usenix ;login:

OCTOBER 2012 VOL. 37, NO. 5

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

usenix THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

UPCOMING EVENTS

OSDI ’12: 10th USENIX Symposium on Operating Systems Design and Implementation
October 8–10, 2012, Hollywood, CA, USA
www.usenix.org/conference/osdi12

Workshops Co-located with OSDI ’12 Include:

HotDep ’12: Eighth Workshop on Hot Topics in System Dependability
October 7, 2012
www.usenix.org/conference/hotdep12

MAD ’12: 2012 Workshop on Managing Systems Automatically and Dynamically
October 7, 2012
www.usenix.org/conference/mad12

HotPower ’12: 2012 Workshop on Power Aware Computing and Systems
October 7, 2012
www.usenix.org/conference/hotpower12

Middleware 2012: ACM/IFIP/USENIX 13th International Conference on Middleware
December 3–7, 2012, Montreal, Quebec, Canada
www.usenix.org/conference/middleware2012

LISA ’12: 26th Large Installation System Administration Conference
December 9–14, 2012, San Diego, CA, USA
www.usenix.org/conference/lisa12

FAST ’13: 11th USENIX Conference on File and Storage Technologies
February 12–15, 2013, San Jose, CA, USA
www.usenix.org/conference/fast13

NSDI ’13: 10th USENIX Symposium on Networked Systems Design and Implementation
April 3–5, 2013, Lombard, IL, USA
www.usenix.org/conference/nsdi13

HotOS XIV: 14th Workshop on Hot Topics in Operating Systems
May 13–15, 2013, Santa Ana Pueblo, NM, USA
www.usenix.org/conference/hotos13
Submissions due: January 10, 2013

2013 USENIX Federated Conferences Week
June 25–28, 2013, San Jose, CA, USA

HotCloud ’13: 5th USENIX Workshop on Hot Topics in Cloud Computing
June 25–26, 2013

WiAC ’13: 2013 USENIX Women in Advanced Computing Summit
June 25, 2013

USENIX ATC ’13: 2013 USENIX Annual Technical Conference
June 26–28, 2013

HotStorage ’13: 5th USENIX Workshop on Hot Topics in Storage and File Systems
June 27, 2013

USENIX Security ’13: 22nd USENIX Security Symposium
August 14–16, 2013, Washington, DC, USA

EDITOR
Rik Farrow
rik@usenix.org

MANAGING EDITOR
Rikki Endsley
rikki@usenix.org

COPY EDITOR
Steve Gilmartin
proofshop@usenix.org

PRODUCTION
Arnold Gatilao
Casey Henderson
Michele Nelson

TYPESETTER
Star Type
startype@comcast.net

USENIX ASSOCIATION
2560 Ninth Street, Suite 215,
Berkeley, California 94710
Phone: (510) 528-8649
FAX: (510) 548-5738

www.usenix.org

;login: is the official magazine of the USENIX Association. *;login:* (ISSN 1044-6397) is published bi-monthly by the USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

$90 of each member's annual dues is for a subscription to *;login:*. Subscriptions for nonmembers are $90 per year. Periodicals postage paid at Berkeley, CA, and additional offices.

POSTMASTER: Send address changes to *;login:*, USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710.

©2012 USENIX Association
USENIX is a registered trademark of the USENIX Association. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. USENIX acknowledges all trademarks herein. Where those designations appear in this publication and USENIX is aware of a trademark claim, the designations have been printed in caps or initial caps.

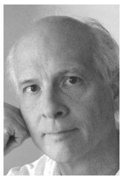
usenix ;login:

OCTOBER 2012, VOL. 37, NO. 5

OPINION

Musings

RIK FARROW



Rik is the editor of *;login:*.
rik@usenix.org

As a certified armchair researcher, I settled into my armchair in a dark room, watching distant flashes of lightning over the desert landscape surrounding me. The scattered storms brought to mind the problem of weather forecasting, or rather, the common failings of accurate forecasts beyond the next several days.

Like other High Performance Computing (HPC) problems, weather forecasting requires enormous computational resources. The world is a big place, after all, and the weather is notoriously fickle. Other HPC problems, such as protein folding and fluid dynamics, suffer from similar issues; duplicating nature with a set of processors is an approximation, at best.

And to make matters worse, the approximations we have today must be performed on von Neumann machines.

Fish

Instead of imagining a weather system, as I sit in my armchair I imagine a school of fish. I love undersea videos of large schools of fish, light flashing off their bodies as they move in coordinated synchrony to ward off predators. These fish are not connected by networks, but use simple biological mechanisms that allow them to function as a unit—something at which our computers today are not so good.

Shifting focus a bit, I imagine dividing software into three big buckets: control, stream processing, and cells. Control software makes the decisions, such as starting other processing, and is characterized by many tests and branches. I visualize it as a tangle of threads.

Stream processing is also familiar. We now have powerful GPUs that use pipelines of specialized processors to transform a stream of data. We see stream processing in simple things, such as playing a movie or an audio recording, as well as in big data projects, where clusters of servers filter through data distributed throughout the cluster to produce the reduced result. For this, I see streams of data, with processors located along the stream like beads on a chain.

But weather forecasting, fluid dynamics, protein modeling, and the modeling of nuclear explosions do not fit into either the control or stream paradigms. In my mind, they best match cellular automata, where the state of adjacent cells affects the transition of a cell to its future state.

In weather modeling, the cells are enormous—on the scale of tens of kilometers. Worse still, the cells are three-dimensional, not at all like the classic 2D cellular automata. And, finally, weather is affected by radiation, which will have effects not just from adjacent cells, but from the edges of the model—the surface of the earth and space itself.

As if weather is not hard enough, imagine dealing with fluid dynamics, such as modeling the turbulent flow of air across the wing of an airfoil. Here the cells are not boxes sitting on geography, but particles flowing and interacting. What each particle does affects its neighbors, and their neighbors, and so on.

Our computers are designed for control and stream processing, and do quite well at this. HPC needs to work with cells or flows or interacting particles, and include non-local effects such as radiation. Ideally, our HPC systems would work like a school of fish, rearranging themselves into the most efficient formation. Instead, we have to work with systems attached to racks, wired into networks, and arranged as a fixed grid of processors and memory. No wonder this is a hard problem to solve.

The Lineup

We open this issue with an article based on a position paper presented during HotPar '12. Rob Knauerhase, Romain Cledat, Justin Teller, and Mark Handley describe future directions for HPC systems, in particular, replacing the operating system with something much lighter weight. When one considers that supercomputers have tens of thousands of cores, running a full-fledged OS like Linux on each core would be a tremendous waste of resources—and systems such as Blue Gene already run much smaller execution engines. Knauerhase et al. explain where Intel plans to go—and is already moving—with their new Xeon Phi [1, 2] line of coprocessors.

Olivier Bonaventure, Costin Raiciu, and Mark Handley report on MPTCP, an extension to TCP that supports multiple paths without changes to client or server applications. MPTCP will be useful in datacenters, but also for mobile devices, because MPTCP is designed to *just work*—just like the IP stack we have grown accustomed to using. One huge issue in the development of MPTCP was dealing with middleboxes, which may modify header information. MPTCP neatly works around these issues, in addition to falling back transparently when a link stops working.

Kamau Wangũhũ describes VXLAN, a method for extending broadcast networks across routed networks. Virtual machines that work together may expect to be located on the same broadcast (Layer 2) network, when in reality, they may be placed wherever it is currently convenient to instantiate them. VXLAN provides transparent tunneling, so that VM instances on other networks appear to be local from the point of view of the VM.

Amandeep Khurana has written an introduction to HBase, a database that runs over Hadoop. Using HBase requires that people accustomed to using SQL databases rethink how they design their schema, and Khurana clearly provides suggestions for using HBase efficiently.

Stuart Kendrick decided that he should publish the results of his months-long study of 10 years of outages. Kendrick has the fortune of having a fairly complete record of system and software failures, and he thought that his current position demanded that he analyze this data properly. The results may not surprise you; for example, software issues lead to the most outages. If you are responsible for maintaining many systems plus SLAs simultaneously, I suggest reading Kendrick's analysis, which does contain information useful for starting your own analysis.

Jacob Farmer managed to squeeze in an interview about a new project he and his company are working on with Harvard Medical School. The school decided to work with Cambridge Technologies to see whether they could do a better job of attaching meaning to the files that they were storing or archiving, and Farmer explains what this project means to anyone interested in making sense of the vast amounts of stored data we deal with these days.

Charles Polisher shared his and his coworkers' experience with a series of mysterious power supply failures in a datacenter. No clear cause ever emerged, but Polisher does describe the fixes that may have ended their problems.

David Blank-Edelman takes us down a different path this time. In his August '12 column, David described a tool that permits manipulating XML and HTML-structured data using paths to access that data. This time, David explores Augeas, a tool designed specifically for manipulating configuration files using paths. Like the XML Path Language, Augeas can make managing configuration files via paths much simpler than the standard pattern or field matching approaches.

David Beazley leads us down simultaneous paths in parallel by explaining how to use the multiprocessing library in Python. David first provides an example of how threads work, then shows how the multiprocessing library actually allows a Python program to use all of the cores, instead of the single core permitted by the thread library in Python. David ends his column with a simple RPC server.

Dave Josephsen exhibits his usual enthusiasm, this time for a new library called 0MQ. 0MQ abstracts away a lot of the issues with writing multicast, publish-subscribe, clients, and servers. Dave begins with a simple example of a server written using 0MQ, which actually is simpler than writing the same server without this library.

Inspired by Charles Polisher's description of failed power supplies, Robert Ferrell takes us on many more paths toward failures. For Robert, power supply failures are not sufficient, and he entertains us with other related sources of outages.

Elizabeth Zwicky continues down the path that she has been following toward more effective management, starting with two books full of ideas for dealing more effectively with team members. Then she takes a look at a book that provides the real story of the Macintosh, one without a focus on Steve Jobs, but on what actually went on in Silicon Valley garages. She then takes a hard look at a Frederick Brooks book and finishes up with a book on IPv6.

Mark Lamourine reviews two books about Coffeescript. I now have a really good idea of exactly what Coffeescript is about, and whether either or both of these books would be suitable support for learning this JavaScript replacement.

Evan Teran rounds out our reviews section with an in-depth review of *Inside Windows Debugging.* Debuggers are critical tools for developers and security geeks like Teran, who says that this book demystifies how Microsoft debuggers work.

Finally, we have summaries from the 2012 USENIX Annual Technical Conference. As always, many of the sessions at USENIX conferences are recorded, and these videos and audio recordings appear on the Web as soon as they are processed. We plan to start posting summaries online after they have been edited, which means you can read summaries soon after an event instead of waiting until they appear in *;login:*. Summaries from the Hot Topics in Parallelism workshop and Federated Conferences Week are available online now.

The storms I've been watching have moved off to the north. From past experience—and using online weather radar—I know I can see lightning flashes further than 40 miles away. I take a deep breath as I settle deeper into my armchair.

I try to imagine a supercomputer as fluid as a school of fish, and I fail. All I can see are the racks of servers and bundles of hardwired network cables that make up the inside of today's HPC supercomputers. Even as our technology advances the von Neumann designs based on Turing's mathematics, the problems technology needs to solve require something new, something more fluid. Something I am failing to imagine.

References

[1] Knight's Corner (Xeon Phi) uses embedded Linux: http://software.intel.com/en-us/blogs/2012/06/05/knights-corner-open-source-software-stack/.

[2] Intel's Xeon Phi many-core coprocessor: http://www.anandtech.com/show/6017/intel-announces-xeon-phi-family-of-coprocessors-mic-goes-retail/.

For Extreme Parallelism, Your OS Is Sooooo Last-Millennium

ROB KNAUERHASE, ROMAIN CLEDAT, AND JUSTIN TELLER

Rob Knauerhase is a Research Scientist with Intel Labs. His current focus is observation-driven dynamic adaptation (resource mapping and scheduling) in system software. Other research interests include distributed systems, machine virtualization, and programming-system technologies for parallel applications. Knauerhase received a Master of Computer Science from the University of Illinois at Urbana-Champaign, and a Bachelor of Science in Engineering from Case Western Reserve University. He has 32 patented inventions, along with various and sundry publications and invited speaking events. Knauerhase is a Senior Member of the IEEE.
rob.knauerhase@intel.com

Romain Cledat is a Research Scientist at Intel Labs, where he has been contributing to Intel's research on exascale computing, particularly in the area of compilers and runtimes. Romain was previously a student at the Georgia Institute of Technology, where he received an MS in Electrical and Computer Engineering in 2005 and a PhD in Computer Science in 2011.
romain.e.cledat@intel.com

Justin Teller earned a BS degree in Electrical Engineering from Ohio University in 2002, an MS degree in Electrical Engineering from the University of Maryland, College Park in 2004, and a PhD in Electrical Engineering from Ohio State University in 2008. After graduating, Teller worked at Intel on high-throughput computing products before moving to Intel Labs to research scaling system software to exascale (and beyond).
justin.teller@gmail.com

Introduction

High-performance computing has been on an inexorable march from gigascale to tera- and petascale, with many researchers now actively contemplating exascale (10^{18}, or a *million trillion* operations per second) systems. This progression is being accelerated by the rapid increase in multi- and many-core processors, which allow even greater opportunities for parallelism. Such densities, though, give rise to a new cohort of challenges, such as containing system software overhead, dealing with large numbers of schedulable entities, and maintaining energy efficiency.

We are studying software and processor-architectural features that will allow us to achieve these goals. We believe that exascale operation will require significant out of the box thinking, specifically in terms of the role of operating systems and system software. In this article, we describe some of our research into how these goals can be achieved.

Motivation

Historic parallelism has come from banding processors together on a task, either in a large distributed system (whether in a grid, cloud, or other HPC/supercomputer configurations), a smaller cluster of server nodes, or a number of sockets on a motherboard. In each case, there are interesting problems for division of labor, interconnect tradeoffs (e.g., latency, bandwidth), and so forth. Concurrently, the microprocessor industry is trending toward many-core processors, uniting an increasing number of CPUs inside one processor; indeed, the recently announced Intel® Xeon Phi™ coprocessor [11] boasts more than 50 cores on a chip.

Meanwhile, processor fabrication technology has enabled cores to run at ever-lower voltages, which has worked out well for some high core count uses (throughput computing, graphics, etc.); however, even including optimistic public estimates of energy-efficiency improvements over time, one can extrapolate that an exascale computer would require at least 500 MW of power, or roughly the output of a small nuclear power plant.

Despite the advances in many-core processor capabilities, we anticipate that exascale operation will require a staggering amount of computational resources. Such extreme systems will also entail an unprecedented amount of complexity in managing the effective coordination and use of resources. In our research, we have been exploring how notions of system software—current and old ideas, along with some new ideas—should change in order to run an exascale machine effectively.

Philosophy

In 2010, the Defense Advanced Research Projects Agency (DARPA) generated a solicitation for innovative joint-research proposals [3] on extreme-scale systems. The DARPA challenges, in addition to performance, included aggressive energy efficiency (both in terms of performance/watt and minimizing energy spent on intra-system communication), dynamic adaptability (to changes in workload, hardware, or external goals), and programmability. Our philosophies, software prototypes, and experiments have been strongly influenced by participation in this program.

Our research hypothesis for exascale system software is that *a fine-grained, event-driven execution model with sophisticated observation and adaptation capabilities* will be key to exascale system software. To this end, we break applications down into dataflow-inspired [4] codelets [18], which are invoked by a runtime environment based on satisfaction of data and control dependencies. Dependencies are specified either by the programmer or by a high-level compilation system.

Extreme-Scale Hardware

Based on current trends and our own hardware research efforts, we anticipate that a number of hardware characteristics will be typical of exascale systems.

As mentioned, in contrast to the previous upward spiral of core speed, industrial trends have been producing designs that include a larger number of modest-speed processors. Although the optimal arrangement of these cores—number per die, die per chip or socket, and so forth—is far from clear, our prototypes assume that hardware will (must) provide an efficient—fast, low-latency, low-power—inter-core communication mechanism.

Further, we expect that power requirements will require cores that are relatively simple, perhaps with shared-ISA heterogeneity to allow different alternatives for hardware acceleration. In the same vein, we anticipate that extreme-scale systems are all but certain to operate cores at near threshold voltage—with implicit challenges for reliability—while also allowing for dynamic voltage/frequency scaling controlled by system software.

We predict that the nature of I/O within the system will change. As fabrication processes and architectures never-endingly drive computation to be less expensive, communication will become relatively more expensive. At exascale, communication (moving a bit) is likely to be at least as expensive as computation (manipulating that bit), which requires a new focus on efficient data placement and code/data positioning optimizations from software.

Lastly, we foresee changes in memory organization. Power concerns will dictate a larger amount of per-core memory, perhaps scratchpad memory, additional register files, or various types of cache. Also, there will almost certainly be similar structures at a block (grouping of cores) level, and/or at a chip level, and so forth. Given the propensity of hardware architects to use fractal designs, replicating interconnected structures, our software and hardware prototypes comprehend a deep memory hierarchy, with classes of latency/power memories: local, group-local, neighbor-group-local, and so forth. Such structures provide both challenges and opportunities for system software.

Unsuitability of OS Functions for Exascale

Traditionally, the role of the operating system is to provide a variety of services in order to expose a common programming interface to programmers and applications, irrespective of the underlying hardware. Thus, an OS *actively* tries to abstract away and hide the nature of the hardware. This approach has proven to be quite successful, enabling the programmer to focus on the essence of his application without worrying about the nitty-gritty of managing hardware devices, memory, or threads and tasks. Also, this approach enabled programmers to write once and run their code unmodified on different generations of processors; however, our research has led us to believe that many of the traditional OS functions are in the best case suboptimal, and in the worst case directly counter to the energy and performance goals we are seeking. To that end, we have become iconoclastic about the notion of a traditional operating system for extreme-scale systems and have focused on either omitting (avoiding the need for) or simplifying much of the functionality one imagines in an OS. Instead, our research is exploring the development of a sophisticated, yet lighter-weight, *runtime environment* to manage an exascale machine.

Extreme-Scale System Software

In this section, we introduce the programming model we envision for exascale systems. In the following sections, we will specifically delve into memory and thread management.

Programming Model

Our programming model is heavily inspired by the dataflow-style codelet ideas we recently described in [18]. Dating as far back as the late 1980s [4], dataflow programming and machines are not new, but we re-examine these ideas in a contemporary environment so that we can benefit from both the insights and the shortcomings of the prior work.

The model decomposes traditional tasks into stateless codelets; after a codelet finishes, both the code and context can be destroyed safely, thereby conserving energy. It includes an explicit description of the dependencies among codelets and between codelets and data to allow for better co-location within an extreme-scale system. Codelets may additionally be automatically derived from higher-level representations, such as Concurrent Collections [12].

A key objective of the programming model is to reduce energy consumption by facilitating the co-location of data and computation, thereby devoting a larger part of the energy available to actual computation (as opposed to merely shuffling data around). Furthermore, explicit data and computation placement will allow high-level tools, such as compilers and performance analysis toolkits, to provide the runtime with greater insight into the performance and energy usage of a program, thereby enabling the programmer to home in on "energy bottlenecks."

LEVERAGING META-INFORMATION

Our research is also investigating means to pass more information from the programmer and compiler into the runtime. In a traditional compilation system, data available to the compiler (or generated by the compiler's analyses) is not made available to the OS, being largely discarded when the binary is created. One example is the tradeoff between code space and loop overhead when a loop is unrolled: under different conditions, different versions of the code will have higher efficiency. Our research indicates that preserving awareness of tradeoffs like this can enable a runtime to more intelligently (co-)locate codelets, increasing performance and saving energy. The compiler can also generate various versions of the code, allowing the runtime to choose, based on environmental conditions and system goals, which version to run.

Other metadata will also prove crucial in a heterogeneous system as the codelets are able to expose their requirements or preferences in terms of hardware. The runtime scheduler then appropriately schedules these codelets, trying to optimize for their preferences, available resources, environmental constraints (such as temperature in various parts of the chip), and data locality.

Runtime Environment

Programs written to the model above are executed by a lightweight (relative to traditional operating system kernels) runtime environment such as our Intel Research Runtime (a component of the nascent Open Community Runtime [17]), or commercial alternatives such as the SWARM runtime [15, 5]. The runtime design deliberately empowers "hero" programmers by exposing more of the hardware features and execution details while still allowing "regular" programmers to write correct code.

The runtime leverages the fine-grained nature of codelets to allow explicit placement of code: for example, allowing a group of codelets which communicate or use the same data structures to be topologically grouped together for energy efficiency. Absent such direction, it can automatically use tuning hints or observation to (re-)locate code throughout the system. In particular, in a heterogeneous environment, it can select among alternate codelet implementations based on availability and proximity of certain cores. Likewise, for data placement, the runtime enables direct access to memory features such as DMA and inter-core networks, and can autonomously move data closer to code using techniques described below.

Because our anticipated hardware allows very fine-grained power and clock control of cores (clock-gating or changing frequency to save energy, and turning off unused hardware), the runtime also manages the overall energy profile of the system. Given different external policies (e.g., "deliver answer as fast as possible regardless of energy used" versus "take time to deliver an answer with minimal power consumption") and overall system state (e.g., voltage-related or permanent core failures), the runtime will turn on or off various parts of the system and schedule codelets in accordance with the specified policy. For example, if the policy specifies that the end-goal is performance, the runtime would schedule everything as fast as it could on the most powerful cores it could find. On the other hand, if energy was a concern, the runtime may turn off certain cores and therefore utilize fewer resources.

SEPARATION OF CONTROL

An important function of the OS is to manage access from user code to hardware resources. Traditionally, a kernel runs in a privileged execution mode while user code runs with fewer privileges and has specific channels (system calls) to interact with the hardware. Implicit in most OS implementations is the need to switch between user and kernel modes. This switch incurs overhead which in some cases can be very expensive (hundreds of cycles) [16]. For complex exascale machines, such overhead (in terms of time, as well as power) is likely to be unsupportable.

Our system software similarly separates *control code* and *execution code* but does so in a way that enables us to get rid of many expensive OS functionalities. The runtime designates a small number of cores to be "control engines" (CEs), which execute the runtime itself in a distributed fashion across the system, and the majority to be "execution engines" (XEs) that run user code.

Our system thus disregards the notion of ring boundaries entirely, choosing instead to separate privileges by space rather than by time. Application (user) code is only run on XEs, and our control software runs on CEs. This division obviates the need for traditional processor "modes" and their associated overhead. Several types of security concerns are likewise alleviated; combined with hardware features (e.g., configurable range-checked access control and "locking" to render portions of memories immutable) and associated compiler support, our system obtains much of the security benefit of user/kernel division. Our simulations include hardware support to implement fast and power-efficient communication between the XEs and their controlling CE to replace the functionality traditionally provided by system calls.

This division of duties also allows specialization of each core type for power and performance as desired. For example, CEs do not need accelerator hardware for advanced math, but they may benefit from special low-latency interconnects or particular atomic instructions specialized for queue processing. Likewise, depending on the assumed needs of applications, XEs can contain variably sized floating-point hardware, optimized implementations of arithmetic and transcendental functions, or other hardware (e.g., encryption logic) as needed.

ABSENCE OF DEVICE DRIVERS

Since only system code runs on the CEs, there is no need for device drivers in the usual sense of the term. If a CE does not directly connect to a peripheral, it can forward its request to a CE that does. The system runtime for CEs that do directly support devices includes device functions (e.g., to manage device state, or to accommodate special instructions or interfaces) as appropriate.

A drawback of this approach is the difficulty in supporting a wide array of peripherals. However, in the time frame of our research, exascale machines are unlikely to be off-the-shelf commodity designs, and their owners are likely to have programmers or contracts to support hardware upgrades. User-code requests that entail peripheral devices are treated as data dependencies by the scheduler and are satisfied through runtime code on the CEs.

Memory Management

The OS has traditionally abstracted hardware in its memory management, in particular through the use of virtual memory, which provides, in part, a contiguous address space independent of the underlying hardware.

Modern OSes rely on virtual memory to provide each individual process with its own address space, thereby hiding the details of the backing physical storage medium (whether it is RAM or disk) from the programmer and allowing him to ignore the issue of code overlay. Virtual memory, however, also comes with two major costs: (1) the hardware and energy costs of doing the translation between virtual and physical addresses and (2) the loss of visibility into the varying characteristics of the physical memory (distance, energy costs, etc.).

The former can be dealt with [6] and is not a focus here; however, the latter is exacerbated in exascale systems where physical constraints make it necessary to have deep memory hierarchies to be able to simultaneously provide fast memory for data actively used by computations as well as very large ones for input and output data. The loss of visibility into the memory hierarchy can have drastic energy consequences. For example, without considering the energy required for the memory controller, it takes about 75 picoJoules per bit to move data from DRAM while it only takes about 0.5 pJ per bit to move data within the chip. This double-order of magnitude difference means that applications and their data must be very carefully positioned so that energy may be spent on computation rather than communication.

Can Virtual Memory Be Improved to Perform with Exascale?

Virtual memory provides undeniable advantages and, before dismissing it, we should consider whether it can be improved to provide better visibility and energy efficiency in exascale systems.

Fundamentally, the important issue is reducing the *distance* between data and computation. At first glance, virtual memory could be modified to provide this benefit as it adds a level of indirection (between the virtual address and the physical one). One can imagine a system where this mapping is influenced by the closeness of the physical page to the computation. This mapping could, and would, dynamically change at runtime as usage of the page changes. For example, suppose there are two cores, C1 and C2, each with its own RAM (M1 and M2). While threads on C1 are accessing a particular virtual page, it could be mapped to M1 where it is as close as possible to C1. However, if threads on C1 stop accessing it and threads on C2 start accessing it, the virtual page could be remapped to a physical page on M2 to minimize access costs.

APPLICATION FOCUS VERSUS SOFTWARE FOCUS

The problem with the above solution is that virtual memory is managed in an application-agnostic manner at a granularity that only makes sense from a hardware perspective. In particular, a single page may contain objects that have very different access patterns and would ideally be placed in two different physical locations. A programmer could force these objects to live in different pages, but this could result in huge memory fragmentation.

Solution: Make Data Objects First Class Citizens

The solution we are pursuing is to do away with all the memory "support" mechanisms an OS provides and allow the programmer to directly manipulate *data-blocks* as first-class entities. Note that HPC programmers are already bypassing OS memory support by writing custom allocators that are tuned to a particular architecture and paging mechanism. We posit that taking this further is essential for exascale computing.

Conceptually, a data-block is simply a contiguous chunk of memory with metadata annotations to allow it to be moved throughout the memory hierarchy. At a high level, data-blocks are similar to pages but bear a crucial difference: they are *semantically meaningful*. In other words, a data-block only contains data that is related in a way that makes sense to a particular computation. This is not true for pages (especially "huge" pages such as those supported in Linux), which can contain data pertinent to different and unrelated computations.

Note that data-blocks also introduce a level of indirection (since their base address in memory can change), but this can be efficiently dealt with in hardware and with compiler support. In spite of this additional indirection, the advantages provided by using data-blocks in exascale systems are very appealing:

- Each data object can be precisely placed in memory.
- As access patterns change, the objects can be moved either by the programmer, or automatically by the runtime. This also enables multiple cores to pass the same data object to each other (as in a pipeline) and have it optimally placed at each stage of the pipeline.
- Runtime observation systems, with the assistance of hardware, can track accesses to data objects, thereby enabling optimizations based on access patterns.

Note that the increased control given to the programmer does not necessarily mean that she has to manually manage everything and deal with all the complexities of fine-grained memory management. We envision a system where a runtime, aided by programmer hints or sophisticated observation via hardware performance-monitoring units, would optimally move data-blocks so as to reduce the energy required to access data.

Scheduling Management

The threading metaphor is perhaps the only parallel abstraction a modern OS exposes to the programmer. However, the scheduling granularity of threads is ill-suited for new programming models (namely, our own, as well as others such as Cilk, TBB, Habanero, X10, Chapel and Fortress).

Thread Parallelism for Exascale

A traditional OS is responsible for mapping threads to hardware resources and context switching them as needed to ensure that they all get equal access to computing resources. While this approach eliminates the need to know the number of underlying parallel resources—in line with the OS's objective to abstract away the hardware—emerging parallel programming models rely on being able to *precisely* schedule much smaller-grained tasks and frequently implement a runtime layer on top of traditional threads to manage dependencies and the mapping of tasks to OS-provided threads.

These fine-grained schedulers and the OS may have misaligned goals (the OS being more concerned about fairness, for example, which may be less of a concern to certain runtimes that may favor critical path execution), and this leads to kludges in the runtime to ensure that the OS does not perform "optimizations" that harm performance. These hacks include (1) using processor affinity to prevent thread migration, and (2) carefully matching the number of OS threads to the number of hardware-supported threads to avoid context switching among others. The latter hack demonstrates that the runtime is actively attempting to break through the OS abstraction and trying to get to the underlying hardware characteristic (here, the number of hardware threads). Essentially, the runtime and the programmer work very hard to avoid many of the services the OS provides since the goals of the OS (fairness in execution, responsiveness, etc.) are not shared by exascale programming.

CONTEXT SWITCHING

Specifically, avoiding context switching is paramount in exascale systems since the cost of context switching in terms of energy (as well as time) is nontrivial. Furthermore, to reduce the energy needed to access data, extreme-scale systems will increase the amount of hardware-provided thread-local storage such as registers and private scratchpad memories. In this situation, avoiding context switching becomes all the more critical.

Our program decomposition into small codelets [18] allows us to run a single codelet per XE without overhead from preserving ephemeral state (swapping registers, managing scratchpads, etc.). Since XEs are plentiful, the scheduling system can narrow or widen its scheduling of parallel code according to system state, workload, and overall power/performance tradeoff policies. Additionally, as the runtime understands that a finishing task's context is no longer needed, it can put that core into a very low-energy (destructive to memory) sleep state.

Required Threading Notions

The scheduling of codelets really only requires two notions: affinity and dependencies.

AFFINITY

A modern OS will provide hooks for threads with the affinity interface expressing links between threads and hardware resources. Conversely, our runtime provides an interface to define affinity between and among tasks. This approach allows the runtime to better understand the relationships among all tasks, such as which tasks are related and therefore share data. With this information, the runtime can schedule tasks onto the hardware resources in such a way to increase locality, or to optimize for other system operation goals. Furthermore, since the entire programming system and runtime are aware of the presence of possibly specialized XE hardware, the programmer is able to create specialized tasks to run on that specialized hardware. Both the compiler and programmer are also able to generate multiple equivalent versions of a task to run more efficiently on a heterogeneous computing substrate.

DEPENDENCIES

Dependency information is also entirely expressed within the runtime's tasking interface. This concept is not novel among runtime interfaces (the TBB task graph [10] is one example). However, our runtime makes this dependency information available to the lowest levels of the system software. One particularly exciting area we are exploring is how this interfaces with the memory management portion of the runtime. As the memory subsystem can see how tasks are related and vice versa, the runtime can make better scheduling decisions. For instance, tasks can be scheduled to minimize data movement by either relocating data or moving a task to execute closer to its data. As all the dependencies are known to the runtime, related tasks can be moved at the same time, leading to an overall more efficient system. This data-directed scheduling research is an area from which we anticipate significant results from our work and that of the broader community-related work. Many of the ideas expressed above are admittedly not entirely our invention, with some having a rich heritage of prior research. Previously, however, these ideas were developed separately, independent both of an extreme-scale (hardware, software, and energy) focus and of tradeoffs among interesting combinations. As process technology and new architectures continue to drive many-core systems to unprecedented levels of parallelism, we are reaching a stage where bringing all these ideas together in a cohesive system and testing them on real hardware will not only help validate them, but also provide concrete solutions to the power-efficiency problems facing the industry.

