Kittyhawk: Enabling cooperation and competition in a global, shared computational system

J. Appavoo
V. Uhlig
A. Waterland
B. Rosenburg
D. Da Silva
J. E. Moreira

Kittyhawk represents our vision for a Web-scale computational resource that can accommodate a significant fraction of the world's computation needs and enable various parties to compete and cooperate in the provisioning of services on a consolidated platform. In this paper, we explain both the vision and the system architecture that supports it. We demonstrate these ideas by way of a prototype implementation that uses the IBM Blue Gene®/P platform. In the Kittyhawk prototype, we define a set of basic services that enable the allocation and interconnection of computing resources. By using examples, we show how higher layers of services can be built by using our basic services and standard open-source software.

Introduction

Our aim is to develop a sustainable, reliable, and profitable computational infrastructure that can be easily used to create and trade goods and services. The globalization of computation and acceleration of commerce are inevitable, given the trends in digitalization of information and enablement of communications. In a sense, computation and commerce are indistinguishable. In the future, it is likely that virtually all information will be digital and will be manipulated, communicated, and managed in a digital fashion. In the future, the line between computation, global commerce, and humanity's information will become indistinct.

Kittyhawk is our vision of efficient, pervasive, worldwide computational capacity and commerce. We view worldwide computational capacity as requiring a significant fraction of the capacity of all available servers currently installed. At the time of this writing, the worldwide installed base of volume servers is reaching about 40 million units with an annual growth of about 4 million units [1]. The theoretical limit of an IBM Blue Gene*/P [2] installation is 16 million connected nodes; thus, a small double-digit number of geographically dispersed data centers would theoretically be sufficient to host the worldwide capacity of all currently installed volume servers.

Toward the fulfillment of the Kittyhawk vision, we explore and establish a practical path to realizing the promise of utility computing. The idea of utility computing is not new [3, 4], and the idea of using large-scale computers to support utility computing is not new [5]. However, practical realization has yet to be achieved. In a prior publication [2], we briefly described the exploration of a global computational system on which the Internet could be viewed as an application. In this paper, we present a commerce-centric vision of utility computing, a corresponding system model, and a prototype. The model comprises four components: 1) resource principals, 2) nodes, 3) control channels, and 4) communication domains. We are developing a prototype based on the Blue Gene/P system that lets us explore the feasibility of the model for providing the building blocks for utility computing, and we present three Internet-style usage scenarios built on the prototype.

Key aspects and related goals associated with our approach include the following. First, we must acknowledge the Internet as the current model of global computing. A practical, global-scale computational system must provide a migration path for the Internet and be able to support its salient features. Second, we must enable commerce by supplying the system with primitives for distributed ownership as well as for

0018-8646/09/$5.00 © 2009 IBM

consumption-based usage via metering and billing. A global system must enable the creation and trading of value and ensure the ability for cooperation and competition within the system. Third, we must provide open building blocks with respect to hardware access, specifications, and example construction via open-source software. Success of a global system will come from enabling the ingenuity of others. Fourth, we must focus on provisioning metered, low-level, simple, scalable hardware-based building blocks. A global system should constrain software as little as possible to enable diversity and a wide range of future solutions.

The transformation of computation into a basic infrastructure for humanity is more than just producing an easy-to-access high-performance computing resource. It requires addressing some fundamental social and political questions. Many of the critical issues were identified and predicted by early work in utility computing [6–8]. We identify a few key challenges with the following three quotations from John McCarthy [7]. First, we must consider the issue of what we call *monopolies of ingenuity*. McCarthy writes that "the main danger to be avoided is the creation of services of limited scope...." Second, we must consider *monopolies of scale*. McCarthy writes, "Another problem is to avoid monopolies; the intrinsic nature of the system permits any person who can write computer programs to compete with large organizations in inventing and offering imaginative services, but one can worry that the system might develop commercially in some way that would prevent that. In general we should try to develop information services in such a way as will enhance the individuality of its user." Third, we must consider *monopolies of service*. "The major force that might tend to reduce competition is the exclusive possession of proprietary programs or files. Therefore it is desirable to separate the ownership of programs performing services from the ownership of the service bureaus themselves...."

The Kittyhawk technical design attempts to make these issues explicit in order to make the provisioning of computation a basic and accountable operation. Specifically, our goal is to show that it is feasible to separate metered hardware capacity from all layers of software, including virtualization and operating systems. We argue that this leads to a number of advantages.

Enabling creativity

No system-imposed software restricts the kinds of services that can be developed. This leads to an open, publicly accessible hardware platform with no software or usage restrictions, in contrast to closed private systems that restrict direct hardware access or can only be used through system-imposed firmware, hypervisors, operating systems, or required middleware. The only goal of an open hardware system would be to support the provisioning of hardware capacity so that others may provide a diversity of products and services, rather than provisioning of capacity as a secondary goal of utilizing excess resources of a system serving other primary purposes. The owner of a closed system, which can be used for proprietary services while also providing publicly available capacity, has little incentive for only providing hardware capacity and will typically have implicit control and interests that potentially conflict with those of the public services. This control can be exerted in ways ranging from low-level technical controls, such as biasing virtual scheduling priorities in favor of the proprietary services, to simply refusing access to competing groups and individuals.

Open and flexible scaling

In a truly open hardware infrastructure, the provider is not biased to limiting capacity based on the requester's identity or purposes. In contrast, the provider's goal is to maximize usage by customers, whoever they may be. In this way, questions of resource limitations and access become economic, social, and political issues for all of us to become engaged in. Our goal is not to argue whether we should approach these issues through auditing and regulation, free-market based mechanisms, or some form of hybrid, regulated public utility. Our goal is to make these issues explicit, so that all options can be explored, by isolating the hardware capacity and exposing it as a commodity. In such an environment, no single provider of software or data services is beyond competition or validation. Given today's software techniques, the actual limiting factor in implementing functionally competitive services is access to scalable numbers of general-purpose networked processors, memory units, and storage devices. Given that the hardware capacity is independently provisioned, its resource counts and limits can be made public either through regulation, pricing, or both. Thus, the decision as to whether additional resources are needed in the system becomes a public one. Scarcity of resources can manifest itself either in higher price or in the inability to satisfy all requests for service. Either way, it can be a cooperative and open decision process by entrepreneurs, public overseers, and providers. Entrepreneurs may compete with service offerings, which will reduce prices. Public regulation can concurrently allocate additional funds, to increase the system resources, whenever regulation limits the providers' ability to increase system capacity. Additionally, the provider need not acquire the burden of speculatively over-provisioning the system, because risk and resource limits can be expressed through price and auditing.

A real and transformable commodity

A hardware commodity can be independently audited, priced, and regulated, as society deems appropriate. A hardware base can provide an underlying commodity that can be audited and used to compare software-based service offerings. Given that the price of the hardware commodity is known, competitive service offerings can be compared with respect to price and function or with respect to hardware commodity usage (which may be made public by regulation). It also avoids the perils associated with basing our digital economy on implicit financial derivatives that can be manipulated on purpose or by accident. Virtual time, virtual resources, and other secondary units such as business operations can all serve legitimate pricing roles, but unless they are based on an actual and accountable commodity they lack the openness needed to avoid manipulation. To better understand this, consider that a virtual machine may be considered more as a promise to execute than a guarantee, and measuring the real time that a virtual machine has run says little about the actual resources consumed. That is not to say that virtual machine techniques do not offer actual value. For example, a virtualization service may increase resource utilization and may thus be able to provide resources less expensively (although perhaps less predictably) than a non-virtualized system. However, without an auditable underlying resource, virtualization may not prevent manipulation of price or function. To our knowledge, there is no single virtualization or scheduling technique that is suitable for all situations, with various types of workloads and competing priorities, and thus the common point for comparison enabled by an underlying hardware commodity can have great value.

