who would do that? That makes it a bit easier.

For the open source community in general, or working on some project with someone else outside my company who I've never met before, I might do a double take sometimes and think, "What are they trying to say there?" and I'll re-read stuff. So that's the first stick in the mud that I run into on open source. But you know, it's just a matter of trying to assume the best out of people. If you use more words, it's easier to figure out how a person's tone is, or what their context is. Being more explicit in text is always a good thing.

JAXmag: What are your favourite GitHub hacks?

Beer: One of the things that I really like is within GitHub itself. We have a lot of conversations that are on the web – within the pull requests, within the issues – and the thing that I like is, when having these conversations, if you actually highlight some bit of text in an issue or pull request and just hit "r", it actually takes it and puts it down in your input box for your comment as a quote. It keeps the context of which questions I'm answering, and helps keep the conversation going.

Besides that, within our repository, if you just hit the "t" button and start searching for different files. This is kind of a fuzzy search – but if you're in a big project that has lots of different files and you might not remember which one you're looking for, you can start typing things out, and as you do so, it'll start filtering that down for you to find those files. It's kind of the same vein as search, which is also very, very helpful.

I think the explore and trending part of GitHub is the best place for new people to go. Find out the popular people on GitHub, and then also find people who are similar to you, and follow them on GitHub. If they're similar to you, they'll be starring stuff which is also very similar to your interests. So if you see the things that they've starred. you'll get those newsletters, and then get ideas, and grow your network that way. Find the interesting projects, and things like that. That's what people now, and forever, when they are new to GitHub, should do. Make your own little community for yourself, and keep exploring.

Advert

WRITE FOR US!
Developer.Press is looking for e-book authors.

Still searching for the code to success?

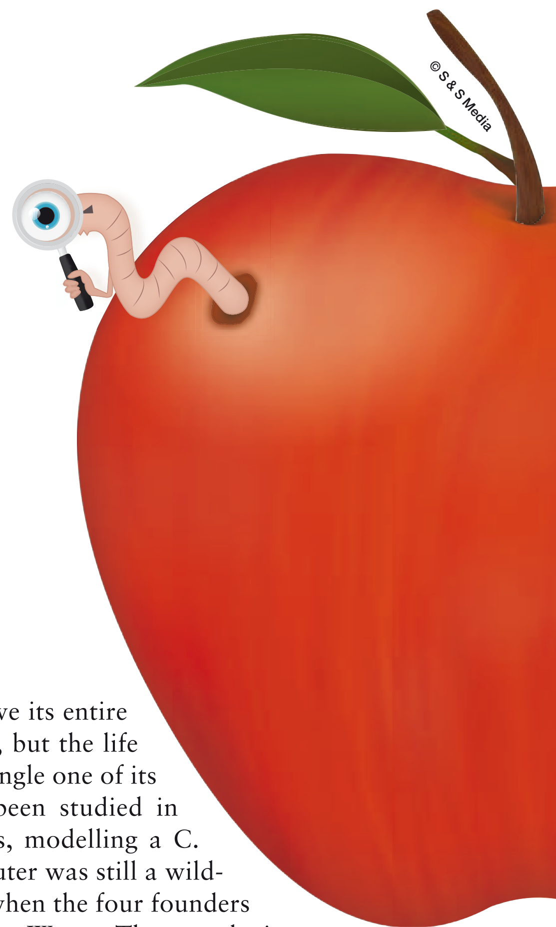
Take a shortcut.

Buy now at: www.developerpress.com

Also available to buy on: amazonkindle | Available on the iBookstore | Google play

How Java helped OpenWorm wriggle to life

Small worms and big data



by Elliot Bentley

Streaming 200 MB per second to browsers is just one of many engineering challenges faced when simulating a living organism. A video of a wriggling microscopic worm seems an unlikely news story, but over Christmas 2013, it was featured on dozens of sites including Mail Online, Engadget and BoingBoing. This was no ordinary worm, however: it was a computer simulation, the culmination of over two years' work by an international team of scientists and engineers. Together they have been building OpenWorm, "the world's first virtual organism in a computer", and this two minute long video was recorded to show off its muscle systems in action. Under the surface of this worm is an incredibly complex and entirely open source Java application built with OSGi and Spring. OpenWorm co-founder Matteo Cantarelli told JAXenter that Java was an obvious choice for a project of this scale.

"Basically, we wanted an enterprise solution, because what we're building will have to scale up a lot," he said in a phone interview. "When you compare the speed of Python and Java when looking at how big the objects in memory are, those kinds of things – for something robust and long-term, Java was what we ended up going with." Cantarelli originally trained in electrical and systems engineering, before working for around six years as a software engineer. In his spare time, however, he was exploring a very different side of software: simulating life itself. "I started a process of self-educating, just reading loads of [biology] books," says Cantarelli. Eventually, he met a group of like-minded people over the internet – or as Cantarelli puts it, "A couple of individuals who had the same dream: to simulate a C. *elegans*".

Cut it, and it'll bleed Java

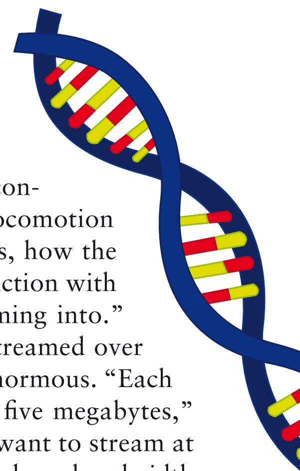
Caenorhabditis elegans, or just C. *elegans* to friends, is a microscopic nematode worm that, thanks to its prevalence and simplicity, has become one of the most-studied organisms in the world. Not only was it was the first multicellular organism to have its entire genome sequenced, but the life patterns of every single one of its 1,031 cells have been studied in depth. Despite this, modelling a C. *elegans* on a computer was still a wildly ambitious task when the four founders began work on OpenWorm. The complexity of even the simplest of organisms rivals some of the toughest problems in computer science; and to make things even more complex, the group had settled on a flexible client-server architecture that needed to be scalable.

Everything in OpenWorm is written twice – firstly by the scientist half of the team, who create models based on real-life observations in Python and C++, and then a second time by the engineering half as a highly modular Java application. "We wanted to build something with a client-server architecture, and we wanted something robust," says Cantarelli, "and Java has a very good track record as offering server-based applications. We wanted something that was, for instance, strongly typed." "It's not plain Java, because that wouldn't cut it. We used a lot of OSGi, in the sense that we want the whole architecture to be modular and independent, so we want good dependencies decoupling. And using Spring offers benefits along the same line." Eclipse Virgo was chosen because it was "one of the first web servers to allow OSGi within a web server container".

