

Next Generation Multi-Tenant Virtualization Cloud Computing Platform

Buddhika Siddhisena*, Lakmal Warusawithana*, Mithila Mendis**

*Thinkcube Systems (Pvt.) Ltd, Level 10, Access Towers, 278, Union Pl, Colombo 2, Sri Lanka

**Access Telecom Technologies (Pvt.) Ltd, Level 10, Access Towers, 278, Union Pl, Colombo 2, Sri Lanka

bud@thinkcube.com, lakmal@thinkcube.com, mithila@accesslk.com

Abstract— Virtualization and Cloud Computing technologies have gone mainstream as vendors try to reinvent themselves to offer new cloud services. While hardware based virtualization has many benefits, it lacks from a high-level of scalability required to offer cost effective cloud computing to the masses. Multi-tenant virtualization remedies this bottle neck by focusing on software based virtualization. Unfortunately most multi-tenant implementations depend on modifying existing applications to function or require new applications to be developed. In this paper we present a novel approach to build a multi-tenant platform which could run unmodified LAMP applications in a flexible, secure and scalable manner.

Keywords— Multi-tenant, virtualization, cloud computing, SaaS, LAMP platform

I. INTRODUCTION

Virtualization technology has gained popularity as the driving technology behind “Cloud Computing”. The appeal of virtualization is in the ability to run a higher number of virtual machines than what is physically available as hardware servers. This is accomplished by utilizing unused idle cycles present when the system is running at an off peak load, for running several virtual machines thereby maximizing the hardware utilization in a cost effective manner.

Initially virtualization has gained acceptance for its ability to improve efficiency in the data center[1], but has since been re-purposed as an Infrastructure as a Service (IaaS) enabler to a wider consumer base through services such as Amazon's EC2™[2] and Rackspace's Cloud Servers™. However, hardware based virtualization has serious limitations when it comes to scaling to large numbers of concurrent virtual machines[3] running on a given physical hardware server. Hence resulting in a higher per virtual machine instance cost compared to a traditional shared hosting solution.

In contrast Multi-tenant virtualization offers a much more cost effective alternative to hardware virtualization, while maintaining most of its benefits. Multi-tenant virtualization provides a higher degree of concurrent virtual environments by sharing the hardware/software platform in a secure manner to provide isolation between instances. This comes at a cost of

reduced flexibility when compared to hardware virtualization, but more importantly requires applications to be modified or developed from scratch to target a specific multi-tenant virtualization platform[4]. While the loss in Operating System level flexibility is an acceptable trade-off when it comes to offering virtualized web applications, the prerequisite to modify existing applications or to completely build new ones targeted at a specific platform adds an additional overhead to adoption and long term maintenance. Furthermore there is little guarantee that such modifications will not introduce additional bugs or security issues that could compromise the privacy and security of users in a multi-tenant environment.

In this paper we present a novel approach that overcomes the main shortcomings of both hardware and software based virtualization – scalability and flexibility. We will present our findings and ongoing research into building a Multi-tenant virtualization platform, which will be referred hereafter as “Uranus”[5], that focuses on increased scalability and security, while remaining flexible by not requiring applications to be modified in any special way in order to support multi-tenancy.

II. ARCHITECTURE

We have based our virtualization platform, Uranus, on the popular Linux, Apache, Mysql, PHP (LAMP)[6] platform with the aim of virtualizing any of the thousands of existing LAMP based application.

Uranus consists of two main functional layers as shown on figure 1. The “Reverse Proxy Layer” is where the users HTTP requests first encounter the system. This layer consists of one or many reverse proxy servers that route the users HTTP calls to the corresponding back-end LAMP server found in the virtualization layer. These reverse proxy servers are implemented using the Apache Web Server running in a reverse proxy mode. The virtualization layer consists of several LAMP components, each consisting of a separate Apache and MYSQL instance. Apart from these two vital layers, there lies the DNS layer which also plays a key part in routing users traffic to the system. Therefore the DNS system should be configured to resolve domain names of the sites that

are being virtualized to the appropriate reverse proxy server. However the DNS system has been left outside the domain of Uranus for better deployment flexibility.

