Refactoring human roles solves systems problems

Jeremy Elson and Jon Howell
Microsoft Research

Abstract

Several persistent problems in system administration, deployment, configuration, and usability stem from a common root: the conflation of roles between developers, users, hardware wranglers, and software integrators. We develop a taxonomy of such conflations, and show that identifying each conflation characterizes one or more familiar systems problems. Furthermore, the taxonomy suggests directions for solving these problems: refactor components to produce interfaces that cleanly decouple these roles. Recent trends in web-based client applications and cloud-based services have demonstrated the feasibility of such changes. We describe recent projects our group has done that further decouple the roles, and we use the taxonomy to propose new projects.

1 Introduction

Today's computing landscape is defined largely by the commoditization of software. In other words, companies that make aluminum widgets don't need to learn how to write software in order to use computers for their accounting. Software specialists—that is, developers—write the software, and sell it to everyone who needs to do accounting. The developers can afford to spend a lot of time making the software better because that time is amortized over everyone who buys it. Meanwhile, widget companies can spend time on more widget research rather than software development. Everyone benefits: the global increase in efficiency yields both better software *and* better widgets.

Commodity software became ubiquitous because it decouples two distinct roles that were once conflated:

- *developer*—writes software
- *user*—uses software to get work done

These roles are well known. However, there are two more roles, equally pervasive but less often discussed, that remain conflated, both with each other and with the first two roles:

- *hardware wrangler*—buys, powers, cools, networks, and repairs hardware
- *software integrator*—ensures independent software components work together, applies security patches

We observe that many of the frustrations that people feel towards computers today can be neatly categorized as an unnecessary coupling of two or more computing roles. *I can't buy a new computer* (as a hardware wrangler) *because I don't have time to reinstall all my software* (as a software integrator).

Explicitly separating these four roles also sheds light on the excitement around recent innovations in cloud computing: they are appealing precisely because they decouple the roles. For example, in Amazon's EC2 [7], software integrators only need to define virtual machine images; Amazon wrangles the hardware that runs them. Microsoft's Azure [4] and Google's App Engine [2] decouple development from both hardware wrangling and software integration: Developers only provide a program that handles individual web requests, while Microsoft and Google integrate DNS, virtual machines, web server software and load-balancers, ensuring every web request that arrives is routed to an instance of the developer's program.

In both of these cases, just as in the creation of the software industry, everyone benefits; there is a global increase in efficiency. People who want to be web developers or software integrators can do so without spending time building datacenters. Datacenter experts can leverage their expertise over a larger set of users.

This paper has two goals. First, in Section 2, we show that conflations of these four roles account for many of today's problems in both desktop and server computing.

We consider a series of role conflations and describe the consequent problems. Second, we assert that framing the problem this way provides a useful blueprint for solutions: roles should be decoupled by introducing clean interfaces at role boundaries. Section 3 describes some projects we have built that follow this blueprint, and proposes new projects to solve problems at other boundaries.

2 Problems

The introduction described four computing roles: hardware wrangler, developer, software integrator, and user. This section examines every pair of these roles, and considers the problems that arise when they are unnecessarily conflated. Our analysis is summarized in Table 1. Only *user* and *developer* are consistently decoupled today; the other pairs give rise to long-standing and familiar problems.

2.1 *user* and *software integrator*

Perhaps the most troublesome and well-known conflation is of user and software integrator. In today's desktop computing world, users are constantly forced into managing their software and operating systems. Systems researchers pay much attention to user's configuration problems [18, 14, 16]. But why are users in the business of configuring software in the first place? That Grandma needs tools to help her manage her registry, install new device drivers, or try to recover from "**DLL Hell**" is clear evidence of a conflation of roles.

2.2 *user* and *hardware wrangler*

Most desktop computing users wrangle their own hardware. But why should who owns the hardware be so closely coupled to who can use the hardware? Users' state, unfortunately, is often tied to the hardware they've purchased. Decoupling has benefits: An Internet **kiosk** becomes as useful as your laptop. A **dead hard disk** becomes a minor inconvenience; an assistant can replace it as easily as refilling the paper in a printer. You can borrow a friend's laptop hardware, using the software you are familiar with, and she need not worry about exposing personal data to you.

The vision of "Your environment available anywhere" has been long held by researchers, but can now be seen in a new light: the user of a software shouldn't have to care who wrangled the hardware, or when. The roles should be deconflated.