Cantarelli stresses the importance of this modularisation. Not only are systems within the worm kept separate, but the organism's data is decoupled too – providing a solid foundation with which different organisms could also be simulated. "The idea is that the architecture and the platform will be reusable entirely," says Cantarelli. "The worm will be just data."

Burrowing OpenWorm through the internet

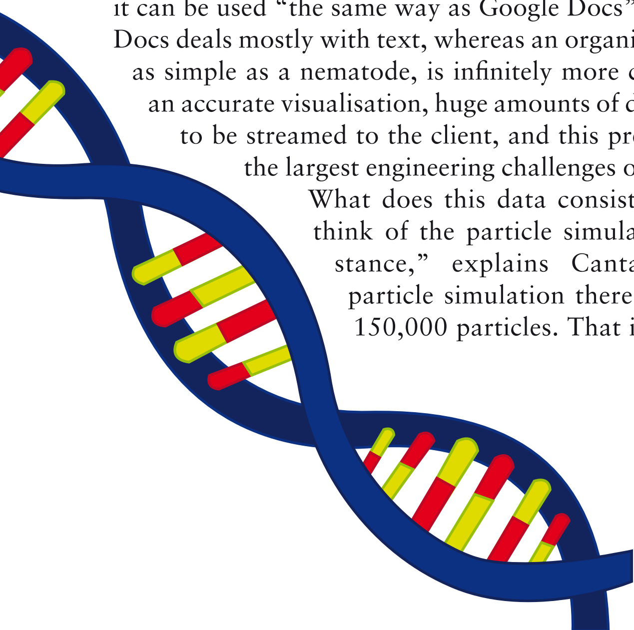
The current build of OpenWorm runs on Amazon EC2, but Cantarelli says there is ongoing research into a "more di-



verse composition for the different nodes of the network". Some aspects of the simulation might require high-performance computing, for example, while others might use a parallelised architecture. At this stage, though, no permanent decisions have been made.

OpenWorm is split like an MVC application, where the worm model is calculated on the server, rendered in a client, and user instructions sent back to the server for processing. In web browsers, the team are using JavaScript and WebGL to render the model (you can have a play with it yourself, and there's also iOS client). Cantarelli says there's already an API for manipulating the server-based simulation, but it doesn't yet have a GUI equivalent.

This client-server model may be forward-thinking, but Cantarelli describes it as an "ambitious choice". The idea, he says, is to make the worm simulation so accessible that it can be used "the same way as Google Docs". But Google Docs deals mostly with text, whereas an organism, even one as simple as a nematode, is infinitely more complex. For an accurate visualisation, huge amounts of data will need to be streamed to the client, and this presents one of the largest engineering challenges of the project.



What does this data consist of? "If you think of the particle simulation, for instance," explains Cantarelli, "that particle simulation there has roughly 150,000 particles. That is the current model, so it's not even the final model, and it is only one aspect [of the entire model].

"And by one aspect, I mean it is just concerned with the particle physics of the locomotion of the worm, so how the muscle stretches, how the body stretches, how the body wriggles, friction with the liquid or gel that the worm is swimming into." Compared to the type of data typically streamed over the web, the output of this calculation is enormous. "Each timestep is in the order of magnitude of five megabytes," says Cantarelli. "Now, imagine that you want to stream at 35 frames per second ... you're looking at a huge bandwidth [requirement]."

As a result, this highly detailed model outputs a whopping 200 MB per second – around a hundred times the bandwidth needed for 4 K video. Work hasn't yet started on optimising this stream, but Cantarelli says that it will require some creative thinking on the part of the engineers.

It's one of many technical challenges yet to be dealt with, but Cantarelli and his engineers are on track to dealing with them. In just two years OpenWorm has already grown from a four-person side project to a sprawling organisation. "I wake up every day and there's new stuff happening, and there's new people doing things, new people contributing, and it just makes me very happy," says Cantarelli. "The project now has a life of its own!"

Imprint

Publisher
Software & Support Media GmbH

Editorial Office Address
Software & Support Media Limited
24 Southwark Bridge Road
London SE1 9HF
United Kingdom
www.jaxenter.com

Editor in Chief: Sebastian Meyen
Editors: Lucy Carey
Authors: Brent Beer, Elliot Bentley, Steve Millidge, Andreas Neumann, Terence Yin
Copy Editor: Jennifer Diener
Creative Director: Jens Mainz
Layout: Flora Feher

Sales:
Timo Winter
+49 (0) 69 630089-62
twinter@sandsmedia.com

Entire contents copyright © 2014 Software & Support Media GmbH. All rights reserved. No part of this publication may be reproduced, redistributed, posted online, or reused by any means in any form, including print, electronic, photocopy, internal network, Web or any other method, without prior written permission of Software & Support Media GmbH.

The views expressed are solely those of the authors and do not reflect the views or position of their firm, any of their clients, or Publisher. Regarding the information, Publisher disclaims all warranties as to the accuracy, completeness, or adequacy of any information, and is not responsible for any errors, omissions, inadequacies, misuse, or the consequences of using any information provided by Publisher. Rights of disposal of rewarded articles belong to Publisher. All mentioned trademarks and service marks are copyrighted by their respective owners.

Image Licensed by Ingram Image

Analytics worth spinning

Developing Distributed Applications with Apache Twill

by Andreas Neumann and Terence Yim

For the past couple of decades, Java programmers have been building vast numbers of web apps generating petabytes of log data. Analyzing these web logs can create real business value – for example, by optimizing the web app for observed user behavior patterns, or by personalizing the user experience within the app. But until recently, companies have discarded their web logs because they were either too hard to analyze, or too expensive to store with traditional relational databases. Apache Hadoop [1] is a free, open source technology that runs on commodity hardware clusters. Its distributed file system (HDFS) makes it cheap to store large amounts of data, and its scalable Map/Reduce analysis engine makes it possible to extract insights from that data. Map/Reduce [2] is a flavor of batch-driven data analysis, where the input data is partitioned into smaller batches that can be processed in parallel across many machines in the Hadoop cluster. But Map/Reduce, while powerful enough to express many data analysis algorithms, is not always the optimal choice of programming paradigm. It's often desirable to run other computation paradigms in the Hadoop cluster – here are some examples:

Ad-hoc querying: The SQL language is widely known from relational databases, and many users would like to query their big data using SQL. Apache Hive [3] can execute a SQL query as a series of Map/Reduce jobs, but it has shortcomings in terms of performance. Recently, some new approaches such as Apache Tajo [4], Facebook's Presto [5] and Cloudera's Impala [6] drastically improve the performance, but require running services other than Map/Reduce in the Hadoop cluster.

