

# KESTREL: JOB DISTRIBUTION AND SCHEDULING USING XMPP

---

A Thesis  
Presented to  
the Graduate School of  
Clemson University

---

In Partial Fulfillment  
of the Requirements for the Degree  
Master of Science  
Computer Science

---

by  
Lance Joshua Thomas Stout  
May 2011

---

Accepted by:  
Dr. Sebastien Goasguen, Committee Chair  
Dr. Steve Stevenson  
Dr. Pradip Srimani

# Abstract

A new distributed computing framework, named Kestrel, for Many-Task Computing (MTC) applications and implementing Virtual Organization Clusters (VOCs) is proposed. Kestrel is a lightweight, highly available system based on the Extensible Messaging and Presence Protocol (XMPP), and has been developed to explore XMPP-based techniques for improving MTC and VOC tolerance to faults due to scaling and intermittently connected heterogeneous resources. Kestrel provides a VOC with a special purpose scheduler for VOCs which can provide better scalability under certain workload assumptions, namely CPU bound processes and bag-of-task applications.

Experimental results have shown that Kestrel is capable of operating a VOC of at least 1600 worker nodes with all nodes visible to the scheduler at once. When using multiple sites located in both North America and Europe, the latencies introduced to the round trip time of messages were on the order of 0.3 seconds. To offset the overhead of XMPP processing, a task execution time of 2 seconds is sufficient for a pool of 900 workers on a single site to operate at near 100% use. Requiring tasks that take on the order of 30 seconds to a minute to execute would compensate for increased latency during job dispatch across multiple sites.

Kestrel's architecture is rooted in pilot job frameworks heavily used in Grid computing, it is also modeled after the use of IRC by botnets to communicate between compromised machines and command and control servers. For Kestrel, the extensibility of XMPP has allowed development of protocols for identifying manager nodes, discovering the capabilities of worker agents, and for distributing tasks. The presence notifications provided by XMPP allow Kestrel to monitor the global state of the pool and to perform task dispatching based on worker availability. In this work it is argued that XMPP is by design a very good fit for cloud computing frameworks. It offers scalability, federation between servers and some autonomy of the agents.

During the summer of 2010, Kestrel was used and modified based on feedback 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, 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 ever achieved. Several architectural issues were encountered during the course of the experiment and were resolved by moving from the original JSON protocols used by Kestrel to native XMPP equivalents that offered better message delivery confirmation and integration with existing tools.

# Acknowledgments

I would like to thank Dr. Goasguen for bringing me in to the Cyber Infrastructure Research Group. I have learned a great deal about distributed computing and XMPP thanks to his guidance and motivation. I am also grateful for my many discussions with Dr. Stevenson over the last few years discussing programming languages and their histories, and just life in general.

Mike Murphy has my gratitude for his help trying to teach me the proper methods for academic writing, the many design discussions we had, and for being a good friend. I want to also thank Michael Fenn for his help and experience with using the Palmetto cluster and how to work with KVM.

I would like to acknowledge Jerome Lauret of Brookhaven National Laboratory and Matthew Walker of MIT for their help and feedback in running experiments with the STAR VO, Xiaoshu Wang of the Renaissance Computing Institute (RENCI) for his feedback and bug reporting, and David Wolinsky of the University of Florida for his help in conducting IPOP scalability tests.

Nathan Fritz of &yet and author of the Python XMPP library SleekXMPP, featured in O'Reilly's *XMPP: The Definitive Guide* deserves a great many thanks. He has trusted me enough not only to grant me commit access to the main source repository for the SleekXMPP library, but to also grant me co-authorship status as well. I hope that trust has not been misplaced; as a direct result of the work presented here, SleekXMPP has been expanded and refined from what Nathan had called a "hobbyist project" to a production ready 1.0 release. Features developed as part of Kestrel that were generic in nature have made their way into the main SleekXMPP library.

# Table of Contents

<b>Title Page</b> . . . . .	<b>i</b>
<b>Abstract</b> . . . . .	<b>ii</b>
<b>Acknowledgments</b> . . . . .	<b>iv</b>
<b>List of Tables</b> . . . . .	<b>vii</b>
<b>List of Figures</b> . . . . .	<b>viii</b>
<b>List of Listings</b> . . . . .	<b>x</b>
<b>1 Introduction</b> . . . . .	<b>1</b>
1.1 Virtual Organization Clusters . . . . .	2
1.2 Job Scheduling and Distribution in Clouds . . . . .	3
1.3 Motivations . . . . .	4
<b>2 Related Work</b> . . . . .	<b>6</b>
<b>3 XMPP: Extensible Messaging and Presence Protocol</b> . . . . .	<b>8</b>
3.1 The XMPP Network . . . . .	8
3.2 Jabber IDentifiers (JIDs) . . . . .	10
3.3 Types of XMPP Agents . . . . .	11
3.4 Stanzas . . . . .	12
3.4.1 Message . . . . .	13
3.4.2 Presence . . . . .	13
3.4.3 Info/Query (IQ) . . . . .	14
3.5 Security Considerations . . . . .	14
3.6 XMPP Extension Proposals (XEPs) . . . . .	16
3.6.1 Service Discovery: XEP-0030 . . . . .	16
3.6.2 Entity Capabilities: XEP-0115 . . . . .	17
<b>4 Kestrel</b> . . . . .	<b>18</b>
4.1 Architecture . . . . .	19
4.1.1 Well-Known JIDs . . . . .	19
4.2 Clients . . . . .	21
4.2.1 Job Management . . . . .	21
4.3 Workers . . . . .	23
4.3.1 Joining the Pool . . . . .	23
4.3.2 Task Execution . . . . .	24
4.4 Manager . . . . .	28
4.4.1 Scheduling and Dispatching . . . . .	28

4.4.2	Responding to Worker Failures . . . . .	30
4.5	Evolution of the Implementation and Lessons Learned . . . . .	30
4.5.1	JSON over XMPP Message Stanzas . . . . .	31
4.5.2	Customized IQ Stanzas . . . . .	32
<b>5</b>	<b>Clustering and Federation . . . . .</b>	<b>33</b>
5.1	Clustering . . . . .	33
5.1.1	Challenges with Redis . . . . .	34
5.1.2	Library Support . . . . .	34
5.1.3	Mapping SQL databases to Redis . . . . .	35
5.1.4	Matching Jobs with Workers . . . . .	35
5.1.5	Serialization . . . . .	36
5.1.6	XMPP Server Load Balancing . . . . .	37
5.2	Federation . . . . .	37
<b>6</b>	<b>Results . . . . .</b>	<b>40</b>
6.1	Responding to Changes in VOC Size . . . . .	41
6.2	Latency . . . . .	42
6.3	Pool Sizes . . . . .	42
6.3.0.1	VM Image Considerations . . . . .	44
6.3.0.2	IPOP/Condor Experiment . . . . .	45
6.3.0.3	Kestrel Experiment . . . . .	47
6.4	Kestrel Task Throughput and Dispatch Rates . . . . .	48
6.5	Clustered Manager Performance . . . . .	53
6.6	STAR . . . . .	53
6.6.1	Description . . . . .	54
6.6.2	Results . . . . .	56
<b>7</b>	<b>Conclusions and Discussion . . . . .</b>	<b>59</b>
	<b>Bibliography . . . . .</b>	<b>61</b>

# List of Tables

4.1 Available commands for the command line client. . . . .	23
---	----

# List of Figures

3.1	A demonstration of a basic XMPP network, including client agents, internal and external server components, and XMPP servers. Communications from entities associated with one server are not routed by any intermediary server. Messages sent from the client marked “A” to the client marked “B” will always be sent from server labeled “1” to the server labeled “2”, never along the path from “1” to “3” to “2”.	9
3.2	Daily usage graph from the Russian XMPP service provider Jabber.ru. Average use is around 22,000 simultaneous client connections during peek daytime hours.	10
4.1	The logical network communication paths for a Kestrel pool. Messages may be routed through at most two XMPP servers for any of the paths shown, including the server exposing the manager components. These potential additional servers are indicated by the square on each path. Using XMPP’s built-in federation, worker instances can be easily distributed across multiple XMPP servers.	20
4.2	Monitoring job statuses through a normal instant messaging 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 it from the roster.	21
4.3	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.	25
4.4	Kestrel builds a scheduling overlay on top of Cloud providers. Virtual machines 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 setup a instant messaging channel to receive tasks. The Kestrel manager can make use of the XMPP federation capabilities to build a scalable pool and establish flocking mechanisms.	27
4.5	The task dispatching 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.	29

5.1	Layering federated manager instances into a tree architecture. . . . .	39
5.2	Creating cycles in the federation links between managers means that all managers in the cycle have access to all of the workers tracked by the managers in the loop. It becomes possible for a task to then be forwarded endlessly around the loop; therefore, a task should be marked with the number of hops it has made and once it has reached a configured limit, not be forwarded any further. . . . .	39
6.1	The results of a week long test monitoring the number of available worker agents. A Condor job was created to queue several thousand instances of memory-constrained worker VMs for execution, with a second script resubmitting the job as needed to refill the request. Kestrel was able to recognize roughly 4,500 simultaneous worker instances. . . . .	41
6.2	The differences in response times between the three grid sites. Only the site at CERN produced significant latency which could distinguish worker agents from that site. . .	43
6.3	Aggregate disk throughput for the two 4Gb Fibre Channel ports on Clemson's Palmetto cluster's disk array when starting 1000 VMs. The observed throughput of 154Mb/sec was well below the maximum throughput of the two channels. . . . .	44
6.4	The results of starting an IPOP/Condor pool of 1600 nodes in one batch . . . . .	46
6.5	The results of starting an IPOP/Condor pool of 1600 nodes in 200 batches . . . . .	46
6.6	Kestrel Pool of 1600 Agents . . . . .	47
6.7	Openfire Showing Available Agents . . . . .	48
6.8	Trial #1 executing "sleep 0" . . . . .	49
6.9	Trial #2, executing "sleep .5" . . . . .	50
6.10	Trial #3, executing "sleep 2" . . . . .	51
6.11	Trial #4, executing "sleep random(0,10)" . . . . .	52
6.12	The relative time for job completion using an increasing number of clustered manager instances. . . . .	54
6.13	The number of online, available, and busy workers according to the Kestrel manager during the course of the experiments. . . . .	55
6.14	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, only periodic bursts of tasks could be executed before several concurrency and networking errors would deadlock the manager, preventing further execution until the system could be restarted. . . . .	56
6.15	Number of PYTHIA events generated during simulations. The spike in the event rate corresponds with the new Kestrel version's introduction that allowed for more tasks to execute and for longer periods of time. . . . .	58

# List of Listings

3.1	A basic message stanza. The type value of “chat” indicates that this stanza is meant to be part of a thread of conversation, or chat, like one would have in a one-on-one chat in an instant messaging client. While a conversation may expect a reply, there is no guarantee provided by XMPP for a response stanza after sending a message stanza.	13
3.2	A demonstration of using an Iq stanza to retrieve roster information. Unlike message and presence stanzas, an Iq stanza is always guaranteed a response.	15
4.1	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.	22
4.2	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 <code>kestrel:tasks</code> 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 <code>kestrel:tasks:capabilities</code> node of the agent’s profile. The resulting list of features are the capabilities that the worker offers for use during match making.	24
4.3	A job submission stanza with two requirements. The <code>queue</code> attribute specifies the number of times that the job will be executed, where each instance is considered a single “task”. Various <code>requires</code> 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.	26
4.4	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.	29
4.5	A JSON formatted message used in the initial versions of Kestrel for submitting a command and cleanup task; in this case the commands <code>./run_program.sh</code> and <code>./restart_vm.sh</code> . The requirements that a worker agent had to meet in order to accept the task are given by the list containing <code>Python</code> and <code>STAR</code> . To indicate that multiple copies of the command should be executed, the <code>queue</code> parameter was supplied with the value 1000. The initial <code>type</code> value included in the message was used for routing the message to the proper message handler.	31
5.1	The code for determining the set of available workers to available tasks for a given job.	35

# Chapter 1

## Introduction

Cloud computing has sprung as a new paradigm for both enterprise and scientific application development and deployment. However, the term “cloud computing” remains vague and broad in application, with considerations by Foster [32] and Vaquero [74] notwithstanding, making it difficult to agree on a single, specific definition. Therefore, for the purposes of this work, the term “cloud computing” will be deemed to refer to the on-demand allocation of virtualized resources over a wide area network. Within this model, a cloud is established through instantiating virtual machines from one or more virtualization enabled grid sites, or other cloud providers, such as Amazon’s EC2 service [2].

Such an infrastructure provides several benefits for scientific computing users. Since the computational work will be done on a virtual machine instead of a physical machine, users are able to package their own environment suitable for the desired application by preparing a VM image based on a standard worker node VM and customizing it with the all of the required supporting libraries and programs needed for their application, independent of the available library support of the physical host machines. By tailoring VMs to applications, it is not as necessary for a job scheduler to perform complex matching between jobs and the capabilities of worker agents, as performed through the ClassAd match-making approach used by Condor [71]; each job may simply be tagged with the name of the VM configuration created for the specific application. In addition VMs provide better application encapsulation compared to regular operating system and cluster scheduling. Such encapsulation permits improved resource utilization via consolidation mechanisms and improved security by isolating user’s applications from each other [30]. From the resource owner’s point

of view, virtualized resources also enable greater flexibility in the system administration because multiple operating systems can be offered at the same time.

## 1.1 Virtual Organization Clusters

Such an arrangement for the use of the term “cloud” corresponds to the Virtual Organization Cluster (VOC) model [51, 52, 50], which enables the creation of virtual cluster environments that are compatible across sites, transparent to end users, implementable in both phased and non-disruptive manners, optionally customizable by Virtual Organizations (VOs), and designed according an an architectural specification. To establish a VOC, a Virtual Organization (VO) starts virtual machines over a wide area network through multiple providers, then a network overlay is created amongst the VM instances. Since the virtual machines used in the VOC can be spawned from a single image, VOC environments are nominally homogeneous across grid sites. The providers used for the virtual machines may include existing grid sites with virtualization capabilities and corporate cloud providers. It has been demonstrated that a VOC can be created using resources provided simultaneously by an Open Science Grid (OSG) site, Amazon EC2 instances, and CERN resources [69].

The VOC provides elasticity in its capabilities by automatically growing or shrinking the number of provisioned VMs in the cluster based on the state and length of the job queues, allowing the VOC to operate transparently to the end user without explicit resource reservation requests; the transparency also applies to both the VO deploying the VOC and other entities which choose not to use VOCs, which enables the addition of VOC implementations to existing production grid sites without disruption of current operational infrastructure.

The method by which the VOC is able to autonomically provision itself to adapt to changing workload demands is through the use of pilot jobs which obtain the physical resources required from other grid sites or cloud providers. A pilot job starts a virtual machine, which may then be dynamically configured, or “contextualized” [39], at boot time. The new VM once operational may then contact a dedicated server provided by the VO to locate the VOC and join the cluster and its overlay network. The mechanisms for creating the overlay network for spanning sites over a wide area network vary by implementation, but include IPOP [34], Violin [58] and ViNe [73].

However, the VOC must still rely either on the ability of the VM resources to lease scheduling

mechanisms from the physical site or on the use of virtual networks which enable the VO to run its own standard job scheduler. Dedicated grid interconnection middleware such as Globus [31] can be used to permit a VOC to join a grid as a site dedicated to the use of a single VO; other methods include direct submission mechanisms allowing VO members to submit workloads to the VOC without the use of grid middleware. While a general purpose scheduler such as Condor [71] may be used for this purpose, if the jobs to be executed by the virtual cluster do not require simultaneous interprocess communications, such as bag-of-tasks applications [23], then a scheduling system which can operate directly over a wide area network, such as the Kestrel system described herein, can be deployed without the need for a traditional, general purpose overlay network. For the purpose of this work, the cases where virtual networks are not used or cannot be used due to policy constraints by grid administrators and where a scheduler provided by the VO requires access to all VM resources regardless of the network characteristics are considered.

## 1.2 Job Scheduling and Distribution in Clouds

While a significant body of work has been done in operating system virtualization [22], virtual networks [73, 34] and provisioning of virtual machines [39, 67, 54], 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 [24].
- Virtual machine images need to be transferred to all sites involved as well as made available on all physical nodes through either staging of shared file systems [64].
- Networking of the virtual machine instances need to enable network agnostic job scheduling.
- Users need to be taught a new access mechanism or use existing tools that have been adapted for Clouds.

