Journal of Grid Computing manuscript No.
(will be inserted by the editor)

Using Kestrel and XMPP to support the STAR experiment in the Cloud

Lance Stout · Matthew Walker · Jérôme Lauret · Sebastien Goasguen · Michael A. Murphy

the date of receipt and acceptance should be inserted later

Abstract This paper presents the results and experiences of adapting and improving the Many-Task Computing (MTC) framework Kestrel for use with bag of tasks applications and the STAR experiment in particular. Kestrel is a lightweight, highly available job scheduling framework for Virtual Organization Clusters (VOCs) constructed in the cloud. Kestrel uses the Extensible Message and Presence Protocol (XMPP) for increasing MTC platform scalability and mitigating faults in Wide Area Network (WAN) communications.

Kestrel's architecture is based upon pilot job frameworks used extensively in Grid computing, with fault-tolerant communications inspired by command-and-control botnets. The extensibility of XMPP has allowed development of protocols for identifying manager nodes, discovering the capabilities of worker agents, and for distributing tasks. Presence notifications provided by XMPP allow Kestrel to monitor the global state of the pool and to perform task dispatching based on worker availability. Based upon test results, we conclude that

L. Stout, S. Goasguen
School of Computing
120 McAdams Hall
Clemson University
Clemson, SC 29634-0974 USA
E-mail: lstout@clemson.edu
E-mail: sebgoa@clemson.edu

M. Walker
Massachusetts Institute of Technology
Cambridge, MA 02139 USA
E-mail: mwalker@mit.edu

J. Lauret
Brookhaven National Laboratory
Upton, NY 11973 USA
E-mail: jlauret@bnl.gov

M. Murphy
Coastal Carolina University
Conway, SC 29528 USA
E-mail: mmurphy2@coastal.edu

XMPP is by design a good fit for cloud computing frameworks, since it offers scalability, federation between servers, and some autonomicity of the agents.

Since its inception, Kestrel has been modified based on its performance managing operational scientific workloads from the STAR group at Brookhaven National Laboratories. STAR provided a virtual machine image with applications for simulating proton collisions using PYTHIA and GEANT3. A Kestrel-based Virtual Organization Cluster, created on top of Clemson University's Palmetto cluster, CERN, and Amazon EC2, was able to provide over 400,000 CPU hours of computation over the course of a month using an average of 800 virtual machine instances every day, generating nearly seven terabytes of data and the largest PYTHIA production run that STAR has achieved to date.

1 Introduction

Since 2007, cloud computing has quickly become a new computing paradigm in both the enterprise and scientific applications domains. A definition of cloud computing has been considered in [16] and [44]; however, the term is so broad that it has been difficult to agree on a specific definition. In this paper, cloud computing refers to the on-demand provisioning and allocation of virtualized resources over a Wide Area Network (WAN), using virtual machines instantiated on multiple cloud providers or grid sites supporting virtualization. This design enables scientists to customize their own computing environments by preparing a standard worker node that will meet all the requirements of their applications. In addition, it provides stronger application encapsulation compared to scheduling jobs on shared physical clusters. Such encapsulation permits improved resource utilization via consolidation mechanisms and improved security by isolating user applications from each other [14]. From the resource owner's point of view, virtualization of resources also enables greater flexibility in system administration, as multiple operating systems can be offered simultaneously.

This approach to cloud computing is an extension of the Virtual Organization Cluster (VOC) model for grid computing [26]. In a VOC, a Virtual Organization (VO) starts virtual machines over a wide area network using multiple providers, then a network overlay is built within the VM instances. A VOC can be built using existing grid sites that have virtualization capabilities, or using corporate cloud providers. It has been demonstrated that one can build a VOC using an Open Science Grid site, EC2 resources, and a large-scale computational cluster at CERN [6, 40]. Elastic resource provisioning is implemented by watching the job scheduler queues and automatically provisioning virtual machines to build a VOC large enough to handle the workload. First-generation VOCs rely either on the ability of the VMs to lease scheduling mechanisms from the physical site or on virtual overlay networks that enable the VO to run its own standard job scheduler. In this paper we explore the case where virtual overlay networks are impractical or impossible, requiring a VO-specific job scheduler to access all VM resources with minimal networking assumptions.

While a significant body of work has been done in operating system virtualization [8], virtual networks [43, 18], and provisioning of virtual machines [23, 38, 28], little has been done on job scheduling in cloud overlays, often assuming that existing scientific computing job schedulers could be used in these environments. Clouds built over wide area networks offer great technical challenges:

- Cloud providers and grid sites need to trust each other as well as the image producers in order to instantiate the same images [9].
- Virtual machine images need to be transferred to all sites involved as well as made available on all physical nodes through either staging or shared file systems [36].

- Job scheduling needs to be network-agnostic and adaptable to a wide variety of network conditions.
- Users need to be taught a new access mechanism or use existing tools that have been adapted for clouds.

Although existing tools [41, 12, 29, 47] can certainly evolve to be utilized in clouds, this paper presents a novel job scheduling framework with a network-agnostic design intended for cloud systems constructed over a WAN: Kestrel [39, 40]. Unlike the push models used by traditional job schedulers, which transmit jobs directly to worker nodes, Kestrel is designed around a pull model in which worker nodes retrieve their workloads from the scheduler. Such a model has been used extensively in grid computing in the form of pilot jobs, whereby users develop their own job overlays on top of the traditional grid scheduling framework. Such overlays operate in a manner similar to the BOINC [7] middleware, with the grid schedulers used solely to probe remote providers and request resources in a manner similar to using a cloud Infrastructure as a Service (IaaS) Application Programming Interface (API). Examples of grid pilot job frameworks include PANDA [24], DIRAC [42], and GlideinWMS [37].