2.3 *developer* and *hardware wrangler*

People who build scalable server applications use two terms to describe two different scaling strategies: *scaling up* means run an application on an ever-more-powerful computer, with faster multicore CPUs, more RAM, and more disks. *Scaling out* means running on more and more computers connected by a network. Why does this distinction exist? In both cases, a hardware wrangler is buying more CPUs, RAM, and disks. Why is development somehow fundamentally different when those resources come with extra cases and power supplies? The consequence is that the developer of a high-scale application cannot escape an intimate knowledge of the hardware wrangler's job: the topology and configuration of the actual cluster running the software.

Of course, a single machine differs from a cluster in both its failure modes and its inter-CPU access latencies. But are these differences so fundamental as to require a completely different development model, as they do today?

Even software written against the scale-out model, at large enough scales, must plan for "rack locality." Developers who write communication-intensive code for large clusters must do so in consideration of the cluster's interconnections. Ideally, those interconnections would be solely at the discretion of the hardware wrangler and hidden from the developer.

2.4 *SW integrator* and *hardware wrangler*

The ISP or startup that deploys its own data center to host a web mail or blog hosting service finds itself at the intersection of *software integrator* and *hardware wrangler*. The organization wishes to integrate existing software into a solution tailored for its customers, but it finds itself learning to manage a **one-off data center** as well.

2.5 *developer* and *software integrator*

One instance of this conflation occurs in the cloud: A web developer may want to reason about individual web requests. Yet today it is difficult for developers to write scalable web applications without carefully integrating their development efforts with the external factors, including the load-balancing method in use and the operating system configuration.

Another instance of this conflation occurs at the client machine. Today, the interface between the web application developer and the user's experience has become very wide and rich, comprising HTML, HTTP, CSS, JPEG, PNG, SVG, Java, Javascript, Flash, Silverlight, various video and audio codec standards, PDF, and so on. The **wide web interface** is so wide that two client

	user	hardware wrangler	software integrator
developer	solved by the software industry	§2.3 scale up vs. scale-out	§2.5 cloud: vertical service integration client: wide web ifc
software integrator	§2.1 client: DLL hell, shared configuration	§2.4 one-off data centers	
hardware wrangler	§2.2 client: kiosk, dead hard disk cloud: render farm		

Table 1: This matrix identifies problems that occur when two roles (column and row headers) are conflated. Some conflations occur at the client machine; others occur "in the cloud" (at server machines).

installations rarely exhibit compatibility, much less performance equivalence; the poor developer is left building layers of compatibility machinery to account for individual browser implementations. Worse yet, the wide standard evolves rapidly. If a web developer wishes to exploit new functionality in a new version of a client component, such as IE or Flash, she must either wait for widespread deployment of that specific version, provide degraded fallback functionality in her application, or require clients to accept a version upgrade. This last demand is invasive, because upgrading the client component extends the client TCB, and thus constitutes an extension of trust. That trust decision is made by the software integrator; perhaps a corporate IT department, but very often by an individual user forced to act as software integrator.

3 Solutions

In the previous section, we described the problems that exist when computing roles are conflated. Now, we turn to the solution space, analyzing it using the same taxonomy. Many solutions have a common theme: they decouple previously conflated roles, and create simple, explicit interfaces at role boundaries. This section shows that our taxonomy gives structure to a number of existing and proposed systems, and we posit that it also suggests useful future research directions.

3.1 *user* and *software integrator*

Web applications are popular in part because they address this conflation. Users can just use a web application, without having to install or configure it. In a world that consists entirely of web applications, the user needs to do nothing but get a working web browser. The managers of web-app sites take on the role of software integrators, ensuring that their applications reference every required component, in the version required. This software stack is logically "installed" every time the user visits the site and uninstalled the moment the user closes a window.

With web applications, the software integration task has not just moved; it has been dramatically simplified. Application isolation enforced by web browsers, originally designed for security, had the unexpected additional benefit of preventing shared state, such as configuration and libraries. This prevents the inter-application dependencies that plague desktop computers. Unlike on a desktop operating system, "installing" a new web application will never break the configuration of an old one.

Web applications have not solved this problem completely. Only a tiny fraction of legacy desktop software has been rewritten in Javascript or another "web language". However, recent work, including the authors' Xax [9], and the concurrent Native Client work by Chen et al. [17], might be a solution. Both projects demonstrate native execution environments that allow legacy code to be executed in secure silos. The native-code silos still enforce both the security required by the web's trust model and the isolation that removes inadvertent inter-application dependencies. This ability opens the door to widespread redeployment of conventional desktop software as web applications, extending the deconflation afforded by today's web applications to virtually all software. Of course, there are still thorny details. Much software—music players, photo management, video editors—requires access to local devices or storage, and cannot be made into web applications until there is a safe model for such access. Projects such as Google Gears [3] are working in this direction.