While existing tools [71, 27, 55, 77] can certainly evolve to be utilized in the context of a “cloud”, the work presented herein describes a new job scheduling framework developed specifically for use in cloud computing by not relying on direct network access between entities. Named Kestrel [70, 69],

this framework is based on the Extensible Messaging and Presence Protocol (XMPP) [61, 62], a robust standard for messaging used in large scale instant messaging deployments, and recently adopted as the only approved instant messaging protocol for the Department of Defense [78]. The assumptions and design principles made to create Kestrel were that traditional push models used in job scheduling did not meet the requirements of the users who developed their own job overlays on top of the traditional grid scheduling frameworks. Instead users developed their own pull model akin to the BOINC [20] middleware and solely used the grid scheduling to probe remote providers and request resources. This probing and request mechanism is very similar to the use of a cloud IaaS API. Examples of such pull based models, also known as pilot job frameworks, are PANDA [43], DIRAC [72] and GlideinWMS [65]. In addition, virtual networks will become extremely difficult to create due to the shortage of IPv4 addresses and the slow adoption of IPv6. Therefore, users increasingly will be faced with virtualized resources started behind multiple layers of NAT. Push based models together with lack of trust from the site to the image producer will result in a hostile environment for the virtualized resources. Kestrel aims to tackle this problem by looking at solutions from the instant messaging world and XMPP in particular. XMPP was designed for hostile networking environments as well as intermittent, heterogeneous resources. XMPP has also demonstrated its ability to scale to hundreds of thousands of clients through deployments such as Google Talk and Facebook Chat, a scale that clouds will soon reach with the continued deployment of multi-core systems.

### 1.3 Motivations

The motivating scenario is derived from experience using virtual machines on Clemson University's 10,000 core Palmetto cluster in KVM user mode, as per administrative policy. The result is a networking environment in which each virtual machine is behind its own layer of NAT, in addition to a NAT boundary imposed on the entire cluster. While it is true that different networking arrangements, such as bridged networking, would alleviate such conditions, experience shows that is not always an option. It follows that if a system which can perform adequately for a user's job requirements under such restrictions can be created, then it would also perform in more favorable network environments.

The primary contributions of this work are:

- XMPP is presented as a fundamental communication protocol for building distributed systems

for scientific computing.

- Highlight the design decisions behind the multiple iterations of Kestrel, based on experimental results and feedback from partner organizations.
- Performance of Kestrel, and XMPP-based systems in general, are demonstrated through micro-benchmarks and response delays for systems spanning grid sites.
- Large scale results from real-world scientific computing usage are presented showing Kestrel's potential as a cloud-based job scheduling overlay framework.

The remainder of this work is organized into several chapters covering related works (Chapter 2) and background material on XMPP (Chapter 3). The architecture of Kestrel and the design choices behind its implementation are discussed in Chapter 4. Performance data and results from real world usage in collaboration with STAR are presented in Chapter 6, with final conclusions drawn in Chapter 7.

## Chapter 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 [71, 29]. 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. To do so, submit and execute nodes are directly connected to each other, bypassing the system's central matchmaker, once a job has been matched with an execute node. While this arrangement is adequate when all of the compute elements are in the same subnet behind a NAT boundary, using Condor with machines on both sides of a NAT boundary introduces complications. Falcon [57] has been designed to achieve high dispatch rate on large scale systems over-performing Condor in situation where resources are pre-allocated and the scheduling decision is kept to a minimum. Kestrel has similar features since a Kestrel pool is created once all agents have started in the Cloud, however Kestrel has a traditional scheduler based on Condor matchmaking mechanism.

Several workflow engines [26, 55] and scripting languages [77] have been developed to run jobs efficiently on computational grids; they have mostly been designed with Globus [31] grids in mind where the middleware uses gatekeepers to access resources. These tools can be adapted to use Clouds and provide a way to optimize workflow based on Cloud utility pricing [27] but they may not fare well in infrastructure without inbound network connection to virtualized resources and without shared file system.

The BOINC [20] middleware and existing pilot job frameworks [43, 65, 72] 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 the closest to Kestrel. Kestrel differs in the use of XMPP which is a standard for messaging and offers great scaling.

A solution to allow Condor to operate with NAT traversal is the Internet Protocol over Peer-to-Peer system, or IPOP [34]. 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 [35]. Since it is a full networking stack, IPOP provides TCP, UDP, and TLS/SSL network transports; end-user applications may run unmodified while using IPOP. It was used to build an autonomic Cloud overlay following the VOC model in [19]. Other virtual networks have been studied but not used in Cloud overlays [58, 73].

A second solution for using Condor with NAT traversal is the Generic Connection Broker (GCB) [7]. GCB provides a replacement library for Berkeley sockets which allows a program to make normal socket I/O calls, but with the results brokered by an intermediary server outside of the NAT boundary. The main advantage of this arrangement is the same as that of IPOP, namely that a full networking stack is provided which allows an existing application to use the NAT traversal capabilities without the need for rewriting the application; a simple recompilation may suffice. One disadvantage to this approach include causing all nodes behind the NAT boundary to share the same IP address, namely that of the broker server [4]. The GCB also does not provide any presence information about the nodes for which it brokers communications.

While working on optimizing meander line radio frequency identification (RFID) antennas, Weis and Lewis [76] developed a parallelized computation system based on XMPP. The 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 specific purpose, ad-hoc grid system. Kestrel uses a similar architecture, but is intended as a more general purpose job distribution and scheduling framework. At a similar time XMPP was used in Bioinformatics web services [75] and used within the Taverna workflow engine [55] showing that other communities are considering XMPP as a suitable protocol for messaging and using it to develop new services and mechanisms.

## Chapter 3

# XMPP: Extensible Messaging and Presence Protocol

At the start of 1999, an open instant messaging protocol was released by Miller under the name Jabber [9], along with a server implementation, *jabberd*. Unlike the various commercial instant messaging protocols provided by America On-Line (AOL) [3], Microsoft [15] and Yahoo [17] that required reverse engineering to create third-party clients, Jabber was an open specification based on streaming Extensible Markup Language (XML).

In the following year, the Internet Engineering Task Force (IETF) finished work on both a model and requirements for an *Internet Messaging and Presence Protocol* (IMPP) [42][41]; no implementation was produced by the working group, and in 2002 the Jabber community had organized enough to submit the Jabber protocol as a proposed implementation for IMPP, but with the name changed to the *Extensible Messaging and Presence Protocol* [61][62]. In early 2004, the XMPP specifications were formalized and approved by the IETF as Proposed Standards [9]. The protocol continues to be updated and expanded by the XMPP Standards Foundation (XSF).

### 3.1 The XMPP Network

XMPP defines two wire-format protocols to enable server to client and server to server communications. The result is a decentralized client-server architecture, similar in structure to the

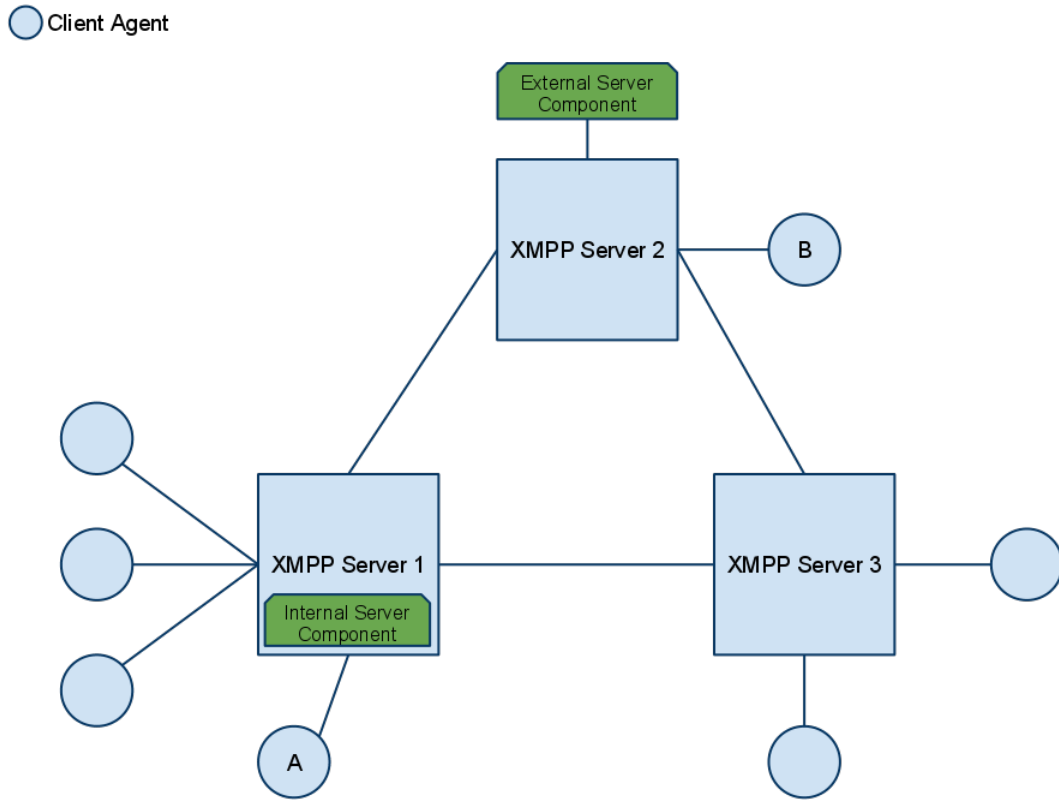


Figure 3.1: A demonstration of a basic XMPP network, including client agents, internal and external server components, and XMPP servers. Communications from entities associated with one server are not routed by any intermediary server. Messages sent from the client marked “A” to the client marked “B” will always be sent from server labeled “1” to the server labeled “2”, never along the path from “1” to “3” to “2”.

one used for email with the *Simple Mail Transfer Protocol* (SMTP). Each server maintains long-lived connections to many clients, and optionally connections to other XMPP servers. For the most popular server implementation, ejabberd [5] [18], accepting upwards of twenty-thousand clients on a regular basis is possible [10], particularly when implementation specific clustering is used to spread server loads across multiple physical machines [21]. Figure 3.1 shows a snapshot of the daily usage of the the Russian Jabber.ru XMPP server, which processes approximately fourteen thousand client connections at any given time. Other large XMPP installations, notably Google’s GTalk [8] and Facebook’s chat [6] services, are able to accept, through clustering, several million clients at once.

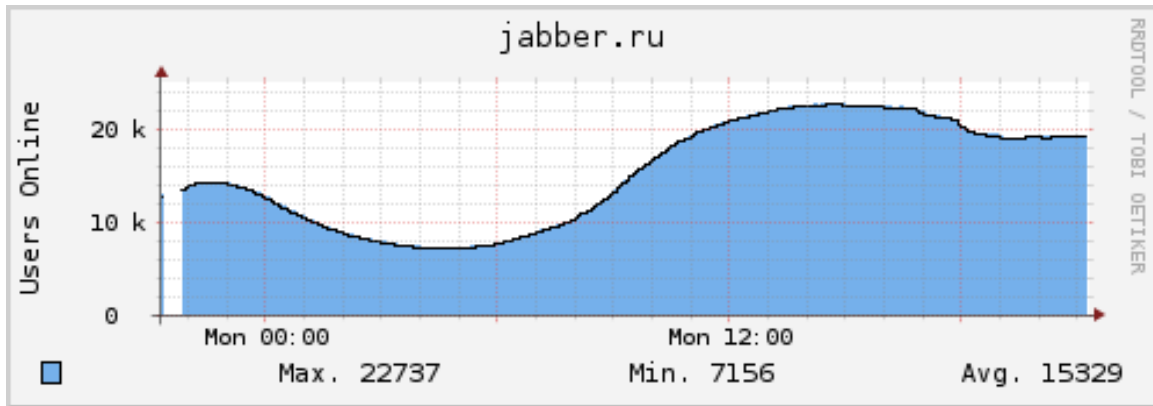


Figure 3.2: Daily usage graph from the Russian XMPP service provider Jabber.ru. Average use is around 22,000 simultaneous client connections during peak daytime hours.

While many server implementations provide the ability to cluster, XMPP itself does not define a clustering mechanism. Instead, it provides a mechanism for separate server installations to *federate*, allowing communications between clients hosted by either server. Unlike the federation allowed by SMTP, XMPP mandates at most two servers will process any message between two clients; in SMTP, the number of intermediate servers is unbounded. The limit on intermediate servers is aimed at limiting message spoofing and other types of spam common in the SMTP network. Figure 3.1 shows the various configurations an XMPP network may take.

## 3.2 Jabber Identifiers (JIDs)

Each entity within an XMPP network is addressable using one or more *Jabber Identifiers*, or JIDs [61]. The JID used by any connected entity is globally unique, based on the Domain Name System (DNS). JIDs commonly look similar to e-mail addresses, but with an additional section: `localpart@domainpart/resourcepart`. The `<domainpart>` portion of the JID is the only required section. JIDs for servers consist solely of the domain name exposing the XMPP service (DNS service, or SRV, records can be used to allow a server to support a domain different from the domain of the machine on which it is running [61]). User accounts on a server are identified by the `<localpart>` section of the JID; when only the form `localpart@domainpart` is used, the result is called a *bare JID*. When a client connects to a server using an account's bare JID, then the connection is assigned a resource value to distinguish it from any other connections already established by

that user account; a JID with all three sections is referred to as a *full JID*, and it is the full JID that is globally unique. To demonstrate, if a client connects to the `kestrel.cs.clemson.edu` server using the bare JID `user@kestrel.cs.clemson.edu`, then the particular connection established may be assigned the full JID `user@kestrel.cs.clemson.edu/133742`.

### 3.3 Types of XMPP Agents

Within the network are four types of entities, or agents, which can be marked on a continuum of scalability. The first is the regular XMPP client; many implementations for clients exist, notably for use with instant messaging by a human. However, many client agents exist as automated programs, or *bots* [62], to provide services to other clients in the network. Each client connection is associated with a unique full JID (server implementations are free to decide to reject duplicate full JIDs connections or terminate the existing connection in favor of the new one [61]). Under the XMPP Design Guidelines, clients are typically kept simple, offloading as much functionality as possible onto the server, making it easier for multiple implementations to work together [60].

The next step up in terms of scalability is the external server component. XMPP acknowledges that servers will not always provide built-in support for all desired features; thus, servers may be easily extended through components. External components are separate programs, possibly running on a different machine than the server, that are trusted by the server. A standard wire-protocol is used between servers and components, making external components interoperable with any server implementation. Each component is assigned a subdomain from domain used by the server, and all JIDs using that subdomain are forward by the server with no additional processing to the component; it is left to the component how to interpret the JIDs provided to it. For example, a multi-user chat component will treat the username portion of a JID as the chat room name, and the resource as a participant's nickname within the room, i.e. for the component `conference.jabber.org`, the JID `sleek@conference.jabber.org/Lance` would refer to the participant named "Lance" within the "sleek" chat room on the `jabber.org` server.

The greater scalability afforded by a component is due to the server's lack of responsibility for the JIDs used by the component. For a normal client, the server is tasked with maintaining a roster of subscriptions between the client and other entities in the network. For human centered clients, this relationship works well due to typically small roster sizes for which the server is optimized. As

roster sizes increase, two scalability issues emerge. The first is that the optimization assumption of small rosters is violated, potentially impacting performance based on the storage mechanisms used. The second is that a client's roster is sent to the client in a single stanza as part of the initialization procedure of the XMPP session. Stanzas are not interruptible – all other stream communications to the client are blocked while the roster is transmitted [47]. For rosters with several thousand entries, this places a severe penalty on start up times for agents and on the amount of networking traffic used [48]. For a component, no roster is maintained by the server, leaving that responsibility to the component itself.

The final two entity types are internal server components and servers themselves. Internal server components receive the same benefits as external components, but they are implemented in a server-dependent fashion. Their main advantage is the ability to hook directly into the server's internals to provide services that would not be as easily done otherwise. The final tier of potential scalability is to create a custom implementation that speaks the server to server protocol, eliminating the need for many features provided by traditional servers.

### 3.4 Stanzas

Communications between an XMPP client and server are done using two streams of XML data, one for each direction of communication. Each stream starts with the `<stream:stream xmlns="etherx.jabber.org/streams" />` element, and all *stanzas* sent or received during the session will be top-level children of one of these two elements. At the conclusion of an XMPP session, the stream elements are terminated, creating two valid XML documents [61]. The sections of XML transmitted during the session are not full XML documents, and as such are referred to as stanzas instead. Each stanza is guaranteed to possess certain attributes, namely the attributes `to` and `from` which provide the JIDs identifying the sender and recipient of the stanza [61]. In general, the `from` attribute of a stanza is set by the server in order to combat impersonation [61]. While there are several elements that may appear at the top-level of a stream element during the course of establishing the session, there are three main elements that are defined to be stanzas: `<message />`, `<presence />`, and `<iq />`. Each type of stanza differs in both routing and response behaviors.

### 3.4.1 Message

The most basic of the three stanza types, a `<message />` stanza is used to push data from one agent to another. While this is typically done in a “fire and forget” manner, some servers support more advanced message processing that can provide better guarantees of message delivery [46]. However, even with reliable delivery, there is no guaranteed response from a `<message />` stanza. The most common use for `<message />` stanzas is as an envelope for messages from a chat session between two agents. Other major uses include broadcasting event notices from a publish-subscribe mechanism.

