Kestrel: An XMPP-Based Framework for Many Task Computing Applications

Lance Stout, Michael A. Murphy, and Sebastien Goasguen
School of Computing
Clemson University
Clemson, SC 29634-0974 USA
{lstout, mamurph, sebgoa}@clemson.edu

ABSTRACT

This paper presents a new distributed computing framework for Many Task Computing (MTC) applications, based on the Extensible Messaging and Presence Protocol (XMPP). A lightweight, highly available system, named Kestrel, has been developed to explore XMPP-based techniques for improving MTC system tolerance to faults that result from scaling and intermittent computing agent presence. By leveraging technologies used in large instant messaging systems that scale to millions of clients, this MTC system is designed to scale to millions of agents at various levels of granularity: cores, machines, clusters, and even sensors, which makes it a good fit for MTC.

Kestrel's architecture is inspired by the distributed design of pilot job frameworks on the grid as well as botnets, with the addition of a commodity instant messaging protocol for communications. Whereas botnet command-and-control systems have frequently used a combination of Internet Relay Chat (IRC), Distributed Hash Table (DHT), and other Peer-to-Peer (P2P) technologies, Kestrel utilizes XMPP for its presence notification capabilities, which allow the system to maintain continuous tracking of machine presence and state in real time. XMPP is also easily extensible with application-specific subprotocols, which can be utilized to transfer machine profile descriptions and job requirements. These sub-protocols can be used to implement distributed matching of jobs to systems, using a mechanism similar to ClassAds in the Condor High Throughput Computing (HTC) system.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Design, Experimentation

MTAGS '09 November 16th, 2009, Portland, Oregon, USA.

Keywords

MTC, XMPP, grid, distributed, cluster, scheduling

1. INTRODUCTION

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" [1] 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 (VOC) [2, 3] where compute nodes can be created and destroyed on demand.

Kestrel is designed to explore methods of creating a fault-tolerant, scalable, and cross-platform job scheduling system for heterogeneous, intermittently connected compute nodes. Kestrel addresses the problem posed by interrupted connectivity of both computing agents and user agents by building on the Extensible Messaging and Presence Protocol (XMPP) [4, 5]. 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 computings 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 heterogeneos agents for real-time computing. The cross-platform criterion is achieved by implementing the system in Python [6] using the SleekXMPP [7] library.

The remainder of this paper is organized as follows. Related work is presented in section 2. The network architecture of Kestrel is discussed in section 3, after which the specific network protocols are presented in section 4. Section 5 describes the message protocol format used for transmission of Kestrel messages over the network. An overview of the architecture of the current implementation of Kestrel is presented in section 6. Conclusions and future work are described in section 7.

2. RELATED WORK

The Condor High Throughput Computing (HTC) system

specializes in managing computationally intensive jobs and is used around the globe for managing grids and clusters [8, 9]. The Condor architecture is composed of a central manager running collector and negotiator daemons, submit nodes running a schedd daemon, and execute nodes running a startd daemon. The submit and execute nodes communicate with the central manager to share ClassAds describing the capabilities of the machine. Periodically, job requests are matched against the capabilities advertised in the ClassAds. Once a match is made and the affected nodes notified, a direct connection is established between the matched submit and execute node to allow both the job executable and the job output to be transferred between the machines. While Condor is an excellent system for cycle scavenging and cluster computing on physical hosts, the direct connections between submit host and worker cannot cross NAT boundaries. In addition, since collector updates occur periodically instead of continuously, operation may be sub-optimal in environments with intermittently connected worker nodes. Kestrel is inspired by the Condor architecture but replaces all communications with XMPP which adds a real-time computing capability as well as extremely large scale.

An XMPP based scheduling system was implemented in Weis and Lewis [10] for ad-hoc grid computing.This system was developed specifically to parallelize the computation of optimized meander line radio frequency identification (RFID) antennas. With a static list of machines available for use by the scheduler, the controlling XMPP client would contact any machines not connected to the XMPP server to start the XMPP worker daemon [10]. While similar in structure to Kestrel, the scheduler was intended for distributing jobs of a single type to known machines; it was not meant as a general MTC framework.

A proposed hybrid peer to peer (P2P) botnet, proposed in Wang et al. [11], replaces the single command and control machine found in typical botnet architectures with a group of servant bots interconnected via a P2P protocol. As an additional means to avoid detection, the botnet does not use Internet relay chat (IRC) – commonly used by other botnets – for its communications channel. [11] Kestrel utilizes a similar distributed command and control architecture for increasing fault-tolerance; indeed, multiple XMPP servers offer both redundancy and horizontal scalability.