Other participants in DARPA's UHPC program [2] have identified similar concerns for performance and energy efficiency along the lines of what we describe here. Current HPC technologies, such as MPI [7, 9] and OpenMP, reflect a large software investment and will also need to be improved and scaled to exascale machines; what we propose in this article does not preclude these other developments. Efforts such as Kitten [14] by Sandia National Labs, IBM's Blue Gene CNK [8], and the Barrelfish research OS [1] also aim to support extreme-scale systems with varying focus on performance, power, and scalability.

Conclusion and Future Work

We believe that extreme-scale systems are likely to break many of the known and beloved design conventions of both hardware and software. Existing challenges for performance, power efficiency, adaptability, and programmability will be greatly exacerbated when going to exascale operation. Our research, therefore, is pursuing some fairly radical concepts with respect to traditional definitions of OS functionality. We are taking advantage of hardware and software co-design to experiment with separating control and application duties across heterogeneous cores, as well as resurrecting prior dataflow-inspired techniques in our codelet execution model.

While much work remains to be done, results to date are encouraging. Our simulations show good reduction in the overhead (especially energy) that a traditional OS would incur, especially in terms of context-switching and memory management. Leveraging prior work, we plan to introduce more dynamic observation features in our runtime and hope to further optimize both power and performance by automatically migrating data closer to code and/or code closer to data.

Exascale computing has become a primary interest throughout industry, academia, and government. In partnership with both academia and government, we are continuing to explore ideas such as those discussed in this article. With continued progress, we hope to usher in a new HPC era, in which scientists and engineers will have access to a new generation of "extreme-scale" supercomputing systems with unprecedented computational power *along with* supportable energy consumption.

Acknowledgments

We are indebted to Shekhar Borkar for championing our research in the exascale community. Thanks also to DARPA and the Department of Energy for their interest in (and funding of) exascale HPC research. We are also grateful to the HotPar'12 workshop community for their feedback on the paper on which this article is based [13], and to USENIX ;login: for giving us the opportunity to share our research in this forum.

This research was, in part, funded by the US Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the US Government.

References

[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture For Scalable Multicore Systems," *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (SOSP), 'ACM, 2009, pp. 29–44.

[2] B. Dally, "Power, Programmability, and Granularity: The Challenges of Exascale Computing" (IPDPS keynote): http://techtalks.tv/talks/54110/, 2011.

[3] DARPA, UHPC BAA: http://tinyurl.com/38zvgm4, March 2010.

[4] J.B. Dennis and G.R. Gao, "An Efficient Pipelined Dataflow Processor Architecture," *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, 'IEEE Computer Society Press, 1988, pp. 368–373.

[5] Eti Web site: http://www.etinternational.com/.

[6] D. Fan, Z. Tang, H. Huang, and G. Gao, "An Energy Efficient TLB Design Methodology," *Proceedings of the 2005 International Symposium on Low Power Electronics and Design (ISLPED '05), 2005,* pp. 351–356.

[7] A. Geist, "MPI Must Evolve or Die," *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer-Verlag, 2008, p. 5.

[8] M. Giampapa, T. Gooding, T. Inglett, and R.W. Wisniewski, "Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK," *Proceedings of the 2010 ACM/IEEE International Conference for High Performance*

Computing, Networking, Storage and Analysis (SC '10) IEEE Computer Society, 2010, pp. 1–10.

[9] W. Gropp, "MPI at Exascale: Challenges for Data Structures and Algorithms," *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Springer-Verlag, 2009, p. 3.

[10] Intel threading building blocks: http://www.threadingbuildingblocks.org.

[11] Intel Xeon Phi coprocessor: http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html.

[12] Intel Concurrent Collections for C++ 0.7 for Windows and Linux: http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/, 2011.

[13] R. Knauerhase, R. Cledat, and J. Teller, "For Extreme Parallelism, Your OS Is Sooooo Last-Millennium," *HotPar 2012: 4th USENIX Workshop on Hot Topics in Parallelism,* USENIX, 2012.

[14] Sandia National Laboratories, "Kitten Lightweight Kernel": https://software.sandia.gov/trac/kitten, 2012.

[15] C. Lauderdale and R. Khan, "Towards a Codelet-Based Runtime for Exascale Computing: Position Paper," *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era* (EXADAPT '12) ACM, 2012, pp. 21–26.

[16] J. Liedtke, "On Microkernel Construction," *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15),* ACM, 1995.

[17] Open Community Runtime Google code repository: http://code.google.com/p/opencommunityruntime/, 2011.

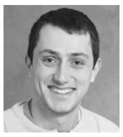
[18] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G.R. Gao, "Using a 'Codelet' Program Execution Model for Exascale Machines: Position Paper," *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era* (EXADAPT '11), ACM, 2011, pp. 64–69.

An Overview of Multipath TCP

OLIVIER BONAVENTURE, MARK HANDLEY, AND COSTIN RAICIU

Olivier Bonaventure is a Professor at Catholic University of Louvain, Belgium. His research focus is primarily on Internet protocols. Bonaventure is the author of the open source textbook *Computer Networking: Principles, Protocols and Practice.*
Olivier.Bonaventure@uclouvain.be

Mark Handley is Professor of Networked Systems at University College London. Handley is the author of 32 RFC Internet Standards documents, including those for the Session Initiation Protocol (SIP) and PIM Sparse Mode multicast routing. He was one of the co-inventors of distributed hash tables and was the recipient of the 2012 IEEE Internet Award.
m.handley@cs.ucl.ac.uk



Costin Raiciu is currently an Assistant Professor at the Computer Science Department of University Politehnica of Bucharest. In 2011, Raiciu received a PhD from University College London, working under the supervision of Mark Handley and David Rosenblum. Raiciu's research interests are mainly in networking; in recent years, he has been working on designing and implementing Multipath TCP.
costin.raiciu@cs.pub.ro

TCP has remained mostly unchanged for 20 years, even as its uses and the networks on which it runs have evolved. Multipath TCP is an evolution of TCP that allows it to run over multiple paths transparently to applications. In this article, we explain how Multipath TCP works, and why you should want to start using it.

Introduction and Motivation

The Transmission Control Protocol (TCP) is used by the vast majority of applications to transport their data reliably across the Internet. TCP was designed in the 1970s, and neither mobile devices nor computers with many network interfaces were an immediate design priority. On the other hand, the TCP designers knew that network links could fail, and they chose to decouple the network-layer protocols (Internet Protocol) from those of the transport layer (TCP) so that the network could reroute packets around failures without affecting TCP connections. This ability to reroute packets is largely due to the use of dynamic routing protocols, and their job is made much easier because they don't need to know anything about transport-layer connections.

Today's networks are multipath: mobile devices have multiple wireless interfaces, datacenters have many redundant paths between servers, and multihoming has become the norm for big server farms. Meanwhile, TCP is essentially a single-path protocol: when a TCP connection is established, the connection is bound to the IP addresses of the two communicating hosts. If one of these addresses changes, for whatever reason, the connection will fail. In fact, a TCP connection cannot even be load balanced across more than one path within the network, because this results in packet reordering, and TCP misinterprets this reordering as congestion and slows down.

This mismatch between today's multipath networks and TCP's single-path design creates tangible problems. For instance, if a smartphone's WiFi loses signal, the TCP connections associated with it stall; there is no way to migrate them to other working interfaces, such as 3G. This makes mobility a frustrating experience for users. Modern datacenters are another example: many paths are available between two endpoints, and multipath routing randomly picks one for a particular TCP connection. This can cause collisions where multiple flows get placed on the same link, thus hurting throughput to such an extent that average throughput is halved in some scenarios.

Multipath TCP (MPTCP) [1, 7, 8] is a major modification to TCP that allows multiple paths to be used simultaneously by a single transport connection. Multipath TCP circumvents the issues mentioned above and several others that affect TCP. Changing TCP to use multiple paths is not a new idea; it was originally proposed more than 15 years ago by Christian Huitema in the Internet Engineering Task Force (IETF), and there have been a half-dozen more proposals since then to similar effect. Multipath TCP draws on the experience gathered in previous work, and goes further to solve issues of fairness when competing with regular TCP and deployment issues as a result of middleboxes in today's Internet. The Multipath TCP protocol has recently been standardized by the IETF, and an implementation in the Linux kernel is available today [2].

Overview of MPTCP Operation

The design of Multipath TCP has been influenced by many requirements, but there are two that stand out: application compatibility and network compatibility. Application compatibility implies that applications that today run over TCP should work without any change over Multipath TCP. Next, Multipath TCP must operate over any Internet path where TCP operates.

Many paths on today's Internet include middleboxes, such as Network Address Translators, firewalls, and various kinds of transparent proxies. Unlike IP routers, all these devices do know about the TCP connections they forward and affect them in special ways. Designing TCP extensions that can safely traverse all these middleboxes has proven to be challenging [4].

Before diving into the details of Multipath TCP, let's recap the basic operation of normal TCP. A connection can be divided into three phases:

- connection establishment
- data transfer
- connection release

A TCP connection starts with a three-way handshake. To open a TCP connection, the client sends a SYN (for "synchronize") packet to the port on which the server is listening. The SYN packet contains the source port and initial sequence number chosen by the client, and it may contain TCP options that are used to negotiate the use of TCP extensions. The server replies with a SYN+ACK packet, acknowledging the SYN and providing the server's initial sequence number and the options that it supports. The client acknowledges the SYN+ACK, and the connection is now fully established. All subsequent packets in the connection use the IP addresses and ports used for the initial handshake. They compose the tuple that uniquely identifies the connection.

After the handshake, the client and the server can send data packets (*segments,* in TCP terminology). The sequence number is used to delineate the data in the different segments, reorder them, and detect losses. The TCP header also contains a cumulative acknowledgment, essentially a number that acknowledges received data by telling the sender which is the next byte expected by the receiver. Various techniques are used by TCP to retransmit the lost segments.

After the data transfer is over, the TCP connection must be closed. A TCP connection can be closed abruptly if one of the hosts sends a Reset (RST) packet, but the usual way to terminate a connection is by using FIN packets. These FIN packets indicate the sequence number of the last byte sent. The connection is terminated after the FIN segments have been acknowledged in both directions.

Multipath TCP allows multiple *subflows* to be set up for a single MPTCP session. An MPTCP session starts with an initial subflow, which is similar to a regular TCP connection as described above. After the first MPTCP subflow is set up, additional subflows can be established. Each additional subflow also looks similar to a regular TCP connection, complete with SYN handshake and FIN tear-down, but rather than being a separate connection, the subflow is bound into an existing MPTCP session. Data for the connection can then be sent over any of the active subflows that has the capacity to take it.

To examine Multipath TCP in more detail, let's consider a simple scenario with a smartphone client and a single-homed server. The smartphone has two network interfaces: a WiFi interface and a 3G interface. Each has its own IP address. The server, being single-homed, has a single IP address. In this environment, Multipath TCP would allow an application on the smartphone to use a single TCP connection that can use both the WiFi and the 3G interfaces to communicate with the server. The application does not need to concern itself with which radio interface is working best at any instant; MPTCP handles that for the application. In fact, Multipath TCP can work when both endpoints are multihomed (in this case, subflows are opened between all pairs of "compatible" IP addresses), or even in the case when both endpoints are single homed (in this case, different subflows will use different port numbers and can be routed differently by multipath routing in the network).

Let's walk through the establishment of an MPTCP connection. Assume that the smartphone chooses its 3G interface to open the connection. First the smartphone sends a SYN segment to the server. This segment contains the MP_CAPABLE TCP option, indicating that the smartphone supports Multipath TCP. This option also contains a key that is chosen by the smartphone. The server replies with a SYN+ACK segment containing the MP_CAPABLE option and the key chosen by the server. The smartphone completes the handshake by sending an ACK segment.

At this point, the Multipath TCP connection is established and the client and server can exchange TCP segments via the 3G path. How could the smartphone also send data through this Multipath TCP connection over its WiFi interface?

Naively, the smartphone could simply send some of the packets over the WiFi interface; however, most ISPs will drop these packets, as they would have the source address of the 3G interface. Perhaps the client could tell the server the IP address of the WiFi interface and use that address when it sends over WiFi. Unfortunately, this will rarely work: firewalls and similar stateful middleboxes on the WiFi path expect to see a SYN packet before they see data packets. The only solution that will work reliably is to perform a full SYN handshake on the WiFi path before sending any packets that way, so this is what Multipath TCP does. This SYN handshake carries the MP_JOIN TCP option, providing enough information to the server that it can securely identify the correct connection with which to associate this additional subflow. The server replies with MP_JOIN in the SYN+ACK, and the new subflow is established.

An important point about Multipath TCP, especially in the context of smartphones, is that the set of subflows that are associated with a Multipath TCP connection is not fixed. Subflows can be dynamically added and removed from a Multipath TCP connection throughout its lifetime, without affecting the bytestream transported on behalf of the application. Multipath TCP also implements mechanisms that allow adding and removing new addresses even when an endpoint operates behind a NAT, but we will not detail them here. If the smartphone moves to another WiFi network, it will receive a new IP address. At that time, it will open a new subflow using its newly allocated address and tell the server that its old address is not usable anymore. The server will now send data towards the new address. These options allow smartphones to easily move through different wireless connections without breaking their Multipath TCP connections [6].

Assume now that two subflows have been established over WiFi and 3G; the smartphone can send and receive data segments over both. Just like TCP, Multipath TCP provides a bytestream service to the application. In fact, standard applications can function over MPTCP without being aware of it—MPTCP provides the same socket interface as TCP.

Because the two paths will often have different delay characteristics, the data segments sent over the two subflows will not be received in order. Regular TCP uses the sequence number in each TCP packet header to put data back into the original order. A simple solution for Multipath TCP would be to reuse this sequence number as is. Unfortunately, this simple solution would create problems with some existing middleboxes, such as firewalls. On each path, a middlebox would only see half of the packets, so it would observe many gaps in the TCP sequence space. Measurements indicate that some middleboxes react in strange ways when faced with gaps in TCP sequence numbers. Some discard the out-of-sequence segments, whereas others try to update the TCP acknowledgments to "recover" some of the gaps. With such middleboxes on a path, Multipath TCP cannot safely send TCP segments with gaps in the TCP sequence number space. On the other hand, Multipath TCP also cannot send every data segment over all subflows; that would be a waste of resources.

To deal with this problem, Multipath TCP uses its own sequence numbering space. Each segment sent by Multipath TCP contains two sequence numbers: the subflow sequence number inside the regular TCP header, and an additional data sequence number (DSN) carried inside a TCP option. This solution ensures that the segments sent on any given subflow have consecutive sequence numbers and do not upset middleboxes. Multipath TCP can then send some data sequence numbers on one path and the remainder on the other path; old middleboxes will ignore the DSN option, and it will be used by the Multipath TCP receiver to reorder the bytestream before it is given to the receiving application.

Congestion Control

One of the most important components in TCP is its congestion controller, which enables it to adapt its throughput dynamically in response to changing network conditions. To perform this functionality, each TCP sender maintains a congestion window, which governs the amount of packets that the sender can send without waiting for an acknowledgment. The congestion window is updated dynamically, growing linearly when there is no congestion and halved when packet loss occurs. TCP congestion control ensures fairness: when multiple connections utilize the same congested link, each of them will independently converge to the same average value of the congestion window.

What is the equivalent of TCP congestion control for multipath transport? To answer this question, we define three goals that multipath congestion control must obey. First, we want to ensure fairness to TCP. If several subflows of the same

MPTCP connection share a bottleneck link with other TCP connections, MPTCP should not get more throughput than TCP. Second, the performance of all the Multipath TCP subflows together should be at least that of regular TCP on any of the paths used by a Multipath TCP connection; this ensures that there is an incentive to deploy Multipath TCP. The third, and most important, goal is that Multipath TCP should prefer efficient paths, which means it should send more of its traffic on paths experiencing less congestion.

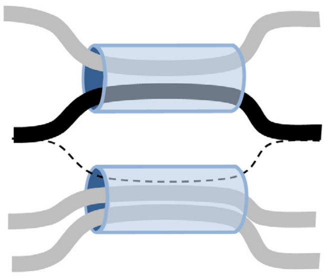


Figure 1: (Resource pooling) Two links, each with capacity 20 pkts/s. The top link is used by a single TCP connection, and the bottom link is used by two TCP connections. A Multipath TCP connection uses both links. Multipath TCP pushes most of its traffic onto the less-congested top link, making the two links behave like a resource pool of capacity 40 pkts/s. Capacity is divided equally, with each flow having throughput of 10 pkts/s.

Intuitively, this last goal ensures wide-area load balancing of traffic: when a multipath connection is using two paths loaded unevenly (see Figure 1), the multipath transport will prefer the unloaded path and push most of its traffic there; this will decrease the load on the congested link and increase it on the less-congested one. If a large enough fraction of flows are multipath, the effect is that congestion will spread out evenly across collections of links, creating “resource pools,” links that act together as if they are a single, larger-capacity link shared by all flows.

Multipath TCP congestion control achieves these goals through a series of simple changes to the standard TCP congestion control mechanism. Each subflow has its own congestion window that is halved when packets are lost, as in standard TCP. Resource pooling is implemented in the increase phase of congestion control; here Multipath TCP will allow less-congested subflows to increase proportionally more than congested ones. Finally, the total increase of Multipath TCP across all of its subflows is dynamically chosen in such a way that it achieves goals one and two above. More details can be found in [3, 4].

Implementation and Performance

Next, we briefly cover two of the most compelling use cases for Multipath TCP by showing a few evaluation results. We focus on mobile devices and datacenters, but note that Multipath TCP can also help in other scenarios. For example, multihomed Web servers can perform fine-grained load balancing across their uplinks, whereas dual-stack hosts can use both IPv4 and IPv6 within a single Multipath TCP connection.

The full Multipath TCP protocol has been implemented in the Linux kernel; its congestion controller has also been implemented in the ns2 and htsim network simulators. The results presented in this article are from the Linux kernel implementation, which is available for download at [2].

Our mobile measurements focus on a typical mode of operation in which the device is connected to WiFi, the connection goes down, and the phone switches to 3G. Our setup uses a Linux laptop connected to a WiFi and a 3G network, downloading a file using HTTP. We compare Multipath TCP with application-layer handover, where the application detects the loss of the interface, creates a new connection, and uses the HTTP range header to resume the download. Figure 2 shows the instantaneous throughputs for Multipath TCP and TCP with application-layer handover. The figure shows a smooth handover with Multipath TCP, as data keeps flowing despite the interface change. With application-layer handover, there is a downtime of three seconds where the transfer stops because the application takes time to detect the interface down event and ramp up 3G. In summary, Multipath TCP enables unmodified mobile applications to survive interface changes with little disruption. A more detailed discussion of the utilization of Multipath TCP in WiFi/3G environments may be found in [6].

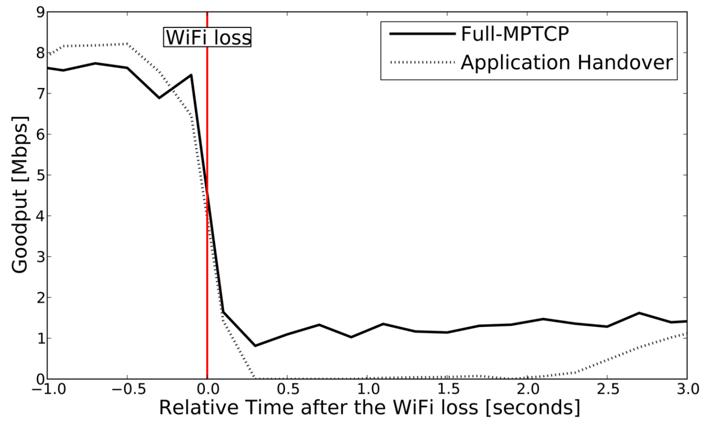


Figure 2: (Mobility) A mobile device is using both its WiFi and 3G interfaces, and then the WiFi interface fails. We plot the instantaneous throughputs of Multipath TCP and application-layer handover.

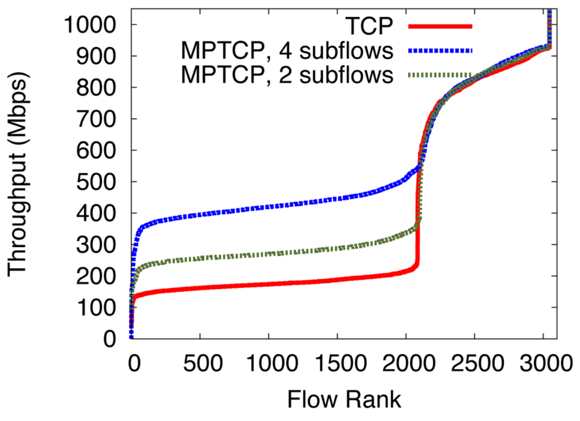


Figure 3: (Datacenter load balancing) This graph compares standard TCP with two- and four-flow MPTCP, when tested on an EC2 testbed with 40 instances. Each host uses iperf sequentially to all other hosts. We plot the performance of all flows (x axis) in increasing order of their throughputs (y axis).

Next, we show results from running Multipath TCP in the Amazon EC2 datacenter. Like most datacenters today, EC2 uses a redundant network topology, where many paths are available between any pair of endpoints, and where connections are placed randomly onto available paths. In EC2, we rented 40 machines (or instances) and ran the Multipath TCP kernel. We conducted a simple experiment in which every machine tested the throughput sequentially to every other machine using first TCP, then Multipath TCP with two and with four subflows. Figure 3 shows the sorted throughputs measured over 12 hours and demonstrates that Multipath TCP brings significant improvements compared to TCP in this scenario. Because the EC2 network is essentially a black-box to us, we cannot pinpoint the root cause for these improvements; however, we have performed a detailed analysis of the cases where Multipath TCP can help and why [5].

Conclusion

Multipath TCP is the most significant change to TCP in the past 20 years; it allows existing TCP applications to achieve better performance and robustness over today's networks, and it has been standardized at the IETF. The Linux kernel implementation shows that these benefits can be obtained in practice; however, as with any change to TCP, the deployment bar for Multipath TCP is high. Only time will tell whether the benefits Multipath TCP brings will outweigh the added complexity it produces in the endhost stacks.

References

[1] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses": http://tools.ietf.org/html/draft-ietf-mptcp-multiaddressed-09.

[2] C. Paasch, S. Barre, J. Korkeaniemi, F. Duchene, G. Detal, et al., "MPTCP Linux Kernel Implementation": http://mptcp.info.ucl.ac.be.

[3] C. Raiciu, M. Handley, and D. Wischik, "Coupled Congestion Control for Multipath Transport Protocols," RFC 6356, October 2011.

[4] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley, "Design, Implementation and Evaluation of Congestion Control for Multipath TCP," USENIX NSDI, March 2011.

[5] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving Datacenter Performance and Robustness with Multipath TCP," ACM SIGCOMM 2011, Toronto, Canada, August 2011.

[6] C. Paasch, G. Detal, F. Duchene, C. Raiciu, and O. Bonaventure, "Exploring Mobile/WiFi Handover with Multipath TCP," ACM SIGCOMM Workshop on Cellular Networks (Cellnet'12), 2012.

[7] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, "How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP," USENIX Symposium of Networked Systems Design and Implementation (NSDI'12), San Jose (CA), 2012.

[8] Multipath TCP resources: http://www.multipath-tcp.org.

VXLAN
Extending Networking to Fit the Cloud

KAMAU WANGŨHŨ

Kamau Wangũhgũ is a Consulting Architect at VMware and a member of the Global Technical Service, Center of Excellence group. Kamau's focus at VMware is in Cloud Architecture and virtual networking futures. Kamau is also a VMware Certified Design Expert (VCDX #003) and was involved in a number of VCDX defenses at the beginning of the program. Kamau has also presented and been involved in all VMworlds, some Technical Solution eXchanges, and Partner Exchanges around the world. Previous to working for VMware, Kamau performed infrastructure architecture work for high transaction systems. Follow Kamau on his blog at http://www.borgcube.com/blogs, or on twitter at http://twitter.com/Borgcube.
Kamau@BORGcube.com

When assembling a cloud infrastructure, you want the flexibility of locating your physical resources anywhere in your datacenter. Networks in a datacenter, on the other hand, may constrain virtual machine (VM) location due to the need for shared Layer 2 connectivity among correspondent virtual machines. VXLAN is an overlay network that overcomes this constraint, allowing you to locate your VMs anywhere in the datacenter without the network being the limiting factor. In this article, I will explain what VXLAN is and how the technology is implemented.

Scalable Networks

One of the biggest infrastructure obstacles faced in building out cloud environments is providing sufficient numbers of isolated networks for tenants. In addition, there is a need to decouple the scaling of these networks from the constraints imposed by the physical network topology and datacenter network architecture. These problems are in no way constrained to cloud-based networking, but are exacerbated by the need for large numbers of on-demand and scalable networks consumed in cloud environments. A key requirement for cloud networking is the need to place VMs anywhere in the datacenter for scaling purposes. This needs to be done while maintaining Layer 2 adjacency between these VMs in order for them to use the same IP address space. However, this requirement implies that the physical networking infrastructure between these communicating VMs should be a Layer 2 (broadcast) network.

Enter VXLAN

VXLAN, in a nutshell, is an overlay Layer 2 over Layer 3 technology that provides physical infrastructure-independent networking to VMs. The motivation behind VXLAN overlay networks was to address the scale and isolation needs encountered in cloud-based infrastructures without imposing any requirements on the physical network infrastructure. VXLAN addresses the flexibility and location-independence requirements of the virtual infrastructure without requiring any networking hardware changes.

In this article, I will concentrate on virtual infrastructure and cloud implementations, as that is where most of the benefits will come into play today. Figure 1 is a key to the icons used in this article.

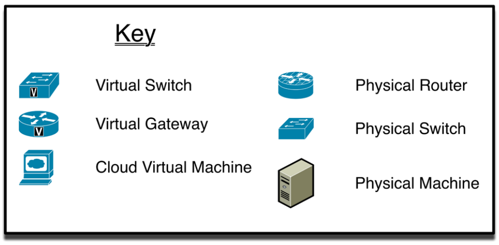


Figure 1: Key to icons used in this article

What Is VXLAN

As an overlay Layer 3 network, VXLAN provides physical infrastructure-independent networking to VMs. Each overlay network is known as a VXLAN Segment and is identified with a 24-bit VXLAN Network Identifier (VNI). This makes it possible to overlay over 16 million networks on a single VXLAN fabric. This is illustrated in Figure 2 as a networking fabric overlaid on virtual switches across multiple server clusters.

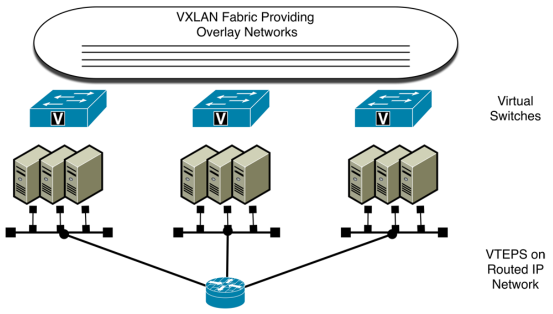

VXLAN Fabric Providing Overlay Networks
Virtual Switches
VTEPS on Routed IP Network

Figure 2: VXLAN fabric overlaid on multiple virtual switches associated with clusters of servers on different Layer 3 networks

A VXLAN fabric is a homogeneous namespace maintained by a single entity on which overlay networks are instantiated. Connectivity between different VXLAN fabrics will not be discussed in this article.

The VXLAN fabric uses the networking provided by the virtual switches to interconnect the VMs and the VXLAN tunnel endpoints (VTEP). VXLAN traffic is tunneled by VTEPs through the physical Layer 3 network infrastructure to other VTEPs. The VTEP is the component of the VXLAN networking stack that encapsulates and decapsulates the VXLAN tunnel traffic that is overlaid on the physical network as illustrated in Figure 3. The VTEP is usually implemented as part of the hypervisor in order to reduce latency. Each VTEP has knowledge of all the virtual machines that it handles and gleans information about virtual machines handled by other VTEPs through data plane-based learning. Here, an unknown destination MAC frame is transmitted over the multicast group associated with that VXLAN segment. When it arrives at the individual VTEPs, they learn the association between the VTEP IP address and the VNI + VM MAC address.

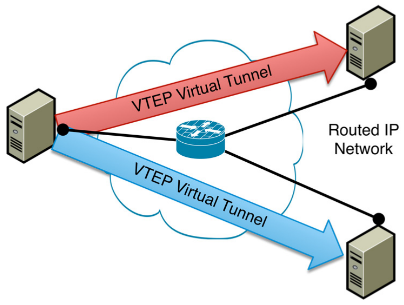


Figure 3: Hosts participating in a VXLAN exchange virtual machine traffic over logical VTEP tunnels that are established through a routed IP network.

The VTEP creates point-to-point tunnels between itself and other VTEPs for the transport of VXLAN encapsulated traffic. The encapsulated traffic can be transported over dedicated transport VLANs or it can share existing VLANs if traffic volumes allow. A VTEP does not provide access to the physical network for the virtual machines it fronts. Access to networks outside the VXLAN segments is provided through the use of a gateway. The simplest use case for a gateway is as a Layer 2 bridge between VXLAN and VLAN environments as shown in Figure 4. It is also possible for routers, or Layer 3 switches, to be VXLAN aware so that they can forward traffic at Layer 3.

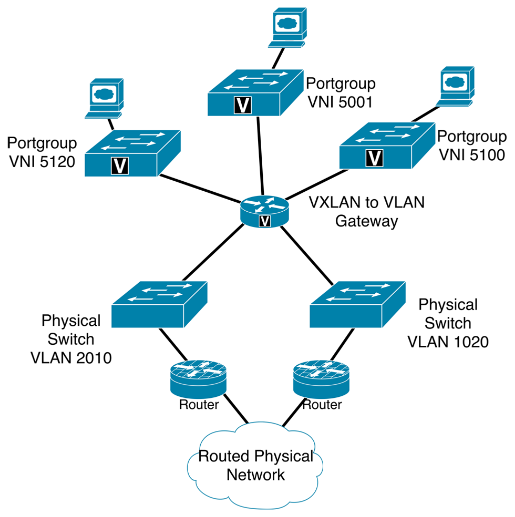


Figure 4: A VXLAN to VLAN gateway is needed to allow virtual machines on a VXLAN-backed network to communicate with nodes outside its Layer 2 network.

When a virtual machine is communicating with other virtual machines on its Layer 2 network, it is not aware in any way of VXLAN. The original Layer 2 frame is encapsulated in an outer UDP packet along with a VXLAN header as illustrated in Figure 5. The VXLAN header contains the VNI (added by the VTEP) that identifies which isolation network the packet belongs to. This UDP frame is then transported on an IP-based network like any normal IP traffic.

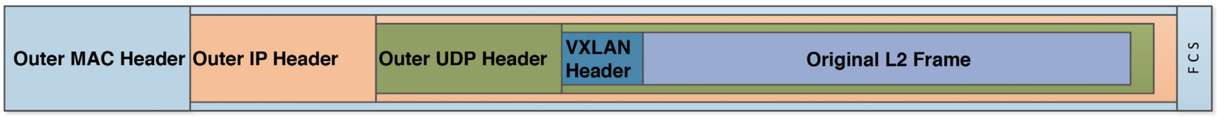


Figure 5: Virtual machines Layer 2 packet is encapsulated in a UDP packet after a VXLAN header is added to the original packet. This allows the packet to be tunneled in an IP Network.