A design based upon a pull model is chiefly motivated by the heterogeneity of networking resources among different cloud IaaS providers. With the increasing shortage of IPv4 addresses and slow transition to IPv6, virtualized resources are often provided with network connectivity behind a Network Address Translation (NAT) boundary without support for inbound connections. The combination of NAT with provider security and firewall policies effectively creates a hostile network environment for the creation and maintenance of overlay networking systems. Kestrel avoids these issues by using solutions from both command-and-control botnets and instant messaging systems. Central to this approach is the use of the Extensible Messaging and Presence Protocol (XMPP), a robust standard for instant messaging recently adopted as the only approved instant messaging protocol for the Department of Defense [48]. XMPP has been designed for hostile networking environments as well as intermittently connected, heterogeneous communications agents. XMPP scales to hundreds of thousands of clients simultaneous clients, as demonstrated by instant messaging systems such as Google Talk and Facebook Chat.

The remainder of this paper is organized as follows: related work is presented in section 2. The XMPP protocol is introduced in section 3, after which Kestrel is detailed in section 4. Section 5 presents the STAR collaboration and describes the experiments carried out by STAR using Kestrel. Results are presented in section 6, followed by conclusions and future work in section 7.

2 Related Work

Designed to scavenge unused CPU cycles, the Condor High Throughput Computing (HTC) system executes computationally expensive operations over time on machines that would otherwise go unused [41, 13]. The architecture of Condor is based on linking processes together across the various submission and execution nodes in the cluster so that job data and output may be transferred. Inbound connectivity between the submit and execute nodes is required by default once a job is matched to an execute node. While this arrangement is adequate when all of the compute elements are in the same subnet, using Condor with machines on opposite sides of a NAT boundary introduces complications that require the generic connection broker. For large-scale systems, Falkon [30] has been designed to achieve a high dispatch rate, outperforming Condor in situations where resources are pre-allocated and the scheduling decisions are kept to a minimum. Kestrel has features similar to both

Condor and Falkon: the execution pool is created from pre-allocated cloud agents, then traditional scheduling, based on the Condor matchmaking mechanism, is used.

Several workflow engines [11, 29] and scripting languages [47] have been developed to run jobs efficiently on computational grids. These tools generally have been designed for use with Globus [15] grids, where the middleware uses gatekeepers to access resources. These tools can be adapted for use in cloud environments and provide a way to optimize workflow based on cloud utility pricing [12], but they may not fare well in infrastructure that lacks both inbound network connectivity and shared filesystem support.

The BOINC [7] middleware and existing pilot job frameworks [24, 37, 42] correspond to a pull based model that would work well in clouds. BOINC relies on volunteers to contribute cycles and on communities to prepare their applications as a project. While a large number of projects have been created, BOINC is not considered a general purpose job scheduler. The pilot job frameworks are more similar to Kestrel, although Kestrel differs by providing an XMPP-based messaging system with high scalability, coupled with a task distribution framework inspired by botnet architectures.

A solution to allow Condor to operate with NAT traversal is the Internet Protocol over Peer-to-Peer system, or IPOP [18]. IPOP provides a generic networking stack with an underlying P2P network for data transmission. Unlike the regular networking stack on the host machine, IPOP does not require any centralized control or routing configuration. As a result, the pool of IPOP peers forms a virtual network that is able to span across NAT boundaries [19]. Since it is a full networking stack, IPOP provides TCP, UDP, and TLS/SSL network transports, enabling end-user applications to run without modification. IPOP has been used to construct an autonomic VOC as a cloud overlay, using the Condor scheduler [6]. Other virtual networks have been studied but have not used in cloud overlays [31, 43].

Weis and Lewis [46] presents an XMPP-based parallelized computation system for use with optimizing meander line Radio Frequency Identification (RFID) antennas. This system was formed from a static pool of available hardware machines, XMPP clients for each worker machine, and an XMPP-based scheduler. The scheduler bootstrapped the system by contacting each machine in the list of workers through SSH and starting an XMPP agent if one was not already running. The end result was a special-purpose, ad-hoc grid system. Kestrel uses a similar architecture, but is intended as a general-purpose job distribution and scheduling framework. Other special-purpose schedulers based on XMPP have been used in Bioinformatics web services [45] and the Taverna workflow engine [29], illustrating applications of the XMPP protocol for development of new services and mechanisms.

3 XMPP

The *Extensible Messaging and Presence Protocol* (XMPP) [32][33] is an open standard for XML-based communications in near real time. Small chunks of XML, called stanzas, are sent by clients to servers to be routed to other clients. When and how the routing occurs forms the basis for much of XMPP's functionality. XMPP provides a large range of services which applications, such as Kestrel, may utilize. These services include channel encryption, authentication, presence monitoring, unicast and multicast messaging, service discovery, capabilities advertisement, and federation.

While XMPP is based on XML, it does not use "proper" XML as might be expected. While at the end of an XMPP session, two valid XML documents will have been created (one for each communication direction), the individual messages are not complete XML documents. For this reason, they are instead referred to as "stanzas". Each stanza

is addressed to a Jabber ID, or JID, which identifies entities within the global XMPP network. JIDs look similar to email addresses, but with an extra piece of information attached: `user@domain/resource`. The `domain` portion of the JID is mapped to an XMPP server through DNS, allowing multiple XMPP servers to be federated in the same fashion as SMTP servers. The `resource` portion of the JID is optional. Without the resource, the JID is called a bare JID and refers to the user's overall account. With a resource, the JID is called a full JID, and refers to a particular connection from a user's account. Since resources are used to identify connections, they must be unique for any bare JID [33].

The Presence part of XMPP is what makes XMPP more than a plain message queuing or broadcasting system. Every XMPP entity may broadcast its current "presence" and status, such as if the entity is busy, idle, or offline. These broadcasts are routed by an XMPP server to all other entities that have subscribed to that agent's presence. The table of subscriptions, both incoming and outgoing, for an entity is called the "roster", which is similar to the "buddy list" in other instant messaging systems. Applications that use presence notifications may adjust their behavior based on the current state of the XMPP network in near real time.