3. NETWORK ARCHITECTURE

The network in a Kestrel pool is comprised of four types of nodes: managers, workers, users, and XMPP servers. It is possible for more than one node to be present on a single physical or virtual machine. The XMPP servers may be configured to act as a cluster using a distributed database to keep roster and status information synchronized [12, 13]. All other nodes then connect to an XMPP server. Since an intermediary is used for communication instead of requiring nodes to connect directly to each other, NAT traversal is not an issue.

User nodes are client applications used to submit job requests or query the status of previously submitted jobs. These nodes are highly transient, connected only while sending a user request to a manager node and waiting for a reply. Job execution can take place without requiring the submitter's computer to remain connected to the pool. Upon the user's next connection to the system, completion notices can be requested for previously submitted jobs.

Manager nodes serve as the equivalent of a collector and negotiator combination in a Condor pool. As such, every message sent in a Kestrel pool is either sent to or sent from a manager node. Due to the large number of messages processed by a manager, particularly presence notifications, there are two classes of managers, which use two different XMPP implementation methods. The first class of manager is a regular XMPP client application that is useful for pools with only a few hundred to a thousand nodes [14]. By using a regular client, the XMPP server can immediately send offline presence notifications to worker and user nodes if the manager disconnects, since the server maintains the manager's roster. However, once a pool has grown past several thousand machines, there is a significant increase in startup times because the roster must be sent to the manager, blocking any other messages. Thus, the second class of manager is an XMPP server component that is able to act like a normal client, except that it maintains its own roster instead of relying on the server. If either type of manager goes offline, the rest of the pool will still be able to complete any previously scheduled jobs.

Worker nodes track local machine status and handle actual job execution. Using XMPP presence notification, a worker alerts the manager by changing its online status to "available" when it is available to run queued jobs. Once a job is started, the worker node status changes to "busy," allowing the manager to bypass the claimed worker when scheduling new jobs. In the event that a worker is terminated or the underlying machine is powered off, the manager will receive an offline presence notification allowing any jobs that had been running on the failed worker to be rescheduled on another worker.

4. NETWORK PROTOCOL

4.1 Identifiers

Every entity inside an XMPP system has a unique identifier known as a Jabber ID, or JID [4]. These identifiers have the form `username@server/resource`. The username and server portion of the JID, or bare JID, appears identical in structure to a regular email address. If specified, the optional resource string allows for an identity to have multiple connections to the XMPP server. In Kestrel, the pool of worker agents is organized by assigning a username to each physical machine, allowing the host to be identified by the bare JID. Each CPU core of the machine is individually addressed by adding a resource to the JID. Thus, the third core in a quad core machine could be identified by `worker362@kestrelpool/3`. Security is provided by allowing each agent in the pool to have its own password.

An alternative implementation would be to use a single bare JID to address all worker nodes in the pool, with each worker uniquely addressable by adding a resource identifier, producing a JID such as `worker@kestrelpool/362`. By utilizing different usernames, workers can be logically split into categories by physical location, operating system, or intended use. Using this naming convention has the benefit of drastically reducing the number of entries in the manager agent's roster. However, this implementation requires that all machines sharing the same bare JID also share the same password.

Kestrel is able to utilize either naming convention. In addition, Kestrel is extensible to other naming conventions,

as long as each agent in the pool has a unique JID.

4.2 Message Types

The XMPP protocol provides three different message types. The first type, presence notification, is used to indicate the availability of an agent and can carry a status message describing the agent's current state. Chat messages are used mainly for sending instant messages between humans. These messages do not require any acknowledgment in reply. XMPP guarantees that chat messages will be received in order, but not that they will arrive. Messages requiring a response are called "iq" requests. These requests return status codes in response to each query message. Within an iq message, both content that adheres to the ad-hoc command format [15]or other, custom namespaced XML content can be transferred.

