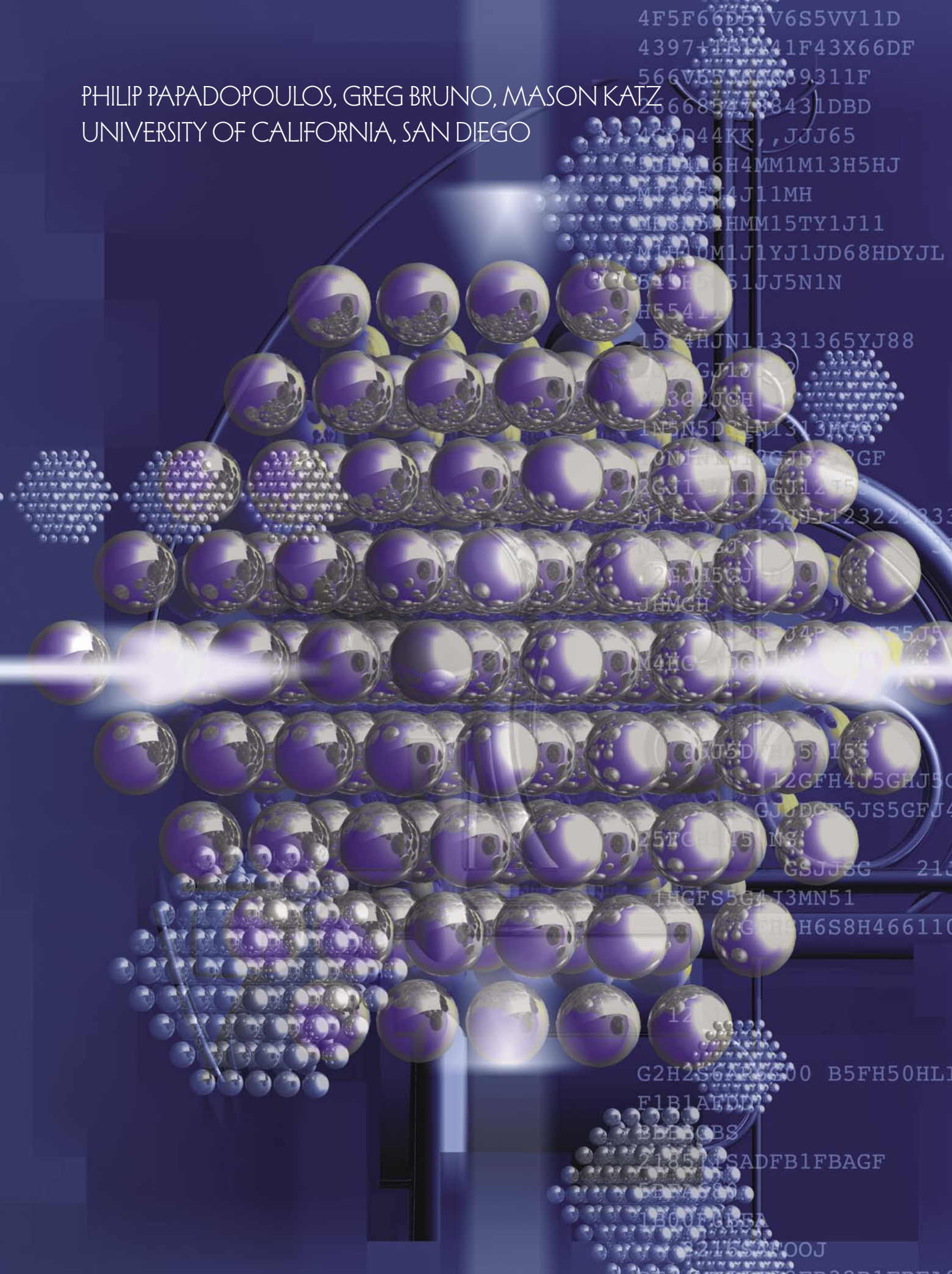
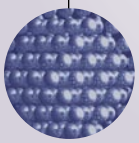
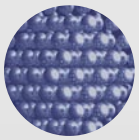
PHILIP PAPADOPOULOS, GREG BRUNO, MASON KATZ
UNIVERSITY OF CALIFORNIA, SAN DIEGO

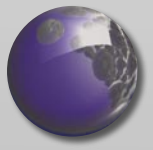


Beyond Beowulf Clusters





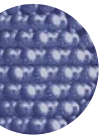
As clusters grow in size and complexity, it becomes harder and harder to manage their configurations.



