UNIVERSITY OF TROMSØ UIT

FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Hubble: a platform for developing apps that manage cloud applications and analyze their performance

Robert Molund Pettersen

INF-3990
Master's Thesis in Computer Science
May, 2012

Abstract

The ability to deliver computing as a metered service has made the cloud an attractive platform for deployment of applications. Using the cloud, enterprises experience a decrease in maintenance overhead, faster deployment, and that cloud elasticity can be exploited to meet fluctuating resource demands.

This thesis presents Hubble, a platform for developing apps that manage cloud applications and analyze their performance. Hubble provides apps with support for persistent storage of performance data, creating secure channels for communication with cloud instrumentation and management software, and interfaces to aid with analytical computations on performance data.

We present and evaluate several apps that have been developed for Hubble. These provide functionality spanning from retrieval of performance data, visualization of performance, and management of cloud services.

Acknowledgments

First and foremost, I want to thank my supervisor Åge Kvalnes for his outstanding expertise and high availability. Whether it was weekend or in the AM, you where always there with an answer and motivation.

Further, I would like to thank the rest of the iAD group for helpful discussions and being there when I needed to rant about subtleties of large frameworks and their vague documentation.

I would also like to thank my parents for providing me with useful resources throughout my academic career, such as food and water.

Contents

List of Figures

List of Tables

List of Acronyms

SVOSH	Secure Vortex Shell	7
SVOCH	Secure Vortex Channel	xi
PerfMon	Performance Monitor	ix
SSH	Secure Shell	27
SFTP	Secure File Transfer Protocol	40
SCP	Secure Copy Protocol	30
RPC	Remote Procedure Call	7
GPB	Google Protocol Buffers	7
LINQ	Language Independent Query	11
API	Application Programming Interface	viii
MCC	Microsoft Chart Controls	45
TPL	Task Parallel Library	40

Chapter 1

Introduction

The ability to deliver computing as a metered service has made the cloud an attractive platform for deployment of applications. With the cloud as a platform, companies and enterprises experience that applications can be deployed faster, manageability improves, maintenance overhead decreases, and that cloud elasticity can be exploited to meet fluctuating and unpredictable resource demands.

The cloud offers resources such as disk, network, and CPU, and while many public cloud providers offer pay-as-you-go computing, varying approaches to infrastructure, virtualization, software services, and pricing models makes it non-trivial to select a provider that fits a particular need.

Li et. al.[9] compare different cloud offerings and their pricing models. From this work, it is evident that a pricing model at one cloud provider could be suited for some need, while the pricing model of another provider better suited for other needs. For example, for a long running computation with non-urgent completion time requirements, fanning out to use Amazon Spot Instances[1] when the price of those resources drop below an acceptable threshold might be desirable to the owner of the computation.

Another example could be choosing between two different CPU offerings. One offers very cheap, but slow, CPU and the other offers very fast, but expensive, CPU. In the process of selecting an offering, pertinent questions are: will the faster CPU result in a commensurate increase in application

[1]http://aws.amazon.com/ec2/spot-instances/

performance? Measured in some application level metric, what is the cost of given service for each of the offerings?

For companies that deploy applications in the cloud, tools and mechanisms to facilitate answers to questions such as these are highly desirable. Currently, a cloud tenant typically has to rely on interfaces that are cloud-specific, if available at all. For example, in Microsoft Azure, custom probing tools need to be employed to gather performance data from the Diagnostics Monitor, whereas Amazon provides performance summaries via tenant account web pages.

In this thesis we present Hubble, a platform for developing apps that manage cloud applications and analyze their performance. The design and implementation of Hubble is the subject of this thesis.

1.1 Problem definition

The goal of this thesis is to design and implement a platform for developing apps that manage cloud applications and analyze their performance. The platform should offer the following functionality:

1. Enable secure communication of performance data produced by cloud instrumentation to an app.
2. Provide efficient mechanisms for storing and performing query-based retrieval of structured performance data.
3. Offer interfaces for aiding apps in performing analytical computations on performance data.
4. Offer interfaces for aiding apps in controlling the cloud environment.

1.2 Scope and Limitations

Usually, an app platform provides means to deploy apps to a central repository, often referred to as an app store, such that users can locate and install selected apps on their instance of the platform. Further, an app platform usually provides functionality for an app to create and manage graphical user interfaces. The existence of this functionality is either to reduce app complexity, or to conserve the aesthetics of apps for the platform.

Due to time limitations, the design and implementation of a graphical user interface and app store for Hubble will not be considered.

1.3 Method and Approach

Three paradigms divide the discipline of computing[3]: (i) *Theory*, (ii) *Abstraction*, and (iii) *Design*.

Theory is based on mathematics and is comprised of four steps for developing a coherent, valid theory:

1. Characterize objects of study (definition)
2. Hypothesize possible relations among them (theorem)
3. Determine whether the relationships are true (proof)
4. Interpret results

The *abstraction* paradigm is an experimental scientific method, and is used to investigate a phenomenon based on the following four steps:

1. Form a hypothesis
2. Construct a model and make a prediction
3. Design an experiment and collect data
4. Analyze results

The *design* paradigm is founded in engineering and follows four steps to form the basis for constructing a system aimed at solving a problem:

1. State requirements
2. State specifications
3. Design and implement the system
4. Test the system

The focus of this thesis will be on the *design* paradigm. First, a definition of the requirements for the system will be outlined. Then the system will be implemented based on these requirements, followed by testing of the systems functionality. This will be repeated till the system behaves satisfactory. Finally an evaluation of whether the system provides the functionality needed to solve the problem will be presented.

1.4 Outline

The rest of this thesis is organized as follows.

Chapter 2 provides background information for understanding the design and implementation, and related work.

Chapter 3 describes the design and implementation of Hubble, an app platform for management and analysis of cloud applications and their performance.

Chapter 4 describes the design and implementation of apps developed for running on top of the Hubble platform.

Chapter 5 evaluates the design and implementation through experiments.

Chapter 6 provides a summary and concludes the thesis.

Chapter 2

Background and Related Work

This chapter outlines key aspects of the Vortex architecture and present related work.

2.1 Vortex Architecture

Vortex is designed to maximize scheduler control over resource consumption. The kernel is structured as a graph of resources, a *resource grid*, where each resource implements common operating system functionality such as a file system, a network protocol, etc. Resources communicate asynchronously through message passing, with each message containing a tag to identify the requesting activity. An activity is typically equivalent to a process. Schedulers control when to dispatch a message to a resource, thereby also controlling when and how a resource is multiplexed among activities.

2.1.1 lVortex kernel

The Vortex kernel has been designed and implemented using three design principles [8]; (i) Measure all resource consumption, (ii) Identify the unit to be scheduled with the unit of attribution and (iii) Employ fine-grained scheduling.

The first principle assures us that all resource consumption in the system is observed and measured. This is important to get a complete view of the resource usage in the system.

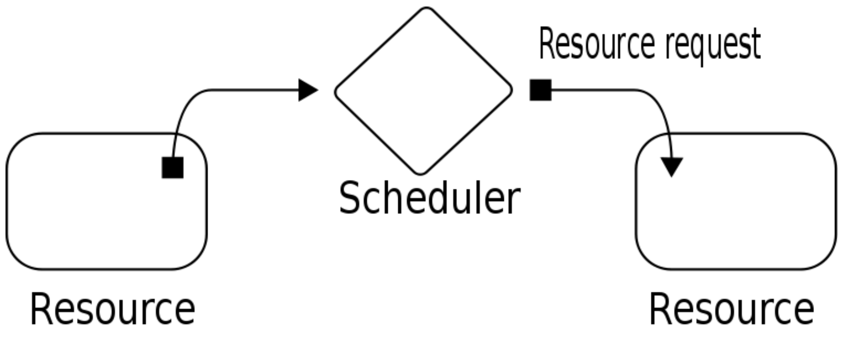


Figure 2.1: Schedulers control when to dispatch resource requests.

This is achieved by making all resource requests be passed as messages between resources. Schedulers are inter-positioned between resources as seen in Figure 2.1 and will be able to measure resource consumption external to the resource when dispatching messages.

The second principle asserts the correctness of performance data, as the resource usage as part of executing a message will be attributed to the activity associated with the message.

The last principle, which is the most significant for this thesis, forces the kernel to employ fine-grained scheduling. This means that the scheduler have complete control over the different resources in the system, by dividing them into many fine-grained resources that can be controlled separately. For instance when accessing the file system, the request could traverse a file block cache, a volume manager, and a device driver resource or a subset of these resources. The scheduler can control requests to the block cache based on memory consumption whereas the amount of data transferred might be a desirable metric at the disk driver level.

2.1.2 Vortex Services

For resource management and security isolation, Vortex defines the concept of a *service*. Services are organized in a strict hierarchical manner.

Resources are allotted to services, which in turn can be utilized by processes and threads. Services can run multiple processes, but their collective resource consumption can never exceed what is available to their governing service. Processes within a service have local autonomy over available resources; they are free to decide on a policy for how resources are shared among themselves, and also to create and delegate resources to sub-services.

The service abstraction provides security isolation by limiting what names and resources are visible and accessible to processes. The root file system of a service must be a strict post-fix of the root of its parent service. Moreover, a process can only see processes attached to its service or a descendant service.

2.2 The Svosh Suite

We developed the Secure Vortex Shell (SVOSH) Suite in previous work [11]. SVOSH offers authentication, encryption, and integrity of messages sent between clients and Vortex. As part of this thesis we have reimplemented large parts of SVOSH. In particular, we have extended SVOSH with support for secure channels and replaced SVOSH's previous I/O subsystem with interfaces to an asynchronous I/O engine that was developed recently. We refer to the new version as Secure Vortex Channel (SVOCH). SVOCH is further described in Section 3.4.

2.3 Protocol Buffers

Protocol Buffers[1] is a language and platform neutral functionality for serializing structured data for use in communications protocols. Google Protocol Buffers (GPB) was developed by Google. In Hubble, we base wire-level representation of performance data on GPB.

Google initially developed GPB to deal with an index server request/response protocol, but GPB is now widely used within Google for storing and interchanging all kinds of structured information. GPB resembles the Apache Thrift protocol used by Facebook. The main difference is that GPB include a Remote Procedure Call (RPC) stack that is used for nearly all inter-machine communication.

Google Protocol Buffers have native language bindings for Java, C++ and Python, but have through third party developers been ported to most other languages and platforms.

[1]http://code.google.com/p/protobuf/

Google Protocol Buffers are not self describing, but utilize indexes on field names to achieve data compression of field separators. Data structures are defined in a separate *.proto* file, and both simple and complex data types are supported, as well as recursive data structures.

Once the data structure has been defined, a language specific compiler will produce simple accessors as well as methods for serializing and parsing the whole data structure to/from raw bytes.

There are a number of advantages over other wire-level protocol formats like XML. Depending on the language and implementation, the raw data produced by protocol buffers is 3 - 10 times smaller in size, and take 20 - 100 times faster to parse.

2.4 Related Work

2.4.1 App platforms

App platforms are emerging in many, often unexpected, areas. A characteristic of these platforms is that they provide rich, domain-specific, APIs for third party developers to create advanced applications with little effort.

Spotify[1] is a music streaming that offers unlimited streaming of millions of tracks through a client application that can be installed on multiple platforms. The application also acts as an app platform where third party developers can create apps that can utilize APIs for searching, organizing, and playing music. Apps can be developed to suggest music based on the users mood, or virtual rooms can be created where users can suggest music to be played for all users participating in the room.

Spotify apps are developed using a combination of HTML5, CSS, and JavaScript. HTML5 canvases are used to construct the user interface, which is styled using CSS. The API is offered through JavaScript modules, which provide functionality to search for music, create collections and play lists, display album art, and so on.

The Spotify API is very restricted when it comes to interacting with music. No external storage is supported, and play lists and collections are stored

[1]http://www.spotify.com

internally by utilizing the strict API. Only the name of the collections and their content can be changed.

Facebook[1] is the worlds largest social networking site, with over 500 million active users posting status updates and sharing pictures from all over the world. Facebook also offers an app platform on which developers can create apps that augment the social API already developed by Facebook. The most prevalent app category on the Facebook app platform is games. This is presumably because of the readily available social API that facilitates social channels that allow users to interact with each other in the games running on the platform.

Facebook apps can be developed in any language that supports web programming, such as PHP, Python, Java or C#. Similar to Spotify apps, Facebook apps utilize HTML5 canvases for graphical user interfaces. The API is written in JavaScript and PHP, and provides functionality to authenticate users, retrieve social graphs, and create new social channels to facilitate communication between users of the apps.

Compared to Spotify, Facebook has a more open API in the area of deployment and storage. Apps can be deployed directly to the Facebook page, or by utilizing an external connection, the app can be deployed at a company web-server. If the app needs storage for various data, such as player scores or app settings, an external storage provider can be utilized at the app developers discretion.

Microsoft offers an app platform for their mobile devices, called Windows Phone, which supports the C#, VB, and XAML programming languages. XAML is used to design the user interface and C# or VB is used to create the app logic. The platform offers an API for connecting to Microsoft services such as Live for game integration, Bing for maps and searches, and hardware devices like GPS and accelerometer for positioning.

The platform offers an isolated storage component for each app, which can be utilized as a database or binary storage. In this respect the platform does not restrict the isolated storage, and the developer can choose to design the storage in a way that is optimal for the app.

The platform does however have restrictions on the graphical user interfaces.

[1]http://www.facebook.com

The graphical user interfaces are restricted to a set of pre-defined buttons and shapes. Even the font is constrained. This might seem like an unreasonable restriction at first, but it ensures that the end-user experience is preserved across different apps.

Several other vendors offer platforms with varying APIs for app development. Android, Google Chrome, and Apple iOS are examples of vendors that offer an app platform. Other vendors are emerging for platforms running on TV's and other peripheral devices.

Similar to these app platforms, Hubble offers a platform for creating domain-specific applications. Hubble positions its storage policy close to the one offered by Spotify. By having a strict storage policy, we can offer a unified interface for retrieving performance data across cloud providers.

To our knowledge, Hubble is the first platform for development of apps that manage cloud applications and analyze their performance.

2.4.2 Systems

Ganglia[10] and HP Cluster Management Utility[7] are two systems designed for cluster monitoring. Both systems collect performance data at the granularity of cluster nodes and rely on low frequency sampling to improve system scalability. Depending on the type of deployed instrumentation, Hubble can be configured to provide functionality similar to Ganglia and HP Cluster Management Utility.

Supermon[14] is similar to Ganglia, but focuses on high frequency sampling, even in the presence of many nodes. To reduce the data volume in deployments with many nodes, Supermon only retrieves performance data pertaining observed entities. For example, Supermon can be configured to only retrieve the available memory for each node in the cluster. In Hubble, the frequency at which performance data samples are collected is programmable. Hubble does not, however, currently support collection of specific performance data entities; upon request, the monitor responds with all entities in a performance data sample. As described in Section 6.3, an interesting extension to Hubble would be for apps to supply an Xquery-like query when requesting performance data from a node. The query could then be evaluated at the node and only matching entities returned to the requesting app.

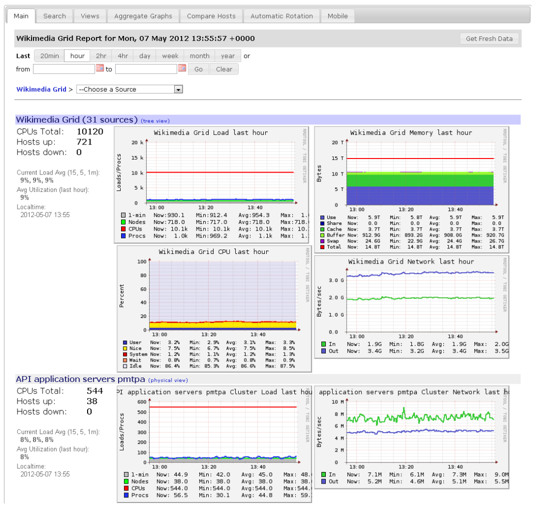

Main
Search
Views
Aggregate Graphs
Compare Hosts
Automatic Rotation
Mobile
Wikimedia Grid Report for Mon, 07 May 2012 13:55:57 +0000
Get Fresh Data
Wikimedia Grid (31 sources)
Wikimedia Grid Load last hour
Wikimedia Grid Memory last hour
Wikimedia Grid CPU last hour
Wikimedia Grid Network last hour
API application servers pmtpa

Figure 2.2: Ganglia illustration from monitoring the WikiMedia Foundation cluster.

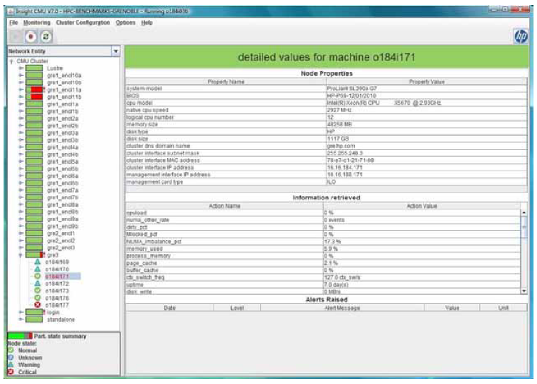


Figure 2.3: HP Cluster Management utility, illustration from [7].

Otus[12] is similar to Ganglia, but samples data at process-level. Its goal is to provide detailed post-analysis charts, not real-time analysis. Hubble allows for analysis of a running system.

Fay[5] and DTrace[1] are two powerful platforms for gathering and analyzing software execution traces used to diagnose system behavior on both single machines and on clusters. Both frameworks introduce the notion of a probe that can be inserted into applications or kernels to extract performance data. This work is complementary to Hubble, as both a Fay and DTrace probe could be used as instrumentation techniques. In particular, Fay offers functionality that could work as drop-in replacements for several Hubble components. For example, in addition to technology for safely inserting probes into a kernel or process address space, Fay provides support for evaluating queries written in a form of Language Independent Query (LINQ). These queries can specify that performance data is to be collected from one or more machines and also how to aggregate and combine the collected data (in an efficient and distributed way). Fay could for example be used in Hubble.control (see Section 3.5.1) as a replacement for communication with the Hubble monitor.

Astrolabe[16] is a information management service, which monitors the performance of a collection of distributed resources, reporting summaries back to the user. The summaries are calculated on-the-fly using an aggregation approach that is intended to bound the rate of information flow at each participating node. In contrast, Hubble is designed to retrieve performance data at the lowest level possible, and instead offer aggregation as a post-processing option.

VMware VFabric Application Performance Manager[17] and IBM Tivoli Monitoring[15] are two enterprise monitoring systems designed to monitor existing enterprise cloud solutions from VMware, XEN, and KVM among others. While these systems usually are relatively expensive, and rely on specialized infrastructure to be able to retrieve useful performance data, our system aims to be generally applicable to all types of cloud infrastructure.

2.4.3 Visualization

Visualization systems come in many forms, ranging from textual representation to abstract graphical representation and the more common chart representations.

The Unix *top* process performance visualizer is one of the most used visualization systems on Unix systems, and comes bundled with most Linux/Unix systems. *Top* visualizes each process by a line of text that dynamically changes based on the load in the process, as seen in Figure 2.4.

