StackMon: A Lightweight Monitoring Framework for OpenStack Clouds

Brian Laub and Arun Narasimhan
CS6210 Fall 2012 Final Project
Professor Karsten Schwan

Georgia Institute of Technology
Atlanta, GA USA
{blaub6, arunpn}@gatech.edu

Abstract: Performance monitoring and resource utilization are becoming increasingly important in large-scale, virtualized cloud computing platforms. As developers begin to rely more on Infrastructure-as-a-Service paradigms, detailed monitoring of virtualized resources will continue to help guide application and system design. The StackMon project aims to create a monitoring system for virtualized clouds built on the OpenStack toolset. We develop a small, lightweight monitoring framework based on a publish/subscribe messaging model, capable of capturing performance metrics from the Xen hypervisor, and aggregating data in a centralized location for analysis. We test our approach by implementing a plugin for the OpenStack Dashboard to visualize performance metrics in real-time, and consider our approach with respect to future research efforts for large-scale monitoring in cloud computing systems.

1. Introduction

Cloud computing has seen an explosion in popularity. The flexibility of the cloud offers tremendous benefits to systems designers and application developers today. Commercial cloud platforms such as Amazon Web Services [4] or Windows Azure [5] provide a virtualized datacenter through a commodity business model, where users pay only for the resources they use saving them the expense of maintaining an in-house datacenter. This architecture is extremely enticing, but also deceptive from the standpoint of traditional systems design. Porting an application to the cloud can often come at the expense of some unforeseen performance fluctuations due to virtualization, network latency, or multi-tenancy. Detailed monitoring of virtual machine performance characteristics can therefore be an invaluable tool for users of cloud resources.

The OpenStack [6] framework offers a full-scale open-source implementation of a cloud computing platform. While commercial systems such as Amazon Web Services provide monitoring systems, we observe that OpenStack's toolset provides relatively few options in this regard. As such, we set out to develop a proof-of-concept for building precise instance measuring into an OpenStack deployment, and to offer a stepping stone for future researchers to explore large-scale monitoring systems for virtualized cloud environments built on the OpenStack framework.

Our contributions in this project are:

- demonstration of a simple, lightweight monitoring system for virtualized datacenters

- integration of our monitoring system into an OpenStack IaaS deployment

The rest of this paper is organized as follows: in Section 2, we offer some background on monitoring systems, with a summary of related work. Section 3 describes the details of our system design and implementation. Section 4 presents a performance evaluation, our experiences deploying our system in a research cloud environment, and considers the overall effectiveness of our approach. In Section 5, we offer suggestions for improvements and future efforts. Section 6 details our conclusions and summarizes our contribution to the research community. In the appendix, we relate our work to concepts studied in CS6210 this semester.

2. Background and Related Work

System monitoring in a large-scale distributed environment offers many challenges. Cloud platforms are increasingly exhibiting many of the same characteristics previously offered only by high-performance computing platforms. As such, monitoring can be an important aspect of system design and deployment in the cloud. To gain a deeper understanding of large-scale system monitoring, we studied existing cluster monitoring projects to get an idea for some of the goals, and challenges, of past work. We discovered that there have been a variety approaches taken to solve the problem of monitoring the resource utilization and performance of cluster-based systems like the ones typically used in cloud computing platforms.

Supermon [11] presents a flexible set of tools for high speed scalable cluster monitoring. It introduces the concept of a "mon" (single node data server) and "supermon" (data concentrator which aggregates data samples). Supermon's approach uses a protocol based on symbolic expressions for data representation, demonstrating scalability in heterogenous environments.

The RVision project [8] provides flexibility in selecting events to be monitored, and allows users to to expand the monitoring capabilities with self-defined procedures for monitoring of specific hardware or system events. Our approach uses a publish/subscribe methodology, which was influenced in part by the ideas presented by RVision. We were also inspired by this to consider carefully the dataset we monitor in our system.

Ganglia ([1], [2]) is a well-known cluster monitoring project originally developed at the University of California, Berkeley. It uses a hierarchical approach for scalability. Attributes in Ganglia are replicated within clusters using multicast methods and aggregated via a tree structure. Each node in the tree hierarchy shares the processing load by providing distilled summaries of monitoring data to its parent, thereby reducing load on the system while enabling multiple-resolution view of the tree. Compared to Supermon which requires *O(Number of hosts)* network connections, Ganglia requires just one connection (to its multicast channel).

The OVIS project [9] extends the basic monitoring idea by proposing automated analysis of data sets. More traditional systems use simple concepts such as thresholds to indicate problems. OVIS proposes statistical methods such as Bayesian inference to characterize system behavior in a

large-scale cluster. Other projects consider similar functionality. The Monalytics [10] system combines data distribution and aggregation with arbitrary analysis tasks, allowing dynamic deployment and reconfiguration of monalytics graphs. In [7], the authors build upon Monalytics by designing a system capable of constructing software overlays called Distributed Computation Graphs to support a variety of analytics functions.