Listing 3.1: A basic message stanza. The type value of “chat” indicates that this stanza is meant to be part of a thread of conversation, or chat, like one would have in a one-on-one chat in an instant messaging client. While a conversation may expect a reply, there is no guarantee provided by XMPP for a response stanza after sending a message stanza.

```
<message to="eduardo@fosters.lit"
        from="cheese@fosters.lit"
        type="chat">
  <body>I like chocolate milk.</body>
</message>
```

### 3.4.2 Presence

The `<presence />` stanza is used to multicast data, namely presence and status information, to other agents that have subscribed for it. Each `presence` stanza may include a status message, a value indicating the current state of the agent (such as free for chat, away, do not disturb, etc), and a priority value indicating the relative importance of the connection amongst other connections from the same bare JID. Some servers use the priority value when routing stanzas by sending stanzas addressed to the bare JID to the resource with the highest priority value. While it is possible to include additional information in a `<presence />` stanza, it is highly discouraged. Presence stanzas are typically the most frequently stanzas used within an XMPP network, and thus bandwidth usage must be considered and minimized [62].

Presence stanzas whose `type` attribute value is in one of `subscribe`, `subscribed`, `unsubscribe`, or `unsubscribed` are used to create and remove presence subscriptions, and thus the roster. Subscriptions are directional, and it is not required to establish mutual subscriptions. When a presence stanza is sent to the server without a JID for particular recipient, the server will

broadcast the stanza to all connected agents that have subscribed to the particular agent's presence. The process for establishing a subscription is done by first sending a `<presence />` stanza possessing a `type` value of `subscribe`, or `<presence type="subscribe" />`. If the receiving entity approves the request, it will respond with a `<presence type="subscribed" />` stanza. The receiving entity may then choose to repeat the process in reverse to subscribe to the initial agent's presence. If the entity refuses the request, a `<presence type="unsubscribed" />` stanza is issued instead. Ending a subscription is done by sending a `<presence type="unsubscribe" />` stanza to an entity. The receiving agent will then respond with a `<presence type="unsubscribed" />` stanza.

### 3.4.3 Info/Query (IQ)

The final stanza is the `<iq />` stanza, which is short for "Info/Query". In contrast with `<message />` stanzas, `<iq />` stanzas are guaranteed to have a response, either from another agent, or the server itself. There are four types of `<iq />` stanzas, two for requests and two for responses. An initial `<iq />` request may possess a `type` value of either `get` or `set`. A `get` type is roughly equivalent to the HTTP `GET` method. These requests are meant to request some form of data from another agent, usually without other side effects. A `set` type corresponds to the HTTP `POST` method, and is used to send information to another agent, with the usual intention for it to be saved or perform some side effect.

When an `<iq />` is successfully processed, a response `<iq />` with a `type` value of `result` will be returned. A response with a `type` value of `error` indicates that an error occurred somewhere during the exchange, such as the intended recipient was not found or the recipient could not handle the request. It is possible to associate the initial `<iq />` stanza with its response by examining its `id` attribute. Every stanza used by XMPP includes an `id` attribute, and that value is used as the `id` value of a reply stanza.

## 3.5 Security Considerations

In order to provide basic security to XMPP communications between clients and servers, the Transport Layer Security (TLS) [28] and Simple Authentication and Security Layer (SASL) [44] must be used. The use of TLS provides channel encryption for the XML stream to prevent tampering

Listing 3.2: A demonstration of using an Iq stanza to retrieve roster information. Unlike message and presence stanzas, an Iq stanza is always guaranteed a response.

```
<iq from="mac@fosters.lit/home"
  type="get"
  id="42">
  <query xmlns="jabber:iq:roster" />
</iq>
<iq to="mac@fosters.lit/home"
  from="mac@fosters.lit"
  type="result"
  id="42">
  <query xmlns="jabber:iq:roster">
    <item jid="bloo@fosters.lit" subscription="both" />
    <item jid="frankie@fosters.lit" subscription="both" />
    <item jid="eduardo@fosters.lit" subscription="both" />
  </query>
</iq>
```

and eavesdropping [61]. While it is a requirement to support the use of TLS in both client and server implementations, it is still possible for a server administrator to make its use optional. Client agents requiring a secure channel must negotiate the use of TLS using the STARTTLS [61] stream feature. The method of encryption used by TLS is commonly based on digital certificates, which must be validated by a client agent when presented to ensure a valid chain-of-trust for the server's certificate; this may be done either by explicitly comparing the presented certificate against a known, trusted version, or checking that the certificate has been issued by a trusted Certificate Authority (CA).

While TLS prevents tampering, it does not guarantee the identity of the client initiating the communications. SASL provides several mechanisms for allowing a client to authenticate an identity. These mechanisms include:

1. PLAIN: The client's password is transmitted in base64 encoding, relying on the underlying stream's channel encryption for security.
2. DIGEST-MD5: A more secure form of authentication whereby the client is challenged by the server to provide a hashed version of the user's password, along with the username, server name, security realm, etc [40]. The main advantage is that the password itself is never transmitted.
3. EXTERNAL: The SASL EXTERNAL mechanism allows for a client to authenticate itself using a digital certificate.

While each connection between entities in the XMPP network may be encrypted, the core

XMPP standards do not define a method for end-to-end encryption of data. Several methods for achieving end-to-end encryption have been proposed, but a consensus has not yet been reached. An older method is documented in XEP-0027: Current Jabber OpenPGP Usage [49] whereby message or presence stanza contents may be encrypted using an OpenPGP key. Unfortunately, IQ stanza contents can not be encrypted in this manner. A newer proposal uses IQ stanzas to implement a secondary communications channel, whose contents is encrypted using TLS [45].

## 3.6 XMPP Extension Proposals (XEPs)

While the core XMPP specifications are maintained by the IETF, the XMPP Standards Foundation is responsible for maintaining additional extensions to the specifications and protocols, adding new features or documenting current best practices. Such extensions are referred to as XMPP Extension Proposals, or XEPs, each of which is assigned a four digit number starting from 0001. XEPs are grouped into several categories, including *Draft Standard*, *Final Standard*, *Experimental*, *Deferred*, among others [59]. A XEP in the final standard state is considered finished, and will not be updated further. A draft standard XEP is considered stable and reliable for implementation, while an experimental XEP is encouraged for implementation but may still undergo significant revision. Several XEPs are considered fundamental to providing robust and useful XMPP services.

### 3.6.1 Service Discovery: XEP-0030

It is useful and necessary for an entity within an XMPP network to query other entities about their identities, supported features, and any related entities within the network. For this purpose, XEP-0030 Service Discovery [37] defines the use of “disco”, short for service discovery, using `<iq />` stanzas. Disco is broken into two categories: information and items. The information portion is used to query and report identities and feature support. The items portion is used to return a list of JIDs related to an agent.

An important concept introduced by XEP-0030 is the idea of a “node” for a JID. A node refers to an aspect, or facet of the services provided by a JID [37]. For example, an agent may provide two distinct services. By specifying a node, the disco query can be limited to only requesting information about one particular service.

An identity in XMPP is a combination of a category, type, name, and lang. The

category and type values are selected from a predefined set that is used to identify the agent as a human facing client, or a server component providing a gateway service to another instant messaging network. Features are specified by listing the recognized namespaces used by the supported features.

An agent may also be responsible, or be the authoritative reference, for other JIDs used in the network. One such case would be a multi-user chat component's responsibility for JIDs for the chat rooms it provides. Each JID provided in the query results may also include a node name to refer to particular services or information resources provided by a single JID.

### **3.6.2 Entity Capabilities: XEP-0115**

While service discovery provides a useful and generic mechanism for querying for features supported by other entities, the bandwidth required to perform to walk through the various nodes provided by a JID to build a complete disco profile can be considerable when working with many agents. XEP 0115, Entity Capabilities, allows for simplifying the disco process by including a hashed version of a node's identities and features in a `<presence />` stanza. When an agent receives such a presence notice, it will query the sender to retrieve the full contents matching the given hash value, provided that the hash value is one that has not been seen before. Under the assumption that many agents will have identical sets of identities and features, due to using the same underlying client libraries, the issue of excess bandwidth consumption is minimized.

## Chapter 4

# Kestrel

Many-Task Computing (MTC) applications span a broad range of possible configurations, but “utilizing large numbers of computing resources over short periods of time to accomplish many computational tasks, where the primary metrics are in seconds” [56] is one of the emphasized aspects. Marshalling and releasing computational resources with the temporal granularity needed for such applications is problematic on the grid and in overlays such as Virtual Organization Clusters (VOCs) where compute nodes can be created and destroyed on demand, without warning, by either the initial requester or a grid administrator. Kestrel was designed to explore methods of creating a fault-tolerant, scalable, and cross-platform job scheduling system for heterogeneous, intermittently connected compute nodes. While a VOC is nominally composed of homogeneous resources through the use of a single VM image for its constituent nodes, such arrangements are neither guaranteed nor required.

Kestrel addresses the problem posed by interrupted connectivity of both computing agents and user agents by building on XMPP. The presence notification system provided by XMPP allows Kestrel manager nodes to track resource availability in the compute node pool in real time. Kestrel can immediately remove computing node from a list of available nodes, preventing the scheduling of jobs on nonexistent machines. Other intermittently connected agents do not necessarily have to be of a computational nature. For example, a small swarm of robotic agents with network capabilities could be joined in a resource pool where scheduled jobs are physical actions to be performed. The scale of current instant messaging systems indicates that an XMPP based framework would allow for the creation of extremely large pools of heterogeneous agents for real-time computing. The cross-

platform criterion was achieved by implementing the system in Python using the SleekXMPP [33] library.

## 4.1 Architecture

Based upon the traditional master-worker architecture, a Kestrel pool is made from four classes of agents: clients, workers, managers, and servers. Logically, the client, worker, and manager entities form a star network, with a manager as the hub; however, since XMPP is used, the actual topology will look as described in section 3.1. It is possible for multiple entities to reside on the same physical or virtual machine. Since all agents connect to an XMPP server for routing communications instead of connecting directly to each other, the issues of NAT traversal are eliminated; the result is an overlay network that allows agents to communicate freely irrespective of the underlying networking setup, whether on physical or virtual machines.

### 4.1.1 Well-Known JIDs

Since the Kestrel manager is an XMPP component, it may be addressed by many JIDs with different username portions. For example, the JIDs `pool@manager.example.org` and `job_42@manager.example.org` will both be delivered to the Kestrel manager. Kestrel reserves two particular usernames for interaction with clients and workers. The first is `submit` which may accept 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 an IM client as shown in figure 4.2

■ Potential Intermediary XMPP Server

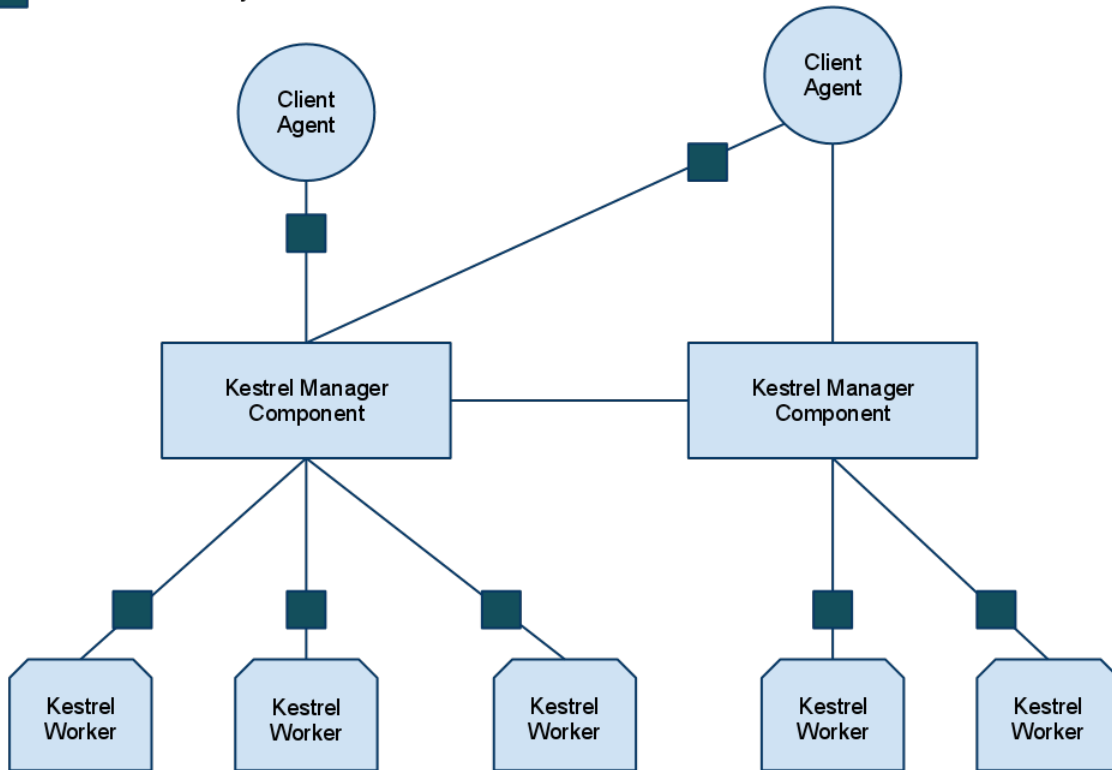


Figure 4.1: The logical network communication paths for a Kestrel pool. Messages may be routed through at most two XMPP servers for any of the paths shown, including the server exposing the manager components. These potential additional servers are indicated by the square on each path. Using XMPP's built-in federation, worker instances can be easily distributed across multiple XMPP servers.

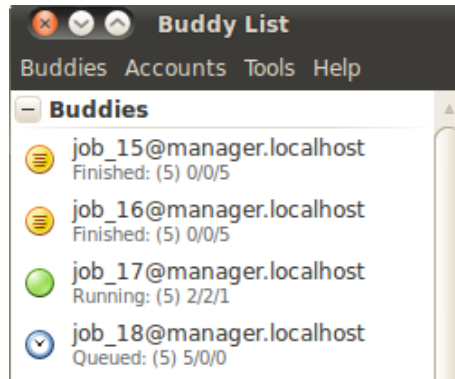


Figure 4.2: Monitoring job statuses through a normal instant messaging client. Once accepted by the manager, jobs request to be added to the user’s roster to deliver current status updates. Running jobs display themselves as available while queued jobs are displayed as away. For finished jobs, an extended 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 it from the roster.

## 4.2 Clients

Client agents are used by end users for submitting and managing jobs. As such, a client implementation can be a minimal XMPP client with just the logic needed to submit jobs and status requests, but more full-featured and existing clients such as Pidgin [11] or GTalk [8] can also be used. As part of the typical use case, using the client implementation supplied with Kestrel, client agents join the system intermittently and for just long enough to send a single command to the manager and receive a response. Clients that remain connected will receive presence subscription requests from JIDs representing the user’s jobs, which send status updates as tasks are dispatched or completed, allowing for continuous status monitoring.

### 4.2.1 Job Management

Creating and monitoring jobs can now be carried out through the command line instead of having to rely upon an instant messaging client such as Pidgin [11] or Adium [1]. However, as shown in figure 4.2, important job status information can be received through a client program as part of the job JID’s presence status updates, such as the number of queued, running, and completed tasks. Jobs that are accepted, and assigned a JID, may be 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

Listing 4.1: 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.

```
<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>
```

job and terminate running tasks. From the command line, jobs may be submitted, canceled, and queried for their current status using the commands shown in table 4.1.

The protocol for submitting a job, shown in figure 4.3, resembles the stanza used to initiate a task, but may also include a variable number of `<requires />` elements. These requirements may be 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 simply the public interface for Kestrel through other 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 `action=cancel` in a job stanza similar to that shown in figure 4.3, a job cancellation request may be submitted.

At any point, 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 4.1 to the JID `submit@manager.example.org` 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

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

Table 4.1: Available commands for the command line client.

job. Issuing the status request to `pool@manager.example.org` instead will return the number of online, available, and busy workers.

## 4.3 Workers

A worker is also an XMPP client connection, and is little more than a wrapper for executing command line statements. Having a simple worker implementation is encouraged by the XMPP Design Guidelines [60] which states that functionality should be kept in servers when possible. As such, running a Kestrel worker agent is a very lightweight process, allowing a worker to be easily included even in virtual machines with little available memory. Trials using a Micro Core Linux [66] based virtual machine have succeeded in running worker agents with a total memory usage of 36Mb for the entire VM. Each worker agent is expected to maintain lengthy connections with the XMPP server so that tasks are able to execute during a single connection session. However, workers are not expected to achieve 100% uptime. A loss of connection while executing a task will cause the server to broadcast an offline presence that will alert the manager to reschedule the task if necessary.

### 4.3.1 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 4.3. Workers attempt to join the pool by subscribing 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 [37] as shown in figure 4.2 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 [37]). Every Kestrel worker will advertise its support for executing tasks in a Kestrel pool by including the

Listing 4.2: 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.

```
<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>
```

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 add a noticeable increase to 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 [38] 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.