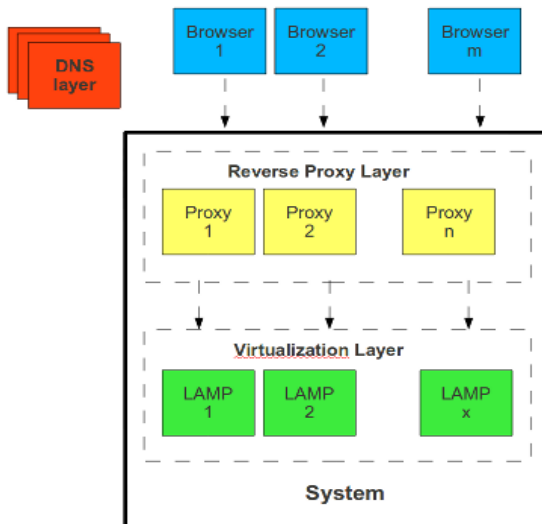


Figure 1. High-level Architecture diagram

A. Reverse Proxy Layer

As previously mentioned, the reverse proxy layer consists of one or many Apache servers running on the standard HTTP port (80) but using a reverse proxy configuration. This is implemented using the Apache modules *proxy*, *proxy-connect* and *proxy-http*[7]. To further enhance the performance and scalability of our system, we also made use of the Apache caching modules *cache* and *mem_cache*. This reduces the number of HTTP requests made to the back-end virtualization layer by caching hot objects at the reverse proxy layer. Snippet 1 shows a typical code snippet of an Apache configuration file from a reverse proxy for an example website *site1.com*. The ProxyPass port, 9101 shown in the snippet is the port of the Apache server within the virtualization layer to which the HTTP request is being handed over to.

```

<VirtualHost *:80>
    ServerName site1.com
    ServerAlias www.site1.com

    ProxyPass / http://site1.com:9101/
    ProxyPassReverse / http://site1.com:9101/

    <IfModule mod_mem_cache.c>
        CacheEnable mem /
        MCacheSize 4096
        MCacheMaxObjectCount 100
        MCacheMinObjectSize 1
        MCacheMaxObjectSize 2048
    </IfModule>

</VirtualHost>

```

Snippet 1. Apache configuration for reverse-proxy

B. Virtualization Layer

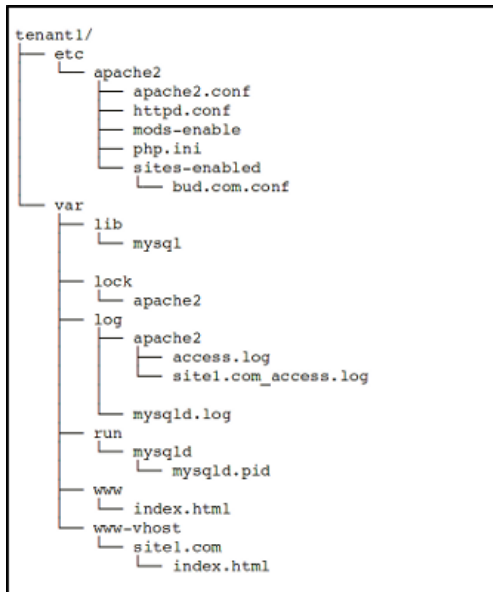
The virtualization layer of Uranus primarily consists of several Apache and MYSQL servers that implement the LAMP stack to which individual LAMP applications would be deployed. This virtualization layer is implemented using a multi-tenant concept where each tenant exists as a separate system user with a dedicated home directory which is shared by the tenant's own Apache and MYSQL processes. But unlike in hardware virtualization where each virtual machine has a complete copy of an OS image including all the applications, in our architecture we share the system's existing application files. So even though we don't share the same application process, we share the physical executable binary files and therefore benefit from being able to update just one system and have all the virtual instances take advantage of it as opposed to having to update several virtual machine systems separately.