Even in a world where all applications are web applications, the user would still be required the manage the considerable desktop software stack between the bare metal and the web browser, plus all its attendant extensions. Might it be possible to extend the web application model all the way down to the hardware? Perhaps even device drivers could be made as ephemeral and replaceable as any web application. When a PC is turned on, perhaps it could only be told which web application

it should run; the hardware manufacturer might provide a list of potential device drivers, and the software developer a list of constraints, e.g., this CAD program requires at least DirectX 19.4. The nascent "AXA" project at MSR's operating systems group is working in this direction.

3.2 *user* and *hardware wrangler*

The web application deployment model begins to deconflate these roles: part of the model is that user's documents are often stored in the cloud, not locally, decoupling users' state from a particular hard drive. However, projects such as Gears that give web applications access to local storage can break this abstraction. If our supposition about the importance of decoupling is correct, the right architecture makes local disks nothing more than a cache for data that is stored canonically and durably in the cloud.

Even converting entirely to cloud storage isn't enough deconflation for hard-core users who need to wrangle lots of extra hardware for special tasks. Artists and video editors wrangle racks of machines into render farms; engineers data-mine over terabytes of data across hundreds of machines. This conflation is partially addressed by data-flow languages, as we will describe in Section 3.3, and partially by our recent "Utility Coprocessor" project. The Utility Coprocessor [10] is a library that lets desktop software wrangle hardware from a utility computing service (e.g., EC2), and automatically ensure that the farm's software configuration matches the user's for the length of the computation. The user no longer wrangles computers, racks, and cables; instead, he can attach his desktop application to a large-scale interactive render farm with a credit card.

3.3 *developer* and *hardware wrangler*

The dichotomy between scale-up and scale-out (§2.3) still pervades general purpose applications. Most developers must still think explicitly about machine boundaries when writing scalable applications. However, there has been much recent progress in specific domains. The recent emergence of the Map-Reduce model [8, 12, 1] has proven successful at letting developers specify dataflow problems that implicitly harness thousands of machines. Microsoft Azure and Google App Engine, which we mentioned in Section 1, let developers think solely about the code required to service a single web request. The underlying infrastructure is responsible for scaling: a high-performance clustered filesystem, an automatically expanding pool of machines on which to instantiate the application, and load balancing.

An important limitation of these environments is that they require services to be written afresh in a new environment. Programs like Postfix and Wordpress, coded against a more conventional POSIX single-system image, however, encode valuable protocol and domain knowledge. Our group's nascent Getwell project aims to produce a software stack that can be integrated with commodity legacy software to make it transparently and elastically scalable. Just as the Utility Coprocessor (§3.2) helps the developer package scalability into a parallelizable desktop application, Getwell helps the developer package scalability into a service. The software integrator "installs" the software on a utility reservation (cluster of cloud machines), and configures it to provide the desired service and interact with other services from the integrator. The Getwell library automatically wrangles and scales hardware resources using the utility abstraction, so only the hardware wrangler thinks about resource allocation, statistical multiplexing, spare capacity, and forklifts.

As with App Engine's Datastore or Azure Storage, a central component of Getwell is a scalable, fault-tolerant file system. By meeting legacy software at the POSIX file system interface, we hope to exploit its tolerance of file system latency and its recovery after process failure, and thereby avoid the transparency trap of distributed shared memory [13].

Projects like Monsoon [11] and Vahdat's proposed datacenter network topology [6] also aim to decouple hardware wranglers from developers. Using commodity components configured in a fat tree, they construct data center networks with full bisection bandwidth. Within such a layout, software that once worried about "rack locality" can safely abstract away the network topology while still extracting maximal performance from the individual machines.

3.4 *SW integrator* and *hardware wrangler*

The recent emergence of a practical utility computing interface, such as implemented by Amazon's EC2, addresses this problem. The interface between software integrator and hardware wrangler is now explicit and narrow: a virtual machine image. The software integrator defines it, and the hardware wrangler instantiates it.

Some corporate IT departments use a similar approach to manage desktop hardware: one group of software integrators defines and manages a standard system image; another group of hardware wranglers deploys that image onto every desktop and laptop purchased. This approach partly relieves the conflation, although because each corporation uses a different set of software, each corporate IT department must repeat the software integration task. Ideally, the corporation itself might like to act only as a

"user," outsourcing even software integration.