### 4.3.2 Task Execution

The actual programs executed by workers should typically be small shell scripts that manages the task’s execution cycle, such as downloading input data and uploading any output. These scripts will always receive a *final* parameter which is the task’s ID value; however, additional pa-

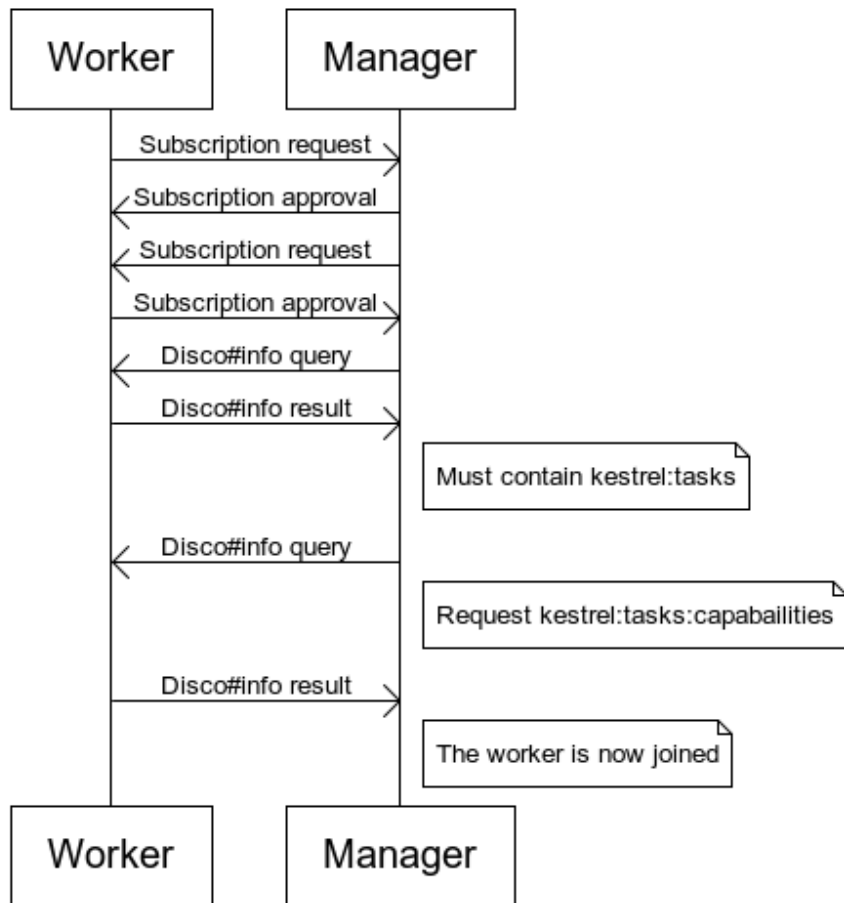


Figure 4.3: The presence subscription and service discovery process for joining the Kestrel pool. A worker first initiates a bi-directional presence subscription with the manager, allowing the manager to know when the worker is available or offline. The manager then queries the worker’s service descriptions to verify that the agent is a worker, and to find the capabilities the worker provides that may be used in match making.

Listing 4.3: 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.

```
<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>
```

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, 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 does not. While data transfers may be included in a future release using XMPP’s file sharing capabilities, `wget` and similar command line utilities are recommended for now. Transferring files is a largely solved problem with scalable solutions. The suggested method is to run a HTTP server (or server cluster) to distribute input files. The HTTP protocol supports range queries to retrieve sections of a file if needed. A simple upload form processor may be used to handle receiving output data from workers.

Once a task’s command has been executed, an optional second command may be executed for cleanup purposes. For 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.

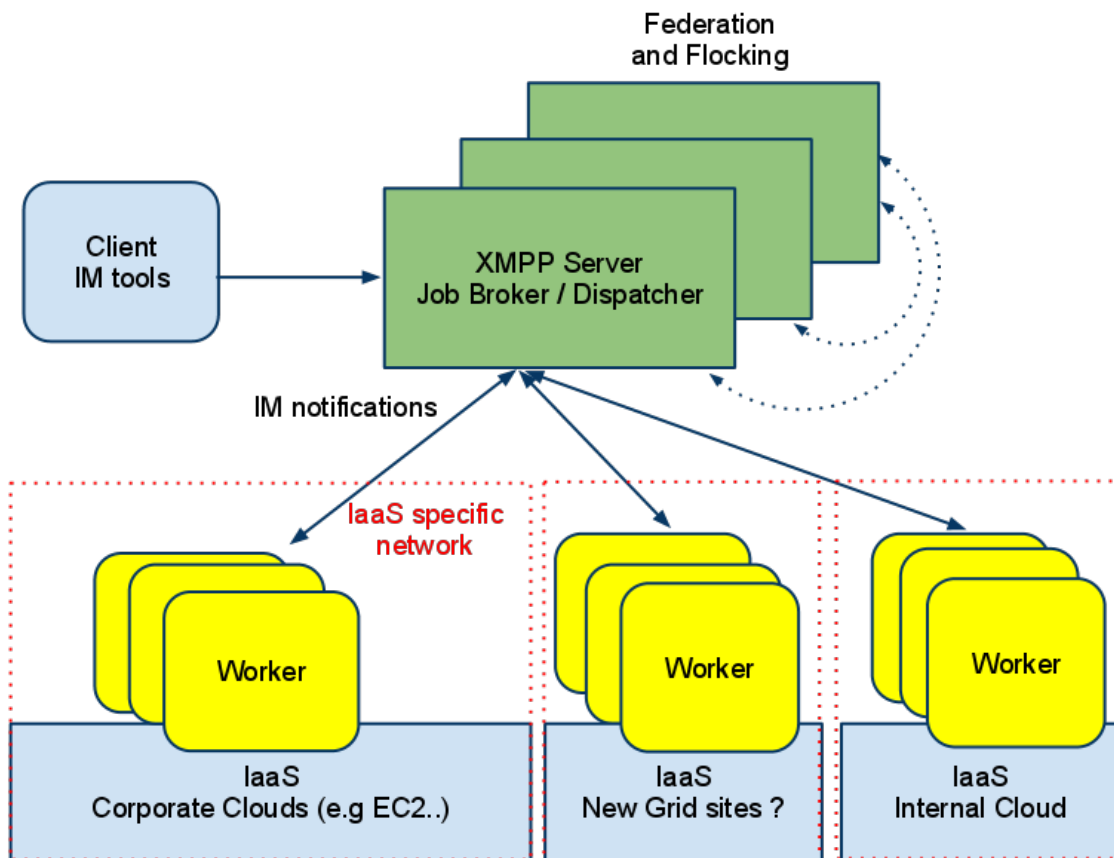


Figure 4.4: Kestrel builds a scheduling overlay on top of Cloud providers. Virtual machines instances containing the Kestrel worker agent can be started on any Cloud provider using multiple provisioning systems. Once started the instances join the Kestrel pool irrespective of the networking setup used by each provider. The Kestrel agents initiate the connection and setup a instant messaging channel to receive tasks. The Kestrel manager can make use of the XMPP federation capabilities to build a scalable pool and establish flocking mechanisms.

## 4.4 Manager

The Kestrel manager is an XMPP component in order to scale to handle several thousand worker agents [48]. Manager instances are typically run on the same machine as the XMPP server, but could also be placed in a VM in a cluster alongside the worker VMs. While the current backend for the manager is a SQLite [36] database which can not be shared easily by multiple processes, using a larger database or a hash value store such as Redis [63] 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 would create a clustered manager that still 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.

### 4.4.1 Scheduling and 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 4.5. 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 (as shown in figure 4.4), 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.

As part of the new requirements presented with carrying out the STAR experiment, a new task attribute was 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, it 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 for cleanup scripts to shutdown and restart the virtual machine without causing the manager to treat the shutdown as a network failure and reassigning the task to another worker.

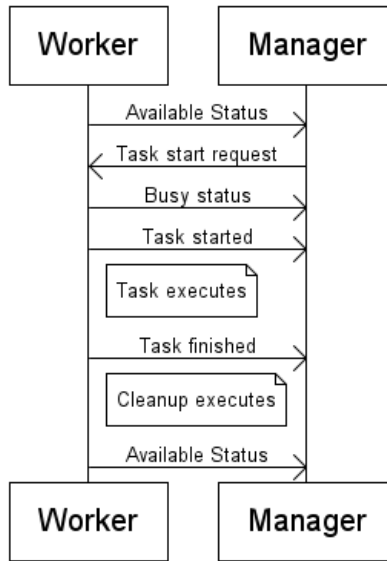


Figure 4.5: The task dispatching and execution process. A match making request is triggered when the manager receives an available presence from a worker. If a matching job is found, a task is marked pending and sent to the worker; if an error is returned or a timeout occurs, the task is returned to 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.

Listing 4.4: 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.

```

<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>
  
```

### 4.4.2 Responding to Worker Failures

Given that it is expected for worker agents to disconnect for various reasons, whether it be network unreliability or faults in the worker VM itself, the Kestrel manager must respond gracefully to such outages. These failures may occur during one of four states:

1. The worker was idle without any assigned task. In this case, the manager simply removes the worker from consideration for new tasks.
2. The worker was executing a task. Here a policy must be defined to determine how to treat the task that was being executed. The approach currently taken by Kestrel is to discard the current progress of the task and reassign it to the queue to be executed again. However, it is possible that the worker's unavailability was caused by a transient network disruption and that the worker would soon become available again. In situations where such scenarios are expected to be common, an alternative policy to delay the reassignment of the task back to the queue may be useful.
3. The worker was executing a task cleanup command. The purpose of the cleanup command is intended to allow for cases where a VM is restarted to free disk space on the physical machine. The only actions required by the manager is to mark the worker as unavailable for future tasks.
4. The worker had been assigned a task, but had not responded to the request. Here the disconnection occurred during the window between the time the manager issued a task request to the worker and the worker's receipt of the request. The task request itself is timed by the manager so that if a response is not received within ten seconds the task is inserted back into the queue.

## 4.5 Evolution of the Implementation and Lessons Learned

During the two years of development, Kestrel underwent significant revisions to the underlying architecture, each time re-purposing more of the existing XMPP technologies for the tasks of job scheduling and distribution. Kestrel has thus moved from simply treating an XMPP connection as a basic socket connection with a few added features to leveraging several existing protocols to allow for interoperability with other XMPP agents, service discovery, and using standard formats for data exchange. As part of this process, the underlying XMPP library for Python used by

Listing 4.5: A JSON formatted message used in the initial versions of Kestrel for submitting a command and cleanup task; in this case the commands `./run_program.sh` and `./restart_vm.sh`. The requirements that a worker agent had to meet in order to accept the task are given by the list containing `Python` and `STAR`. To indicate that multiple copies of the command should be executed, the `queue` parameter was supplied with the value `1000`. The initial `type` value included in the message was used for routing the message to the proper message handler.

```
{"type": "submit",
 "command": "./run_program.sh",
 "cleanup": "./restart_vm.sh",
 "requirements": ["Python", "STAR"],
 "queue": 1000}
```

Kestrel, SleekXMPP [33], was improved and expanded to provide support for the features required by Kestrel; complex functionality unrelated to job distribution has migrated down to the library level, such as advanced support for dynamic handling of service discovery requests and external storage for component rosters.

#### 4.5.1 JSON over XMPP Message Stanzas

The original architecture called for using JavaScript Object Notation (JSON) [25] to encode application specific information and to then use XMPP message stanzas to carry the information between the manager and worker agents in the Kestrel pool. Figure 4.5 shows an example of a basic job submission under this model. The main advantage of this method was that few changes in infrastructure were required. Common XMPP client programs such as GTalk or Pidgin could send JSON formatted messages directly, and support for parsing the JSON messages already existed as part of a class project that served as the precursor to the Kestrel project (under the same name). While workable, the design had several disadvantages that necessitated changes during the summer of 2010 STAR experiment.

The first disadvantage was that combining JSON data with XMPP is an unnecessary step that adds overhead, both in terms of parsing time and the increase in complexity from managing two data formats. The XMPP Design Guidelines [60] provided by the XSF [16] dictates that XML should be used for any sub-protocol implemented on top of XMPP for this reason.

The second shortcoming, the cause of several performance and stability issues, was the reliance on normal XMPP message stanzas for all communications between agents in the pool. XMPP message stanzas do not provide strong guarantees of delivery with standard server installations;

rather, they are treated as “fire and forget” payloads [62] with no expectation of a response in receipt. Ensuring consistency between Kestrel’s central manager’s view of the state of the worker pool, in particular those workers with pending jobs, and the actual state of the pool became difficult with large numbers of workers over an extended period of time.

#### 4.5.2 Customized IQ Stanzas

To resolve the deficiencies of the JSON over message stanza model, the communication channels were moved to use custom XML payloads in IQ stanzas. The main benefit of this model was the guaranteed response for any IQ stanza sent. In the case of the recipient disconnecting while the IQ is broadcast, the XMPP server itself will respond with an error IQ stanza on behalf of the disconnected agent. In addition, the SleekXMPP library provides a built-in mechanism for blocking execution while waiting for an IQ response, with an optional timeout value to handle cases where the recipient received the IQ from the server, but does not respond. In addition, in conjunction the customized XML payload, the protocols described in sections 4.4.1 and 4.2.1 were developed to manage the various work-flows required to dispatch tasks and manage jobs. As a final improvement, a SQLite [36] database was used to store all application state using SQLAlchemy [12]. A limitation of SQLAlchemy with SQLite was only a single thread may access the database. All operations touching the database were therefore serialized, which introduced delays in initializing the pool when large numbers of workers contacted the manager at once.

## Chapter 5

# Clustering and Federation

The initial form of Kestrel as described in Chapter 4 is constrained by the use of a single, central manager that imposes both a bottleneck in task processing and a single point of failure. Two methods for allowing the use of multiple manager instances are clustering and federation. There are two primary difference between these methods. The first is the relationship between the manager instances and any backend storage mechanisms, and the second is public identity presented by any single manager agent.

### 5.1 Clustering

A clustering mechanism allows multiple manager instances to share the same identity, or JID. In addition, each clustered instance will share access to the same backend data storage service, allowing state to be synchronized between all instances. The result is effectively a single manager instance that has been spread across multiple processors. Users and worker agents may interact with this singular, logical manager with each message handled by a different physical machine. The loss of a single manager instance within the cluster should not impact the functionality of the entire cluster, with the exception that any processing done by that instance at the time of disconnection may be lost and require re-submission. As such, a clustered system may achieve high availability by ensuring the continued presence of at least one manager instance in the cluster at any given moment. An analogous technique would be load balancing for HTTP servers for the same domain.

### 5.1.1 Challenges with Redis

For Kestrel, Redis [63] was chosen for use as a shared data store between manager instances for the atomic operations it supports on complex data types such as dictionaries, sets, and lists. Converting the Kestrel manager to use Redis was a challenging process:

- Underlying library support was not written with clustering or use with Redis in mind.
- Remapping the current SQL database schema to the data structures provided by Redis.
- New algorithms for matching workers and jobs based on job requirements and worker capabilities were needed to match the new data storage patterns. The previous algorithm relied on SQL's `LIKE` operation which is not available with Redis.
- The data which had to be stored in Redis was not data that could be stored easily, such as callback functions or other closures.
- The load balancing offered by the XMPP server routed incoming messages to a random manager instance. Thus the instance that sends an `<iq />` stanza was not necessarily the same instance that would receive the reply.

### 5.1.2 Library Support

The first step in implementing clustering support was enabling the underlying SleekXMPP library used by Kestrel to work in such an arrangement. The SleekXMPP plugins that provided support for rosters, service discovery, and ad-hoc commands were originally created with the assumption of running in a single process, and used simple in-memory data structures for session state and storage as a result. Thus adding support for external data storage, and the adaptors for working specifically with Redis was needed. These adaptors have been turned into a sister library for SleekXMPP, named SleekRedis [68].

Most of the work done was to build in hooks for using generic external data storage mechanisms, and make explicit interfaces for how new storage mechanisms could be added. In the case of ad-hoc commands, the session data was originally stored in a Python dictionary. Since custom classes may implement the `__getitem__`, `__setitem__`, and `__delitem__` methods to provide support for the dictionary syntax, adding the external storage support was easier.

### 5.1.3 Mapping SQL databases to Redis

The second challenge was representing all of the data from the older SQL database to Redis key-value pairs. Whereas in a SQL table one may have a field to mark an entry as AVAILABLE or BUSY, the pattern in Redis is to create sets for each option. Thus, for workers there are the sets `workers:available`, `workers:online`, and `workers:busy`, each of which contains the JIDs of the relevant worker agents. However, this caused the proliferation of keys and sets to segregate workers and jobs in their various states of execution. To ensure that all related keys were updated appropriately for changes in a job or worker's state, the use of Redis' pipelines were required. A pipeline is similar to a transaction in a typical SQL database with ACID support. While each individual Redis command is atomic, a pipeline enqueue multiple commands and ensures that the entire sequence is executed atomically.