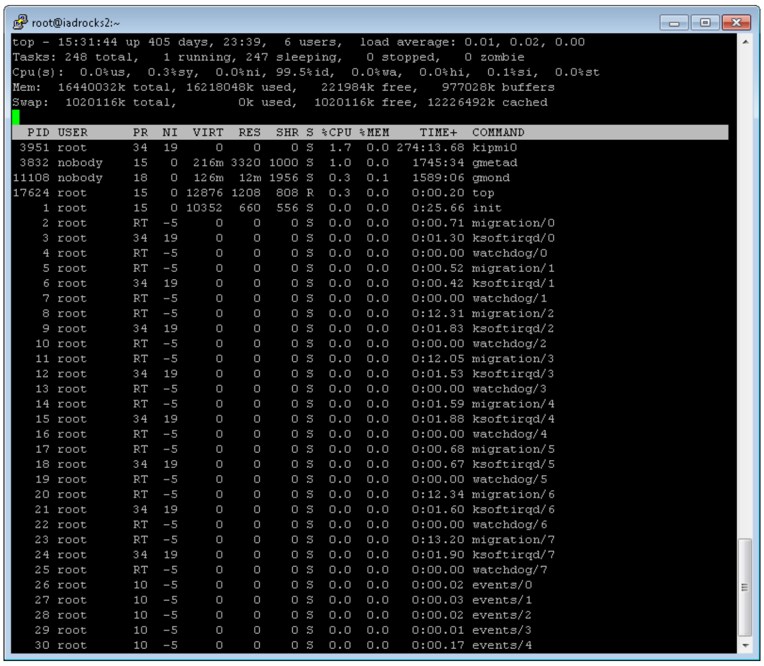
LavaPS[1] and PSDoom[2] are two quite different abstract visualization system that each have their unique way of representing performance statistics.

LavaPS disguises it self as a lava lamp, with colored blobs representing processes running on the monitored system. The blobs move faster the more CPU usage the process has, and grows larger the more memory the process consumes, as illustrated in Figure 2.5.

PSDoom on the other hand, simulates the 3D shoot-em-up game Doom, where processes are represented as monsters, and gives a more interactive representation of the monitored system than LavaPS. A user can get an overview of the load of the system by looking at how crowded the different rooms are. An illustration is seen in Figure 2.6.

Both LavaPS and PSDoom are visualization techniques that can capture trends and the big picture of running systems. But mining macro-level

[1]http://www.isi.edu/ johnh/software/lavaps/



```
root@iadrocks2:~
top - 15:31:44 up 405 days, 23:39,  6 users,  load average: 0.01, 0.02, 0.00
Tasks: 248 total,   1 running, 247 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,  0.3%sy,  0.0%ni, 99.5%id,  0.0%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:  16440032k total, 16218048k used,   221984k free,   977028k buffers
Swap:  1020116k total,        0k used,  1020116k free, 12226492k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3951 root      34  19     0    0    0 S  1.7  0.0 274:13.68 kipmi0
 3832 nobody    15   0  216m 3320 1000 S  1.0  0.0   1745:34 gmetad
11108 nobody    18   0  126m  12m 1956 S  0.3  0.1   1589:06 gmond
17624 root      15   0 12876 1208  808 R  0.3  0.0   0:00.20 top
    1 root      15   0 10352  660  556 S  0.0  0.0   0:25.66 init
    2 root      RT  -5     0    0    0 S  0.0  0.0   0:00.71 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:01.30 ksoftirqd/0
    4 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/0
    5 root      RT  -5     0    0    0 S  0.0  0.0   0:00.52 migration/1
    6 root      34  19     0    0    0 S  0.0  0.0   0:00.42 ksoftirqd/1
    7 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/1
    8 root      RT  -5     0    0    0 S  0.0  0.0   0:12.31 migration/2
    9 root      34  19     0    0    0 S  0.0  0.0   0:01.83 ksoftirqd/2
   10 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/2
   11 root      RT  -5     0    0    0 S  0.0  0.0   0:12.05 migration/3
   12 root      34  19     0    0    0 S  0.0  0.0   0:01.53 ksoftirqd/3
   13 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/3
   14 root      RT  -5     0    0    0 S  0.0  0.0   0:01.59 migration/4
   15 root      34  19     0    0    0 S  0.0  0.0   0:01.88 ksoftirqd/4
   16 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/4
   17 root      RT  -5     0    0    0 S  0.0  0.0   0:00.68 migration/5
   18 root      34  19     0    0    0 S  0.0  0.0   0:00.67 ksoftirqd/5
   19 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/5
   20 root      RT  -5     0    0    0 S  0.0  0.0   0:12.34 migration/6
   21 root      34  19     0    0    0 S  0.0  0.0   0:01.60 ksoftirqd/6
   22 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/6
   23 root      RT  -5     0    0    0 S  0.0  0.0   0:13.20 migration/7
   24 root      34  19     0    0    0 S  0.0  0.0   0:01.90 ksoftirqd/7
   25 root      RT  -5     0    0    0 S  0.0  0.0   0:00.00 watchdog/7
   26 root      10  -5     0    0    0 S  0.0  0.0   0:00.02 events/0
   27 root      10  -5     0    0    0 S  0.0  0.0   0:00.03 events/1
   28 root      10  -5     0    0    0 S  0.0  0.0   0:00.02 events/2
   29 root      10  -5     0    0    0 S  0.0  0.0   0:00.01 events/3
   30 root      10  -5     0    0    0 S  0.0  0.0   0:00.17 events/4
```

Figure 2.4: Example from the Unix *top* visualization. Each line represent a process, stating the current load.

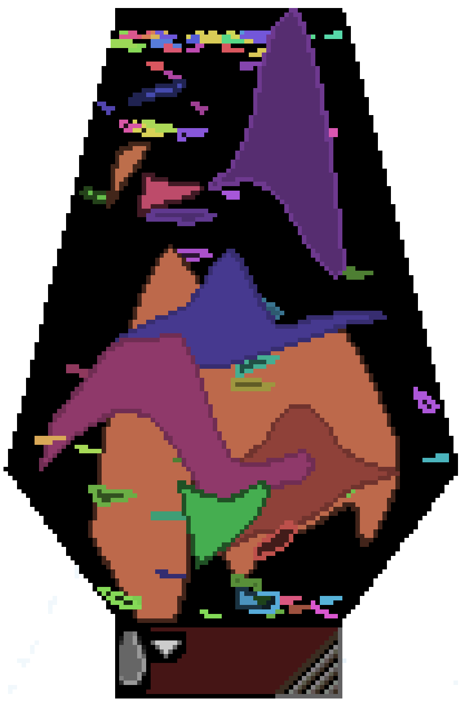


Figure 2.5: Illustration from the abstract Lava PS visualization system. Image from http://www.isi.edu/ johnh/software/lavaps/.

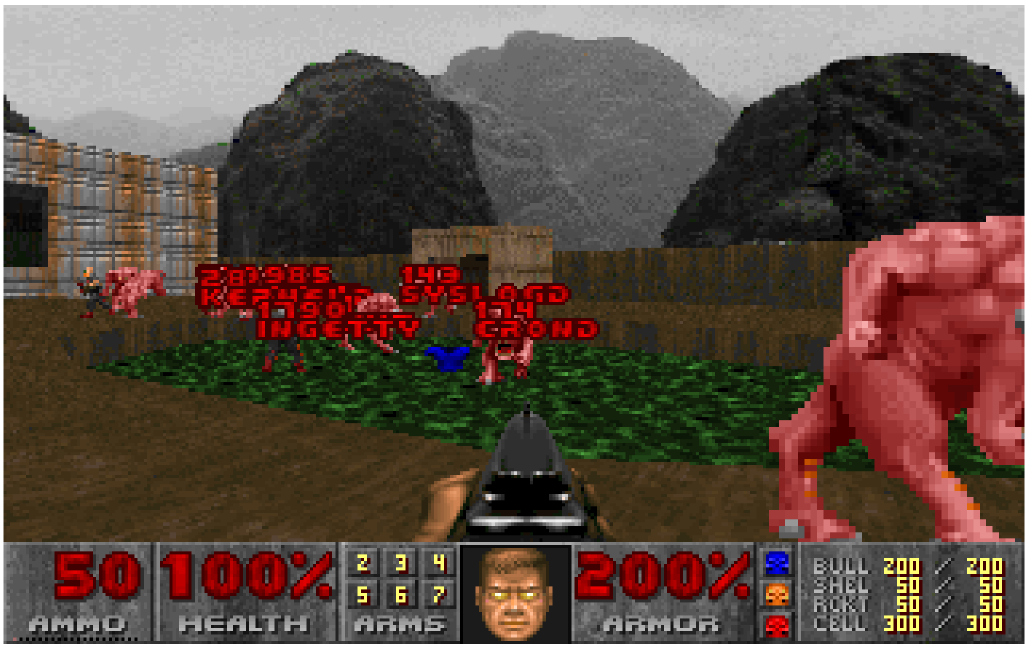


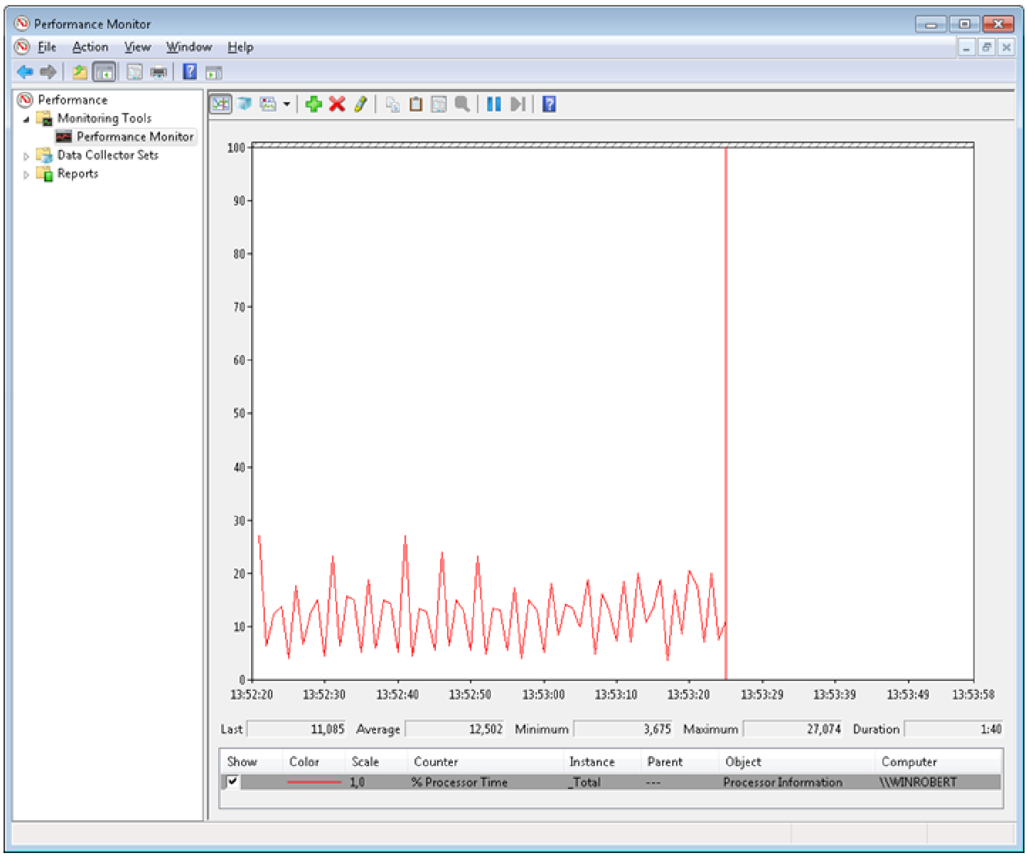
Figure 2.6: Screenshot from the game inspired PSDoom visualization system. Illustration from http://psdoom.sourceforge.net.

information from these techniques can prove difficult. Hubble tries to visualize performance characteristics at the lowest level available, and give valuable information about other parts of the system as well as those parts being visualized.

Microsoft Performance Monitor (PerfMon) is one of the more traditional visualization frameworks that utilizes line plots. PerfMon is installed on most Windows distributions, and has a wide array of pre-defined sources of data to visualize, from CPU load to memory utilization of different parts of the system. An illustration of the PerfMon visualization system is seen on Figure 2.7.

PerfMon can also be configured to connect to a remote host, and visualize performance characteristics from that host instead of the local host.

While PerfMon provides visualization of the performance characteristics of the monitored system, it is not very flexible when it comes to navigating the different components of the system. Hubble tries to visualize the performance data in a intuitive way, and at the same time give the user the option to navigate all components, all the way down to a macro-level



Last 11,085 Average 12,502 Minimum 3,675 Maximum 27,074 Duration 1:40

Show	Color	Scale	Counter	Instance	Parent	Object	Computer
✓	🟥	1,0	% Processor Time	_Total	---	Processor Information	\\WINROBERT

Figure 2.7: Illustration of Performance Monitor (PerfMon), visualizing the Cpu load of the local computer.

so that every angle of a process can be thoroughly investigated.

Chapter 3

Hubble Design and Implementation

This chapter describes Hubble, a platform for apps that manage cloud applications and analyze their performance

3.1 Architecture

Figure 3.1 depicts the Hubble architecture. Hubble consists of three main components: (i) a client-side app platform and API, (ii) a performance monitor residing in the cloud, and (iii) a storage database that can reside either in the cloud or at the client side.

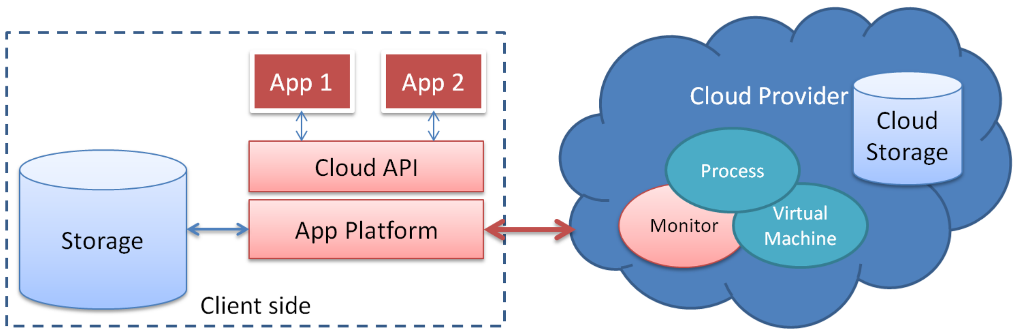


Figure 3.1: Overview of the Hubble architecture.

The app platform provides an API that can be used to develop apps for administration and analytics of cloud services and virtual machines. The app platform provides a portal for connecting to the cloud and controlling applications running in the cloud.

The Hubble platform provides an API for retrieving and storing performance data in the database, securing communication with processes in the cloud, and an analytical interface for performing analysis on the performance data.

The Hubble platform is implemented in the .NET architecture and as such supports app implementations in a wide array of languages, including C#, VB and D#. These languages have libraries that can facilitate graphical user interfaces for apps running on the platform.

The monitor residing in the cloud gathers performance data from multiple sources, and can take advantage of powerful probing frameworks like Fay[5] or DTrace[1] for providing performance data. The monitor gathers performance data, but actual retrieval of the performance data is performed by an app.

3.2 Performance data

Different clouds may provide different opportunities for deploying instrumentation that collects performance data. For example, Microsoft Azure does not provide built-in performance monitoring interfaces, instead custom probing tools need to be employed to gather performance data from the Diagnostics Monitor. In contrast, a Vortex cloud can provide detailed performance data about how different operating system resources are utilized. Also, a cloud deployment may involve simple single-process applications, or applications that consist of multiple processes that span multiple virtual machines.

The disparity in what type of instrumentation may be possible, in combination with potentially complex application deployments, led us to define a common model and format for performance data. All instrumentation must provide performance data that adhere to this model. Similarly, apps can assume that any performance data is structured according to the model. The common data model does as such facilitate and promote creation of apps that are portable across cloud platforms and different operating systems.

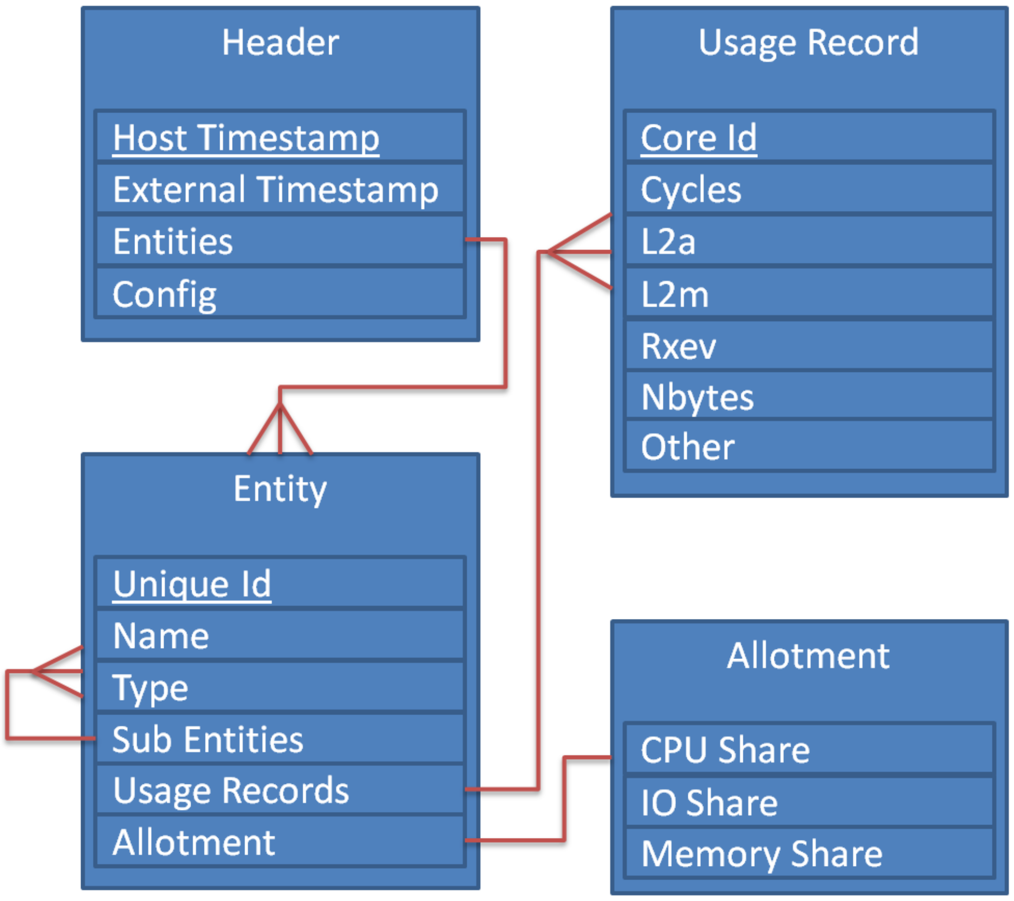


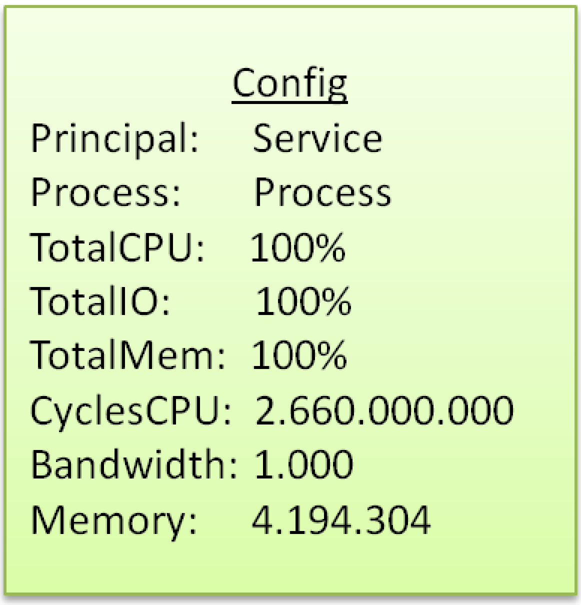
Figure 3.2: Relational diagram illustrating structure of the data format.