Real-time Stream Processing: Due to their batch nature, Map/Reduce jobs have a high latency before results are available, often several hours. Many applications require much lower latencies. For example, a fraud detection algorithm should deliver its results much faster – if possible in real-time. Examples of real-time engines include Apache Storm [7] and Apache Samza [8]. Ideally such an engine could run in the same cluster as the Map/Reduce jobs that process the same data, to avoid duplication of data and infrastructure.

Message-Passing (MPI): MPI [9] is another scalable approach to big data analysis – it is a stateful process that runs on each node of a distributed network. The processes communicate with each other by sending messages, and alter their state based on the messages they receive. For example, it has been shown [10] that Google's PageRank algorithm can be approximated in near real-time by an MPI algorithm.

Distributed Testing: When a cluster is not fully utilized, it makes sense to use its spare resources for other purposes such as testing. For example, running all tests for the Hadoop distribution takes many hours on a single machine. If these tests can be parallelized and run on many machines concurrently, the test time can be greatly reduced. Another example of a distributed test is a load test for a web service that sends a high number of requests concurrently from many machines.

You name it: Is it possible to run such non-Map/Reduce jobs on a Hadoop cluster? For the longest time, the answer to all these questions was: No! Or, only if one had a PhD in distributed systems engineering, and figured out a way to "disguise" the jobs as Map/Reduce. This was mainly due to the fact that in Hadoop, the cluster resource management and the Map/Reduce engine were very tightly integrated with each other.

The good news is that the latest version of Hadoop decouples these two responsibilities. Hadoop 2.0's new resource manager YARN allows using the Hadoop cluster for any computing needs, by programming against YARN's interfaces and protocols. However, YARN's programming model is complex and poses a steep learning curve to developers who do not have experience with Hadoop.

Fortunately, this all changes with the recently incubated Apache Twill, which exposes YARN's power with a Java thread-like programming model. Twill also implements commonly needed patterns of distributed applications such as central log collection, service discovery and life cycle management. Read on to learn more about the history of Hadoop, YARN and how to develop distributed applications using Twill.

Hadoop and YARN

In original Hadoop, the cluster management is tightly coupled with the Map/Reduce programming paradigm. A Map/Reduce job analyzes data by dividing the work between a number of specialized tasks called mappers and reducers. Each task runs on one of the machines of the cluster, and each machine has a limited number of slots for running tasks concurrently. Hadoop's job tracker daemon is responsible for both managing the cluster's resources and driving the execution of the Map/Reduce job: it reserves and schedules slots for all tasks, configures, runs and monitors each task, and if a task fails, it allocates a new slot and reattempts the task. After a task finishes, the job tracker cleans up temporary resources and releases the task's slot to make it available for other jobs.

There are two major issues with this approach: It limits scalability, because the single job tracker daemon becomes a bottleneck and single point of failure for the many jobs in a large cluster. Additionally, specializing the execution logic to Map/Reduce makes it difficult to use the cluster for other distributed programing paradigms – such jobs have to "disguise" themselves as mappers and reducers in order to be able to run.

Hadoop 2.0 changes all of this. It has a new resource manager named YARN [11] that is decoupled from job execution. In YARN, the compute resources of a cluster are organized into "containers", where each machine can host multiple containers at the same time.

The Resource Manager (RM) maintains the state of all active containers as well as the available capacity of all nodes. In addition, every machine runs a Node Manager that is responsible for launching and monitoring containers on that machine. Every job is controlled by a separate task, the Application Master (AM). The AM itself runs in a container and communicates with the RM to request, receive and release containers for the tasks of the application, and with the Node Managers to launch tasks in those containers. It makes all logical decisions about the execution of the application, such as the number of tasks to run, the distribution of work among the tasks, the order of execution of tasks etc. This new architecture delivers better scalability because there is no single bottleneck in the management of job execution, and it provides more flexibility as to what types of applications can run in the cluster. The AM is now part of the application code, and hence every application can have its own, custom AM – the Map/Reduce AM is simply provided out of the box as part of the Hadoop distribution.

Anatomy of a YARN application

A YARN application consists of three parts: A YARN client, an application master, and the actual application tasks. Although the tasks are typically programmed in Java, a task can be an arbitrary shell command, an executable compiled from C++ code, or a Python script. The AM allocates containers for the applications tasks and controls the execution. It must continuously communicate with the RM to send a heartbeat signal. Via the same protocol, the AM can request and receive more containers from the RM, return

containers that it does not need any longer, and receive notification about containers that have terminated.

The YARN client initiates the job, requesting the RM to start the AM. While this appears to be a simple exercise, it actually turns out to be quite complex: Before the AM can be started, the client must make sure that all required resources (code, libraries, executables, configuration, etc.) are available in the container that will run the AM. How does that happen? Hadoop uses a distributed file system (HDFS) that is shared between all the machines in the cluster. The YARN client copies all required files to HDFS and informs the RM of the location. The files are then available to all machines of the cluster. Whenever a Node Manager starts a task, it copies all of the task's resources to a working directory on the local file system.

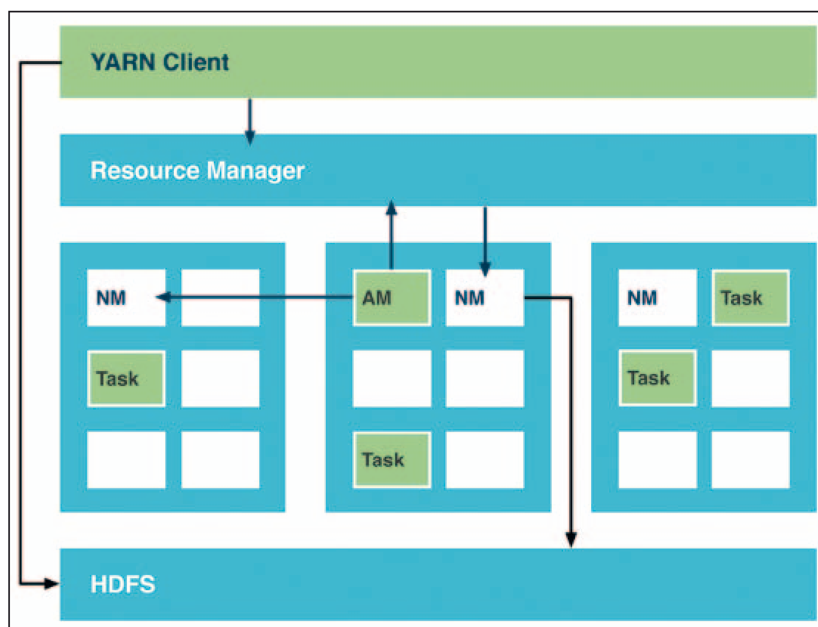


Figure 1: The execution of jobs in a YARN cluster

Let's take a closer look at how the YARN client submits a job: First it copies all of the tasks resources to a location in HDFS. Then it connects to the RM to request an application id. Using this application id, it then submits a request to the RM to start the AM. This request contains:

- The application id.
- The HDFS locations of all resources.
- Environment settings for the AM such as the class path.
- Resource limits for the AM, for example, how much memory it is allowed to use.
- The shell command to execute the AM. If the AM is a Java program, this command would be java. This command is also responsible – if desired – for redirecting standard output/error to files by using shell syntax.

After the request is submitted, the AM is started asynchronously, and the client can now check the status of the application using the application id. The RM will allocate a container on a machine that has enough capacity to run the AM. Then it requests the Node Manager of that machine to start the AM. The Node Manager creates a working directory for the AM, copies all the local resources from HDFS, executes the Application Master using the provided shell command, and starts monitoring it.

Now the AM is running in a container, but none of the actual application's tasks has been started. This is the task of the AM. It decides how many tasks to run and in what order, how many resources they are allowed to use, and how to deal with task failures. For every task it needs to start, the AM first requests a container from the RM. Then it starts the task in a way that is very similar to how the YARN client starts the AM, except that the AM sends its request directly to the Node Manager responsible for the allocated container.

The AM must regularly communicate with the RM to send a heartbeat. This is important because if this heartbeat is not received for a long time, the RM will assume that the AM has crashed and kill all the (now orphaned) tasks that belong to the application. In the same call used for the heartbeat, the AM can also request new containers and release containers that it no longer needs. In its response to the heartbeat, the RM informs the AM of new containers that have been allocated for it, and it also informs the AM of all containers that have been terminated since the last request.

Note that the RM works asynchronously, and a request for new containers is typically not fulfilled by the response to that request. The AM must keep sending its heartbeat to the RM and will eventually receive the containers it requested. It is therefore important that the AM manages its state carefully, always remembering what pending requests for containers it has, what active containers it owns etc. Eventually, all the tasks of the application are finished, and the application manager releases all containers and terminates itself. The RM can now reallocate the capacity of the released containers to other applications.

Introducing Twill

YARN has been called the "Operation System of the Data Center", and it honors this title with very generic and powerful APIs and protocols. But with that power comes complexity, and learning YARN is a steep uphill climb. The sample application that is bundled with YARN, which does nothing but execute a simple shell command in each container, sums up to more than a thousand lines of code. And due to the asynchronous nature of the YARN protocol, the AM must always carefully remember all containers it has requested, received and started, a complex task that opens up high chances for coding mistakes.

Many applications, however, do not require all the options of YARN. For example, in order to use the Hadoop cluster for a distributed load test – all that is needed is a

number of agents – each doing only one thing – sending as many requests as possible to the service being tested. If this was being implemented as a single JVM, each agent would run as a Java thread, and Java executor services make it extremely easy to manage that. Can distributed applications be made as easy as Java threads (at least for some applications)? The answer is: Yes, with Apache Twill [12], which was incubated at Apache in late 2013 with the goal of reducing the complexity of developing distributed Java applications with YARN.

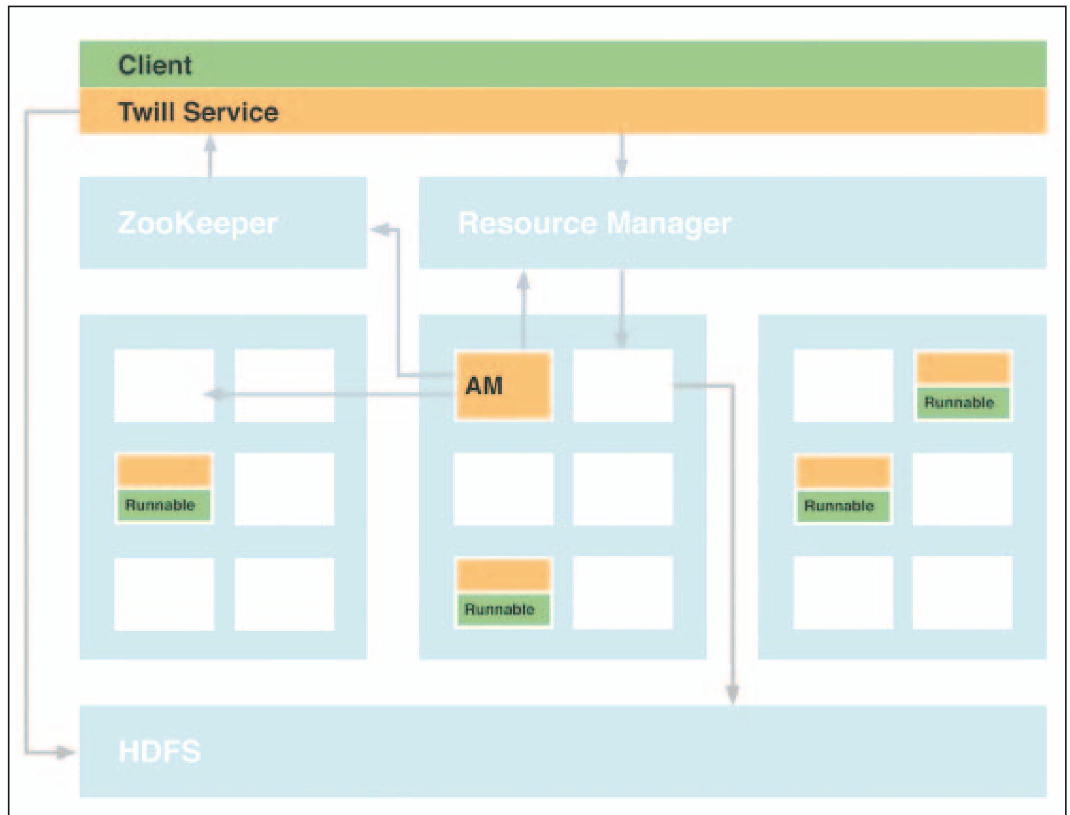


Figure 2: The interactions of a Twill application with YARN

Analogy to Standalone Java Application

It is not difficult to see that a YARN application is very similar to what many Java developers are familiar with: Standalone Java Application. (see Table 1)

This close resemblance suggests that it could be as easy to write a distributed application as it is to write a threaded Java application. This thought inspired the development of Twill with the goal of enabling every Java programmer to write distributed applications.