Systems that either do not take into account these issues, or address them implicitly, have associated social and political risks and ramifications that may prevent their wide adoption. What is worse, they may be widely adopted and then used as a tool of oppression, as Parkhill [8] warns. Provisioning the utility will itself require commitment to responsible stewardship, integrity, and societal trust. Successful global computation will demand the trustworthiness and reliability we associate with personal banking, and the transparent and ubiquitous accessibility we associate with electricity and water utilities in the developed world. Whether such a steward exists, and whether we would trust it, are open questions. That is why it is imperative that we openly debate these questions rather than implicitly allow the system to evolve unchecked and affect our socioeconomic fabric. McCarthy points out that the development of these systems is inevitable unless regulation disallows it, and from this perspective the purpose of our system is to be a counterpoint to the proprietary systems of the past as well as the present, and to serve as a catalyst for debate and study in global computers.

Whereas great responsibility will be placed on the service provider, there will also be great potential for sustained revenue growth, in proportion to the worldwide demand for service. The aim is to embrace the relationship between commerce and computing, and the trend toward global scale for both. As discussed, we propose a path that enables pervasive and universal use of computation at a global scale by way of practical building blocks, with the goal to foster an open, sustainable, and efficient global computing infrastructure for humanity. As such, the system must have a sustainable economic model. The goal is not to create a vendor-specific cloud computing system in order to sell only the value of consolidation, but rather create a sustainable environment in which many parties can compete and cooperate in the creation and trade of digital goods and services.

The remainder of this paper is organized as follows. In the next section, we present the system model we adopt in order to implement the vision of a flexible, scalable, and extensible computing platform. Next, we discuss our prototype implementation, which makes use of the scalability of the Blue Gene/P system. We describe the functionality provided and several use cases that exemplify how that functionality can be used to build extensible and scalable services. This is followed by a brief overview of the related work that has inspired much of our activity. In the last section, we discuss the status of the work and our conclusions.

Kittyhawk and cloud computing

Three main aspects are associated with cloud computing. The first is the mass adoption of virtual server computing technologies to construct “hardware as service” offerings from large pools of PC-based commodity hardware. The second is the desire to outsource hardware ownership, utilizing networked computing resources to host infrastructure remotely, thereby isolating hardware infrastructure and its management (purchase, operation, and maintenance) as a separate, third-party product. The third aspect is to provide the basis for service-oriented computing models in which software is utilized to construct service products that are purchased and available on the Internet and hosted on third-party hardware infrastructure. The approach we take in Project Kittyhawk is to explore a fine-grain stratification that focuses on creating a fundamental environment for creating cloud computing offerings.

Ultimately all computing products must utilize some real physical computing resources that have attendant physical costs (materials, space, and energy). In Project Kittyhawk, we attempt to make this fact explicit and transparent so that real resources can be the building blocks for construction, auditing, and billing. Server virtualization is a technology that provides services beyond just a standardized compatible hardware interface. Virtualization has inherent scheduling and overcommitment policies and management capabilities. Thus, rather than conflating virtualization with the lowest-level offering, we attempt to make configurable hardware the basic unit of resources for service construction. As discussed, this allows virtualization and hardware to be isolated offerings, open for competition, auditing and comparison.

In order to permit cloud computing where predictability, overcommitment, and isolation of hardware are choices to made based on cost and need, we are exploring how to advance hardware to be better suited to cloud computing. We need not rely solely on virtualization to turn 1980s-based computing technology into the building blocks for cloud computing.

System model

In order to allow all additional value created to be diverse and open to competition, our system model focuses on providing simple, low-level primitives for building applications by others. In our view, the job of the hardware is to provide a set of building blocks that can be utilized by others to construct value in any manner they see fit. The goal is not to provision all capabilities, or eliminate all inefficiencies, but to allow them to be identified and addressed by others. It is critical that the ability to acquire resources be simple so that any application can scale and grow. Diversity and competition should be encouraged by allowing all software layers to be provided by others.

As mentioned, our model provides a low-level system with open interfaces. Basic examples of use are provided, demonstrating that resources can be acquired with ease.

Revisiting utility computing

Rather than base our model on a hardware system that enforces a specific hardware model, such as a cluster or a shared-memory multiprocessor architecture, we have chosen to explore hardware systems that are scalable hybrids, that is, machines that are consolidated but do not fix the communication model to a shared-memory primitive. We believe that hybrid systems are better suited to utility computing, as their fundamental resources are naturally thought of as homogeneous communication, computation, and memory in physical units that can be allocated, composed, and scaled.

We expect that, in time, resource usage will be based on standard measurement units that will be cooperatively established. These units will permit the metering and trade of the hardware “commodities” independent of what they are used for. For the moment, however, we believe it is important to begin with some simple metering models and units that can evolve. Although not discussed in any detail, a basic assumption in the remainder of our discussion is that all resource usage will be metered and billed.

Storage

In order to isolate value propositions, we view storage for data retention and archiving to be separate offerings that can utilize storage located remotely or locally. It is clear, however, even in our simple experimentation, that the notion of storage can and will evolve quickly beyond traditional models that have hard boundaries between memory, communication, disks, and software. Scalable hybrid data systems that seamlessly provide data retention properties and availability by trading off locality, density, redundancy (both physical and logical with respect to the data itself), and communication costs are natural in a global-scale system. Rather than architecting system-provided solutions, we suggest that storage is a key area in which many people will innovate, and that fixing a storage model would only serve to stifle competition and ingenuity.

Resource principals, nodes, communication domains, and control channels

We propose a set of system primitives, accessed via admission control, management, and hardware configuration components. The primitives serve as building blocks on which others can construct computational environments. As mentioned in the Introduction, the primitives are resource principals, nodes, control channels, and communications domains.

Figure 1 illustrates our system model. It identifies the building blocks listed above and shows how they can be organized to form two related topologies: control topologies and communication topologies. The following subsections describe each building block in turn.

Resource principals

To ensure economic viability, all resource usage must be tracked and metered and charged to an owner. We refer to owners as *resource principals* or simply *principals*. The front-end service interface of the system uses the component Admission Control, as illustrated in Figure 1. This component establishes and manages the set of valid principals. For any resource allocation, a provider views a principal as the owner. The component Management Services provide to principals the facilities to acquire resources from the system. The allocated resources are configured as specified by the principal and usage is charged to the allocating principal. A standard, secret-key system is used to validate principals. The system makes