Listing 5.1: The code for determining the set of available workers to available tasks for a given job.

```
def job_matches(self, job):
    requirements = self.redis.smembers('job:%s:requirements' % job)
    p = self.redis.pipeline()
    matches = {}
    workers = self.redis.smembers('job:%s:workers' % job)
    for worker in workers:
        if self.redis.sismember('workers:available', worker):
            task = self.redis.srandmember('job:%s:tasks:queued' % job)
            if task is not None:
                p.smove('job:%s:tasks:queued' % job,
                       'job:%s:tasks:pending' % job,
                       task)
                p.set('job:%s:task:%s:is_pending' % (job, task), 'True')
                p.expire('job:%s:task:%s:is_pending' % (job, task), 15)
                p.sadd('worker:%s:tasks' % worker,
                      '%s,%s' % (job, task))
                p.set('job:%s:task:%s' % (job, task), worker)
                log.debug('MATCH: Matched_worker_%s_to_' % worker + \
                          'task_%s,%s' % (job, task))
                matches[task] = worker
    p.execute()
    return matches
```

### 5.1.4 Matching Jobs with Workers

The process for matching a worker with a job, and vice versa, was performed using a LIKE query in the SQLite database, but there is no equivalent command in Redis. By rephrasing the

problem, a Redis compatible method was found. Instead of performing a search whenever a match is needed, all possible matches are computed whenever a worker is added to the pool, or a job is submitted. The sets of matching job IDs are stored for each worker, and likewise for each job. When a match is requested, a random member of the applicable set is selected and checked for availability.

Listing 5.1 shows the process of matching workers to tasks for a given job. First, the set of known workers suitable for the job is retrieved. For each available worker, a task is randomly chosen and assigned to the worker, marking the task as pending. Redis provides support for expiring key values after a given delay; the feature is used here to indicate the allowable time to wait for a task assignment to be acknowledged by the worker. A periodic cleanup function exams all tasks marked as pending and if the associated `is_pending` key is not found, then the assignment has timed out and must be reset.

### 5.1.5 Serialization

The largest hurdle was dealing with data that could not be serialized. In particular, Python's `pickle` facility for serializing code objects will not serialize object methods, and it can not unserialize functions that were created on-the-fly. The trouble was that those types of entities were used heavily for ad-hoc commands, which stored function pointers to callback functions. The first attempt at resolving this was to remove the use of on-the-fly generated functions and use only defined object methods. After that change, an attempt was made to add support for serializing methods. Existing code to perform this operation was found from the Twisted Python project [14]. However, the underlying SleekXMPP library still required the use of a generated function, which prevented unserializing the stored methods.

An initial goal of adding Redis support to the ad-hoc commands plugin was to make the support transparent to an end user. However, as a compromise the current approach is to initialize the plugin with all of the potential callback functions. The plugin then hashes the names of the functions and stores the mapping in memory. It is the hash of the function's name that is stored if a function is detected in the session data to be stored in Redis. The current limitation to this approach is that using different methods with the same name will not work as expected. Situations like this would occur if one callback was the method `do_foo` of object `a`, and a second callback of method `do_foo` of object `b` were registered.

Other serialization issues included restoring stanza and JID objects. Upon unserializing the

objects, errors would occur because the necessary classes were not loaded. To resolve this, instead of serializing the objects directly, the underlying data was extracted and saved. Each key in the session data dictionary that required this treatment was saved in a list in a special `--XML--`, `--JID--`, or `--FUNC--` key.

### 5.1.6 XMPP Server Load Balancing

XMPP server implementations, in particular ejabberd, can provide support for load balancing external server components, such as the Kestrel manager. The method for balancing was to route an XMPP stanza to a manager instance based on the stanza's sender. Thus a single instance would always receive requests from a given client agent. The problem arose when a different instance sent a message to a client; it was the other instance that would receive the reply. Thus every manager instance had to be prepared to receive responses to queries that it did not send. The main culprit for these situations was the use of ad-hoc commands. With the changes described above for serializing code objects, it was possible for each manager to retrieve the original query data whenever a response was received.

## 5.2 Federation

Federation is a mechanism for allowing separate manager instances to share information while retaining separate identities. An implication of a clustered model is that the clustered instances all belong to the same organization, in contrast to the federated model which allows manager instances from various organizations to interact. Also, unlike clustering, federation does not necessitate the persistence of data in the event that an instance goes offline. Examples of federated systems would be SMTP and XMPP servers.

Implementing federation for Kestrel was more straightforward than clustering since syncing state using Redis was not needed. The method of federation chosen for Kestrel was to allow a manager to share tasks with another manager by masquerading as a worker. The lifecycle for federation between manager  $M_1$  and  $M_2$  is as follows:

1.  $M_1$  is instructed to federate with  $M_2$ .
2. For each unique set of features supported by the workers for  $M_1$ ,  $M_1$  registers as a virtual

worker to  $M_2$ , with the addition of the feature `kestrel:federation`.

3.  $M_2$  begins dispatching tasks to the virtual “workers.” The `kestrel:federation` feature may be used by  $M_2$  to only dispatch to virtual workers when no local workers are available.
4.  $M_1$  forwards received tasks to its own available workers (or potentially other managers).
5. If all workers providing a given set of features are all busy,  $M_1$  may send  $M_2$  a busy presence notice to prevent receiving new tasks requiring those features.
6.  $M_1$  sends a task complete notice to  $M_2$  when the task is completed, following the standard worker protocol.

Note that if the manager  $M_2$  goes offline it is not required that  $M_1$  take responsibility for any jobs tracked by  $M_2$ . Using both clustering and federation can ensure continued task tracking after the loss of a given instance. Also note that step 4, where  $M_1$  forwards a task to a worker, can create an infinite loop if both  $M_1$  and  $M_2$  are mutually registered as workers. Such an arrangement may be partially avoided by having each manager refuse to forward a task back to the manager from which the task was received. Situations where a larger loop is created, as shown in figure 5.2, may require other special handling. One possibility would be to simply mark a task as not to be forwarded after a certain number of hops.

The result of federation can effectively turn manager instances into routers, or congregators for workers. Such an architecture as shown in figure 5.2 could use a manager for each unique set of worker features such that each manager is responsible for a homogeneous set of workers, turning it into a simple queue of tasks without any need for matching. A higher, federated manager instance may then perform matching and track state of a smaller number of workers, albeit virtual ones. The arrangement in 5.2 allows all managers to have access to every worker in the global pool by establishing a loop.

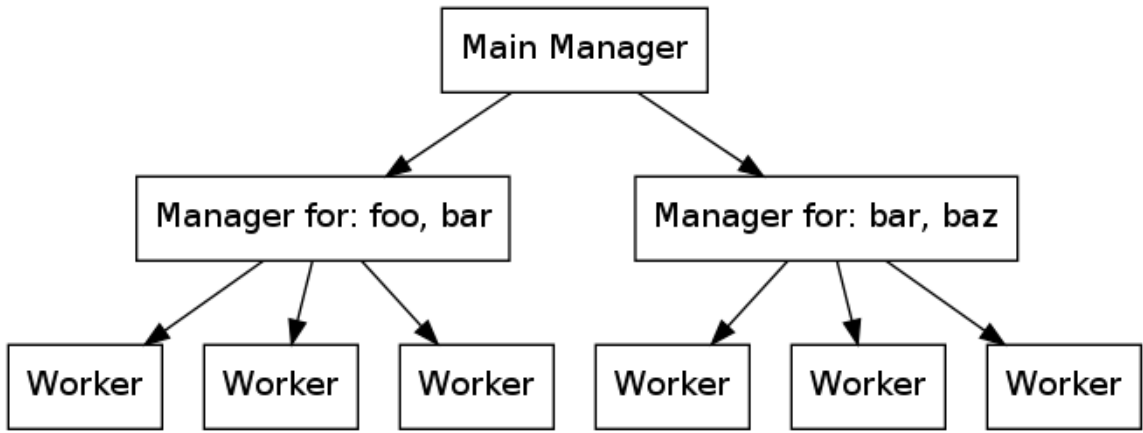


Figure 5.1: Layering federated manager instances into a tree architecture.

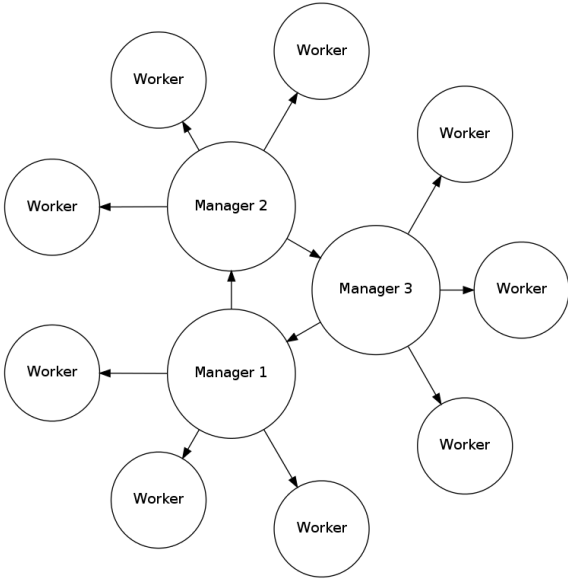


Figure 5.2: Creating cycles in the federation links between managers means that all managers in the cycle have access to all of the workers tracked by the managers in the loop. It becomes possible for a task to then be forwarded endlessly around the loop; therefore, a task should be marked with the number of hops it has made and once it has reached a configured limit, not be forwarded any further.

## Chapter 6

# Results

When scaling a Virtual Organization Cluster (VOC) [51, 52, 50] from a single site to multiple sites across a wide area network (WAN), several challenges arise, including Network Address Translation (NAT) traversal and increased latency. As Kestrel has been developed to provide a scheduler and dispatcher on top of an XMPP-based overlay for creating the VOC, its fitness for its stated objectives must be demonstrated. Thus, to show Kestrel's ability to act as a scheduler and overlay for both building and scaling a VOC across multiple sites, four criteria must be met. The first of these is that the system must be able to respond to the changes in the number of available workers as nodes are provisioned or decommissioned through the underlying grid mechanism. The second is that Kestrel must be able to acknowledge and manage a large number of worker nodes across NAT boundaries, and the third is that the latencies involved in cross-site communication should not excessively impede job dispatch rates. In high throughput computing, the focus is on long running tasks which overshadow the time spent during dispatch [71, 29]. However, for many-task computing where many short lived tasks are typically scheduled [56], the dispatch time can introduce significant overhead. For Kestrel to scale across sites, it should not be affected by latency during job dispatch, given tasks that take on the order of thirty seconds or a few minutes in duration to execute. Thus, the fourth criterion is establishing the expected dispatch rates for tasks in order to determine a lower bound for the desired task execution time.

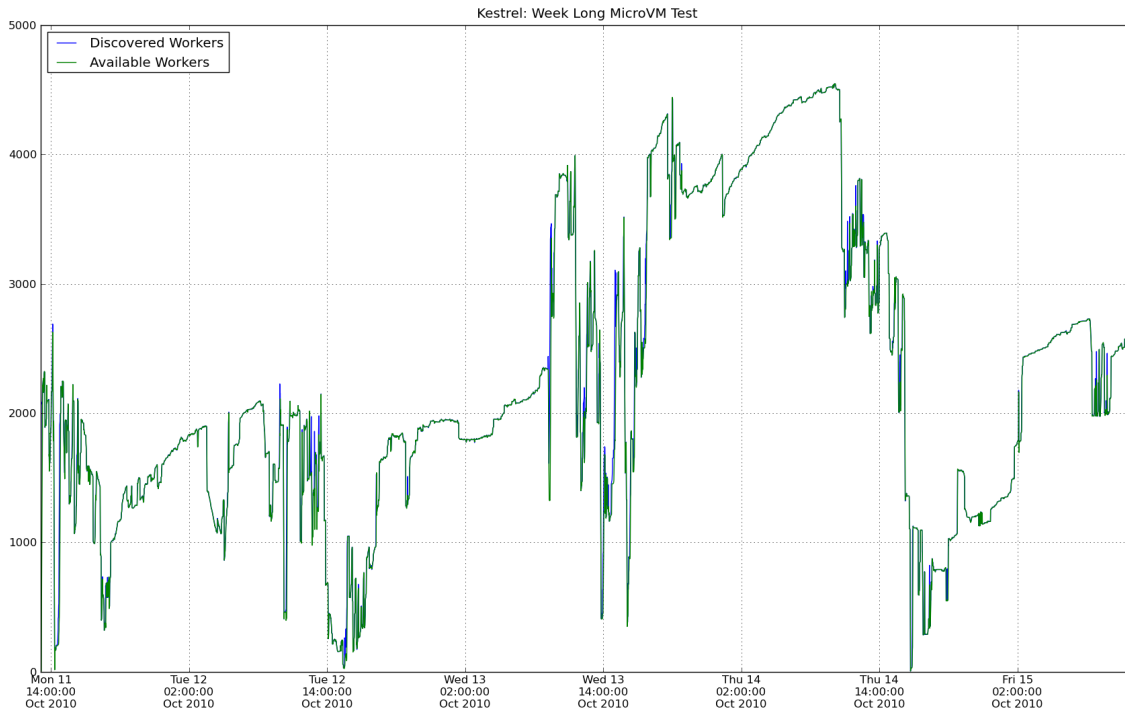


Figure 6.1: The results of a week long test monitoring the number of available worker agents. A Condor job was created to queue several thousand instances of memory-constrained worker VMs for execution, with a second script resubmitting the job as needed to refill the request. Kestrel was able to recognize roughly 4,500 simultaneous worker instances.

## 6.1 Responding to Changes in VOC Size

In order to test Kestrel’s capability for responding to growing and shrinking worker pools based on the changing number of provisioned VMs from an underlying grid mechanism, a week long test monitoring this behavior was performed. The test consisted of first creating a worker VM image with minimal memory usage so that multiple VMs could be easily started on a single physical node. Using an image based on Micro Core Linux [66], a minimum requirement of 32Mb of memory for the VM was achieved. The next phase of the test was submitting Condor jobs to provision VMs based on the minimal image. The intent was to flood idle Condor slots with Kestrel workers and track the changes in the size of the pool as seen by the Kestrel manager.

The results of the test as shown in figure 6.1 indicate that a maximum of roughly 4,500 workers was obtained mid-week during the night. During the day, the number of available Condor slots reduced as expected as physical machines were used by other users, but spikes in idle slots

during daytime usage did occur which the Kestrel manager was able to detect.

## 6.2 Latency

To determine the latency introduced by XMPP, a multi-site pool was started using Kestrel. The sites included Clemson University’s Palmetto cluster, Amazon EC2, and CERN. The procedure for the experiment was to have a single node send a “ping” message to all the worker agents in the pool; to make processing easier, each “ping” message includes the time at which it was sent. These workers in turn replied to the sender with a “pong” message that included the sent time in the “ping” message. The current time was compared to the time stored in the “pong” message, and the round trip time (RTT) was calculated. Prior to running the experiment, the hypothesis was that it should be clear from a RTT graph that multiple sites were used in the pool since the data points should form clusters. A concern, however, was that the differences between Clemson’s Palmetto cluster and Amazon’s EC2 would not be distinct enough and would overlap.

The results of the experiment (illustrated in Figure 6.2) showed that a worker pool formed using XMPP can operate across multiple sites. The virtual machines at CERN responded in approximately 0.3 seconds whereas the the virtual machines on Palmetto and EC2 responded in roughly 0.1 seconds. The response times for Palmetto and EC2 overlapped considerably. The RTT times as measured through Kestrel were considerably longer than equivalent measurements through the system ping command, owing to the significant CPU overhead required to parse XML messages received via XMPP.

## 6.3 Pool Sizes

To establish the need for a special-purpose WAN scheduler, it was necessary to test the scalability of disk subsystems and general-purpose overlay networks by starting large number of VMs, as described in Section 6.3.0.1. To compare the behavior of a general-purpose overlay network to a purpose-built WAN scheduler, an overlay combination of IPOP and Condor (section 6.3.0.2) was tested against Kestrel (section 6.3.0.3) in an environment with pervasive NAT boundaries. If the number of VMs (and thus IPOP nodes) that could be instantiated on a TOP100 cluster did not exceed the limits of the general-purpose overlay network, then a special-purpose scheduler is

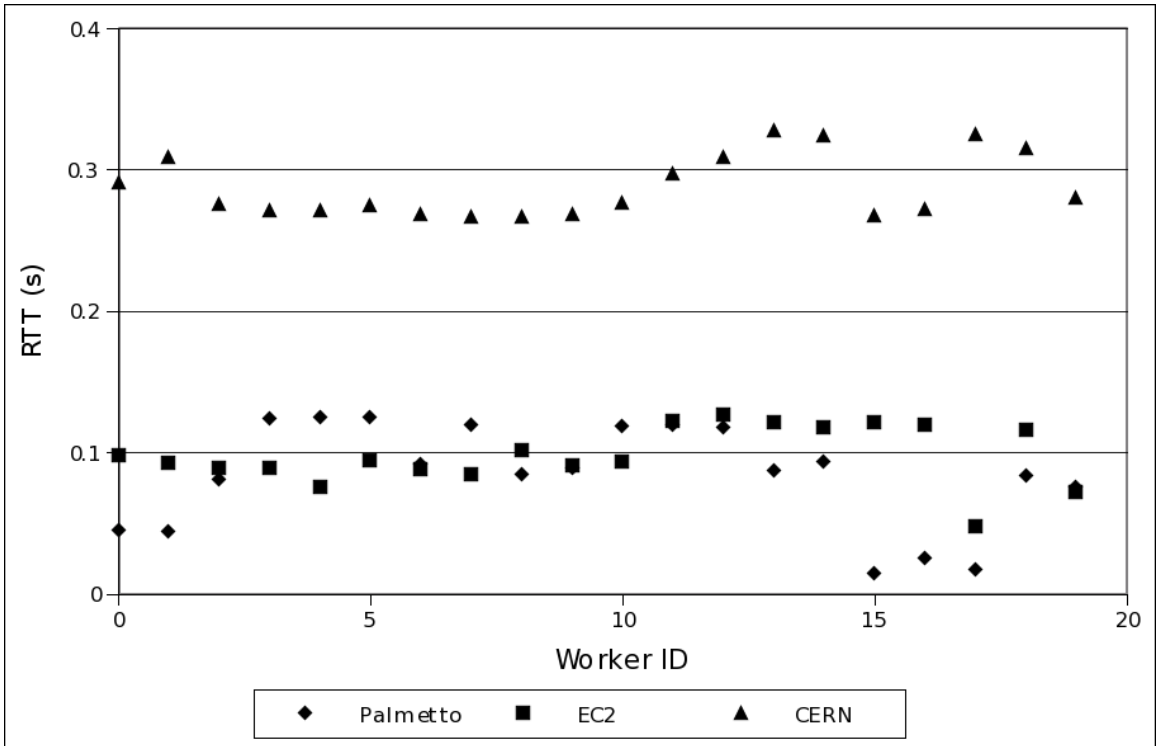


Figure 6.2: The differences in response times between the three grid sites. Only the site at CERN produced significant latency which could distinguish worker agents from that site.