Hello World

With Twill, all tasks are implemented as a *TwillRunnable*, which is essentially a Java Runnable, and the application is given a specification of how many runnables to start, with rules for ordering tasks and constraints for the resources each task may consume. Twill then uses a generic YARN client and Application Master to allocate the containers, start all runnables, monitor them and if necessary restart them – the only API that the developer needs to understand is the Twill interface, and the above mentioned distributed shell example shrinks to less than 50 lines of code.

For example, let's take a look at the minimal Twill application – a classic Hello-World [13]. We start with implementing the runnable that will be executed:

```
public class HelloWorld {
  public static Logger LOG = LoggerFactory.getLogger(HelloWorld.class);

  public static class HelloWorldRunnable extends AbstractTwillRunnable {
    @Override
    public void run() {
      LOG.info("Hello World. My first distributed application.");
    }
  }
}
```

To submit this application with Twill, all we need is to create and start a runner service and submit the runnable. We also make use of Twill's central log collection and attach a log handler that prints all log messages to standard out (more about that later). The result of starting the app is a controller object that can be used to manage the application from now on. In this example, all we do it wait for completion.

```
public static void main(String[] args) throws Exception {
  TwillRunnerService twillRunner = new YarnTwillRunnerService(
    new YarnConfiguration(), "localhost:2181");
  twillRunner.startAndWait();
```

YARN	Standalone Java App
YARN Client	Java command that provides options and arguments to the application.
Application Master	*Main()* method preparing threads for the application.
Container Task	Runnable implementation, where each runs in its own thread.

Table 1

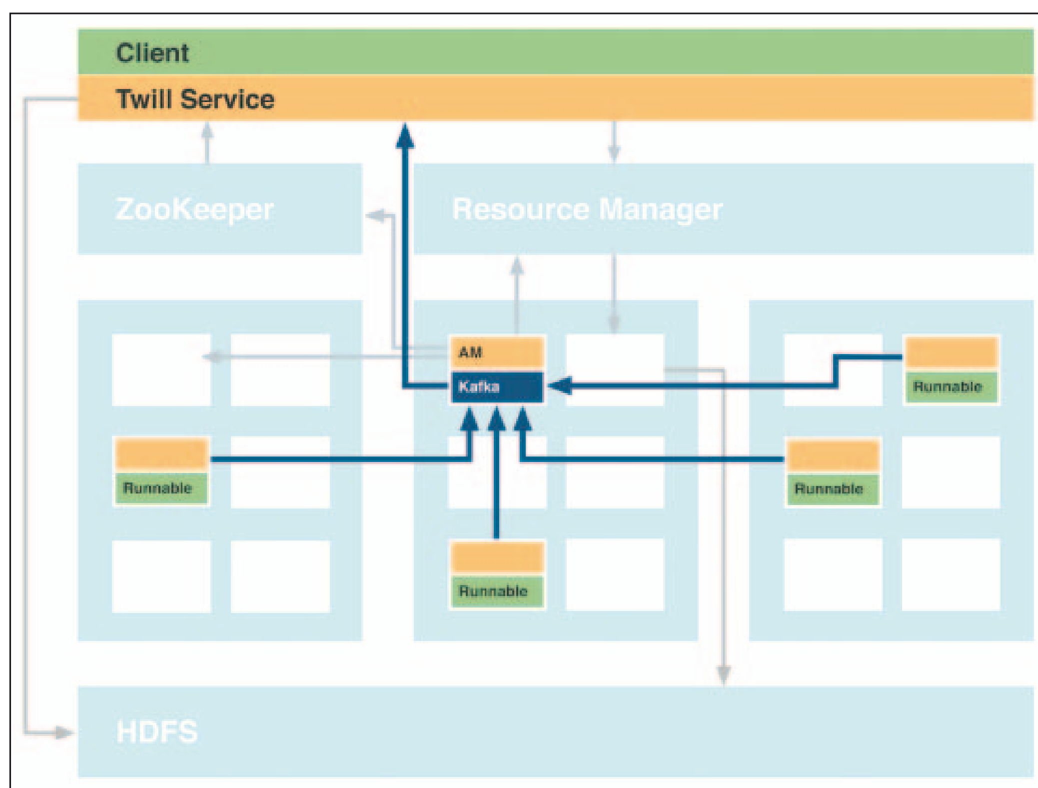


Figure 3: Log collection in Twill using Apache Kafka

```
  TwillController controller = twillRunner.prepare(new HelloWorldRunnable())
  .addLogHandler(
    new PrinterLogHandler(
      new PrintWriter(System.out, true)))
  .start();

  Services.getCompletionFuture(controller).get();
}
```

Distributed Application Patterns

Twill also addresses a number of common patterns of distributed applications:

Persistence of State: Twill saves the state and configuration of every application in ZooKeeper. That means that even if the client terminates or crashes, a new client can always reconnect to a running Twill application. Apache ZooKeeper [14] is a highly reliable distributed coordination and synchronization service.

Real-time Log Collection: The tasks of an application will typically emit log messages. For ease of operations, it is desirable to make all logs available for inspection and search in a central place at that time that they happen. YARN does not provide any assistance with this kind of log collection, and application developers must implement it themselves. Again, YARN leaves it to the developer to implement log collection. Twill addresses that by embedding Apache Kafka in each application. Kafka [15] is a scalable, distributed publish/subscribe messaging system. Twill has an option to start a Kafka instance along with the AM. It then injects a log appender into each runnable that publishes the messages to Kafka. From there, all logs can be retrieved using the Twill client, as shown in **Figure 3**.

Application Life Cycle: Once an application is running, YARN has only two options: either wait for it to finish, or end it by killing all its containers. But often it is desirable to shut the application down gracefully, for example allowing it to persist its state before quitting. Or sometimes it is necessary for an application to pause in order to resume at a later time.

It can also be useful to reconfigure a running application by signaling all of its tasks to re-read their configuration. For example, an application that uses a machine-learned model for classification may need to reread the model every time the model is rebuilt (by an independent process). Twill allows these kinds of actions with a simple, ZooKeeper-based message protocol between the Twill client and the runnables.