Kestrel uses presence notifications to track availability of worker agents and chat messages for communication of commands and data between agents. Other XMPP based scheduling systems have used an iq message approach using an XML protocol for inter-agent messages. Such an arrangement was described by Weis and Lewis for use in an ad-hoc grid computing system for generating RFID antennas [10]. However, to simplify development and testing, Kestrel has used chat messages that can be sent via a stock instant messaging client. Thus, the pool can be managed using Pidgin [16], Adium [17], or any other XMPP capable instant messaging client, as well as through normal command-line programs. The content of the chat messages follows a custom sub-protocol using JavaScript Object Notation (JSON) [18] instead of XML. The content is stored in a JSON dictionary, which always includes a "type" attribute specifying the purpose of the message. For example, a type of `worker_profile` would indicate a message containing machine attributes sent from a worker agent to a manager agent, as shown in figure 1. A typical sequence of messages generated during the lifetime of a worker are shown in figure 2.

```
<message from="manager@kestrel_pool"
         to="worker42@kestrel_pool">
  {"type": "profile_request"}
</message>
```

Figure 1: A request for details about the machine managed by a worker agent.

5. PROFILE FORMATS

The format chosen to represent machine profiles (the Kestrel equivalent of a Condor ClassAd) is based on JavaScript Object Notation (JSON). The JSON standard provides for key-value pairs in the form of dictionaries and also support for lists and other data structures. By using this format, a machine profile can be easily read and parsed using standard library calls. A few machine attributes are universal, such as the name of the operating system, the number of CPU cores, and the amount of available RAM. Additional, optional attributes for a machine can be specified through the use of tags. Each machine profile may contain a "provides" section, consisting of a list of custom tags as illustrated in figure 3.

The format for a job request is similar to a machine profile. The difference is that it specifies what a job requires

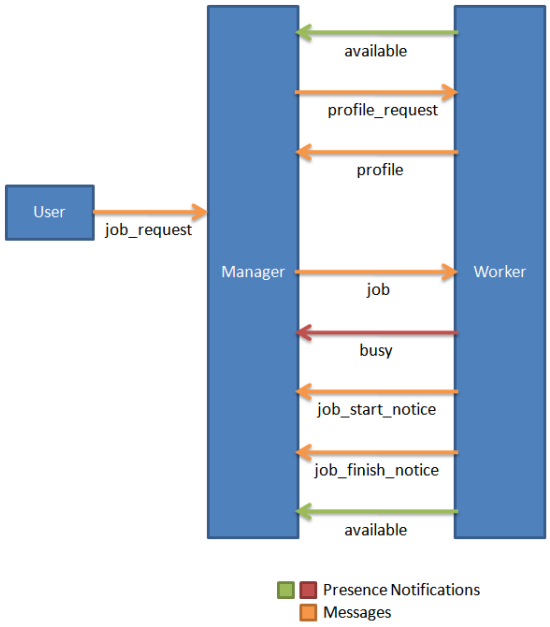

Figure 2: Flow of information during the life cycle of a worker agent.

```
{
  "os": "Ubuntu",
  "os_version": "9.04",
  "cores": "4",
  "ram": "4032",
  "provides": ["PYTHON_2.6", "MERCURIAL_1.2"]
}
```

Figure 3: Sample Machine Profile

rather than what a machine provides. The JSON standard does not have a built in facility for expressing the relationships needed for requirement specifications. To overcome this limitation, a way to express relationships using dictionaries and lists was developed. Each job request contains the attribute "requires" which can be either a dictionary or a list. In the case of a list, then it is just a listing of custom tags that must be matched. A dictionary signifies that more specific rules are requested. Each key in the dictionary is the name of the attribute being compared in the relationship. Equality is expressed by assigning a string or numeric value to that attribute. Thus, to express "cores == 2" the following would be used:

```
"requires": {"cores": 2}
```

However, sometimes there is a set of possible values that are acceptable. In this case, a list of values is assigned to the attribute to represent an OR relationship. The relationship "cores == 2 OR cores == 4" would be expressed as:

```
"requires": {"cores": [2, 4]}
```

Another use case is specifying relationships other than equality, such as greater than relations. Also, since specifying multiple such relationships would normally make sense in an AND expression, dictionaries are used to represent

these requirements. For the relation "cores > 1 and cores < 10" then the representation would be:

```
"requires": {"cores": {">": 1, "<": 10}}
```

The list and dictionary semantics can be combined to make more powerful expressions. Specifying that a job requires a machine where "cores = 4 OR cores > 6" is true, then the "requires" attribute would be:

```
"requires": {"cores": [4, {">": 6}]}
```

Likewise, the statement "cores != 2 AND cores != 5" can be represented as:

```
"requires": {"cores": {"!=": [2, 5]}}
```

Any job request that requires both complex relationship rules and matching against a set of tags, the "has" attribute can be used in the "requires" section.

```
{
  "command": "monte_carlo.py",
  "queue": 1000,
  "requires": {"cores": 4,
               "has": "PYTHON_2.6"}
}
```

Figure 4: A job request for executing monte_carlo.py 1000 times on quad-core machines that have Python 2.6 installed.