The data model is recursive and designed around the notion of *entities* that can describe both resource usage and resource allotment.

Figure 3.2 illustrates the data model. The header contains the time at which the performance data sample was constructed by instrumentation, along with an optional repeatable configuration field.

Each sample contains two different timestamps, the *host* and the *external* timestamps. The host timestamp is set by instrumentation and is expected to be of high accuracy with respect to the ordering of events internally in the host. For example, instrumentation could use the Cpu timestamp counter register on x86-based architectures to provide cycle-accurate timestamps.

The external timestamp is set by the monitor and must be drawn from a real time clock source. Typically, the monitor would use an Ntp-derived clock as



Config

Principal:	Service
Process:	Process
TotalCPU:	100%
TotalIO:	100%
TotalMem:	100%
CyclesCPU:	2.660.000.000
Bandwidth:	1.000
Memory:	4.194.304

Figure 3.3: A sample configuration that could be sent from a monitor, describes the total number of shares and resources at the remote host.

a source for the external timestamp. By using the host timestamp an app can make strong assumptions about the time between samples originating from the same host. For example, if the host timestamp indicates that a sample was produced 20000 microseconds after another sample, the app can assume that this is correct. By using the external clock timestamp, an app can correlate samples originating from different hosts. Here, the external timestamp clock source limits accuracy. Typically, an Ntp-derived clock can be expected to be accurate within a few milliseconds.

The configuration field can contain translations for entity names, for human readability, or other static data like amount of resources available, speed of network interfaces, and the like. The configuration is usually only sent once, or upon request by an app, to reduce the amount of data communicated. A sample can be seen in Figure 3.3. The sample describes a system with a 2.6Ghz cpu, with 4Gb of ram and a 1Gb nic. The host uses a percentage distribution when distributing resource allotments.

The header also contains an optional repeatable field for entities, which in turn can contain an optional number of sub entities. All fields are made optional to promote creation of apps that are robust to situations where a cloud cannot provide a certain type of performance data.

Entities may optionally contain a number of usage and allotment records. These are optional since some entities may serve as organizational entities, like process groups or services.

Each entity must also specify an identifier that is unique to the host from which the performance data originates. The use of unique identifiers enables apps to reason about changes to the cloud environment. For example, if an identifier is present in one performance data sample but not in a (time-wise) later sample, an app can assume that the entity has been removed from the cloud environment since it is not consuming CPU, I/O, or memory.

The type field describes the type of entity and corresponds to a defined enumeration of standard entities found in the cloud, including computer, principal, process, thread, CPU, memory, etc.

The usage records contains fields for number of cycles, cache accesses and misses, number of bytes transferred, and a separate field for application specific statistics. The application-specific field can for example be number of clients served, number of disk accesses, or other metrics.

We have chosen to implement this data model in Google Protocol Buffers (GPB), as GPB provide a platform independent format for reading and writing serialized and compressed data in an efficient way. GPB have native language bindings for Java, C++ and Python, but through third party developers, have been ported to most other languages and platforms.

There are a number of advantages of GPB over other wire-level protocol formats. Compared to XML, depending on the language and implementation, the raw data produced by GPB is 3 - 10 times smaller in size, and 20 - 100 times faster to parse.

To support GPB on the Vortex platform we ported the C implementation provided by a third party developer[1] (see Section 3.3.1 for more information).

3.3 Cloud monitor

Different clouds might offer different opportunities for instrumentation. For example, the Amazon Elastic Compute Cloud allows a tenant access to

[1] http://code.google.com/p/protobuf-c/

and control over its environment at the level of processes and the virtual machine kernel, but performance data from the hypervisor is restricted to summaries provided by Amazon via tenant account web pages. In such an environment, the monitor can deploy instrumentation that access common kernel interfaces, such as the Linux */proc* interface, or rely on more invasive instrumentation such as Fay[5], DTrace[1], or other probing frameworks.

The inability to deploy instrumentation at any level in the cloud infrastructure implies that Hubble can make few assumptions about exactly what performance data can be gathered by the monitor. Moreover, differences in cloud environments imply that instrumentation code must be crafted specifically for a given cloud. These restrictions led us to place few requirements on the monitor, with respect to functionality and interfaces.

The monitor resides in the cloud and must provide an interface for Hubble apps to connect and collect the gathered performance data, and the monitor must provide a control interface whereby it can be configured to operate in pull or push mode. The control interface is also required to respond to capability requests, which will reveal which configuration options are available.

In pull mode, the monitor must provide performance data upon a request, and in push mode, the monitor must obtain performance data from its instrumentation code at specified time intervals and communicate this to the requester. Other configuration options may include a scope which the monitor is limiting the gathering of performance data to.

Beyond this, the monitor is required to obey the formating on the provided performance data as described earlier in Section 3.2. The monitor is also required to start the performance sample with a node entity that describes the current host. This is to be able to separate performance data from different clouds. Ensuring that performance data is securely communicated is handled by Hubble (see Section 3.4).

In the following we describe two monitor implementations. One for a cloud based on the Vortex system, and a second for a cloud where the tenant environment is based on the Linux/BSD operating system.

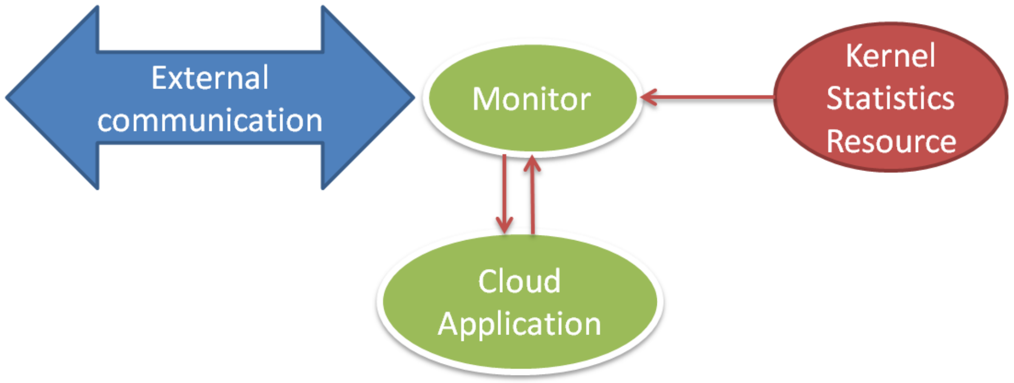


Figure 3.4: Overview of the architecture of the Vortex monitor. The monitor is designed as a process which interfaces with the kernel statistics resource to retrieve performance data. User level applications can be instrumented through separate channels.

3.3.1 Vortex Monitor

The Vortex monitor is the monitor implementation that has received most attention in this thesis. The implementation makes use of instrumentation code placed in the Vortex kernel, which extracts the same performance data as used by kernel-side schedulers.

The Vortex monitor is a user-level process that implements the interface required for apps to request performance data. Similar to a Unix system, Vortex processes operate with input and output channels. The monitor assumes that these are secure communication channels connected to Hubble (see Section 3.4). Upon startup, the monitor takes control over process input and output and then waits for incoming requests.

The monitor is structured around a request queue where incoming app requests are placed. A request can be of type pull or push. When receiving a pull mode request, the monitor responds with performance data. Upon receiving a push mode request, the monitor sets up a timer that, upon expiration, inserts a pull request into the monitor request queue, causing the monitor to respond as if it had received a pull request.

Vortex is structured around services as an organizational unit, that can be alloted cpu, i/o and memory resources. These resources can be utilized by processes running under the service or further delegated to sub services. The processes utilize the resources through aggregates for the different resources.

For example, a process that needs CPU cycles would request these by resource clients associated with a CPU aggregate for that process.

To handle a request for a performance data sample, the monitor first performs a Vortex system call to open the kernel statistics interface[1]. This call returns a Vortex resource identifier that subsequently can be read from to retrieve a performance data sample.

As part of our work we have replaced the existing XML-based Vortex statistics interface with an interface that returns performance data in the Google Protocol Buffers (GPB) format. This entailed porting a C-based GPB implementation by a third party developer[2] to operate within the Vortex kernel environment.

As part of the porting, we made some optimizations to the original GPB implementation. In particular, the GPB implementation relied a two-phase construction of serialized data, whereby the data is first constructed as a graph using dynamic allocation of memory to represent nodes, followed by a graph traversal to produce a serialized byte-array representation.

To improve performance and reduce memory requirements, the implementation was modified to allocate memory for graph nodes from a pre-allocated array. This was possible since construction of the performance data sample only entails adding new nodes to the graph, not modifying or removing existing nodes. Thus, a series of expensive *malloc()* calls could be satisfied by code that used a simple counter to keep track of the next byte of free array memory. A side-effect of this scheme is better cache locality, since nodes are placed sequentially in memory and traversal can be expected to touch fewer cache lines.

Also, the kernel-side logic for traversing Vortex data structures to retrieve performance data has been re-implemented. The logic is based on a depth-first traversal algorithm with respect to Vortex services. The logic starts by creating an entity that represent the current computer node and then proceeds to traverse the services the authenticated user have access to. At each service, all processes and their associated I/O, memory and CPU aggregates are recorded, as explained above. Further, each of the aggregates

[1]This interface can be used by a process to retrieve performance data on itself or other processes.

[2]http://code.google.com/p/protobuf-c/

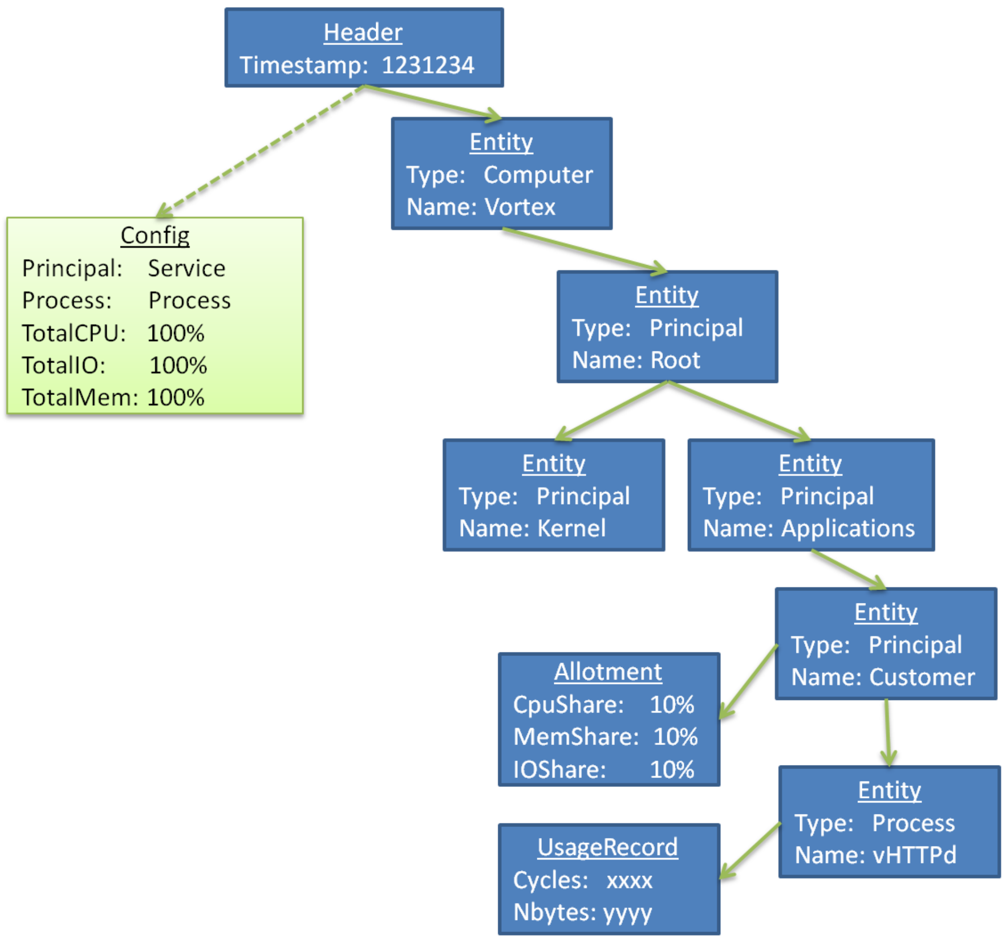


Figure 3.5: Simplified example result of statistical reading from the Vortex kernel. The customer entity has been allotted 10% of the available resources. The optional configuration section is colored green.

are descended into, and their specific performance data is recorded.

A performance data sample is further augmented by the monitor with configuration entities. As an example, consider Figure 3.5, which shows Vortex definitions for entities as well as information such as the total amount of resources available for a customer entity.

The information in the configuration section is typically static, and as such is only supplied by the monitor in the first performance data sample. Though, an app can request the information by setting a flag in a pull request.

For some of our experiments we needed metrics that typically would entail instrumentation of the application. In particular, we needed access to the number of accepted clients to a web server. While this could have been obtained by straightforward instrumentation of the web server, we extended the Vortex kernel instrumentation code to collect the number of accepts on open listen sockets.

A general interface for instrumentation to communicate with the monitor has been implemented, but it is not used in any of the experiments presented in this thesis.

3.3.2 Linux/BSD Monitor

The Linux/BSD monitor is a user level implementation that makes use of existing user level tools to gather performance statistics about processes belonging to the authenticated user. The monitor was implemented as a proof of concept that the solution is extensible to multiple platforms.

The monitor was implemented in roughly 100 lines of python, and consists of a set of functions that wraps Linux/BSD command line tools to retrieve performance data samples. The outline of the monitor architecture can be seen on Figure 3.6.

Google Protocol Buffers (GPB) have native support for Python, and can create language bindings to our entity data format without having to port a special preprocessor.

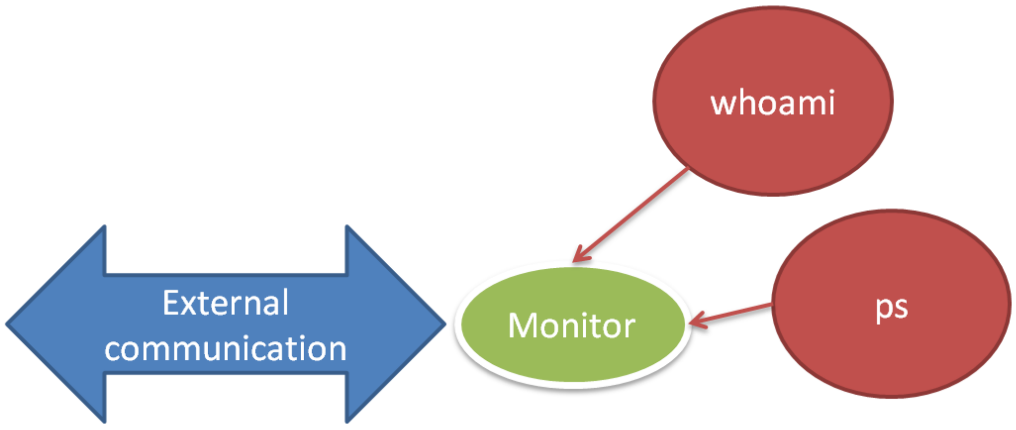


Figure 3.6: The Linux/BSD monitor utilizes the *whoami* and *ps* command line tools to retrieve performance data from running processes.

Upon request from the client, the python script starts by creating an entity representing the current host. Further, the script gets the current user through *whoami*, and adds this entity as a principal entity.

After adding the principal entity, the script utilizes the *ps* command line tool for getting information about the users current running processes and associated performance data.

The output from the *ps* tool is parsed to retrieve both the current CPU and memory usage. As the performance information obtained through the tool is normalized to a percentage of the total available resources, the total amount of shares is set to 100% in the configuration field.

After all running processes have been added, the script serializes the entities using the native language binding made available through GPB native Python preprocessor. The result is returned to the requester.

3.4 Secure client/cloud communication

A performance data sample can reveal information that is potentially sensitive. For example, performance data have been used as a source of information for malicious attackers[13]. Moreover, instrumentation must be considered trusted since the code has full access to a tenant's cloud environment. Thus, authentication, integrity, and confidentiality must be ensured for communication between client-side and the cloud.

For secure communication Hubble relies on use of the Secure Shell (SSH) protocol[18]. SSH uses public-key cryptography for authentication and offers encryption mechanisms to ensure communication integrity and confidentiality. The SSH protocol is supported on most cloud platforms, either as a native maintenance entrance for a tenant, or as a service that can be launched inside a tenant's virtual machine environment.

Another facet of the SSH protocol is that is designed to allow multiplexing of several logical *channels* over a single SSH connection. By creating a channel, a separate communication channel can be established between a client- and server-side application. For example, the protocol defines well-known channel types for shell access and file transfers. The ability to perform file transfers can for example be used to deploy monitor code, should the monitor be unavailable at the cloud host.

The SSH protocol is also designed to allow creation of custom channels, as defined in RFC 4254[1]. Hubble exploits this feature to create separate communication channels between apps and the cloud.

In previous work we implemented a SSH server for Vortex[11]. This work was heavily modified and extended for the work presented in this thesis. In the following we present the current design and implementation of the Vortex SSH server.

3.4.1 Secure Channels

Previously we have implemented Secure Vortex Shell (SVOSH) [11] that provides users with shell access over an SSH connection. We have reimplemented this work to allow custom sub-systems to attach to the secure channels. The resulting authentication and encryption engine was named Secure Vortex Channel (SVOCH) as it provide secure channels, not only shell communication.

During the rewrite process, several bugs were also uncovered and corrected. Some of these bugs were related to the buffer management that would allow a sliding window protocol for each of the separate secure channels. In the previous implementation, multiplexing several channels was not explored and the bug was undetected. But as more channels where multiplexed at the same time, the buffer overflow bug surfaced.

As the Vortex operating system have evolved since the first implementation of the SVOSH, the implementation was further rewritten to take advantage of the fully asynchronous communication engine[2], the AIO-engine. The AIO-engine resides in user space, and uses a thread-pool to efficiently take care of I/O operations and their continuations.

The AIO-engine also maintains a cache of recently used components used when performing I/O communication in Vortex, such as IOStreams and flows. This reduces the latency when creating new channels and improves performance.

Further the AIO-engine exposes a notion of AIO-channels that are used for asynchronous communication. These channels are used to setup communication in the engine.