There are three types of entities within an XMPP network: servers, clients, and components. Clients are the most basic entities and are associated with a single full JID. Every client has a roster that is sent to it by the server upon establishing a connection. As rosters become large, scaling becomes problematic [25]. While XMPP clients and servers may participate in multiple threads of communication at once, only a single stanza may be sent or received in a single direction. Large rosters therefore prevent any other actions from taking place until the entire roster has been received. In addition, servers are typically optimized for instant messaging applications where clients have small rosters of only a few hundred entries. Components resolve this issue simply by not using rosters. As a means to extend a server with extra functionality, a component is given control over an entire subdomain from the XMPP server. All stanzas addressed to a JID in that subdomain will be received by the component. If the component needs roster-like behavior, it must track presence subscriptions itself.

4 Kestrel

Kestrel is a job scheduling system based on the Extensible Messaging and Presence Protocol, or XMPP [32][33]. The original environment for which Kestrel was designed to manage was disparate collections of virtual machines pooled together into a single cluster, or a Virtual Organization Cluster (VOC) [26], as shown in figure 1. Compute nodes in VOCs are temporary and can be created and destroyed on demand; these virtual machines also may be killed by the hosting resource provider without warning. XMPP provides a solution for managing heterogeneous, intermittently connecting compute elements through the use of service discovery and presence notifications; Kestrel builds on these features with XMPP subprotocols for requesting and dispatching tasks in a cloud environment.

4.1 Design Changes

Based on the previous year of implementation and production use experiences presented in [40], Kestrel's protocol has been refined for greater scalability. The original protocol was based on XMPP message stanzas containing JavaScript Object Notation (JSON) [10]

encoded data. This protocol was changed to a pure XML-based representation for the STAR experiment.

Removing the JSON encoding, which was utilized instead of XMPP in the earliest versions of Kestrel, reduced the total parsing time in the system, improving scalability through efficiency, while reducing implementation complexity. The SleekXMPP library [17] was selected to provided an efficient mechanism for creating and extracting XML data from XMPP stanzas. With these adjustments, Kestrel was designed to comply with the XMPP Design Guidelines [34], provided by the XMPP Standards Foundation (XSF) [4], by using XML for its protocol.

Another major scalability improvement was realized through the use of an extra XMPP stanza type called an Info/Query (IQ) stanza [33]. Regular XMPP message stanzas did not provide strong guarantees of delivery for normal server installations; rather, messages were treated as "fire and forget" payloads [33]. Ensuring consistency between the Kestrel central manager's view of the state of the resource pool, in particular nodes that have pending jobs, and the actual state of the workers became less efficient under high workloads with large numbers of workers. IQ stanzas added request-response behavior similar to the semantics of the HTTP methods `GET` and `SET`, reducing message matching time. Further, the use of XMPP IQ stanzas guaranteed that every IQ request would receive a response, even if that response was generated by the XMPP server itself to indicate an error had occurred. In effect, these properties of the XMPP IQ mechanism de-centralized some of the scheduler processing operations, sharing work between Kestrel and the XMPP server, thereby improving efficiency.

4.2 Architecture

Based on the traditional master-worker architecture, Kestrel systems are made of three types of agents: clients, workers, and masters. Client agents are basic XMPP clients with only the logic needed to submit job submission and status requests. During the typical use case, client agents join the system intermittently and only long enough to execute a single scheduler request. Currently, a command-line script provided with Kestrel is the preferred client program, but commodity instant messaging clients may also be used to interact with a Kestrel pool.

A worker is a simple XMPP client that implements a wrapper for executing shell scripts. Such a simple worker implementation is consistent with the XMPP Design Guidelines [34], which state that functionality should be kept in servers when possible. As such, a Kestrel worker agent is a lightweight process, allowing a worker to be easily included in virtual machines used in limited-memory deployments. Unlike Kestrel clients, workers are intended to maintain lengthy connections with the XMPP server so that tasks may have time to execute. However, worker nodes are not expected to maintain 100% availability, and the offline presence that is triggered by a disconnection will alert the manager to reschedule tasks if necessary.

The Kestrel manager is implemented as an XMPP component in order to scale to several thousand worker agents [25]. Manager instances are typically run on the same machine as the XMPP server, but they could also be placed in a cluster of virtual machines alongside the workers. While the current backend for the manager is an SQLite [20] database that cannot be shared easily by multiple processes, using a larger database or a hash value store (such as Redis [35]) would allow multiple manager instances on different machines to service a single XMPP server, using the common backend to maintain consistent state. Such an arrangement

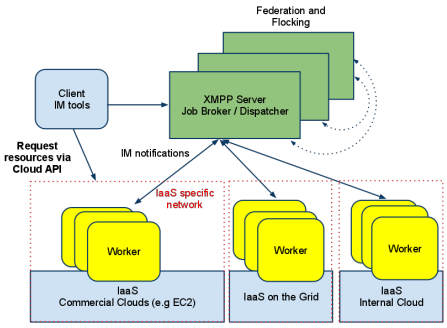

Fig. 1 Kestrel builds a scheduling overlay on top of cloud providers. Virtual machine instances containing the Kestrel worker agent can be started on any cloud provider using multiple provisioning systems. Once started, the instances join the Kestrel pool irrespective of the networking setup used by each provider. The Kestrel agents initiate the connection and set up an instant messaging channel to receive tasks. The Kestrel manager can make use of XMPP federation capabilities to build a scalable pool and establish flocking mechanisms.

would create a clustered manager that acts as a single, logical manager instance. However, current research is focused on manager federation such that Kestrel managers servicing separate XMPP servers from different organizations may share jobs, similar to the flocking mechanism in Condor.

4.3 Well-Known JIDs

Since the Kestrel manager is an XMPP component, it may be addressed by many JIDs with different username portions. For example, both the JIDs `pool@manager.example.org` and `job_42@manager.example.org` will be delivered to the Kestrel manager. Kestrel reserves two particular usernames for interaction with clients and workers. The first is `submit`, which accepts job submissions and status requests from clients. The second is `pool`, which is contacted by workers attempting to join the pool and by clients requesting the pool's status.

Each job accepted by the manager is also given a unique JID username of the form: `job_#`, where the `#` is replaced by the job's ID value. For example, a job with an ID of 37 may be referenced as `job_37@manager.example.org`. Each job submitted to Kestrel may be composed of multiple "tasks", which are the individual instances of the job's command to execute. The resource portion of a job's JID refers to a particular task, e.g. a worker receiving a task from the JID `job_37@manager.example.org/99` knows that it is executing task 99 from job 37. Once a job has been accepted by the manager, a subscription request is made from the job's JID to the client's JID, allowing the user to easily monitor the status of the job from a commodity Instant Messaging client, as shown in figure 2.