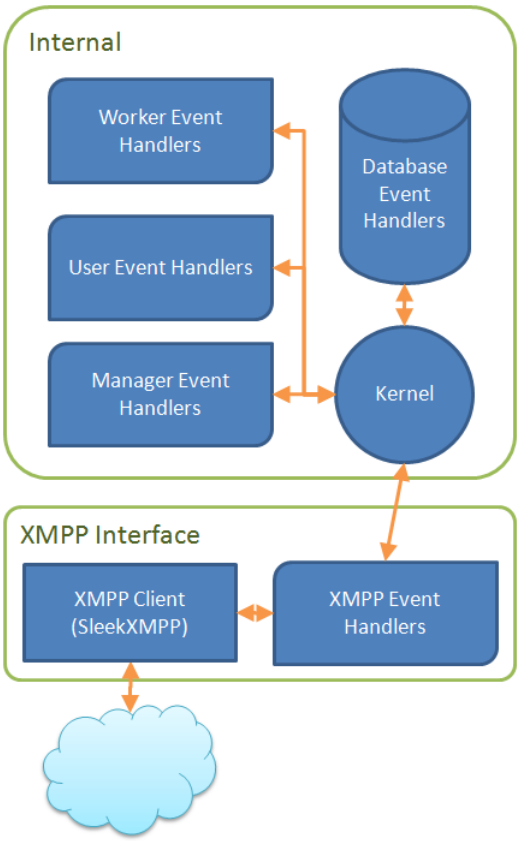

Figure 5: Program Architecture.

6. PROGRAM ARCHITECTURE

The Kestrel software has several logical components, including an XMPP interface, database interface, and job scheduling mechanism. Early versions of Kestrel combined these components into a single tier with all system logic located in the XMPP client code. Unit testing of the early implementations revealed issues related to tight coupling. As a result, the implementation architecture was changed to an event driven design.

Event driven architectures allow sections of a program to be separated from each other, resulting in loose coupling. Decoupling boundaries can be established whereby all the components inside a boundary can directly access each other, and messages can be sent across the decoupling border through events [19].In Kestrel, these events are managed by a central kernel. Event names are registered with the corresponding event handler at application initiation time. The registration allows for the different program variations for worker, manager, and users to be created using a single code-base by selecting role-specific events.

Figure 5 shows the relationship between the various collections of event handlers. These collections are either for use internally in any of the three program variations, or are used as an interface between the outside world and the internal system. For production use, XMPP event handlers provide the interface component. For testing purposes, a unit test runner provides a separate, more limited interface.

6.1 Event Types

There are three main types of events used in Kestrel. The first group is related to data storage. Instead of calling database functions or passing SQL directly to a database while processing XMPP events, events are triggered for create, read, update, or delete (CRUD) operations on each type of entity. For example, an `update_worker_entry` event is triggered whenever a worker comes online, resulting in an update to its online status in the database. For read operations, the database event handler will also return the appropriate database record.

The second group of events can be split into subgroups matching the three program variations. Worker events are related to receiving job requests and the starting and stopping of job execution. User events are the simplest group, receiving responses to queries to the manager about the status of jobs and handling any cancellation requests.

The largest collection of events are manager-related. In addition to events triggered by worker status changes and user requests, there are job scheduling events. The `schedule_jobs` event is triggered after every job request and cancellation and whenever a worker becomes available. This event is not periodic, since a job that cannot be scheduled immediately will still not be runnable until the state of the pool changes. Since scheduling is a separate event in the system, the scheduling algorithm is easily replaceable by specifying a different event handler during registration.

The final set of events is related to node communication. In the current implementation of Kestrel, these events are handled by XMPP-specific functions that form an interface between the internal system and the network. In particular, there are many cases where messages must be sent between nodes, such as notifying a worker of a job match. In that

case, a `send_job_request` event is triggered to alert the XMPP interface to send the actual message using the data provided in the event. It would be possible to create other interfaces using other network protocols as long as the same set of events could be provided.