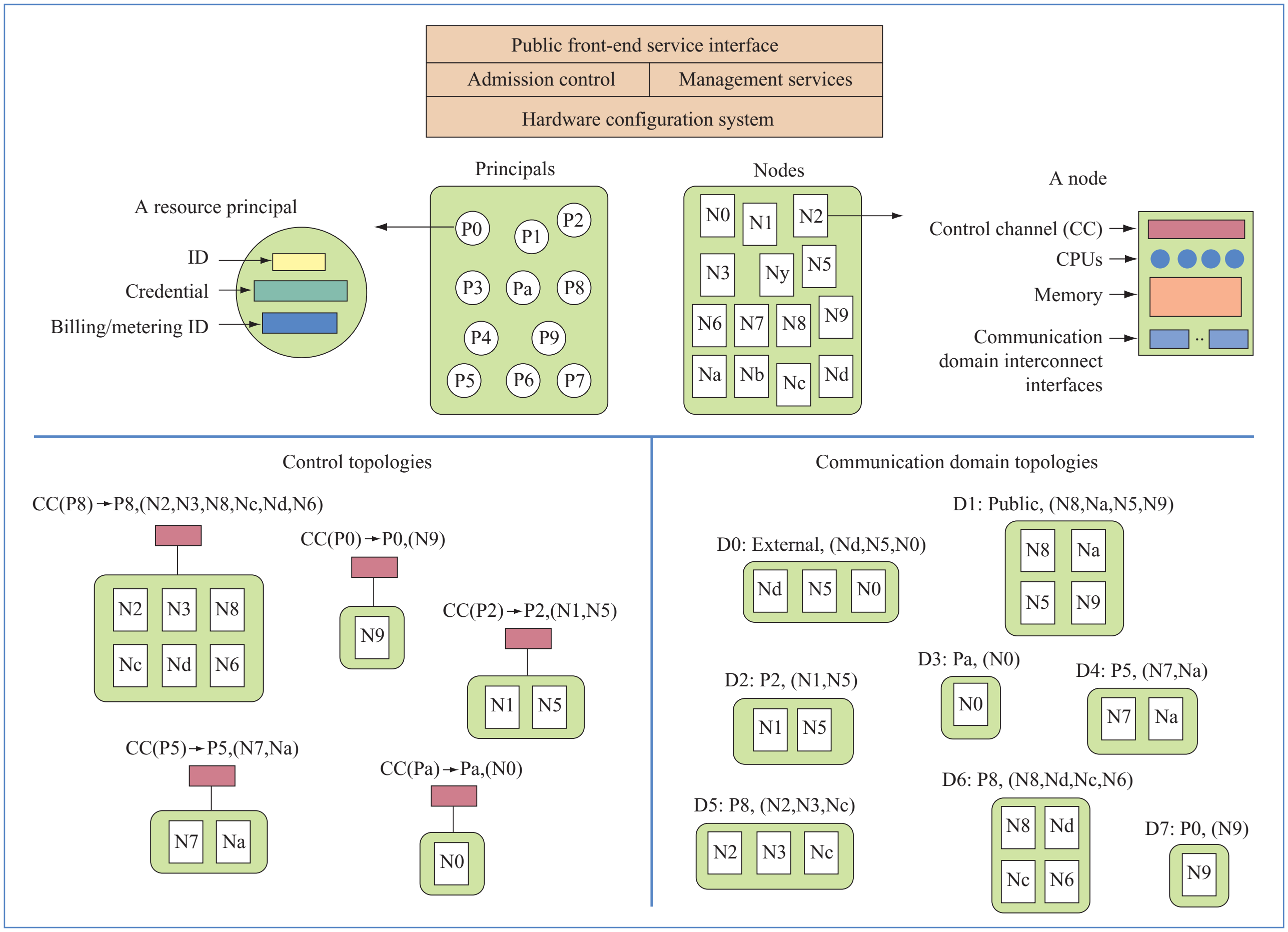

Figure 1

Conceptual illustration of the system model. The right arrow symbol indicates that the control channel provides the principal, P, with control of those nodes. (D: domain; N: node; CC: control channel.)

no restriction on how the principal uses the resources it owns. The system negotiates a unique ID for each principal, a credential in the form of a secret-key, as billing identity.

Nodes

Nodes serve as the basic unit of hardware resource that a principal can acquire. Nodes are comprised of a documented physical combination of CPUs, memory (uniformly accessible to all the CPUs of the node), and communication interfaces. The communication interfaces are divided into two categories. Each node is configured with a control interface that provides the owning principal access to the node via an encrypted channel. All other interfaces are optional and depend on the configuration specified by the principal. A unique interface is configured for each communication domain to which the node belongs. The front-end service interface to the system, using the admission control and hardware configuration components of the system, configures and assigns nodes to a principal. After configuration, the system has no further interactions with the node other than management operations requested by the principal (e.g., change of configuration, deallocation, or reset).

Control channels

When a principal acquires a set of nodes, components Admission Control, Management Services, and a Hardware Configuration System allocate and configure a unique control channel for the allocated nodes. This control channel provides basic access to the nodes to the owning principal. Principals can transfer the credential for the control channel, established during allocation, to other parties. The control channel itself is a lowbandwidth console-like component that enables remote access and control of the nodes by the principal. As is true of all components, the interface to the control channel is open and public, so that anyone can write software to interact with the control channel. The provider is responsible for pre-loading the nodes with initial, simple, replaceable software that the principal can use to manipulate the allocated nodes. The principal may load any software onto the nodes via the control channel, possibly replacing the low-level boot-loader and importing customized applications, such as security models and communications protocols, over the control channel.

Vendors may offer additional value in the form of preconfigured nodes with any firmware, hypervisors, operating systems, applications, etc. It is therefore critical that the hardware is well documented, and basic software is provided in open-source form. The software should be of high enough quality that it can be used as-is to support value-added products and services.

A critical requirement for the system model is that it provide good support for both single- and multi-node usage scenarios. Control and management of, and interaction with, a single node should be simple and efficient—encouraging use and software development—and the transition to environments with multiple nodes should also be simple and straightforward. Key to this seamless transition is a secure, flexible, and scalable control channel. It is not, however, necessary that the control channel have the best interface or make use of the most efficient protocol, as clients are many and diverse, with requirements and preferences that cannot be predicted. Rather it is more important that the hardware specification be clear, that the system support scalable parallel communication, and that the software that utilizes the control channel be replaceable so that others can provide custom control protocols and management systems. It has been our experience that some form of flexible hardware multicast method is desirable.

The lower left portion of Figure 1 illustrates a number of node allocations and associated control topologies. Each allocation groups the allocated nodes with a unique control channel. If more than one node is in an allocation, then all the nodes share the same control channel. The system must provide a portal to the owning principal of the allocation so that the principal can access the nodes via the control channel in a secure, authenticated fashion.

The control channel provides principals with access to resources without imposing restrictions on how these resources are used. It provides the means through which higher-level models of resource representation (e.g., 9p-based protocols [9]) can be constructed.

Communications domains

In our model, a communication domain is a set of nodes that are permitted to communicate with one another. A node may belong to more than one domain. Each domain to which a node belongs is manifest on the node as an interconnect interface, whose implementation is unspecified. However, we assume that the consolidated nature of the system will result in interconnects that support communication modes akin to integrated system buses, thus enabling a wide range of protocol and communication models to be explored. It is required that the interconnect and per-node devices be open and clearly specified, so that the device drivers/interface software to be developed is not constrained, enabling communication models and protocols ranging from Ethernet and shared memory to Transmission Control Protocol/Internet Protocol (TCP/IP) and Message Passing Interface (MPI).

Flexible communication topologies are critical to the ability of the system to reflect the technical, social, and financial relationships that a global system must support. We propose that at least three basic types of communication domains be supported. First, a *private* communication domain is one created by a principal and in which nodes can communicate only with other nodes within that domain. A principal may create any number of private domains; an acquired node may be placed in one or more of these domains. Given the specification for the communication device, the user may program the device to use any protocol. The system's hardware must enforce these domains without relying on software cooperation. From the perspective of the software, domains form a physical communication topology to which nodes can be added and removed via requests to the system service interface. As with all resources in our system model, software may be used to virtualize the communication domains.