4.4 Joining the Pool

Creating the Kestrel pool is done using XMPP's service discovery features to detect worker agents and their capabilities, as shown in figure 3. Workers attempt to join the pool by sub-

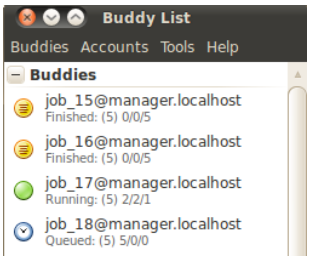

Fig. 2 Monitoring job status through a commodity Instant Messaging (IM) client. Once accepted by the manager, jobs request to be added to the user's roster to deliver current status updates. Running jobs display themselves as available while queued jobs are displayed as away. For finished jobs, an extended away status is shown. The status message for each job includes a sequence of four numbers indicating, in order, the number of tasks requested, queued, running, and completed. Jobs may be canceled from an IM client by removing them from the roster.

scribing to the special manager JID `pool@manager.example.org`. Once the manager is subscribed to the potential worker, Kestrel uses the XMPP service discovery feature [21], as shown in figure 4, to determine if the entity is an actual worker. When an XMPP entity receives a service discovery (or "disco" request), it returns a list of features associated with the entity. The XMPP entity may also group features into various aspects or facets of its intended functionalities (referred to as nodes [21]). Every Kestrel worker will advertise its support for executing tasks in a Kestrel pool by including the feature `kestrel:tasks`. Since worker agents may provide different resources, such as operating systems or installed libraries, workers may also advertise additional capabilities that may be used for task matching. Querying the `kestrel:tasks:capabilities` node of the worker's service discovery profile will provide these features. It should be noted that using service discovery does increase the startup time of a Kestrel-based VOC due to the protocol overhead. However, future implementations should be able to overcome this drawback by using an alternative XMPP extension [22] that includes a hashed version of the worker's features in its initial presence notification. The manager would then only need to request the original, full feature list once per unique hash it receives.

4.5 Dispatching

Assigning tasks to workers is done by sending a task IQ stanza to a worker after it has announced its availability as shown in figure 5. During the time period between sending the stanza and receiving a reply, the task and worker are marked as pending in the manager's internal data store to prevent assignment conflicts. The current design of Kestrel limits each worker to accepting only one task instead of multiple concurrent tasks; the rationale is to simplify matchmaking for the manager by reducing the number of possible worker states. Future implementations may remove this limitation. In the event that an error is returned because the worker has reached its task limit or is no longer online, the task is returned to the queue to be matched with another worker (figure 6).

To support the specific requirements of the STAR experiment (section 5), a new task attribute has been added for carrying out a cleanup phase. After the main command for a task has been executed, the worker will report that it has completed the task. However, the worker will not make itself available to receive a new task until the cleanup command has completed, if one has been provided. Such an arrangement allows cleanup scripts to shut

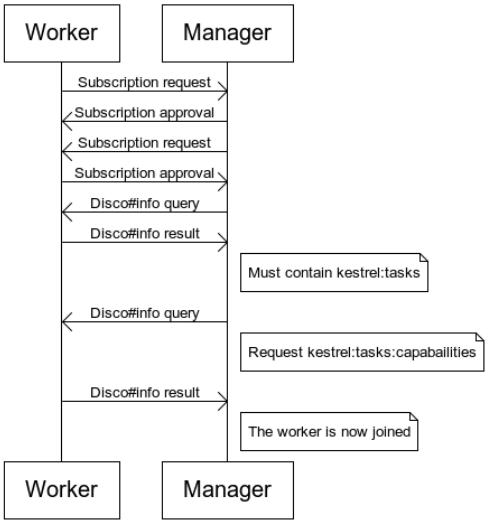

Fig. 3 The presence subscription and service discovery process for joining the Kestrel pool. A worker first initiates a bi-directional presence subscription with the manager, allowing the manager to know when the worker is available or offline. The manager then queries the worker's service descriptions to verify that the agent is a worker, and to find the capabilities the worker provides that may be used in match making.

```
<iq to="worker21@example.org" type="get">
  <query xmlns="http://jabber.org/protocol/disco#info" />
</iq>
<iq from="worker21@example.org" type="result">
  <query xmlns="http://jabber.org/protocol/disco#info">
    <feature>kestrel:tasks</feature>
  </query>
</iq>
<iq to="worker21@example.org" type="get">
  <query xmlns="http://jabber.org/protocol/disco#info"
         node="kestrel:tasks:capabilities" />
</iq>
<iq from="worker21@example.org" type="result">
  <query xmlns="http://jabber.org/protocol/disco#info">
    <feature>Python2.6</feature>
    <feature>Linux</feature>
  </query>
</iq>
```

Fig. 4 Recognizing an XMPP agent as a Kestrel worker, and discovering its capabilities. The manager first sends an IQ "disco" stanza to the agent; the agent's response must include the `kestrel:tasks` feature in order to be accepted as a worker. Once the worker has been recognized, a second "disco" query is issued, this time to the `kestrel:tasks:capabilities` node of the agent's profile. The resulting list of features are the capabilities that the worker offers for use during match making.

down and restart the virtual machine without causing the manager to treat the shutdown as a network failure and reassigning the task to another worker.