[1]http://www.ietf.org/rfc/rfc4254.txt
[2]Ongoing work, not yet published

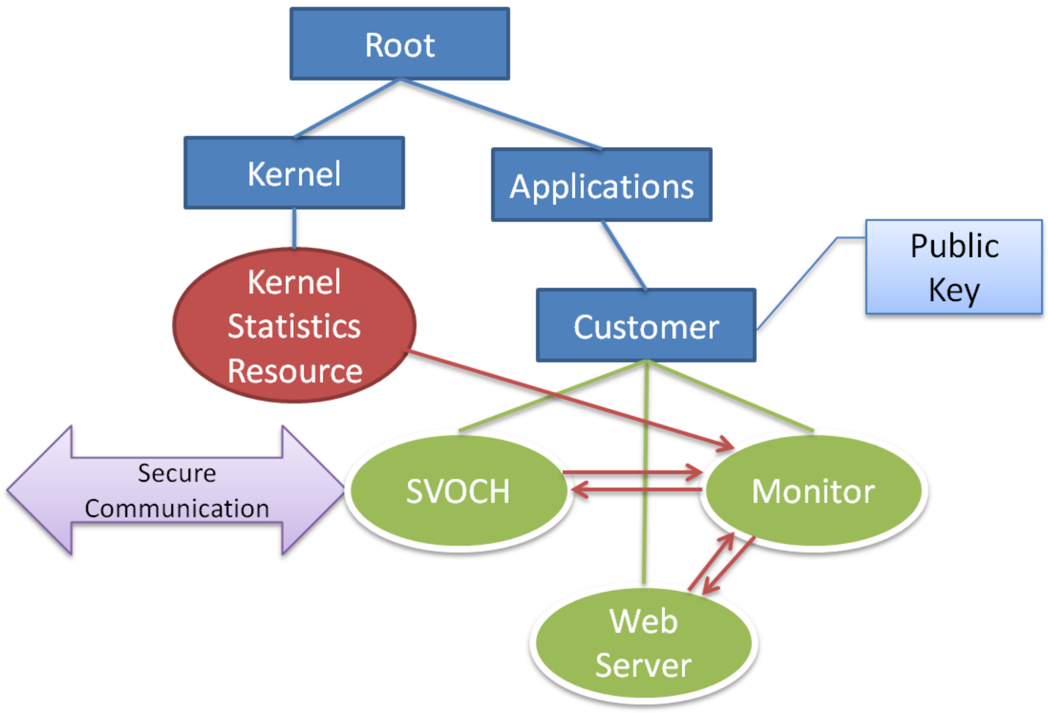


Figure 3.7: Overview of process communication in Vortex. The blue boxes are Vortex services, green circles are processes and red circles are kernel resources. The red lines illustrates AIO-channels.

Each service can have multiple public keys associated with them that can be used for authentication. Figure 3.7 illustrates an example setup where the customer service has one associated public key.

The client authenticates with the service that is to be monitored using his private key, and establishes a secure SSH channel to an instance of SVOCH.

SVOCH sets up the necessary AIO-channels and multiplexes the incoming data to the destination process. Figure 3.7 illustrates an example where the monitor process has been attached to a duplex channel for requests and delivery of performance data.

The monitor further uses a read channel from the kernel statistics resource for reading performance data, and a read-write channel to a web server process for requesting and reading application level performance data.

SVOCH has further been extended with capabilities for dynamically changing the alloted resources of processes and services running on Vortex.

This capability is implemented in the form of a user level process that accepts as arguments a service identifier and a specification of resource allotments. When launched, the process performs a series of system calls to set resource allotment according to what is requested.

Finally the Secure Copy Protocol (SCP) protocol has been implemented in SVOCH to facilitate secure copying of files and folders. The SCP functionality is implemented through a user level process that translates SCP protocol messages to the creation of files and folders. Contents of files are delivered as binary data through SVOCH.

3.5 App platform API

This section introduces the API that the Hubble platform provides for developing apps. The API provides functionality for creating secure channels to the cloud, storing and retrieving performance data to the database, and an analytical interface to aid an app in common analytical tasks.

An overview of the API provided by the Hubble app platform can be seen in Table 3.1. *hubble.control* provides an interface for setting up secure channels to the cloud and performing common operations such as starting processes, images, and adjusting resource allotments. Since different cloud providers have different APIs for connecting and managing resources, for each provider there needs to be a separate implementation of this interface.

hubble.storage provides an interface to the database, offering functionality both for inserting new performance data, and to retrieve different aggregated statistics. One single implementation of the interface is sufficient across cloud providers since the internal database interface is the same.

hubble.analyze provides tools to analyze and perform normalization on the

Hubble App Platform API

Interface	Functionality provided
hubble.control	Control Interface
hubble.storage	Storage Interface
hubble.analyze	Analytical Interface

Table 3.1: The Hubble app platform API.

hubble.control interface

Interface	Functionality
newChannel	Create new channels
newEnvironment	Create new environment
adjustAllotment	Adjust allotments

Table 3.2: The Hubble control interface.

data retrieved from the database. Some of the functions made available in this interface need cloud specific implementations, while other generic functionality is available across cloud providers.

3.5.1 Hubble Control Interface

hubble.control is the interface that provides functionality for setting up secure channels to the cloud and performing common management tasks.

Table 3.2 provides an overview of the control interface functionality.

newChannel creates a new secure channel, and takes two parameters. The first parameter describes the SSH channel type, and is typically one of shell or execute, but custom channel types are also supported. The second parameter describes the remote process that is to be attached at the remote end of the secure channel.

The implementation of this function revolves around SSH protocol messages for setting up a new secure channel in the existing SSH connection. The returned identifier can be used to communicate with the remote process or read exit status messages.

newEnvironment instantiates a new environment in the cloud, and requires two parameters. The first parameter describes the image that is to be started, along with information about where to find the image and whether the image has to be transfered to the cloud before executing it. The second parameter describes the alloted resources the new virtual machine should be alloted.

adjustAllotment adjusts the alloted resources for an already started cloud environment, and requires one parameter that describes the new resource allotment.

The implementation of these functions are highly dependent on the available API at the cloud provider. In our Vortex implementation of the interface, the *newEnvironment* translates to the creation of a new service with the given allotment and a single process running as specified by the image. In Amazon EC2, this would translate to the creation of a new virtual machine.

adjustAllotment is implemented using the API developed to adjust allotment of already existing services running on Vortex.

Both implementations use the control interface to create new secure channels to the respective administrative interfaces at the Vortex side, and passing the parameters required to achieve the desired effect.

3.5.2 Hubble Storage Interface

Hubble provides a storage interface for persisting performance data samples. To reduce client/cloud communication, app developers are encouraged to access performance data through the Hubble storage interface instead of interacting directly with the monitor by use of the Hubble control interface. An overview of the storage interface is presented in Table 3.3.

All functionality related to retrieving performance data requires one parameter and one optional parameter. The first parameter is a list of unique ids of entities that performance data are to be aggregated over. The second parameter is an optional specification of the start and stop time.

The insert function requires one parameter, an entity object as described in Section 3.2. The implementation will be required to follow the relations and take care of placing the data in the correct tables.

hubble.storage interface

Interface	Functionality
getCPU	Get per CPU cycle usage
getCores	Get per core cycle usage
getMem	Get Memory usage
getIO	Get I/O usage
getApp	Get Application level metric
putEntity	Insert new Usage sample

Table 3.3: The Hubble storage interface.

The size in bytes of a performance data sample will vary depending on type of instrumentation, number of reported entities, etc. For example, a Vortex sample is typically in the order of 12KB. Accumulating samples at a high rate and over longer periods of time can thus result in substantial storage requirements. For example, assuming that each sample is 12KB and that samples are obtained at a rate of one per second, storing samples over a 12 month period would require approximately 378GB of storage space.

The database interface implementation is the same across cloud providers. When designing the Hubble storage, several approaches were implemented and tested. Initially, performance data was stored in Entity Framework 4 containers. These are in-memory structures that allow fast inserts and lookups, and provide the programmer with explicit control over when data is persisted to disk. However, our experience was that Entity Framework 4 containers suffer from exceedingly long persist times where no concurrent reads or writes can be performed. For example, even with a small amount of samples (less than 10), we experienced periods of up to $4-5$ seconds where a container was inaccessible due to being persisted.

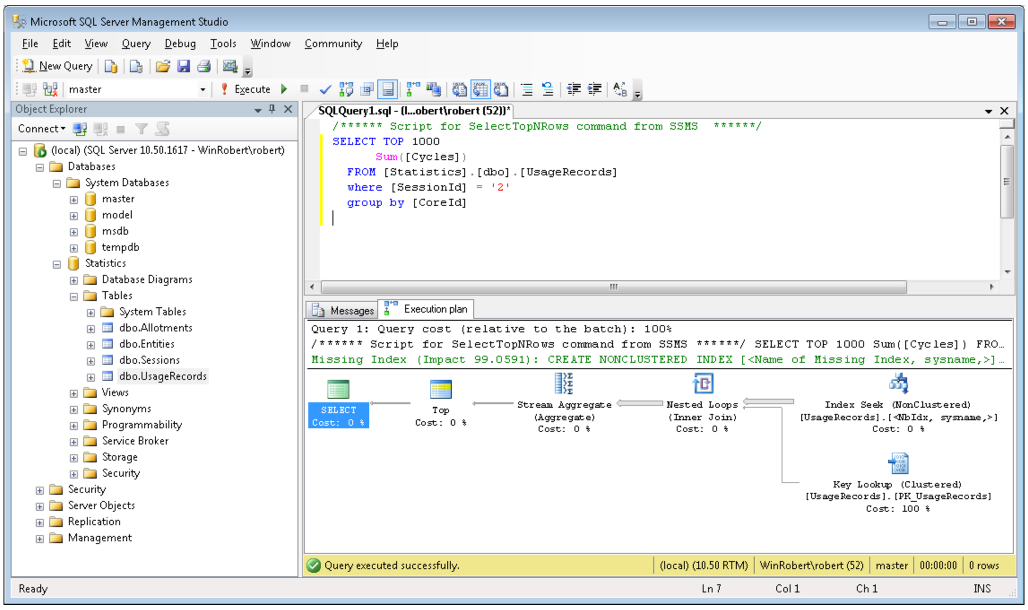
Storing performance data in flat files (with a log rotation approach) was considered. This approach, however, was deemed likely to result in poor performance due to the need for repeated scans when searching and aggregating.

Hubble currently relies on a database approach for storing performance data. The implementation uses a Microsoft SQL table scheme that can be deployed either to a local Microsoft SQL Server instance or, which has been tested, to remote Microsoft Azure Storage.

To avoid partial updates and improve write performance, one single transaction is used in the implementation of the put functionality. This ensures that partial data cannot be read before all data belonging to the specific sample has been committed. The transaction spans a bulk insert job for the entire sample to improve performance.

To further improve the performance of the database, Microsoft SQL Server Management Studio was used to detect missing indexes and get suggestions for how to improve the query. An illustration is shown in Figure 3.8.

With the information obtained from the Management Studio, we have created several non-clustered, non-unique indexes on the tables. Non-


Microsoft SQL Server Management Studio
File Edit View Query Debug Tools Window Community Help
New Query
master
Execute
Object Explorer
Connect
(local) (SQL Server 10.50.1617 - WinRobert\robert)
Databases
System Databases
master
model
msdb
tempdb
Statistics
Database Diagrams
Tables
System Tables
dbo.Allotments
dbo.Entities
dbo.Sessions
dbo.UsageRecords
Views
Synonyms
Programmability
Service Broker
Storage
Security
Security
Server Objects
Replication
Management
SQLQuery1.sql - (l...obert\robert (52))*
Messages
Execution plan
SELECT Cost: 0 %
Top Cost: 0 %
Stream Aggregate (Aggregate) Cost: 0 %
Nested Loops (Inner Join) Cost: 0 %
Index Seek (NonClustered) [UsageRecords].[<NbIdx, sysname,>] Cost: 0 %
Key Lookup (Clustered) [UsageRecords].[PK_UsageRecords] Cost: 100 %
Ready
Ln 7
Col 1
Ch 1
INS

```sql
/****** Script for SelectTopNRows command from SSMS  ******/
SELECT TOP 1000
      Sum([Cycles])
  FROM [Statistics].[dbo].[UsageRecords]
  where [SessionId] = '2'
  group by [CoreId]
```

```
Query 1: Query cost (relative to the batch): 100%
/****** Script for SelectTopNRows command from SSMS ******/ SELECT TOP 1000 Sum([Cycles]) FRO…
Missing Index (Impact 99.0591): CREATE NONCLUSTERED INDEX [<Name of Missing Index, sysname,>]…
```

```
Query executed successfully.   (local) (10.50 RTM)   WinRobert\robert (52)   master   00:00:00   0 rows
```

Figure 3.8: Microsoft SQL Server Management Studio in the process of analyzing a query. A suggestion has been made to create a non-clustered index to improve query performance.

clustered indexes have the property that the physical order or the items in the database is independent of their indexed order. This removes the constraint that the items need to be physically sorted as well as logically sorted. Benchmarks performed in the Management Studio show these indexes reduces the average query time of a single query from 1 second to 10ms.

The implemented get functionality returns LINQ enabled identifiers, which can be iterated directly over or used indirectly in other functionality. The get functionality is implemented by translating the parameters to a LINQ query that will, when executed, return the database items included in the list of unique ids with the associated resource usage. The LINQ enabled identifiers have their query execution deferred till the actual elements are read.

3.5.3 Hubble Analytical Interface

The analytical interface aims to provide functionality to analyze performance data in an efficient manner. Table 3.4 outlines the interface.

movingAverage is an interface that enhances the LINQ enabled identifier returned from the storage interface with moving average calculations. The interface requires three parameters, where the first describes the moving average formula. The formula can be exponential moving average, triangular moving average, or other moving average formulas. The second parameter describes the interval the moving average should be calculated over, and the third parameter is the LINQ identifier returned from a storage interface functionality.

Our implementation exploits the financial formula component of the .NET framework to provide a wide array of moving average formulas. The financial formula is applied directly to the LINQ identifier, so the formula is executed when the items are iterated.

normalize normalizes the performance data retrieved through the storage interface, and requires two parameters. The first parameter describes the interval the data should be normalized over and the second is the LINQ enabled identifier from the storage interface.

The implementations are not allowed to iterate over the elements in the LINQ identifier, since this could lead to multiple iterations of the data, and possibly un-needed data transfers from the database. Instead the implemented functions need to enhance the LINQ query with the functionality required from the interface so that the query is executed only once to obtain the desired result.

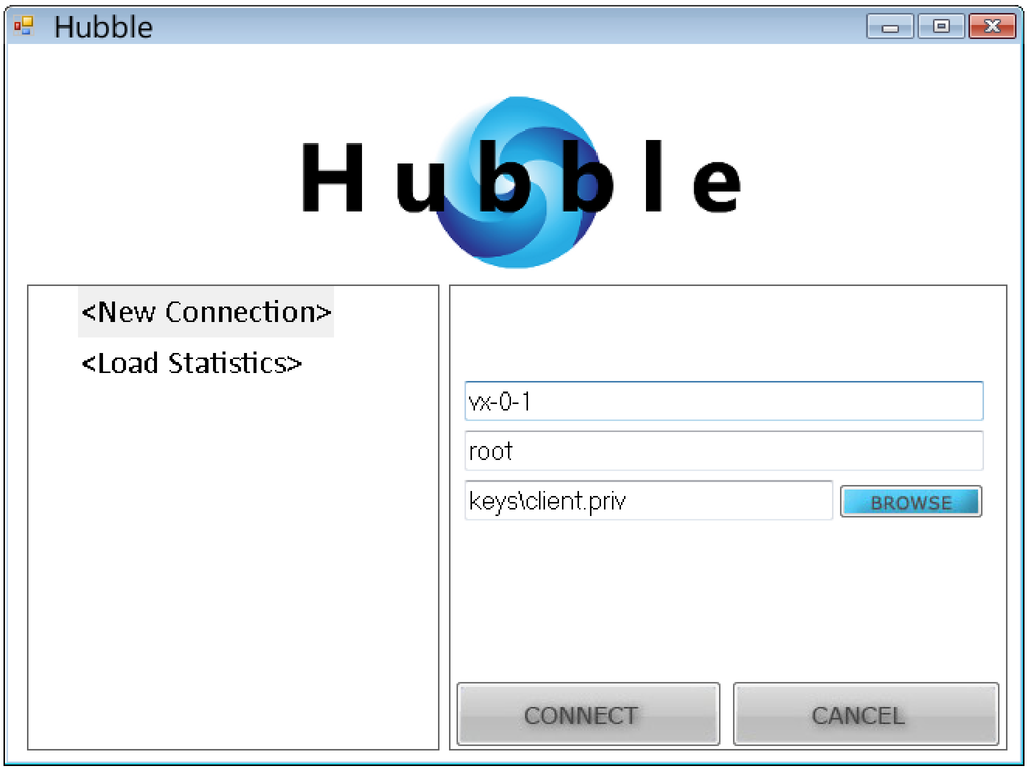
bigData is our interface for large scale analytics, and is meant to provide a simple interface for starting MapReduce, Dryad or other large scale analytics jobs on large datasets collected to the database. The function requires one parameter, which is a LINQ query describing the analytical job.

An implementation of bigData could take advantage of local GPU resources for performing MapReduce jobs through Mars[6] locally, or instantiate

hubble.analyze interface

Interface	Functionality
movingAverage	Get the moving average
normalize	Normalize
bigData	Large scale analytics

Table 3.4: The Hubble analytical interface.



vx-0-1
root
keys\client.priv

Figure 3.9: Login screen of Hubble. Requires the user to provide a host name, service name and a private key file.

virtual machines in the cloud for running MapReduce jobs remotely.

Another opportunity for the bigData interface is to facilitate cheap compute resources like the Amazon EC2 Spot instances. Spot instances is a new way of exploiting left-over compute resources. At any time there are data centers in the Amazon EC2 cloud that are under utilized, either because the region that the data center serves currently are in the night or other reasons. These compute resources are volatile and can be disrupted at any time if the resources are needed elsewhere in the Amazon cloud. Regardless, these compute resources are well designed for MapReduce jobs, as the workers in a MapReduce job are idempotent and can be restarted at a later time.

Because of lack of resource to instantiate MapReduce jobs we have not been able to test an implementation of this interface.

3.6 Connecting and launching apps

The app platform has a user interface for connecting to the cloud and launching apps. The design aims to be simplistic and require a minimum of user input. The login screen, as seen in Figure 3.9, requires the user to provide a host name, a service identifier and a private key file for authentication during cloud login. The credentials we have chosen to support is the public-key authentication scheme, as described in Section 3.4.

Before being able to connect to the cloud, the client needs to make sure that the public key is installed at the remote host, and that it gives access to the processes that are to be monitored. Most enterprises already have a public-key infrastructure, and by supporting this authentication method, the enterprises can make use of this solution without implementing new authentication mechanisms.