Service Discovery: Running a service in YARN means that the service may start on any of the machines of the cluster. Clients of the service need to find out what IP addresses or host names to connect to, hence the service must announce its actual location to the clients after it starts up. This is tricky to implement, because the location must also be withdrawn after an instance of the service terminates or crashes. Using ZooKeeper, Twill implements a simple service discovery protocol that allows clients to find the service using a Twill client.

Programming with Twill: Resources specification

There are situations where you will need more than one instance of your application. For example, when using Twill to run a cluster of Jetty Web Servers. Moreover, different applications would have different requirements on system resources, such as CPU and memory. By default, Twill starts one container per *TwillRunnable* with 1 virtual core and 512 MB of memory. You could, however, customize it when starting your application through *TwillRunner*. For example, you can specify 5 instances, each with 2 virtual cores and 1 GB of memory by doing this:

```
TwillRunner twillRunner = ...
twillRunner.prepare(new JettyServerTwillRunnable(),
                    ResourceSpecification.Builder.with()
                                         .setVirtualCores(2)
                                         .setMemory(1, SizeUnit.GIGA)
                                         .setInstances(5)
                                         .build())
          .start();
```

Notice that this specifies virtual cores and not actual CPU cores. The mapping is defined in the YARN configuration – see [16] for more details.

Multiple runnables

Just like you can have multiple threads doing different things, you can have multiple *TwillRunnable* in your application. All you need to do is implement the *TwillApplication* interface and specify the runnables that constitute your application. Say your application contains a Jetty server and a log processing daemon, your *TwillApplication* will look something like this:

```
public class MyTwillApplication implements TwillApplication {
 @Override
 public TwillSpecification configure() {
  return TwillSpecification.Builder.with()
  .setName("MyApplication")
  .withRunnable()
  .add("jetty", new JettyServerTwillRunnable()).noLocalFiles()
  .add("logdaemon", new LogProcessorTwillRunnable()).noLocalFiles()
  .anyOrder()
  .build();
 }
}
```

Note that the call to *anyOrder()* specifies that every *TwillRunnable* in this application can be started in no particular order. If there are dependencies between runnables, you can specify the ordering like this:

```
// To have the log processing daemon start before the Jetty server
.withRunnable()
.add("jetty", new JettyServerTwillRunnable()).noLocalFiles()
.add("logdaemon", new LogProcessorTwillRunnable()).noLocalFiles()
.withOrder()
.begin("logdaemon")
.nextWhenStarted("jetty")
```

File localization

One nice feature in YARN is that it can copy HDFS files to a container's working directory on local disk, which is an efficient way to distribute files needed by containers across the cluster. Here is an example of how to do so in Twill:

```
.withRunnable()
.add("jetty", new JettyServerTwillRunnable())
.withLocalFiles()
// Distribute local file "index.html" to the Jetty server
.add("index.html", new File("index.html"))
// Distribute and expand contents in local archive "images.tgz"
// to the container's "images" directory
.add("images", new File("images.tgz"), true)
// Distribute HDFS file "site-script.js" to a file named "script.js".
// "fs" is the Hadoop FileSystem object
.add("script.js", fs.resolvePath(new Path("site-script.js")).toUri()) .apply()
```

In Twill, the file that needs to be localized doesn't need to be on HDFS. It can come from a local file, or even an external URL. Twill also supports archive auto-expansion and file rename. If no file needs to be localized, simply call *noLocalFile()* when adding the *TwillRunnable*.

Arguments

Just like a standalone application, you may want to pass arguments to alter the behavior of your application. In Twill, you can pass arguments to the individual *TwillRunnable* as well as to the whole *TwillApplication*. Arguments are passed when launching the application through *TwillRunner*:

```
TwillRunner twillRunner = ... twillRunner.prepare(new MyTwillApplication())
// Application arguments will be visible to all runnables
.withApplicationArguments("--debug")
// Arguments only visible to instance of a given runnable.
.withArguments("jetty", "--threads", "100")
.withArguments("logdaemon", "--retain-logs", "5")
```

The arguments can be accessed using the *TwillContext* object in *TwillRunnable*. Application arguments are retrieved by calling *TwillContext.getApplicationArguments()*, while runnable arguments are available through the *TwillContext.getArguments()* call.

Service discovery

When launching your application in YARN, you don't know where your containers will be running, and the hosts can change over time due to container or machine failure. Twill has built-in service discovery support – you can announce a named service from runnables and later on discover their locations. For example, you can start the Jetty server instances on a random port and announce the address and port of the service.

```
class JettyServerTwillRunnable extends AbstractTwillRunnable() {
 @Override
 public void initialize(TwillContext context) {
  // Starts Jetty on random port
  int port = startJetty();
  context.announce("jetty", port);
 }
}
```

You can then build a router layer to route those requests to the cluster. The router will look something like this:

```
TwillController controller = ... ServiceDiscovered jettyService = controller.
                                                        discoverService("jetty");
// The ServiceDiscovered maintains a live list of service endpoints.
// Everytime the .iterator() is invoked it gives the latest list of endpoints.
// Iterator<Discoverable> itor = jettyService.iterator();
// Pick an endpoint from the list of endpoints.
// ...
```

Controlling live applications

As mentioned in the Hello-World example, you can control a running application using *TwillController*. You can change the number of instances of a runnable by simply doing this:

```
TwillController controller = ... ListenableFuture<Integer> changeComplete =
                                        controller.changeInstances("jetty", 10);
```

You can then either block until the change is completed or observe the completion asynchronously by listening on the future.

Future

Twill is a powerful distributed application platform. Yet it was only incubated in Apache recently, and it is still in its early stages. It is clear that many more features have to follow, such as:

Non-Java applications: At this time Twill is a pure Java framework. But there are many systems engineers who believe that other programming languages ultimately deliver better performance. Allowing the runnables to be written in different languages would open up Twill to those developers, while still performing the (higher-level) tasks of management and control in Java. This will also allow running existing distributed engines that were not written in Java, such as Impala, or not written for YARN, such as HBase or Cassandra (see [18] for an alternative approach).
Suspending applications: Some applications are not time-critical and can be run when the load in the cluster is low (for example, distributed testing). This requires the ability to suspend the application when the cluster load goes up, to free the resources until the load is low enough to resume. While this appears simple, it implies the ability of the application to save its state when suspended and restoring its state upon restart. It also requires restarting the application in containers in locality with the original containers. For example, HBase's region servers store their files in HDFS, and it is crucial for performance that a region server is resumed on the same machine (that ensures a local replica of the HDFS files).
Central metrics collection: Metrics are crucial in understanding the behavior and performance of distributed systems. Twill can use the same Kafka instance that it already uses for log collection to also collect metrics from all the containers and make them available through the controller.