If the destination virtual machine happens to be on the same virtual switch as the source (on the same physical host), then no VXLAN encapsulation happens, and the original Layer 2 frame is passed to the destination.

The outer IP packet has the source address of the source VTEP (VTEPS) and the destination IP of the destination VTEP (VTEPD) fronting the destination MAC address contained in the original Layer 2 frame. VTEPD is discovered when a node sends out an ARP looking for the MAC address corresponding to an IP address. If VTEPS does not know the IP address of the VTEPD, the following process is followed to discover this information:

1. VNI of the VXLAN segment is matched to its associated multicast address.
2. ARP request is encapsulated in a multicast packet whose address corresponds to the multicast address associated with this VNI.
3. Multicast packet is received by all VTEPs that have subscribed to this multicast address, because they are configured with virtual machines in the VNI.
4. All VTEPs glean from this multicast packet the IP address of VTEPS and the source virtual machine's MAC address, which are added to local lookup tables.
5. All VTEPs forward the ARP request to their port group that is associated with the VNI in the multicast packet.
6. The destination virtual machine responds to the ARP request as normal.
7. VTEPD encapsulates the response and sends out a unicast packet back to VTEPS with the ARP response.
8. VTEPS decapsulates and forwards the packet on to the associated port group.
9. In the process of decapsulating the packet, VTEPS gleans the destination node's MAC address and VTEPD IP address from the packet and adds them to its local lookup tables for future unicast communication.

All unknown destination packets, broadcasts, and multicasts are treated in a similar manner, by encapsulating them in multicast packets. This makes it possible for VXLAN-backed networks to support any type of protocol that rides on top of Ethernet, even non IP-based protocols.

VM to VM traffic is encapsulated and sent as unicast traffic between the VTEPs. Only unknown, broadcast and multicast traffic is encapsulated and sent out as multicast traffic.

IGMP snooping [1] needs to be enabled on the physical switches. This enables the physical switches to treat multicast traffic, not like broadcasts, but with a little more intelligence. Physical switches with IGMP snooping enabled will filter out ports that have not subscribed to multicast traffic, thus reducing the amount of data the attached nodes need to process. For IGMP snooping to work, a router, or a Layer 3 switch, needs to be configured as an IGMP queryer. The router will send regular queries about multicast subscription on the network, spurring responses from all the VTEPs about the multicast addresses they are subscribed to. The physical switch snoops these responses and uses the information to maintain its multicast subscription tables (Figure 6).

For VXLAN to work in an environment where VTEPs are not all in the same Layer 2 network (i.e., there are routers used for forwarding between the VTEPs which are on different Layer 3 networks), multicast routing needs to be enabled across the IP network. This is usually done through a protocol like PIM (Protocol-Independent Multicast).

What Does VXLAN Buy Me?

VXLAN creates a network abstraction layer over available physical networks. With a VXLAN fabric in a datacenter, it is possible to overlay multiple VXLAN-backed Layer 2 networks all over the datacenter, providing Layer 2 adjacency to VMs hosted in the datacenter. This makes it possible to create on-demand networks on top of this fabric, allowing unconstrained virtual machine placement within the datacenter and, at the same time, affording unencumbered virtual machine mobility.

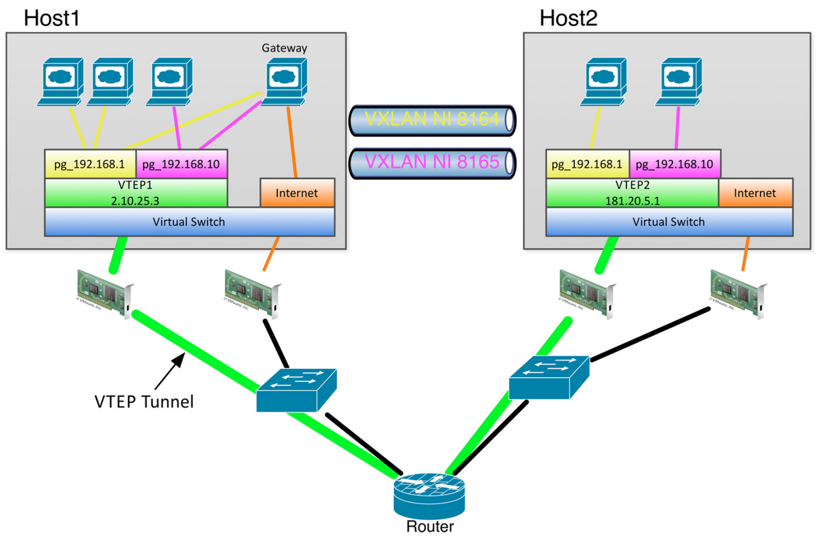


Figure 6: Physical VXLAN topology showing virtual machine connectivity and tunneled VXLAN traffic together with regular traffic from a VXLAN to VLAN gateway

There is no requirement in the VXLAN specification (IETF draft [2]) as defined, for the nodes on a VXLAN backed Layer 2 network to be virtual. There is a requirement though on the physical hosts, or Layer 2 physical switches, for a VXLAN stack in order for physical nodes to attach to a VXLAN fabric.

Acknowledgments

Many thanks go to T. Sridhar for his edits and mentorship.

References

[1] IGMPv3 and IGMP Ssnooping switches: http://tools.ietf.org/html/draft-ietf-idmr-snoop-00.

[2] A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks: http://tools.ietf.org/html/draft-mahalingam-dutt-dcops-vxlan-01.

Introduction to HBase Schema Design

AMANDEEP KHURANA

Amandeep Khurana is a Solutions Architect at Cloudera and works on building solutions using the Hadoop stack. He is also a co-author of *HBase in Action.* Prior to Cloudera, Amandeep worked at Amazon Web Services, where he was part of the Elastic MapReduce team and built the initial versions of their hosted HBase product. amansk@gmail.com

The number of applications that are being developed to work with large amounts of data has been growing rapidly in the recent past. To support this new breed of applications, as well as scaling up old applications, several new data management systems have been developed. Some call this the *big data* revolution. A lot of these new systems that are being developed are open source and community driven, deployed at several large companies. Apache HBase [2] is one such system. It is an open source distributed database, modeled around Google Bigtable [5] and is becoming an increasingly popular database choice for applications that need fast random access to large amounts of data. It is built atop Apache Hadoop [1] and is tightly integrated with it.

HBase is very different from traditional relational databases like MySQL, PostgreSQL, Oracle, etc. in how it's architected and the features that it provides to the applications using it. HBase trades off some of these features for scalability and a flexible schema. This also translates into HBase having a very different data model. Designing HBase tables is a different ballgame as compared to relational database systems. I will introduce you to the basics of HBase table design by explaining the data model and build on that by going into the various concepts at play in designing HBase tables through an example.

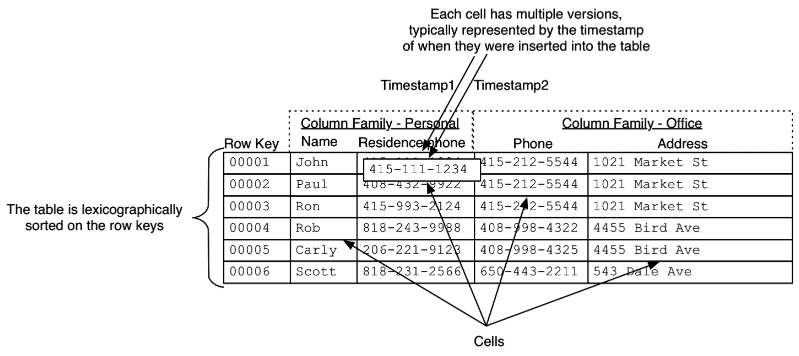
Crash Course on HBase Data Model

HBase's data model is very different from what you have likely worked with or know of in relational databases. As described in the original Bigtable paper, it's a sparse, distributed, persistent multidimensional sorted map, which is indexed by a row key, column key, and a timestamp. You'll hear people refer to it as a key-value store, a column-family-oriented database, and sometimes a database storing versioned maps of maps. All these descriptions are correct. This section touches upon these various concepts.

The easiest and most naive way to describe HBase's data model is in the form of tables, consisting of rows and columns. This is likely what you are familiar with in relational databases. But that's where the similarity between RDBMS data models and HBase ends. In fact, even the concepts of rows and columns is slightly different. To begin, I'll define some concepts that I'll later use.

- **Table:** HBase organizes data into tables. Table names are Strings and composed of characters that are safe for use in a file system path.
- **Row:** Within a table, data is stored according to its row. Rows are identified uniquely by their *row key*. Row keys do not have a data type and are always treated as a *byte[ ]* (byte array).
- **Column Family:** Data within a row is grouped by column family. Column families also impact the physical arrangement of data stored in HBase. For this reason, they must be defined up front and are not easily modified. Every row in a table has the same column families, although a row need not store data in all its families. Column families are Strings and composed of characters that are safe for use in a file system path.
- **Column Qualifier:** Data within a column family is addressed via its column qualifier, or simply, column. Column qualifiers need not be specified in advance. Column qualifiers need not be consistent between rows. Like row keys, column qualifiers do not have a data type and are always treated as a *byte[ ]*.
- **Cell:** A combination of row key, column family, and column qualifier uniquely identifies a cell. The data stored in a cell is referred to as that cell's value. Values also do not have a data type and are always treated as a *byte[ ]*.
- **Timestamp:** Values within a cell are versioned. Versions are identified by their version number, which by default is the timestamp of when the cell was written. If a timestamp is not specified during a write, the current timestamp is used. If the timestamp is not specified for a read, the latest one is returned. The number of cell value versions retained by HBase is configured for each column family. The default number of cell versions is three.

A table in HBase would look like Figure 1.

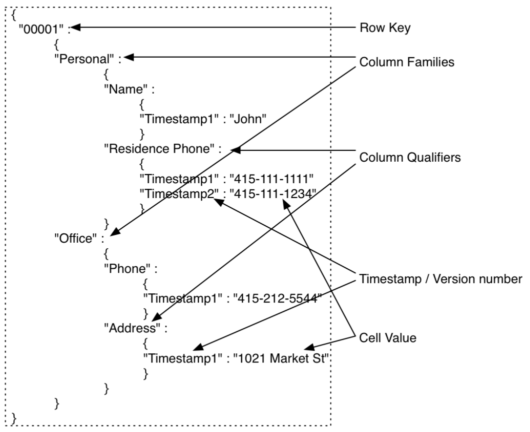

Each cell has multiple versions, typically represented by the timestamp of when they were inserted into the table
Timestamp1
Timestamp2
The table is lexicographically sorted on the row keys
Cells

	Column Family - Personal		Column Family - Office	
Row Key	Name	Residence phone	Phone	Address
00001	John	415-111-1234	415-212-5544	1021 Market St
00002	Paul	408-432-9922	415-212-5544	1021 Market St
00003	Ron	415-993-2124	415-212-5544	1021 Market St
00004	Rob	818-243-9988	408-998-4322	4455 Bird Ave
00005	Carly	206-221-9123	408-998-4325	4455 Bird Ave
00006	Scott	818-231-2566	650-443-2211	543 Dale Ave

Figure 1: A table in HBase consisting of two column families, Personal and Office, each having two columns. The entity that contains the data is called a cell. The rows are sorted based on the row keys.

These concepts are also exposed via the API [3] to clients. HBase's API for data manipulation consists of three primary methods: *Get, Put,* and *Scan*. Gets and Puts are specific to particular rows and need the row key to be provided. Scans are done over a range of rows. The range could be defined by a start and stop row key or could be the entire table if no start and stop row keys are defined.

Sometimes, it's easier to understand the data model as a multidimensional map. The first row from the table in Figure 1 has been represented as a multidimensional map in Figure 2.



```
{
  "00001" :
      {
      "Personal" :
          {
          "Name" :
              {
              "Timestamp1" : "John"
              }
          "Residence Phone" :
              {
              "Timestamp1" : "415-111-1111"
              "Timestamp2" : "415-111-1234"
              }
          }
      "Office" :
          {
          "Phone" :
              {
              "Timestamp1" : "415-212-5544"
              }
          "Address" :
              {
              "Timestamp1" : "1021 Market St"
              }
          }
      }
}
```

Figure 2: One row in an HBase table represented as a multidimensional map

The row key maps to a list of column families, which map to a list of column qualifiers, which map to a list of timestamps, each of which map to a value, i.e., the cell itself. If you were to retrieve the item that the row key maps to, you'd get data from all the columns back. If you were to retrieve the item that a particular column family maps to, you'd get back all the column qualifiers and the associated maps. If you were to retrieve the item that a particular column qualifier maps to, you'd get all the timestamps and the associated values. HBase optimizes for typical patterns and returns only the latest version by default. You can request multiple versions as a part of your query. Row keys are the equivalent of primary keys in relational database tables. You cannot choose to change which column in an HBase table will be the row key after the table has been set up. In other words, the column *Name* in the *Personal* column family cannot be chosen to become the row key after the data has been put into the table.

As mentioned earlier, there are various ways of describing this data model. You can view the same thing as if it's a key-value store (as shown in Figure 3), where the key is the row key and the value is the rest of the data in a column. Given that the row key is the only way to address a row, that seems befitting. You can also consider HBase to be a key-value store where the key is defined as row key, column family, column qualifier, timestamp, and the value is the actual data stored in the cell. When we go into the details of the underlying storage later, you'll see that if you want to read a particular cell from a given row, you end up reading a chunk of data that contains that cell and possibly other cells as well. This representation is also how the *KeyValue* objects in the HBase API and internals are represented. *Key* is formed by [row key, column family, column qualifier, timestamp] and *Value* is the contents of the cell.

```
00001  ──►  { Personal : { Name : { Timestamp1 : John }, Residence Phone : { Timestamp1 : 415-111-1234 } },
                        { Office : { Phone : { Timestamp1 : 415-212-5544 }, Address : { Timestamp1 : 1021 Market St } } }

00001 , Personal  ──►  { Name : { Timestamp1 : John }, Residence Phone : { Timestamp1 : 415-111-1234 } }

00001, Personal:Residence Phone  ──►  { {Timestamp1 : 415-111-1111 } , { Timestamp2 : 415-111-1234) )

00001, Personal:Residence Phone , Timestamp1  ──►  { 415-111-1111 )

00001, Personal:Residence Phone , Timestamp2  ──►  { 415-111-1234 )
```

Figure 3: HBase table as a key-value store. The key can be considered to be just the row key or a combination of the row key, column family, qualifier, timestamp, depending on the cells that you are interested in addressing. If all the cells in a row were of interest, the key would be just the row key. If only specific cells are of interest, the appropriate column families and qualifiers will need to be a part of the key

HBase Table Design Fundamentals

As I highlighted in the previous section, the HBase data model is quite different from relational database systems. Designing HBase tables, therefore, involves taking a different approach from what works in relational systems. Designing HBase tables can be defined as answering the following questions in the context of a use case:

1. What should the row key structure be and what should it contain?
2. How many column families should the table have?
3. What data goes into what column family?
4. How many columns are in each column family?
5. What should the column names be? Although column names don't need to be defined on table creation, you need to know them when you write or read data.
6. What information should go into the cells?
7. How many versions should be stored for each cell?

The most important thing to define in HBase tables is the row-key structure. In order to define that effectively, it is important to define the access patterns (read as well as write) up front. To define the schema, several properties about HBase's tables have to be taken into account. A quick re-cap:

1. Indexing is only done based on the *Key*.
2. Tables are stored sorted based on the row key. Each region in the table is responsible for a part of the row key space and is identified by the start and end row key. The region contains a sorted list of rows from the start key to the end key.
3. Everything in HBase tables is stored as a *byte[ ]*. There are no types.
4. Atomicity is guaranteed only at a row level. There is no atomicity guarantee across rows, which means that there are no multi-row transactions.
5. Column families have to be defined up front at table creation time.
6. Column qualifiers are dynamic and can be defined at write time. They are stored as *byte[ ]* so you can even put data in them.

A good way to learn these concepts is through an example problem. Let's try to model the Twitter relationships (users following other users) in HBase tables. Follower-followed relationships are essentially graphs, and there are specialized graph databases that work more efficiently with such data sets. However, this particular use case makes for a good example to model in HBase tables and allows us to highlight some interesting concepts.

The first step in starting to model tables is to define the access pattern of the application. In the context of follower-followed relationships for an application like Twitter, the access pattern can be defined as follows:

Read access pattern:

1. Who does a user follow?
2. Does a particular user A follow user B?
3. Who follows a particular user A?

Write access pattern:

1. User follows a new user.
2. User unfollows someone they were following.

Let's consider a few table design options and look at their pros and cons. Start with the table design shown in Figure 4. This table stores a list of users being followed by a particular user in a single row, where the row key is the user ID of the follower user and each column contains the user ID of the user being followed. A table of that design with data would look like Figure 5.

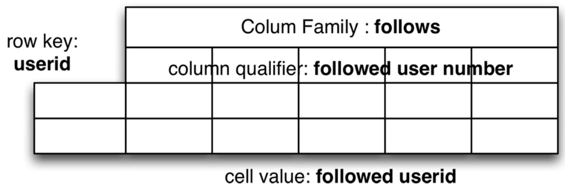

row key:
userid
Colum Family : follows
column qualifier: followed user number
cell value: followed userid

Figure 4: HBase table to persist the list of users a particular user is following

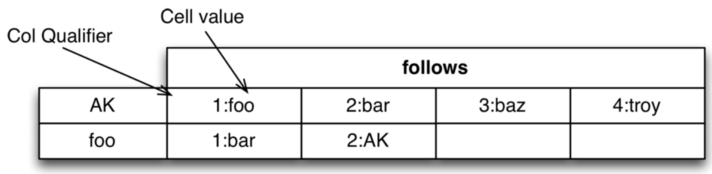


Figure 5: A table with sample data for the design shown in Figure 4

This table design works well for the first read pattern that was outlined. It also solves the second one, but it's likely to be expensive if the list of users being followed is large and will require iterating through the entire list to answer that question. Adding users is slightly tricky in this design. There is no counter being kept so there's no way for you to find out which number the next user should be given unless you read the entire row back before adding a user. That's expensive! A possible solution is to just keep a counter then and the table will now look like Figure 6.

	follows				
AK	1:foo	2:bar	3:baz	4:troy	count:4
foo	1:bar	2:AK	count:2		

Figure 6: A table with sample data for the design shown in Figure 4 but with a counter to keep count of the number of users being followed by a given user

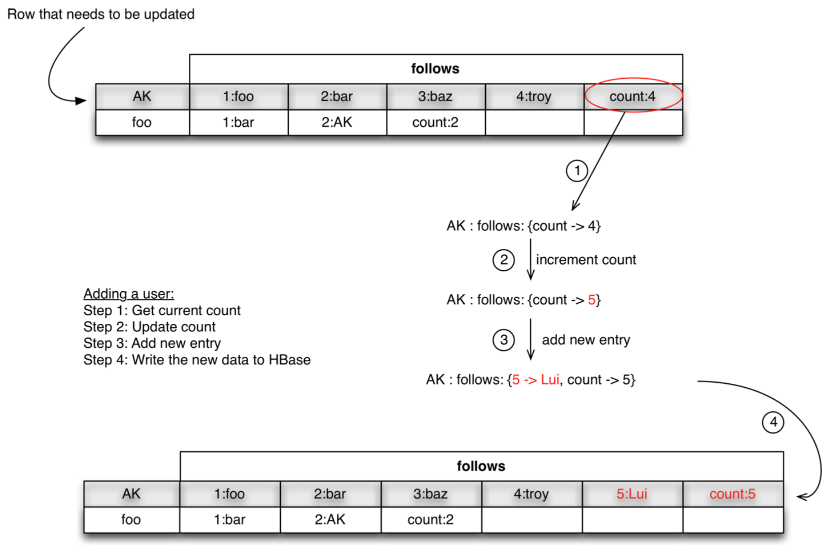


Figure 7: Steps required to add a new user to the list of followed users based on the table design from Figure 6

The design in Figure 6 is incrementally better than the earlier ones but doesn't solve all problems. Unfollowing users is still tricky since you have to read the entire row to find out which column you need to delete. It also isn't ideal for the counts since unfollowing will lead to holes. The biggest issue is that to add users, you have to implement some sort of transaction logic in the client code since HBase doesn't do transactions for you across rows or across RPC calls. The steps to add users in this scheme are shown in Figure 7.

One of the properties that I mentioned earlier was that the column qualifiers are dynamic and are stored as *byte[ ]* just like the cells. That gives you the ability to put arbitrary data in them, which might come to your rescue in this design. Consider the table in Figure 8. In this design, the count is not required, so the addition of users becomes less complicated. The unfollowing is also simplified. The cells in this case contain just some arbitrary small value and are of no consequence.

	follows			
AK	foo:1	bar:1	baz:1	troy:1
foo	bar:1	AK:1		

Figure 8: The relationship table with the cells now having the followed user's username as the column qualifier and an arbitrary string as the cell value.

This latest design solves almost all the access patterns that we defined. The one that's left is #3 on the read pattern list: who follows a particular user A? In the current design, since indexing is only done on the row key, you need to do a full table scan to answer this question. This tells you that the followed user should figure in the index somehow. There are two ways to solve this problem. First is to just maintain another table which contains the reverse list (user and a list of who all *follows* user). The second is to persist that information in the same table with different row keys (remember it's all byte arrays, and HBase doesn't care what you put in there). In both cases, you'll need to materialize that information separately so you can access it quickly, without doing large scans.

There are also further optimizations possible in the current table structure. Consider the table shown in Figure 9.

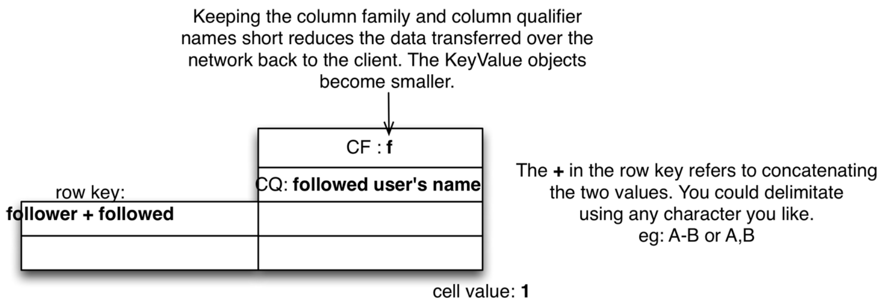
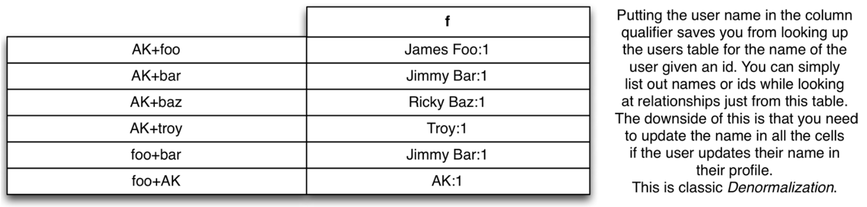

Keeping the column family and column qualifier names short reduces the data transferred over the network back to the client. The KeyValue objects become smaller.
CF : **f**
CQ: **followed user's name**
row key:
follower + followed
The **+** in the row key refers to concatenating the two values. You could delimitate using any character you like. eg: A-B or A,B
cell value: **1**

Figure 9: The relationship table with the row key containing the follower and the followed user

There are two things to note in this design: the row key now contains the follower and followed user; and the column family name has been shortened to f. The short column family name is an unrelated concept and could very well be done in the previous table as well. It just reduces the I/O load (both disk and network) by reducing the data that needs to be read/written from HBase since the family name is a part of every KeyValue [4] object that is returned back to the client. The first concept is what is more important here. Getting a list of followed users now becomes a short *Scan* instead of a *Get* operation. There is little performance impact of that as Gets are internally implemented as Scans of length 1. Unfollowing, and answering the question "Does A follow B?" become simple delete and get operations, respectively, and you don't need to iterate through the entire list of users in the row in the earlier table designs. That's a significantly cheaper way of answering that question, specially when the list of followed users is large.

A table with sample data based on this design will look like Figure 10.



	f
AK+foo	James Foo:1
AK+bar	Jimmy Bar:1
AK+baz	Ricky Baz:1
AK+troy	Troy:1
foo+bar	Jimmy Bar:1
foo+AK	AK:1

Putting the user name in the column qualifier saves you from looking up the users table for the name of the user given an id. You can simply list out names or ids while looking at relationships just from this table. The downside of this is that you need to update the name in all the cells if the user updates their name in their profile. This is classic *Denormalization*.

Figure 10: Relationship table based on the design shown in Figure 9 with some sample data

Notice that the row key length is variable across the table. The variation can make it difficult to reason about performance since the data being transferred for every call to the table is variable. A solution to this problem is using hash values in the row keys. That's an interesting concept in its own regard and has other implications pertaining to row key design which are beyond the scope of this article. To get consistent row key length in the current tables, you can hash the individual user IDs and concatenate them, instead of concatenating the user IDs themselves. Since you'll always know the users you are querying for, you can recalculate the hash and query the table using the resulting digest values. The table with hash values will look like Figure 11.

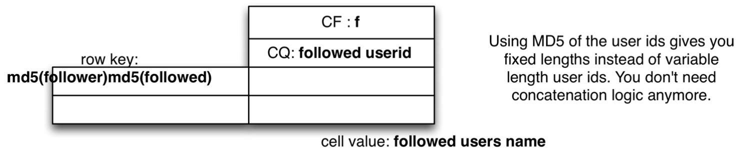

CF : **f**
CQ: **followed userid**
row key:
md5(follower)md5(followed)
cell value: **followed users name**
Using MD5 of the user ids gives you fixed lengths instead of variable length user ids. You don't need concatenation logic anymore.

Figure 11: Using MD5s as a part of row keys to achieve fixed lengths. This also allows you to get rid of the + delimiter that we needed so far. The row keys now consist of fixed length portions, with each user ID being 16 bytes.

This table design allows for effectively answering all the access patterns that we outlined earlier.

Summary

This article covered the basics of HBase schema design. I started with a description of the data model and went on to discuss some of the factors to think about while designing HBase tables. There is much more to explore and learn in HBase table design which can be built on top of these fundamentals. The key takeaways from this article are:

- Row keys are the single most important aspect of an HBase table design and determine how your application will interact with the HBase tables. They also affect the performance you can extract out of HBase.
- HBase tables are flexible, and you can store anything in the form of *byte[ ]*.
- Store everything with similar access patterns in the same column family.
- Indexing is only done for the *Keys*. Use this to your advantage.
- Tall tables can potentially allow you faster and simpler operations, but you trade off atomicity. Wide tables, where each row has lots of columns, allow for atomicity at the row level.
- Think how you can accomplish your access patterns in single API calls rather than multiple API calls. HBase does not have cross-row transactions, and you want to avoid building that logic in your client code.
- Hashing allows for fixed length keys and better distribution but takes away the ordering implied by using strings as keys.
- Column qualifiers can be used to store data, just like the cells themselves.
- The length of the column qualifiers impact the storage footprint since you can put data in them. Length also affects the disk and network I/O cost when the data is accessed. Be concise.
- The length of the column family name impacts the size of data sent over the wire to the client (in KeyValue objects). Be concise.

References

[1] Apache Hadoop project: http://hadoop.apache.org.

[2] Apache HBase project: http://hbase.apache.org.

[3] HBase client API: http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/package-summary.html.

[4] HBase KeyValue API: http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/KeyValue.html.

[5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation* (OSDI '06), USENIX, 2006, pp. 205–218.

What Takes Us Down?

STUART KENDRICK

Stuart Kendrick works as a third-tier tech at the Fred Hutchinson Cancer Research Center (FHCRC) in Seattle, where he dabbles in troubleshooting, deep infrastructure design, and developing tools to monitor and manage devices. He started earning money as a geek in 1984, writing in FORTRAN on CRAY-1 for Science Applications International Corporation, worked in desktop support, server support, and network support at Cornell University, and reached FHCRC in 1993. He has a BA in English, contributes to BRIITE (http://www.briite.org), and spends free time on yoga and CrossFit.
skendric@fhcrc.org

What are the causes of IT service disruption? With access to an email archive recording both planned and unplanned events, I figured I could identify ways to reduce downtime. This turned out to be neither as easy nor as useful as I had hoped: the exercise raised questions but little that was actionable. Still, the path I took may help you analyze your own data.

Environment

I work at the Fred Hutchinson Cancer Research Center, a nonprofit biomedical research institute specializing in cancer and infectious diseases. I pay attention to deep infrastructure: power, cooling, cabling, transport (Ethernet, IP, WiFi, Fibre Channel), interstitial services (DNS, DHCP, authentication, directory services), email, storage, and file services. These days, I spend my time managing our Problem Management process and leading Root Cause Analysis efforts. (Yes, the ITIL borg is extending its filaments into our brains.)

In the mid-90s, the network team started posting service-affecting incidents, both planned and unplanned, to an email list. Over time, more and more departments followed suit, and more and more techs subscribed to the list. We've refreshed that list server over the years, trashing its archives each time. The current archive starts in October 2000.

In our culture, planned downtime goes over well—we negotiate service level agreements (SLAs) with our divisions specifying when we can take down applications; we notify users; they modify their work flows to dodge the windows during which we are disrupting service; everyone is happy (for some value of happiness). But unplanned downtime is another matter—no one enjoys that, and we invest effort to avoid it.

Today, the Center employs 2500 staff with an annual budget of $450 million (85% from federal grants and contracts) and has 8000 active Ethernet ports, 11,000 active IP addresses, 450 KW datacenter cooling, 1 PB+ mass storage, and national and international collaborations. Roughly 30% of the end-stations run Windows, another 10% Linux, 5% OS X, and the remainder fall into the miscellaneous category (IP phones, printers, etc . . . well most of those run Linux, but I want the Linux figure to reflect desktops/laptops/servers only).

The Outages List

Techs send email to this list using a standardized format. In our lingo, all service-affecting events are called “outages”.

```
Subject:      Exchange 2003 Cluster Issues
Severity:     Critical (Unplanned)
Start:        Monday, May 7, 2012, 11:58
End:          Monday, May 7, 2012, 12:38
Duration:     40 minutes
Scope:        Exchange 2003
Description:  The HTTPS service on the Exchange cluster crashed, triggering
              a cluster failover.
User Impact:  During this period, all Exchange users were unable to access
              email.
              Zimbra users were unaffected.
Technician:   [xxx]
```

Figure 1: Example of an unplanned outage

```
Subject:      H Building Switch Upgrades
Severity:     Major (Planned)
Start:        Saturday, June 16, 2012, 06:00
End:          Saturday, June 16, 2012, 16:00
Duration:     10 hours
Scope:        H2 Transport
Description:  Currently, Catalyst 4006s provide 10/100 Ethernet to end-
              stations. We will replace these with newer Catalyst 4510s.
User Impact:  All users on H2 will be isolated from the network during this
              work.
              Afterward, they will have gigabit connectivity.
Technician:   [xxx]
```

Figure 2: Example of a planned outage

You can’t see this in the template above, but we also categorize outages by window:

```
Prime      Monday - Friday 7am - 6pm
SLA        Sunday 8pm - Monday 4am or Wednesday midnight - 4am
Shoulder   Any other time
```

Figure 3: Windows

Yes, that service level agreement (SLA) window looks pretty darn generous . . . we can take down services every single week for eight hours straight?! Conceptually, yes, but in practice, research institutes live and die by grant applications. Those grant applications have submission deadlines, and those deadlines pop up almost every week. Thus, most of those SLA windows get blocked.