A second type of communication domain is *public*. The public communication domain is a single system-provided common communication domain to which a principal can add acquired nodes. This domain serves as a common public communication domain. Nodes that communicate with one another must agree on the protocol to be used.

Third, the *external* communication domain is a single system-provided domain that connects nodes to the outside, or external, world, thus allowing selected nodes to communicate with the outside world by way of a specific network protocol such as TCP/IP. Communication in this domain is restricted to the protocol technology provided for external communications. A principal may request that specific nodes be added to the public and/or external domains.

The lower-right-hand portion of Figure 1 illustrates example communication-domain topologies configured

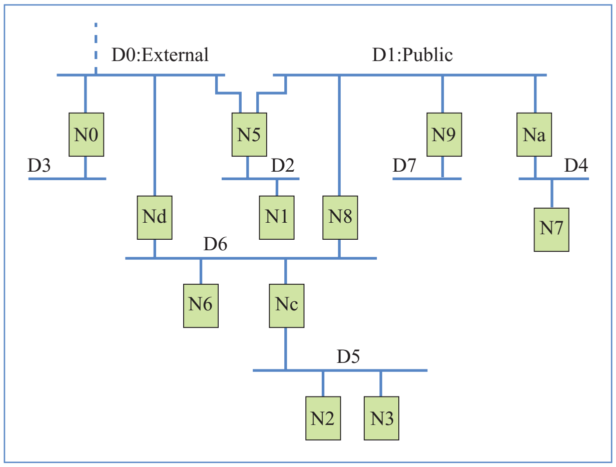

Figure 2

The communication domain topology of Figure 1 as a network-style topology.

for a set of allocations. Each domain is labeled with a unique identifier. The external and the public domains are D0 and D1 respectively. Each private domain has an associated owning principal and set of nodes. **Figure 2** illustrates the same topology as a network-style topology.

Scalable, flexible, and efficient support for communication domains is critical to our system model. We suggest that the ability to scope communication is essential to supporting competition and cooperation. Our system model can be utilized to flexibly support arbitrarily complex Internet (as well as intranet) networking. In our system model, traditional networking hardware, such as switches, bridges, routers, and sprayers, are implemented as software components that can be run on appropriately configured nodes. However, future usage models may avoid such components by implementing new communication and usage models.

Metering

Resources of all types—processor, memory, network and storage—are constantly monitored and resource usage is debited to the resource principals by way of trusted system interfaces. These interfaces provide information necessary to enable software to make educated decisions about resource usage and then optimize cost.

We expect that different pricing models (such as fixed, variable, and volume pricing) will develop over time. A trusted non-forgeable mechanism for metering resource usage is a key requirement for commerce for any pricing model. Increasingly accurate metering fosters optimization. The energy market is a good example; whereas households typically pay a fixed price per kilowatt-hour independent of the time of day, industrial energy is often variably priced, an incentive for businesses to move energy-intensive tasks into night hours. However, only accurate metering over short time intervals enables such variable pricing.

Prototype

To help validate our model, we are constructing a prototype. In this section, we describe the prototype and some of the scenarios we have implemented on it. We begin by summarizing our Blue Gene/P hardware platform. Additional details can be found in Reference [2].

Blue Gene/P

A Blue Gene/P node contains four 850-MHz cache-coherent IBM PowerPC* microprocessor cores in a system-on-a-chip with dynamic RAM (DRAM) and interconnect controllers. The nodes are physically grouped onto node cards, 32 nodes per card, and 16 node cards are installed in a midplane. There are 2 midplanes in a rack, providing a total of 1,024 nodes and, in its current configuration, a total of 2 terabytes of random access memory (RAM). Multiple racks can be joined to construct an installation. The theoretical hardware limit to the number of racks is 16,384, which would result in an installation with 16.7 million nodes and 67.1 million cores, with 32 petabytes of memory.

Additionally, each node card can have up to two I/O nodes featuring the same basic unit but with an additional 10-Gb Ethernet port. Hence, each rack has an external I/O bandwidth of up to 640 Gb per second; the aggregate I/O bandwidth of the maximum installation is 10.4 petabits per second. It is important to recognize that the nodes themselves can be viewed for the most part as general purpose computers, with processors, memory, and external I/O.

The interconnects seamlessly cross all the physical boundaries (e.g., nodecards, midplanes, and racks), at least with respect to the software, once the machine is configured. There are four relevant networks on Blue Gene/P: (1) a global secure control network; (2) a hierarchical collective network (which provides broadcasts and multicasts); (3) a three-dimensional torus network; and (4) a 10-Gb Ethernet for external communication.

Service interface, admission control, management, and hardware configuration

In order to ensure a scalable system, we have been designing the components of our prototype to be completely self-hosted and distributed. A set of nodes are used to provide the front-end service interface—which involves admission control, management services, and the hardware configuration system—and external connectivity. At this time we have only a small installation and require only a small set of resources to manage our limited hardware. However, the design goal is to ensure that as the system scales, a constant fraction of the additional resources are used to scale the infrastructure components. One cannot design a global system that requires external resources for its base management.

We have constructed a simple, programmable, command-line front-end service interface. The goal is to determine a minimal level of functionality and expressiveness that is sufficient to support additional services. It is important to keep the interface simple, programmable, and efficient so that other more advanced interfaces and management systems can be constructed on top to meet specific technical or business needs. We describe below in some detail the service interface of our prototype.

The main primitive is a simple allocation command, `khget`. To allocate resources one must run `khget` on a system that has TCP/IP connectivity to our Blue Gene* prototype. We have constructed a simple set of components that run on externally connected nodes reserved for the system infrastructure. Currently most components reside on a single node and implement in software the admission control, system management, and hardware configuration functions, to which we collectively refer as the management infrastructure. Although simple in structure, they are designed to be iteratively duplicated in order to enable the distributed management of large hardware installations. All I/O nodes are currently configured to form a distributed scalable external I/O network. A future step toward distributing the management infrastructure is to locate some of these components on the I/O nodes.

The key arguments to `khget` are listed in the following:

- **username**: an identifier used in conjunction with the argument credential (described next) to identify a principal.
- **credential**: an authentication key to be supplied by a principal on first use of `khget` for the corresponding username. The credential is recorded and any subsequent interaction with this username must include proof of the initial key. This feature is currently inactive.
- **number of nodes**: The number of nodes to be acquired (defaults to 1).
- **communication domain configuration**: a set of optional arguments that can be used to configure the nodes with respect to the communication domains to which they belong.
    - **-p n**: *n* private domains should be created for this principal, and all nodes allocated should be added to the domains. The principal is considered the owner of the new domains.
    - **-n a[,b,c. . .]**: a list of pre-existing domains (owned by the principal) to which the newly allocated nodes should be added.
    - **-I**: a node should be added to the internal public communication domain.
    - **-x**: a node should be added to the external communication domain for connectivity to the outside world.

In our prototype, a domain is identified by *netid*, an integer. By default, a private domain is created for every principal, and if no explicit domain configuration is specified and only the one private domain for the principal exists, nodes allocated by the principal are added to this domain.