4.6 Task Execution

The actual programs executed by workers are shell scripts that manage the task's execution cycle, performing steps such as downloading input data, executing a scientific payload,

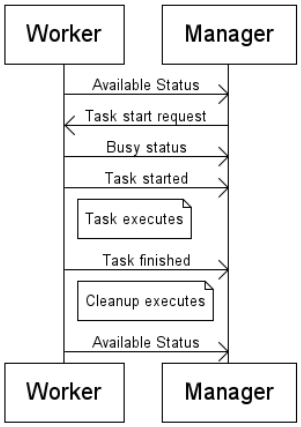

Fig. 5 The task dispatch and execution process. A match making request is triggered when the manager receives an available presence from a worker. If a matching job is found, a task is marked pending and sent to the worker; if an error is returned or a timeout occurs, the task is returned to the queue. Otherwise, the worker broadcasts a busy presence to prevent further job matching, and then notifies the manager that the task has been started. Once the task's command has terminated, a finished notice is sent to the manager to mark the task as completed. An optional cleanup step is then performed before the worker issues an available presence indicating it is ready for the next task.

```
<iq type="set" to="worker17@example.org"
              from="job42@manager.example.org/23">
  <task xmlns="kestrel:task" action="execute">
    <command>/runtask.sh</command>
    <cleanup>/cleanfiles.sh</cleanup>
  </task>
</iq>
<iq type="error" to="job42@manager.example.org/23"
                 from="worker17@example.org">
  <error xmlns="urn:ietf:params:xml:ns:xmpp-stanzas"
         type="cancel">
    <condition>resource-constraint</condition>
    <text>The worker is already in use.</text>
  </error>
</iq>
```

Fig. 6 A task start stanza and an error reply. Note the task's ID number is given by the resource identifier of the job's JID.

and uploading any output. These scripts will always receive a *final* parameter which is the task's ID value; however, additional parameters may be passed when submitting the job by including them with the job's command. The task ID may also be used as a switch to run different applications with a single job. Since Kestrel workers are assumed to be behind a NAT boundary and/or firewall, tasks must operate in a pull model: interactions with external entities must be done through requests starting from the worker node.

Most applications using Kestrel will need to transfer input data to the worker and then transfer output data to the user in some fashion. Some scheduler and dispatch systems, such as Condor, provide built-in methods for data transfer; however, Kestrel currently relies on existing commodity tools instead. While data transfers may be included in a future release using XMPP's file sharing capabilities, `wget` and similar utilities are presently used. Trans-

```
<iq type="set" to="manager.example.org">
  <job xmlns="kestrel:job" action="submit" queue="50000">
    <command>/runtask.sh</command>
    <cleanup>/cleanfiles.sh</cleanup>
    <requires>Python2.6</requires>
    <requires>SleekXMPP</requires>
  </job>
</iq>
```

Fig. 7 A job submission stanza with two requirements. The `queue` attribute specifies the number of times that the job will be executed, where each instance is considered a single "task". Various `requires` elements may be added to limit set of workers that may run the job's tasks. When using Kestrel to manage a single pool shared by various organizations, the organization's identity is usually added as a requirement so that the job will run on the organization's VMs.

ferring files is a largely solved problem with scalable solutions, such as the use of an HTTP server (or server cluster) to distribute input files or sections thereof with range requests. A simple upload form processor may be used to handle receiving output data from workers. Additionally, uploading files from Kestrel worker can be done via GridFTP, using a grid proxy stored on the VM instance during the provisioning step.

Once a task's command has been executed, an optional second command may be executed for cleanup purposes. In most cases, this cleanup script will simply restart the VM to reset its state and release any disk space used by copy-on-write instances. The cleanup command is split from the main task command so that if the worker VM is restarted, the task will not be rescheduled when the manager is notified that the worker has disconnected while running a task.

4.7 Job Management

Creating and monitoring jobs can be carried out through the `kestrel` command-line application.. As shown in figure 2, job status information can be received through a commodity Instant Messaging program, such as Pidgin [2] or Adium [1]. This status information is conveyed via the job JID's presence updates and includes the number of queued, running, and completed tasks. Jobs that are accepted and assigned a JID are added to the user's XMPP roster (also referred to as a "buddy list" in other messaging systems). Removing the job from the roster will cancel the job and terminate running tasks. Alternatively, the `kestrel` command-line interface may be used to submit, cancel, or query jobs using the commands depicted in figure 9.

The protocol for submitting a job, shown in figure 7, resembles the stanza used to initiate a task, but it may also include a variable number of `<requires />` elements. These requirements are matched against the capabilities provided by workers to ensure that the job will only run on VMs that have the appropriate libraries or belong to the proper organization. To simplify the public Application Programming Interface (API) provided by Kestrel to third-party tools, submissions must be sent to the special manager JID `submit@manager.example.org`. By setting the `id` attribute to a job ID and setting the attribute `action="cancel"` in a job stanza similar to that shown in figure 7, a job cancellation request may be submitted.

At any time, the status of a job may be directly requested instead of relying on the summarized status included in the job's presence updates. Issuing an IQ stanza with a `kestrel:status` query (as shown in figure 8) to the JID `submit@manager.example.org`

```
<iq type="get" to="submit@manager.example.org">
  <query xmlns="kestrel:status" />
</iq>
<iq type="result" to="user@example.org">
  <query xmlns="kestrel:status">
    <job owner="user@example.org">
      <queued>3000</queued>
      <running>700</running>
      <completed>300</completed>
      <requested>4000</requested>
    </job>
    <job owner="user@example.org">
      <queued>150</queued>
      <running>50</running>
      <completed>300</completed>
      <requested>500</requested>
    </job>
  </query>
</iq>
```

Fig. 8 A job status request stanza and its response. Since the query was issued to the job submission JID, the statuses of all of the user's jobs are returned. Each status includes the number of requested, queued, running, and completed tasks. Issuing the same query to a job's JID would return the status of the single job.

Command	Result
`kestrel [-q] submit <jobfile>`	Submit a job request.
`kestrel [-q] cancel <jobid>`	Cancel an accepted job.
`kestrel [-q] retry <jobid>`	Return any tasks that completed with errors to the queue to be executed again.
`kestrel [-q] status [<jobid>]`	Request the status of all jobs or a single job.
`kestrel [-q] status pool`	Request the status of the pool.

Fig. 9 Available commands for the command-line `kestrel` application.

will return the breakdown of task states and the owners for all jobs active in the system. The query can be targeted to a job's JID to return the number of queued, running, completed, and requested tasks for that particular job. Issuing the status request to the JID `pool@manager.example.org` will return the number of online, available, and busy workers.