The severity field has intention glued onto it via parentheses: planned (we intended the outage) or unplanned (we were surprised). Severity itself can contain the following values:

Drama	Most of the Center for 60+ minutes during prime time
Critical	One or more buildings or divisions
Major	Multiple floors or multiple departments
Minor	One floor or one department
None	No end-user effect

Figure 4: Severities

You might ask why we bother to have a Severity of None—doesn't sound like an outage if the end-user impact was None, right? Well, the motivation is two-fold. First, most of us want to be kept informed of changes to the environment (because what you change might in fact interfere with something I do), and the outages list serves as that forum. Second, we're trying to flush out errors in our understanding of the environment; if I claim that an event did not cause a service disruption and you know differently, you'll tell me (and then I'll send a correction to Outages).

Complicated and subjective? Yup.

The Outages Database

So I figured I'd write code to grab the list archives and crawl through them, creating a database entry for each outage. How hard could this be? After all, we use this structured template . . . Ahh, the naiveté and eternal optimism of youth: two weeks and 2500 lines of Perl later (the grossest code I've ever written), I ran it, took the partially processed results, imported into my database, and started scrubbing manually. Turns out we don't follow the template exactly: I never knew there were so many ways to write a date/timestamp, techs twink with the spelling of key words regularly, techs tend to send multiple messages describing each outage (the first announcing the event, the last announcing the completion of the event, and often others in between correcting errors or adding new information) . . . I'll quit whining here.

Furthermore, I wanted a feel for Service and Cause, neither of which is specified in the template. I added those during my manual passes.

Service	Description
Application	End-user facing apps (minus email, MIS, and printing)
Email	Exchange, Zimbra, mail relay, spam/malware scrubbers
HPC	High Performance Computing
Interstitial	DHCP, DNS, NTP, authentication, directory services
MIS	Financial Management / Human Resources systems
Power	Electricity
Print	Print servers
Storage	Anything providing file or block services
Transport	Ethernet, WiFi, Remote Access, Fibre Channel
Virtualization	VMware, Xen
Voice	Telephones and pagers

Figure 5: Services

This list reflects the focus of the groups who post to outages. Most of the application support groups do not post; I lump their contributions into a single category.

More interestingly, I wanted to identify the proximate cause of each outage—again, not something defined in the template, so I added this during my manual passes, interpreting the description field, dusting off memories, and making a judgment call.

Cause	**Description**
Cockpit Error	Techie mistake (fat finger, config error, bump the power cord)
Design Failure	Service didn't behave as the designer intended
External Services	Service provider issue (electric utility, ISP, telecom carrier)
Hardware Failure	The magic smoke escaped
Maintenance	Patching, database compression, shuffling data, minor fixes
Malware	Virus or worm infection
Overload	Too much of a good thing
Software Bug	Memory leak, unhandled exception, Blue Screen of Death
Testing	Validating expected behavior, typically involving high-availability
Unknown	Never figured it out
Upgrade	Adding major functionality (new gear, major software update)

Figure 6: Causes

There's a lot of fuzz here. When we popped a ceiling tile trying to trace cables and knocked an unsecured electrical wire loose, it triggered an emergency power off (EPO) event in our largest datacenter. I categorized the service as Power and the cause as Design Failure. I could have categorized the service as Application (every application in that datacenter went down) and the cause as Cockpit Error (the electrician who installed the EPO circuits intended to screw that wire into place but forgot). I didn't because I try to push cause down the OSI stack (Power sits a whole lot lower than Application), and I try to pick the proximate cause as opposed to the root cause: at the moment we popped the ceiling tile, Power did not behave according to the datacenter designer's intent.

Or, to take another example, if the server stayed up, but became too slow to be usable on account of too many users, I categorized that as Overload. If the server crashed because it ran out of RAM (on account of too many users), I categorized that as Software Bug.

Yup, pretty darn subjective.

NOTES

- External services isn't really a cause, but since the service provider space is opaque to us (did the WAN circuit go down due to human error, an unannounced upgrade, or a power event?), we lump them together.
- Hardware Failure lumps together both unplanned events, during which the magic smoke escaped, and planned events, during which we replace, say, a disk controller which is failing diagnostics but hasn't actually fried yet.
- Maintenance is driven by patching, mostly Windows patching.
- Testing is driven by the network team, which reboots redundant switches and routers monthly.

Results

I ended up with ~2300 outages [1] spanning the last 11+ years. Is that an undercount? Definitely. Departments vary in how frequently they report: some use Outages rigorously, others not at all. And of the entries in Outages, I threw away hundreds which my code didn't parse or which were too terse for me to categorize manually.

Q: How often are we surprised?
A: We're surprised half the time.

Planned: 55% Unplanned: 45%

Q: What takes us down?
A: Software bugs take us down.

For unplanned outages, software bugs are the dominant contributor. And for planned outages, a third arise from maintenance, which is driven by patching, i.e., fixing software bugs.

Planned		Unplanned	
Cause	**Proportion**	**Cause**	**Proportion**
External Services	2%	Cockpit Error	13%
Maintenance	32%	External Services	7%
Other	14%	Hardware Failure	12%
Testing	11%	Other	7%
Upgrade	41%	Software Bug	61%

Figure 7: Planned vs. unplanned by cause

Q: When do we go down?
A: In the middle of the day.

If unplanned outages were to occur at random, regardless of time of day, we would predict that some unplanned outages would land during the Prime window (55 hours/week), most during the Shoulder window (101 hours/week), and a few during the SLA window (12 hours/week). But, in fact, we see that far more land during Prime time than we would expect based purely on chance—perhaps because our users are exercising the systems and uncovering bugs in the process.

Window	Predicted	Measured
Prime	33%	67%
Shoulder	60%	31%
SLA	7%	2%

Figure 8: Unplanned outages by window: predicted vs. measured

Q: What causes induce the most pain?
A: The same causes which induce major and minor pain.

I tried slicing and dicing in other ways but did not uncover new information. For example, when focusing just on the most painful events (Drama and Critical), causes break down pretty much the same as they did when considering all severities:

Planned		Unplanned	
Cause	**Proportion**	**Cause**	**Proportion**
External Services	4%	Cockpit Error	17%
Maintenance	30%	External Services	9%
Other	12%	Hardware Failure	8%
Testing	16%	Other	11%
Upgrade	38%	Software Bug	55%

Figure 9: Planned vs. unplanned by severity (Drama + Critical only)

Q: How often does it hurt a lot?
A: Severity shows a normal distribution.

We experience a few of the really painful Drama outages and a few outages with no end-user effect: most land in the middle.

	Planned	**Unplanned**
Drama	0%	5%
Critical	16%	19%
Major	46%	35%
Minor	34%	39%
None	4%	2%

Figure 10: Planned vs. unplanned by severity

Q: What breaks most often?
A: Transport and email are weak spots.

But see caveats below.

	Planned	**Unplanned**
Application	18%	16%
Email	20%	21%
Other	16%	20%
Storage	14%	8%
Transport	32%	35%

Figure 11: Planned vs. unplanned by service

Reality Check

Fuzzy Data

Those cute tables with numbers in them look good . . . but as I apply cultural knowledge, I lose confidence. For example, the network team founded the outages list; the email team jumped onto the bandwagon shortly thereafter: these two groups have been posting the longest and have become ruthless about reporting every event, no matter how embarrassing. Furthermore, they have the most mature monitoring systems, reporting even minor hiccups. Are Transport and Email our most fragile services? Or do they top the list because of cultural factors: habit, conscientiousness, visibility?

Cockpit Error

Reading thousands of descriptions of outages gave me a chance to smile—I remember many of these events from personal involvement and know each of the techs posting to the list. Senior techs tend to acknowledge their errors directly, using language like "I fumbled the configuration," "I accidentally typed rm –rf * from root," "I broke the Internet connection," whereas junior techs tend to slide into passive voice and circuitous language when they describe their errors: "The service went down during trouble-shooting," "It was discovered that the configuration file contained an error," "On investigation and after analysis, the power cord was found to be detached." Where possible, I flagged the cause as Cockpit Error, but I'm confident that I missed plenty. For that matter, I suspect that Cockpit Error leads to unreported outages, as techs try paddling up the Nile in their efforts to dodge embarrassment. We have a remarkably shame/blame-free environment—as far as I know, no tech has ever been fired for making a mistake and causing an outage. In fact, management likes to stress that making mistakes is how we learn (yeah, OK, sometimes they look a little nervous when they make this point, but still, the sentiment is there). How can we boost the Cockpit Error reporting rate?

Who Else Quantifies This Stuff?

A casual search turned up a handful of studies in this space, with variously sized data sets (typically 100–1000 incidents spanning 1–5 years).

	Gray [2]	Kuhn [3]	Enriquez [4]	Oppenheimer [5]			Offord [6]	Kendrick
Published	1990	1997	2002	2003			2011	2012
				SP1	SP2	SP3		
Cockpit Error	13%	25%	38%	33%	36%	19%	42%	17%
Software	58%	14%	7%	27%	25%	24%	38%	55%
Hardware	18%	19%	30%	25%	4%	10%	—	8%
Other	11%	42%	25%	10%	31%	33%	20%	20%

SP = Service Provider

Figure 12: Similar surveys

I'm skeptical that I'm comparing apples to apples here—both environments and methodologies vary widely. For example, Gray, Kuhn, and Enriquez were all analyzing data sets taken from homogeneous systems (Tandem Computers and the Public Switched Telephone Network), while Oppenheimer, Offord, and I are analyzing heterogeneous environments (Windows/Linux-based systems running on IP/Fibre Channel networks). Or, to take another example, Offord extracts his data set from the log of Root Cause Analysis jobs his company has performed for customers—not exactly outages but rather long-running problems. In 42% of their cases, the problem was fixed by making a configuration change, which I recategorized as Cockpit Error, in order to fit his data into my taxonomy—probably not a precise match.

Tentatively, I see all these data sets directing our attention toward software flaws and operator fumbles as places for improvement.

What to Do?

Software Bugs

For us, our unplanned downtime is driven by Software Bugs (~60%). We know that we lag on patching. When a service fails repeatedly, we'll investigate and often find a patch addressing the issue which the vendor shipped months or years prior. I would like to think that if we patched more regularly, we would convert unplanned outages into planned outages. Still, this is a tricky area—most of our teams don't have test environments (we are nonprofit after all)—so we test patches by running them in production, and as we all know, patches can fix issues we weren't having while introducing new issues. How many unplanned outages would we dodge by patching more aggressively?

Testing

Until recently, the network group tested their redundant routers and switches monthly, rebooting them in series, analyzing failure, fixing the issues they uncovered (typically Cockpit Errors, e.g., misconfigurations), working with sysadmins to fix misconfigured servers (servers which weren't configured to take advantage of the dual Ethernet switches in datacenters), and helping the security groups buff up highly available firewalls. Of our really painful planned outages, Testing contributed 16%. I would like to think this approach saved us a similar number of really painful unplanned outages and thus was a win. On the other hand, testing requires substantial staff time. How to quantify the costs and benefits?

Insights

I have been struck by the number of axes on which one can measure an incident. Each of the authors I cite developed their own taxonomy. To recap, here's mine:

Function	**Our Term**	**Description**
Pain Level	Severity	Drama, Critical, Major, Minor, None
Intention	Planned	Planned or Unplanned
Time Frame	Window	Prime, Shoulder, SLA
End-User Impact	Service	The thing that went down
Proximate Cause	Cause	What caused the downtime

Figure 13: Taxonomy

I am troubled by how subjective my categorization process is—I made multiple passes through the database, recategorizing as I became more familiar with my data; nevertheless, I expect that I made inconsistent choices. Also, many outages don't fit the taxonomy cleanly: what to do with a planned outage which incurred unplanned consequences? Or an outage which knocked out multiple services? And cause remains tricky—an outage has so many causes, how to pick just one?

Still, at the end of the day, I'm headed back to problem management meetings to suggest patching and testing as ways to convert unplanned events into planned ones.

Doubt is uncomfortable; certainty is absurd. —Voltaire

References

[1] See http://www.skendric.com/problem/incident-analysis for the summarized data.

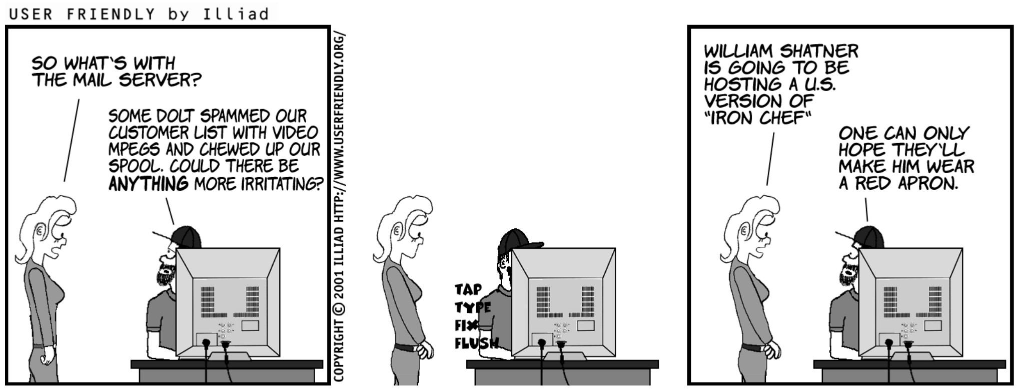
[2] Jim Gray, "A Census of Tandem System Availability Between 1985 and 1990," IEEE Transactions on Reliability, vol. 39, no. 4, pp. 409–418, Oct. 1990.On p.6, I used the All Faults column and categorized maintenance +operations + process as Cockpit Error.

[3] Richard Kuhn, "Sources of Failure in the Public Switched Telephone Network," IEEE Computer, vol. 30, no. 4, April 1997, pp. 31–36. I counted only errors from telco staff as Cockpit Error, allocating "Human error—external" to Other.

[4] P. Enriquez, A.B. Brown, and D. Patterson, "Lessons from the PSTN for Dependable Computing," Proceedings of the 2002 Workshop Self-Healing, Adaptive, and Self-MANaged Systems (Shaman), 2002, pp. 1–7. Again, I allocated "Human error—external" to Other.

[5] David L. Oppenheimer, A. Ganapathi, D. Patterson, "Why Do Internet Services Fail, and What Can Be Done About It?" USENIX Symposium on Internet Technologies and Systems, 2003.

[6] Paul Offord, "RPR Statistics," Advance7, October 2011. I map the sum of "bug fix" and "programming" into my Software Bug. Programming is Advance7's term for a bug fixable by internal resources, e.g., a bug found in an in-house application, while bug fix is Advance7's term for a bug found in software acquired externally, e.g., commercial or open source.


USER FRIENDLY by Illiad
SO WHAT'S WITH THE MAIL SERVER?
SOME DOLT SPAMMED OUR CUSTOMER LIST WITH VIDEO MPEGS AND CHEWED UP OUR SPOOL. COULD THERE BE ANYTHING MORE IRRITATING?
COPYRIGHT © 2001 ILLIAD HTTP://WWW.USERFRIENDLY.ORG/
TAP TYPE FIX FLUSH
WILLIAM SHATNER IS GOING TO BE HOSTING A U.S. VERSION OF "IRON CHEF"
ONE CAN ONLY HOPE THEY'LL MAKE HIM WEAR A RED APRON.

The Summer of Popping Power Supplies

CHARLES POLISHER

Charles Polisher is a system administrator for the Sierra Community College District in Rocklin, California. Charles began his career in computing in 1972.
cpolish@surewest.net

Around April 2008, Sierra College (Rocklin, California) had an unusual problem in the datacenter. Nobody remembers exactly when and how it started, but server power supplies began failing in unusual numbers and patterns. The senior system administrator remembers hearing "pop pop pop," perhaps a second apart, followed by the acrid smell of charred electronics. On investigation, the staff discovered that three power supplies in three adjacent racks had failed. Thus began a period of travail that some College staff find painful to remember.

The datacenter had been relocated to a building that had not previously housed a datacenter. Raised flooring was installed, power for a single row of racks was provisioned, and two residential air conditioners were installed. Mains power was piped in from an adjacent building. This was supplemented with a pair of Chloride UPSes, a 50 KW Wacker generator, and a manual transfer switch. As servers were added, AC power was extended underfloor with flex armored conduit. There were some server power supply failures right after the move, but nothing like when the problems started in April.

The datacenter manager recalls that the servers—there were around 35 at the time—mostly remained up during the trouble. Most equipment had redundant power supplies, and usually the failures were detected in time to swap in a replacement before the other supply failed. To decrease time to replacement, we implemented a continuous ping-sweep using Solar Winds' IP Sentry utility. This usually caught overnight server outages.

The highest estimate given for the total number of failed supplies was around 50. At that time the College's kit was mostly Dell. Dell balked at the high rate of replacements, which forced us to begin using third-party replacements. Occasionally a tier-2 server's second power supply would be shifted to make a tier-1 server's power fully redundant, shifting the risk of a total failure to lower-valued services.

The entire episode lasted a few months. The College's facilities people, who had installed the datacenter power, were not able to determine the cause. They recommended a local electrical contractor to consult on the problem.

A number of issues came to light. It was discovered that the underfloor boxes that held the electrical outlets had screws that were a tiny bit too long, which caused problems if the face plates were disturbed. The relative humidity was determined to be just 9%. This eventually resulted in replacing the residential air conditioners with a pair of substantial commercial chillers that also maintain humidity at correct levels. Apparently, the facilities people and the contractor didn't agree on the contribution of unbalanced power draw from the legs of the 3-phase circuit, issues with power factor, or harmonics. But they did agree that grounding was a problem. The addition of 10 to 20 servers over time almost certainly was a contributing factor.

A number of changes were made that together effectively ended the problem. The AC service was upgraded to 400 amps. The generator, undersized for its job, was replaced with a 100 KW Cummins diesel unit. The transfer switch was replaced with an automatic one, and the UPSes were replaced with a much larger pair. Grounding rods were driven into the ground, the floor pedestals and racks were brazed to large-gauge ground wires, a ground buss bar was installed, and all the grounds were connected there.

That was nearly the end of the problems. But in 2011 another (possibly) related problem cropped up. Commercial power failed, the UPS began supplying the datacenter, the generator started and stabilized, and the transfer switch attempted to shift the load to the generator. But, possibly because of a floating neutral, the UPS would not "take" the generator power. Once the batteries ran out, the datacenter went dark. The ground was found to be inadequate. New grounding rods were sunk and the buss bar was replaced.

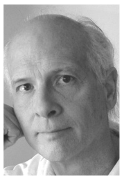
What can we learn from all this? First, I suggest careful attention to power, especially grounding, when building a datacenter. Second, even under adverse circumstances, it is possible to provide relatively reliable services with relatively unreliable parts. Which has been the story of automatic computing since its inception [1].

References

[1] Claude E. Shannon, "Von Neumann's Contributions to Automata Theory," *Bull. Amer. Math. Soc.,* 1958, pp. 123–129: http://www.ams.org/bull/1958-64-03/S0002-9904-1958-10214-1/S0002-9904-1958-10214-1.pdf (http://tinyurl.com/c2538jw).

Jacob Farmer on Managing Research Data
An Interview

RIK FARROW


Rik is the editor of *;login:*.
rik@usenix.org

Jacob Farmer is an industry-recognized expert on storage networking and data protection technologies. He has authored numerous papers and is a regular speaker at major industry events such as Storage Networking World, VMworld, Interop, and the USENIX conferences. Inside the data storage industry, Jacob is best known for having authored best practices for designing and optimizing enterprise backup systems and for his expertise in the marketplace for emerging storage networking technologies. Follow him on Twitter @JacobAFarmer.
jfarmer@cambridgecomputer.com

I am always looking for interesting system administration articles, so when Doug Hughes told me that Jacob Farmer was working on a project where they tagged, that is, added metadata to files stored at the Harvard Medical School, I asked Farmer to write an article about their project. Prodding Farmer, in the form of emailing him questions, worked much better, as I discovered that even if I called in the evening, often Farmer would still be working on the project.

Rik: We heard through the grapevine that Cambridge Computer is doing some kind of data management project with Harvard Medical School that involves metadata, backups, and data migration. What can you tell us about the project?

Jacob: Indeed, my team is engaged with Harvard Medical School to do some novel things with regard to managing unstructured research data. The formal goal is to develop a more intimate understanding of how storage resources are consumed by various research groups. The folks at HMS feel that they are always reacting to storage technology demands. They feel that if they can better understand the needs of their researchers, they can be more proactive in providing storage resources. Like many data-intensive research organizations, HMS would like to have a long-term storage strategy, and that starts by sorting out what they have now and measuring the trends.

Rik: Is this kind of analysis a new service that Cambridge Computer is offering, or is this a one-off professional services engagement?

Jacob: My company has begun to offer this kind of analysis as a service, but what's really going on is that HMS has agreed to be an early adopter for a software product that my company is developing, and this project is simply the first of many projects that we hope to tackle with the software. About a year ago, I shared my product vision with the CIO at Harvard Med School. He really liked the vision and offered to be an early adopter/guinea pig. We have been running our software at HMS for about a year now across about 500 TB worth of files.

Rik: Does your software product have a name, and can you tell us in a nutshell what the vision is that HMS found so appealing?

Jacob: The code name for our product is Starfish. Starfish is a platform that enables users and applications to associate metadata with files and directories in conventional network file systems. The metadata is then used to enable better organization, more insightful reports, and to trigger storage management rules.

Our basic premise is that traditional file systems do not provide enough insight into the files they are storing to allow meaningful reporting or storage management policies. Meanwhile, more structured repositories and content management systems are too restrictive to be embraced by researchers. Researchers are notoriously resistant to efforts by the IT department to put structures and restrictions on the way they work. For instance, they resist directory naming conventions, seldom delete unneeded files, etc. Nonetheless, institutions need to impose rules and structure if they want to have a prayer at managing the explosive volumes of data typical of research computing. Our approach is to layer on the metadata and rules in an unobtrusive way.

Rik: That sounds like a very broad vision. What kinds of specific problems do you anticipate tackling with Starfish?

Jacob: I break the potential solutions into three top-level categories: IT infrastructure, curation/collaboration, and governance. When it comes to IT infrastructure, conventional solutions for backup, archiving, and tiered storage are all limited by the information available in file-system attributes. As such, their policies are not very granular, which is why conventional solutions fail when faced with hundreds of terabytes and zillions of files.

I then lump collaboration and curation together, because they both require meaningful metadata. Our software would enable collaboration by organizing files in ways that would be meaningful to third parties. As for curation, I don't anticipate that our metadata framework would meet the standards of a digital preservation professional, but we enable their mission by providing the lower-level storage management functions that their metadata systems do not perform, and we can facilitate the handoff from research to formal curation.

When it comes to governance, the combination of metadata and rules is very powerful. For instance, Starfish could be used to isolate files with data involving human subjects or export restrictions. Starfish could also be configured to provide administration or auditing for data management plans that were committed in a grant proposal.

Rik: Okay, that is a very broad vision. How does one product do all of that in any meaningful way?

Jacob: Admittedly, Starfish can't do all of these things, but we believe that all of these problem areas have some common elements, and Starfish will provide the framework for tackling a variety of hard problems with a single foundation. Most specifically, you can't do anything without having better insight into the business value of your files. To that end, Starfish provides a framework for associating metadata with files and directories. All of this information is incredibly valuable from a search and reporting standpoint, but it becomes even more valuable when you can do stuff based on the information.

Rik: Would you say that the project at HMS been a success so far? Do you have any data points you can share?

Jacob: I can certainly share a few highlights. We sampled roughly a half petabyte of miscellaneous files. We found that roughly 180 TB had not been touched in more than three years. There were roughly 20 TB of files with words such as *trash* and *junk* in the file name, and another 20 TB with *archive* in the file name. All told about 40% of the total file storage are candidates for being stored on a lower cost tier. We still have to get user buy-in before moving any files, and we believe we have work to do to determine the most user-friendly way to migrate those files away and put them back if they are ever needed.

Rik: Just finding that out sounds useful to me. You mentioned that HMS has a laundry list of projects that they hope your software will enable. What's next?

Jacob: One of our next big projects is to help users visualize the cost of data storage, especially as they relate back to projects and grants. One possibility is to use the costing data for charge-backs. Another is simply to encourage good citizenship. Either way, try to imagine an individual storage consumer being able to visualize how much capacity and cost is in each folder in a directory and then having the tools to demote and promote files from one tier to another with full visibility into the cost implications. That's the kind of stuff we are hoping to do … and at very large scale.

Rik: I can imagine that a diverse research institute such as Harvard Med has at least one of every kind of storage management problem, so are you devoting 100% of your efforts on Harvard, or are you branching out with other early adopter clients?

Jacob: We have about a dozen installations, mostly in the life sciences, but we also have installations in financial services and in semiconductor manufacturing. In each case, the client has a specific objective, and we are working together to see how Starfish can help them meet the objective. For instance, at Fred Hutchinson Cancer Research Center, they are motivated by detailed reporting and by matching storage capacities to grants and projects. At Indiana University Bloomington, we are working together to define best practices for facilitating data management plans for grant-funded research. At Dana-Farber [Cancer Institute], we are exploring data protection and life-cycle management. Meanwhile, I have some commercial clients tinkering with using metadata to automate pipelines and others trying to use their automated pipelines as a way of capturing metadata.

Rik: You say that you are targeting research institutions. I would think these kinds of solutions would benefit any IT department in any industry. How are the needs of researchers different from those of traditional enterprise?

Jacob: Data-intensive research is a good niche for us. First, we have an extensive client base, so we are familiar with the problems, and people know us and are excited to work with us. Second, research data tends to flow through pipelines that really lend themselves to policy-based data management. Typically, there are raw data that fit the WORN paradigm: Write Once, Read Never. Then there are intermediate results, which often fit the WORSE paradigm: Write Once, Read Seldom if Ever. Then there are final results. Often there are multiple steps in the processes. Often there are offshoots of the main research.

Another interesting twist with researchers is that they are more concerned about preserving the old stuff than they are about protecting their most recently created files. New files often can be regenerated, but the old files might be needed to support a publication. For example, if researchers lose data from an experiment they ran yesterday, they can re-run the experiment. If they lose data from five years ago that formed the basis of the paper they are trying to publish, they might fail in their scientific mission. In a bank or insurance company, the business is much more concerned about minimizing downtime and preventing even the tiniest loss of newly created data.

Finally, researchers tend to have funky collaboration needs. Your typical enterprise only shares files within the enterprise, conforming to the paradigm of LDAP and POSIX permissions. Researchers often need to collaborate with other institutions or sometimes isolated users in their same institution. The granting agencies seem to be showing favor to programs that involve cross-institutional collaboration. Meanwhile, the granting agencies are mandating that researchers specify in their grant proposals what their plan is for retaining data and sharing data with interested third parties, which is a daunting problem when you have large data sets stored on a NAS behind the firewall.

Rik: Still, it sounds like this kind of technology could benefit other industries. Why limit yourself?

Jacob: Yes, there are potential applications for our software in other industries, but there are plenty of good companies pursuing intelligent file management for the traditional enterprise. Meanwhile, research is still a very broad niche. For instance, we have a lot of interest coming from libraries and museums. The libraries, museums, and other institutions of cultural preservation have very comprehensive metadata management systems for digital content curation, but they lack tools that interface with storage devices. For instance, digital librarians love our ability to verify the data integrity of their digital objects in an automated and audited way. They also like our ability to enable tiered storage and backups. One of the really cool things we are looking into is facilitating the data handoff between research computing and the libraries.

Rik: I would like to understand a bit more about the technology. For starters, I'm having a little trouble visualizing where Starfish sits relative to applications, users, and storage devices. It sounds like it would have to sit in the data path.

Jacob: Quite the contrary. Several vendors over the years have tried to virtualize file systems with devices that sit in the data path between the storage device and the users/applications. Just about all of these vendors died out early because these devices introduce latency, complexity, and often impose a least-common denominator effect on your fancy storage devices. Our model is to sit to the side of the storage device for I/O intensive workloads. We will sit in the data path for archival or cloud access.

Rik: If you are out-of-band, how do you control direct access to the files? Do you have to lock down the file servers somehow?

Jacob: You have hit on both our magic and our imperfection. Because we are not in the data path, we have to do our best to figure out what happened, where it happened, and when. Some NAS and file system products can produce a log or post events that we can monitor whenever there are changes made to the file system. It is also possible for us to learn of file system changes through the GUI or API. Worst case, we have to re-crawl the file system from time to time, but the good news is that we have a really fast crawler.

Rik: Okay, but what if a user deletes a bunch of files that are referenced by the metadata. How would you prevent that from happening?

Jacob: We can't really prevent a user from deleting files on a production file system, unless we programmatically modify permissions or apply a read-only flag. If there are metadata associated with deleted files, our system can report on the fact that the files are now gone. If you are really worried about files getting deleted, then you configure our software to impose a backup policy that puts a copy of the file in a safe place. Now if the file is deleted, we can still associate the metadata with the backup copy and we can present an option to restore the file back to the production file server.

Rik: So, in summary, it sounds like you are crawling file systems on a regular basis and making a big database that tracks each file and directory. Is there any secret sauce or unique intellectual property that differentiates you in the marketplace?

Jacob: Candidly, our software is not doing anything that has not been done before by clever sysadmins or programmers around the world. What makes the software special is that we engineered enterprise-class software to do all of these things robustly and reliably and that scales to handle the capacities and numbers of files that you find in big research institutions.

I will give you an example from one of our early adopters. A few years ago, the client's IT department had written a Perl script to crawl their file systems, make a database record for each directory, and then associate directories with various engineering projects. The goal was to be able to look up a project in the database and see all relevant directories across all file servers and geographies. Similarly, they wanted to look up a directory and see what project it was associated with. They wrote the software. It worked. But then they grew from a handful of file servers to hundreds and from a few dozen terabytes to petabytes. The software buckled under the load. Then they had some turnover in the IT department, the code was abandoned, and now they have no solution to the problem. Our software just drops in and gives them the same functionality, just at a larger scale. We could do a whole lot more for them, but this is all they need and they can't seem to find it anywhere else other than writing it themselves.

Rik: When members of the USENIX community think of Cambridge Computer, we don't necessarily think of a software company. Does this project represent a departure from your traditional business model?

Jacob: Yes and no. Yes, in that this is the most ambitious software project we have ever taken on. We have done software applications before, but typically for very niche-y solutions or for developing tools for our field services people to gather metrics at our clients' sites.

That said, I feel that this project is otherwise a natural extension of my day job. In my traditional business, I work like a broker or agent. I help my clients narrow down and select the right storage technologies for their project. I try to do that without bias toward any particular vendors, and I get paid in the form of commissions when the client buys something. When my clients present me with a problem, I have all the incentive in the world to help them find the right vendor because that vendor will invariably pay me a finder's fee or commission. Over the past few years, my research clients have been presenting me with problems for which I can't find viable solutions, no matter how far and wide I go shopping. After a few years of this, I felt the inspiration to make it myself. In other words, I would never have the vision or the impetus for this project if I were not talking all day long to research institutions about their storage needs.

Rik: What kinds of resources are you investing in the project? Would you describe this as a skunkworks project, or something more formal?