`khget` returns a simple text object identifying the components of the new allocation. We provide a simple script to parse and operate on this object to aid in programming and automation. The object includes the following:

- **control channel**: The current version of the prototype identifies a control channel for the allocated nodes as an ssh (secure shell, a security protocol for logging into a remote server) user and host, such as `con34@khcc.research.ibm.com`. The principal can access the control channel and interact with the default software on the nodes by issuing an ssh command to this identifier. In the future, the key associated with the principal will be required for establishing a successful connection. Currently, however, we are not enforcing this requirement. We chose ssh as the protocol for the control channel because it is widely used and heavily audited. There is no doubt that vulnerabilities in ssh have occurred and will continue to occur, but our belief is that we would do much worse with a one-off, home-grown solution. We use the hierarchical collective network for control channel traffic inside Blue Gene/P. This network guarantees reliable delivery of the messages (in hardware) and its broadcast and combining functions are a good fit for the control traffic.
- **netid list**: The list of communication domains to which the nodes belong. Domains are identified by their netids (a list of integers).
- **node list**: The list of nodes that were allocated, each assigned a unique node number.

In the following, we discuss examples of khget use. One such example is *khget -c jonathancred jonathan*. If this is the first time the principal identified by the pair (*jonathan, jonathancred*) is being used in khget, admission control initializes it as a new principal. Additionally, (1) If this is the first allocation for this principal, a new private communication domain is created and assigned to the principal. (2) A single node is allocated to the principal and configured to be a part of the domain. (3) A new control channel is created and the node is configured appropriately.

Finally, the node is removed from the free pool and released to be managed by the principal. The node is preloaded with default software, which the principal can use for initial interaction with the node. Our prototype currently preloads the node with an open source boot-loader (U-Boot) and a Linux** kernel [10]. As a convenience we pre-configure the U-Boot environment so that when booted the configured communication domains can be utilized as Ethernets, with the device drivers we have added to the Linux kernel. From this point, the user can access the node via the returned control channel (by performing ssh with the identified host as the identified user). At this point the principal can utilize U-Boot to load arbitrary software onto the node and boot it as desired. We describe specific usages in the section on examples, below. It is important to note the principal is free to overwrite anything, even U-Boot itself, with customized low-level software that can implement either simpler or more complex function, including alternative security models.

The invocation of *khget -c jonathancred jonathan 100* is similar to the first invocation, except that 100 nodes are requested, instead of just one. In this case, the default behavior of the control channel is to broadcast all communication to all nodes in the allocation set. For example, if the principal sends to the control channel the string *mrw 0x10000000* 1024, and the default U-Boot is still executing, all U-Boot instances execute the command mrw with arguments 0x1000000 and 1024. This command (which we added to U-Boot) takes *n* bytes (in this case 1,024) from the console channel and writes them to the specified memory location (in this case 0x10000000). At this point, the next 1,024 bytes sent to the control channel are written to the memory location 0x10000000 on all nodes. Because this communication is implemented on top of hardware-supported broadcast, the performance scales with the number of nodes. Although the support added to U-Boot for our control channel is simple, we have discovered that to have a simple, scriptable way to manage hundreds or thousands of nodes in parallel is a very powerful feature. Clearly, considerably more complex control environments can be constructed, either by extending U-Boot to be more fully aware of the broadcast nature of the control channel, or by using altogether new control software. We have also added support in Linux for a console device driver that operates on the control channel. At this point the driver is rather simple, but in this case also we have found that a secure, scriptable, broadcast control channel can be the basis for a well-managed computational environment built out of standard software. It is simple to write scripts that broadcast commands to all nodes of an allocation, causing, for example, all nodes to configure an Ethernet interface and mount a file system from a server on that Ethernet. While performance is not the main focus in this paper, the fact is that the broadcast control channel makes the booting and managing of thousands of nodes not much more difficult than managing just one node.

The command *khget -x -c jonathancred jonathan 20* allocates and configures 20 nodes for principal (*jonathan, jonathancred*). In this case, the -x ensures that the nodes are also part of the external domain and can communicate with the external world. Although, ideally, we would like most communication to migrate to internal domains on public and private networks inside the system, it is clear that external communication is required to interact with the rest of the world, migrate function, and access external data sources. In our prototype, the -x ensures that the nodes have access to the Ethernet connected to the rest of our corporate infrastructure. In this way, principals can construct services that have external connectivity. This connectivity can be used both for providing services to the external world and for accessing services, such as storage, in the external network.

The command *khget -i -n 128,45 -p 1 -c jonathancred jonathan 1* creates a node configured to be part of four communication domains: the public internal domain, existing private domains 128 and 45, and a new private domain created for this allocation (-p 1). If all of these domains are used as Ethernet domains, such a node could act as a bridge or switch at the Ethernet level or as a gateway at the TCP/IP level (assuming TCP/IP use). Given, however, that arbitrary software can be run, it is also possible that the new domain will be used by nodes in the domain to run a custom tightly coupled protocol designed to implement a shared computing environment, and that this node is to act as the external connection (in this case to the public network and networks 128 and 45) for this environment.

The current khget is only a prototype and clearly needs to be extended to incorporate additional information, such as billing and metering credentials. Its purpose is primarily as a vehicle for experimentation with a scalable and programmable environment.

Example scenarios

In this section, we describe three scenarios using the infrastructure described above and open-source software. We believe that a key feature of the system model is its ability to make use of a large body of open-source software, which can serve as building blocks in developing useful services. The actual value of the model will be realized as others begin to construct environments that we have not designed for, or even thought about. In order to explore the model and establish a body of open-source software for the system, we have focused on examples that provide Internet-like functionality. We show that arbitrary communication topologies can be constructed and that scalable consolidated hardware can be used to support standard environments in a utility computing model with the notion of distributed ownership.

We begin with a brief summary of the usage model and elements common to all scenarios. Following the allocation of a set of nodes, the user is directed to a publicly accessible machine and uses the credentials provided at registration time to access a console channel to the allocated nodes. Initially the nodes are running the open-source firmware, U-Boot. From here a user can reload the entire memory content of the nodes and restart the nodes. We specify how to write software that communicates on the console channel and communication domains in addition to the standard documentation necessary to write software to manage the nodes.

In addition to U-Boot, we also provide a version of the Linux kernel with device drivers for the communication domains, which enable a domain to communicate using Ethernet protocols. When using this software, each communication domain appears as a separate Ethernet network.

Using this software stack, we can build arbitrary TCP/IP network topologies and Internet examples using the built-in networking capabilities of Linux. A user may also write custom software to perform these functions. We create examples in which small organizations each have a private Ethernet and one or more gateway machines to the public communication domain, allowing them to interoperate.

While the standard Internet examples may be what people immediately prefer to consider, the system is not restricted to them. It is just as viable for someone to allocate nodes and run custom software that exploits the communication devices in different ways, such as running a more tightly coupled environment, in which the nodes form a single system image. Such a system could also use another communication domain to connect to other networks.

RAM disks as open source appliances

Using the `khget` interface, a principal can acquire nodes, and the control channel can communicate with the nodes, U-Boot, and Linux kernel to build self-contained scalable open-source appliances. These appliances are stateless and can be deployed and configured via the control channel.

Using U-Boot and the control channel, a principal can load arbitrary data into the memories of the allocated nodes. The load operations for the nodes take place in parallel. By loading the nodes with a Linux kernel and a RAM-based root file system, one can use the nodes to perform custom functions. We refer to such RAM-based root file systems as "software appliances." As in other approaches aiming at scalability—such as Bproc (Beowulf Distributed Process Space) [11] and Warewulf/Perceus [12]—our appliances are stateless.

Such software appliances form the basic building blocks for the construction of arbitrary, special, or general-purpose environments. We illustrate this approach by using appliances to create a farm of Web servers.