The dedicated home directory of a tenant user is only accessible to that specific tenant's processes and all data such as websites, databases, log files are completely isolated in a sandbox like environment. Hence Uranus provides both process and data isolation for a high-level of security. This can be further improved by using an OS level "Mandatory Access Control (MAC)" system such as SELinux[8] or AppArmor[13].

Our current implementation of the file system was accomplished using an LVM partition[14] on the underlying Linux system, with user level quota to limit the use of disk space per tenant. The underlying file system is easily extendible to other file systems, where we have experimented extending it using ZFS[15] on Linux. Unfortunately at the time of writing, there isn't a stable implementation of native ZFS on Linux and the Fuse based approach has poor performance results that would make it acceptable for production use.

1) Apache Virtualization: Each Apache process of a tenant runs on its own unprivileged port as the tenant user. This not only benefits from achieving process isolation but also from being able to customize each process instance based on the required resources as well as being able to prioritize its CPU allocation in a much more granular manner. Besides the raw Apache configuration level customizations, the system also provides PHP level customizations using a custom *php.ini* file per tenant.

Each tenant is capable of running multiple LAMP and static websites within their allocated directory. This was accomplished by replicating a similar directory structure to that of the base OS we use, Ubuntu Linux. Snippet 2 depicts a typical tenant's sand-boxed file system. Some non critical directories were omitted from the diagram due to space constraints.



Snippet 2. Tenant's filesystem

```

<VirtualHost *:9101>
    ServerAdmin webmaster@sitel.com
    ServerName sitel.com
    ServerAlias www.sitel.com
    DocumentRoot /tcmvm/tenant1/var/www-vhost/sitel.com
    <Directory />
        Options FollowSymLinks
        AllowOverride All
    </Directory>
    <Directory /tcmvm/tenant1/var/www-vhost/sitel.com>
        Options Indexes FollowSymLinks MultiViews
        AllowOverride All
        Order allow,deny
        allow from all
    </Directory>

    ErrorLog "/tcmvm/tenant1/var/log/apache2/sitel.log"
    CustomLog "/tcmvm/tenant1/var/log/apache2/sitel.err"
    ServerSignature On
</VirtualHost>

```

Snippet 3. Apache configuration for tenant

Snippet 3 shows a typical Apache configuration for a tenant. There can be several such virtual host configurations per tenant, if they were to run multiple sites using multiple domains.

2) MYSQL virtualization : When virtualizing MYSQL, we took a slightly different approach to that of Apache. Instead of using our own MYSQL management scripts, we relied on the existing `mysqlld_multi` tool[9] for launching multiple MYSQL instances. Functionally it is similar to that of Apache in the sense there is a separate MYSQL process started for each tenant running on its own MYSQL port and having its own database. The main difference is in that the MYSQL process is

started using a common system user called `mysql` and requires a common database user account and password within each tenant's MYSQL database. This common database account is required by the `mysqlld_multi` script in order to have control over shutting down a tenant's MYSQL server instance. While this can be considered by some to be somewhat of a security risk, it is worth mentioning that this administrative account does not have any permission other than to shut-down the database server. So it can not be used to access the tenant's application data. Nevertheless such an account is not absolutely required and offers more of a convenience to managing MYSQL instances centrally, and therefore can be disabled if one wishes.