6.2 Event Anatomy

Each event carries two pieces of information: the data passed during event triggering and an event trace (figure 6). When an event is triggered at the top level of the program, the event trace is empty. When the event handler finishes processing and returns to the top level, the trace will contain the name of every event triggered as a result of the initial event. In addition to the name of each event, the trace includes the data passed to that event. For events that do not occur at the top level, the trace will already contain information on the events that have already been triggered. The trace is important because it allows chains of events to be checked with unit tests.

```
[
    ("worker_available", {
       "jid":"worker@kestrel_pool"}),

    ("update_worker_entry", {
       "jid":'worker@kestrel_pool",
       "status":"AVAILABLE"}),

    ("schedule_worker", {
       "jid":'worker@kestrel_pool"})
]
```

Figure 6: An event trace generated by a worker changing its status to available.

7. CONCLUSIONS AND FUTURE WORK

The use of XMPP for communicating between compute nodes enables resource tracking in real time. By eliminating delays between the disconnection of a compute node and the manager node receiving notice of the disconnect, fewer jobs can be scheduled for phantom machines. The real time tracking applies to the manager nodes as well, allowing the system to respond to any outages or overloading by spawning new manager nodes immediately, retaining both fault tolerance and scalability.

Future extensions to Kestrel could include more robust mechanisms for mitigating potential data loss after node disconnects. For example, job queue data must be maintained whenever a manager goes offline. Other information in the pool is inherently decentralized: each worker knows the specifics of the local machine upon which it is running. Thus, as workers come back online, machine state information is restored in the system. However, job information is centralized in manager nodes. In the event of termination or outage, if a manager is replaced with a new node that has no prior knowledge of jobs in the pool, then all pending jobs will be lost. In a more optimal case, the same manager node is able to restart with its database intact, thereby avoiding job losses except for jobs submitted during the outage.

To ensure that jobs are retained after the loss of the manager, a simple archival strategy can be used. In addition to executing jobs, worker nodes are employed as archivists, saving job requests as backups. Upon receiving a job request, the manager chooses a number of workers and then forwards the job for storage rather than execution. When a worker node comes online, the profile it sends to the manager includes a flag indicating whether or not data have been archived on the worker . The manager can request that the archived data be sent, so that the database can be updated with any new job entries provided by the worker. Different archival storage policies are possible, enabling different sets of workers to be selected to store portions of the database.

8. REFERENCES

[1] 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*, Nov. 2008, pp. 1–11.

[2] 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.

[3] 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.

[4] P. Saint-Andre. (2004, October) Extensible messaging and presence protocol (xmpp): Core. IETF. [Online]. Available: http://www.ietf.org/rfc/rfc3920.txt

[5] ——. (2004, October) Extensible messaging and presence protocol (xmpp): Instant messaging and presence. IETF. [Online]. Available: http://www.ietf.org/rfc/rfc3921.txt

[6] Python programming language. Python Software Foundation. [Online]. Available: http://www.python.org

[7] N. Fritz. [Online]. Available: http://code.google.com/p/sleekxmpp/

[8] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor – a distributed job scheduler," in *Beowulf Cluster Computing with Linux*, T. Sterling, Ed. MIT Press, October 2001.

[9] M. L. Douglas Thain, Todd Tannenbaum, "How to measure a large open source distributed system," *Concurrency and Computation: Practice and Experience*, vol. 8, no. 15, December 2006.

[10] 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*, May 2009, pp. 1–4.

[11] C. Z. P Wang, S Sparks, "An advanced hybrid peer-to-peer botnet," in *Dependable and Secure Computing, IEEE Transactions on : Accepted for future publication*, vol. PP, 2003.

[12] Ejabberd. Process One. [Online]. Available: http://www.process-one.net/en/ejabberd/

[13] badlop. (2006, 12) Ejabberd. [Online]. Available: http://www.ejabberd.im/features

[14] J. Moffitt. (2008, August) Thoughts on scalable xmpp bots. [Online]. Available:

http://metajack.im/2008/08/04/thoughts-on-scalable-xmpp-bots/
[15] M. Miller. (2005, June) Xep-0050: Ad-hoc commands. XMPP Standards Foundation. [Online]. Available: http://xmpp.org/extensions/xep-0050.html
[16] Pidgin, the universal chat client. [Online]. Available: http://www.pidgin.im
[17] Adium. [Online]. Available: http://www.adium.im
[18] D. Crockford. Introducing json. [Online]. Available: json.org
[19] J. van Hoof. (2006, November) How eda extends soa and why it is important. [Online]. Available: http://soa-eda.blogspot.com/2006/11/how-eda-extends-soa-and-why-it-is.html