Jacob: This is a real engineering effort with a seven-figure budget and full-time dedicated employees. My director of engineering is a former client of mine. In his past job, he built a SaaS application that provided e-discovery services for large legal cases. His system handled several billion files with all kinds of complex search and metadata. We are trying to build something similar, except even larger scale and for scientists instead of lawyers.

Rik: Have you considered making the software available as open source to encourage wider adoption?

Jacob: We have not made any decisions with regard to making the code available through an open source license. We are committed to openness by exposing a comprehensive API, but for now we feel we have to stay focused on building a really robust core product.

Rik: Final question: How can the USENIX community help?

Jacob: Two ways. For starters, we are hiring developers, so everyone please keep your ears open for developers who might be a fit for us. We are looking for professional software engineers with skills in large-scale, big databases, Python, Django, storage management, etc. We are also still open to taking on a few more early adopters, especially if they are willing to collaborate with us on defining feature sets and doing user-acceptance testing.

COLUMNS

Practical Perl Tools

Mind Those Stables, I Mean Files

DAVID BLANK-EDELMAN

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010.
dnb@ccs.neu.edu

In the last column, we had a lovely visit with the XML Path Language, or XPath, followed by a brief look at how to bring its power to bear using Perl. Path-like things have been a bit of a leitmotif here over the past few columns, so I thought we might want to continue in this direction. This time we're going to look at a library/tool that takes this abstraction and puts it to practical use in an interesting way we haven't seen yet. As regular readers of this column will tell you, I loves me a good abstraction now and then. Sure do.

Just a quick warning: like last column, the majority of the words will describe the tool/API because once you have that all down pat, bringing Perl into the picture is both easy and a bit of an anti-climax (until you realized how cool what you just did with a single line of code actually is).

Augeas

The tool I have in mind to explore today is called Augeas. Augeas was developed under the aegis (oh, it is just raining Greek mythology references today) of the Red Hat's Emerging Technologies group who have sponsored a number of spiffy projects. Determining exactly what Augeas is can be tricky because the project site [1] describes the tool as:

- An API provided by a C library
- A command line tool to manipulate configuration from the shell (and shell scripts)
- Language bindings to do the same from your favorite scripting language
- Canonical tree representations of common configuration files
- A domain-specific language to describe configuration file formats

Or more concisely, "Augeas is a configuration editing tool. It parses configuration files in their native formats and transforms them into a tree. Configuration changes are made by manipulating this tree and saving it back into native config files."

But that still probably doesn't make things crystal clear unless you've already had the itch this is designed to scratch. I think I can demonstrate the problem to you so you can see exactly why this particular solution is so lovely. Let's take a little tour together around the /etc directory of your average UNIX root filesystem (kindly keep your hands inside the car at all times and don't forget to stop at the gift shop on the way out).

First stop, an excerpt from the local hostname to IP address mapping file, /etc/hosts::

```
127.0.0.1 localhost
127.0.1.1 precise32
```

Next some lines from the file that contains the authentication information for all of the local users on a machine, /etc/passwd. Here's an excerpt from the stock Ubuntu 12.04 password file::

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
```

Here are some lines from the config file for the service that runs commands at a set interval or time:

```
    # m h dom mon dow user        command
    17 *   * * *  root    cd / && run-parts --report /etc/cron.hourly
    25 6   * * *  root   test -x /usr/sbin/anacron || ( cd / && run-parts
--report /etc/cron.daily )
```

And finally, a few lines from the config file for the service that rotates log files:

```
/var/log/wtmp {
    missingok
    monthly
    create 0664 root utmp
    rotate 1
}

/var/log/btmp {
    missingok
    monthly
    create 0660 root utmp
    rotate 1
}
```

As much as it might be fun to try and write an entire column just by quoting files, I think this is probably enough to make my point. You don't have to be a seasoned sysadmin to see that none of these file formats look the same. And although a seasoned sysadmin can read, understand, and write these different formats in her or his head, getting a single piece of code to do that is much trickier. I've written my share of individual pieces of sed/awk/cut/Perl code to parse and rewrite all of these formats and more. Many times I've had to start from scratch because the formats were more different than similar.

Here's where Augeas comes into the picture. Augeas provides a framework for creating plugins (which they call *lenses*, a term I don't like all that much) that can read and write the formats for all of these different kinds of files and many more. When Augeas reads in the data from a file, it transforms the data into a tree that you can manipulate. When Augeas writes that tree data back out again, the data is written in the native format for that file.

Tree Talk

Just like we had to do some head scratching in my last column about how to go from a XML or HTML document to a tree, again I think in behooves us to make sure we get the concept. In fact, extra scrutiny is warranted in this case because that tree's construction and manipulation is central to the whole process.

Augeas trees always have at least two children at the top level: /augeas and /files. /augeas is a place to find information about augeas operations and configuration. Information like which lens is being used to parse a file, any errors reported in that parsing, global configuration information, etc. all live in this part of the tree. We're not going to talk about this branch at all in this column, but knowing about it may come in handy for you some day.

The branch of the tree that interests us the most is /files, where you can find (you guessed it) the configuration data found in your files. If we wanted to see the data from the files above, we would look in:

```
/files/etc/hosts/
/files/etc/passwd/
/files/etc/crontab/
/files/etc/logrotate.conf/
```

respectively. This list of paths brings up two interesting points:

1. We'll be referencing our configuration data using its position in the filesystem. Why is that so interesting? It is worth noting because it give you a sense of what Augeas is *not*. Augeas is not trying to be an all-encompassing abstraction framework that elides all of the details of how a system is implemented. If you are using a system that keeps its hosts file in /var/etc/hosts for some bizarre reason, Augeas will not "standardize" the data by putting it in /files/etc/hosts. At best Augeas can be taught to use the "hosts" lens when it sees something in /var/etc with that name, but it isn't going to provide a standardized tree independent of the underlying filesystem details.
2. I left the trailing slash on those paths to pique your curiosity. All of the data for each file lives in a sub-tree that starts with the name of the file. From that point in the tree, we see a further elaboration of point #1. The data found in the different file branches is represented in a way best suited for each file type. Again, pay attention to what Augeas is *not* doing here; it is not trying to represent every configuration file format in some uniform tree structure.

To see best what I mean in detail #2, let's look at the sub-trees from the first two files in our list. I'm going to reproduce just the part of the tree related to the lines I excerpted above. In /files/etc/hosts, we can see:

```
/files/etc/hosts
/files/etc/hosts/1
/files/etc/hosts/1/ipaddr = "127.0.0.1"
/files/etc/hosts/1/canonical = "localhost"
/files/etc/hosts/2
/files/etc/hosts/2/ipaddr = "127.0.1.1"
/files/etc/hosts/2/canonical = "precise32"
```

In /files/etc/passwd, we see this:

```
/files/etc/passwd
/files/etc/passwd/root
/files/etc/passwd/root/password = "x"
/files/etc/passwd/root/uid = "0"
/files/etc/passwd/root/gid = "0"
/files/etc/passwd/root/name = "root"
/files/etc/passwd/root/home = "/root"
/files/etc/passwd/root/shell = "/bin/bash"
/files/etc/passwd/daemon
/files/etc/passwd/daemon/password = "x"
/files/etc/passwd/daemon/uid = "1"
/files/etc/passwd/daemon/gid = "1"
/files/etc/passwd/daemon/name = "daemon"
/files/etc/passwd/daemon/home = "/usr/sbin"
/files/etc/passwd/daemon/shell = "/bin/sh"
```

Let me draw your attention to one similarity and one difference between the two trees. Both file trees have leaves representing the different fields of their respective records. For /etc/hosts, you can see a leaf for each IP address; for /etc/passwd, you can see a leaf for each uid, gid, shell, etc.

But now a key difference to note: the first file has a sub-tree for every line in the file, and the second has a sub-tree for every login name. In the first case, the lens author has chosen to let you distinguish the "record" you are working with by line number (.../1/..., .../2/..., and so on); in the second case, you would specify the record of interest using the login name instead of its place in the file.

Eagle-eyed readers are no doubt thinking, "Hey, everything has been bunnies hopping through the meadow so far, but what happens when the records are duplicated or a record has multiple leaves of the same type?" Ok, let's find out. Let's edit our hosts file to have this as the first line:

```
127.0.0.1        localhost huey dewey louie
```

and let's change /etc/passwd to have two bin accounts (not a good idea, I know):

```
bin:x:2:2:bin:/bin:/bin/sh
bin:x:22:2:bin:/bin:/bin/sh
```

Augeas borrows the numeric predicate notation from the XPath playbook to handle these cases and creates sub-trees like this:

```
/files/etc/hosts/1
/files/etc/hosts/1/ipaddr = "127.0.0.1"
/files/etc/hosts/1/canonical = "localhost"
/files/etc/hosts/1/alias[1] = "huey"
/files/etc/hosts/1/alias[2] = "dewey"
/files/etc/hosts/1/alias[3] = "louie"
```

and:

```
/files/etc/passwd/bin[1]
/files/etc/passwd/bin[1]/password = "x"
/files/etc/passwd/bin[1]/uid = "2"
/files/etc/passwd/bin[1]/gid = "2"
/files/etc/passwd/bin[1]/name = "bin"
/files/etc/passwd/bin[1]/home = "/bin"
/files/etc/passwd/bin[1]/shell = "/bin/sh"
/files/etc/passwd/bin[2]
/files/etc/passwd/bin[2]/password = "x"
/files/etc/passwd/bin[2]/uid = "22"
/files/etc/passwd/bin[2]/gid = "2"
/files/etc/passwd/bin[2]/name = "bin"
/files/etc/passwd/bin[2]/home = "/bin"
/files/etc/passwd/bin[2]/shell = "/bin/sh"
```

This means we will use a number within a square bracket to distinguish which branch of the tree we care about. An important thing to note here is Augeas is strict about ordering within the tree. Lines from a file go into the tree—and come out of it—in the same order they are found in the file.

Augeas' Muse, XPath

So you don't think the XPath reference a moment ago is unintentional, Augeas lets you reference things such as

```
/files/etc/passwd/bin[last()]
```

to get the last *bin* line from the configuration data. And indeed, we can now bring some of the power of XPath's query capability we saw in the last column to Augeas. For example, we could request all of the logins with the bogus shell of /bin/false:

```
/files/etc/passwd/*[shell = '/bin/false']
```

One example from the Augeas documentation (slightly modified):

```
/files/etc/hosts/*/ipaddr[../alias='huey']
```

This query finds the IP addresses of the host with the alias *huey*.

There are a bunch more selection and querying operations you can do using a syntax that is a kissing-cousin to XPath. Rather than enumerate them all, I'd like to point you to last *;login* issue's Practical Perl Tools column and the Augeas documentation. Now let's look at what you can do with everything we've discussed so far. For instance, what if you could do something you couldn't do with XPath, such as change values and whole records?

Let's Do It

Changing things sounds pretty spiffy, no? Let's get at it. The way I recommend you start working with Augeas is to install it on a machine using either a pre-built package or grab the latest source from github [2]. For build instructions, see the HACKING [3] file once you clone the repository.

Two quick OS X-related warnings as of this writing:

1. A key part of Augeas 0.10.0, as made available through both Homebrew and MacPorts, builds fine but fails as soon as you try to run it. If you want to run Augeas on OS X, you'll have to build from source, which leads to:
2. The latest version of Augeas requires a more recent version of Bison than the OS X default of 2.3. MacPorts can build a compatible Bison version; Homebrew mainline does not have that available.

For more details on where to get Augeas, including the language-specific binding (Perl is only one of many available), see their download page [4]. You may also be able to download the language binding of your choice from the same pre-built source as your main Augeas distribution. For example, Ubuntu makes the libconfig-augeas-perl package available.

The main distribution comes with a nifty command-line tool called augtool, which is a great way to play around with Augeas in an interactive setting. One hint if you are going to do this: augtool can take a -- root flag that will chroot() it to a directory of your choice. This lets you play with augtool on a directory of configuration files mocked up to look like a real root filesystem (one such directory comes with the distribution) without worrying about zorching your real configuration data. Another good way to experiment is within a throw-away virtual machine.

Augtool has built-in help, but mostly you can stick to a few simple commands:

- *ls* to show an Augeas path (e.g., ls /files/etc/passwd/bin)
- *print* to recursively print out the contents of a sub-tree, which is the command I used for the previous output
- *match* to do an XPath-like search on the tree
- *set* to set a value (more on this later)
- *rm* to remove an entire sub-tree (e.g., to remove a line from a config file)
- *ins* to insert a sub-tree (e.g., to insert a line into a file)
- *save* to write the tree data back out to a file with all changes made

Let's play with a few of these commands. I promised change, so let's change one of the alias values in the first line of the hosts file we were using:

```
$ sudo augtool
augtool> set /files/etc/hosts/1/alias[3] 'phooey'
augtool> save
Saved 1 file(s)
augtool> quit
$ grep phooey /etc/hosts
127.0.0.1 localhost huey dewey phooey
```

Here we've set the third alias value for the first line and confirmed that change actually took. Now, let's deal with that icky duplicate bin entry in /etc/passwd:

```
$ sudo augtool
augtool> rm /files/etc/passwd/bin[2]
rm : /files/etc/passwd/bin[2] 7
augtool> save
Saved 1 file(s)
augtool> quit
$ grep bin:x /etc/passwd
bin:x:2:2:bin:/bin:/bin/sh
```

Gone, baby gone. Augtool also has told us the size of the sub-tree we just removed.

Augtool also works for searches like the ones I mentioned above:

```
augtool> match /files/etc/passwd/*[shell = '/bin/false']
/files/etc/passwd/syslog = (none)
/files/etc/passwd/messagebus = (none)
/files/etc/passwd/ntp = (none)
/files/etc/passwd/vboxadd = (none)
/files/etc/passwd/statd = (none)
```

The `= (none)` part of the results is meant to indicate that the nodes selected by the query do not have values of their own (i.e., they are at the top of their respective file tree). Here's an example where we are asking for something that does contain a value:

```
augtool> match /files/etc/hosts/*/ipaddr[../alias='huey']
/files/etc/hosts/1/ipaddr = 127.0.0.1
```

Using This from Perl Is Going to Be Trivial, Right?

Yeah, yeah it is. All of the things we were just doing in augtool look remarkably similar using the Config::Augeas Perl module:

```
use Config::Augeas;

my $aug = Config::Augeas->new();
$aug->set('/files/etc/hosts/1/alias[3]', 'kablooey');
$aug->save;
```

Or, another example

```
use Config::Augeas;

my $aug = Config::Augeas->new();
my @matches = $aug->match(q{/files/etc/passwd/*[shell = '/bin/false']});
print join ("\n",@matches);
```

yields

```
/files/etc/passwd/syslog
/files/etc/passwd/messagebus
/files/etc/passwd/ntp
/files/etc/passwd/vboxadd
```

Don't you love it when a plan comes together? Now you have super powers when it comes to reading and editing system config files from Perl thanks to Augeas. Take care and I'll see you next time.

References

[1] Augeas: http://augeas.net/.

[2] Augeas on github: https://github.com/lutter/augeas.

[3] HACKING: https://github.com/lutter/augeas/blob/master/HACKING.

[4] Augeas download page: http://augeas.net/download.html

Secrets of the Multiprocessing Module

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses.
dave@dabeaz.com

One of the most significant additions to Python's standard library in recent years is the inclusion of the multiprocessing library. First introduced in Python 2.6, multiprocessing is often pitched as an alternative to programming with threads. For example, you can launch separate Python interpreters in a subprocess, interact with them using pipes and queues, and write programs that work around issues such as Python's Global Interpreter Lock, which limits the execution of Python threads to a single CPU core.

Although multiprocessing has been around for many years, I needed some time to wrap my brain around how to use it effectively. Surprisingly, I have found my own use differs from those often provided in examples and tutorials. In fact, some of my favorite features of this library tend not to be covered at all.

In this column, I decided to dig into some lesser-known aspects of using the multiprocessing module.

Multiprocessing Basics

To introduce the multiprocessing library, briefly discussing thread programming in Python is helpful. Here is a sample of how to launch a simple thread using the threading library:

```
import time
import threading

def countdown(n):
    while n > 0:
        print "T-minus", n
        n -= 1
        time.sleep(5)
    print "Blastoff!"

t = threading.Thread(target=countdown, args=(10,))
t.start()
# Go do other processing
...
# Wait for the thread to exit
t.join()
```

Granted, this is not a particularly interesting thread example. Threads often want to do things, such as communicate with each other. For this, the Queue library provides a thread-safe queuing object that can be used to implement various forms of producer/consumer problems. For example, a more enterprise-ready countdown program might look like this:

```
import threading
import Queue
import time

def producer(n, q):
    while n > 0:
        q.put(n)
        time.sleep(5)
        n -= 1
    q.put(None)

def consumer(q):
    while True:
        # Get item
        item = q.get()
        if item is None:
            break
        print "T-minus", item
    print "Blastoff!"

if __name__ == '__main__':
    # Launch threads
    q = Queue.Queue()
    prod_thread = threading.Thread(target=producer, args=(10, q))
    prod_thread.start()

    cons_thread = threading.Thread(target=consumer, args=(q,))
    cons_thread.start()
    cons_thread.join()
```

But aren't I supposed to be discussing multiprocessing? Yes, but the above example serves as a basic introduction.

One of the core features of multiprocessing is that it clones the programming interface of threads. For instance, if you wanted to make the above program run with two separate Python processes instead of using threads, you would write code like this:

```
import multiprocessing
import time

def producer(n, q):
    while n > 0:
        q.put(n)
        time.sleep(5)
        n -= 1
    q.put(None)
```

```
def consumer(q):
    while True:
        # Get item
        item = q.get()
        if item is None:
            break
        print "T-minus", item
    print "Blastoff!"

if __name__ == '__main__':
    q = multiprocessing.Queue()
    prod_process = multiprocessing.Process(target=producer, args=(10, q))
    prod_process.start()

    cons_process = multiprocessing.Process(target=consumer, args=(q,))
    cons_process.start()
    cons_process.join()
```

A Process object represents a forked, independently running copy of the Python interpreter. If you view your system's process viewer while the above program is running, you'll see that three copies of Python are running. As for the shared queue, that's simply a layer over interprocess communication where data is serialized using the pickle library.

Although this example is simple, multiprocessing provides a whole assortment of other low-level primitives, such as pipes, locks, semaphores, events, condition variables, and so forth, all modeled after similar constructs in the threading library. Multiprocessing even provides some constructs for implementing shared-memory data structures.

No! No! No!

From the previous example, you might get the impression that multiprocessing is a drop-in replacement for thread programming. That is, you just replace all of your thread code with multiprocessing calls and magically your code is now running in multiple interpreters using multiple CPUs; this is a common fallacy. In fact, in all of the years I've used multiprocessing, I don't think I have ever used it in the manner I just presented.

The first problem is that one of the most common uses of threads is to write I/O handling code in servers. For example, here is a multithreaded TCP server using a thread-pool:

```
from socket import socket, AF_INET, SOCK_STREAM
from Queue import Queue
from threading import Thread

def echo_server(address, nworkers=16):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(5)

    # Launch workers
    q = Queue(nworkers)
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()

    # Accept connections and feed to workers
    while True:
        client_sock, addr = sock.accept()
        print "Got connection from", addr
        q.put(client_sock)

def echo_client(work_q):
    while True:
        client_sock = work_q.get()
        while True:
            msg = client_sock.recv(8192)
            if not msg:
               break
            client_sock.sendall(msg)
        print "Connection closed"

if __name__ == '__main__':
    echo_server(("",15000))
```

If you try to change this code to use multiprocessing, the code doesn't work at all because it tries to serialize and pass an open socket across a queue. Because sockets can't be serialized, this effort fails, so the idea that multiprocessing is a drop-in replacement for threads just doesn't hold water.

The second problem with the multiprocessing example is that I don't want to write a lot of low-level code. In my experience, when you mess around with Process and Queue objects, you eventually make a badly implemented version of a process-worker pool, which is a feature that multiprocessing already provides.

MapReduce Parallel Processing with Pools

Instead of viewing multiprocessing as a replacement for threads, view it as a library for performing simple parallel computing, especially parallel computing that falls into the MapReduce style of processing.

Suppose you have a directory of gzip-compressed Apache Web server logs:

```
logs/
    20120701.log.gz
    20120702.log.gz
    20120703.log.gz
    20120704.log.gz
    20120705.log.gz
    20120706.log.gz
    ...
```

And each log file contains lines such as:

```
124.115.6.12 - - [10/Jul/2012:00:18:50 -0500] "GET /robots.txt HTTP/1.1" 200 71
```

```
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /ply/ HTTP/1.0" 200
11875
210.212.209.67 - - [10/Jul/2012:00:18:51 -0500] "GET /favicon.ico HTTP/1.0"
404 369
61.135.216.105 - - [10/Jul/2012:00:20:04 -0500] "GET /blog/atom.xml HTTP/1.1"
304 -
...
```

This simple script takes the data and identifies all hosts that have accessed the robots.txt file:

```
# findrobots.py

import gzip
import glob

def find_robots(filename):
    '''
    Find all of the hosts that access robots.txt in a single log file
    '''
    robots = set()
    with gzip.open(filename) as f:
        for line in f:
            fields = line.split()
            if fields[6] == '/robots.txt':
                robots.add(fields[0])
    return robots

def find_all_robots(logdir):
    '''
    Find all hosts across an entire sequence of files
    '''
    files = glob.glob(logdir+"/*.log.gz")
    all_robots = set()
    for robots in map(find_robots, files):
        all_robots.update(robots)
    return all_robots

if __name__ == '__main__':
    robots = find_all_robots("logs")
    for ipaddr in robots:
        print(ipaddr)
```

The above program is written in the style of MapReduce. The function find_robots() is mapped across a collection of filenames, and the results are combined into a single result—the all_robots set in the find_all_robots() function.

Suppose you want to modify this program to use multiple CPUs. To do so, simply replace the map() operation with a similar operation carried out on a process pool from the multiprocessing library. Here is a slightly modified version of the code:

```
# findrobots.py

import gzip
import glob
import multiprocessing

# Process pool (created below)
pool = None

def find_robots(filename):
    '''
    Find all of the hosts that access robots.txt in a single log file
    '''
    robots = set()
    with gzip.open(filename) as f:
        for line in f:
            fields = line.split()
            if fields[6] == '/robots.txt':
                robots.add(fields[0])
    return robots

def find_all_robots(logdir):
    '''
    Find all hosts across and entire sequence of files
    '''
    files = glob.glob(logdir+"/*.log.gz")
    all_robots = set()
    for robots in pool.map(find_robots, files):
        all_robots.update(robots)
    return all_robots

if __name__ == '__main__':
    pool = multiprocessing.Pool()
    robots = find_all_robots("logs")
    for ipaddr in robots:
        print(ipaddr)
```

If you make these changes, the script produces the same result, but runs about four times faster on my machine, which has four CPU cores. The actual performance will vary according to the number of CPUs available on your machine.

Using a Pool as a Thread Coprocessor

Another handy aspect of multiprocessing pools is their use when combined with thread programming. A well-known limitation of Python thread programming is that you can't take advantage of multiple CPUs because of the Global Interpreter Lock (GIL); however, you can often use a pool as a kind of coprocessor for computationally intensive tasks.

Consider this slight variation of our network server that does something a bit more useful than echoing data—in this case, computing Fibonacci numbers:

```
from socket import socket, AF_INET, SOCK_STREAM
from Queue import Queue
from threading import Thread
from multiprocessing import Pool
```

```
pool = None   # (Created below)

# A horribly inefficient implementation of Fibonacci numbers
def fib(n):
    if n < 3:
        return 1
    else:
        return fib(n-1) + fib(n-2)

def fib_client(work_q):
    while True:
        client_sock = work_q.get()
        while True:
            msg = client_sock.recv(32)
            if not msg:
                break
            # Run fib() in a separate process
            n = pool.apply(fib, (int(msg),))
            client_sock.sendall(str(n))
        print "Connection closed"

def fib_server(address, nworkers=16):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(5)

    # Launch workers
    q = Queue(nworkers)
    for n in range(nworkers):
        t = Thread(target=fib_client, args=(q,))
        t.daemon = True
        t.start()

    # Accept connections and feed to workers
    while True:
        client_sock, addr = sock.accept()
        print "Got connection from", addr
        q.put(client_sock)

if __name__ == '__main__':
    pool = Pool()
    fib_server(("",15000))
```

If you run this server, you'll find that it performs a neat little trick. For each client that needs to compute fib(n), the operation is handed off to a pool worker using pool.apply(). While the work takes place, the calling thread goes to sleep and waits for the result to come back. If multiple client threads make requests, the work is handed off to different workers and you'll find that your server is processing in parallel. Under heavy load, the server will take full advantage of every available CPU. The fabled GIL is not an issue here because all of the threads spend most of their time sleeping.

Note that this technique of using a pool as a coprocessor also works well in applications involving asynchronous I/O (i.e., code based on select-loops or event handlers), but because of space constraints, you'll just have to take my word for it.

Multiprocessing as a Messaging Library

Perhaps the most underrated feature of multiprocessing is its use as a messaging library from which you can build simple distributed systems. This functionality is almost never mentioned, but you can find it in the multiprocessing.connection submodule.

Setting up a connection between independent processes is easy. The following is an example of a simple echo-server:

```
# server.py
from multiprocessing.connection import Listener
from threading import Thread

def handle_client(c):
    while True:
        msg = c.recv()
        c.send(msg)

def echo_server(address, authkey):
    server_c = Listener(address, authkey=authkey)
    while True:
        client_c = server_c.accept()
        t = Thread(target=handle_client, args=(client_c,))
        t.daemon = True
        t.start()

if __name__ == '__main__':
     echo_server(("",16000), "peekaboo")
```

Here is an example of how you would connect to the server and send/receive messages:

```
>>> from multiprocessing.connection import Client
>>> c = Client(("localhost",16000), authkey="peekaboo")
>>> c.send("Hello")
>>> c.recv()
'Hello'
>>> c.send([1,2,3,4])
>>> c.recv()
[1, 2, 3, 4]
>>> c.send({'name':'Dave','email':'dave@dabeaz.com'})
>>> c.recv()
{'name': 'Dave', 'email': 'dave@dabeaz.com'}
>>>
```

As you can see, this is not just a simple echo-server as with sockets. You can actually send almost any Python object—including strings, lists, and dictionaries—back and forth between interpreters. Thus, this connection becomes an easy way to pass data structures around. In fact, any data compatible with the pickle module should work. Further, there is even authentication of endpoints involving the authkey parameter, which is used to seed a cryptographic HMAC-based authentication scheme.

Although the messaging features of multiprocessing don't match those found in a library such as ZeroMQ (0MQ), you can use the messaging to perform much of the same functionality, if you're willing to write a bit of code. For example, here is a server that implements a Remote Procedure Call (RPC):

```
# rpcserver.py
from multiprocessing.connection import Listener, Client
from threading import Thread

class RPCServer(object):
    def __init__(self, address, authkey):
        self._functions = { }
        self._server_c = Listener(address, authkey=authkey)

    def register_function(self, func):
        self._functions[func.__name__] = func

    def serve_forever(self):
        while True:
            client_c = self._server_c.accept()
            t = Thread(target=self.handle_client, args=(client_c,))
            t.daemon = True
            t.start()

    def handle_client(self, client_c):
        while True:
            func_name, args, kwargs = client_c.recv()
            try:
                r = self._functions[func_name](*args,**kwargs)
                client_c.send(r)
            except Exception as e:
                client_c.send(e)

class RPCProxy(object):
    def __init__(self, address, authkey):
        self._conn = Client(address, authkey=authkey)
    def __getattr__(self, name):
        def do_rpc(*args, **kwargs):
            self._conn.send((name, args, kwargs))
            result = self._conn.recv()
            if isinstance(result, Exception):
                raise result
            return result
        return do_rpc

# Sample usage
if __name__ == '__main__':
    # Remote functions
```

```
def add(x,y):
    return x+y
def sub(x,y):
    return x-y

# Create and run the server
serv = RPCServer(("localhost",17000),authkey="peekaboo")
serv.register_function(add)
serv.register_function(sub)
serv.serve_forever()
```

To access this server as a client, in another Python invocation, you would simply do this:

```
>>> from rserver import RPCProxy
>>> c = RPCProxy(("localhost",17000), authkey="peekaboo")
>>> c.add(2,3)
5
>>> c.sub(2,3)
-1
>>> c.sub([1,2],4)
Traceback (most recent call last):
  File "", line 1, in
  File "rpcserver.py", line 37, in do_rpc
    raise result
TypeError: unsupported operand type(s) for -: 'list' and 'int'
>>>
```

Final Words

The multiprocessing module is a tool worth keeping in your back pocket. If you are performing MapReduce-style data analysis, you can use process pools for simple parallel computing. If you are writing programs with threads, pools can be used like a coprocessor to offload CPU-intensive tasks. Finally, the messaging features of multiprocessing can be used to pass data around between independent Python interpreters and build simple distributed systems.

References

[1] Multiprocessing official documentation: http://docs.python.org/library/multiprocessing.html.

iVoyeur
Crosstalk

DAVE JOSEPHSEN

Dave Josephsen is the author of *Building a Monitoring Infrastructure with Nagios* (Prentice Hall PTR, 2007) and is senior systems engineer at DBG, Inc., where he maintains a gaggle of geographically dispersed server farms. He won LISA '04's Best Paper award for his co-authored work on spam mitigation, and he donates his spare time to the SourceMage GNU Linux Project.
dave-usenix@skeptech.org

The very moment that my three-year-old niece picked up the little scrap of PVC pipe, I knew what would happen. I saw in my mind's eye her baseball-bat swing into the back of her cousin's head, and knew it for the inevitable truth as predictably as the quadratic formula. As he lay crying on the floor it occurred to me that, being what we are, certain objects speak to us. Tools meet the hand and, for better or worse, beg to be used. They inspire us to action, but we need not learn to obey them; on the contrary, we spend years learning to resist.

Our primordial connection to tools is why there is a store in the mall where I can purchase a samurai sword, and it's why we say things like, "When all you have is a hammer, everything looks like a nail." I think it's something so deeply ingrained in us that it transcends physical objects. That may be presumptions of me, but personally, I often encounter software "solutions in want of a problem," by which I mean tools that are just so great I rack my brain trying to come up with a use for them.

So when I come across a quote like the following from Jon Gifford, "0MQ is unbelievably cool—if you haven't got a project that needs it, make one up" [1], I strongly relate. I even get excited. I can almost feel this 0MQ whatever-it-is grasped lightly in my hand, its weight as substantial as its balance is remarkable. It sings to me, and I realize it was made for me, and I for it; together, we are unstoppable. It shows me things: things that were, and are yet to be; things to smash; things, in fact, that cry out begging to be smashed. Together, we will drive elephants over the Alps, cross the Rubicon, defeat the infamous El Guapo and take back what is rightfully OURS.

You see how I get. Great tools inspire great deeds, especially if you can find a reason to use them, which, it turns out, is the case with 0MQ. But first things first; introductions are in order.

0MQ [2] (pronounced "Zero MQ") is, for lack of a better description, a sort of socket library. The goal of the project, briefly stated, is to "connect any code to any code, anywhere," and they're doing it by providing a simple, cross-platform, language-agnostic, asynchronous, messaging and concurrency protocol. 0MQ sockets automatically handle multiple simultaneous connections, provide fair queueing, and cooperate to form scalable multicore applications, using an interface you're already familiar with if you've done any socket programming. It is literally easier to use than it is to describe, but compared to traditional sockets there are two big differences.

First, compared to conventional sockets, which present a synchronous interface to either reliable, connection-oriented byte streams (SOCK_STREAM) or to unreliable, connection-less datagrams (SOCK_DGRAM), 0MQ sockets present an abstraction of an asynchronous message queue. The specific queueing behavior depends on the type of socket chosen by the programmer, but generally speaking, the programmer creates the socket, connects the socket, drops in a message, and that's it. There is no need to deal with the memory management associated with stream data on the client-end; 0MQ sockets know the size of the message and respond accordingly. Whereas conventional sockets transfer streams of bytes or datagrams, 0MQ sockets transparently transfer whole messages, quickly and safely.

Second, conventional sockets limit the programmer to one-to-one (two peers), many-to-one (many clients, one server), or in some cases one-to-many (multicast) relationships. Most 0MQ sockets (with one exception) can be connected to multiple endpoints, while simultaneously accepting incoming connections from multiple peers. Peer endpoints in either case may be other hosts on the network, other processes on the local machine, or other threads in the same process. The exact semantics of the message delivery depend on the type of socket chosen by the developer.

As I've said, there are several types of 0MQ sockets, each belonging to a "pattern," which corresponds to a style of distributed application framework. The available patterns are:

- Request-reply (classic TCP style client-server)
- Publish-subscribe (multicast, empowered, and simplified)
- Pipeline (think Hadoop style parallel application frameworks)
- Exclusive pair (inter-thread communication)

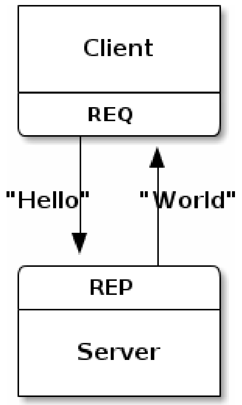


Figure 1: The 0MQ request-reply pattern

Sockets belonging to the same pattern are designed to work together to implement the pattern. The request-reply pattern, for example, is composed of a request socket and a reply socket. An application may use as many different patterns and socket types as it needs to accomplish its task, but socket types belonging to different patterns generally can't connect to each other directly. You cannot, for example, connect a PULL (pipeline) socket to a PUB (publish-subscribe) endpoint, but you could certainly write a single application that used both the publish-subscribe and pipeline patterns.

In this article, I'm only going to talk about two patterns, the request-reply, which is the most simplistic and therefore ideal for introductory examples, and the publish-subscribe pattern, which is the pattern that has me wanting to smash things.

The request-reply pattern, illustrated in Figure 1, is used for sending requests from a client to one or more instances of a service, and receiving subsequent replies to each request sent. Below is a Ruby script that implements a server on port 4242, which will wait for input from a 0MQ REQ (request) socket, and will reply with the word "foo":