5 Scientific Cloud Experiment

5.1 STAR

Maintained by the Brookhaven National Laboratories, the STAR experiment is a multi-purpose detector at the Relativistic Heavy Ion Collider (RHIC) dedicated to understanding phenomena of quantum chromodynamics (QCD) [3]. One of the main programs at RHIC involves the collisions of heavy nuclei, for example gold, at energies up to 200 GeV. These collisions provide a way to study phases of matter that occurred during the very early phases of the universe. The second major program is the study of the spin of the proton. As the world's only polarized proton accelerator, RHIC is ideal for understanding the contributions gluons make to the proton's spin. The simulation provided for the experiment with Kestrel was related to the spin program.

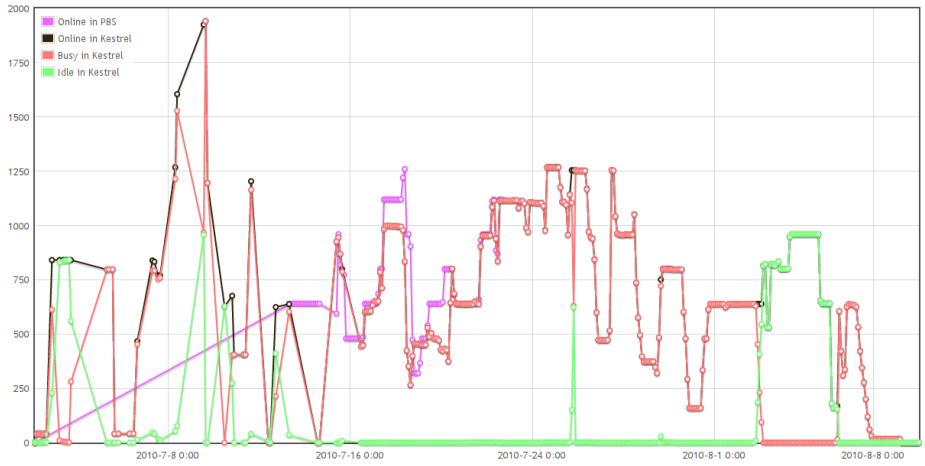

Fig. 10 The number of online, available, and busy workers according to the Kestrel manager during the course of the experiments.

5.2 Description

The STAR group became interested in using Kestrel based upon its performance in terms of manageable pool sizes, dispatch rates, and startup times when tested via simulated workloads and microbenchmarking applications [40]. By collaborating with the Cyberinfrastructure Research Group at Clemson University, the STAR group was able to deploy a large set of simulations across different resources, providing valuable data and feedback for tuning Kestrel at scale.

STAR's goal was to use Kestrel to create a VOC for running proton collision simulations. The STAR software stack consisted of over 2.5 million lines of code, and deployment required a vast number of external libraries and multiple compilers, such as PYTHIA and GEANT3. In order to support such a large collection of specific software packages, STAR provided a virtual machine image containing the deployment of a single STAR library on the Scientific Linux 5.3 operating system. A copy of the STAR offline database was also installed, and a Virtual Organization Cluster (VOC) was instantiated using machines provided by CERN, Amazon EC2, and Clemson's Palmetto cluster, with a total worker pool size expected at near one thousand active workers over the course of one month. Distributing and starting the VMs on Palmetto was conducted with the Portable Batch System (PBS) [27], while provisioning the VM instances at CERN was done with the EC2 API implemented by Opennebula [38]. The principal steps taken to set up the experiments were:

1. Create a virtual machine containing the STAR code and databases, along with Kestrel and the the configuration data identifying the manager's JID and the worker capabilities. The command to start the Kestrel worker automatically at boot time was added to the VM's `/etc/rc.local` file.
2. Stage the virtual machine on the Clemson Palmetto cluster on a shared filesystem. Both NFS and PVFS were tested and performance was presented in [40].
3. Start virtual machine instances using PBS and KVM in snapshot mode, with eight VMs per physical node. The snapshot mode avoided transferring the base image to all nodes

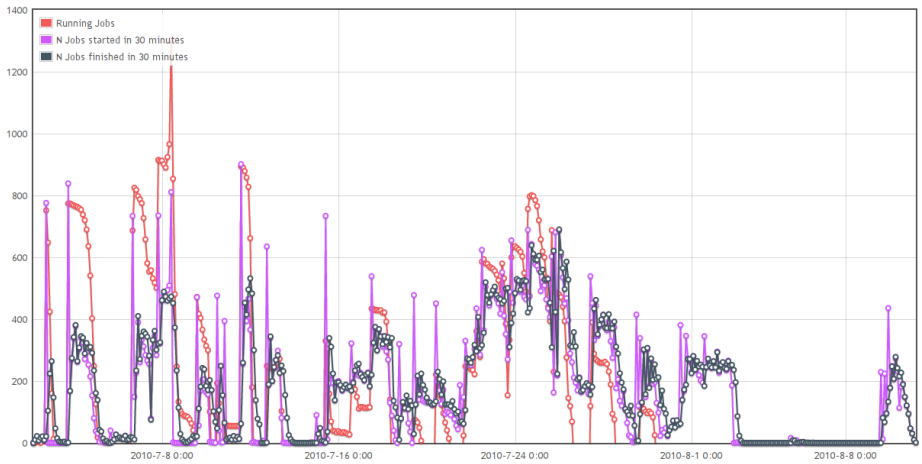

Fig. 11 The number of simulation tasks executing over the course of the experiment, as reported by STAR's worker applications. As can be seen in the first half of the graph, tasks executed in bursts due to hardware over-subscription resulting from inadequate tuning. Execution stabilized once tuning was complete.

by creating a local copy-on-write disk while keeping the base image on the shared file system.

4. Monitor the number of workers in Kestrel using both the Kestrel client, and the web interface for the XMPP server, ejabberd [5].
5. Submit and manage jobs from a Kestrel client.
6. Auto-refill virtual machine instances as PBS jobs expire after a maximum of three days of wall time or if the jobs get preempted based on allocation policies.