The VScope project [3] tackles scalability by proposing a middleware for troubleshooting time sensitive data center applications, and introduces methods like Dynamic Watch, Scope and Query, Guidance and Distributed Processing Graphs. The authors propose lightweight anomaly detection as a "first pass" filter, followed by targeted, dynamically deployed, detailed system monitoring on a subset of servers to concentrate diagnostics where they are likely to be more fruitful.

With respect to communications layers, we also explored EVPath [12], an ongoing research effort at Georgia Tech that provides a flexible event-driven messaging system for building overlay networks. A previous research project developed at Georgia Tech, called DiCoMo (for "Distributed Cloud Monitor") explored distributed monitoring at the operating system level for clustered environments. DiCoMo is built on EVPath, and collects statistics using standard interfaces provided by POSIX and the Linux kernel. Our work is similar conceptually to DiCoMo, but we extend the monitoring concept to the hypervisor level, and focus monitoring for cloud datacenter clusters.

All of the concepts from these papers and past projects have influenced our project design and development in one way or another, even if they did not show immediately in our prototype implementation. The main goal of our project is to examine developing a system influenced by the design of past monitoring systems with applications to the OpenStack platform, and hopefully to provide a basis for future research in this area.

3. Design and Implementation

3.1 Design Overview

Our system contains three core components that facilitate monitoring of virtual machine instances in a cloud environment:

- a *monitoring infrastructure* that collects performance data, and transmits it to a central location
- a *data aggregation layer* that stores monitor data for future analysis
- a *visualization layer* that runs queries to analyze performance data

Figure 1 displays a high-level overview of our architecture, including interaction between components.

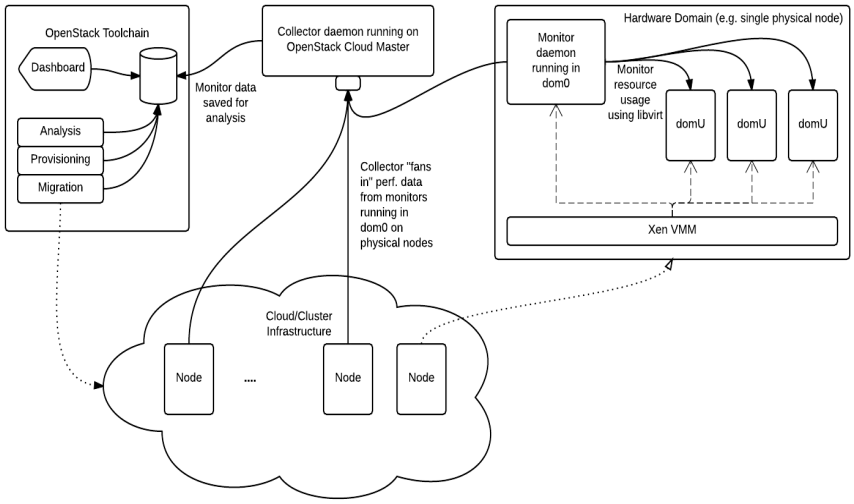

Figure 1: Overall StackMon System Design

For the monitoring infrastructure, we have built our communications on a publish/subscribe messaging model. There are two key artifacts: a *monitor* daemon and a *collector* daemon, representing "publishers" and "subscribers" respectively. Monitor daemons run in Xen's dom0 on a physical host in the cloud infrastructure, and collect performance metrics about active VM instances. This data is published via the network to a subscribed collector daemon. The collector daemon aggregates data collected from all monitors in the system and publishes it to the data aggregation layer, which is responsible for storing it in a database. The visualization layer queries the databases managed by the data aggregation layer to analyze, visualize, or otherwise do something useful with the data.

3.2 System Components

In this section, we provide a more thorough description of each system component's role and implementation.

Monitor Daemon

The monitor daemon is deployed on an OpenStack compute node. It monitors instance performance metrics using libvirt [13], and reports these metrics to a collector daemon at regular intervals. Our current implementation only supports the Xen hypervisor, but can be adapted to work with other hypervisors supported by libvirt as well, such as KVM [14] or VMware ESX [26].

The daemon architecture is currently built around a polling interface. The top level event driver is a set of `DomainStatPublisher` objects. This is a generic interface that is responsible for publishing monitored data about a set of domains to the network. In the current implementation, a single concrete object, `LibvirtDomainStatPublisher`, collects and publishes this data, but other types

of metric gathering abstractions are possible.

On each update interval, the daemon asks its set of `DomainStatPublishers` to collect a set of data and publish it. The `LibvirtDomainStatPublisher` encapsulates an `ActiveDomainWatcher` object that resolves a list of active domains in the system on each update. When a new instance is launched, a callback is issued to the stat publisher to notify it that a previously untracked domain has appeared. In response, the stat publisher instantiates a `DomainStatTracker` object for the domain. Similarly, when an instance is terminated, a callback is triggered that allows the stat publisher to remove the `DomainStatTracker` for the instance that was terminated. Additionally, a message is published to the collector notifying it that the domain has terminated, to allow it to clean up any resources associated with the instance.