In the early '90s, the Berkeley NOW (Network of Workstations) Project under David Culler posited that groups of less capable machines (running SunOS) could be used to solve scientific and other computing problems at a fraction of the cost of larger computers. In 1994, Donald Becker and Thomas Sterling worked to drive the costs even lower by adopting the then-fledgling Linux operating system to build Beowulf clusters at NASA's Goddard Space Flight Center. By tying desktop machines together with open source tools such as PVM (Parallel Virtual Machine), MPI (Message Passing Interface), and PBS (Portable Batch System), early clusters—which were often PC towers stacked on metal shelves with a nest of wires interconnecting them—fundamentally altered the balance of scientific computing. Before these first clusters appeared, distributed/parallel computing was prevalent at only a few computing centers, national laboratories, and a very few university departments. Since the introduction of clusters, distributed computing is now, literally, everywhere.

There were, however, ugly realities about clusters. The lack of tools meant that building 16 or 32 machines to work

Beyond Beowulf Clusters



closely together was a heroic systems effort. Open source software was (and often still is) poorly documented and lacked critical functionality that more mature commercial products offered on the "big machines." It often took months to get a cluster up and running and took highly trained experts to get it into that condition. It took even longer for applications to run reasonably well on these cheaper machines, if at all.

Nonetheless, the potential of building scalable and cheap computing was too great to be ignored, and the community as a whole grew more sophisticated until clusters became the dominant architecture in high-performance computing. Midsize clusters are now about 100 machines in strength, big clusters consist of 1,000 machines, and the biggest supercomputers are even larger cluster machines. For HPC (high-performance computing), clusters have arrived. Most of these are either Linux-based or a commercial Unix derivative, with most of the top 500 machines running a Linux derivative. There is a new trend toward better hardware integration in terms of blades. This helps eliminate significant wiring clutter.

The past 12 years of clusters have honed community experience—many can turn out "MPI boxes" (homogeneous hardware that enables message-passing parallel applications), and there are several software tools that understand clusters and allow non-experts to go from bare metal (e.g., that cluster SKU from your favorite computing hardware company) to functioning cluster made up of hundreds of individual nodes (computers) in a few hours. At the National Center for Supercomputing Applications, the Tungsten2 Cluster (512 nodes) went from purchase order placement to full production in less than a month and was one of the 50 fastest supercomputers in the world in June 2005.

It seems that the problems with clusters have been solved, but their wild success means that everyone wants to do more with them. While retaining roots born in HPC, clusters of Web servers, tiled display walls, database servers, and file servers are becoming commonplace. Nearly every entity in the modern machine room is essentially a clustered architecture. Building a specialized MPI box (classic Beowulf cluster) is a small subset of what is needed to support the needs of computational researchers.

Early clusters were tractable for experts because all the hardware was identical (purchased that way to simplify things) and every node in the cluster ran only a single software stack. To build these machines, the expert carefully created a "model" node through painstaking, handcrafted system administration. Then he or she took an image of this model and cloned bits onto all the other nodes. When changes were needed, a new model was built, and the nodes were recloned. The community learned that for clustered applications to function properly, software must be consistent across all nodes. A variety of mechanisms and underlying assumptions have been used to achieve consistency with varying degrees of success. Consistency itself isn't intractable, but achieving it becomes significantly more important as the complexity of the infrastructure increases.

The reason for the complexity of the modern machine room isn't just that all nodes are part of a particular logical cluster, it's that every one of the clusters needs a different software configuration. In Rocks, software that we have developed to provision and manage clusters rapidly, we call these configurations *appliances*. If every appliance has its own "handcrafted" model, the cost of the cluster goes up dramatically, uniformity of security policy across clusters is tied to the ability of the cluster experts to apply changes uniformly to all model nodes, and each (sub)cluster must use relatively identical hardware. This administrator-heavy model leads to inconsistency in the enterprise and relies too much on human "wetware," when what is needed is a programmatic approach.