Because we run a standard Linux 32-bit PowerPC kernel, we can draw from a large pool of open-source software packages to construct our appliances. Both Debian** and Fedora** Linux distributions support 32-bit PowerPC. We developed a tool to automate the process of building these software appliances. The tool runs on a standard Linux installation and extracts and packages the necessary files to form a stand-alone root file system for a specific application.

Reference [2] presents the sizes of the software appliances our tools automatically constructed from a typical Linux root file system of approximately 2 GB. For example, our shell appliance has 3 MB, and a Ruby-on-Rails** Web site appliance has 12 MB. The appliances are generally 5% of the size of the full root file system.

Listing 1 illustrates the commands necessary to: 1) Acquire a node with external connectivity, 2) Load the Linux kernel (packaged as a U-Boot image), 3) Load the apache "software appliance" (packaged as a compressed cpio root file system image as supported by the Linux kernel), 4) Boot the node by writing the U-Boot command `run kboot` to U-Boot via the control channel, and 5) Query the Web server running on the booted node.

`khdo` is a simple script we wrote to parse the return information from `khget` and perform standard tasks such as the U-Boot commands necessary to load a kernel and RAM disk and write specified strings to the control channel. In addition, `khdo` provides a way to iterate over the lists that `khget` returns (e.g., `peripcmd`, which iterates over the IP addresses returned by `khget`). As a convenience, our prototype currently provides IP addresses for each node.

Listing 1 Shell commands that allocate, load, launch, and query a Web server.

```
Shell commands that allocate, load, launch, and query a Web server.
> nodes=$(khget -x $USER 1)
> echo "$nodes" | khdo loadkernel ./uImage
> echo "$nodes" | khdo loadramdisk ./apache.cpio.gz.uimg
> echo "$nodes" | khdo write "run kboot"
> echo "$nodes" | khdo peripcmd "wget http://%ip%/"
```

This scenario illustrates the power of the model in automating the construction of services. We could generalize these commands in a shell script that parameterizes the appropriate values. Also, simply replacing the value of 1 in the `khget` command to 100 would launch 100 Apache instances with no other changes. Because of the broadcast nature of the control channel and self-contained nature of the appliances, scalable performance is ensured. The approach can be used to build more complex environments, including environments in which the nodes cooperate to provide a service. Note that if a user were to launch as many as hundreds of thousands of web server appliances, the resultant allocation would look flat rather than hierarchical. The multicast nature of our allocation scheme combined with software appliances that run entirely in RAM allows us to avoid hierarchical boots.

A principal can construct an intranet by using communication domains. Network configurations and management functions can be automated through simple scripts. We have used the scenario described here as a building block for more complex environments.

Figure 3 shows the network topology for this scenario. The service interface, from which resources can be requested, is labeled `SrvIface` in the figure. The service interface can be accessed either from the external network or from the internal public network. A machine configured to invoke `khget` must have network access to the service interface. All other components in Figure 3 represent resources that belong to the intranet owner. We now describe the various components used in constructing the example scenarios, including the commands used to allocate and interconnect these components.

Gateway (GW)

This node, which serves as a gateway for the private intranet, is allocated with the command *gw=$(khget -x -i -p 1 -c ./corp1cred corp1 1)*. The command also establishes connectivity to three communication domains: (1) the external domain, (2) the internal public domain, and (3) a new private domain to serve as the main internal network, which we refer to as `pnet`. The node acts as a gateway to the external and public networks for the nodes and associated services on the private network (`pnet`). On this node we run a Linux kernel and an appliance based on the Debian PowerPC 32-bit distribution that contains the necessary tools for the gateway function. The communication domains are defined on this node as three Ethernet interfaces (`eth0`, `eth1`, and `eth2`) using the network drivers we have developed.

File Server (FS)

This node acts as a traditional network file server for the internal network. It is allocated by the command *fs=$(khget -n pnet -p 1 -c ./corp1cred corp1 1)*, which allocates the node and establishes connectivity to two communication domains: (1) the main internal network

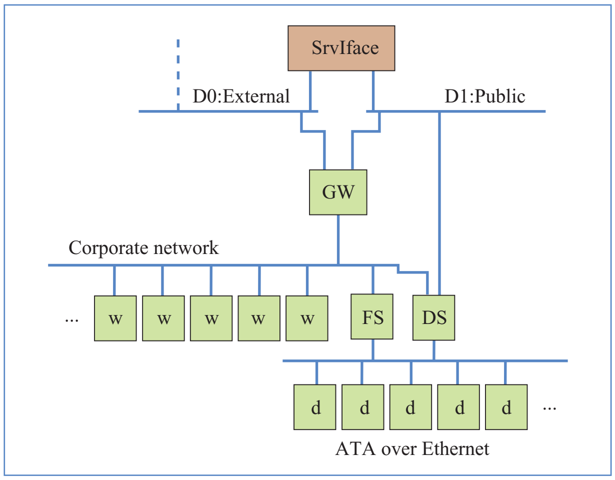

Figure 3

Example of a private Intranet. [SrvIface: system service interface (khget server); GW: gateway; DS: disk server; w: worker; d: ATA over Ethernet (AOE) disk; AOE: ATA over Ethernet; ATA: AT attachment.]

(pnet) and (2) a new private network to host ATA-over-Ethernet (AOE) [13] storage, which we refer to as disknet (ATA stands for AT attachment, where AT originates with IBM Personal Computer/AT*). On this node, we run a Linux kernel and an appliance based on the Debian PowerPC 32-bit distribution that contains the necessary tools for AOE, LVM2 [14], NFS [15], and basic networking. The communications domains are again manifested on this node as two Ethernet interfaces (eth0 and eth1).

Disk Server (DS)

This node provides a new and unique service for constructing AOE "disks nodes," developed specifically for the prototype system. It is allocated by the command *ds=$(khget -i -n pnet,disknet -c ./corp1cred corp1 1)*. On this node we again run a Linux kernel and another Debian-derived appliance that contains AOE, basic networking, and the utilities needed to interface with the system service interface (khget). The appliance also contains two custom components developed for this example: 1) a Debian-derived "vblade [16] appliance" that contains the AOE vblade server and a startup script to launch the vblade server exporting the RAM of the node as an AOE disk; and 2) a simple shell script (khgetdisk) that acquires nodes, loads the vblade appliance and an appropriately configured Linux kernel, and boots the nodes. The script also manages the AOE device namespace, allocating names and configuring vblade instances appropriately. The node is allocated to be part of three communication domains: 1) the public internal network (in order to have access to the system service interface), 2) the main internal network (pnet), and 3) the private disknet network in order to create and manage the "disk nodes." The node does not actually need access to the public network, as it can use the gateway node to reach the service interface. This configuration was used primarily to illustrate flexibility rather than to adhere to a specific networking policy. Similarly, it is not necessary to use a separate node for this function, but we wanted to explore the flexibility of using nodes to provide reusable service components.

AOE disk (d)

These nodes are created by the disk server via khgetdisk and act as RAM-based AOE disks. The file server accesses these nodes as AOE disks, on disknet, and configures them into logical volume groups, logical volumes [14] and file systems. Thus, file systems spanning several disks are created and exported on the main private network using Network File System (NFS). These nodes are added on demand by invocations of khgetdisk [*n*] where *n* is the number of disks to construct. The file server can use the standard features of LVM2 (logical volume manager 2) [14] and ext3 [17] to enlarge existing file systems or create new ones from the additional disks.

Workers (w)

These nodes, which act as servers, are allocated by the command *khget -n 3 -c corp1cred corp1 [n]*, where *n* is the number of workers. These nodes are loaded with whatever appliances they need to run. These workers can mount the file systems exported by the file server.