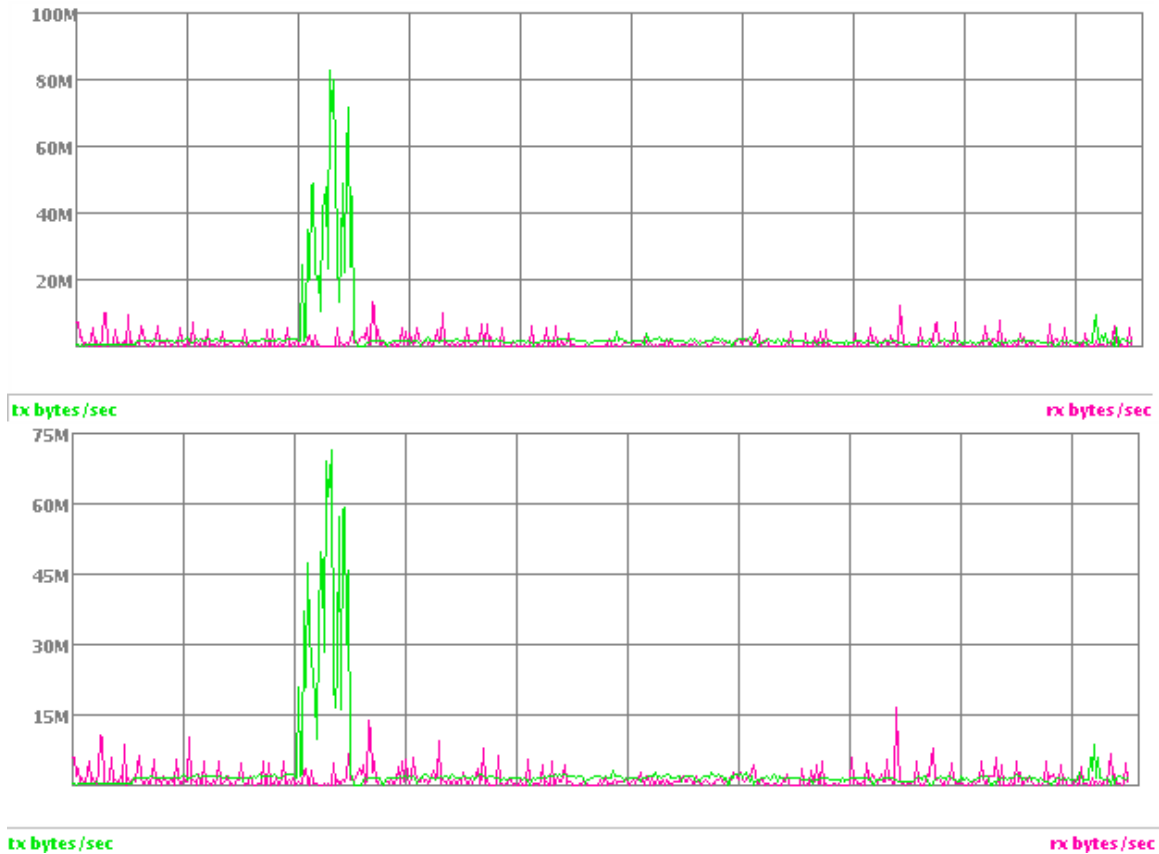


Figure 6.3: Aggregate disk throughput for the two 4Gb Fibre Channel ports on Clemson’s Palmetto cluster’s disk array when starting 1000 VMs. The observed throughput of 154Mb/sec was well below the maximum throughput of the two channels.

unnecessary.

### 6.3.0.1 VM Image Considerations

The simultaneous booting of thousands of VMs has the potential to place considerable strain on the test cluster’s disk subsystem. This risk was mitigated by the use of the *snapshot* mode in QEMU/KVM. In snapshot mode, QEMU/-KVM did not write to the disk image from which the VM was instantiated. Instead, each instance of QEMU/KVM created a local copy-on-write file which overlaid the image of the original disk. Thus, many VMs could be instantiated from a single disk image. Furthermore, QEMU/KVM would only read blocks from this image as they were requested by the guest OS, greatly reducing the total amount of data that had to be transferred. The combination of these two factors allows the disk to be placed on a shared network store.

Results of a test that measured the aggregate disk throughput needed to boot 1000 VMs have been presented in Figure 6.3. A trace of the two 4Gb Fibre Channel ports connected to the Clemson University Palmetto Cluster’s disk array was obtained from the storage management system. Packets were automatically load-balanced between the two ports, reaching a peak total transfer rate of approximately 154MB/s. This rate was well below the maximum capacity of the disk array. Thus it can be concluded that starting 1000 VMs with QEMU/KVM’s snapshot mode did not place an onerous load on a cluster’s disk subsystem.

### 6.3.0.2 IPOP/Condor Experiment

The scalability test for IPOP/Condor was conducted by starting a set of VMs using QEMU/KVM. Each VM utilized QEMU’s user-mode networking stack for connectivity. This stack implemented a subset of the TCP/IP protocol and was used to provide network access to the VM without the need for administrative privileges on the host. Since the VM’s network interface was not directly bridged to the the physical interface, the user-mode networking stack was required to implement NAT. With this setup, each VM was in a separate private network, maximizing the number of NAT traversals IPOP was required to provide. Furthermore, the Clemson Palmetto Cluster was located behind a NAT edge router to the Internet. Since the IPOP bootstraps were used for testing purposes, initial connections must be made through two layers of NAT, further stressing the overlay network.

Two separate test scenarios differed in temporal characteristics. In the first test, a large number of IPOP nodes entered the overlay simultaneously. In the second test, nodes entered the overlay in a more incremental fashion. The total number of nodes was the same in both cases.

In order to measure the number of nodes which entered the overlay, an observation node was employed. This node monitored the status of the Condor pool. A worker node was considered to have entered the overlay whenever it appeared in the Condor pool.

As illustrated in Figure 6.4, the catastrophic failure of the overlay when 1600 nodes attempted to enter within a short interval. Figure 6.5 shows the results of 200 batches of 8 nodes each entering the overlay. The overlay remained active, but approximately 300 of the requested nodes never entered the overlay. Additionally the maximum number of nodes in the overlay at any one time was approximately 500.

These tests represented a worst-case scenario for IPOP because of the rapid arrival of many

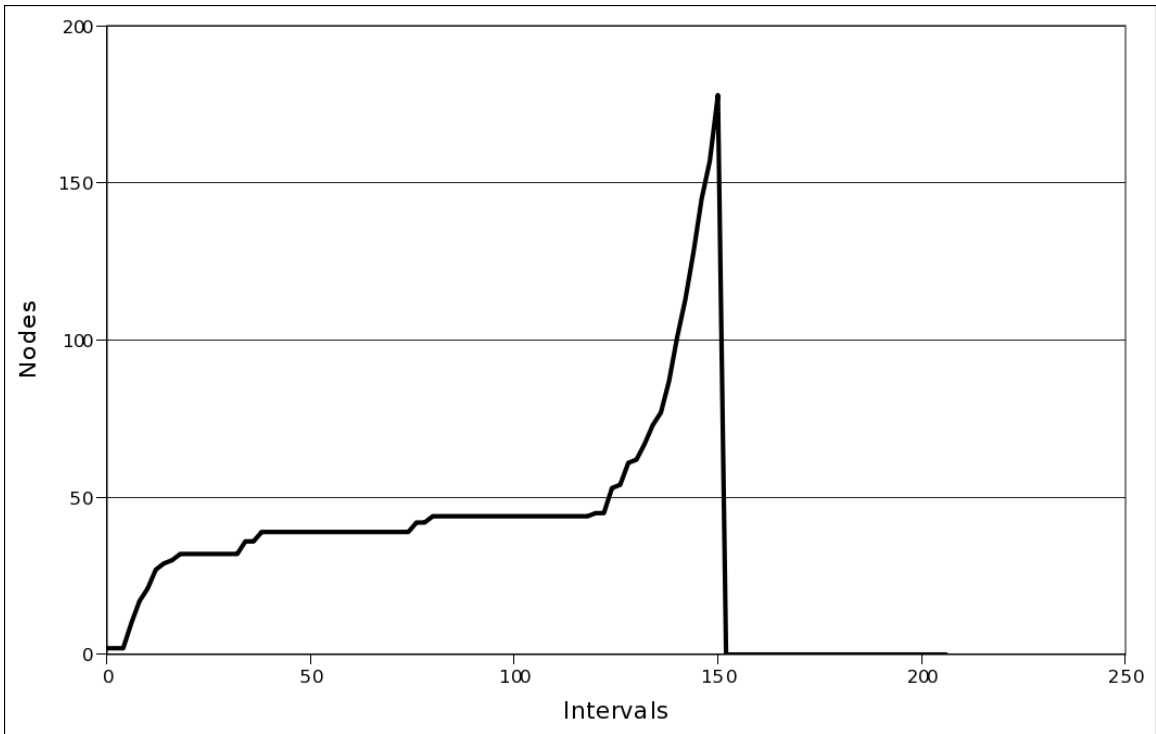


Figure 6.4: The results of starting an IPOP/Condor pool of 1600 nodes in one batch

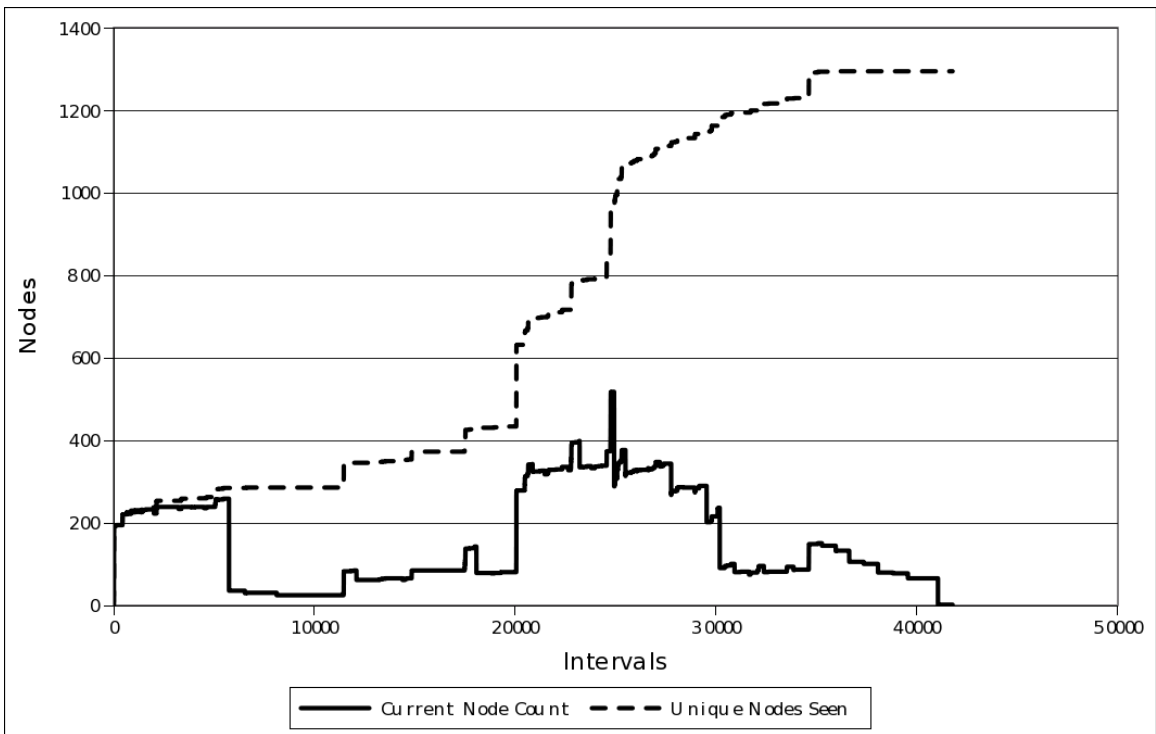


Figure 6.5: The results of starting an IPOP/Condor pool of 1600 nodes in 200 batches

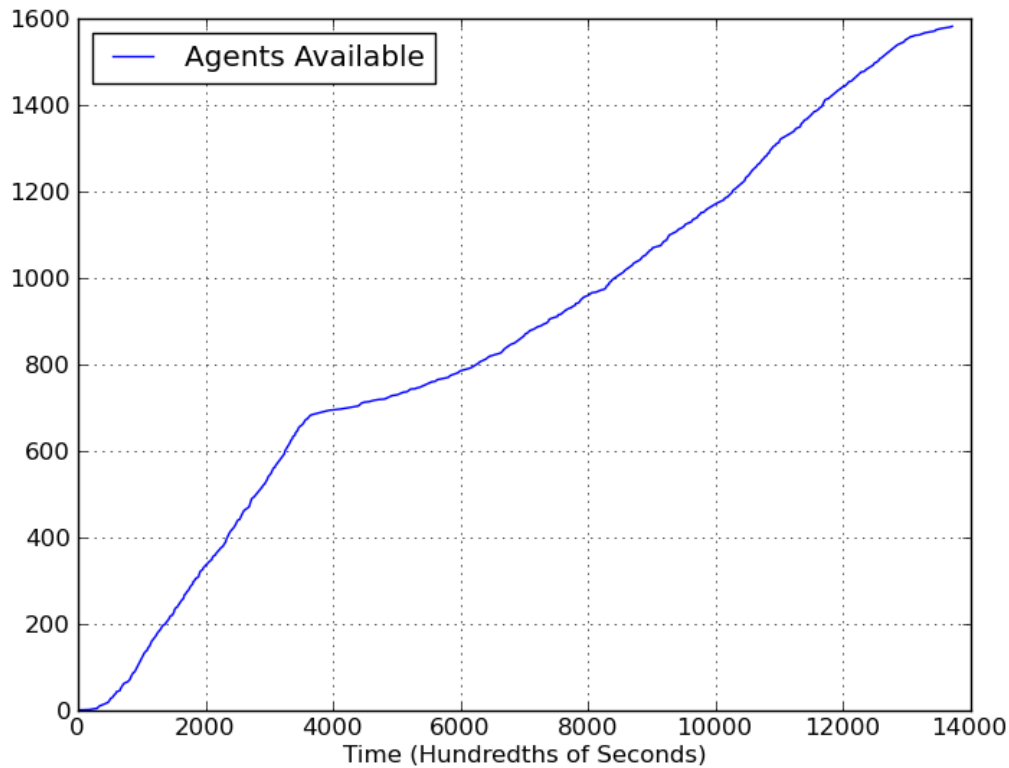


Figure 6.6: Kestrel Pool of 1600 Agents

nodes behind two layers of NAT. In this situation, the overlay became unbalanced, collapsing entirely.

### 6.3.0.3 Kestrel Experiment

As a special-purpose scheduler based on XMPP, Kestrel did not have to implement a full, general-purpose networking stack like IPOP. Instead, Kestrel performed the specific function of task scheduling and distribution. However, applications distributed using Kestrel would not have access to a full network stack except for the one provided by the VM. To test how well Kestrel performed, 1600 VMs were instantiated on the Palmetto cluster using 200 physical nodes with 8 VMs per node. As a measure to reduce strain on the XMPP server, each worker waited for a random interval of up to 30 seconds before joining the pool. Since XMPP uses a pull-based model, the double NAT layer does not pose any issues.