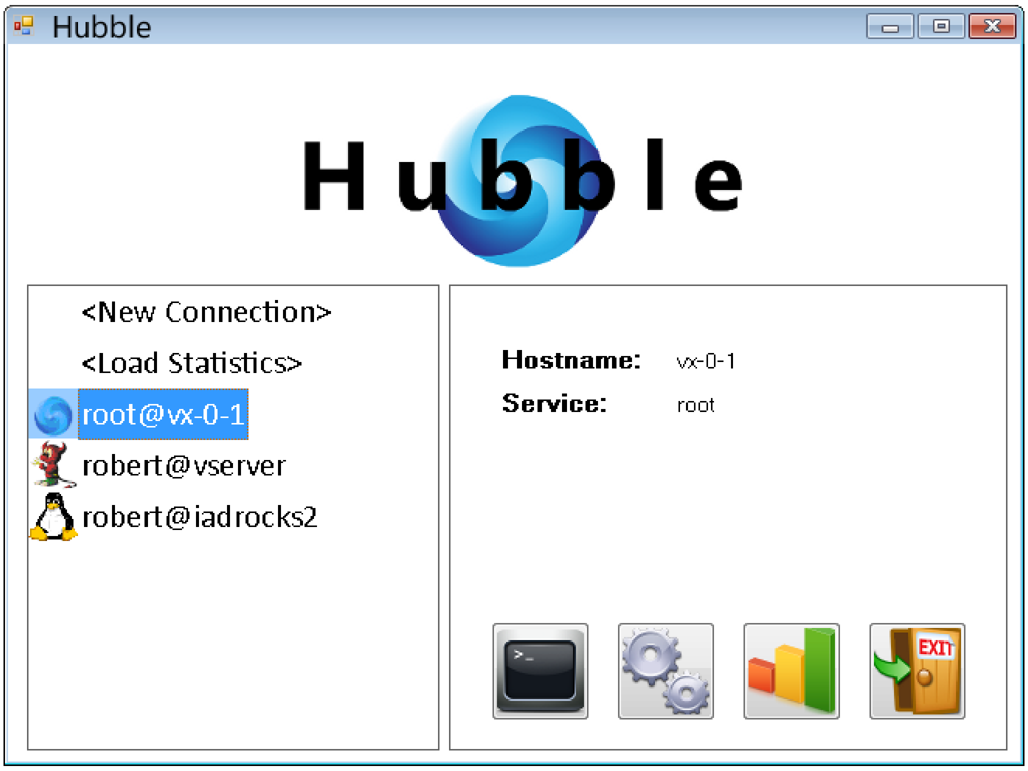
Hubble uses the cryptographic library from Renci[1] since it already supports the encryption standards often used by cloud providers. We have chosen to utilize a well tested library for our encryption and integrity for two reasons; (i) we can support more authentication methods than is feasible to implement in the timespan of this thesis, and (ii) to be able to assert that the encryption methods validate and are correct. The library also have hooks for Secure Copy Protocol (SCP) for deploying the monitor code in the cloud if this should be needed.

After the necessary information have been provided, Hubble will try to authenticate the client, using the provided credentials, to the specified cloud. The authentication process is explained in detail in our work with SVOSH[11]. If the provided credentials are validated, a secure channel is created between the client side app platform and the remote host SSH server.

After connecting to the cloud, Hubble will probe for the capabilities of the remote host. These probes include shell and execute functionality, but also whether the monitor is deployed at the cloud provider. The apps that are compatible with the functionality at the cloud will be presented to the user, as illustrated on Figure 3.10.

All the available apps utilize the same connection and can multiplex several

[1]http://sshnet.codeplex.com/



Hostname: vx-0-1
Service: root

Figure 3.10: Hubble connected to a Vortex, Linux and a BSD node, with the compatible apps listed on the right hand side.

separate data channels on the same secure channel if needed. Each of the separate data channels maintain their own sliding window protocol to prevent congestion and exercise back-pressure to be able to consume data as it is generated at the remote host.

At this point the user may launch any number of apps.

Chapter 4

Apps

This chapter describes apps developed to run on top of the Hubble platform.

As part of our work we have implemented many Hubble apps. The functionality provided by these apps can be categorized as (i) retrieval of performance data, (ii) visualization of performance characteristics, and (iii) management of cloud services.

4.1 Rover: Performance data retrieval

We have named our performance data retrieval app Rover after Mars Rover, which is a probe running on Mars gathering planetary data. Rover was the first app developed for Hubble, and provides functionality for collecting performance data from the monitor residing in the cloud and for storing this data in the Hubble database.

Figure 4.1 illustrates the Rover architecture. Rover is implemented in C# on the .NET framework, and encompasses 7463 lines of code. Of these lines, roughly 6000 was auto-generated to handle the structure of the GPB format.

The Rover architecture is centered around a *queue*, which is protected by a synchronization construct (a .NET monitor). When Rover receives data, the raw wire-level protocol data is placed in the queue and the synchronization construct signals that there are data available in the queue.

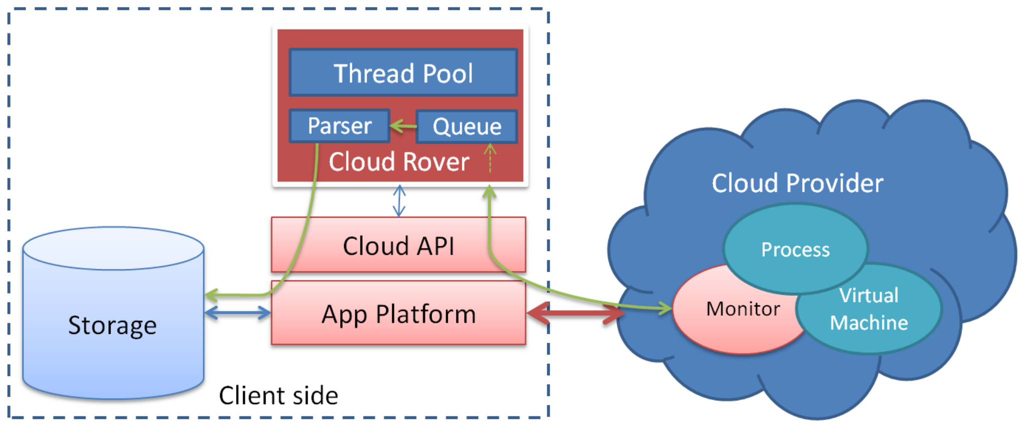


Figure 4.1: Overview of the Rover architecture. Components that are included in Rover are in the red box. Rover communicates with the monitor and database through the API.

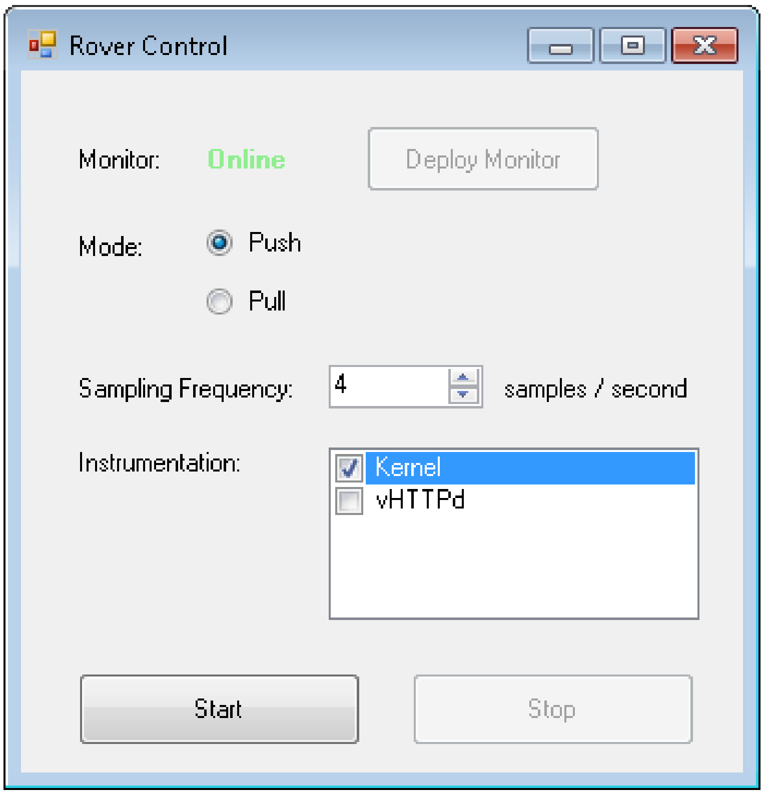
The signal will wake up a thread in a *thread pool*. The thread pool is implemented using the Task Parallel Library (TPL). TPL provides an efficient and scalable use of system resources by using algorithms like hill-climbing to determine and adjust the number of threads to maximizes throughput. To complement this, work-stealing algorithms are employed to provide load-balancing.

Upon wakeup, the workers in the thread pool will dequeue packets from the queue, and parse the hierarchical structure of the entity wire-level data format explained in Section 3.2 into single entity objects. Each of the entity objects will in turn be passed to the insert function in the storage interface.

Upon launch, Rover uses the Hubble control interface (see Section 3.5.1) to establish a secure communication channel to the cloud. This channel is then used to connect to the monitor, or to deploy the monitor using a SCP channel if needed.

After ensuring that the monitor is present at the cloud, Rover sends a request to the remote SSH server instructing that the monitor process be attached to the remote end of the secure channel. This way of using SSH to secure the communication of an otherwise insecure application is similar to the one used with the Secure File Transfer Protocol (SFTP) in the OpenSSH system. At this point Rover has a secure duplex channel to the monitor.

Rover continues by requesting the monitor capabilities. These capabilities



Monitor: Online

Mode: [✓ Push] Pull

Sampling Frequency: 4 samples / second

Instrumentation: [✓ Kernel] vHTTPd

Figure 4.2: The Rover control interface allows the user to deploy the monitor should it not be present, instruct the mode of retrieval and the sampling frequency and which instrumentation code should be retrieved.

include whether the monitor supports pull or push based retrieval, or both, and whether the client can exercise control over what instrumentation that is deployed (see Section 3.3).

Different monitoring scenarios might have different requirements with respect to sampling frequency. Some scenarios might require one sample each hour, while other scenarios might require several samples per second. A third scenario might even require different sampling frequencies at different times of the day. As such, Rover exposes a graphical user interface, shown in Figure 4.2, that presents monitor capabilities and where the user can configure Rover operation.

After the configuration has been entered, Rover starts retrieving performance data and writing this data to storage.

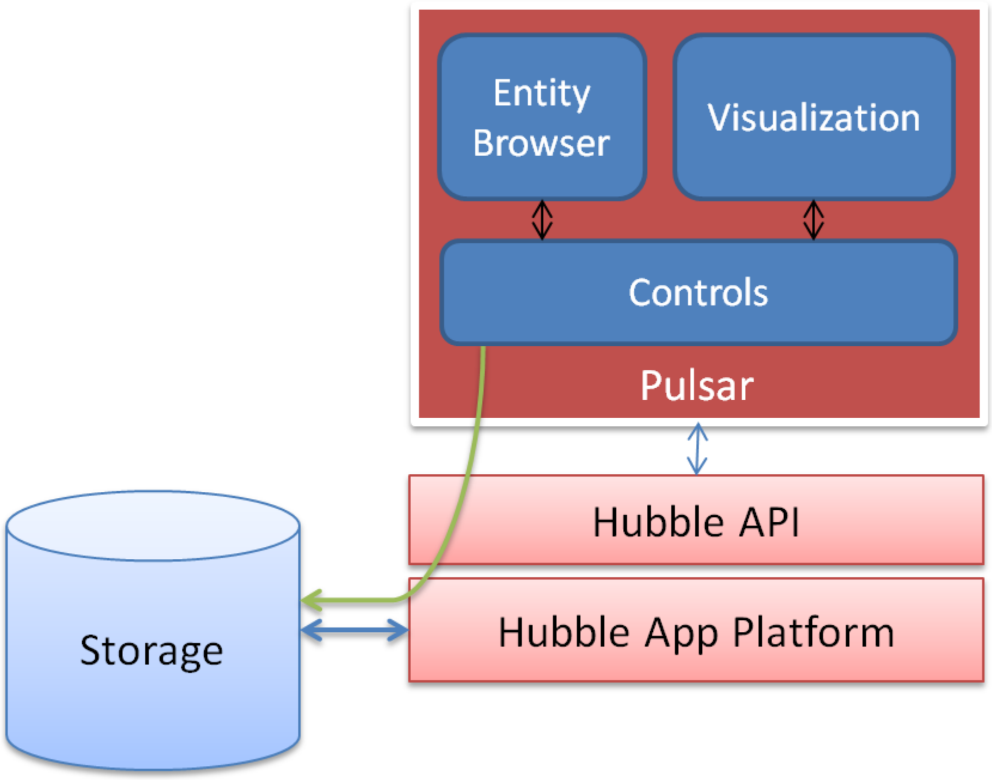


Figure 4.3: The architecture of the Pulsar app running on the Hubble platform.

4.2 Pulsar: Performance data visualization

Visualization of performance data can have many uses. Charts can verify whether alloted resources are in fact available, implementation anomalies can be detected, and one can learn about resource usage pattern for different periods of the day. These are only a few examples of uses.

Pulsar is a visualization app implemented in C# for deployment on the Hubble platform. It is built using Windows Forms, and is assembled using custom components which consists of standard .NET controls. Pulsar encompasses 3153 lines of code, where 990 lines are related to auto-generated data structures.

Pulsar can be configured to request performance data from Hubble storage at a specific rate, using current time as a timestamp, or Pulsar can operate on data in Hubble storage that were collected within a specified time window. Using the former mode of operation, Pulsar visualizes performance data as it is stored by Rover and the visualization reflects the current state of the cloud environment.

As shown in Figure 4.3, there are three main components in the Pulsar app: (i) an entity browser, (ii) a control interface, and (iii) a visualization view,

The entity browser presents the user with the available entities in the performance data (see Section 3.2), and communicates user-selected entities to the control interface. The control interface enables the user to make choices with respect to how the data will be visualized. The visualization view in turn visualizes the entities, utilizing the chosen options from the control interface.

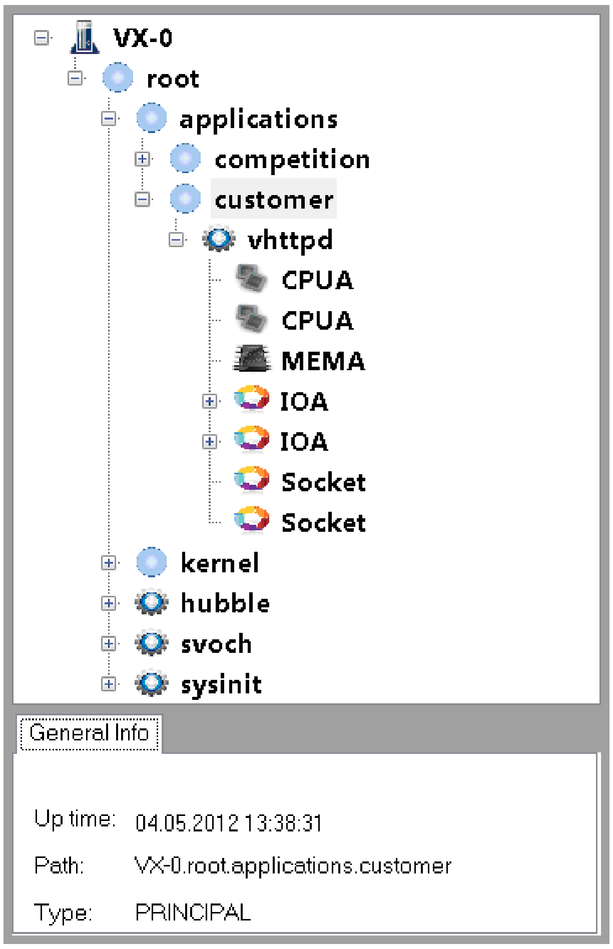
4.2.1 Entity Browser

The Pulsar Entity browser examines performance data samples and constructs a tree view layout to visualize entities based on the parent-child relationship of entities as found in the performance data. As shown in Figure 4.4, the resulting visualization resembles a typical Windows explorer layout. This makes searching for specific entities similar to finding a folder in Windows explorer.

The entity browser parses each entity and places the relevant data in an visual entity object, containing the unique ID and the human readable identificator found in the database. The entity browser separates the different entity types using custom-created icons representing the type of entity (see Section 3.2).

The contents of the entity browser is continuously updated as the visualization progresses. The contents of the browser will reflect the time interval that is currently visualized; if an entity is present in a performance data sample within the visualized time interval, the entity will be presented in the browser.

When the user selects an entity, the browser descends the entity hierarchy to extract the unique identifiers of entities (see Section 3.2) below the selected entity. Also, the browser ascends the hierarchy to find the nearest principal entity to discover what resources are available to the selected entity. These unique identifiers are then communicated to the control interface.



Up time: 04.05.2012 13:38:31
Path: VX-0.root.applications.customer
Type: PRINCIPAL

Figure 4.4: The Pulsar browser component facilitates browsing of the entities in the database. Entities are assigned a type specific icon that represent the type of resource.

4.2.2 Control interface

The Pulsar control interface serves two purposes. The first is to shape the performance data, by exercising control over how much data will be gathered and by applying analytical formulas using Hubble's analytical interface. The second is controlling layout of the chart in the visualization view.

The control interface operates on the set of unique identifiers received from the Entity Browser. Using the Hubble storage interface, the control interface obtains LINQ enabled identifiers for the corresponding performance data.

The control interface provides the user with a graphical user interface to control how the performance data is visualized. The user can choose from

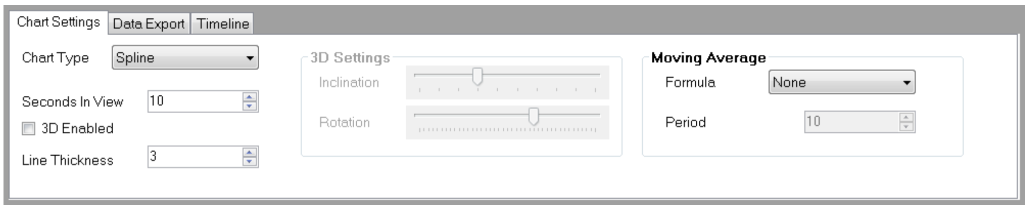


Chart Type: Spline
Seconds In View: 10
3D Enabled
Line Thickness: 3
Formula: None
Period: 10

Figure 4.5: The Pulsar control interface lets the user control the layout of the graph and shape the performance data.

a variety of different plots, ranging from line and spline charts to bar and column charts. The user can also exercise control of how many seconds will be presented in the chart.

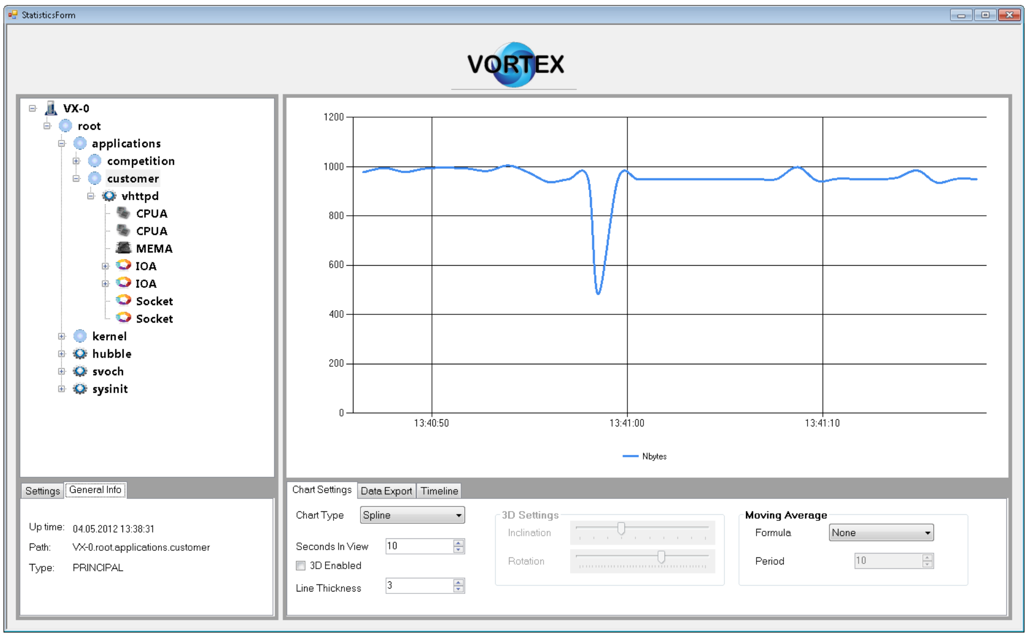
The user can also specify that the charts are displayed in 3D. In 3D mode, the user can further enhance the visualization by adjusting the inclination and rotation degree of the chart.