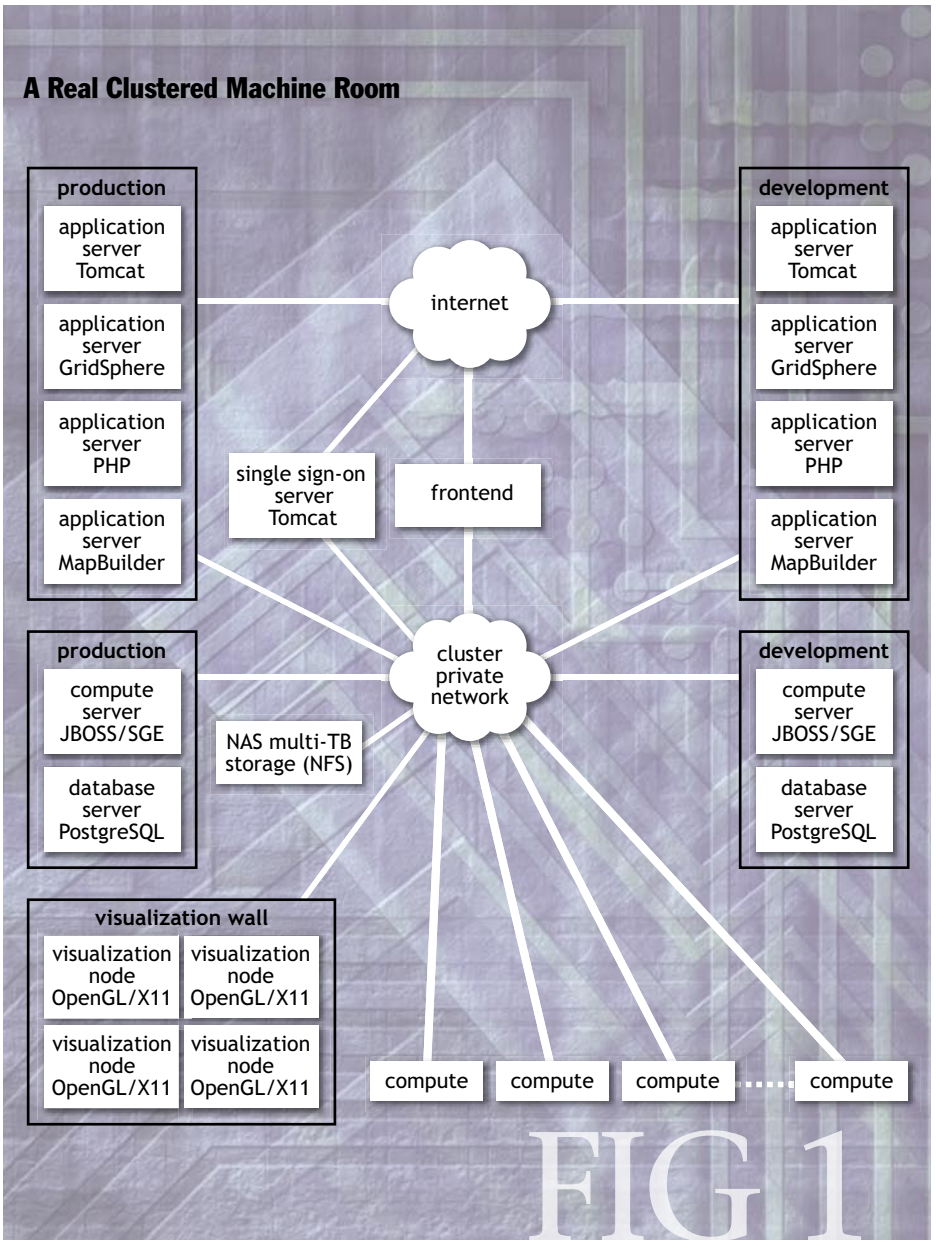
THE CAMERA EXAMPLE

To make this discussion more concrete, we describe a new resource called CAMERA (Community Cyberinfrastructure for Advanced Marine Microbial Ecology Research and Analysis) that we are building in collaboration with

the J. Craig Venter Institute. The goal of CAMERA (http://www.camera.calit2.net) is to build for computational biologists in the emerging field of meta-genomics a community resource that is initially focused on the study of microbial communities. CAMERA provides physical resources to enable searches and comparisons across a variety of gene sequence data.