The `LibvirtDomainStatPublisher` maintains a set of `DomainStatTracker` objects, one for each active instance in the system. A `DomainStatTracker` is responsible for gathering metrics for a single domain in the system (we include a tracker for dom0 as well). When asked to publish its data, the stat publisher forwards this request to each of its active stat trackers, each of which queries the libvirt API to collect monitored data about the instance it is responsible for. These queries are assembled using libvirt API calls and gathered into a `DomainStats` structure. Table 1 summarizes the data our current prototype collects from the hypervisor. For the collection method, we either return raw values as reported by the libvirt API, or collect a percentage of usage based upon successive sampling relative to the wall clock in dom0 (or some combination, based upon the metric being gathered).

Metric	**Collection Method**	**Description**
CPU Utilization (%)	Sampling	% of vCPU currently being used by the domain
Memory Utilization (%)[1]	Direct Poll	% of memory currently being used by the domain (mem used out of max allowed)
Block Device (VBD) Statistics	Sampling/Direct Poll	Various block device statistics reported by libvirt, including: read/write requests, read/write sizes (bytes), disk errors
Network (VIF) Interface Statistics[2]	Sampling/Direct Poll	No. packets and bytes transmitted/received

Table 1: VM performance metrics collected by a `DomainStatTracker`

After collecting all data from each `DomainStatTracker` in the system, the stat publisher aggregates this data into an `AggregateDomainStats` structure. This structure is serialized, and published as a

[1] Memory utilization is currently always reported as "100%" for reasons unknown. We conjecture this may be due to ballooning in Xen.

[2] Though we have built support for monitoring VIF statistics, this is currently not thoroughly tested due to limitations in our cloud testbed.

bytestream to the network to a subscribed collector.

Our current polling interval is 1 second. Some testing has indicated this is sufficient to catch most relatively coarse-grained performance statistics (e.g. extended CPU spikes), and requires fairly low overhead.

Collector Daemon

A *collector* daemon acts as a subscriber in the system. It listens for resource metrics on a particular socket and internally publishes it to the data aggregation layer. This works on a event-driven model - whenever a subscribed-for message is received, it triggers an event that causes the data to be published to a listening process to update some databases.

The basic collector daemon in the current prototype is fairly straightforward - it simply parrots the data it receives to another publish socket. In turn, other applications can connect and take action when data is published. This amounts to a simple "fan-in fan-out" architecture, whereby the collector provides a centralized, stable point for monitor daemons to broadcast their data to. This flexibility allows for different data aggregation layers or other visualization tools to be built, as well as for more flexible collector designs (discussed below).

Intermediate Collector Daemon

As the size of the OpenStack deployment grows, having a single collector for all the Monitors becomes a bottleneck. Thus, we have also experimented with a hierarchical design for the publish/subscribe architecture to improve scalability. An "Intermediate Collector" acts as a mid-level aggregator, collecting data from a subset of all monitors in the system and forwarding them to the next collector up in the hierarchy. This helps in isolating failures and also in decentralizing data collection to some extent.

The Intermediate Collector performs forwarding of data to the next level of Collector in an event driven fashion. For instance, when a Monitor daemon sends a control message to the Intermediate-Collector stating that a domain went offline, we bubble up this information to the next collector in the topology until it reaches the "Master" Collector where we could present this data in the OpenStack dashboard to the user.

For large-scale deployments, we envision a hierarchy designed around the datacenter itself. As an example, consider a datacenter with high density (many nodes per rack, and many racks per datacenter). The simple fan-in architecture of the basic collector might quickly exhaust operating system resources. A hierarchical system can alleviate this. We consider this approach further in Section 5.

Data Aggregation

The data collection layer is responsible for flushing received monitor data to stable storage. Our current prototype uses the Whisper [15] RRD database protocol to do this. This part of our project is implemented in Python for simplicity (this was done to cut down on cross-language

programming, as Whisper is also written in Python). In our current prototype, a "proxy" object (called `WhisperRRDProxy`) acts as an intermediate between the collector and the data aggregator. `WhisperRRDProxy` connects to the collector daemon's fan-out socket, transforms the data received into a simpler, text-based protocol that our Python script can understand, and write it to the script's stdin.

Our Python script (`whisper_updater.py`) is responsible for maintaining the set of RRDs on disk for each active domain. It creates a round-robin database for each monitored metric (see Table 1) for the domain. The current implementation retains 3 hours worth of data - approximately 128KB per metric. `whisper_updater.py` also receives messages when an instance is terminated, allowing it to remove the RRDs for that instance.

OpenStack Dashboard Plugin

Once `whisper_updater.py` has flushed data to disk, the OpenStack Dashboard can be used to run queries on the RRD databases and display monitored data to users. We implemented this as a proof-of-concept to test the effectiveness of our monitoring infrastructure. The current prototype simply displays a "snapshot" (most recent update) of vCPU, memory utilization, and I/O requests, as well as the vCPU 1, 5, and 15 minute load averages. Our code leverages existing Django code written for the OpenStack Dashboard, which allows it to integrate fairly seamlessly with a deployment. We developed this code as part of an OpenStack deployment on Ubuntu Linux 12.04.

As the OpenStack Dashboard is a Django [16] web application, this component is written entirely in Python.