```
require 'rubygems'
require 'ffi-rzmq'
context = ZMQ::Context.new(1)
socket = context.socket(ZMQ::REP)
socket.bind("tcp://*:4242")
```

```
while true do

request = ''
rc = socket.recv_string(request)
socket.send_string("foo")
end
```

In pseudo-code, the server ("responder" in 0MQ parlance):

- creates a socket of type REP (response);
- binds it to TCP port 4242 on 0.0.0.0/0 ;
- blocks waiting for a connection from a REQ (request socket);
- responds with the string "foo" once a connection is made (the message sent from the client is stored in the 'request' variable).

In the interest of word count, I'm going to keep the code listings to a minimum and just tell you that the client program does pretty much exactly the same thing in reverse; it:

- creates a socket of type REQ (request);
- connects it to TCP port 4242 on one or more servers;
- sends a message;
- blocks waiting for a response from the server.

The REQ and REP sockets work in lockstep; the client must send and then receive (in a loop, if necessary), and the server must receive and then send. Attempting any other sequence generates an error code. The server and client can start in any order, and 0MQ transparently handles multiple connections on either side, providing fair queueing if, for example, one client makes a connection every 500 milliseconds and another connects only once every two seconds. The developer is free to implement whatever protocol he likes over the connection, binary, or text; 0MQ does not know anything about the data you send except its size in bytes.

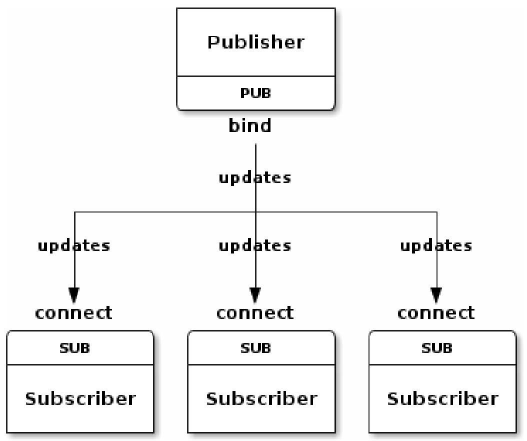


Figure 2: The 0MQ publish-subscribe pattern

The publish-subscribe pattern, depicted in Figure 2, is a multicast pattern, used for one-to-many distribution of data from a single publisher to multiple subscribers. A host wishing to distribute data creates a socket of type PUB, binds it to an address, and writes data to it as often as the application demands. A host wishing to receive data from the publisher creates a socket of type SUB, and connects it to the server. The PUB-SUB socket pair is asynchronous: the client receives in a loop while the server sends. Neither socket blocks waiting for the other.

Subscribers may connect to more than one publisher, using one "connect" call each time, and the received data will be interleaved on arrival so that no single publisher drowns out the others. A publisher that has no connected subscribers will simply /dev/null all its messages. Subscribers in a Pub-Sub relationship may specify one or more filters to control the content they're interested in receiving.

Now, if you're one of the four people who read this column with any regularity (hi Mom), I can safely assume that you have a large unruly gaggle of monitoring systems that you've managed to integrate. I can pretty safely make this assumption because the preponderant quantity of words I devote to this column that aren't dedicated to ranting or poorly disguised fart jokes are devoted to integrating monitoring tool A with monitoring tool B. That I cannot introduce a new tool without talking about how to integrate it with the other tools you're already using is testimony to an interesting fact; namely, that nobody uses a single tool for monitoring anymore.

Of course, “nobody” is a lot of somebodys. I’m certainly wrong there. Somebody somewhere probably uses one big tool for monitoring, but they’re probably not reading a column that talks about integrating various tools all the time, unless they can’t find a better source for ranting and poorly disguised fart jokes (unlikely). The rest of us are using a bunch of tools—this for systems availability, that for router metrics, the other for Web analytics . . . you get the point. The tools themselves are beginning to reflect this, as each monitoring tool slowly makes the realization that it isn’t the center of the universe, that it is, in fact, a piece of a solution to a much larger need.

Five years ago we built tools like Cacti [3], which does everything from metrics polling, through data storage and into display—utterly self-contained. Today, we build tools like Graphite [4], a powerful display and analysis engine that’s optimized for simplicity of input—built for integration. I gave up on Cacti because it’s a dead end: you put SNMP data in, and there it stays forever. I love Graphite, because it takes any kind of data from any kind of system and is happy to share it everywhere. Graphite gives me a place to combine and compare the output of systems that cannot be made to talk to each other.

But what if we could go a few steps further. Imagine, every monitoring system—Ganglia [5], sflow [6], Nagios [7] , collectd [8], SNMPd [9], Graphite, etc.—publishing availability data and metrics using 0MQ Pub-Sub. All monitoring and metrics data available in a common syntax, using a common, lightweight, fast, message bus in a manner that didn’t require a centralized server or single point of failure. Every agent could publish data to any collector that was interested all the time, and every collector could have access to any data it wanted.

Now there’s a pipe to smack your cousin in the head with. The very fact of 0MQ’s existence makes our lack of that pipe seem like hubris and stupidity, although some baby-steps have begun in the general direction I’m describing[10, 11]. I suspect the real challenge will be in coming up with a standard syntax for describing availability and metric data and getting a couple of the big players to adopt it. Maybe someone should hold a BoF.

Take it easy.

References

[1] http://www.slideshare.net/IanBarber/zeromq-is-the-answer-php-tek-11-version.

[2] http://www.zeromq.org/.

[3] http://www.cacti.net.

[4] http://graphite.wikidot.com/.

[5] http://ganglia.sourceforge.net/.

[6] http://www.sflow.org/.

[7] http://www.nagios.org.

[8] http://collectd.org/.

[9] http://net-snmp.sourceforge.net.

[10] https://github.com/mariussturm/nagios-zmq.

[11] https://github.com/jedi4ever/gmond-zmq.

/dev/random

ROBERT G. FERRELL

Robert G. Ferrell is a fourth-generation Texan, literary techno-geek, and finalist for the 2011 Robert Benchley Society Humor Writing Award.
rgferrell@gmail.com

Fall is in the air, not to mention all over the ground and floating in my new pool. With autumn comes cooler weather, which may seem like a blessing in this year of record-breaking heat, even for some of you who ordinarily dread the encroachment of Jack Frost and his minions. Good thing all those climate scientists who've been predicting global warming are wrong, eh? Otherwise, this might be a worrying trend. In my neck of the scrub brush—San Antonio, Texas—any lessening of the thermal oppression is welcomed by all and sundry.

One of the topics I saw proposed for this month's issue of the magazine you are currently holding concerned self-destructing DC power supplies. The topic got me thinking about all of the computers I've owned and administered in my life—a number that requires scientific notation to express—and their propensity for autolysis. Computers, at least the ones that have dominated my existence for the past 35 years, tend to experience three fundamental failure modes: insidious, inopportune, and incendiary. I3 == bad juju. (Notice that I use two equals signs rather than one, which brands me as an authentic coding geek. Pay attention next time.)

Insidious failures are those that happen when you're not looking, often when the computer isn't even powered up. Main board and memory failures generally fall into this category. Many's the otherwise fair morning, vibrant with the promise of a productive day, that has been ruined utterly by a POST failure; serenade of clinical, cynical beep codes; or just flat refusal to power up. Those weird, unpredictable errors that spring from thermal cycling, transient memory boo-boos, and the infamous *gremlins in the system* are all examples of this genre of frustration-inducing activity.

The inopportune fail mode is for the most part blatant and occurs, as the name suggests, in such a manner as to maximize the deleterious consequences. A classic example was the *Blue Screen of Death* (BSOD) fail during Bill Gates' 1998 COMDEX introduction to Windows 98. For the next four or five years, I giggled every time I thought of that. In fact, I'm still giggling. Another member of this genus is the computer that locks up just as you are reaching to save an hour's worth of complex word processing or spreadsheet data. This failure is also known as a TYHO (Tear Your Hair Out) error.

My favorite failures have to be the various BSOD incidents in simulators for military aircraft and surface ships. *If they happen in real missions, that's classified and I can't talk about it.*

> "You have a bogey at nine o'clock, Major. Engage him with your missiles!"
> "Um, I tried that, but first the console said something about a memory read error, then the little hourglass came up. Now it doesn't respond at all."

"You're hit! Bail out! Repeat: bail out!
"No can do. It says *'Action failed. Please see your system administrator.'"*
"Climb out of the cockpit manually, then."
"Can't. Canopy interlocks are computer-controlled. Wait...the screen is coming back on. Just in time, too. Hold it...never mind."
"Why? What does it say?
"'It is now safe to turn off your computer.'"
"Bummer. Any last words, Major?"
"Linux rocks!"

The third—and perhaps least-predictable—failure mode is what I have termed *incendiary* for two reasons: this failure tends to make the failure recipient very hot-headed, and it literally can trigger open flames. Although I didn't actually read the DC power supply failure article/proposed article/outline—whatever form and degree of completion in which it manifested—I imagine that such an event could easily fall within this category of failure. The foremost example that springs to mind for me is a PCI graphics card I once owned that decided to fail with a pronounced flourish. First the housing for the fan that cooled the GPU exploded noisily, followed fairly rapidly by a little jet of flame that might have burned down my entire house had I not been nearby when it chose to erupt. Using keyboard cleaner compressed air, a move that incurs extra geek points, I snuffed out the flame and yanked the wreckage out of my mainboard. The scene looked like an appropriately scaled aircraft had crashed into the GPU with the loss of all souls; in aircraft accident investigation parlance, this would be a CFIS, or *Controlled Flight into Semiconductor.* Since that incident, I have avoided this particular manufacturer because I prefer my domicile in the pre-combustion state.

Of course, incendiary doesn't necessarily have to equate to high temperature. Incendiary can also refer to occurrences that by their nature engender anger. Not that I wasn't angry when my video card started spitting fire, but that anger was secondary to sheer panic. Other incendiary failures do not result in panic or ignition of household furnishings, such as the infamous autocorrect *feature* that has torpedoed nearly every Internet user at one time or another, usually at a most unpropitious moment.

Let us, for the sake of exemplification, take a hypothetical situation in which a charming young man of good breeding wishes to express gratitude to his mother-in-law for the thoughtful gift of a pair of booties knitted by her very hand for the newborn child of the young man and his wife, the gift-giver's daughter. As said wife is driving the booties home from Mom's house, the young man composes a short—but heartfelt—expression of these thanks on his smartphone. "Dear Mom," he types, "Thank you so very much for the booties. Sherry and I love them, and so will little Radisson." Tragically, the smartphone really isn't, and its dictionary has not come pre-loaded with the word *booties.* The phone decides to auto-correct with a word that means *seabirds related to the gannet.* Mom's response is outside the scope of this treatise and can be found in a companion volume called, "Why Wars Start."

What about hardware or software failures that don't fall easily into one of the above categories? I happen to have come up with a suitable answer to this question, the deeply ingrained wisdom of which, upon sober reflection, should be apparent to even the most simple-minded of my readers: Stop engaging in useless hair-splitting and get—for the love of all that is good and wholesome and promotes sound UNIX-related IT policy—a freaking life.

BOOKS

Book Reviews

ELIZABETH ZWICKY, WITH MARK LAMOURINE

Team Geek: A Software Developer's Guide to Working Well with Others

Brian W. Fitzpatrick and Ben Collins-Sussman

O'Reilly, 2012. 160 pp.

ISBN 978-1-449-30244-3

The Developer's Code: What Real Programmers Do

Ka Wai Cheung

Pragmatic Bookshelf, 2012. 141pp.

ISBN 978-1-934356-79-1

These make a nice pair; each is the programmer's equivalent of a glossy management book, with simple, good advice well packaged. Simple ideas are easy to read and easy to believe in and sadly very, very hard to implement. I like these ideas a lot.

Team Geek is a unified book in smoothly integrated chapters with pictures; *Developer's Code* is in short essays. Otherwise, they differ primarily as suggested by the subtitles. *Team Geek* is mostly about teamwork, as a team member or team leader; *Developer's Code* is mostly about subjects that are closer to the code itself.

In *Team Geek,* as always, I disagree with some of the details of the specifics (the compliment sandwich, like the direct order, has its place; the trick is recognizing that place on the rare occasions you encounter it). But that's inevitable, and the general outlines are spot-on. Most people underestimate the importance of being honest and nice.

Revolution in the Valley

Andy Hertzfeld et al.

O'Reilly, 2005. 290 pp.

ISBN 978-1-449-31624-2

The real story of the Macintosh. If you're at all interested in computing history, or "how it was done" stories, you should read this. It's a top-notch job of being as non-fictional as possible, and it beautifully captures the combination of insanity and exhilaration that goes into groundbreaking work.

This is not a story I was part of, and it's before I came to Silicon Valley, so in some sense it's not a world I know. And yet, at the same time, it is the world I knew then (a time when we saved the foam that one of our computers came in because it was handy for sleeping on when you had been programming too long and needed a nap). Furthermore, it is also the world of the most recent startup I was at. Having the stories told by the participants gives this a sense of truth that most tales of computing history don't possess.

This came in (reprinted) for review right around the time Steve Jobs died, and I put off looking at it partly because of the noise at the time. Although Steve Jobs appears, this is not a book about him, and it depicts him with the full complexity of feelings that people had at the time.

The Design of Design

Frederick P. Brooks, Jr.

Addison-Wesley, 2010. 448 pp.

ISBN 978-0201362985

I love the author's classic book *The Mythical Man Month* with a passion, and I was therefore prepared to adore this as well. Unfortunately, I did not. It's nice enough; there are interesting insights into a number of things, most of them not directly related to programming, and anybody who has used JCL is sure to enjoy the discussion of how it came to be the worst programming language ever (the author's description, although I wholeheartedly agree).

If you're the kind of person who wants to engage in meta-cognition about the nature of design as an interdisciplinary undertaking, this is an interesting resource. If you were hoping for a straightforward sequel to *The Mythical Man Month,* you are liable to be disappointed.

Understanding IPV6, Third Edition

Joseph Davies

Microsoft Corporation, 2012. 640 pp.
ISBN 978-0-7356-5914-8

Another case of dashed expectations. I want to like IPv6. I want to understand IPv6. This is not the book to help a person do either. It is a detailed and careful reference work for IPv6, particularly on Microsoft platforms. If that's what you need—packet layouts, protocol details, all the relevant registry settings—then this is a fine book.

I would recommend skipping the prose, however. This is the paragraph, in Chapter 1, at which I gave up:

"One must consider, however, that the Internet, once a pseudo-private network connecting educational institutions and United States government agencies, has become an indispensable worldwide communications medium that is an integral part of increased efficiency and productivity for commercial organizations and individuals, and it is now a major component of the world's economic engine. Its growth must continue."

Things get better, but not lighter, when the topic turns to protocol details.

—Elizabeth Zwicky

CoffeeScript: Accelerated JavaScript Development

Trevor Burnham

Pragmatic Bookshelf, 2011. 127 pp.
ISBN 978-1-93435-678-4

I hadn't heard of CoffeeScript until I saw these books on the shelf at my local book store. I leafed through them wondering what kind of cutesy software would have that name and found a complete new browser language to replace JavaScript.

CoffeeScript is an attempt to address some of the long-lamented flaws of the JavaScript syntax. It adopts the design philosophy of more recent scripting languages, especially Python and Ruby. At the same time, it maintains close ties to JavaScript, even allowing in-line JavaScript if needed.

CoffeeScript is actually a translated language. The "compiler" produces JavaScript suitable for inclusion in HTML pages or for execution by a Node.js interpreter. The translator is in fact written in CoffeeScript and can execute in a browser (both authors explain how, and then why you shouldn't). More typically, it runs in a Node.js interpreter, and the resulting JavaScript output is what is used in production. It irks me to call the CoffeeScript interpreter a "compiler," but I will follow CoffeeScript convention in this.

CoffeeScript: Accelerated JavaScript Development is not a lengthy tome. Burnham states in the preface that readers should have at least some experience with JavaScript. It might be even more help if they can code Ruby. It is implicit that they should have some background in software development as well. You'll see why in a moment.

The opening chapter glosses how to install Node.js and the CoffeeScript compiler. It gives an example of how to compile CoffeeScript source code to JavaScript. It also shows how to run it as an interpreter in interactive mode.

Only three language constructs (writ large) are explicitly covered: Functions in Chapter 2, Collections and Iteration in Chapter 3, and Classes and Modules in Chapter 4. Burnham relies on the code examples, the reader's experience, and the similarities to Ruby to help the reader along. (There is also a Cheat Sheet for JavaScripters in Appendix 3.) The final two chapters discuss using CoffeeScript with JQuery and writing server side code with Node.js.

In keeping with the "pragmatic" theme in the Pragmatic Programmers series, the preface includes the description of a project that is used to illustrate the new concepts. Each chapter contains a coding section based on that project. The chapters close with a set of example questions designed to highlight the problems that most JavaScript programmers encounter when learning CoffeeScript.

Burnham provides three appendices to his book. I've already mentioned the cheat sheet for converting JavaScript constructs to CoffeeScript and back. The other two are the annotated answers to the chapter example questions and a section listing six different ways to run CoffeeScript.

CoffeeScript is a great introduction to the CoffeeScript language for someone who is familiar with and frustrated by the warts of JavaScript. It will be a very welcome book for someone who must use JavaScript and is comfortable with Ruby.

Programming in CoffeeScript

Mark Bates

Pearson Education, 2012. 283 pp.
ISBN-13: 978-03-2182010-5

If you're a JavaScript coder, *Programming in CoffeeScript* is meant to sit in a handy place on your desk. The Pearson "Developer's Library" series promises reference works for professional developers, and Bates' book fills the bill.

This is the first book I can remember that has a test for the reader in the preface. Bates presents a fragment of JavaScript and tells the reader to put the book down and get comfortable with JavaScript before returning. I think that's probably prudent as well.

The preface also contains the mandatory "How to Install CoffeeScript" section, but Bates reduces it to four short paragraphs in which he explains that he can't do better in print than the writers on the CoffeeScript Web site.

The first half of *Programming in CoffeeScript* is, as you would expect, an exposition of the language syntax and programming constructs. Bates walks thoroughly and methodically from a chapter on literals and variables through to classes and inheritance. He follows what I think is a fairly new convention of combining the traditional chapters on loops and arrays into a single chapter entitled "Collections and Iterations." This comes from the Functional Programming school, but I find it reasonable to highlight the relationship between collections and their most common operations. He still includes a chapter on classic logic and control structures but again combines them in a way that makes sense to me.

Bates' explicit reliance on his reader's prior knowledge lets him avoid lengthy expositions that can bog down texts that try to teach both a language and the theory of programming. This makes for a crisp, smooth read and makes the book suited for use as a long-term reference.

Bates also uses the reader's prior knowledge and the CoffeeScript compiler's remarkably clean output to his advantage. Every sample of CoffeeScript code in the book is followed by the resulting JavaScript code and by the output it generates when executed. I can hear the horrified gasps of aging C and C++ programmers who have looked under the cover of cpp(1) and cfront(1) output and of all those who hate XML because they cut their teeth on machine-generated data streams. The JavaScript is both nicely formatted and actually maps cleanly back to the CoffeeScript source. I've tried it myself, and it's not just the result of good typesetting. There are some constructs that take an extra look (the section on Binding and the -> and => operators at the end of the chapter on Classes jumps to mind), but Bates highlights them well and covers them in more detail as needed. In general, the samples in both languages give the reader a cross-reference which helps to clarify both the intent of the example and the proper meaning and use of the CoffeeScript language. I learned a few things about JavaScript too.

It is in the second half of the book that Bates dives into the practicum of using CoffeeScript.

It seems (and I think this is a good thing) that it has become accepted that no language is complete without both a unit-testing framework and some form of task-based build tool. Again, CoffeeScript borrows from Ruby. Cake is included with CoffeeScript. The structures generally follow Ruby's Rake (which in turn builds on lessons from Maven, Make, and others). Bates provides a nice set of sample tasks and then indicates that he generally uses Rake himself.

Because CoffeeScript translates to JavaScript, you can use any of the JavaScript unit-test frameworks to test your CoffeeScript. Bates suggests a framework called Jasmine, which is modeled on Ruby's RSpec and which has native CoffeeScript support.

The book concludes with four chapters in which Bates builds a Node.js service on CoffeeScript. By itself this wouldn't be a big deal, but in the process Bates introduces a number of other components which will be needed for real applications. These include an application framework, database integration, and client-server communications with both JQuery and a CoffeeScript MVC framework named Backbone.js.

Programming in CoffeeScript is a great introduction to the CoffeeScript language and development ecosystem. It should also get more than its share of dog-earing from regular use.

—Mark Lamourine

NOTES

USENIX Member Benefits

Members of the USENIX Association receive the following benefits:

Free subscription to *;login:*, the Association's magazine, published six times a year, featuring technical articles, system administration articles, tips and techniques, practical columns on such topics as security, Perl, networks, and operating systems, book reviews, and reports of sessions at USENIX conferences.

Access to *;login:* online from October 1997 to this month: www.usenix.org/publications/login/

Access to videos from USENIX events in the first six months after the event: www.usenix.org/publications/multimedia/

Discounts on registration fees for all USENIX conferences.

Special discounts on a variety of products, books, software, and periodicals: www.usenix.org/membership/specialdisc.html.

The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.

For more information regarding membership or benefits, please see www.usenix.org/membership/ or contact office@usenix.org. Phone: 510-528-8649

USENIX Board of Directors

Communicate directly with the USENIX Board of Directors by writing to board@usenix.org.

PRESIDENT
Margo Seltzer, *Harvard University*
margo@usenix.org

VICE PRESIDENT
John Arrasjid, *VMware*
johna@usenix.org

SECRETARY
Carolyn Rowland, *National Institute of Standards and Technology*
carolyn@usenix.org

TREASURER
Brian Noble, *University of Michigan*
noble@usenix.org

DIRECTORS
David Blank-Edelman, *Northeastern University*
dnb@usenix.org

Sasha Fedorova, *Simon Fraser University*
sasha@usenix.org

Niels Provos, *Google*
niels@usenix.org

Dan Wallach, *Rice University*
dwallach@usenix.org

CO-EXECUTIVE DIRECTORS
Anne Dickison
anne@usenix.org

Casey Henderson
casey@usenix.org

Students: Grab a Grant

Did you know that, with the help of our partners, USENIX offers student grants to cover USENIX conference and workshop registration fees and assist with travel and expenses?

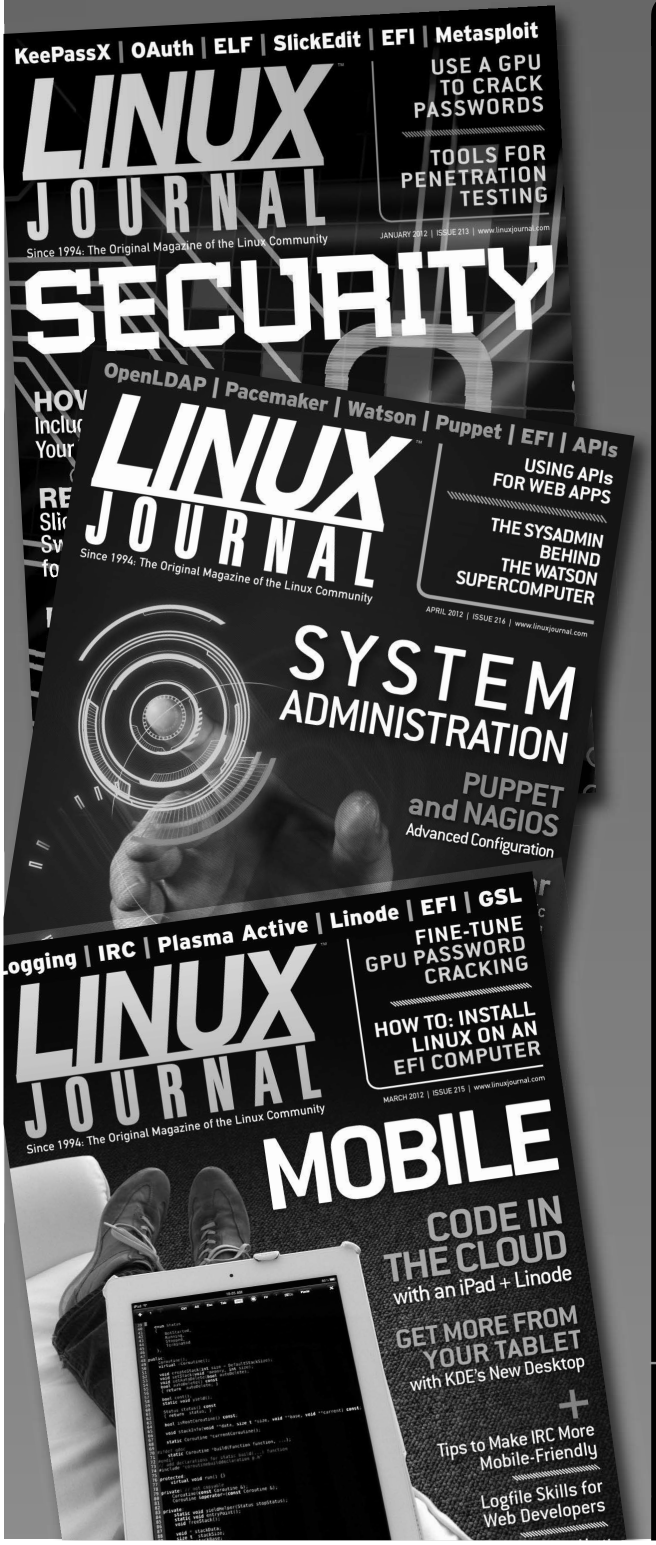
Recently we received a note from student grant recipient Roya Ensafi, a graduate student in computer science at the University of New Mexico. "I want to thank you for the USENIX student grant. It meant a lot to me," she writes. "Attending conferences are really a good opportunity not only to hear cool research ideas, but also meet interesting people, which makes it a great break from routine research work. Without the grant I wouldn't be able to come there."

Don't miss the November 5 student grant application deadline for LISA '12. Get the details at www.usenix.org/conference/lisa12/students, and then read "How a USENIX Student Grant Can Lead to a Career in Technology" at www.usenix.org/blog/student-grant-career.

For more news and updates like this, be sure to read the monthly USENIX News email.

Congratulations to the grant recipients who attended USENIX Security '12!

If You Use Linux, You Should Be Reading LINUX JOURNAL™


KeePassX | OAuth | ELF | SlickEdit | EFI | Metasploit
LINUX JOURNAL
USE A GPU TO CRACK PASSWORDS
TOOLS FOR PENETRATION TESTING
SECURITY
OpenLDAP | Pacemaker | Watson | Puppet | EFI | APIs
USING APIs FOR WEB APPS
THE SYSADMIN BEHIND THE WATSON SUPERCOMPUTER
SYSTEM ADMINISTRATION
PUPPET and NAGIOS Advanced Configuration
Logging | IRC | Plasma Active | Linode | EFI | GSL
FINE-TUNE GPU PASSWORD CRACKING
HOW TO: INSTALL LINUX ON AN EFI COMPUTER
MOBILE
CODE IN THE CLOUD with an iPad + Linode
GET MORE FROM YOUR TABLET with KDE's New Desktop
Tips to Make IRC More Mobile-Friendly
Logfile Skills for Web Developers

» In-depth information providing a full 360-degree look at featured topics relating to Linux

» Tools, tips and tricks you will use today as well as relevant information for the future

» Advice and inspiration for getting the most out of your Linux system

» Instructional how-tos will save you time and money

Subscribe now for instant access! For only $29.50 per year—less than $2.50 per issue—you'll have access to *Linux Journal* each month as a PDF, in ePub & Kindle formats, on-line and through our Android & iOS apps. Wherever you go, *Linux Journal* goes with you.

SUBSCRIBE NOW AT:
WWW.LINUXJOURNAL.COM/SUBSCRIBE

Conference Reports

In this issue:

For the complete 2012 USENIX Annual Technical Conference report and summaries from HotCloud '12, HotPar '12, HotStorage '12, and the panel at our first Women in Advanced Computing Summit, visit: www.usenix.org/publications/login.

2012 USENIX Annual Technical Conference (ATC '12)

Boston, MA
June 13-15, 2012

Opening Remarks

Summarized by Rik Farrow (rik@usenix.org)

Gernot Heiser (University of New South Wales) opened ATC by telling us that there had been 230 paper submissions, up by 30% from last year. Forty-one papers were accepted, after three reviewing rounds. Gernot reminded the audience that both OSDI and HotOS were coming up soon. Then Wilson Hsieh (Google) announced the best papers: "Erasure Coding in Windows Azure Storage," by Cheng Huang et al., and "netmap: A Novel Framework for Fast Packet I/O," by Luigo Rizzo.

Cloud

Summarized by Brian Cho (bcho2@illinois.edu)

Demand-Based Hierarchical QoS Using Storage Resource Pools

Ajay Gulati and Ganesha Shanmuganathan, VMware Inc.; Xuechen Zhang, Wayne State University; Peter Varman, Rice University

Imagine you are an IT administrator and your CIO asks that "all storage requirements be met, and when there is contention, don't allow the critical VMs to be affected." To get predictable storage performance, you could (1) overprovision, (2) use storage vendor products that provide QoS, or (3) provide QoS in VMs. This work looks at option 3, the goal being to provide better isolation and QoS for storage in VMs, using storage resource pools.

Ajay Gulati said that existing solutions specify QoS for each individual VM, but this level of abstraction has some drawbacks. Basically, virtual applications can involve multiple VMs running on multiple hosts—thus the need for a new abstraction, storage resource pools. Ajay reviewed an allocation model based on controls of reservation, limit, and shares.

Storage resource pools are placed in a hierarchical tree, with resource pools as intermediate nodes, and VMs at the leaves. The controls are defined on the pools as well as individual VMs. For example, the sales department and marketing department can be different resource pools with separately defined controls. These controls can be defined per-node, depending on parent, and the system can normalize these controls across the entire tree. In reality, a single tree is not used; rather, the tree is split up per datastore, but this was not detailed in the talk.

The system needs to periodically distribute spare resources, or restrict contending resources, among children in the tree, depending on the current system usage. This is done with two-level scheduling—first, split up the LUN queue limit between hosts; second, apply the queue limits by setting mClock at each VM. This is accomplished by two main steps: first, computing the controls per VM, based on demands, and second adjusting the per-host LUN queue depth. This is done every four seconds in the prototype. A detailed example of this on a small tree was presented.

A prototype was built on the ESX hypervisor, involving both a user-space module and kernel changes. Experiments were done with settings of six and eight VMs running different workloads. The results show timelines of throughput (in IOPS) per each VM, before and after changes to the controls. In summary, the system is able to provide isolation between pools, and sharing within pools.

There were no questions following the presentation.

Erasure Coding in Windows Azure Storage

Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin, Microsoft Corporation

▶ *Awarded Best Paper!*

Huseyin Simitci and Cheng Huang presented this paper together, with Huseyin starting. In Windows Azure Storage, the large scale means failures are the norm rather than the exception. In the context of storage, one question is whether to use replication or erasure coding (EC). With replication, you just make another copy, while with EC, you add parity. On failure, with replication you just read known data, while with EC you recreate the data. Both tolerate failure, but EC saves space, or can allow improved durability, with the same amount of space.

The Windows Azure Storage stream layer is an append-only distributed file system. Streams are very large files, split up into units of replication called extents. Extents are replicated before they reach their target size; once they reach the target, they are sealed (become immutable) and then EC is applied in place of replication. Previously, they used Reed-Solomon 6+3 as the conventional erasure coding: a sealed extent is split into six pieces, and these are coded into three redundant parity pieces. Huseyin concluded his part of the talk with a brief overview of practical considerations for EC.

Cheng focused on how Azure further reduces the 1.5x space requirement without sacrificing durability or performance. The standard approach to reduce the space requirement is to use Reed-Solomon 12+4 to decrease it to 1.33x. However, this makes reading expensive, and many times reconstruction happens during the critical path of client reads. It is better to achieve 1.33x overhead while only using six fragments. The key observation used to do this is the probability of failures. Conventional EC assumes all failures are equal, and the same reconstruction cost is paid on failure. However, for cloud storage, the probability of a single failure is much higher than that for multiple failures. So the approach taken is to make single failures more efficient. A 12+2+2 local reconstruction code (LRC) was developed. There are two local parities for each section of six fragments, and two global parities across all 12 fragments. In terms of durability, LRC 12+2+2 can recover from all three failures, and 86% of four failures. So the durability is between EC 12+4 and 6+3, which is "durable enough" for Azure's purposes.

LRC is tunable. You can tune storage overhead and reconstruction cost, given a hard requirement of three-replication reliability. Both Reed-Solomon and LRC are plotted as curves with the axes of reconstruction read cost vs. storage overhead. LRC gives a better curve, and the particular variant can be chosen looking at this curve. In the end, Azure chose 14+2+2, which, compared to Reed-Solomon 6+3, gives a slightly higher reconstruction cost (7 vs. 6) but has a 14% space savings, from 1.5x to 1.29x. Given the scale of the cloud, 14% is a significant amount.