Using KVM/QEMU copy-on-write VM images posed a challenge during the design phase of the experiment due to over-subscription of the hardware. After instantiating eight VMs per physical node, the usable hard drive space per VM was approximately two gigabytes. With such a small space, workers would quickly exhaust their disk allocation after running only a few tasks because the copy-on-write image would never delete data from disk, even when files were deleted inside the VM. Restarting the VMs would free the allocation; however, to accommodate this, Kestrel's protocol had to be changed to include a cleanup phase for tasks, as described in section 4.5. Without the explicit cleanup, the Kestrel manager would treat the VM shutdown as a network error and reassign the task to another worker, preventing the task from ever completing.

6 Results

Due to a naive deployment without tuning, the first half of the experimental period yielded marginal successes in terms of the sustained number of active workers, as demonstrated in figure 10. Since the exact memory and execution requirements of the STAR VMs were not known *a priori*, Kestrel was configured to utilize resources aggressively. As a result, the hardware became over-subscribed, resulting in inefficient bursts of task execution. A tuning process was required to eliminate this over-subscription.

The tuning process produced an immediate increase in sustained worker count and task throughput as shown in figures 11 and 12. However, the original goal of one thousand ac-

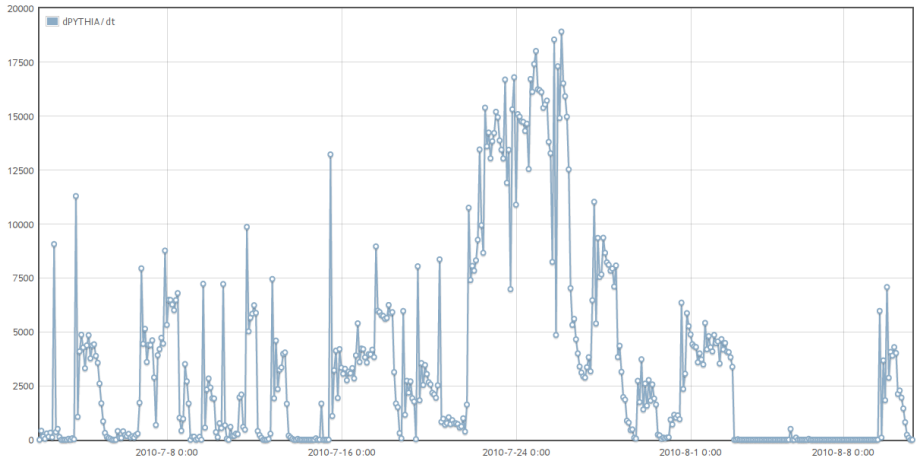

Fig. 12 Number of PYTHIA events generated during simulations. The spike in the event rate corresponds to the point at which system tuning was completed.

tive workers could still not be maintained due to irregular access to the available physical resources in the Palmetto cluster. An allocation on the Palmetto cluster or access to more resources at different sites capable of starting virtual machines would have increased the number of workers, but these resources were not available at the time of the experiment. A week after the main run of simulations was completed, a second, smaller experiment was carried out, creating the secondary surge in task execution seen in the graphs.

Over the course of the month, over eighty thousand tasks were executed, generating more than twelve billion events using PYTHIA, a proton simulation event generator. A subset of the events went through detector simulation using a GEANT3 based package. Finally, STAR reconstruction software was used to simulate a trigger on the remaining events. In all, the simulation ran for over 400,000 CPU hours. Nearly seven terabytes of results were transferred back to Brookhaven National Laboratories for further study. During the month the simulation took place, the CPU hours used amounted to an expansion of approximately 25% over STAR's average capacity. Furthermore, the STAR group has stated that it would have been only able to devote fifty CPUs to this task on its own farm, increasing the real time to completion by a factor of twenty.

7 Conclusions and Future Work

As demonstrated with a successful trial experiment with STAR, Kestrel is capable of managing large-scale scientific applications in the cloud, with the condition that tasks are able to run independently in a bag of tasks model. While Kestrel did not provide file transfer capabilities itself, offloading the responsibility to existing infrastructure such as HTTP and GridFTP servers proved to be a workable solution, handling over seven terabytes of output data. Although the VM images used were up to twenty five gigabytes in size, KVM's snapshot mode reduced the startup times for the VMs by only transferring image data when accessed. Snapshot mode also prevented the main image from being modified by running instances by writing changes to the VM host's local disk; the addition of the task cleanup

command to allow for restarting the VM prevented exhausting the local hard drive's capacity when shared with other VMs.

Current and future work with Kestrel will focus on using multiple managers for a single pool. XMPP already provides the federation infrastructure needed to operate a single pool with multiple XMPP servers, but more work is needed to federate the manager components. Through the use of a shared data store, such as Redis [35], a multi-manager installation can be easily achieved as long as only a single XMPP server is used, since some implementations automatically load-balance components [5]. Linking managers on different servers, in particular servers from different organizations, will require additions to Kestrel's protocol to create the equivalent of "flocking" between Condor systems. The current design under consideration is to use a distributed hash table (DHT) to spread workers among managers, and then allow managers to masquerade as a worker to other managers. The masquerading manager would advertise the union of its workers' capabilities, and it would act as a task router, forwarding any received tasks to one of its workers matching the task.

References

1. Adium. URL `http://www.adium.im`.
2. Pidgin, the universal chat client. URL `http://www.pidgin.im`.
3. The star experiment. URL `http://www.star.bnl.gov/`.
4. Xmpp standards foundation. URL `http://xmpp.org`.
5. Ejabberd. URL `http://www.process-one.net/en/ejabberd/`.
6. L. Abraham, M. Murphy, M. Fenn, and S. Goasguen. Self-provisioned hybrid clouds. In *Proceeding of the 7th international conference on Autonomic computing*, pages 161–168. ACM, 2010.
7. D. Anderson. BOINC: A system for public-resource computing and storage. In *proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004. ISBN 0769522564.
8. P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Nineteenth ACM Symposium on Operating Systems Principles*, 2003.
9. M. Bégin. An EGEE comparative study: Grids and Clouds-evolution or revolution. In *23rd Open Grid Forum (OGF23)*, 2008.
10. D. Crockford. Introducing json. URL `json.org`.
11. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, and M. Livny. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing*, pages 131–140. Springer, 2004.
12. E. Deelman, G. Singh, M. Livny, B. Berriman, and J. Good. The cost of doing science on the cloud: the montage example. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2009.
13. M. L. Douglas Thain, Todd Tannenbaum. How to measure a large open source distributed system. *Concurrency and Computation: Practice and Experience*, 8(15), December 2006.
14. R. J. Figueiredo, P. A. Dinda, and J. A. B. Fortes. A case for grid computing on virtual machines. In *23rd International Conference on Distributed Computing Systems*, 2003.
15. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.