3.3 Implementation Details and Development Experience

We leverage several key open-source software libraries for our implementation of StackMon. To abstract the complexities of network-based message passing, we use ZeroMQ [17], which provides a BSD sockets-like interface with greater flexibility. ZeroMQ is a portable, fully asynchronous messaging framework, and takes advantage of OS-level event-driven communications optimizations (such as `epoll` [18] and `kqueue` [19]), to provide good scalability and performance. The library includes a built-in publish/subscribe paradigm, which helped us prototype our design quickly.

For message serialization, we use MessagePack [20], a lightweight serialization library similar in spirit to JSON [21], but designed to be small and fast. MessagePack contains multiple language bindings which again helped us prototype our system quickly, allowing us to evaluate system functionality using scripts, and then port our code to C++ to improve performance.

Early in our development, we experimented with using EVPath as well. We note that ultimately EVPath provides even more flexibility in designing a communications system (such as the publish/subscribe model we use). Due to time constraints and spin-up time required to learn EVPath, we elected to use other libraries to get a working prototype built quickly. However, our

"core" system components could easily be integrated into a system built on EVPath as well.

4. Evaluation and Experiences

In this section, we provide a brief performance evaluation and discuss our experiences using our monitoring system deployed on an OpenStack research cloud.

4.1 System Utilization and Overhead

To gain some insight on the performance characteristics of our system, we measured two key areas we felt would be important in typical cloud datacenter environments:

- CPU and memory overhead of the monitor daemon running in dom0
- network load

To evaluate the overhead of running our monitor daemon in a Xen dom0 domain, we ran the monitor daemon on a single OpenStack Compute node and launched several VMs. We simulated an arbitrary CPU-intensive workload in each VM. Due to time constraints, we were not able to evaluate memory or I/O intensive guest workloads. The monitor daemon was connected to a single collector daemon during the experiment. Our goal with these experiments is to evaluate the overhead of the monitor daemon running in a cloud environment where many guest VMs may be active and consuming physical resources on the compute node. We tested between 1 and 10 active VMs on the compute node. We believe this represents a fairly typical workload for a physical node in a cloud. While Xen can certainly support more guest domains, a typical cloud deployment would strive to keep the number of guests per physical host low to maximize resources available to VMs.

Our compute node (dom0) testbed runs Ubuntu Linux 12.04 (Linux kernel 3.2.0) running Xen 4.1. The server has 2 quad-core Intel Xeon E5530 CPUs running at 2.4GHz each, and 40GB RAM. Each VM instance was small, and ran CirrOS [22] 0.3.0 with 1 VCPU and 256MB RAM, with no networking configured. We collected our statistics coarsely, using the "ps" command. We ran our simulated workload for 30 seconds, and collected samples at a 1 second interval. For each experiment, we ran the busy-loop CPU stress-test on each active VM (e.g. for 5 VMs running, all 5 VMs were actively using the CPU). Results represent an average across the 30 second sampling period.

Figure 2 displays the CPU and memory resource utilization of the monitor daemon running in dom0, with respect to the entire system.

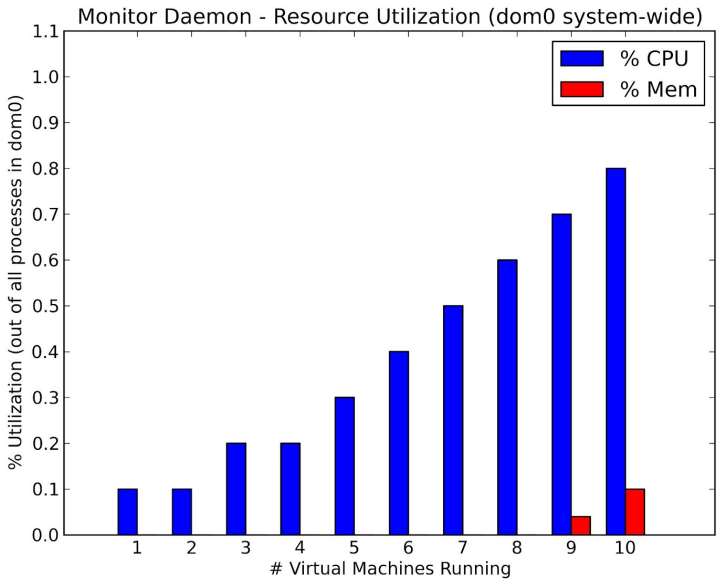

Figure 2: Percentage of Overall System CPU and Memory Utilization by Monitor Daemon Process in Dom0

Our testing demonstrates that CPU utilization of the monitor daemon generally scales linearly with the number of VMs running on a compute node. At first glance, this appears to exhibit relatively poor performance with respect to scalability. However, note that the monitor daemon consumes less than 1% of dom0's overall CPU utilization, even with 10 virtual machines running. The percentage of overall system memory utilized by the monitor daemon does not seem to be significant until up to at least 9 VMs are running, and is still less than 0.2%. During these experiments, the CPU overhead of the collector daemon running on the master node was negligible, less than 0.1% of overall utilization in each case. From this, we conclude that the monitor daemon exhibits relatively low overhead against dom0's overall system resources for typical use-cases, and that the collector is capable of scaling to a reasonable number of VMs with low-impact on performance on the master node.