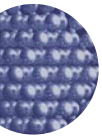
Initially populated with Global Ocean Survey[1] data gathered by the Venter Institute's *Sorcerer II* expedition (where microbial communities from 200 marine sites are being sequenced using shotgun sequencing, which proved to be a critical enabler for sequencing the human genome) and other key genomic data sets, CAMERA provides a "science destination" where scientists can computationally search through multiple data sets, download raw samples, and annotate gene sequences for the community. Underneath, real hardware, applications, data servers, and more must be put in place to provide a smooth computational engine. When a user submits a simple BLAST[2] query at the CAMERA portal, a cadre of back-end servers must work in concert to perform the analysis. This complexity must be hidden from the user for the site to be a useful tool.

Figure 1 illustrates the logical sets—appliances—and physical sets of machines that define CAMERA and include Web application servers, database servers, flat-file servers, compute cluster (classic Beowulf), and single sign-on grid certificate services. In the CAMERA clustered machine room shown in the figure, compute nodes form a classic Beowulf cluster. Application, database, visualization, storage, authentication, and database appliances complete the system.



CAMERA borrows much of its security and Web container infrastructure from BIRN (Biomedical Informatics Research Network; http://www.nbirn.net)[3] and over time will incorporate a large subset of software that is common to other projects. CAMERA-specific structures include application databases, Web applications, and specific computational biology tools and databases such as BLAST, HMMER,[4] and PFAM.[5] Like BIRN, CAMERA must support an ongoing evolution of the infrastructure and define pools for development

Beyond Beowulf Clusters



and production. While it takes a significant cross-section of expertise to define all of these different appliances, CAMERA is defined so that a single systems engineer can rapidly deploy the entire system.

CAMERA today defines 10 different and distinct appliances in its clustered machine room, and its initial deployment consists of nearly 200 different physical machines, each of which must be defined to fill a particular role. Herein lies the inherent complexity of this and many other computational portals—different applications have a litany of requirements of the system software underneath, and every machine or appliance must be configured correctly for the entire portal to work. In the simplest of terms, it comes down to selecting the appropriate software to install for a particular appliance and then configuring all of the software components correctly.

This doesn't sound too difficult until you look at a typical Unix (or Windows) server and see that more than 700 software packages (tens of thousands of files) and hundreds of configuration changes are needed to properly define any single machine. For CAMERA, simple multiplication indicates that across the complex of 10 appliances, 7,000 software packages and 2,000 configuration changes are required for proper definition. For 200 physical machines, these numbers balloon to 140,000 packages and 40,000 configuration changes. The critical issue in the clustered machine room is handling these very large numbers of configuration changes across all of the required logical appliances while simultaneously dealing with a constant flow of software updates.

In addition, for many large cluster applications to work reliably, all nodes used in a particular calculation require a core of software (e.g., libraries, services, kernels, and other items) to be at the identical revision level and configuration. Lazy updates to machines work well enough across an enterprise of *independent* user workstations, but often introduce too much uncertainty across a large collection of machines used for parallel computing.

The classic approach to this problem is for a team of systems administrators to handcraft the configuration of each appliance and then to use disk-cloning or disk-imaging methods to build clusters of similar appliances. For classic Beowulf clusters (aka HPC clusters), this is actually tenable if you just happen to have the on-site systems administration expertise to build the head-node and compute-node configurations. For the clustered machine room, it isn't tenable—it literally takes a savvy administrator days or weeks to craft the configuration of any particular computing appliance. If you put a team of 10 administrators on CAMERA, you might be able to craft 10 appliances in a single week, but then you must take care of consistency across appliances because all of the machines must work together to form a working and distributed system. The complexity of real clustered systems is simply outstripping the usual methods of system definition and administration.

ROCKS CLUSTERING: PROGRAMMATICALLY DEFINING MACHINES

Rocks (http://www.rocksclusters.org) is an open source cluster-building toolkit developed over the past six years. It has been used to build thousands of clusters and has a large and active e-mail discussion list (more than 1,500 subscribers). Rocks is used by CAMERA (and other groups) as a fundamental vehicle to attack the challenges of the clustered machine room. Rocks differentiates itself from the majority of system provisioning and management tools in that it manages a system description rather than a bit image of installed software.[6] State-of-the-art management when Rocks began more than five years ago was to rely on a cluster administrator managing the operating system and configuration of a system as a "golden" disk image. This image was manually configured according to site and individual machine localization requirements.