Of the 1600 VMs submitted, 1598 Kestrel worker agents started properly and connected to

## Client Sessions

Active Client Sessions: **1598** -- Showing 1-100  
Pages: [[1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#) [10](#) [11](#) [12](#) [13](#) [14](#) [15](#) [16](#)] -- Sessions per page:  Refresh:  (seconds)

Name	Resource	Status	Presence	Priority	Client IP	Close Connection
1 <a href="#">worker_998</a>	156a30bb	Authenticated	Online	0	10.125.4.115	
2 <a href="#">worker_993</a>	c0095579	Authenticated	Online	0	10.125.4.143	
3 <a href="#">worker_991</a>	d2a850c8	Authenticated	Online	0	10.125.3.198	
4 <a href="#">worker_990</a>	2ea819bc	Authenticated	Online	0	10.125.4.61	
5 <a href="#">worker_986</a>	1dbb32b4	Authenticated	Online	0	10.125.3.245	
6 <a href="#">worker_976</a>	71d4db74	Authenticated	Online	0	10.125.4.48	
7 <a href="#">worker_962</a>	422e0bce	Authenticated	Online	0	10.125.4.145	
8 <a href="#">worker_961</a>	a9786454	Authenticated	Online	0	10.125.3.239	
9 <a href="#">worker_96</a>	ea4f6c02	Authenticated	Online	0	10.125.4.226	
10 <a href="#">worker_96</a>	4cf84d00	Authenticated	Online	0	10.125.4.111	

Figure 6.7: Openfire Showing Available Agents

the XMPP server. From the server’s web interface all 1598 workers could be seen online at once.

A difficulty was initially encountered that prevented the Kestrel pool from growing beyond approximately 900 agents. The solution was to increase the number of open files allowed per process in `/etc/security/limits.conf` where the default value is 1024 files per process. In order to test the scalability of Kestrel, by allowing 16384 files per process and starting ten worker agents per VM, further experiments have shown that the XMPP server can accept at least 15,000 connections at once. However, Kestrel job dispatching has not yet been tested on a pool of that size.

## 6.4 Kestrel Task Throughput and Dispatch Rates

In order to test the dispatch rate for Kestrel, four experiments were performed. Each experiment scheduled 50,000 sleep commands to about 900 worker agents. The first experiment used “sleep 0” to test Kestrel’s performance under the most demanding workload of constantly scheduling. The second experiment used sleep with a half second delay, and the third used a two second delay. However, since a constant time for every task is not always accurate, a fourth trial used random length delays where each worker would sleep between 0 and 10 seconds.

For each run, the number of agents in three states (online, available, and busy) were tracked. The desired result would show the number of available workers mirror the number of online workers until the job was submitted. After the submission, the number of available workers would go to zero

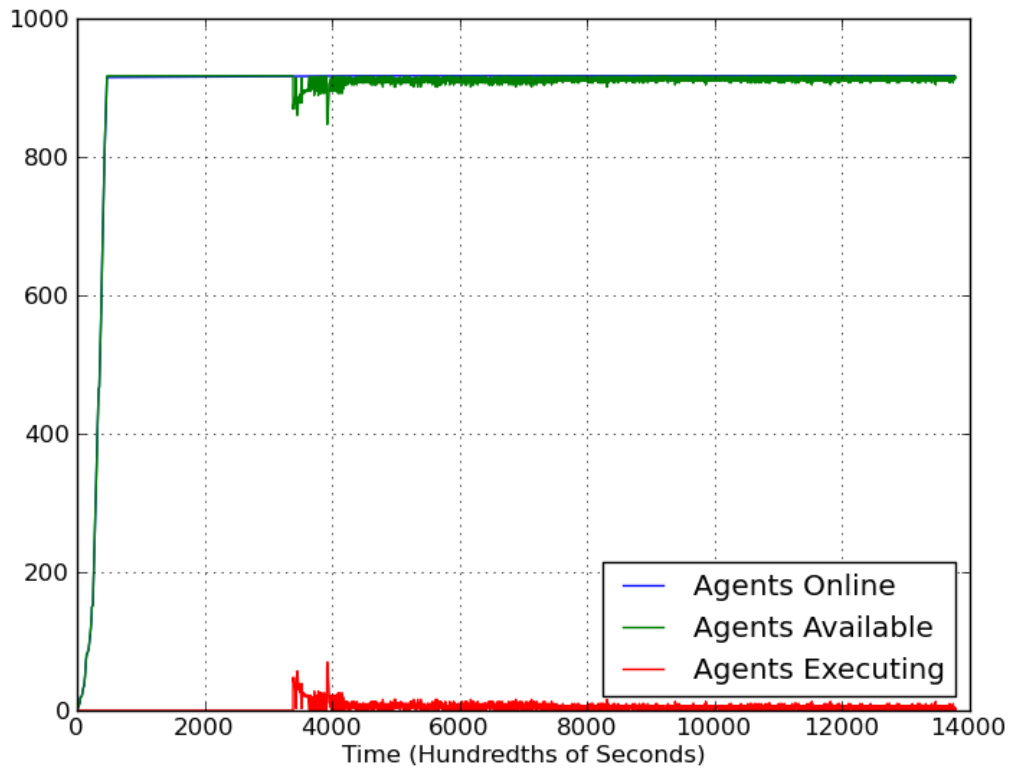


Figure 6.8: Trial #1 executing “sleep 0”

and the number of executing agents would rise to the number of online agents, representing a 100% usage of the pool. Such an arrangement would then last until all the tasks have been executed.

The results of the first experiment, shown in Figure 6.8, were disappointing in that only a small percentage of the pool was used at any given time. Since the scheduling strategy for Kestrel was to dispatch on demand when either a worker is available or a job was submitted, the scheduler could not keep up with the demand. Before the first round of tasks could be submitted to all available workers, the first workers given tasks finished and requested more tasks. The total time for completing the job was 357.38 seconds, giving a throughput rate of almost 140 tasks per second.

The second experiment showed a large improvement over the first in Figure 6.9. The half second delay of “sleep .5” helped reduce the strain on the scheduler, allowing for a larger pool usage. However, roughly only 700 of the 900 workers were able to receive their first task during the half second delay. The result was severe thrashing immediately after the job was submitted as the first

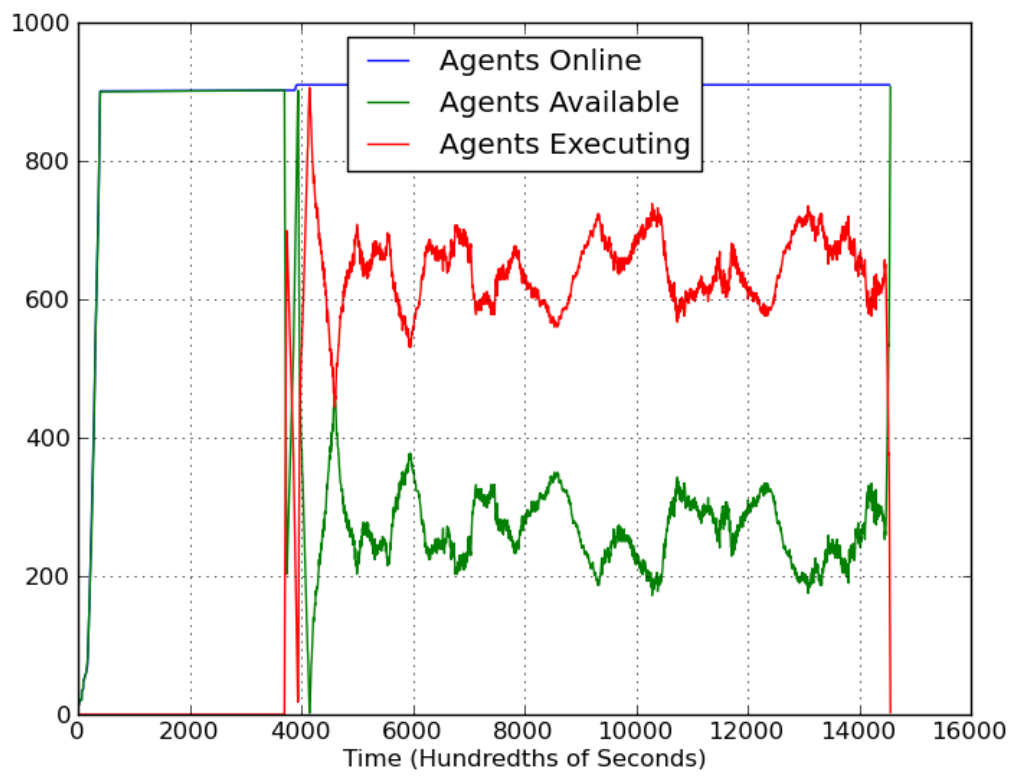


Figure 6.9: Trial #2, executing "sleep .5"

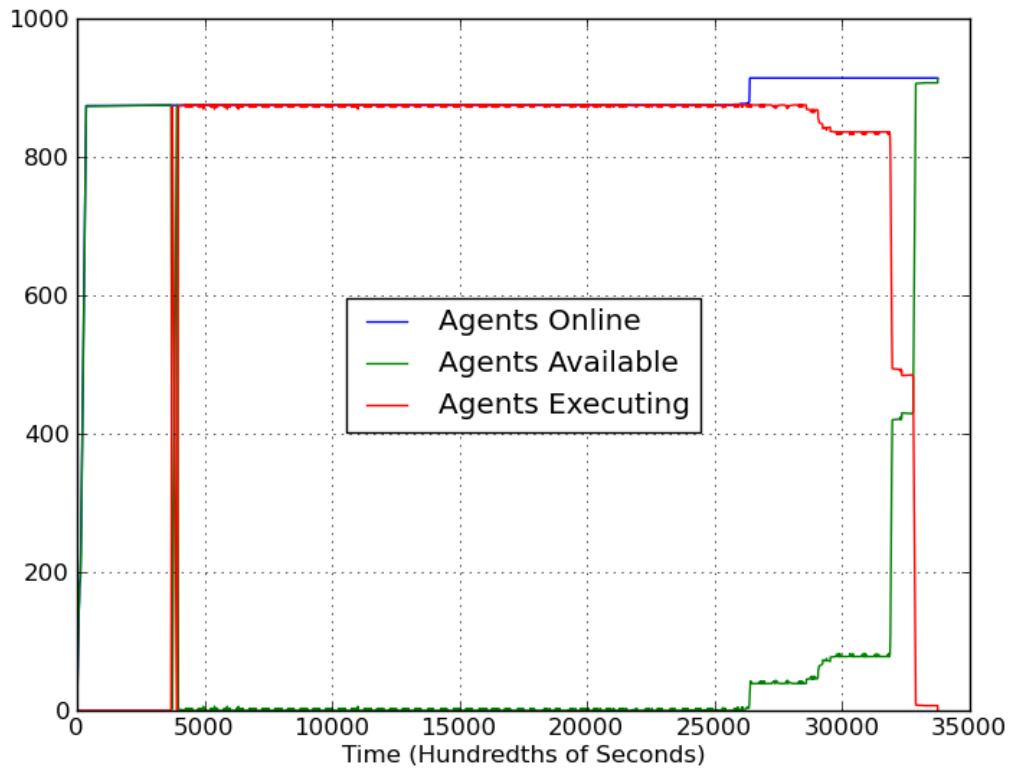


Figure 6.10: Trial #3, executing “sleep 2”

batch of tasks were completed before the last group was fully dispatched. The system stabilized at close to a 66% usage for the remainder of the job. The total time for the job was 108.61 seconds, giving a task throughput rate of 460 tasks per second. The reduction in scheduler load compared to the first experiment allowed for the increased throughput despite the increased time per task.

The third trial in Figure 6.10 followed the expected, ideal pattern. The two second delay was sufficient to dispatch the first round of tasks to the 875 available workers. After the initial round, the scheduler was able to meet the demand for tasks, creating small, periodic blips of available workers, but otherwise remaining at close to 100% usage of the pool. Further experiments were later effected using task times of 5 and 10 seconds, and the results were similar. The time for completing the job was 300.56 seconds, giving a throughput rate of 166 tasks per second. The reduced throughput was due to the longer time per task. The time it took to distribute the 875 initial tasks was .51 seconds, giving a dispatch rate of 1715 tasks per second.



Figure 6.11: Trial #4, executing “sleep random(0,10)”

The fourth trial (Figure 6.11) used a random task length between 0 and 10 seconds. The pool usage remained near 100% for most of the life of the job, but gradually declined at the end as the remaining, long tasks complete and were not replaced. The scheduler was able to distribute 61 tasks in 0.04 seconds before a worker returned for another task, yielding a dispatch rate close to 1500 tasks per second.

## 6.5 Clustered Manager Performance

In order to test clustering performance, the time to completion for jobs was measured with increasing numbers of manager instances in the cluster. The expectation was that completion time would decrease as processing time would decrease for any given instance. Figure 6.5 shows the results of running 1000 microbenchmark tasks of the `sleep 0` command on 256 worker instances. Each manager was run in a MicroCore linux virtual machine with 512Mb of memory.

As hypothesized, the time for job completion decreased as the number of clustered managers increased. The drop in time spent was due to the ability to both process and send XMPP stanzas in parallel. The rate of diminishing returns was affected by the quality of the load balancing between clustered instances provided by the XMPP server. While each run started with all manager instances processing and sending stanzas, over time one of two instances became favored by the balancer, turning the arrangement closer to a double manager installation. Such favoritism did not impact the ability to achieve high availability, however. After terminating a favored manager, the system was able to continue running and tracking tasks. Tweaking the load balancing configuration for the XMPP server may solve this issue.

## 6.6 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) [13]. 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

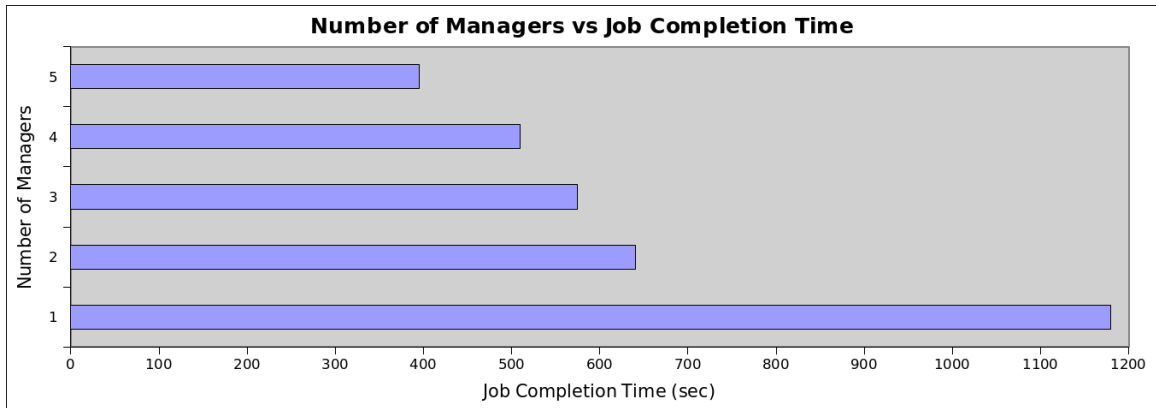


Figure 6.12: The relative time for job completion using an increasing number of clustered manager instances.

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.

### 6.6.1 Description

As a recent project, Kestrel had only been tested using simulated workloads and micro-benchmarks to measure manageable pool sizes, dispatch rates, and startup times [69]. Data and feedback from testing using a real-world scientific application was lacking until the STAR group agreed to participate in a trial program using Kestrel.

STAR’s goal was to use Kestrel to create a VOC for running proton collision simulations. The STAR software stack consists of over 2.5 million lines of code, and deployment requires a vast number of external libraries and multiple compilers, such as PYTHIA and GEANT3. This reason alone made virtualization attractive for STAR. The image used at Clemson contained the deployment of a single STAR library, using Scientific Linux 5.3 as the operating system. A copy of the STAR offline database was also installed. The VOC size was expected to stay close to one thousand active workers over the course of a month using machines provided by both CERN and Clemson’s Palmetto cluster. Distributing and starting the VMs was conducted with the Portable Batch System, PBS [53]. The principal steps taken to setup the experiments were:

- Create 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 was added to the VMs `/etc/rc.local` file to automatically

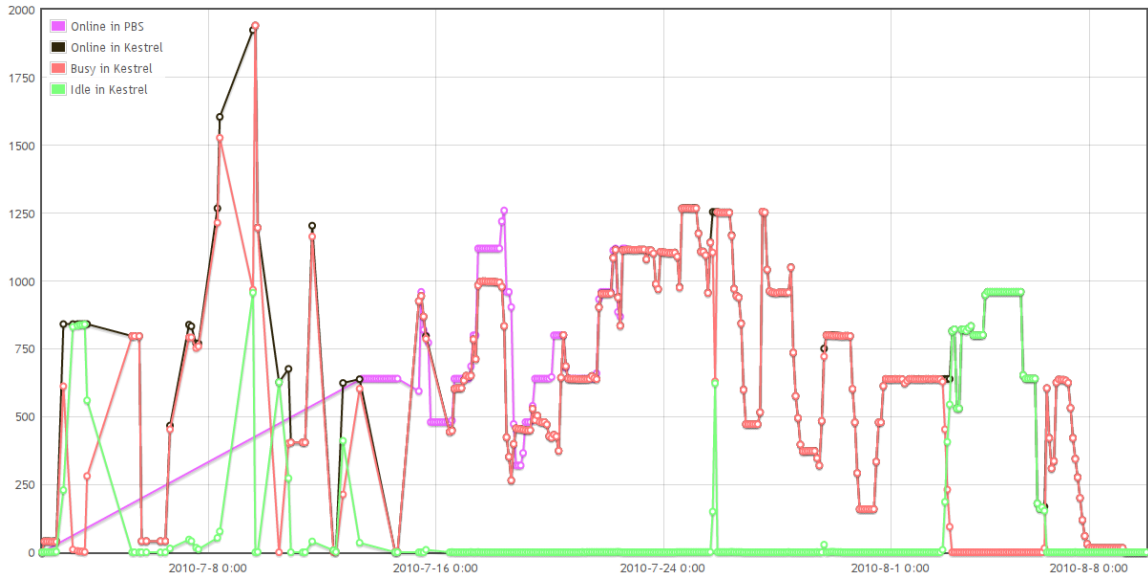


Figure 6.13: The number of online, available, and busy workers according to the Kestrel manager during the course of the experiments.

start after the VM booted.