Additionally, we attempted to measure the monitor daemon process's memory usage. We tracked two metrics:

- the *Resident Set Size (RSS)* - the non-swapped physical memory the process has used
- the *Virtual Memory Size (VSZ)* - the total amount of virtual memory used by the process

When compared to CPU utilization, we found that memory usage is slightly more difficult to accurately determine for a single process. Due to time constraints, we only measured memory utilization using the "ps" command on Linux. While the RSS is a good coarse-grained measurement of overall memory usage for a process, it only accounts for the amount of memory *reserved* by the operating system for the process. Additionally, it potentially includes pages shared between processes (for instance, dynamically-linked libraries). These facts make tracking the RSS and VSZ somewhat inaccurate for our experiments, but nevertheless provide some intuition into how much memory is being used overall by the monitor daemon. Figure 3 shows the RSS and VSZ results collected for the same experiment as above:

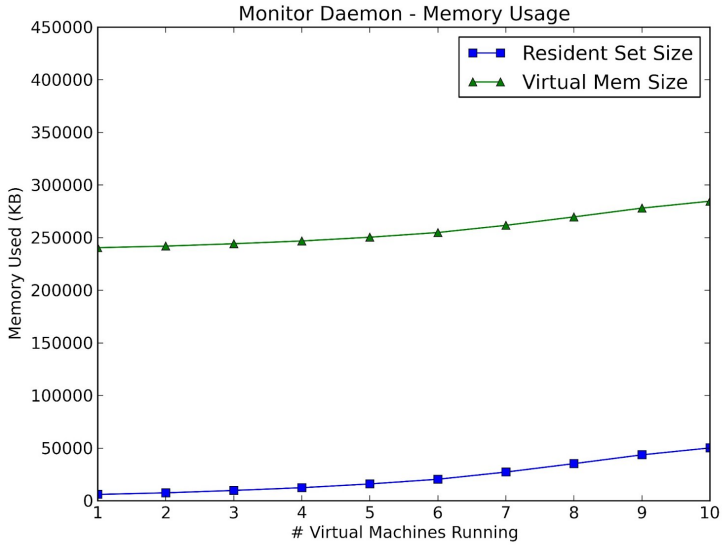

Figure 3: Memory Usage of the Monitor Daemon Process with Respect to Number of Active VMs

We were somewhat surprised by these results. In general, we expected a slight increase in overall memory usage as the number of VMs increased (due to more data collection overall, more objects in memory, etc.), but didn't expect as significant an increase in the RSS as we observed. We believe a number of factors could contribute to this:

- as the number of active VMs increases, the monitor daemon must allocate more `DomainStatTracker` objects to monitor performance data from them. This requires a new libvirt handle for each active domain, which could increase the overall usage. Additionally, as new domains are added, more data is serialized and sent over the network, however we believe this particular aspect to be of minimal impact.
- the inaccuracies of measuring RSS could cause an artificial increase. For instance, adding new VMs (and allocating new libvirt structures) may cause more pages to be reported as shared, which is included in the RSS calculation.
- there could be a memory leak in our code.

Although we found this data somewhat anomalous, we note that overall memory usage is still relatively low for the monitor daemon (less than 50MB for the RSS) even with a large number of VMs running. Again, we conclude that the monitor is relatively memory-efficient with respect to common use-cases for a typical cloud compute node.

We also measured the amount of data transferred on the network between monitor and collector. For this experiment, we measured the bandwidth used by the monitor transmitting performance metrics to a collector. We measured bandwidth using iftop [23], which measured the transmit bandwidth used over a 40 second period (we report the peak for the 40 second window here). We expect linear scale here as well, as the amount of data to be transmitted by a single collector is *O(N)* with respect to the number of VMs running - adding a VM to the host adds a fixed amount of metric data collected from libvirt and included in the serialized message. Figure 4 contains the

results of our testing.

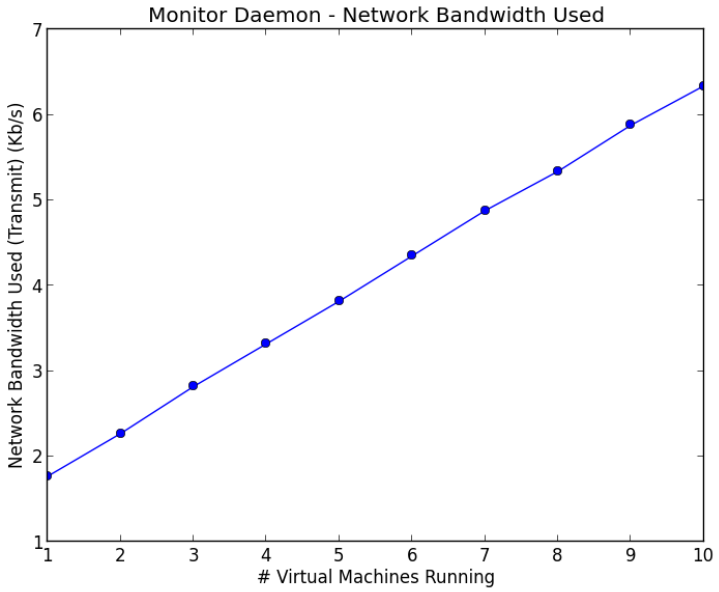

Figure **4**: Network Bandwidth Usage of the Monitor Daemon Process with Respect to Number of Active VMs

For 10 VMs, the bandwidth used by a single monitor is less than 7Kbps. Our program currently collects a small amount of data per VM instance, and this is serialized in a single message sent once per second. We conclude from this that a *single* monitor/collector connection uses a relatively small fraction of network bandwidth. However, a more thorough study of network utilization should be done to characterize network resources used for a large-scale deployment.