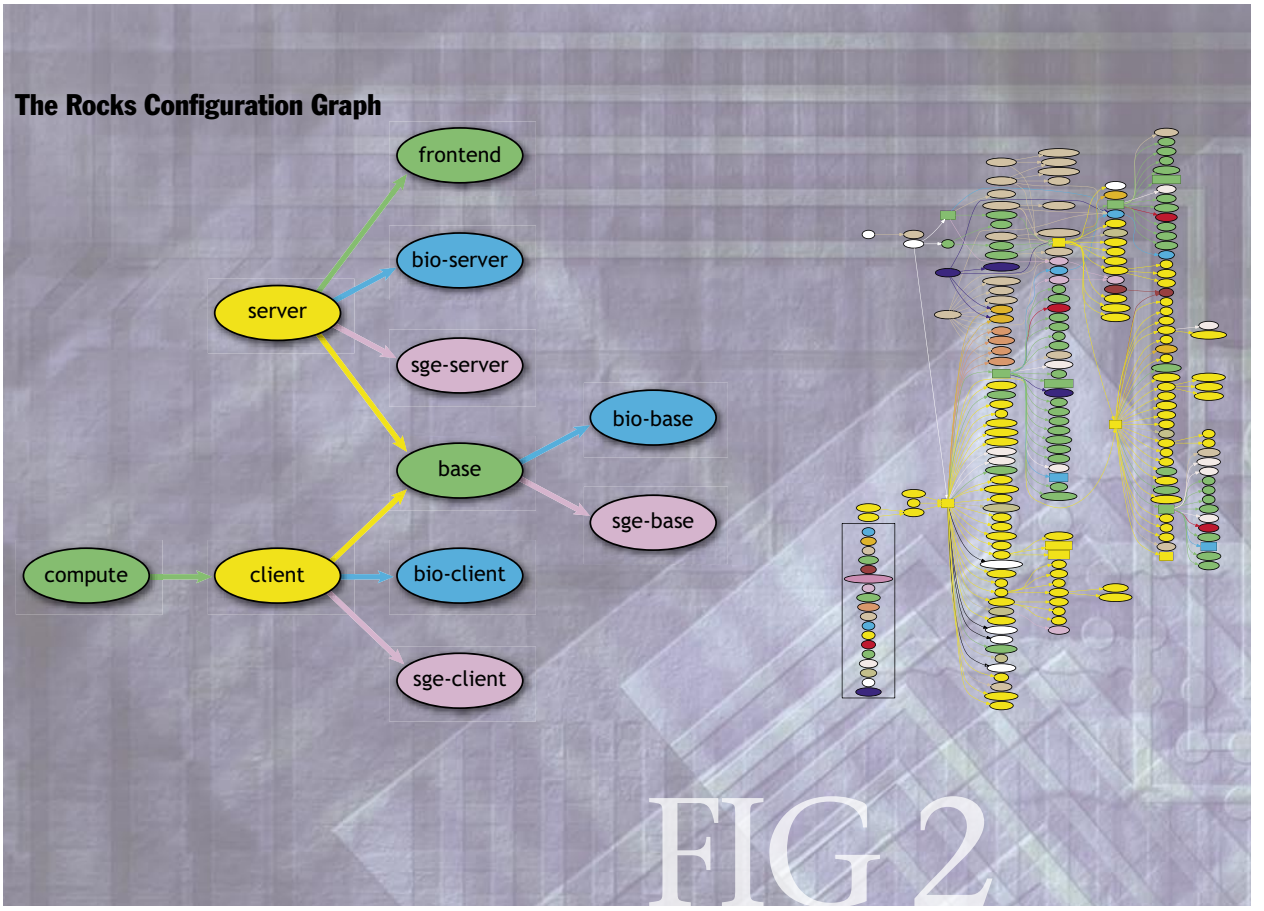
Rocks, on the other hand, manages all software as native operating-system packages and the selection and configuration of these packages for particular nodes as two distinct entities: the *complete operating-system distribution* and the *Rocks configuration graph*.[7] The operating-sys-

tem distribution is the union of all software that could be installed on any particular machine. The configuration graph selects which of these packages will be installed on a particular machine and exactly how to configure that software. The configuration graph is critical to breaking the tyranny of scale in the clustered machine room. Essentially the graph allows different appliances to share significant portions of their individual configurations. While a single appliance requires hundreds of software changes to function correctly, many of these changes are universal across appliances at a particular site. A new appliance might need to add or change only a dozen or so configuration parameters.