This scenario illustrates the following three properties. First, with a simple, scalable, programmable system interface for node allocation and communication domain construction, users can script the creation and population of entire network topologies. The scripts can be parameterized to form reusable building blocks. Second, an easily accessible programmable service interface permits the on-demand scaling of service components. In this case the pool of disk nodes and workers can be easily scaled based on demand. Third, the programming effort required to make use of a large-scale parallel proprietary system such as Blue Gene can be reduced through the use of open-source software. In this scenario, the only new software components (beyond the device drivers we added to the Linux kernel) were the simple shell scripts required to integrate existing open-source software components.

The role of this scenario is not to establish an ideal model for resource usage for a given function. Rather, we illustrate that familiar tools and software can be leveraged to initially construct and scale services. Although using collections of nodes as small disks to create larger storage units may seem wasteful, it illustrates the use of open-source software to build an initial environment that can be scaled. We are not precluding the use of external storage. Rather, we are showing that the high speed networks and open-source software allows for the construction of data repositories that have a familiar interface, in this case a file system. This approach allows the principal to make communication tradeoffs by copying/staging data in and out of the memory-based high-speed interconnected file system formed from the disk nodes. More sophisticated solutions may include simple uses and extensions of the logical volume manager (LVM) to include redundant arrays of inexpensive disks (RAID) and elements from external storage. It may also include altogether new distributed storage protocols and systems that more effectively utilize the memory, computation, and communications resources of the nodes while potentially exploiting redundancy in data and computation.

General-purpose servers

In the previous two scenarios, we used the software appliance approach described earlier to execute function on the nodes. In this subsection we discuss two approaches to running nodes as complete Linux servers.

The first approach uses the infrastructure described in the previous scenario. We use the file server to host an NFS read-only accessible root file system for a Linux distribution such as LVM28. We then launch worker nodes that use the NFS root capabilities of Linux to boot instances of the distribution. At start-up time we relocate the portions of the file system that require write access to a small RAM-based file system using standard Linux operations. The basic approach is described by Daly et al. [18].

The second approach is to use an AOE disk node to host the root file system for another node. In this case we allocate a node to act as an AOE disk, serving its RAM as storage. We populate the disk node with a copy of a distribution's root file system and then allocate another node to be booted using this disk. Additional storage is mounted from one or more internal and external file servers.

Either approach results in nodes that are fully functional Linux servers. We use them to host arbitrary services and user functions including remote Virtual Network Computing (VNC) desktops.

We have developed several other example scenarios using open-source components, including: distCC** [19] farms, Hadoop** [20] farms, and UNIX** development stations. These example scenarios are not intended to be exhaustive with respect to the set of things that can be constructed. Rather they are to illustrate that very little (beyond scalable hardware that can support communication domains and open-source software) is actually required to enable the exploration of the system as a building block for general-purpose computing.

All the scenarios discussed here have been built from readily available open-source software. The system also has the potential to enable new software and services as well as optimized versions of existing software. We conjecture that the ease of acquiring resources (at potentially very large scales), the ease of constructing and configuring communication domains, the benefits of high-performance networks combined with the open-source environment and the underlying metering support, will lead to innovation and the creation of new solutions. In some ways, this vision is not far from what super-computers such as Blue Gene have done for the high-performance computing domain. We are in the process of exploring the creation of environments in which nodes are running custom software stacks. Two specific ideas we are exploring are: (1) a distributed Java** Virtual Machine execution environment based on prior work [21] (in which specialized execution environments are hardware-based, without a hypervisor) and (2) a native Plan 9** distributed computing environment [22].

Related work

The idea of computation as a utility dates at least as far back as the early 1960s, when it was proposed by John McCarthy [4] and Robert Fano [6]. The standard analogy is to compare the delivery of computing to the delivery of electricity. In utility computing, computational resources are provided on demand and are billed according to usage. There is little or no initial cost, and computations with large resource requirements, sustained or peak, can be serviced without having to go through the lengthy process of acquiring, installing, and deploying large numbers of computers.

More recent examples of utility computing include grid computing [23] as well as cloud computing [24]. Both these approaches make large amounts of computational resource available on demand. Danny Hillis, creator of the Connection Machine, suggested that very large machines could be used to provide computing as a utility, much as large generators provide electricity to cities [5].

The analogy with electricity goes further than just production and distribution. Much like electricity is used by industry to power machines and create products that are then sold in the marketplace, computing power can be refined into higher-level services to be offered on-demand. We already see such commercial offerings, as companies use machines obtained from the Amazon Elastic Compute Cloud to power their own Web offerings [25]. Unlike electricity, however, the same distribution mechanism for the basic computing capacity (the Internet) can also be used for the refined product. Computing is probably unique in terms of the speed with which it can be transformed from one product to another product, ready for consumption.

Conclusions and current status

We have implemented prototypes for the first enabling layers of a global shared computational system. These layers provide a simple service, namely the ability to allocate and interconnect a large number of computational resources. These resources are provided in a relatively basic state. That is, we provide basic hardware functionality and the ability to load executable images. Higher-layer services can then use these basic primitives and build more functionality.

We used Blue Gene/P as the implementation platform. The characteristics that make Blue Gene/P ideal for this role are: its scalability, its fast and partitionable interconnect, and its scalable control system. Making use of these characteristics, we created primitives that allow a user to allocate sets of independent nodes and then connect these nodes to one or more communication domains. Using these primitives, and using standard open-source software, we implemented three example scenarios involving higher-level services. First, we created a scalable Web server using software appliances. Second, we created a private virtual data center, with worker nodes, file server nodes, and disk server nodes. Finally, we created a cluster of general-purpose servers. The examples have two characteristics in common. First, they were built using standard (open-source) software; no exotic technologies were required. Second, they scale easily, which is made possible by the Blue Gene technology and our basic services.

We hope that the work described here will encourage others to develop high-end services for this environment. We are already working with other groups in IBM to deliver stream processing services on top of the basic Kittyhawk services. We also have a good deal of interest from cloud computing users.

Many aspects of the prototype were designed with the focus on establishing feasibility; these aspects are not of production quality, and some critical functionality is missing. We plan to remedy these deficiencies as we gain more experience with the prototype. We also plan to explore improvements in hardware that better support our system model. We are currently exploring software device virtualization as a way to evaluate hardware changes required to support hardware-based communication domains.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of Linus Torvalds, Software in the Public Interest, Inc., Red Hat, Inc., David Heinemeier Hansson, Martin Pool, Apache Software Foundation, The Open Group, Sun Microsystems, Inc., or Lucent Technologies in the United States, other countries, or both.