4.2 Experiences

To test our monitoring system beyond a single-host development environment, we deployed it on a research cluster running an OpenStack cloud here at Georgia Tech. Our current testbed contains 15 compute nodes running Xen 4.1. We deployed the monitor daemon on each of these systems and configured them to publish data to a master node, running the OpenStack dashboard.

Our research cloud infrastructure is currently in the early stages of development, and is not yet feature-complete. As such, we were only able to demonstrate basic monitoring functionality (vCPU utilization). However, future work should be able to demonstrate monitoring capabilities of more sophisticated VM instances (e.g. network and disk performance), once our OpenStack cluster is in a more stable state.

Figure 5 displays a screenshot of our OpenStack Dashboard application that queries the data aggregation layer to display monitored VM instances. This particular test involves several VMs running under the same tenant (the “openstack” project). Our system isolates monitoring data between tenants, and only displays data for those instances that are active for the currently selected tenant.

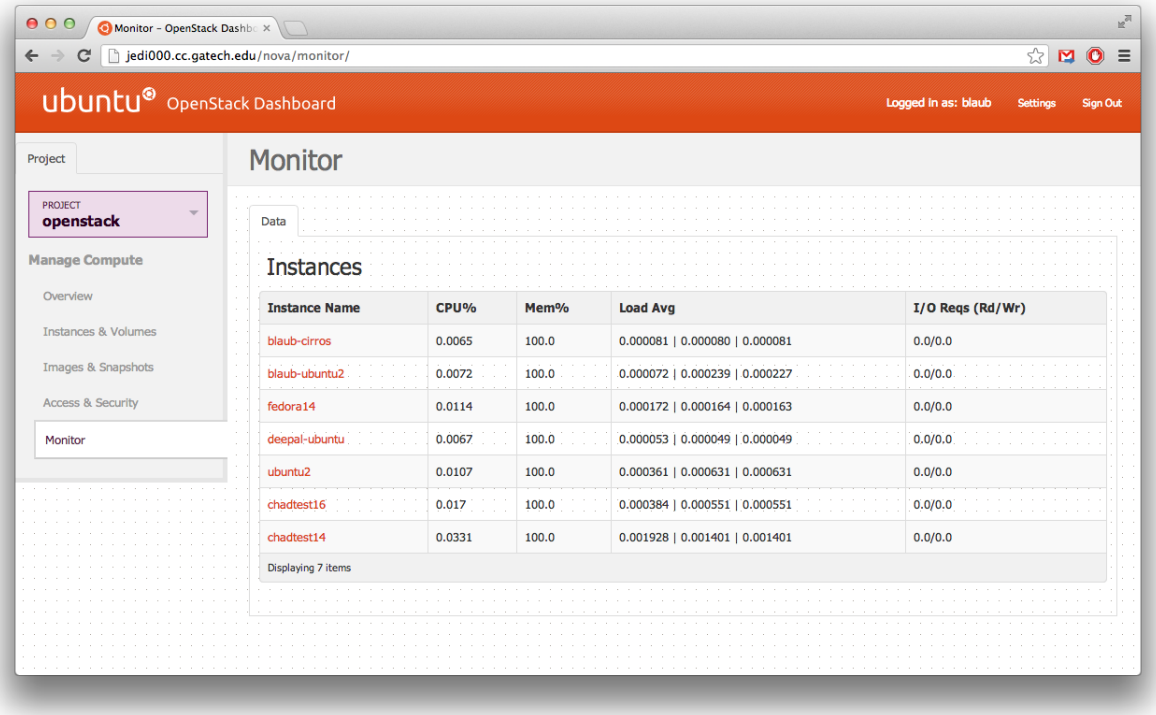

Figure 5: Screenshot of fully-deployed StackMon framework running on an OpenStack research cluster

While our Dashboard plugin provides simple functionality, it could be improved in several ways. Many monitoring systems (for instance, [2] and [27]) provide visualizations of performance data in a graphical format (for example, a graph displaying CPU% over time). We believe this would add real value to an OpenStack deployment using our system. An additional improvement might be to add an alert system to the OpenStack framework based upon our data collection system. This would allow users to be notified automatically (e.g. via e-mail, SMS, etc.) when certain conditions arise, such as an extended CPU spike. Furthermore, this data could be used for some level of automation - for instance, triggering the creation of a new instance to perform load-balancing in response to increased demand, or performing VM migration.