Figure 2 shows two representations of the Rocks configuration graph. The colors in this figure represent the *roll* (a fundamental building block of Rocks) from which the configuration is provided. Different colors indicate different rolls: yellow and green rolls include core Rocks functionality, and blue and violet represent the bio-informatics and SGE (Sun grid engine) rolls.

A roll itself is a set of binary packages and a *subgraph* that is connected into a master graph (represented on the right-hand side of figure 2). The full graph is defined and composed of all rolls selected by the administrator at build time. The binary packages in a roll supply the fundamental building blocks; nodes supplied by the roll describe packages and how to configure them; and graph edges supplied by the roll indicate which nodes should be included for a particular appliance type. This composition of all rolls is automatic and transparent to the end user. In Rocks, even the operating system (in our case, a Red Hat-compatible Linux base) is defined as a roll that contains only packages. This fundamental decomposition of a complete cluster with the ability to recombine these building blocks as a program provides *programmatic extensibility* of a core system without needing an expert systems administrator for every cluster.

In figure 2, the diagram on the left is a simplification of how Rocks defines *server* and *compute* appliances, defined by expanding instructions contained within each connected node. You can see that a Rocks *server* appliance (the master machine in a cluster) includes all the functionality of front-end, bio-server, SGE-server, base, bio-base, and SGE-base configurations, while a Rocks



Beyond Beowulf Clusters

compute appliance includes all the functionality of client, bio-client, SGE-client, base, bio-base, and SGE-base configurations. Also note that four independent rolls were used to build this composite graph. In the full graph, commonality among appliances is quite high, often with just a handful of nodes needed for differentiation. The diagram to the right shows the complete configuration graph for the CAMERA project (on the order of 200 nodes in a single graph). The graph includes the specification for the following appliances: front-end server, compute nodes, database servers, Web applications, security credential handling servers, NFS servers, and tiled-display nodes. Instead of a super administrator for each of these nodes, the specific details of building a particular Web application server are captured programmatically and are therefore reproducible.

HOW TO USE THE CONFIGURATION GRAPH

Rocks controls the provisioning of all appliances from the ground up, meaning a complete operating system installation from bare metal is performed on all machines through the traversal of the configuration graph (data partitions are not destroyed). Capturing the complete configuration of such varied appliance types has been a major effort for Rocks and will continue as people invent more complex system architectures. Building the operating system in this manner allows us always to begin with the same initial system state (the empty set) and to end with a predictable final state. This observation is critical to understanding that this allows both reproducibility and compatibility among appliances in the clustered machine room. The graph is really source code for a complete set of appliances; installation is then equivalent to compilation of the source code.

To build the description of any Rocks appliance, a program enters the graph at a particular known point (the graph is a multirooted tree, and roots of the tree are entry points of the graph) and starts expanding the instructions in each node of the graph. The instructions include packages to be loaded, as well as the configuration of these packages. Once the graph has been traversed, the complete description of a particular appliance has been completed. This description is then handed to the native installer (e.g., Anaconda for Red Hat-based machines), which carries out the instructions. When the installation is complete, the particular appliance is ready for work. While the binding of the configuration graph is loosely tied to Linux, in particular Red Hat, work is being done to abstract away this binding and make Rocks operating system neutral. Solaris is the first operating system that is being targeted.

Needless to say, a great deal of infrastructure is needed to make this work correctly. This infrastructure is itself

captured as a different appliance: a front-end (or head) node. There are a variety of ways to bootstrap a Rocks-configured clustered machine room, and we support building from a set of CDs, as well as over-the-network installations. The bootstrapping process builds an in-memory graph and uses the same programmatic structure.