- Stage the virtual machine on the Clemson Palmetto cluster on a shared filesystem. Both NFS and PVFS were tested and performance was presented in [69].
- Start virtual machine instances using PBS and KVM in snapshot mode, with eight VMs per physical node. The snapshot mode permitted to avoid having to transfer the base image to all nodes and only create a local copy-on-write disk while keeping the base image on the shared file system.
- Monitor the number of workers in Kestrel using both the Kestrel client, and the web interface for the XMPP server, ejabberd [5].
- Submit and manage jobs from a Kestrel client.
- 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. 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

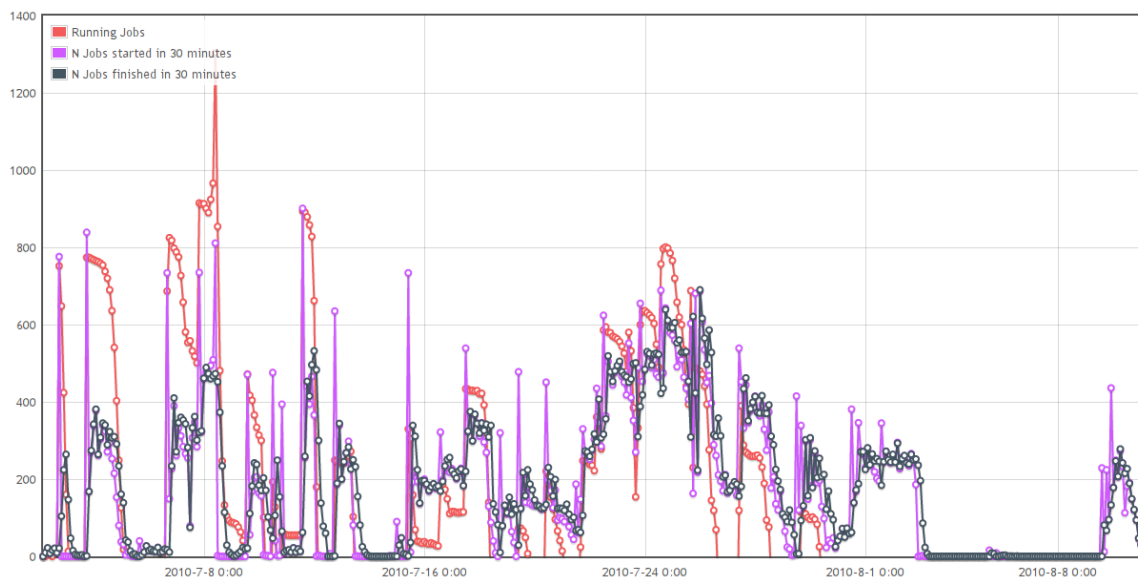


Figure 6.14: The number of simulation tasks executing over the course of the experiment, as reported by STAR’s worker applications. As can be seen in the first half of the graph, only periodic bursts of tasks could be executed before several concurrency and networking errors would deadlock the manager, preventing further execution until the system could be restarted.

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 4.4.1. 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.

## 6.6.2 Results

The first half of the experimental period yielded marginal successes in terms of the sustained number of active workers as demonstrated in figure 6.13. The initial Kestrel manager version used for the trial would quickly lose its synchronization with the state of the worker pool due to concurrency issues in the in-memory data storage. These issues prevented the manager from accepting and maintaining large numbers of workers, even though they were online and connected to the XMPP server. Eventually, after several days of use the manager would deadlock, halting task dispatch and disabling status queries. After several long sessions of the STAR and Clemson teams debugging the system together, the lessons learned were used to make the adaptations and improvements described

earlier in section 4.5.2.

The adapted Kestrel version produced an immediate increase in sustained worker count and task throughput as shown in figures 6.14 and 6.15. However, the original goal of one thousand active workers could still not be maintained due to irregular access to the available physical resources in the Palmetto cluster, and was not caused by errors in Kestrel. 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. Latest research and experiments has shown that Kestrel can dispatch tasks to over 5,000 workers. 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 was taking 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.

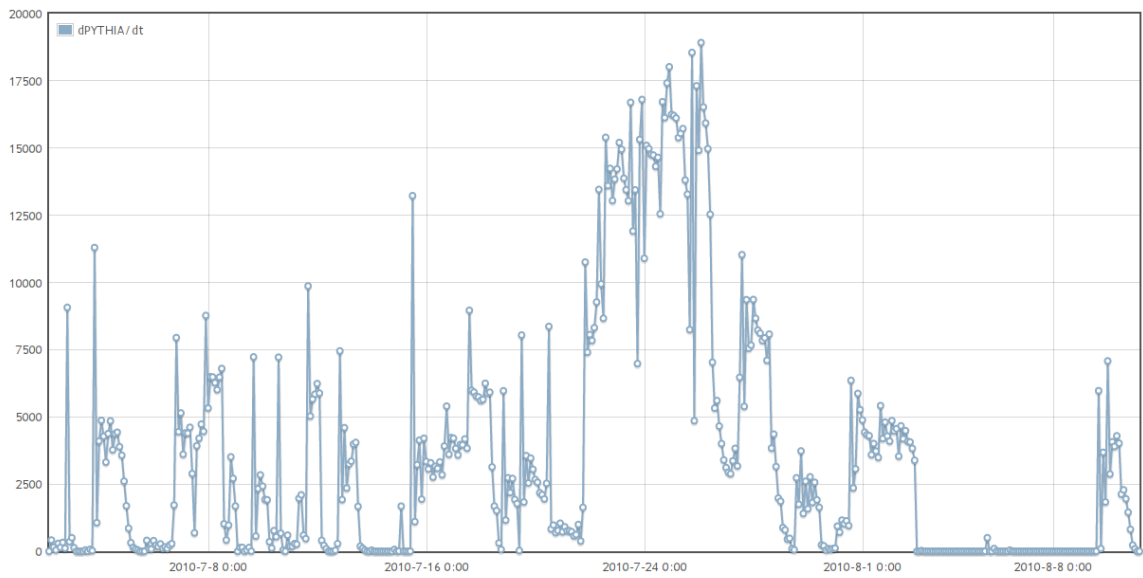


Figure 6.15: Number of PYTHIA events generated during simulations. The spike in the event rate corresponds with the new Kestrel version's introduction that allowed for more tasks to execute and for longer periods of time.

## Chapter 7

# Conclusions and Discussion

Kestrel is able to manage a VOC effectively and distribute tasks with a dispatch rate of more than 1000 tasks per second. Due to the current scheduling algorithm, tasks that are too short will result in reduced performance from increased load on the manager. The minimum desired time was at most 2 seconds for a pool of 900 workers. Given the latency results found in Section 6.2, using tasks that are at least 30 seconds in duration should be sufficient to mask any dispatch delays caused by latency between sites.

As demonstrated through testing, it is possible to construct a VOC with over 1,500 VMs on a single physical cluster using a single VM image without straining the network by using QEMU/KVM in snapshot mode. However, for the case where many different VM images are needed, or one VM image per worker is required, pre-staging VMs could still be more efficient.

Establishing a VOC over a WAN is possible using XMPP. While a 0.3 second latency would cause network I/O intensive programs, such as those based on MPI, to perform poorly, computationally bound applications such as those assumed by the VOC Model should not be affected after the initial dispatch. Due to the possible overlap in RTTs between multiple physical sites, as demonstrated by the results from Palmetto and Amazon EC2, it is not always possible to determine the number of physical sites in a VOC based on RTT data.

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 servers proved to be a work-

able solution, handling over seven terabytes of output data. While 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.

Enabling clustering support for Kestrel is possible to allow for high availability and remove the single point of failure of the central manager. Testing has shown that increasing the number of clustered managers decreases the total time to completion for jobs that have short task execution times, where the time spent sending and processing XMPP stanzas becomes a bottleneck. The federated model can be extended further to a full peer-to-peer (P2P) model; as a federated manager exposes the same interfaces as a normal worker, it becomes possible to unify workers and managers. To do so, a manager would need to be able to execute tasks; as such, it would need to track its own state just like any other worker. By trading task dispatch time due to multiple hops, the requirement of using XMPP server components for the manager relaxes as the need for knowledge of other agents in the pool is reduced and the use of a server backed roster for a client agent becomes feasible. Future work should explore this form of Kestrel and try to remove the distinction between workers and managers.

# Bibliography

- [1] Adium.  
<http://www.adium.im>.
- [2] Amazon Elastic Compute Cloud (Amazon EC2).  
<http://aws.amazon.com/ec2>.
- [3] AOL Instant Messenger.  
<http://www.aim.com>.
- [4] Condor Version 7.0.5 Manual.  
[http://www.cs.wisc.edu/condor/manual/v7.0/3\\_7Networking\\_includes.html#SECTION00473000000000000000](http://www.cs.wisc.edu/condor/manual/v7.0/3_7Networking_includes.html#SECTION00473000000000000000).
- [5] Ejabberd.  
<http://www.process-one.net/en/ejabberd/>.
- [6] Facebook Chat.  
<http://www.facebook.com/sitetour/chat.php>.
- [7] GCB: Generic Connection Brokering.  
<http://www.cs.wisc.edu/condor/gcb>.
- [8] Google Talk.  
<http://www.google.com/talk>.
- [9] History - The XMPP Standards Foundation.  
<http://xmpp.org/about-xmpp/history>.
- [10] Jabber.Ru — The biggest Russian-node instant messaging network.  
<http://www.jabber.ru>.
- [11] Pidgin, the universal chat client.  
<http://www.pidgin.im>.
- [12] SQLAlchemy - The Database Toolkit for Python.  
<http://www.sqlalchemy.org>.
- [13] The STAR experiment.  
<http://www.star.bnl.gov/>.
- [14] Twisted Python.  
<http://twistedmatrix.com>.
- [15] Windows Live Messenger.  
<http://explore.live.com/windows-live-messenger>.

- [16] XMPP Standards Foundation.  
<http://xmpp.org>.
- [17] Yahoo! Messenger - Chat, Instant message, SMS, Video Call, PC Calls.  
<http://messenger.yahoo.com>.
- [18] Public XMPP Services.  
<http://xmpp.org/services>, December 2009.
- [19] 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.
- [20] 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.
- [21] badlop. Ejabberd.  
<http://www.ejabberd.im/features>, 12 2006.
- [22] 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.
- [23] O. Beaumont, L. Carter, J. Ferrante, A. Legrand, L. Marchal, and Y. Robert. Centralized versus Distributed Schedulers for Bag-of-Tasks Applications. *IEEE Transactions on Parallel and Distributed Systems*, 19(5):698–709, May 2008.
- [24] M. Bégin. An EGEE comparative study: Grids and Clouds-evolution or revolution. In *23rd Open Grid Forum (OGF23)*, 2008.
- [25] D. Crockford. Introducing JSON.  
<http://json.org>.
- [26] 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.
- [27] 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.
- [28] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2.  
<http://www.ietf.org/rfc/rfc5246.txt>, August 2008.
- [29] 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.
- [30] 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.
- [31] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputing Applications*, 11(2):115–128, 1997.
- [32] 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.

- [33] N. Fritz and L. Stout. SleekXMPP.  
<http://github.com/fritzzy/SleekXMPP>.
- [34] 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.
- [35] 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.
- [36] D. R. Hipp. SQLite.  
<http://www.sqlite.org/>.
- [37] R. E. Joe Hildebrand, Peter Millard and P. Saint-Andre. XEP-0030: Service Discovery.  
<http://xmpp.org/extensions/xep-0030.html>.
- [38] R. T. Joe Hildebrand, Peter Saint-Andre and J. Konieczny. XEP-0115: Entity Capabilities.  
<http://xmpp.org/extensions/xep-0115.html>.
- [39] K. Keahey and T. Freeman. Contextualization: Providing One-Click Virtual Clusters. In *4th IEEE International Conference on e-Science*, Indianapolis, IN, December 2008.
- [40] P. Leach and C. Newman. Using Digest Authentication as a SASL Mechanism.  
<http://www.ietf.org/rfc/rfc3831.txt>, May 2000.
- [41] G. M. M. Day, S. Aggarwal and J. Vincent. Instant Messaging / Presence Protocol Requirements.  
<http://www.ietf.org/rfc/rfc2779.txt>, February 2000.
- [42] J. R. M. Day and H. Sugano. A Model for Presence and Instant Messaging.  
<http://www.ietf.org/rfc/rfc2778.txt>, February 2000.
- [43] T. Maeno. PanDA: distributed production and distributed analysis system for ATLAS. In *Journal of Physics: Conference Series*, volume 119, page 062036. IOP Publishing, 2008.
- [44] A. Melnikov and K. Zeilenga. Simple Authentication and Security Layer (SASL).  
<http://www.ietf.org/rfc/rfc4422.txt>, June 2006.
- [45] D. Meyer and P. Saint-Andre. XTLS: End-to-End Encryption for the Extensible Messaging and Presence Protocol (XMPP) Using Transport Layer Security (TLS).  
<http://tools.ietf.org/html/draft-meyer-xmpp-e2e-encryption-02>, June 2009.
- [46] M. Millard and P. Saint-Andre. XEP-0079: Advanced Message Processing.  
<http://xmpp.org/extensions/xep-0079.html>.
- [47] J. Moffitt. Binary Data is XMPP's Achilles Heel.  
<http://metajack.im/2008/06/10/binary-data-is-xmpps-achilles-heel>, June 2008.
- [48] J. Moffitt. Thoughts On Scalable XMPP Bots.  
<http://metajack.im/2008/08/04/thoughts-on-scalable-xmpp-bots/>, August 2008.
- [49] T. Muldowney. XEP-0027: Current Jabber OpenPGP Usage.  
<http://xmpp.org/extensions/xep-0027.html>.

- [50] M. A. Murphy, L. Abraham, M. Fenn, and S. Goasguen. Autonomic Clouds on the Grid. *Journal of Grid Computing*, 8(1):1–18, March 2010.
- [51] 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.
- [52] M. A. Murphy, B. Kagey, M. Fenn, and S. Goasguen. Dynamic Provisioning of Virtual Organization Clusters. In *9th IEEE International Symposium on Cluster Computing and the Grid (CCGrid '09)*, Shanghai, China, May 2009.
- [53] B. Nitzberg, J. M. Schopf, and J. P. Jones. PBS Pro: Grid computing and scheduling attributes. pages 183–190, 2004.
- [54] D. Nurmi, R. Wolski, C. Grzegorzczuk, 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.
- [55] 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.
- [56] I. Raicu, I. Foster, and Y. Zhao. Many-task computing for grids and supercomputers. In *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*, pages 1–11, Nov. 2008.
- [57] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falcon: 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.
- [58] P. Ruth, X. Jiang, D. Xu, and S. Goasguen. Virtual Distributed Environments in a Shared Infrastructure. *Computer*, 38(5):63–69, 2005.
- [59] P. Saint-Andre. XEP-0001: XMPP Extension Protocols. <http://xmpp.org/extensions/xep-0001.html>.
- [60] P. Saint-Andre. XEP-0134: XMPP Design Guidelines. <http://xmpp.org/extensions/xep-0134.html>.
- [61] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. <http://www.ietf.org/rfc/rfc3920.txt>, October 2004.
- [62] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. <http://www.ietf.org/rfc/rfc3921.txt>, October 2004.
- [63] S. Sanfilippo. Redis. <http://redis.io>.
- [64] 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.
- [65] I. Sfiligoi. glideinWMS - a generic pilot-based workload management system. In *Journal of Physics: Conference Series*, volume 119, page 062044. IOP Publishing, 2008.

- [66] R. Shingledecker. Micro Core Linux.  
<http://www.tinycorelinux.com/>.
- [67] 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.
- [68] L. Stout. SleekRedis.  
<http://github.com/legastero/SleekRedis>.
- [69] 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.
- [70] 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.
- [71] 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.
- [72] 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.
- [73] 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.
- [74] 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.
- [75] 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.
- [76] 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.
- [77] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu. Parallel scripting for applications at the petascale and beyond. *Computer*, 42(11):50–60, 2009.
- [78] R. Yasin. DOD wants instant messaging tools to speak the same language.  
<http://defensesystems.com/articles/2010/06/23/dod-instant-messaging-test.aspx>, June 2010.