Our experience with the OpenStack platform was somewhat bittersweet. As an open source project, we consider OpenStack to still be relatively immature, comparatively. This follows naturally from the fast-paced movement of cloud computing technologies. We found the OpenStack documentation to be scarce at times, and the tools could often be difficult or confusing to work with. A large portion (larger than expected) of our development time was spent "spinning up" to learn the intricacies of OpenStack. In our opinion, this investment was worth the effort. Although we could have focused more on our monitoring infrastructure's design and implementation, we felt it was important to also apply our system to a full-scale OpenStack deployment to demonstrate its effectiveness and to provide a building block for future research. To this end, we believe we were successful, even if our accomplishments represent only a small step towards larger research goals for cloud computing platforms.

5. Suggestions for Future Work

Our system can be extended in several ways. In the future, we believe scalability will be a key aspect of any monitoring system for a cloud datacenter. Cloud systems incorporate large deployments of physical machines, often across multiple datacenters. The key drawback to our work is the star topology, which can result in a bottleneck. This is almost certainly limited by operating system resources at the master collector, which must maintain a socket connection for each connected publisher. A hierarchical approach could alleviate this. We briefly experimented with this, but more work almost certainly remains. For example, one could envision a system where intermediate collectors are deployed on one server in each physical rack, and a master collector fans-in data from each intermediate collector based on the rack ID. Though we believe a hierarchical approach can improve scalability, many other designs are also possible - for instance, a "push" based design that deploys a graph of monitors as an overlay network to collect only a subset of data at any given type, similar to VScope [3]. Additionally, as previously noted, a communications infrastructure built around EVPath would likely provide even more flexibility than our current implementation.

Future work could additionally focus on improving the reliability of the monitoring system in case of failures. We could potentially have 'heartbeat' messages sent from Collector daemons to Monitors. In the event of an intermediate node failure, it could be detected at the Monitor by the non-reception of heartbeat. The Monitor daemon could then request the Central/Final collector daemon for rerouting of its next hop in the hierarchy.

Our prototype collects data into a round-robin database for simplicity. However, a large-scale deployment would quickly become unmanageable using this method. Future work might examine storing performance data in a large-scale, distributed database system, such as HBase [24] or Cassandra [25]. Distributed databases that scale linearly are ideal for storing large amounts of sensor data collected from monitors, without sacrificing any fidelity. Additionally, a distributed database system might eliminate the star topology bottleneck by allowing localized "clusters" of monitors to publish data directly to the database in batches, without having to go through a central point.

More intricate data analytics are a natural extension to this project as well. Although we have a basic monitoring framework in place, the analysis of the data collected remains a challenging aspect. Cloud platforms often exhibit performance characteristics that can be vastly different from a typical deployment, due to the use of virtualization, multi-tenancy, heterogenous hardware, and other factors. With a monitoring framework in place, and combined with a large-scale distributed database system (discussed above), a data analysis framework would be able to provide real statistical analysis of applications in the cloud.

6. Conclusion

Our project aimed to develop a lightweight performance monitoring framework for use in cloud-style Infrastructure-as-a-Service deployments. We approached this problem by developing a system based upon a publish/subscribe messaging model capable of aggregating data in a clustered computing environment. Our monitor/collector architecture provides a "fan-in" topology capable of centralizing performance data monitored in a virtualized datacenter. We provide a flexible way to collect this data into stable storage, using RRDs as an example. We also demonstrate the effectiveness of our system by building on the OpenStack platform to construct a tool that helps cloud users monitor the performance of their virtual datacenter.

We believe our system provides a useful stepping stone for an array of new work related to this subject. Monitoring systems are extremely important in any clustered computing environment, where applications are often run on hundreds, or even thousands of nodes. Failures are common, and complexities in the datacenter environment can lead to a variety of performance implications. These facts have become increasingly important in cloud computing environments, where issues relating virtualization, datacenter heterogeneity, network performance, and other factors can adversely affect the performance of applications. Though many past projects have explored cluster monitoring, we believe exploring these concepts on the frontier of cloud computing will become increasingly important.

In addition, we believe our project represents a good first step towards adding features to the OpenStack platform. In our experience, OpenStack as a platform is less mature than commercial cloud deployments like Amazon Web Services, but its open nature provides an excellent opportunity for researchers to apply theories to practice. The closed ecosystem of commercial clouds unfortunately offers the research community little help in studying the intricacies underlying large-scale systems design. Our work offers an example of leveraging an open-source platform to help alleviate this problem.

Acknowledgements

The authors would like to thank Chad Huneycutt for assisting with access to cluster resources and for providing invaluable technical support and guidance, and Professor Karsten Schwan for his helpful suggestions and guidance throughout the project.

Notes

The code for this project is available on the web under an MIT open-source license. Please email the authors if you would like access to it.

References

[1] M. Massie, B. Chun, D. Culler. "The Ganglia Distributed Monitoring System: Design, Implementation, and Experience." Pending publication, 2003.
[2] F. Sacerdoti, M. Katz, M. Massie, D. Culler. "Wide area cluster monitoring with Ganglia."

Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on , vol., no., pp. 289- 298, 1-4 Dec. 2003.
[3] C. Wang, I. A. Rayan, G. Eisenhauer, K. Schwan, V. Talwar, M. Wolf, C. Huneycutt. "VScope: Middleware for Troubleshooting Time-Sensitive Data Center Applications." To appear in ACM Middleware 2012.
[4] Amazon Web Services. http://aws.amazon.com.
[5] Windows Azure. http://www.windowsazure.com.
[6] OpenStack. http://www.openstack.org.
[7] C. Wang, K. Schwan, V. Talwar, G. Eisenhauer, L. Hu, and M. Wolf. 2011. "A flexible architecture integrating monitoring and analytics for managing large-scale data centers." In *Proceedings of the 8th ACM international conference on Autonomic computing* (ICAC '11). ACM, New York, NY, USA, 141-150.
[8] T.C. Ferreto, C.A.F. de Rose, L. de Rose. "RVision: An Open and High Configurable Tool for Cluster Monitoring," *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on* , vol., no., pp. 75, 21-24 May 2002.
[9] J. M. Brandt, A. C. Gentile, D. J. Hale, and P. P. Pebay. "OVIS: a tool for intelligent, real-time monitoring of computational clusters," *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* , vol., no., pp.8 pp., 25-29 April 2006.
[10] M. Kutare, G. Eisenhauer, C. Wang, K. Schwan, V. Talwar, and M. Wolf. 2010. "Monalytics: online monitoring and analytics for managing large scale data centers." In *Proceedings of the 7th international conference on Autonomic computing* (ICAC '10). ACM, New York, NY, USA, 141-150.
[11] M. Sottile, R. Minnich. "Supermon: a high-speed cluster monitoring system." *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on* , vol., no., pp. 39- 46, 2002.
[12] EVPath. http://www.cc.gatech.edu/systems/projects/EVPath.
[13] libvirt: The virtualization API. http://libvirt.org.
[14] KVM: Kernel Based Virtual Machine. http://linux-kvm.org.
[15] Whisper. http://graphite.wikidot.com/whisper. (available on the Python Package Index).
[16] Django Web Framework. https://www.djangoproject.com.
[17] ZeroMQ: The Intelligent Transport Layer. http://www.zeromq.org.
[18] kqueue. http://en.wikipedia.org/wiki/Kqueue.
[19] epoll. http://en.wikipedia.org/wiki/Epoll.
[20] MessagePack. http://msgpack.org.
[21] JSON (JavaScript Object Notation). http://www.json.org.
[22] CirrOS: A Tiny Cloud OS. https://launchpad.net/cirros.
[23] iftop. http://www.ex-parrot.com/pdw/iftop.
[24] Apache HBase. http://hbase.apache.org.
[25] Apache Cassandra. http://cassandra.apache.org.
[26] VMWare ESX Hypervisor. http://www.vmware.com.
[27] Amazon CloudWatch. http://aws.amazon.com/cloudwatch.

Appendix: Relation to Concepts Studied in CS6210

Our work draws upon several operating systems concepts studied this semester in CS6210. Perhaps the most prevalent is virtualization. Throughout this project, we were able to gain hands-on experience working with the Xen hypervisor (albeit through the libvirt API). Because our software development took place mostly in Xen's dom0, many of the technical concepts from the Xen paper were apparent to us throughout the project. For instance, in order to better understand how to monitor virtual network interface statistics, we studied the various types of virtualized network configurations offered by Xen, such as bridged versus routed network configurations. All of our testing and evaluation was done using a paravirtualized OS. Since version 3.0, all the required paravirtualization support to run Linux in dom0 or domU is included in the mainline kernel. Early on, the Jedi OpenStack deployment was in a somewhat immature state. During this time, we tested our software using a build of CirrOS for paravirtualized Xen, which required us to build virtual disks and set up a Xen domU configuration from scratch. This experience gave us some extremely important practical experience with the Linux OS and virtualization, that allowed us to connect many of the theories we read about in research papers to actual implementation.

Where possible, we tried to apply many of the basic concepts from papers we studied this semester on operating systems structure. Many filesystem papers we covered, for instance, require basic structures and algorithms to maintain state about files in the system. This in part guided our design for storing RRDs in our system as a hierarchy organized by instance (this is essentially "metadata" in our system). From studying Xen and VMware ESX, we have also learned that efficient resource utilization in the host domain is important in virtualized environments. This guided our decisions at various points in the project - for instance, keeping track of active domains in the system and ensuring that any resources associated with a domain are freed as soon as the instance is terminated.

From our background research, we concluded that scalability is often a very real concern when building systems, even for small applications like the one we developed. Studying systems such as MapReduce, Haystack, Porcupine, and Dynamo have guided our study of scalability in our own system as well. With the rising popularity of cloud computing systems and large-scale datacenters that drive today's modern applications, we believe this will be the most important aspect when considering future work on our project.

Based upon the research projects we studied in class this semester, we also learned that performance evaluation is a critical component of systems design. For this reason, we felt it was important to provide some performance analysis of our work, even if the analysis presented here is slightly short. We tried to apply basic systems performance analysis concepts drawn from many of the papers we studied in class, especially those that evaluated overhead of some software components or system design choices.