At the core of Rocks is extensibility. Rolls allow us to extend any Rocks cluster with new functionality. Adding functionality (such as a queuing system for a classic Beowulf) is as simple as inserting the desired queuing system roll at bootstrap time. From there, configuration is automatic and programmatic. All of these processes help

break down the clustered machine room into manageable bits that can then be recomposed into new systems.

Rocks challenges the conventional wisdom of treating your installed computer as something special—in fact, we advocate wholesale reinstallation of any particular node, as opposed to patch management, for managing any system. Any desktop user is familiar with the message, "Windows just downloaded important updates and has rebooted your computer," and the same feeling of dread if a laptop needs replacement, as it can take weeks to "get it just right." We view the basic problem in this all-too-common scenario as being one where data (what you really care about) and the programs to manipulate that data (complete operating system and applications) are horribly intertwined. A regular reinstallation of the operating system has the beneficial effect of practically separating application data from the application binary.

WRAPPING UP

Clustered machine rooms present significant challenges in scaling configuration. The interdependent nature of systems in building complex endpoints such as CAMERA means that the "business as usual" of locally applying a magic administrator simply breaks down. Rocks is our implementation of capturing the expertise of configuring complex systems and allowing anyone to reproduce any system we build.

It is interesting to note that architects of individual programs (or even programming environments) have source-code control and building and test regimens so that they can reproduce their binary end products in a number of environments. Many commercial and open source projects have automated nightly "check out, build, test" regimens that do not require the programming team to carefully handcraft each and every compilation. It is curious that in modern systems, this same sort of rigor is not usually applied to the sum of the programs that defines any computer system, including your desktop. Q

REFERENCES

1. Venter, J.C., et. al. 2004. Environmental genome shotgun sequencing of the Sargasso Sea. *Science* 304(5667): 66-74.
2. Altschul, S.F., Madden, T.L., Schaffer, A.A., Zhang, J., Zhang, Z., Miller, W., Lipman, D.J. 1997. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research* 25(17): 3389-3402.
3. Grethe, J.S., Jovicich, J., Martone, M.E., Pieper, S., Brown, G.G., Ellisman, M.H., and Gollub, R.L. 2003. The Biomedical Informatics Research Network: Educational and Teaching Resources. Society for Neuroscience Annual Meeting, New Orleans.
4. Durbin, R., Eddy, S., Krogh, A., Mitchison, G. 1998. The theory behind profile HMMs. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press.
5. Sonnhammer, E.L.L., Eddy, S.R., Durbin, R. 1997. Pfam: A comprehensive database of protein domain families based on seed alignments. *Proteins* 28(3): 405-420.
6. Papadopoulos, P.M., Katz, M.J., Bruno, G. 2001. NPACI Rocks: Tools and techniques for easily deploying manageable Linux clusters. IEEE Cluster 2001, Newport Beach, CA (October).
7. Katz, M.J., Papadopoulos, P.M., Bruno, G. 2002. Leveraging standard core technologies to programmatically build Linux cluster appliances. Clusters 2002: IEEE International Conference on Cluster Computing, Chicago (April).

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

PHILIP PAPADOPOULOS is the program director of grid and cluster computing at the San Diego Supercomputer Center at U.C. San Diego and is involved in key research projects including BIRN (Biomedical Informatics Research Network), OptIPuter, GEON (Geosciences Network), and PRAGMA (Pacific Rim Applications and Grid Middleware Assembly). He received his Ph.D. in electrical engineering from U.C. Santa Barbara. He spent five years at Oak Ridge National Laboratory as part of the PVM (Parallel Virtual Machine) development team. He is also well known for the development of the open source Rocks cluster toolkit.

GREG BRUNO is a core developer for Rocks, developed at the San Diego Supercomputer Center at U.C. San Diego. Prior to the Rocks project, he spent 10 years with Teradata Systems developing cluster management software for the systems that supported the world's largest databases. He is a Ph.D. candidate at UCSD, where he is examining how heterogeneous disks affect parallel file systems.

MASON KATZ is the group leader for cluster development for the San Diego Supercomputer Center at U.C. San Diego, where his primary focus is on the Rocks cluster distribution. He is active in PRAGMA (Pacific Rim Applications and Grid Middleware Assembly). He has also worked at the University of Arizona on network security protocols (IPSec) and operating systems (x-kernel, Scout), and in the commercial sector as a realtime embedded software engineer.