The user can further enhance the chart by applying different combinations of the functionality found in Hubble's analytical interface. The user can select different normalization and or moving average formulas that are to be applied to the data before transmitting the resulting LINQ enabled identifier along with the layout parameters to the visualization view. Note that all operations applied to performance data by the control interface are deferred; retrieval of performance data from storage and application of analytical functions occurs in context of the visualization view.

4.2.3 Visualization view

The visualization view is in charge of producing a graphical representation corresponding to the input from the control interface.

The implementation currently uses Microsoft Chart Controls (MCC) for the .NET Framework to produce the graphical representation. MCC is straightforward to use, as it accepts LINQ enabled identifiers as specification of data sources for a chart. Thus, the visualization view can forward the input from the control interface directly to MCC, only specifying additional parameters pertaining to how the data is to be interpreted. These additional parameters specify aspects such as series delimiters, grouping, coloring, line thickness, etc.



Up time: 04.05.2012 13:38:31
Path: VX-0.root.applications.customer
Type: PRINCIPAL

Chart Type: Spline
Seconds In View: 10
3D Enabled: ☐
Line Thickness: 3

Moving Average
Formula: None
Period: 10

Figure 4.6: Pulsar connected to a Vortex node, displaying the network throughput in Mb/s of a service. All components are displayed.

An example of the Hubble interface, including the Entity browser, control interface, and visualization view is shown in Figure 4.6.

We have experimented with other approaches to implementing the visualization view. A commonly used tool for performance visualization on the Microsoft platform is Performance Monitor (PerfMon). PerfMon is not very flexible when it comes to configuration, however, and is only able to show pre-defined graphs.

Figure 4.7 shows performance data from Hubble visualized by PerfMon, along with an architecture for an app that bridges Hubble with PerfMon. The visualization was produced by changing the visualization view component of Pulsar to first evaluate the LINQ enabled identifiers received from the control interface and then post the resulting data to PerfMon. Although we have not done so, the modular structure of the Hubble makes it possible to create a new app that relies on PerfMon for visualization rather than Microsoft Chart Controls (MCC).

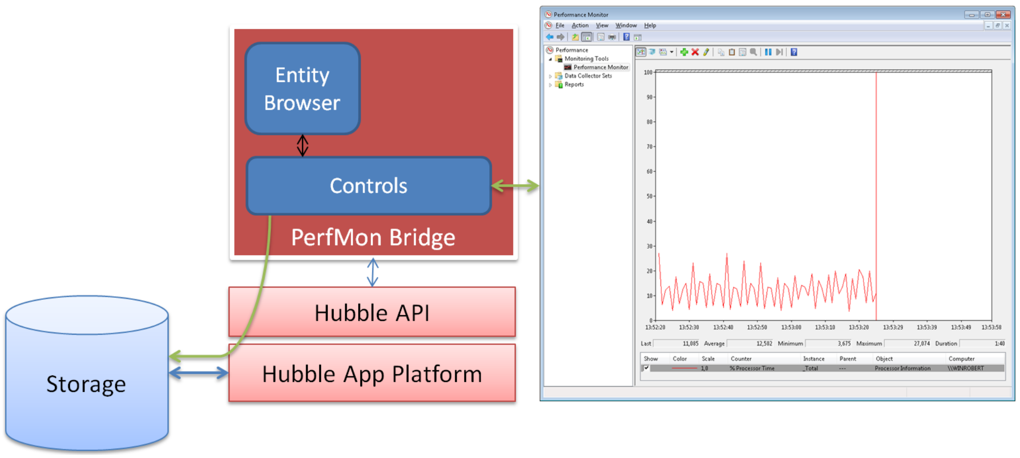


Figure 4.7: Architecture of the PerfMon bridging app.

4.3 Uranus: Cloud management

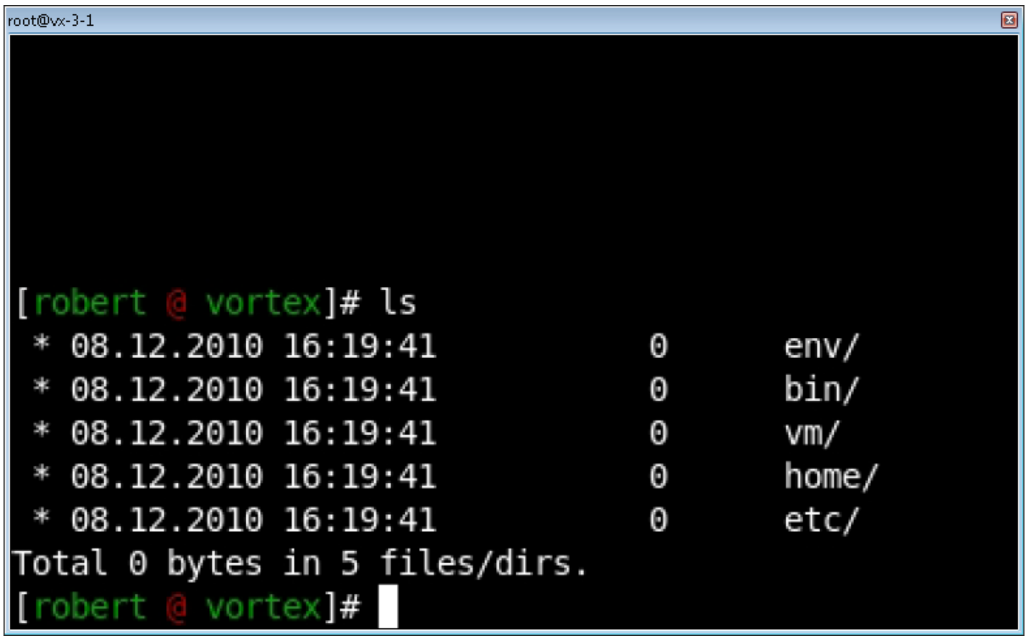
The main use of the SSH protocol is gaining access to execute commands either directly or indirectly through a shell at the remote host. As such we have provided the user with apps to do both these tasks through the already established secure channels. The internal channel multiplexing within the SSH protocol takes care of delivering the messages at the correct remote application. We have chosen to call these apps Uranus, after the Greek god personifying the sky.

When starting the shell app, a separate internal shell channel in the already established secure channel is created through the Hubble control interface. In the process of establishing a shell channel, the system negotiates with the remote host as to which character are to be used as newline character to make output readable. After the channel is established, the platform requests the remote shell be attached to the remote end of the channel.

Uranus consists of two separate, but similar, apps: the shell and the execute app.

The shell app is implemented using a *textbox* .NET control, the same control used when creating text editors. The *textbox* control is augmented with a keystroke interceptor which intercepts keystrokes and transmits them to the remote host over the established shell channel.

Upon receiving data from the remote shell, the shell appends the output



```
[robert @ vortex]# ls
 * 08.12.2010 16:19:41          0      env/
 * 08.12.2010 16:19:41          0      bin/
 * 08.12.2010 16:19:41          0      vm/
 * 08.12.2010 16:19:41          0      home/
 * 08.12.2010 16:19:41          0      etc/
Total 0 bytes in 5 files/dirs.
[robert @ vortex]#
```

Figure 4.8: The shell app utilizes the current active SSH connection to multiplex commands to the cloud and results back to the app.

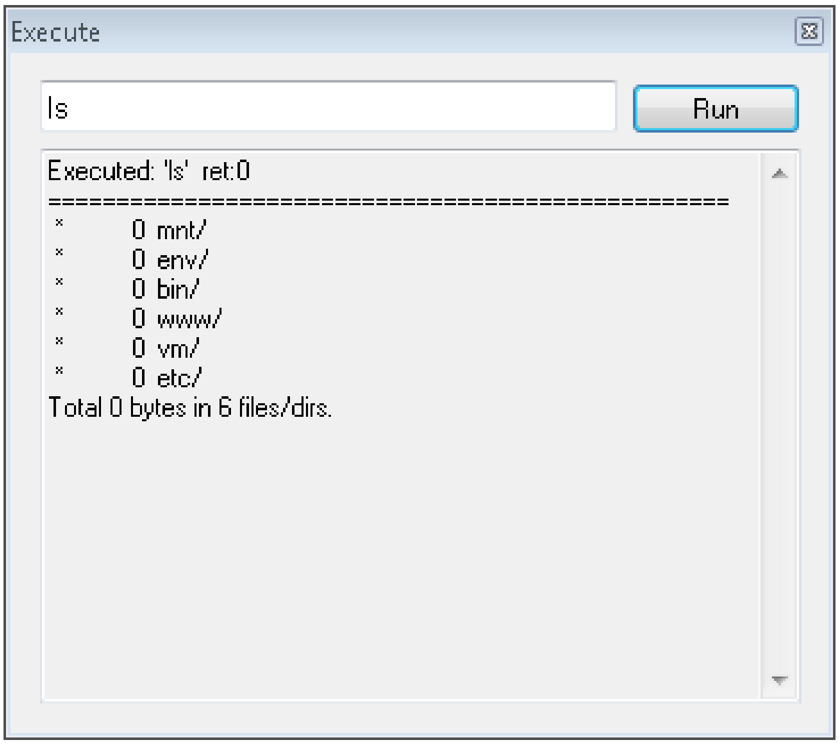
using the correct newline character in the *textbox*. An illustration is seen in Figure 4.8.

The execute app is implemented in a similar fashion, but instead of creating a secure shell channel, the execute app creates an execute channel. The user specifies a process to execute at the remote host in an editable text box. This process is added as the second argument when creating the execute channel.

Both apps will poll the resulting Hubble channel identifier and wait for the cloud-side process to terminate. Any output the process generates along with the exit code will be displayed in a read-only text box. An illustration is seen in Figure 4.9.

If the cloud provider supports an API for configuration of the resources made available to the tenant, an app could be created to automatically request and relinquish resources depending on whether the deployed application meets a configurable application metric.

For example, a web server could be configured to be able to serve 1000 requests per second. If the web server cannot meet this requirement, the app



```
Executed: 'ls'  ret:0
==================================================
*        0  mnt/
*        0  env/
*        0  bin/
*        0  www/
*        0  vm/
*        0  etc/
Total 0 bytes in 6 files/dirs.
```

Figure 4.9: The execute app utilizes the existing SSH channel to transmit commands and receive results back.

would analyze which resources are being used the most and increase these the allotment for these resources. In the web server example this resource could be network or disk bandwidth. If the memory or cpu resource is overly provisioned, the app could be configured to relinquish some of the resources to reduce cost. The app could also be configured to move the application to another provider should the time and day dictate that another provider could provide the required resources for less money.

Although we have not implemented such an app yet, in Chapter 5.2 we demonstrate an experiment where the shell app was used to manually issue resource allotment changes, thereby mimicking the behavior of such an app.

Chapter 5

Experiments

This chapter presents our experience from running experiments with the Hubble platform.

Two types of experiments were performed. The first was a typical scenario where a company plans to deploy an application and wants to learn about its behavior, and then tune the alloted resource depending on observed performance. The second experiment aimed to explore the overhead of monitor operation.

All experiments involve use of cloud nodes running Vortex.

5.1 Experimental setup

For our experiments we used Dell PowerEdge M600 nodes equipped with 2 Intel Xeon E5430, each with 4 cores running at 2.66GHZ, 16GB of RAM, and a 1Gb/s ethernet network interface.

The Vortex nodes ran Vortex build 32768, and the load generator nodes ran CentOS release 5.5.

The client computer running the Hubble platform was a Dell Precision 390 with a single processor equipped with 4 cores running at 2.40GHZ and 8GB of RAM. The computer was running 64bit Windows 7, with Microsoft SQL server 2008 R2 installed for storage.

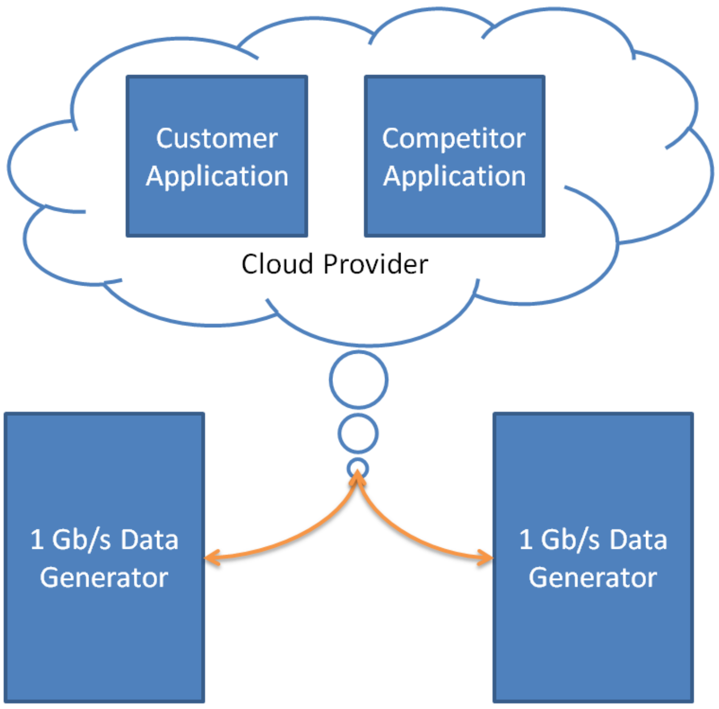


Figure 5.1: Overview of the test setup. Two competing applications are installed in the cloud, while two separate load generators generate external load for the applications.

We utilized Pulsar to generate performance graphs throughout these experiments.

5.2 Application Deployment

In this experiment we explore a scenario where a company plans deployment of an application in the cloud. The goal of the experiment is to investigate whether the Hubble platform can aid in deciding the right pricing model for the application, and whether Hubble can be used to change allotted resource at the cloud provider to ensure that the application can sustain a certain performance level.

The application the company plans to deploy is a typical cloud application that service clients with data based on some parameters in client requests. The service is assumed to be I/O-bound, as negligible computation is involved in determining what data to serve to a client.

As shown in Figure 5.1, we deploy two instances of the application. This was done to make the deployment more realistic, as cloud providers typically co-host tenant environments, causing there to be competition for resources at each cloud node.

The cloud node was connected to two load generator nodes. On these we ran the Apache Benchmarking tool AB[1] to generate load for each of the application instances. AB was configured to run with 32 concurrent requests, where new requests are created upon completion of previous requests (a closed loop setup).

On the cloud node we configured Vortex with Weighted Fair Queuing[4] schedulers. Such schedulers assign weights to clients and ensure that each client receives resources in proportion to its assigned weight. The schedulers are also work-conserving; if at any given time there are idle resources and there is demand from a client, that client will receive resources. If there is competition for idle resources, those resources are shared among demanding clients in proportion to their weight. By comparing a client's weight to the sum of all assigned weights, one can determine the minimum resource entitlement for that client. Any received resources above that minimum originates from idle resources.

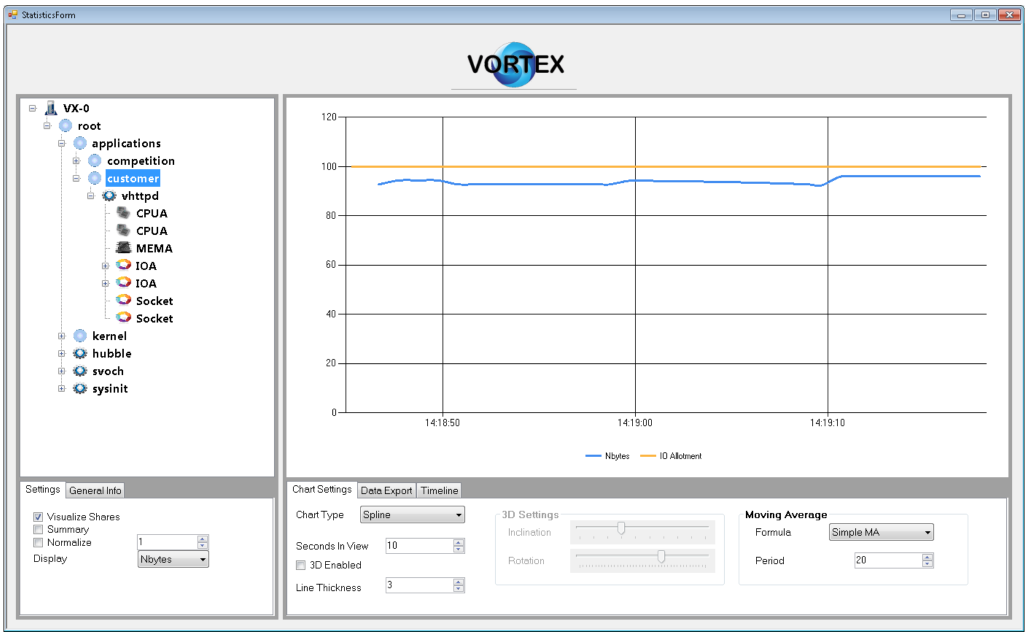
Initially the client application was deployed to the Vortex node with 10% of the available CPU and I/O resources. The resource utilization is shown in Figure 5.2 and Figure 5.3, where utilization is plotted along with resource allotment.

By examining the graphs, it is evident that the I/O resource is used to it's fullest (100 out of 1000 Mbit/s available) while there are ample CPU resources available. This confirms the initial assumption that the application was I/O bound.

Looking at application performance, illustrated in Figure 5.4, the current allotment of resources results in a performance of approximately 50 requests per second.

Intuitively these performance characteristics indicate that increasing the amount of allotted I/O resources would linearly increase the application performance.