Yael Melman, EMC, asked what happens when all three failures are in a single group. Cheng clarified that this does indeed work for all three failure cases, and that there are proofs of this in the paper. Richard Elling, DEY Storage Systems, asked how to manage unrecoverable reads. Cheng clarified that the failures discussed are storage node failures, not disk failures. Hari Subramanian, VMware, asked how to deal with entire disk failures. Huseyin answered that data is picked up by all remaining servers. Hari asked whether failing an entire node for a failed disk is less efficient. Huseyin clarified that entire nodes are not failed in this case, but rather that the granularity of failures considered is indeed disks. Someone asked about the construction cost when creating parity blocks—particularly the bandwidth cost involved. Cheng answered that the entire encoding phase is done in the background, not on the critical path. So you have the luxury of scheduling them as you like.

Composable Reliability for Asynchronous Systems

Sunghwan Yoo, Purdue University and HP Labs; Charles Killian, Purdue University; Terence Kelly, HP Labs; Hyoun Kyu Cho, HP Labs and University of Michigan; Steven Plite, Purdue University

Sunghwan Yoo began with an example using a KV store as motivation. He showed that many failures can happen in the chain of forwarding a request. The techniques used to mitigate these failures are retransmission, sequence numbers, persistent storage, etc. A single development team working on the whole system could make an effort to handle failures, but what if each component was handled by different teams and systems? Guaranteeing global reliability between independently developed systems is hard.

This motivates the development of Ken, a crash-restart-tolerant protocol for global reliability when composing independent components. It makes a crash-restarted node look like a slow node. Reliability is provided by using an uncoordinated rollback recovery protocol. Composability allows components to be written locally and work globally. An event-driven framework allows easy programmability—specifically, it is transparently applicable to the Mace system. These ideas (especially rollback recovery) are not in themselves new; Ken is a practical realization of decades of research.

When Ken receives a message from outside, an event loop begins—within this handler, the process can send messages and make changes to the memory heap. When the handler is finished, a commit is done, storing all changes made to a checkpoint file. An externalizer continually resends messages, to mask failures, making them look like slow nodes.

Another example was given, consisting of a seller, buyer, auction server, and banking server. If any of these systems show crash-restart failures, there are problems. Then Ken was illustrated in more detail. A ken_handler() function gets executed in a similar way to a main() function. Transaction semantics are given within the function. Calling ken_malloc()/ken_set_app_data()/ken_get_app_data() allows use of the persistent heap, while ken_send() provides "fire and forget" messages. Ken can be used in Mace without any changes. Ken provides global masking of failures, and composable reliability, while Mace provides distributed protocols, availability, replication, and handling of permanent failures.

The evaluation consists of micro-benchmarks and an implementation of Bamboo-DHT on 12 machines. The micro-benchmarks show that latency and throughput of Ken depend on the underlying storage type (disk, no sync, and ramfs). The Bamboo-DHT results show MaceKen has 100% data resiliency under correlated failures and rolling restarts, which can happen in managed networks.

Ken and MaceKen are available online: http://ai.eecs.umich.edu/~tpkelly/Ken and http://www.macesystems.org/maceken.

Todd Tannenbaum, University of Wisconsin-Madison, asked if Ken messages have leases. He said that when resends are hidden, applications may not want to wait forever. For example, a seller may not want to wait forever for payment. Sunghwan answered that each event works as a transaction, so events would not lead to incorrect states. Todd again asked how this applies to waiting forever. Sunghwan said that Ken provides fault tolerance for crash-restart failures. Timeouts could be implemented at a higher layer. Someone asked whether Ken can roll back or cancel a transaction. Sunghwan answered that Ken can recover to the latest checkpoint.

Multicore

Summarized by Wonho Kim (wonhokim@cs.princeton.edu)

Managing Large Graphs on Multi-Cores with Graph Awareness

Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan, Microsoft Research

Vijayan Prabhakaran from MSR presented Grace, an in-memory transactional graph management systems, which can efficiently process large-scale graph-structured data by utilizing multicores in machines.

To exploit multicore parallelism, Grace partitions a given graph into smaller subgraphs that can be processed by each core separately, combines the results at a synchronization barrier, and continues the iteration. Vijayan mentioned that many graph algorithms, such as Google's page-rank, will work in this manner. In addition to the graph-specific optimizations, another interesting feature of Grace is supporting transactions by creating read-only snapshots.

In the evaluation, he compared the performances of different graph partitions. As expected, careful vertex partition leads to better performance than random algorithm. However, it was interesting that the vertex partitions do not make a difference when the number of partitions is low because (1) the partitions fit within a single chip and (2) the communication cost between partitions is very low in this case. Rearranging vertexes also improves performance by exploiting vertex locality in each partition. However, dynamic load-balancing among partitions does not improve overall performance.

Alexandra Fedorova, Simon Fraser University, asked about creating well-balanced partitions (static) and load balancing (dynamic). Grace adjusts the vertexes among the graph partitions at runtime to improve overall completion time. Vijayan answered that dynamic load-balancing is still needed because processing time in each partition is affected by multiple factors depending on the algorithms used.

MemProf: A Memory Profiler for NUMA Multicore Systems

Renaud Lachaize, UJF; Baptiste Lepers, CNRS; Vivien Quéma, GrenobleINP

Baptiste Lepers from CNRS presented MemProf, a memory profiler for NUMA systems that enables application-specific memory optimizations by pointing out the causes of remote memory accesses.

In NUMA systems, remote memory accesses have lower bandwidth and higher latency than accesses within the same node. The talk started with showing that many existing systems suffer from inefficient remote memory accesses in their default settings, and that NUMA optimizations can significantly improve their performance. However, existing profiles do not point out the causes of remote accesses, which is needed for making optimization decisions.

MemProf provides information about thread-object interactions in a given program from the viewpoints of both objects and threads. This output is useful for identifying what kinds of memory optimizations are needed. An interesting example presented in the talk is that the authors significantly improved the performance of FaceRec (face-recognition program) by more than 40% simply by replicating a matrix that is remotely accessed. MemProf tracks the object/thread life cycle using kernel hooks, but the overhead is quite low (5%).

Haibo Chen asked if replicating memory objects could increase cache misses. Baptiste answered that such an effect was not visible in the experiments and that the replication helps reduce remote accesses in a program. Baptiste also mentioned that, from MemProf output, users can detect different latencies among multiple nodes in NUMA systems. A follow-up question was about how MemProf can replicate memory objects automatically. Baptiste said that MemProf users should know the memory usage in the program because replication is possible only when memory is not fully utilized. He also mentioned that it would be difficult to optimize a program if the program exhibited different memory access patterns across executions. He also explained memory access patterns in Apache.

Remote Core Locking: Migrating Critical-Section Execution to Improve the Performance of Multithreaded Applications

Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller, LIP6/INRIA

Jean-Pierre Lozi started by showing that memcached performance collapses in manycore systems because lock acquisition time in critical sections increases as more cores are used. To address the lock contention cost, Remote Core Locking (RCL) executes highly contended critical sections in a dedicated server core, which removes atomic instructions and reduces cache misses for accessing shared resources. RCL requires dedicated cores, so it profiles applications to find candidate locks for RCL.

In micro-benchmarks, RCL shows much lower execution time compared to spin-lock, POSIX, and MCS. It was interesting to see that RCL improves the performance even in low-contention settings because execution in a dedicated core improves locality. RCL also significantly improves the performance of existing systems including memcached.

John Griffin from Telecommunication Systems asked what was the CPU utilization in the server cores during benchmarks. Jean-Pierre answered that the server cores are never idle; they always check pending critical sections to execute. Xiang Song from Shanghai University asked how RCL handles nested locks. RCL puts them in the same server core.

Packet Processing

Summarized by Wonho Kim (wonhokim@cs.princeton.edu)

The Click2NetFPGA Toolchain

Teemu Rinta-aho and Mika Karlstedt, NomadicLab, Ericsson Research; Madhav P. Desai, Indian Institute of Technology (Bombay)

Teemu Rinta-aho presented Click2NetFPGA, a compiler toolchain that automatically transforms software (in C++) to functional target hardware design (in NetFPGA).

Although many HLS tools are available, they are not made for people who do not understand hardware. Click2NetFPGA does not require knowledge in target hardware systems, and converts a given Click module to NetFPGA design. The talk mainly focused on the prototype implementation. In Click2NetFPGA, Click modules and configurations are first compiled into LLVM IR, and transformed to VHDL (VHSIC hardware description language) modules using AHIR compile developed from IIT Bombay.

The measurement results showed that Click2NetFPGA can reach only 1/3 of the line speed (1 Gbps) because of the inefficient translation between NetFPGA and Click data models. The presenter introduced their ongoing work on improving the performance of resulting hardware.

In the Q&A session, there was a question about how fast the compiled NetFPGA module is compared to the original Click software. Teemu answered that it could easily get 1 Gbps on a standard PC. Eddie Kohler from Harvard University asked what mistakes would be made if people work on similar projects. Teemu said “ carefully study the source systems,” which was an interesting answer because Eddie Kohler was the person who wrote the source system, the Click Modular Router.

Building a Power-Proportional Software Router

Luca Niccolini, University of Pisa; Gianluca Iannaccone, RedBow Labs; Sylvia Ratnasamy, University of California, Berkeley; Jaideep Chandrashekar, Technicolor Labs; Luigi Rizzo, University of Pisa and University of California, Berkeley

Luca Niccolini presented a software router that achieves energy efficiency by consuming power in proportion to incoming rates with a modest increase in latency.

While network devices are typically underutilized, the devices are provisioned for peak load. However, the devices are power-inefficient and consume 80–90% of maximum power, even with no traffic. Luca showed that CPU is the biggest power consumer in software routers. The authors developed an x86-based software router that adjusts the number of active cores and operating frequency based on incoming rate to improve energy efficiency. The design of the power control algorithm is guided by measurement of power consumption in different settings. It was interesting to see that running a smaller number of cores at higher frequency is more energy-efficient than running more cores at lower frequency.

In the evaluation, Luca showed that the new router consumes power in proportion to the input workload when running IPv4 routing, IPSec, and WAN optimization, saving 50% power. The tradeoff is latency, but it is a modest increase (10 µs). Another promising result was that the router did not incur packet loss or reordering in the experiments.

Someone asked if manipulating packet forwarding tables can overload some other cores. Luca answered that it is possible but the controller could detect such an event and change configuration. Luca also pointed out that reordering did not occur, because queue wakeup latency prevented packets in an empty queue from forwarding earlier than the other packets. Herbert Bos from Vrije University asked about an alternative approach, running different applications to different cores at different frequencies. This was not considered in the work, however.

netmap: A Novel Framework for Fast Packet I/O

Luigi Rizzo, Università di Pisa, Italy

► *Awarded Best Paper!*

Luigi Rizzo explored several options for direct packet I/O such as socket, memory-mapped buffers, running within the kernel, and custom libraries. But these all have issues with performance, safety, and flexibility. From measurement of packet processing time in different levels, Luigi showed that the three main costs come from dynamic memory allocation, system calls, and memory copies. netmap uses preallocated and shared buffers to reduce the cost.

netmap can transmit at line rate on 10 Gbps interfaces while receive throughput is sensitive to packet size because of hardware limitations in the system (e.g., cache line). netmap also improved the forwarding performance of Openvswitch and Click by modifying them to use netmap. It was interesting to see that netmap-Click in userspace can outperform the original Click running in the kernel.

Monia Ghobadi from the University of Toronto asked about inter-arrival times of back-to-back packets. Luigi said that packets were generated with no specified rate in the experiments.

Toward Efficient Querying of Compressed Network Payloads

Teryl Taylor, UNC Chapel Hill; Scott E. Coull, RedJack; Fabian Monrose, UNC Chapel Hill; John McHugh, RedJack

Teryl Taylor from UNC Chapel Hill presented an interactive query system, which can be used for forensic analysis. It is challenging to build an interactive query system for network traffic because network traffic typically has extremely large volumes, multiple attributes, and heterogeneous payloads. Teryl presented a solution which was to build a low I/O bandwidth storage and query framework by reducing, indexing, partitioning data and allowing application-specific data schemas.

In the evaluation, the authors used two data sets: campus DNS data and campus DNS/HTTP. The query system significantly reduced query processing time to sub-minute compared to PostgreSQL and SiLK for different query types (heavy hitters, partition intensive, and needle in a haystack).

There was a question about configuring on the fly what to store about the payload. Teryl answered that it is possible to create/install different versions of payloads. Keith Winstein from MIT asked how difficult it is to write a program that finds interesting patterns about a given suspect trace. Teryl said that using the interactive query system makes a huge difference in finding traffic patterns.

Plenary

Summarized by Rik Farrow (rik@usenix.org)

Build a Linux-Based Mobile Robotics Platform (for Less than $500)

Mark Woodward, Actifio

Mark Woodward told us that he has worked for robotics companies for many years, starting with Denning Mobile Robotics in 1985. But by then, he had already built his own robot, based on a Milton Bradley Bigtrak chassis, a programmable tank from 1979. The Bigtrak had a simple Texas Instruments microcontroller, and Mark used this to lead into a discussion of CPUs that appeared in later robots, such as Motorola 68K and Z80 CPUs, as well as sensors.

Mark wasn't very excited about the state of commercial robotics. He called the Roomba a "Bigtrak with a broom," the Segway as a great example of process control, and self-parking cars as something that sometimes works. He described a project he had worked on while at Denning: a robotic security guard. When they tried to sell their 1985 $400,000 robot, they discovered that watchguard companies preferred to hire guards for a lot less. The robot itself was so expensive, thieves might elect to steal it and ignore what the robot was guarding.

Mark had brought his own robot with him, and he explained the technology he used to build it. For example, it uses a smaller form factor motherboard (ITX) for general processing, for connection to video cameras, and for running text-to-speech processing, so the robot can talk via speakers connected to the motherboard. While the motherboard runs Linux, Mark prefers to use an Arduino for sensors and for motor control. He explained that the motor control was actually very difficult, as simply measuring how much each wheel turns doesn't actually reflect the movement of the robot, as wheels can (and do) slip on many surfaces. The motor control uses a Proportional-Integral-Derivative (PID) algorithm, a commonly used feedback controller.

Mark then provided a list of tools useful for building robots and other hardware products: temperature controlled soldering iron, oscilloscope (Rigil), benchtop power supplies, as well as more mundane items like lawnmower wheels, duct tape, tap and dies, wire ties, and PVC pipe. He also recommended the book *The Art of Electronics* (Paul Horowitz, Winfield Hill), but Clem Cole (Intel) countered that *Practical Electronics for Inventors* (Paul Scherz) is a better and more recent book.

Bill Cheswick (Independent) asked how Mark dealt with surface mounts, and Mark answered that he didn't use them. Clem mentioned that there are workarounds for flowing solder to attach surface mounts. Steve Byar (NetApp) asked about power supplies, and Mark suggested contacting him later. Rik Farrow asked whether he had considered using an optical mouse as a sensor to gain movement information, and Mark said he hadn't, but didn't think it would work. Clem Cole asked about using stepper motors, and Mark described them as "evil," requiring a separate input for each step. Marc Chiarini (Harvard SEAS) asked about making robots like Mark's smaller. Mark pointed out that his robot had an extra large, Plexiglas top that he used for scaffolding, to hold things like speakers and video cameras which could be removed. Marc then asked about the size of the wheels. Mark replied that he is using plastic gearing, so the wheels need to have a large diameter. Ches asked what tools did Mark wish he'd had when he started. Mark said an oscilloscope.

Security

Summarized by Tunji Ruwase (oor@cs.cmu.edu)

Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation

Asia Slowinska, Vrije Universiteit Amsterdam; Traian Stancescu, Google, Inc.; Herbert Bos, Vrije Universiteit Amsterdam

Asia Slowinska presented a tool called BinArmor, that hardens C binaries, even without symbol information, against buffer overflow attacks against both control-data and non control-data. The work was prompted by statistics that show that despite its buffer overflow vulnerabilities, C still remains the most popular programming language. Moreover, current techniques are ineffective for protecting binaries (e.g., legacy code) against buffer overflow attacks. By detecting non-control data attacks, BinArmor provides better protection than taint analysis, which only detects control data attacks. However, BinArmor is prone to false negatives due to its reliance on profiling (as discussed later); i.e., it can miss real attacks.

To harden a program binary against attacks, BinArmor (1) finds the arrays in the program, (2) finds the array accesses, and (3) rewrites the binary, with a novel color tracking code, for buffer overflow detection. The Howard reverse engineering tool (presented at NDSS 2011) is used to detect arrays in binaries without symbol information. Next, profiling runs of the program are used to detect accesses to the detected arrays. Coverage issues of profiling lead to the false negatives in BinArmor. The binary rewrite step assigns matching colors to each pointer and the buffer it references, tracks color propagation, and checks that the color of de-referenced pointers matches the referenced buffer. Protecting fields (and subfields) of C structs requires a more complex coloring scheme, where fields have multiple colors, to permit the field to also be accessed through pointers to the enclosing structs.

Asia then presented the evaluation of BinArmor, which focused on bug detection effectiveness and performance. BinArmor detected buffer overflows in real world applications, including a previously unknown overflow in the htget program. Also, it introduced, at most, a 2x slowdown in real world I/O-intensive programs. The nbench benchmark suite, which is more compute intensive, had a worst case slowdown of 5x, with a 2.7x average slowdown.

A lively question/answer session ensued, with a session-leading number (six) of questioners. Bill Cheswick set the ball rolling by asking if BinArmor detected new bugs; Asia referred to the htget overflow. Andreas Haeberlen from University of Pennsylvania asked how an attacker could adapt to BinArmor. Asia pointed out that the coverage issues of the profiling step could be exploited. Larry Stewart asked how pointers used by memcpy (and other libc functions) were handled by BinArmor, since these pointers travel through many software layers. Asia responded that more pointer tracking would be required for that. Julia Lawall asked if BinArmor currently performed any optimizations, and suggested bounds-checking optimizations in Java. Asia responded that optimizations were future work. Konstantin Serebryany from Google asked if Body Amour reported errors for libc functions that read a few bytes beyond the buffer. Asia clarified that this was not a problem in practice, because the granularity of colors in BinArmor is 4 bytes. Steffen Plotner of Amherst College asked if BinArmor could be used to protect the Linux kernel. Asia responded that they had not tried.

Abstractions for Usable Information Flow Control in Aeolus

Winnie Cheng, IBM Research; Dan R.K. Ports and David Schultz, MIT CSAIL; Victoria Popic, Stanford; Aaron Blankstein, Princeton; James Cowling and Dorothy Curtis, MIT CSAIL; Liuba Shrira, Brandeis; Barbara Liskov, MIT CSAIL

Confidential data, such as credit card information, and medical records, are increasingly stored online. Unfortunately, distributed applications that manage such data, are often vulnerable to security attacks, resulting in high profile data theft. Dan Ports introduced the Aeolus security model, which uses decentralized information flow control (DIFC), to secure distributed applications against data leaks. Dan observed that access control was not flexible enough for this purpose, because the objective is to restrict the use, not the access, of information. Aeolus describes a graph-based security model and programming abstractions for building secure distributed applications.

In summary, Aeolus tracks information flow within a protection boundary to ensure that only declassified information flows outside the boundary. Aeolus achieves this using three concepts: *principals* (entities with security concerns, e.g., individuals), *tags* (the security requirements of data), and *labels* (set of tags). Labels associated with data objects are immutable, while threads are associated with principals and mutable labels (reflecting accessed data). Aeolus also maintains an authority graph, to ensure that declassification (tag removal) is done by authorized principals. Dan further discussed Aeolus programming abstractions and Java implementation.

Evaluations using micro-benchmarks showed that most Aeolus operations are within an order of magnitude of Java method calls. Moreover, Aeolus imposed a mere 0.15% overhead on a financial management service application. The low overhead is because the Aeolus operations are infrequent and relatively inexpensive. Aeolus is available at http://pmg.csail.mit.edu/aeolus.

Someone expressed concern about malicious information flowing into the system. Dan confirmed that Aeolus in fact tracks the integrity of incoming information, and referred the audience to the paper for details. Rik Farrow also expressed a concern that conventional uses of authority graphs, e.g., in banks, often suffered from untimely updates. Dan observed that untimely updates were due to centralized control, thus decentralization in Aeolus helped to avoid the problem.

TreeHouse: JavaScript Sandboxes to Help Web Developers Help Themselves

Lon Ingram, The University of Texas at Austin and Waterfall Mobile; Michael Walfish, The University of Texas at Austin

Third-party code is extensively used by JavaScript applications and, allowed to execute with similar privileges, is therefore trusted to be safe/correct. Lon Ingram demonstrated this was misplaced trust. For example, using a third-party widget that had a hyperlink vulnerability for processing online payments, he showed how this vulnerability could be exploited by an attacker to steal credit card information. He then presented TreeHouse, a system that uses sandboxing to enable safe use of third-party code in JavaScript applications.

TreeHouse is implemented in JavaScript, and is therefore immediately deployable, as no browser changes are required. Moreover, it modifies the Web Worker feature of modern browsers to act as containers for running third-party code. By transparently interposing on privileged operations, TreeHouse enables flexible control of third-party code. Lon then showed how an application can use TreeHouse to implement the required security policies for thwarting the attack in the motivating example.

Experimental results showed that Document Object Model (DOM) use significantly affected TreeHouse overheads. In particular, DOM access can be up to 120k times slower with TreeHouse. Also, TreeHouse increases initial page load latency by 132–350 ms, on average. Consequently, TreeHouse is not suitable for DOM-bound applications or applications with a tight load time. Further information about TreeHouse is available at github.com/lawnsea/Treehouse and lawnsea@gmail.com.

James Mickens from MSR asked whether the prototype chain needed to be protected. Lon said that he would have to think about it. Konstantin Serebryany from Google asked what JavaScript feature would Lon like to change. Lon responded that he would like parent code to run child code with restricted global symbol access. Steve McCaant from UC Berkeley highlighted a regular expression typo in the slide, which Lon acknowledged.

Cloud Terminal: Secure Access to Sensitive Applications from Untrusted Systems

Lorenzo Martignoni, University of California, Berkeley; Pongsin Poosankam, University of California, Berkeley, and Carnegie Mellon University; Matei Zaharia, University of California, Berkeley; Jun Han, Carnegie Mellon University; Stephen McCamant, Dawn Song, and Vern Paxson, University of California, Berkeley; Adrian Perrig, Carnegie Mellon University; Scott Shenker and Ion Stoica, University of California, Berkeley

Stephen McCamant presented Cloud Terminal, a system for protecting sensitive information on PCs. Cloud Terminal assumes that the vulnerabilities in client software stack, including the OS, can compromise the confidentiality and integrity guarantees offered by prior techniques. Therefore, Cloud Terminal proposes a new software architecture for secure applications running on untrusted PCs, with a Secure Thin Terminal (STT) running on client systems, and remote applications in a Cloud Rendering Engine (CRE) VM. As an example, Stephen demonstrated how Cloud Terminal allows a PC user to perform secure online banking without any dependence on the untrusted OS.

On the client system, STT's role is to render graphical data from the remote application, and forward keyboard and mouse events to it. A simple hypervisor, called Microvisor, leverages Flicker and Intel TXT to isolate STT from the client OS. STT was implemented in 21.9 KLOC. CRE runs the remote application in a VM, and connects to STT via a lightweight remote frame buffer VNC protocol with SSL security. CRE incorporates a number of techniques to provide scalability (support for 100s of application VMs) and security.

Cloud Terminal was evaluated with CRE running on a 2 GHz, 16-core system, with 64 GB RAM, while STT ran on a Lenovo W510 laptop. The evaluated applications were AbiWord, Evince, Wells Fargo online banking on Firefox, and Gmail on Firefox. The applications were found to be quite usable, with reasonable display and latency. However, page scrolling was sluggish, but Stephen said that this could be optimized. In terms of cost, Cloud Terminal could provide secure computing services at 5 cents per user per month.

Andreas Haeberlen, University of Pennsylvania, asked if client-side resources could be used to improve performance. Stephen replied that, while this was possible, it would not match the target applications. Someone asked if more client-side devices could be supported in STT. Stephen said supporting the drivers would increase complexity and thus undermine trustworthiness.

Short Papers: Tools and Networking

Summarized by Rik Farrow (rik@usenix.org)

Mosh: An Interactive Remote Shell for Mobile Clients

Keith Winstein and Hari Balakrishnan, MIT Computer Science and Artificial Intelligence Laboratory

Keith Winstein gave a lively talk about a mobile shell, Mosh. Keith began by saying that everyone uses SSH, but SSH uses the wrong abstraction: an octet stream. What you want when you use SSH is the most recent state of your screen. He joked that today's network is not like the ARPANET, which was much faster. The authors developed SSP, the state synchronization protocol, which communicates the differences between the server's concept of a screen and the screen at the client side. Mosh also displays keystrokes, as well as backspace and line kill, immediately, on the user's terminal, underlining characters until the server confirms any local updates.

Mosh still uses SSH to authenticate and start up a mosh_server. When mosh_server starts up, it communicates an AES key over SSH before shutting down that connection. Mosh_client uses that key in aes-ocb mode, which supplies both encryption and an authenticated stream. Neither the mosh_server or client run with privileges. Mosh uses UDP packets, which means that there is no TCP connection to maintain. Using UDP with AES-OCB (AES Offset Codebook mode) is what allows the Mosh user to roam. Mosh also manages its own flow control that adapts to network conditions.

Keith finished with a demo comparing SSH and Mosh. When the IP address changes, SSH doesn't even tell us that the connection is dead, Keith said, and that is "most offensive."

Lois Bennett asked about configuring a firewall to allow Mosh, and Keith replied that you need to keep a range of UDP ports open, depending on how many simultaneous Mosh sessions you expect. He also said they are working to make Mosh more firewall friendly. Someone else wondered how Mosh could behave predictively with Gmail, and Keith responded that Gmail is actually easier to handle than terminal applications like Emacs.

The August 2012 issue of *;login:* includes an article about Mosh.

TROPIC: Transactional Resource Orchestration Platform in the Cloud

Changbin Liu, University of Pennsylvania; Yun Mao, Xu Chen, and Mary F. Fernández, AT&T Labs—Research; Boon Thau Loo, University of Pennsylvania; Jacobus E. Van der Merwe, AT&T Labs—Research

Changbin Liu described a problem with how IaaS cloud providers provision services: if one link in a chain of events fails, the entire transaction fails. For example, starting a server requires acquiring an IP address, cloning the OS image within storage, creating the configuration, and starting the VM. The key idea behind TROPIC is that it orchestrates transactions with ACID for robustness, durability, and safety. TROPIC has a logical layer with a replicated datastore that communicates with the physical data model. TROPIC runs multiple controllers with a leader and followers. If a step fails, TROPIC rolls back to the previous stage, and continues with the failure hidden from the user. TROPIC also performs logical layer simulations to check for constraint violations—for example, allocating more memory than the VM host has, or using the next hop router as a backup router, the very problem that caused the failure of EC2 in April 2011.

They have an 11k LOC Python implementation which they have tested on a mini-Amazon setup deployed on 18 hosts in three datacenters. The code is open source, and will be integrated into Open Stack.

Haibo Chen (Shanghai Jiao Tong University) asked how they can tell the difference between a true error and excess latency. Changin Liu replied that error detection is via error message. If the connection hangs for a minute, TROPIC kills the connection or terminates it. There is more about error handling in the paper.

Trickle: Rate Limiting YouTube Video Streaming

Monia Ghobadi, University of Toronto; Yuchung Cheng, Ankur Jain, and Matt Mathis, Google

Monia Ghobadi explained that the way videos are streamed by YouTube and Netflix results in bursts of TCP traffic. The bursty nature of this traffic causes packet losses and affects router queues. YouTube writes the first 30–40 seconds of a video, followed by 64 KB blocks using a token bucket to establish a schedule. Netflix sends out 2-MB bursts, also causing periodic spikes. Trickle uses the congestion window to rate limit TCP on the server side. Trickle requires changes to the server application. Linux already allows setting a per-route option called cwnd_clamp, and they wrote a small kernel patch to make this option available for each socket.

Monia compared the current YouTube server, ustreamer, with Trickle, using data collected over a 15-day period in four experiments in Europe and India. Trickle reduced retransmissions by 43%. Sending data more slowly also affects queueing delay, with roundtrip times (RTTs) lower than ustreamer by 28%. She then demonstrated a side-by-side comparison of ustreamer and Trickle (http://www.cs.toronto.edu/~monia/tcptrickle.html) by downloading movie trailers. In the demo, Trickle actually worked faster, slowly moving ahead of the display in the ustreamer window because of ustreamer packet losses.