Our taxonomy also suggests that the Azure and App Engine services should be refactored. Today, both services couple a high-level distribution interface with an actual data center, coupling software integration to hardware wrangling. Instead, they should be offered as a software library that sits on top of a narrow, low-level hardware interface (such as the VM interface to EC2). This would enable the developer to select the best scalability library, then sell his software to the software integrator, who in turn could contract with the datacenter hardware wrangler of her choice.

3.5 *developer* and *software integrator*

As described above (§3.1), the web model helps decouple users from software integrators. However, the web standard, which links the developer to the software integrator, is so wide and fast-evolving that it does a poor job decoupling the two. Developers exert effort testing against a variety of browser/OS combinations to ensure compatibility, and the software integrator must vet rapid browser and plugin releases for security and interoperation.

We propose a narrow, slowly-evolving interface between the developer and the client software integrator. Specifically, we observe that, given web-deployable native code, almost all rapidly-evolving code can be treated as library code: layout engines (HTML, CSS), text rendering (Pango, TrueType, PDF), raster and vector rendering (JPG, PNG, SVG), audio/video codecs (MPEG, Ogg), runtime (Javascript, JVM), and even 3D rendering (OpenGL).

Suppose we restrict the client web interface to a strongly-isolated native code container [9, 17] plus a low-level display blit and uncompressed audio interface. A practical model would also address isolated persistent storage [3] and a security policy for communication [15]. We call this model an "exobrowser", as it employs the exokernel model of handling only isolation and resource management in the "kernel". Free to radically alter his library operating system, the developer need no longer negotiate with the software integrator to upgrade to the latest rendering engine or codec (lest it be a Trojan [5]).

4 Summary

The four roles of user, developer, software integrator, and hardware wrangler are often conflated in contemporary system architectures. This paper articulates a taxonomy of such conflations, showing how each conflation appears in common system architectures, leading to problems. Framing those problems according to our taxonomy also points the way to solutions. Each has a common theme: refactor and tighten the interfaces that decouple the roles of the participating humans. Our group has undertaken recent projects to address instances of these problems; this paper uses the taxonomy to identify further problems and propose solutions.

References

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

[2] Google App Engine. http://code.google.com/appengine/.

[3] Google Gears. http://gears.google.com/.

[4] Microsoft Azure. http://www.microsoft.com/azure/.

[5] Zlob trojan. http://en.wikipedia.org/wiki/Zlob_Trojan.

[6] AL-FARES, M., LOUKISSAS, A., AND VAHDAT, A. A scalable, commodity data center network architecture. In *SIGCOMM* (2008), pp. 63–74.

[7] AMAZON WEB SERVICES. EC2 elastic compute cloud. http://aws.amazon.com/ec2.

[8] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004), pp. 137–150.

[9] DOUCEUR, J. R., ELSON, J., HOWELL, J., AND LORCH, J. R. Leveraging legacy code to deploy desktop applications on the web. In *OSDI* (2008), pp. 339–354.

[10] DOUCEUR, J. R., ELSON, J., HOWELL, J., AND LORCH, J. R. The utility coprocessor: Massively parallel computation from the coffee shop. In *Under submission* (2009).

[11] GREENBERG, A., LAHIRI, P., MALTZ, D. A., PATEL, P., AND SENGUPTA, S. Towards a next generation data center architecture: Scalability and commoditization. In *PRESTO Workshop colocated with SIGCOMM* (2008), pp. 57–62.

[12] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In *EuroSys '07* (2007), pp. 59–72.

[13] LI, K., AND HUDAK, P. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst. 7*, 4 (1989), 321–359.

[14] VERBOWSKI, C., KICIMAN, E., KUMAR, A., DANIELS, B., LU, S., LEE, J., WANG, Y.-M., AND ROUSSEV, R. Flight Data Recorder: Monitoring persistent-state interactions to improve systems management. In *OSDI* (2006), pp. 117–130.

[15] WANG, H. J., FAN, X., HOWELL, J., AND JACKSON, C. Protection and communication abstractions for web browsers in mashupos. In *SOSP* (2007), pp. 1–16.

[16] WANG, H. J., PLATT, J. C., CHEN, Y., ZHANG, R., AND WANG, Y.-M. Automatic misconfiguration troubleshooting with PeerPressure. In *OSDI* (2004), pp. 245–258.

[17] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native Client: A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy* (2009).

[18] YUAN, C., LAO, N., WEN, J.-R., LI, J., ZHANG, Z., WANG, Y.-M., AND MA, W.-Y. Automated known problem diagnosis with event traces. In *EuroSys* (2006), pp. 375–388.