[1]http://www.apache.org

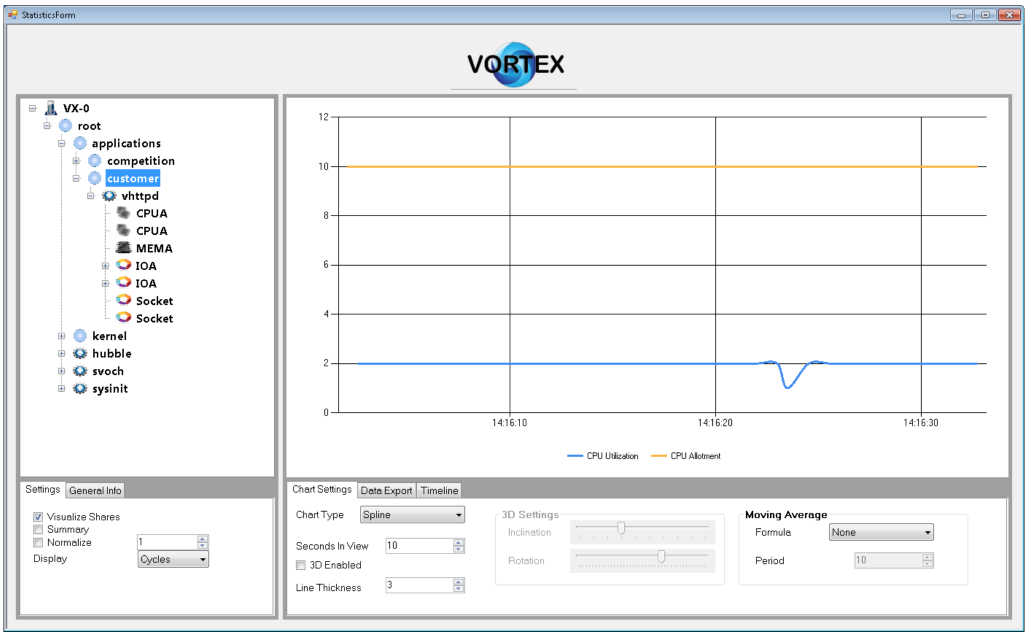


Visualize Shares: ✓
Summary
Normalize: 1
Display: Nbytes

Chart Type: Spline
Seconds In View: 10
3D Enabled
Line Thickness: 3

Moving Average
Formula: Simple MA
Period: 20

Figure 5.2: I/o utilization when operating with 10% of the available network resources, measured in Mbit/s.

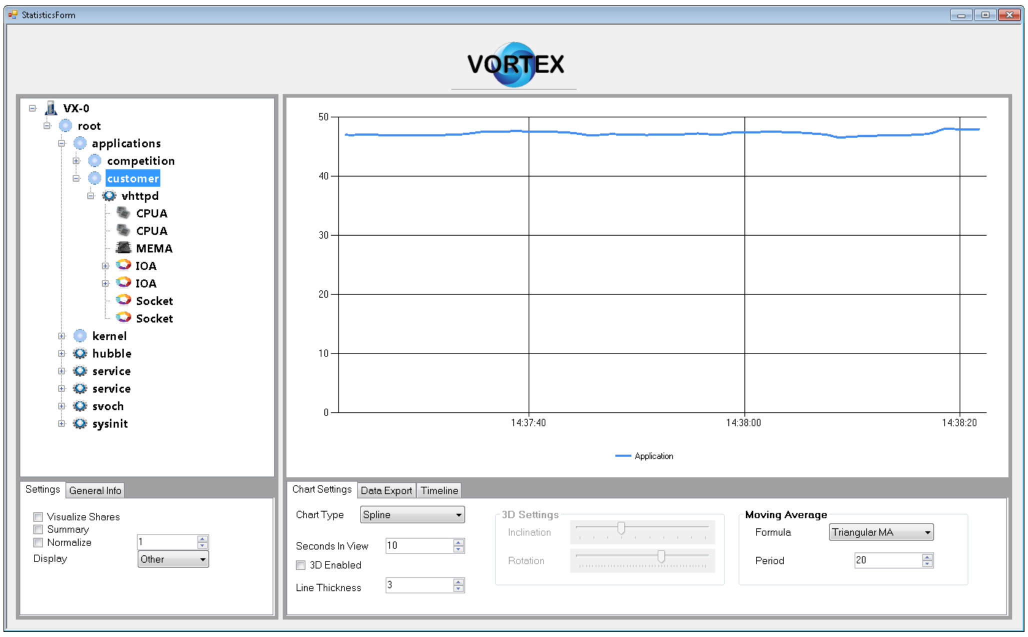


Visualize Shares: ✓
Summary
Normalize: 1
Display: Cycles

Chart Type: Spline
Seconds In View: 10
3D Enabled
Line Thickness: 3

Moving Average
Formula: None
Period: 10

Figure 5.3: CPU utilization when operating with 10% of the available resources, measured in percent.



Visualize Shares
Summary
Normalize 1
Display Other

Chart Type Spline
Seconds In View 10
3D Enabled
Line Thickness 3

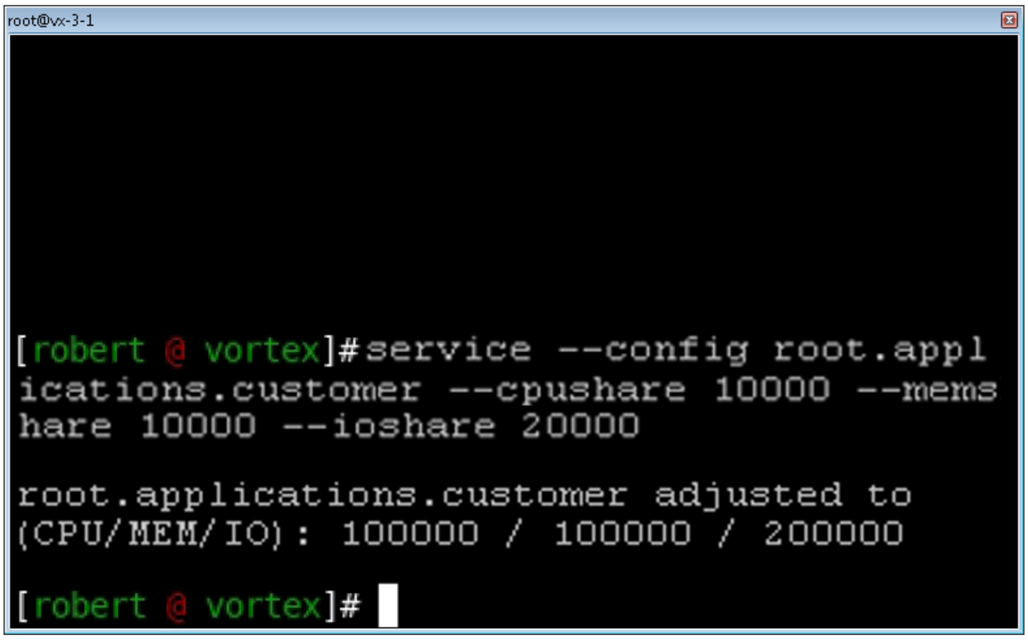
Moving Average
Formula Triangular MA
Period 20

Figure 5.4: Application performance when operating with 10% of the available resources, measured in requests per second.

To test this hypothesis we double the allotted I/O resources, from 10% to 20%, while keeping the allotted CPU resources at 10%. This was accomplished by use of the shell app to access the Vortex API for adjusting resource allotments. Figure 5.5 shows how the adjustment was performed. Note that here we use the shell to execute a command that normally would only be accessible to the cloud provider, i.e. adjusting the amount of resources available to a tenant. As part of this thesis we have implemented user level tools for adjusting the amount of resources available to Vortex services. Since Vortex organizes services hierarchically and only allows a process access to its hosting service and any descendant services, the shell command (and corresponding Vortex-side process) was launched in context of the Vortex root service. This allowed for the command to effectuate changes to the customer tenant resource allotment.

Figure 5.6 and Figure 5.7 illustrates the change in consumption of CPU and bandwidth when increasing the allotment of I/O resources to 20%.

Since our load generators strive to consume all available bandwidth at the different services, we can immediately see the changes in application resource consumption. As expected, we observe that the available network



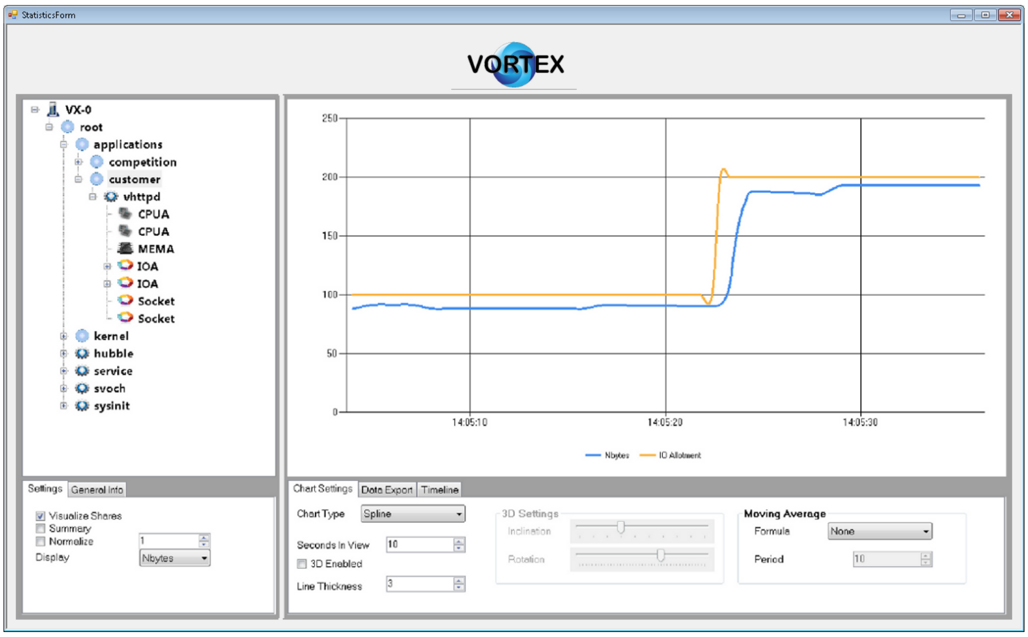
```
root@vx-3-1

[robert @ vortex]#service --config root.appl
ications.customer --cpushare 10000 --mems
hare 10000 --ioshare 20000

root.applications.customer adjusted to
(CPU/MEM/IO): 100000 / 100000 / 200000

[robert @ vortex]#
```

Figure 5.5: Using the shell app we can adjust the alloted resources at the remote host.



Settings | General Info

☑ Visualize Shares
☐ Summary
☐ Normalize: 1
Display: Nbytes

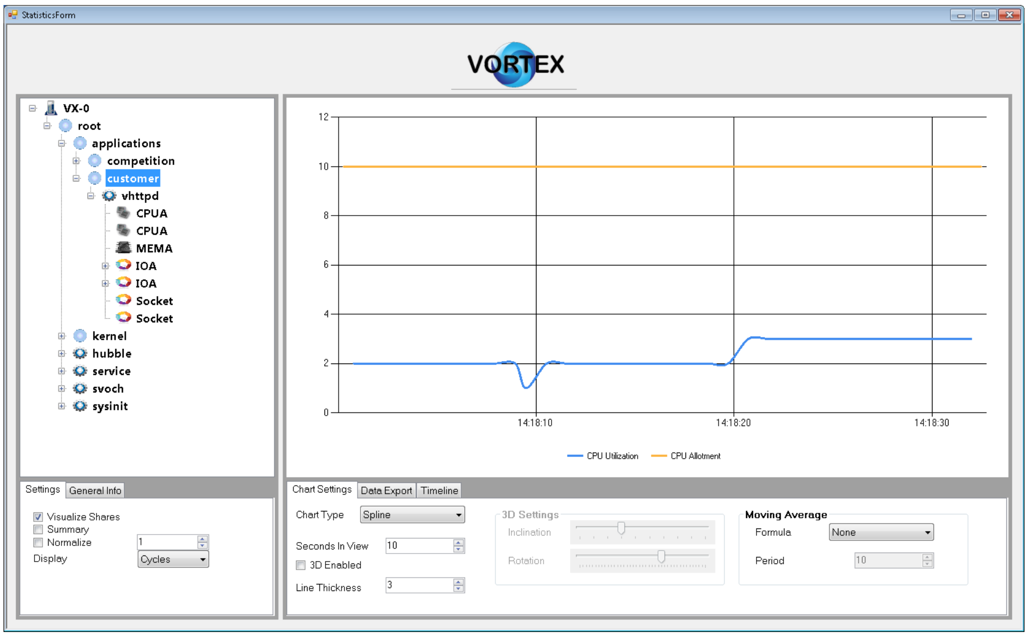
Chart Settings | Data Export | Timeline

Chart Type: Spline
Seconds In View: 10
☐ 3D Enabled
Line Thickness: 3

3D Settings
Inclination
Rotation

Moving Average
Formula: None
Period: 10

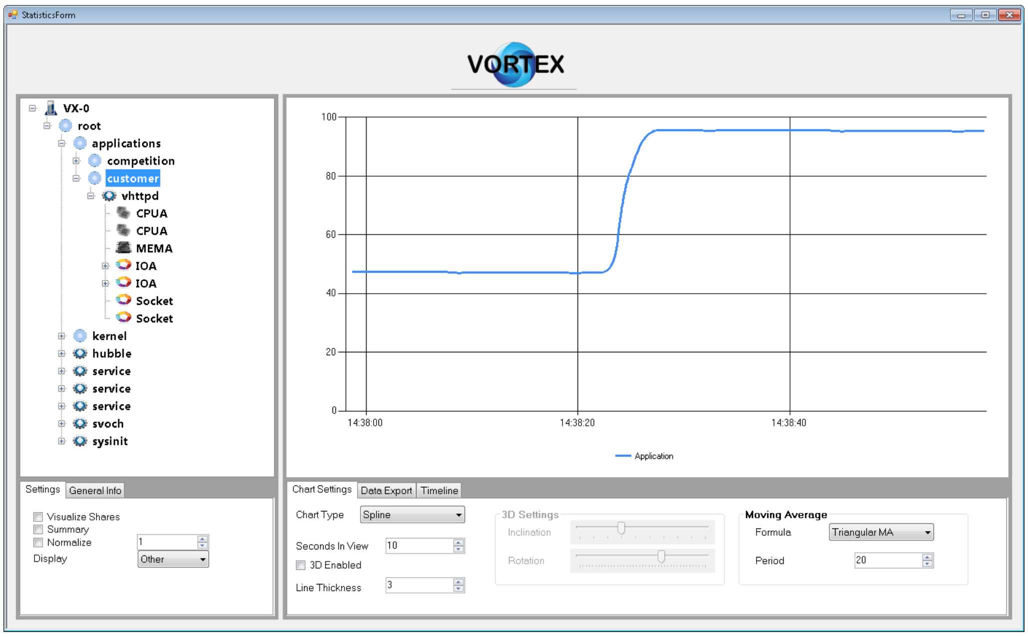
Figure 5.6: I/O utilization after increasing the allotted network resources to 20%, measured in Mbit/s.



Visualize Shares: ✓
Summary
Normalize: 1
Display: Cycles

Chart Type: Spline
Seconds In View: 10
3D Enabled
Line Thickness: 3
Formula: None
Period: 10

Figure 5.7: Cpu utilization after increasing the allotted network resources to 20%, measured in percent.



Visualize Shares
Summary
Normalize: 1
Display: Other

Chart Type: Spline
Seconds In View: 10
3D Enabled
Line Thickness: 3
Formula: Triangular MA
Period: 20

Figure 5.8: Application performance after increasing the allotted network resources to 20%, measured in requests per second.

bandwidth is consumed, while we observe only 1% increase in CPU consumption.

Figure 5.8 shows application performance with the increased allotment of network bandwidth. As expected, the application level performance is commensurate with the increase in available I/O resources.

From these experiments we can conclude that the application requires a cloud environment with few CPU cycles, and if the company wishes to increase application performance, increases in the I/O budget is likely to result in a proportional increase in performance.

5.3 Micro benchmarks

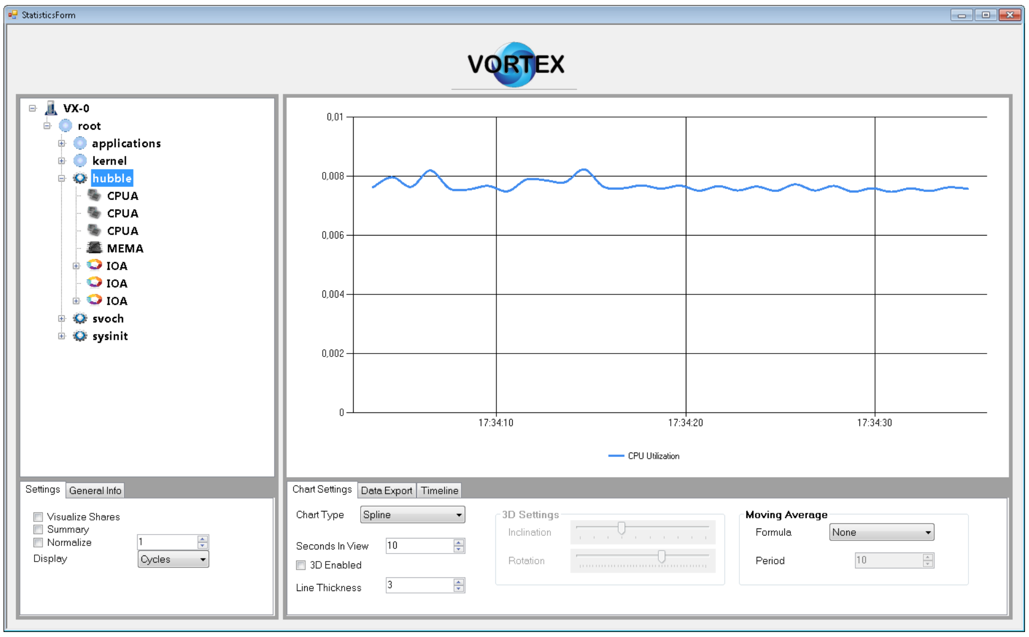
In this experiment we measure the overhead incurred by running the monitor in the cloud. As all resource usage is monitored by the monitor, even resources consumed by the monitor, we are able to use the Hubble platform to measure the resource usage of itself.

Because we are using the SSH protocol to encrypt our data before sending, there is some associated encryption overhead. Since the encryption is performed by SVOCH, a separate process, we are able to differentiate between resources used for monitoring and resources used for encryption.

The user can adjust the rate at which Rover will retrieve updated resource usage records, so we will benchmark our monitor at two different refresh rates. One of our goals was that our monitor would provide users with performance statistics at a very high resolution, and as such we will test our solution at both 1 update per second and 10 updates per second.

When running the experiment we populated the Vortex node with some regular services. This was done to ensure that each performance data sample would contain a substantial number of entities, resulting in both encryption and instrumentation overhead. With these services running, each sample obtained from the monitor encompassed approximately 12Kb of data.

Figure 5.9 and Figure 5.10 shows the CPU utilization of the monitor when running 1 sample per second and 10 samples per second, respectively. The graphs show that the monitor scales its CPU consumption linearly with the number of requests, and that the total consumption is very low.