Someone from Stanford asked if the connection goes back to slow start when the connection is idle. Monia answered that since they are using the same connection, the congestion window clamp still exists. John Griffinwood (Telecom Communications) wondered whether they saw jitter and whether Google had adopted Trickle. Monia answered that Trickle dynamically sets the upped bound and readjusts the clamp if congestion is encountered. While she was working as an intern for Google, they had planned to implement Trickle. Someone from AT&T asked whether mobile users also benefit from this. Monia answered yes.

Tolerating Overload Attacks Against Packet Capturing Systems

Antonis Papadogiannakis, FORTH-ICS; Michalis Polychronakis, Columbia University; Evangelos P. Markatos, FORTH-ICS

Antonis Papadogiannakis told us that when a packet capture system gets overloaded, it randomly drops packets. When a system is being used for intrusion detection, random drops are not good, as the dropped packets may be important. An attacker could even cause the overload by sending packets that result in orders of magnitude slower processing, or using a simpler but more direct DoS attack. Antonis pointed out that existing solutions include over-provisioning, thresholds, algorithmic solutions, selective discarding, and ones that attempt to reduce the difference between average and worst case performance.

Their solution is to store packets until they can be processed. Excess packets are buffered to secondary storage if they don't fit in memory, so all packets will be analyzed. When the ring buffer gets full, packets are written to disk. When the ring buffer has space again, packets are read back and processed. If the system running packet capture is also relaying packets, this will result in additional latency. But this may not be an unreasonable price to pay if you are relying on this system to block attacks.

The limitation to their approach are the delays when relaying and the practical limitation of buffering packets to disk. They tested their implementation using a modified version of libpcap evaluated with Snort, using an algorithmic complexity attack which resulted in an unmodified system losing as much as 80% of packets at one million packets per second. Their system did not lose any packets at this rate. There were no questions.

Enforcing Murphy's Law for Advance Identification of Run-Time Failures

Zach Miller, Todd Tannenbaum, and Ben Liblit, University of Wisconsin—Madison

Zach Miller explained that Murphy causes "bad things" to happen to the software under test. Using ptrace, Murphy captures all system calls and modifies the returned results. Murphy follows POSIX behavior when generating responses, so the results should not be that far afield from things that an application might be expected to handle properly, such as a failed write() system call because of a disk full error. Murphy works with any language, is done in user space, and tests entire software stacks, since it interposes on system calls going to the kernel.

Murphy found a bug in /bin/true, because the command expects read() to succeed. Murphy includes rich constraints, such as regex matching, state, mapping file descriptors to filenames, and other tricks. Murphy can simulate full disks, time going backwards, and other results that are allowed by system calls. Murphy keeps a log of all changes it made, and this log can be replayed to test fixed code. Murphy can also skip through the replay log and suspend the application right before the return result that caused a crash.

They found bugs in C, Perl, Python, and OpenSSL in their testing. At this point, Murphy only works under 64-bit Linux.

Eddie Kohler (Harvard) wondered, if they find a bug under Linux, is it a true bug in other environments? Zach said that because Linux is a POSIX-compliant system, bugs found there will be true for any POSIX-compliant software. Alexander Potapenko (Google) asked about the performance overhead. Zach responded that it varied based on the amount of system calls made by the application under test. It might be as little as six times slower, and as much as 60 times.

Distributed Systems

Summarized by Brian Cho (bcho2@illinois.edu)

A Scalable Server for 3D Metaverses

Ewen Cheslack-Postava, Tahir Azim, Behram F.T. Mistree, and Daniel Reiter Horn, Stanford University; Jeff Terrace, Princeton University; Philip Levis, Stanford University; Michael J. Freedman, Princeton University

Ewen Cheslack-Postava explained that a Metaverse is a 3D space, where everything in the space is editable by users. There are a wide variety of applications, including games, augmented reality, etc. Unfortunately, what you get today is not as pretty as artist renderings. Examples of artist renderings were shown, followed by a very spare screen from Second Life. The reason Second Life looked so spare was because the system won't display things more than a few meters away. A second screen, shown after the user moved a few steps shows a much richer world. The problem is that the system doesn't know how to scale, while not sacrificing user experience.

These are systems problems. The only way currently to scale is to carve the world geographically into separate servers, and limit each server to communication with a few neighboring servers. This work uses the insight that the real world scales, and scales by applying real-world constraints to the system. Because there is a limited display resolution, they use a technique called solid-angle queries. The solid angle dictates how large an object appears, and anything with a large solid angle should show up. So, for example, mountains should show up, even if they are far away. The second thing done is to combine objects. The combination of both solid-angle queries and aggregates is close to ideal.

These techniques are used through a core data structure called Largest Bounding Volume Hierarchy (LBVH) tree structure, which modifies the Bounding Volume Hierarchy (BVH) tree. An example of four objects, in a three-level hierarchy was shown. BVH uses spheres that can contain objects, and hierarchically combines neighboring spheres into ever-larger spheres. The problem with this structure, is that to find large objects to display, a long recursive search has to be done, and because the spheres overestimate size, it's hard to prune parts of the search. LBVH instead stores the largest object in a subtree at interior nodes. Doing this results in 75–90% fewer nodes tested. Other techniques are also presented, showing how to deal effectively with moving objects, and redundant queries. Aggregation is applied by storing an aggregated object of lower quality on each internal node (BVH only stores objects at the nodes). Queries on LBVH across different servers are done efficiently by running large queries across machines, and then filtering those for each individual query.

An example application, Wiki World, was shown. You can automatically find info about objects on Wikipedia. This would not be possible in other systems. Many more systems challenges at the intersection of systems, graphics, PL, databases, etc. are present in this area. An example is audio: for instance, playing a distant siren or the combined roar of a crowd. More info can be found at http://sirikata.com.

Jon Howell, Microsoft Research, asked what workload was used to measure the improvements. Ewen said it is hard to collect or generate workloads. What they used were a synthetic random workload, and a workload collected from Second Life. For their experiments, they tried both workloads. Chip Killan, Purdue, asked how direct communication is done with aggregate objects. Ewen said that you can't do this with aggregate objects currently, which is a limitation in the current system.

Granola: Low-Overhead Distributed Transaction Coordination

James Cowling and Barbara Liskov, MIT CSAIL

James Cowling told us that Granola is an infrastructure for building distributed storage applications. It provides strong consistency without locking for multiple repositories and clients. The unit for an atomic operation chosen is transactions. Why? Because using transactions allows concurrency on a single repository to be ignored. Transactions are allowed to span multiple repositories, avoiding inconsistency between repositories. However, distributed transactions are hard. Opting for consistency, e.g., using two-phase commit, results in a high transaction cost. Opting for performance, e.g., providing a weak consistency model, places the burden of consistency on application developers, which evidence suggests makes their job difficult.

To allow strong consistency and high performance, for at least a large class of transactions, this work provides a new transaction model. There are three classes of operations—first, those that work on a single repository, and then, for distributed operations, coordinated and independent transactions. Granola specifically optimizes for single and distributed independent operations; it provides one-round transactions. An example of a distributed independent operation was shown: consider transferring $50 between two accounts. Each participant must make the same commit/abort decision. Evidence shows this class of operations is common in OLTP workloads. For example, TPC-C can be expressed entirely using single or independent transactions.

Granola provides both a client library and a repository library, and sits between the clients and repositories. Each repository is in fact a replicated state machine. There are two modes for a repository—it will primarily be in timestamp mode, and occasionally switch to locking mode, when coordinated transactions are required. Each transaction is assigned a timestamp, and transactions are executed in timestamp order. Thus, timestamps define a global order. The challenge is how to assign timestamps in a scalable, fault-tolerant way.

For a single repository transaction, the steps of operation are: (1) the repository assigns a timestamp, chosen to be higher than any previous timestamps; (2) the transaction with the timestamp is logged; and (3) the transaction is executed. For distributed, independent transactions, repositories additionally vote to determine the highest timestamp. The steps are (1) propose, (2) log, (3) vote, (4) pick, and (5) run. The transaction will not execute until it has the lowest timestamp of all concurrent transactions. This guarantees a global serialized execution order. Granola provides interoperability with coordinated transactions, by requiring repositories to switch to lock mode. Locking is required to ensure a vote is not invalidated. The protocol is changed to include a preparation phase, and the transaction is aborted if there is a conflict. The repository can commit transactions out of timestamp order. The result will still match the serialized order, even if execution happens out of timestamp order (because of the nature of transactions). Repositories throughout the system can be in different modes.

Experiments were presented using the TPC-C benchmark. Granola scales well. With a higher load of distributed transactions, Granola throughput only goes down to half. This is because there is no locking or undo logging.

Marcos Aguilera, Microsoft Research, commented about the ambiguity of the terminology for consistency, that it could mean either serializability or atomicity. James agreed that database and system communities use different terminology. Marcos then asked if a change doesn't touch the entire repository, if there is a need to switch the entire repository to lock mode. James answered that if there were a separate object model, this would be possible, but in the system the application is just considered a blob, so it is not possible currently.

Timothy Zhu, CMU, asked for suggestions on when to use this system. Is it applicable all the time? James said there are obvious limitations; when there are failures, you have to switch into locking mode, so when you really only want availability, this isn't a great system. Timothy asked if the timestamps are similar to Lamport clocks. James answered that they are basically Lamport clocks, except that voting does not take place in Lamport clocks. Also, Granola in fact makes use of local system clocks at clients for performance.

Zhiwu Xie, Virginia Tech, asked James to compare Granola with the Calvin system. James answered that Calvin has an agreement layer that needs all-to-all communication, so they have higher latency. He believes there is a potential scalability limit because of this, but they showed 100 nodes, which is impressive. Calvin's advantage is that it has more freedom to shift transactions around. Granola is constrained, so it relies on single-threaded execution.

High-Performance Vehicular Connectivity with Opportunistic Erasure Coding

Ratul Mahajan, Jitendra Padhye, Sharad Agarwal, and Brian Zill, Microsoft Research

Ratul Mahajan started by asking how many of the audience have used Internet access on-board a vehicle. There was quite a show of hands. Riders love Internet access—it boosts ridership. But performance can be poor, and service providers don't have a good grasp on how to improve it. A service provider's support suggested, for example, that the user cancel a slow download and retry in approximately five minutes.

Vehicular connectivity uses WWAN links. It's not the WiFi that is bad, but rather that the WWAN connectivity is lossy. This is not due to congestion but is just how wireless behaves. Two methods to mask losses are retransmission and erasure coding (EC). Retransmissions are not suitable for high delay paths. So high-delay should use erasure coding. Existing EC methods are capacity-oblivious, meaning there is a fixed amount of redundancy. The problem is that this fixed amount may be too little or too much, relative to the available capacity. Thus, the main proposal is opportunistic erasure coding (OEC)—this uses spare capacity. The challenge is how to adapt given highly bursty traffic. Real data from MS commuter buses shows that you would have to adapt at very small time-scales.

The transmission strategy for OEC is to send EC packets if and only if the bottleneck queue is empty. This matches "instantaneous" spare capacity and produces no delay for data packets. As for the encoding strategy, conventional codes are not appropriate. These codes don't provide graceful degradation when the amount of redundancy provided is different from that needed. Thus, OEC is designed with greedy encoding. The strategy is that, if the receiver has a lot of packets, then EC has a lot of packets. A good property that is achieved is that each packet transmission greedily maximizes goodput.

PluriBus is OEC applied to moving vehicles. OEC happens between the VanProxy (on the moving bus) and LanProxy (part of the immobile infrastructure). Details of how relevant parameters are estimated were given. Ratul claimed that the aggressive use of spare capacity is not such a bad idea. The observation is that the network is not busy all the time using timeouts, and this means that network traffic only increases by a factor of two. Media access protocols isolate users from each other, so this won't hurt innocent users.

Evaluation was done through a deployment on two buses on the MS campus, with trace-driven workloads, and emulation. The main result is that performance is improved by 4x. The workload was scaled, up to a factor of 8x, to show that losing spare capacity is not a major concern. In emulation, it was shown that OEC outperforms other loss recovery methods. This is because retransmission requires delay, and fixed redundancy ECs are not opportunistic.

Philip Levis, Stanford, asked whether fountain codes could be used instead. Ratul replied that a challenge in PluriBus is that r is dynamic, in addition to being estimated. With fountain codes, k of n packets must arrive, which could not be guaranteed.

Bradley Andrews, Google, asked whether any actual user feedback was collected. Ratul answered there were two main reasons that they did not. First, outsourcers who ran the actual commute buses didn't allow changes, so this couldn't be applied to those buses. Second, during the study, a large shift to smartphones meant that the demand for Internet access on these buses essentially disappeared. Bradley then asked whether there was collaboration with wireless carriers. Ratul explained that permission was not asked of wireless carriers before the study, but once the study was over, the results were shared with carriers.

Masoud Jafavi, USC, asked what the effect of a crowded area would be. Ratul replied that experiments were not done to quantify this, but the feeling is that any kind of damage will not be too large. Rather, the important questions to consider are: Do you or do you not have a dedicated channel to the cell provider? And how many users can get a channel, and how quickly? Ratul commented that PluriBus may hold on to the channel for about 200 ms longer, but compared to the release timeout of five seconds, this is a small fraction of the overall time.

Server-Assisted Latency Management for Wide-Area Distributed Systems

Wonho Kim, Princeton University; KyoungSoo Park, KAIST; Vivek S. Pai, Princeton University

Wonho Kim presented this work on one-to-many file transfer. This may sound like an old problem: e.g., CDN, P2P, Gossip approaches have been around for a while. But these typically focus on bandwidth efficiency or delivery odds. The focus in this work is on the metric of completion time. This requires different strategies. Some motivating use cases are: (1) configuration to remote nodes—e.g., in a CDN; (2) distributed monitoring—e.g., coordinating before measurement; and

(3) developers—e.g., a long develop-deploy cycle in PlanetLab can hurt productivity.

The system developed is LSync. It provides a simple folder sync interface. The lessons and contributions are: (1) existing systems are suboptimal mainly because they are not favorable when there are slow nodes; (2) completion time depends on the set of target nodes, so LSync selects the best set of nodes; (3) end-to-end transfer can be faster than an overlay, because of startup latency, so overlay is used only when appropriate; (4) overlay performance changes at short time scales, so transfers are adapted while they are taking place. Existing systems assume an open client population, so their main goals are maximum average performance, maximum aggregate throughput, etc. LSync focuses only on internal dissemination within a fixed client population. Thus it aims to minimize completion time. This time is dominated by slow nodes.

LSync uses server's spare bandwidth to assist slow nodes. The question is how to do this efficiently. First, look at node scheduling—either do fast first, or slow first. Intuitively, fast first is optimal for mean response time, while slow first gives preference to nodes that are expected to be slow. The results show that in fast first, slow nodes become a bottleneck at the end. Slow first starts slower but ends quicker. But not every scenario requires waiting for 100% sync. LSync allows the specification of a fraction of nodes, called the target sync ratio. LSync integrates node selection with the aforementioned scheduling.

Leveraging an overlay mesh is scalable, but needs to be careful about startup latency. For small files, only using end-to-end transfer (E2E) is faster than using an overlay. For large files, overlay is faster than E2E. So, LSync should adapt to the target ratio, file size, bandwidth, etc. The approach used is that LSync monitors the overlay's startup latency. It splits nodes into an overlay group and E2E group, depending on the overlay connection speed, and tries to match the completion time of both. To deal with overlay performance fluctuation, adaptive switching is used.

Evaluation was done on PlanetLab, using multiple CDNs, and compared against multiple systems. A dedicated origin server was used with 100 Mbps bandwidth. LSync improves over other systems, by choosing E2E vs. overlay rates. Less variation is shown with adaptive switching.

Jon Howell, Microsoft Research, asked for clarification of the target completion ratio—whether or not they care about which specific nodes are completed. Wonho answered that you can do both. You can simply tune the completion ratio if you aren't concerned which set of nodes are completed. If you do care, you can specify a new set of nodes, and then run with completion ratio set to 1.0.

Deduplication

Summarized by Anshul Gandhi (anshulg@cs.cmu.edu)

Generating Realistic Datasets for Deduplication Analysis

Vasily Tarasov and Amar Mudrankit, Stony Brook University; Will Buik, Harvey Mudd College; Philip Shilane, EMC Corporation; Geoff Kuenning, Harvey Mudd College; Erez Zadok, Stony Brook University

Deduplication is the process of eliminating duplicate data in a system and has been the focus of a lot of prior work. Unfortunately, most of the prior work has looked at different data sets, and so it is almost impossible to compare the performance of these different deduplication approaches. A survey of the data sets used by 33 deduplication papers was conducted by the authors and they found that most of the data sets were either private (53%), hard to find (14%), or contained less than 1 GB of data (17%). Thus, there is a need for an easily accessible data set with configurable parameters.

In order to create realistic data sets, Vasily Tarasov presented work to accurately track how file systems mutate over time. They do so by observing consecutive snapshots of real data sets, combined with a Markov model and multi-dimensional analysis. Comparison with the evolution of real file system images shows that the authors' emulation approach very accurately tracks the number of chunks and files over time, as well as the number of chunks with a given degree of duplication. Importantly, the file system profile sizes generated by the authors are 200,000 times smaller than the real profile sizes. The emulation time is proportional to the size of the data set, with a 4 TB data set emulation requiring about 50 minutes.

Haibo Chen from Shanghai Jiao Tong University asked about the differences in numbers between emulation and live file systems. Vasily answered that the emulation is a statistical process and so there would naturally be differences from time to time between emulation and the live system. However, Vasily felt that the emulation was close enough to the live system evolution. Haibo then asked whether the emulation runtime could be reduced by parallelization. Vasily agreed that it could; in their current work, scanning the data sets is done in parallel, but everything else is serialized, and thus, there is potential for parallelization.

An Empirical Study of Memory Sharing in Virtual Machines

Sean Barker, University of Massachusetts Amherst; Timothy Wood, The George Washington University; Prashant Shenoy and Ramesh Sitaraman, University of Massachusetts Amherst

Sean Barker presented this work which analyses the potential of page sharing in virtualized environments. Page sharing is a popular memory deduplication technique for virtual machines in which duplicate pages are eliminated. There has been a lot of prior work in exploiting page sharing for deduplication, with recent publications eliminating more than 90% of duplicate memory pages. However, the levels of sharing typically seen in real-world systems and the factors that affect this sharing remain open questions. The goal of the authors' work is to answer such questions.

The authors looked at a wide variety of memory traces, including uncontrolled real-world traces as well as controlled, configurable synthetic traces. Results indicate that sharing within a single VM (self-sharing) is about 14%, whereas sharing between VMs is only about 2%. Thus, 85% of the potential for deduplication is within a VM, indicating (very interestingly) that page deduplication is quite useful even for non-virtualized systems. Further investigation revealed that most of the self-sharing (94%) is because of shared libraries and heaps. However, the amount of self-sharing is largely impacted by the choice of base OS. Likewise, sharing across VMs is also impacted by the base OSes, with sharing being significant when the VMs have the same base OS as opposed to different base OSes.

The case study drew a lot of questions from the audience. Thomas Barr from Rice University asked whether the authors had looked at sharing larger multiples of page sizes. Sean answered that they didn't look much at coarse-grained sharing since the amount of sharing in this case was much smaller. Ardalan Kangarlou from NetApp asked whether the numbers for sharing in prior work were higher because they looked at synthetic workloads. Sean replied that the amount of sharing depends on the data set, and for the uncontrolled data set that he was looking at, the sharing was much lower. He urged the audience to look at actual data sets. Someone noted that memory contents change from time to time, and wondered whether vendors really benefit from sharing. Sean acknowledged that short-lived data is hard to capture, but that was a separate issue. Jiannan Ouyang from University of Pittsburgh asked about the size of memory footprint in the workload. Sean answered that the VMs they used had 2 GB of memory (each) on them, and since they had lots of applications running, he guessed that a good portion of the 2 GB memory was being used.

Primary Data Deduplication—Large Scale Study and System Design

Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Oltean, Jin Li, and Sudipta Sengupta, Microsoft Corporation

Sudipta Sengupta and Adi Oltean jointly presented this work, which will be part of Windows Server 2012. Sudipta started this presentation, which looks at deduplication in primary data, as opposed to the more common case of backup data. Primary data deduplication is important because of the continuing growth in the size of primary data and because this is the number one technology feature that customers are looking for when choosing a storage solution. The main challenge in primary data deduplication is that it needs to be non-intrusive to the workload.

The key design decision made by the authors is to post-process deduplication, which helps to schedule deduplication in the background when data is not hot. Also, the authors decided to use a larger chunk size (80 KB), which helps to reduce metadata, and thus reduces deduplication overhead. To compensate for the loss in deduplication opportunity due to larger chunk sizes, the authors use chunk compression. The authors also modify the basic fingerprint-based chunking algorithm to reduce the forced chunk boundaries at the maximum chunk size and to obtain a more uniform chunk size distribution. Lastly, in order to reduce the RAM footprint and the number of disk seeks, Adi presented the idea of partitioning the data, then performing deduplication on each partition, and, finally, reconciling the partitions by deduplicating across them. Performance evaluation of this approach reveals that deduplication throughput is about 25–30 MBps, which is about three orders of magnitude higher than previous work. Deduplication takes up 30–40% of one core, leaving enough room (assuming a manycore server) for serving primary workload.

Haibo Chen from Shanghai Jiao Tong University asked about the effects of data corruption on deduplication. Adi replied that they have looked at corruption and recovery, but this was not part of the paper. Essentially, they ensure that in case of a crash, data can be recovered so that the customer has peace of mind. Further, if corruption is in the I/O subsystem or the bus, it will be isolated.

Languages and Tools

Summarized by Asia Slowinska (asia@few.vu.nl)

Design and Implementation of an Embedded Python Run-Time System

Thomas W. Barr, Rebecca Smith, and Scott Rixner, Rice University

Even though there are dozens of microcontrollers around us—e.g., in cars, appliances, and computer electronics—the programming environments and runtime systems for them are extremely primitive. As a result, programming these devices is difficult. To address these issues, Thomas Barr presented Owl, a project which aims to let developers "build a toaster in Python." Owl is a Python development toolchain and runtime system for microcontrollers. It also includes an interactive development environment, so that a user can connect to a device, and type Python statements to be executed immediately. As a result, experimenting with and programming microcontrollers becomes a much simpler task.

Microcontrollers come with limited resources: e.g., 64–128 KB of SRAM, and up to 512 KB of on-chip flash. These constraints require that Python code be executed with low memory and speed overheads. During his presentation, Barr discussed two of the features of Owl that make this possible. First, he explained how a compiled Python memory image is executed directly from flash, without copying anything to SRAM. One of the challenges here is to represent compound objects in such a way that they do not contain references to other objects—only then can they be used directly without an extra dynamic loading step. The next feature concerned native C functions that are called from Python to, for example, access peripherals. Owl provides a mechanism that wraps the C functions automatically, so that a programmer does not need to bother with converting Python objects into C variables, and vice versa. A full description of the Owl architecture is in the paper, and the authors can be reached at embeddedowl@gmail.com.

To demonstrate that the Owl system is practical, Barr showed a video of an autonomous RC car that uses a controller written entirely in Python. The car successfully detected and avoided obstacles as it zoomed around a room. A full description of the architecture of Owl can be found in the paper, and the authors can be reached at embeddedowl@gmail.com.

A questioner wondered how Owl provides access to some sort of global notion of time. Barr said that the virtual machine provides a function call that returns the number of milliseconds since the virtual machine booted. Rik Farrow asked how Owl makes interacting with peripherals simpler for a programmer. Barr explained that the embedded Python interpreter allows the programmer to interactively probe the device. Thus it becomes easy to tell whether a piece of code works as expected.

AddressSanitizer: A Fast Address Sanity Checker

Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov, Google

Even though memory corruption bugs have been known about and fought for years, no comprehensive tool to detect them is available. To address this problem, Konstantin Serebryany presented AddressSanitizer, a memory error detector for C/C++ programs, which prevents out-of-bounds memory accesses and use-after-free bugs. The authors invite others to try it out. It is publicly available at http://code.google.com/p/address-sanitizer/.

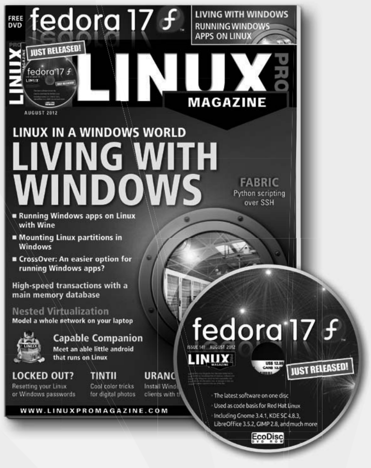
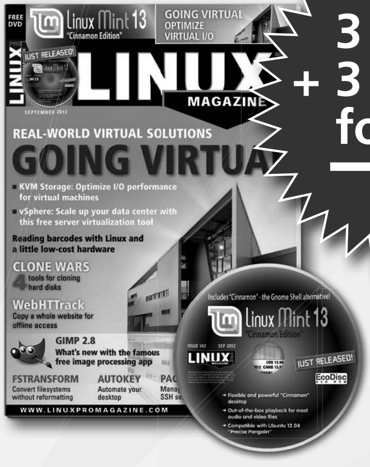
AddressSanitizer is a compiler-level solution—it instruments the protected program to ensure that memory access instructions never read or write, so called, "poisoned" red zones. Red zones are small regions of memory (currently 128 bytes) inserted in-between any two stack, heap, or global objects. Since they should never be addressed by the program, an access to them indicates an illegal behavior. This policy prevents sequential buffer over- and underflows and some of the more sophisticated pointer corruption bugs. To deal with heap-use-after-free errors, AddressSanitizer marks a freed memory region as "poisoned." Until this region is allocated again, any access to it causes an alert. AddressSanitizer uses its own tailored instrumentation of malloc and free, which keeps a released memory region in "quarantine" for as long as possible. By prolonging the period in which the memory buffer is not allocated again, it increases the chances of detecting heap-use-after-free bugs.

AddressSanitizer scales to real-world programs, and the developers at Google have been using it for over a year now. It has detected over 300 previously unknown bugs in the Chromium browser and in third-party libraries, 210 of which are heap-use-after-free bugs. The tool has a fair amount of overhead—it incurs 73% runtime overhead for the SPEC CPU2006 benchmark, and almost none for the I/O intensive Chromium browser.

During his presentation, Serebryany challenged the audience and hardware companies to attempt an implementation of AddressSanitizer in hardware. Rik Farrow asked what instruction would have to be added. Serebryany explained that a hardware version of the check which is performed on memory accesses—to ensure that the accessed memory is not poisoned—would be welcome. It would both improve performance and reduce the binary size. Since the current implementation of AddressSanitizer builds on the LLVM compiler infrastructure, the next questioner asked if Google plans to port it to gcc. Serebryany replied that they have already a version which can successfully compile the SPEC CPU2006 benchmark, but it is not fully fledged yet.

For the complete 2012 USENIX Annual Technical Conference report and summaries from HotCloud '12, HotPar '12, HotStorage '12, and the panel at our first Women in Advanced Computing Summit, visit: www.usenix.org/publications/login.

RISK-FREE TRIAL!

3 issues + 3 DVDs for only $3

PRACTICAL. PROFESSIONAL. ELEGANT.

Enjoy a rich blend of tutorials, reviews, international news, and practical solutions for the technical reader.

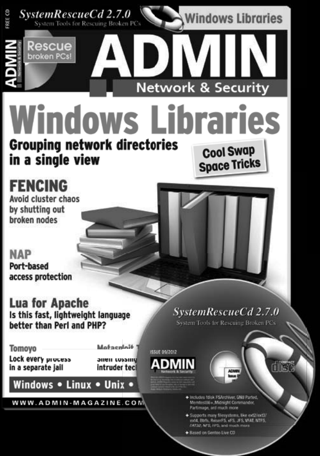
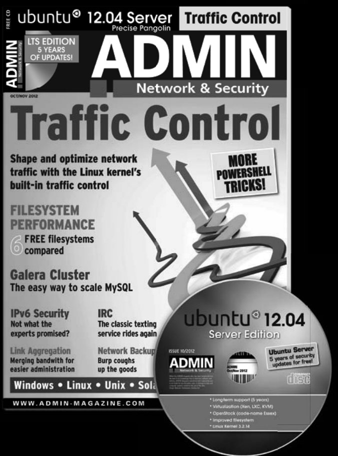
ORDER YOUR TRIAL NOW!

shop.linuxnewmedia.com

2 ISSUES ONLY $15.99

ALSO AVAILABLE

ADMIN: REAL SOLUTIONS FOR REAL NETWORKS

Technical solutions to the real-world problems you face every day.

Learn the latest techniques for better:

- network security
- system management
- troubleshooting
- performance tuning
- virtualization
- cloud computing

on Windows, Linux, Solaris, and popular varieties of Unix.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710

POSTMASTER
Send Address Changes to *;login:*
2560 Ninth Street, Suite 215
Berkeley, CA 94710

PERIODICALS POSTAGE
PAID
AT BERKELEY, CALIFORNIA
AND ADDITIONAL OFFICES

LISA '12
26th Large Installation System Administration Conference
Strategies, Tools, and Techniques

December 9–14, 2012
San Diego, CA

Sponsored by:
usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION
in cooperation with **LOPSA**

Keynote Address by Vint Cerf, *Google*

Join us for 6 days of practical training on topics including:

- **Virtualization with VMWare**
  John Arrasjid, Ben Del Vento, David Hill, Ben Lin, and Mahesh Rajani, *VMware*
- **Using and Migrating to IPv6**
  Shumon Huque, *University of Pennsylvania*
- **Puppet**
  Nan Liu, *Puppet Labs*

Plus 3-day Technical Program:

- **Invited Talks** by industry leaders such as Owen DeLong, Valerie Detweiler, Matt Blaze, and Selena Deckelmann
- **Refereed Papers** covering key topics: storage and data, monitoring, security and systems management, and tools
- **Workshops, Vendor Exhibition, Posters, BoFs, "Hallway Track,"** and more!

Register by November 19th and SAVE!

www.usenix.org/lisa12