An open source project can only thrive with an active community. Many new features and requirements will come from Twill's users, and that journey has just begun at Apache.

Conclusion

With YARN as its "operating system", a Hadoop cluster turns into a virtual compute server. YARN makes it possible to develop massively parallel applications and run them on the Hadoop cluster. Twill adds simplicity to the power of YARN and transforms every Java programmer into a distributed data application developer. It's a revolution – and it has only just got started!

Andreas Neumann develops big data software at Continuuity, and has formerly done so at places that are known for massive scale. He was the chief architect for Hadoop at Yahoo! and also for the foundational content management system that Yahoo! built on Hadoop. Previously he was a research engineer at Yahoo! and a search architect at IBM. Andreas holds a doctoral degree in computer science for his work on querying XML.

@anew68

Terence Yim is a Software Engineer at Continuuity, designing and building a realtime processing system on Hadoop/HBase. Prior to Continuuity, Terence spent over a year at LinkedIn and seven years at Yahoo!, building high performance large scale distributed systems.

@chtyim

References

[1] http://hadoop.apache.org/
[2] http://research.google.com/archive/mapreduce.html
[3] http://hive.apache.org/
[4] http://tajo.incubator.apache.org/
[5] http://prestodb.io/
[6] http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html
[7] http://wiki.apache.org/incubator/StormProposal
[8] http://samza.incubator.apache.org/
[9] http://en.wikipedia.org/wiki/Message_Passing_Interface
[10] http://www2003.org/cdrom/papers/refereed/p007/p7-abiteboul.html
[11] http://hortonworks.com/hadoop/yarn/
[12] http://twill.incubator.apache.org/
[13] http://en.wikipedia.org/wiki/Hello_world_program
[14] http://zookeeper.apache.org/
[15] http://kafka.apache.org/
[16] http://riccomini.name/posts/hadoop/2013-06-14-yarn-with-cgroups/YARN virtual core support
[17] http://incubator.apache.org/
[18] http://wiki.apache.org/incubator/HoyaProposal

©iShockphoto.com/courtneyk

Getting a measure of things

Processing on the Grid

Why do traditional cache semantics sometimes struggles to scale, and what can we do about it?

by Steve Millidge

If you ever have the luxury of designing a brand new Java application there are many, new, exciting and unfamiliar technologies to choose from. All the flavours of NoSQL stores; Data Grids; PaaS and IaaS; JEE7; REST; WebSockets; an alphabet soup of opportunity combined with many programming frameworks both on the server side and client side adds up to a tyranny of choice.

However, if like me, you have to architect large scale, server-side, Java applications that support many thousands of users then there are a number of requirements that remain constant. The application you design must be high-performance, highly available, scalable and reliable.

It doesn't matter how fancy your new lovingly crafted JavaScript Web2.0 user interface is, if it is slow or simply not available nobody is going to use it. In this article I will try and demystify one of your choices, the Java Data Grid and show how this technology can meet those constant non-functional requirements while at the same time taking advantage of the latest trends in hardware.

Latency: The performance killer

When building large scale Java applications the most likely cause of performance problems in your application is latency. Latency is defined as the time delay between requesting an operation, like retrieving some data to process, and the operation occurring. Typical causes of latency in a distributed Java application are:

- IO latency pulling data from disk
- IO latency pulling data across the network
- Resource contention for example a distributed lock
- Garbage Collection pauses

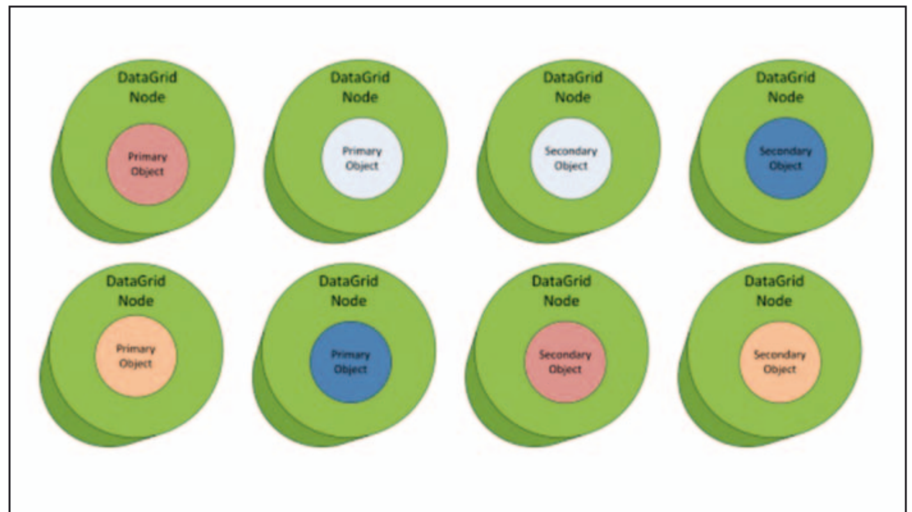


Figure 1: Java Data Grids

For example typical ping times across a network range from; 57 μs on a local machine; 300 μs on a local LAN segment through to 100 ms from London to New York. When these ping times are combined with typical network data transfer rates; 25 MB–30 MB/s for 1 Gb Ethernet; 250 MB/s–350 MB/s for 10 Gb Ethernet a careful trade-off between operation frequency and data granularity must be made to achieve acceptable performance. Ie. if you have 100 MB of data to process the decision between making 100 calls across the network each retrieving 1 MB, or 1 call retrieving the full 100 MB will depend on the network topology. Network latency is normally the cause of the developer cry, "It was fast on my machine!"

Latency due to disk IO is also a problem, a typical SSD when combined with a SATA 3.0 interface can only deliver data at a sustained data rate of 500–600 MB/s so if you have Gigabytes of data to process disk latency will impact your application performance.

The hardware component with the lowest latency is memory, typical main memory bandwidth, ignoring Cache hits, is around 3–5 GB/s and scales with the number of CPUs. If you have 2 processors you will get 10 GB/s and with 4 CPUs 20 GB/s etc. John McCalpin at Virginia maintains a memory benchmark called STREAM (http://www.cs.virginia.edu/stream/) which measures the memory throughput of many computers with some achieving TB/s with large numbers of CPUs. In conclusion:

Memory is FAST: And therefore, for high performance, you should process data in memory.

Network is SLOW: Therefore for high performance minimise network data transfer.