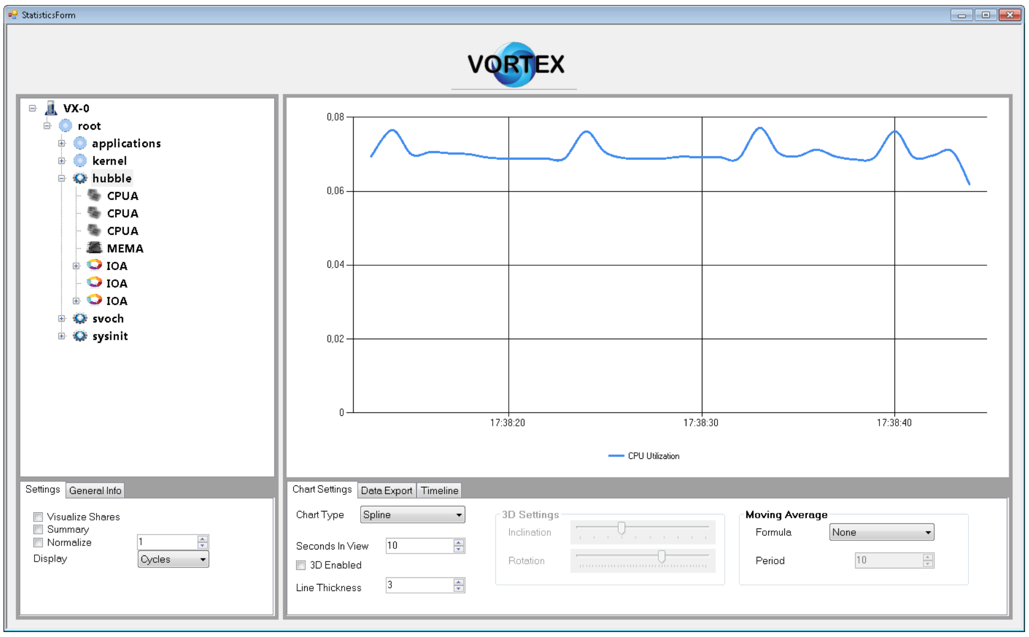


Visualize Shares
Summary
Normalize 1
Display Cycles

Chart Type Spline
Seconds In View 10
3D Enabled
Line Thickness 3

Formula None
Period 10

Figure 5.9: CPU utilization of the monitor at 1 request pr second, measured in percent.

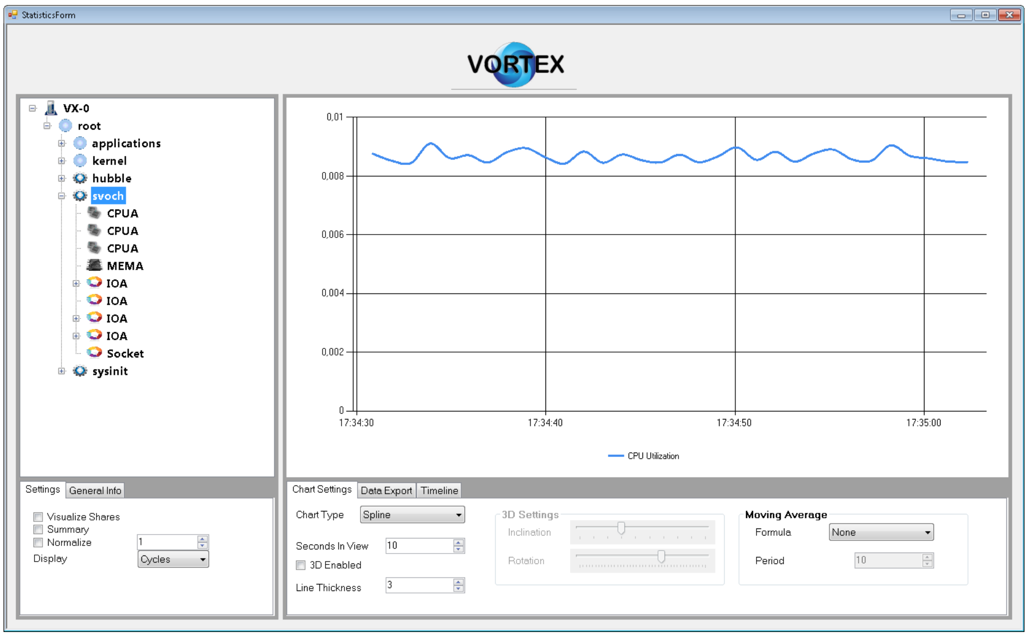


Visualize Shares
Summary
Normalize 1
Display Cycles

Chart Type Spline
Seconds In View 10
3D Enabled
Line Thickness 3

Formula None
Period 10

Figure 5.10: CPU utilization of the monitor at 10 requests pr second, measured in percent.



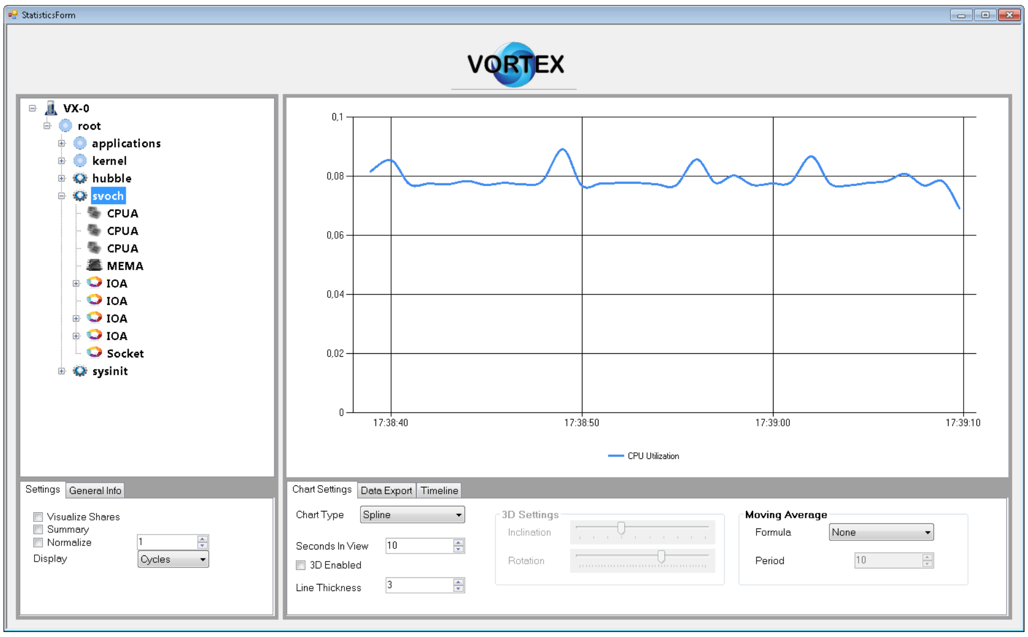
Visualize Shares
Summary
Normalize 1
Display Cycles

Chart Type Spline
Seconds In View 10
3D Enabled
Line Thickness 3

3D Settings
Inclination
Rotation

Moving Average
Formula None
Period 10

Figure 5.11: cpu utilization of SVOCH at 1 request pr second, measured in percent.



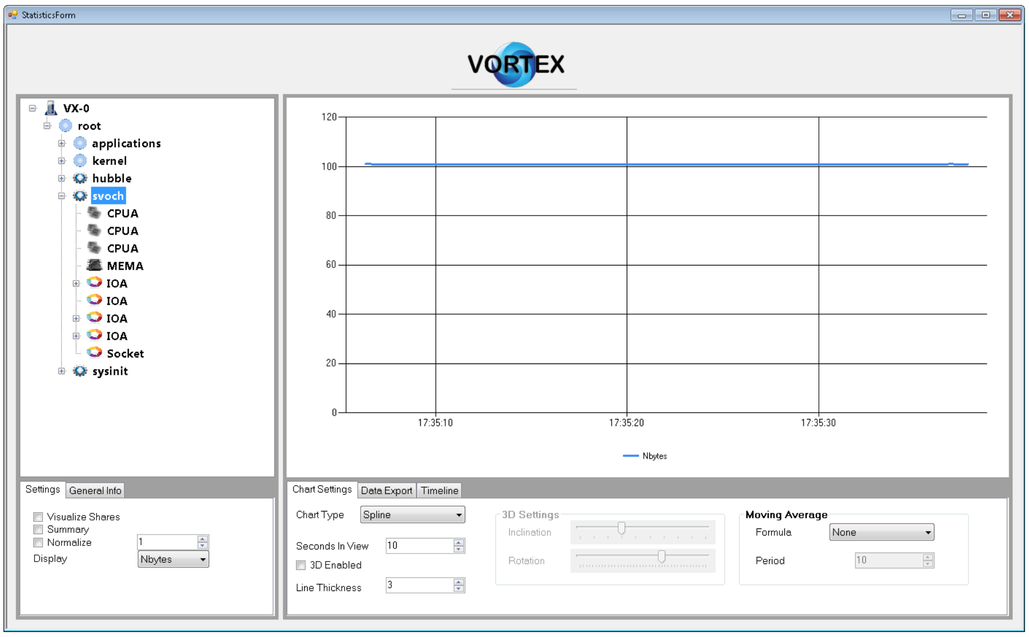
Visualize Shares
Summary
Normalize 1
Display Cycles

Chart Type Spline
Seconds In View 10
3D Enabled
Line Thickness 3

3D Settings
Inclination
Rotation

Moving Average
Formula None
Period 10

Figure 5.12: cpu utilization of SVOCH at 10 requests pr second, measured in percent.

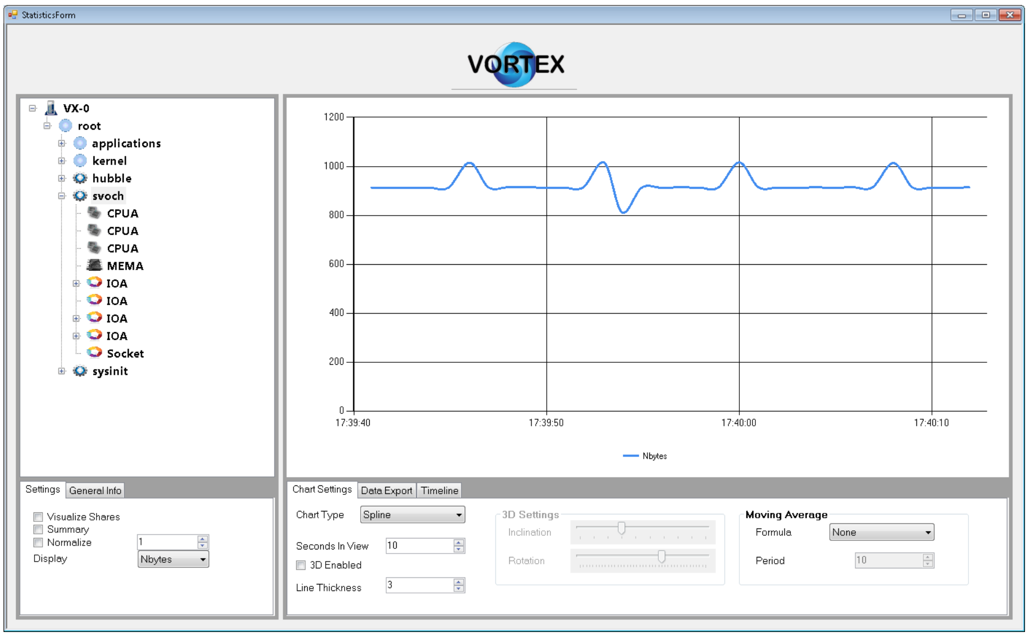


Visualize Shares
Summary
Normalize 1
Display Nbytes

Chart Type Spline
Seconds In View 10
3D Enabled
Line Thickness 3

Moving Average
Formula None
Period 10

Figure 5.13: Bandwidth utilization in Kbit/s of the monitor at 1 request pr second.



Visualize Shares
Summary
Normalize 1
Display Nbytes

Chart Type Spline
Seconds In View 10
3D Enabled
Line Thickness 3

Moving Average
Formula None
Period 10

Figure 5.14: Bandwidth utilization in Kbit/s of the monitor at 10 requests pr second.

Figure 5.11 and Figure 5.12 shows the cpu utilization of SVOCH when running 1 sample per second and 10 samples per second respectively. The graphs show that with respect to cpu consumption, the encryption engine scales linearly with the number of samples per second.

Figure 5.13 and Figure 5.14 shows the bandwidth usage when running 1 sample per second and 10 samples per second, respectively. The bandwidth usage is in the expected range. 12KB of data sent per second equates to approximately 100Kb/s. Again, resource consumption scales linearly with the number of requests per second.

Chapter 6

Conclusion

This thesis has presented the design and implementation of Hubble, an app platform for designing cloud management and analytical apps to aid in administration, deployment and performance monitoring of cloud applications.

6.1 Summary

We have presented Hubble, an app platform for administration and analysis of cloud applications. The platform and all its components, including the apps, encompass 23208 lines of code, which is split evenly between C and C# code.

The platform provides a well formed API for developing apps that can establish secure communication and retrieve performance data about applications running in the cloud.

A monitor component has been designed and implemented to facilitate collection of performance data from instrumentation code running in the cloud.

Further, the platform provides a persistent database for storing structured performance data retrieved from the monitor. The database is designed for efficient inserts and allows query-based retrieval and aggregation of performance data.

Several apps that utilize the API have been designed, implemented, and tested in a Vortex cloud environment. Rover, which is a performance data retriever, enables secure transfer of the collected performance data from the monitor to the database.

Pulsar visualizes different performance characteristics from the performance data found in the database. Pulsar exposes a rich interface that allows the user to choose from a wide variety of different chart plots, normalization, and moving average computations to enhance the visualization.

Uranus is an administration suite that enables the user to create and manage processes, adjust resource allotments, and perform other administrative tasks should the cloud provider support it.

Together, these apps provide a dashboard for assisting in application deployment scenarios, profiling performance characteristics of the deployed applications, and administration of these.

6.2 Discussion

The problem that we set out to solve involved creating an app platform that could facilitate administration and analysis of cloud applications. A platform has been design and implemented, Hubble, and its effectiveness has been evaluated through the implementation of several apps. These apps utilize the platform API to perform different performance analysis and management tasks.

In the following we discuss each requirement presented in Section 1.1 and assess whether the requirement has been met.

6.2.1 First requirement

The first requirement defined for our platform was: *Enable secure communication of performance data produced by cloud instrumentation to an app.*

To satisfy this requirement a monitor component was designed and implemented to facilitate collection of performance data from instrumentation running in the cloud.

In its control interface (see Section 3.5.1), Hubble offers functionality for setting up secure communication channels to the cloud. As described in Section 3.4, Hubble uses the public-key based Secure Shell (SSH) protocol to ensure authenticity, integrity, and confidentiality for client/cloud communication channels. To support the Hubble secure channel functionality for Vortex, we made substantial improvements to previous work (see Section 3.4.1).

6.2.2 Second requirement

The second requirement defined was: *Provide efficient mechanisms for storing and performing query-based retrieval of structured performance data.*

To satisfy this requirement we designed a database that could store structured performance data (see Section 3.5.2). The database was enhanced with indexes to optimize query performance and allow efficient aggregation of performance data.

Apps utilize the Hubble storage interface for persisting new, and querying for existing, performance data.

6.2.3 Third requirement

The third requirement defined for our platform was: *Offer interfaces for aiding apps in performing analytical computations on performance data.*

To satisfy this requirement we developed the analytical interface of Hubble (see Section 3.5.3). This interface provides apps with common analytical functionality such as normalization, aggregation, and moving averages.

The interface is prepared to be extended with large-scale analytical operations that may involve cloud computing resources or local GPU resources for efficient and scalable analysis. Specifically, apps are presented with LINQ enabled identifiers when using both the Hubble storage and analytical interfaces. These allow query execution and data analysis to be deferred until materialization is needed, e.g. when results are visualized or written to disk. Further, LINQ offers mappings to distributed computation frameworks such as Dryad[19].

6.2.4 Fourth requirement

The fourth and last requirement defined was: *Offer interfaces for aiding apps in controlling the cloud environment.*

This requirement is hard to satisfy in a general manner, since different cloud providers provide different interfaces for controlling the cloud environment.

We have to the extent possible, provided mechanisms that allow the fulfillment of this requirement. Apps can utilize the Hubble control interface for remote shell and execute capabilities (see Section 4.3).

To demonstrate that it is possible to dynamically adjust the allotted resources of a cloud environment, we have augmented Vortex with interfaces for adjusting the resources allotted to an existing service. We demonstrated this capability in Section 5.2.

6.3 Future work

The Hubble platform opens up for a wide range of app possibilities. But because of time restrictions and other limiting factors there are still functionality that could be improved or extended to facilitate even more functionality.

The app platform could be extended with APIs for window management. This would make the apps more aesthetic, and provide users of the platform with a familiar user interface across apps. This functionality would also ease development as the developers would not need to worry about window creation and management.

The platform could be further extended with means to deploy apps to the platform, optionally from a central repository or app store. This would allow developers to submit their apps to the central repository and make them available for users to browse and deploy on their platforms.

The monitor has not been thoroughly tested in a multi-node cloud, and we recognize that there is work to be done in the field of aggregating performance data from multiple hosts in an efficient manner.

Our *bigData* interface has a lot of potential, and could be implemented in a way that could facilitate remote cloud computing resources or local GPU resources to perform the actual large-scale analytical job. We have already started planning JVM support on Vortex to be able to offer MapReduce functionality.

References

[1] Beauchamp, T., and Weston, D. DTrace: The reverse engineer's unexpected swiss army knife. In *Blackhat Europe* (2008).

[2] Chao, D. Doom as an interface for process management. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (New York, NY, USA, 2001), CHI '01, ACM, pp. 152–157.

[3] Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J., and Young, P. R. Computing as a discipline. *Commun. ACM 32*, 1 (Jan. 1989), 9–23.

[4] Demers, A., Keshav, S., and Shenker, S. Analysis and simulations of a fair queuing algorithm. In *Proceedings of Special Interest Group on Data Communication* (Austin, Texas, September 1989), pp. 3–12.

[5] Erlingsson, U., Peinado, M., Peter, S., and Budiu, M. Fay: extensible distributed tracing from kernels to clusters. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 311–326.

[6] He, B., Fang, W., Govindaraju, N. K., Luo, Q., and Wang, T. Mars: A mapreduce framework on graphics processors.

[7] HP Cluster Management Utility. http://h20311.www2.hp.com/HPC/ and documentation therein.

[8] Kvalnes, Å., Johansen, D., Valvåg, S., Renesse, R. v., and Schneider, F. Design principles for isolation kernels. Tech. rep., University of Tromsø, 2011.

[9] Li, A., Yang, X., Kandula, S., and Zhang, M. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th annual conference on Internet measurement* (New York, NY, USA, 2010), IMC '10, ACM, pp. 1–14.

[10] Massie, M. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing 30*, 7 (July 2004), 817–840.

[11] Pettersen, R. Public-key based authentication and administration of vortex services. Unpublished Bachelor Thesis, University of Tromsø, 2010.

[12] Ren, K., López, J., and Gibson, G. Otus: resource attribution in data-intensive clusters. In *Proceedings of the second international workshop on MapReduce and its applications* (New York, NY, USA, 2011), MapReduce '11, ACM, pp. 1–8.

[13] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security* (New York, NY, USA, 2009), CCS '09, ACM, pp. 199–212.

[14] Sottile, M. J., and Minnich, R. G. Supermon: a high-speed cluster monitoring system. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on* (2002), pp. 39–46.

[15] IBM Tivoli Monitor. http://www.ibm.com/software/tivoli/ and documentation therein.

[16] Van Renesse, R., Birman, K. P., and Vogels, W. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst. 21*, 2 (May 2003), 164–206.

[17] Vmware vFabric Application Performance Manager. http://www.vmware.com/products/application-platform/vfabric-application-performance-manager/ and documentation therein.

[18] Ylonen, T., and Lonvick, C. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), Jan. 2006.

[19] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Kumar, P., and Currey, G. J. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language.