16. I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. Ieee, 2009.
17. N. Fritz and L. Stout. Sleekxmpp. URL `http://github.com/fritzy/SleekXMPP`.
18. A. Ganguly, A. Agrawal, P. O. Boykin, and R. Figueiredo. IP over P2P: Enabling self-configuring virtual IP networks for grid computing. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006. doi: 10.1109/IPDPS.2006.1639287.
19. A. Ganguly, A. Agrawal, P. O. Boykin, and R. Figueiredo. WOW: Self-organizing wide area overlay networks of virtual workstations. In *15th IEEE International Symposium on High Performance Distributed Computing*, 2006. doi: 10.1109/HPDC.2006.1652133.
20. R. Hipp. Sqlite. URL `http://www.sqlite.org`.
21. R. E. Joe Hildebrand, Peter Millard and P. Saint-Andre. Xep-0030: Service discovery. URL `http://xmpp.org/extensions/xep-0030.html`.
22. R. T. Joe Hildebrand, Peter Saint-Andre and J. Konieczny. Xep-0115: Entity capabilities. URL `http://xmpp.org/extensions/xep-0115.html`.
23. K. Keahey and T. Freeman. Contextualization: Providing one-click virtual clusters. In *4th IEEE International Conference on e-Science*, Indianapolis, IN, December 2008.
24. T. Maeno. PanDA: distributed production and distributed analysis system for ATLAS. In *Journal of Physics: Conference Series*, volume 119, page 062036. IOP Publishing, 2008.
25. J. Moffitt. Thoughts on scalable xmpp bots, August 2008. URL `http://metajack.im/2008/08/04/\-thoughts-on-scalable-xmpp-bots/`.
26. M. A. Murphy, M. Fenn, and S. Goasguen. Virtual Organization Clusters. In *17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2009)*, Weimar, Germany, February 2009.
27. B. Nitzberg, J. M. Schopf, and J. P. Jones. Pbs pro: Grid computing and scheduling attributes. pages 183–190, 2004.
28. D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid-Volume 00*, pages 124–131. IEEE Computer Society, 2009.
29. T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Greenwood, T. Carver, M. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 2004. ISSN 1367-4803.
30. I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a Fast and Light-weight tasK executiON framework. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–12. IEEE, 2007.
31. P. Ruth, X. Jiang, D. Xu, and S. Goasguen. Virtual distributed environments in a shared infrastructure. *Computer*, 38(5):63–69, 2005.
32. P. Saint-Andre. Extensible messaging and presence protocol (xmpp): Core, October 2004. URL `http://www.ietf.org/rfc/rfc3920.txt`.
33. P. Saint-Andre. Extensible messaging and presence protocol (xmpp): Instant messaging and presence, October 2004. URL `http://www.ietf.org/rfc/rfc3921.txt`.
34. P. Saint-Andre. Xep-0134: Xmpp design guidelines. URL `http://xmpp.org/extensions/xep-0134.html`.

35. S. Sanfilippo. Redis. URL http://code.google.com/p/redis/.
36. M. Schmidt, N. Fallenbeck, M. Smith, and B. Freisleben. Efficient Distribution of Virtual Machines for Cloud Computing. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 567–574. IEEE, 2010.
37. I. Sfiligoi. glideinWMS a generic pilot-based workload management system. In *Journal of Physics: Conference Series*, volume 119, page 062044. IOP Publishing, 2008.
38. B. Sotomayor, R. Montero, I. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13(5):14–22, 2009.
39. L. Stout, M. A. Murphy, and S. Goasguen. Kestrel: an xmpp-based framework for many task computing applications. In *MTAGS '09: Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers*, pages 1–6, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-714-1. doi: http://doi.acm.org/10.1145/1646468.1646479.
40. L. Stout, M. Fenn, M. A. Murphy, and S. Goasguen. Scaling virtual organization clusters over a wide area network using the kestrel workload management system. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 692–698, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-942-8. doi: http://doi.acm.org/10.1145/1851476.1851578. URL http://doi.acm.org/10.1145/1851476.1851578.
41. T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – a distributed job scheduler. In T. Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.
42. A. Tsaregorodtsev, V. Garonne, and I. Stokes-Rees. DIRAC: A Scalable Lightweight Architecture for High Throughput Computing. In *Fifth IEEE/ACM International Workshop on Grid Computing (GRID '04)*, Pittsburgh, PA, November 2004.
43. M. Tsugawa and J. A. B. Fortes. A virtual network (ViNe) architecture for grid computing. In *20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, 2006. doi: 10.1109/IPDPS.2006.1639380.
44. L. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1): 50–55, 2008.
45. J. Wagener, O. Spjuth, E. Willighagen, and J. Wikberg. XMPP for cloud computing in bioinformatics supporting discovery and invocation of asynchronous web services. *BMC bioinformatics*, 10(1):279, 2009. ISSN 1471-2105.
46. G. Weis and A. Lewis. Using xmpp for ad-hoc grid computing - an application example using parallel ant colony optimisation. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–4, May 2009. doi: 10.1109/IPDPS.2009.5161115.
47. M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel socripting for applications at the petascale and beyond. *Computer*, 42(11):50–60, 2009. ISSN 0018-9162.
48. R. Yasin. Dod wants instant messaging tools to speak the same language, June 2010. URL http://defensesystems.com/articles/2010/\-06/23/\-dod-instant-\-messaging-\-test.aspx.