The question then becomes is it feasible to process many Gigabytes of data in memory? With the costs of memory dropping it is now possible to buy single servers with 1 TB of memory for only a few £30K–£40K and the latest SPARC servers are shipping supporting 32 TB of RAM so Big Memory is here.

The other fundamental shift in hardware at the moment is the processing power of single hardware threads is starting to reach a plateau with manufactures moving more into providing CPUs with many cores and many hardware threads. This trend forces us to design our Java applications in a fashion that can utilise the large number of hardware threads appearing in modern chips.

Parallel is the Future: For maximum performance and scalability you must support many hardware threads.

Data Grids

You may wonder what all this has to do with Java Data Grids. Well, Java Data Grids are designed to take advantage of these facts of modern computing and enable you to store many 100 s of GB of Java objects in memory and enable parallel processing of this data for high performance.

A Java Data Grid is essentially a distributed key value store where the key space is split across a cluster of JVMs and each Java object stored within the grid has a primary object on one of the JVMs and a secondary copy of the object on a different JVM. These duplicates ensure High Availability as if a single JVM in the grid fails then no Java objects will be lost.

The key benefits of the partitioned key space in a Data Grid when compared to fully replicated clustered Cache are that the more JVMs you add the more data you can store and access times for individual keys are independent of the number of JVMs in the grid.

For example, if we have 20 JVM nodes in our Grid each with 4 GB of free heap available for the storage of objects then we can store, when taking into account duplicates, 40 GB of Java objects. If we add a further 20 JVM nodes then we can store 80 GB. Access times are constant to read/write objects as the grid will go directly to the JVM which owns the primary key space for the object we require.

JSR 107 defines a standards based API to data grids which is very similar to the *java.util.Map* API as shown in Listing 1.

Many Data Grids also make use of Java NIO to store Java objects "off heap" in Java NIO buffers. This has the advantage that we can increase the memory available for

Listing 1

```
public static void main( String[] args )
{
  CacheManager CacheManager = Caching.getCachingProvider().
                                                    getCacheManager();
  MutableConfiguration<String, String> config = new MutableConfiguration<String,
                                                                  String>();
  CacheManager.configureCache("C2B2",config);
  Cache Cache = CacheManager.getCache("C2B2");
  Cache.put("Key", "Value");
  System.out.println(Cache.get("Key"));
}
```

storage without increasing the latency from garbage collection pause times.

Parallel processing on the Grid

The problem arises when we store many 10 s of GB of Java objects across the Grid in many JVMs and then want to run some processing across the data set. For example, we may store objects representing hotels and their availability on dates. What happens when we want to run a query like "find all the hotels in Paris with availability on Valentines day 2015"? If we follow the simple Map API approach we would need to run code like that shown in Listing 2.

However the problem with this approach, when accessing a Data Grid, is that the objects are distributed according to their keys across a large number of JVMs and every "get" call needs to serialize the object over the network to the requesting JVM. Using the listing above this could pull 10s of GB of data over the network which as we saw earlier is *slow*.

Thankfully most Java Data Grid products allow you to turn the processing on its head and instead of pulling the data over to the code to process they send the code to each of the Grid JVMs hosting the data and execute it in parallel in the local JVMs. As typically the code is very small in size only a few KB of data needs to be sent across the network.

Processing is run in parallel across all the JVMs making use of all the CPU cores in parallel. Example code, which runs the Paris query across the Grid, for Oracle Coherence, a popular Data Grid product is shown in Listing 3 and 4. Listing 3 shows the code for a Coherence EntryProcessor which is the code that will be serialized across all the nodes in the data grid.

This *EntryProcessor* will check each hotel as before to see if there is availability for Valentine's day but unlike in Listing 2 it will do so in each JVM on local in-memory data. JSR107 also has the concept of an *EntryProcessor* so the approach is common to all Data Grid products.

Listing 4 shows the Oracle Coherence code needed to send this processor across the Data Grid to execute in parallel in all the grid JVMs.

Processing data using *EntryProcessors* as shown in Listings 3 and 4 will result in much greater performance on a Data Grid than access via the simple Cache API. As only a small amount of data will be sent across the network and all CPU cores across all the JVMs will be used to process the search.

Listing 2

```
public static void main( String[] args )
{
  CacheManager CacheManager = Caching.getCachingProvider().
                                                    getCacheManager();
  MutableConfiguration<String, String> config = new MutableConfiguration<String,
                                                              String>();
  CacheManager.configureCache("ParisHotels",config);
  Cache hotelCache = CacheManager.getCache("ParisHotels");
  Date valentinesDay = new Date(2015,2,14); // I know it is deprecated
  for (String hotelName : hotelNames ) {
    Hotel hotel = (Hotel)hotelCache.get(hotelName);
    if (hotel.isAvailable(valentinesDay)){
      System.out.println("Hotel is available" + hotel);
    }
  }
}
```

Listing 4

```
public static void main( String[] args )
{
  NamedCache hotelCache = CacheFactory.getCache("ParisHotels");
  Date valentinesDay = new Date(2015,2,14); // I know it is deprecated
  Map results = hotelCache.processAll((Filter)null, new
                                          HotelSearch(valentinesDay);
}
```

Fast Data: Parallel processing on the Grid

As we've seen, using a Data Grid in your next application will enable you to store large volumes of Java objects in memory for high performance access in a highly available fashion. This will also give you large scale parallel processing capabilities that utilise all the CPU cores in the Grid to crunch through processing Java objects in parallel. Take a look at Data Grids next time you have a latency problem or you have the luxury of designing a brand new Java application.

Steve Millidge is the founder and Technical Director at C2B2, Expert Group member for JSR 286 (Portals), JSR 347 (Data Grids) and JSR 107 (Caching). He has used Java extensively since pre1.0 and has been a field based professional service consultant for over 10 years and has extensive experience of deploying large scale production Java Middleware Systems. Steve has spoken at a number of events on performance and scalability of web systems including Java One, JBoss World, UK Oracle User Group Conference and Special Interest Groups, Jax London, The Server Side Symposium, Community One and regularly presents technical workshops on Data Grids and Big Scale Java.

Listing 3

```
Public class HotelSearch implements EntryProcessor {
  HotelSearch(Date availability) {
    this.availability = availability;
  }
  Map processAll(Set hotels) {
    Map mapResults = new ListMap();
      for (Entry entry : hotels) {
        Hotel hotel = (Hotel)entry.getValue();
        if (hotel.isAvailable(this.availability)) {

        }
      }
    }
  }
```