References

1. J. G. Koomey, “Estimating Total Power Consumption by Servers in the U.S. and the World,” Final report, Advanced Micro Devices, Inc., February 15, 2007.
2. J. Appavoo, V. Uhlig, and A. Waterland, “Project Kittyhawk: Building a Global-scale Computer: Blue Gene/P as a Generic Computing Platform,” *Operating Syst. Rev.* **42**, No. 1, 77–84 (2008).
3. J. B. Dennis, “Toward the Computer Utility: A Career in Computer System Architecture,” written for the 50th reunion of the MIT class of 1953, Belmont, MA, August 2003; see *http://csg.csail.mit.edu/Users/dennis/essay.htm*.
4. J. McCarthy, “MIT Centennial Speech of 1961,” *Architects of the Information Society: Thirty-five Years of the Laboratory for Computer Science at MIT*, S. L. Garfinkel, Ed., MIT Press, Cambridge, MA, 1999.
5. W. D. Hillis, “The Connection Machine,” *Sci. Am.* **256**, No. 6, 108–115 (1987).
6. E. E. J. David and R. M. Fano, “Some Thoughts about the Social Implications of Accessible Computing,” *Proceedings of the AFIPS 1965 Fall Joint Computer Conference*, Spartan Books, 1965, pp. 243–247.
7. J. McCarthy, “The Home Information Terminal,” *Proceedings of the International Conference on Man and Computer*, Bordeaux, France, 1970, S. Karger AG, Basel, 1972, pp. 48–57.
8. D. F. Parkhill, *The Challenge of the Computer Utility*, Addison-Wesley, Reading, MA, 1966.
9. Bell Labs, 9P: A Simple File Protocol to Build Sane Distributed Systems; see *http://9p.cat-v.org/*.
10. Linux Online, Inc.,The Linux Home Page; see *http://www.linux.org/*.
11. E. Hendriks, “Bproc: The Beowulf Distributed Process Space,” *Proceedings of the 2002 International Conference on Supercomputing* (ICS 2002), June 22–26, 2002, New York, ACM, New York, 2002, pp. 129–136.
12. A. Kulkarni and A. Lumsdaine, “Stateless Clustering Using OSCAR and PERCEUS,” *Proceedings of the International Symposium on High Performance Computing Systems and Applications* (HPCS 2008), June 9–11, 2008, Quebec City, Canada, IEEE, New York, pp. 26–32.
13. S. Hopkins and B. Coile, *AoE (ATA over Ethernet)*, The Brantley Coile Company, May 2006.
14. LVM2 Resource Page; see *http://sourceware.org/lvm2*.
15. Linux NFS Overview; Sourcefourge.net; see *http://nfs.sourcefourge.net/*.
16. ATA over Ethernet Tools, Sourceforge.net; see *http://sourceforge.net/projects/aoetools/*.
17. S. Tweedie, “Ext3, Journaling File System,” *Proceedings of the Ottawa Linux Symposium*, June 26–29, 2002, Ottawa, Canada.
18. D. Daly, J. H. Choi, J. E. Moreira, and A. Waterland, “Base Operating System Provisioning and Bringup for a Commercial Supercomputer,” *Proceedings of the 21st International Parallel and Distributed Processing Symposium* (IPDPS 2007), March 26–30, 2007, Long Beach, CA, IEEE, New York, pp. 1–7.
19. M. Pool, distcc: A Fast, Free Distributed C/C++ Compiler; see *http://distcc.samba.org/*.
20. Apache Software Foundation, Hadoop; see *http://hadoop.apache.org/core/*.
21. G. Ammons, J. Appavoo, M. A. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. S. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski, “Libra: A Library Operating System for a JVM in a Virtualized Execution Environment,” *Proceedings of the 3rd International Conference on Virtual Execution Environments* (VEE 2007), San Diego, CA, June 13–15, 2007, ACM, New York, pp. 44–54.
22. E. Van Hensbergen, C. Forsyth, J. McKie, and R. Minnich, “Holistic Aggregate Resource Environment,” *Operating Syst. Rev.* **42**, No. 1, 85–91 (2008).
23. I. Foster and C. Kesselman, *The Grid 2: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, San Francisco, CA, 2003.
24. R. Buyya, C. S. Yeo, and S. Venugopal, “Market-oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities,” *Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications* (HPCC-08), Dalian, China, September 25–27, 2008, IEEE, New York, pp. 5–13.
25. J. Bezos, “Startup School 2008,” Omnisio, Inc.; see *http://omnisio.com/startupschool08/jeff-bezos*.

Received August 12, 2008; accepted for publication November 3, 2008

Jonathan Appavoo *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (jappavoo@us.ibm.com).* Dr. Appavoo is a Research Staff Member in the Advanced Operating Systems department at the IBM T. J. Watson Research Center. He received a B.S. degree in computer science from McMaster University in 1993, and M.S. and Ph.D. degrees in computer science from the University of Toronto in 1998 and 2005, respectively. In 2003, he joined IBM, at the Thomas J. Watson Research Center, where he completed his doctoral work on the IBM Research K42 operating systems. His work has focused on scalable systems software for large-scale, general-purpose multi-processors. His current research interest is in exploring structure in the complete execution of a modern computer including the stochastic interactions with the software, and exogenous events. He is interested in how the structure can be defined, quantified, and exploited. In 2007, along with his colleagues Volkmar Uhlig and Amos Waterland, he established Project Kittyhawk to explore conjectures about global scale computers and computation.

Volkmar Uhlig *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (vuhlig@us.ibm.com).* Dr. Uhlig is a Research Staff Member in the Advanced Operating Systems department at the IBM T. J. Watson Research Center. He received an M.S. degree in computer science from Dresden Technical University in 1995, and a Ph.D. degree in computer science from the University of Karlsruhe in 2005. He subsequently joined IBM at the Thomas J. Watson Research Center, where he has worked on x86 virtualization, Project Kittyhawk, and cloud computing. He is author of a number of scientific papers and patents. Dr. Uhlig is a member of the Association for Computing Machinery.

Amos Waterland *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (apw@us.ibm.com).* Mr. Waterland is a member of the Advanced Operating Systems department at the IBM T. J. Watson Research Center. He received a B.S. degree in computer science from the University of Oklahoma in 2002. He subsequently joined IBM, where he has worked on contracts for NASA and experimental operating systems. In 2004, he received an IBM Outstanding Technical Achievement Award for his work on embedded operating systems.

Bryan Rosenburg *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (rosnbrg@us.ibm.com).* Dr. Rosenburg received his Ph.D. degree in computer science from the University of Wisconsin–Madison in 1986 and has been a Research Staff Member at the IBM T. J. Watson Research Center since that time. At IBM, he has been involved with the operating system for the RP3 large-scale NUMA (non-uniform memory architecture) shared-memory multiprocessor, with a program visualization project intended to give programmers insight into the behavior of all levels of the systems on which their programs are running. He has also been involved with the K42 scalable operating system project and with the Libra library operating system project.

Dilma Da Silva *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (dilmasilva@us.ibm.com).* Dr. Da Silva is a Research Staff Member and Manager in the Advanced Operating Systems department. She received her Ph.D. degree in computer science from the Georgia Institute of Technology in 1997. Prior to joining IBM, she was an Assistant Professor at the University of São Paulo, Brazil. She has published more than 60 scientific papers. She is a director at CRA-W (Committee on the Status of Women in Computer Research) and a founder of the Latinas in Computing group.

José E. Moreira *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598 (jmoreira@us.ibm.com).* Dr. Moreira received B.S. degrees in physics and electrical engineering and an M.S. degree in electrical engineering, all from the University of São Paulo, Brazil. He received his Ph.D. degree in electrical engineering from the University of Illinois Urbana–Champaign. Since joining IBM in 1995, he has been involved in several high-performance computing projects, including the teraflop-scale ASCI (Accelerated Strategic Computing Initiative) Blue-Pacific, ASCI White, and Blue Gene/L, for which he was the system software architect. From 2006 to 2008, he was the chief architect of the commercial scale-out initiative at IBM Research to develop superior scalable solutions for commercial computing. He is currently working on a joint project between IBM Research and IBM Systems and Technology Group to develop the next generation of enterprise servers. Dr. Moreira is a member of the Institute of Electrical and Electronics Engineers and the Association for Computing Machinery.