Snippet 4 shows the configuration changes made to MYSQL in order to support MYSQL virtualization. Each tenant will have a `[mysqlld$tenantID]` section where `tenantID` is an integer number allocated per tenant.

```

[mysqlld_multi]
mysqlld = /usr/bin/mysqlld_safe
mysqladmin = /usr/bin/mysqladmin
user      = multi_admin
password  = aim3MeR6

[mysqlld1]
datadir = /tcmvm/tenant1/var/lib/mysql
socket = /tcmvm/tenant1/var/lib/mysql/mysql.sock
pid-file = /tcmvm/tenant1/var/run/mysqlld/mysqlld.pid
user = mysqlldmulti_admin
port = 7101
server-id=1
log-bin=mysql-bin
log-error=/tcmvm/tenant1/var/log/mysqlld.log

```

Snippet 4. MYSQL configuration setup

When implementing the virtualization layer we had to overcome two main challenges. The first was in getting PHP applications to access MYSQL as they normally would without having to specify a custom MYSQL port or socket file. This was achieved using a custom `php.ini` which specified the MYSQL port and path to the socket file. The second issue appears only with certain types of PHP applications that blindly relied on the returned value of `$_SERVER['HTTP_HOST']` for the correct host name. As the tenant's server is behind a proxy, this returned value is incorrect for our purpose as it contains the custom port. This problem was overcome by instructing the PHP interpreter to automatically include a PHP script file that corrects the problem by replacing `$_SERVER['HTTP_HOST']` with the correct value available via the `$_SERVER['HTTP_X_FORWARDED_SERVER']`. After overcoming these challenges our platform was able to deploy unmodified LAMP applications with success.

III. DEPLOYMENT & TESTING

The design architecture explained above leads to some interesting deployment scenarios ranging from a single standalone setup running on a physical server to a multi-tier

hardware setup or a completely virtualized setup running on an existing IAAS platform.

Figure 2 depicts one possible setup that provides some scalability in terms of growing the number of tenants beyond the initial capabilities of the underlying hardware, providing storage scalability and redundancy using a SAN. The proxy server should be a powerful one with moderate amount of memory/CPU in order to handle the many thousands of concurrent connections and caching of hot objects. For simplicity of design, we have omitted including server redundancy with a fail over mechanism, but this is very much advised for a production grade deployment. We also did not include a DNS server as part of the deployment architecture as we assumed this was an existing functionality within the network.

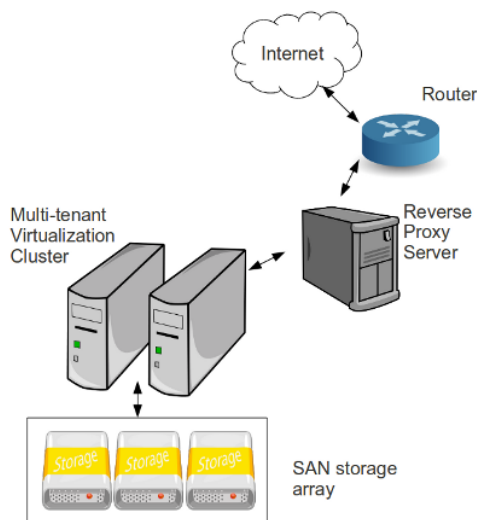


Figure 2. Recommended deployment diagram

We conducted several tests within our lab to determine performance as the number of virtual machines increased. For our test environment we deployed the system on a single server having the hardware/software specifications as shown in Table 1 and Table2.

TABLE 1. HARDWARE SPECIFICATION OF TEST SERVER

Item	Qty	Description
CPU	1	Intel Xeon Processor X3440 Quad-Core 2.53 GHz w 8MB Cache
Memory	4	2GB DDR3
Disk	2	500GB RAID 1
NIC	1	1GB Ethernet

TABLE 2. SOFTWARE SPECIFICATION OF TEST SERVER

Feature	Description
OS	Ubuntu LTS 10.04 Server, 64bit

Swap partition	8GB
Desktop Environment	LXDE
Apache	Version 2.2.14
PHP	Version 5.3.2
MYSQL	Version 5.1.41
Enabled Services (only significant)	• SSH • X-Windows • Apache/MYSQL

A. Testing Rationale

Our main motivation for developing the platform was to be able to scale to tens of hundreds if not thousands of simultaneous LAMP environments on a single physical server. This formed the basis for our test experiment which involved simultaneously executing a large number of Apache/MYSQL instances and benchmarking their performance against an increasing HTTP traffic load.

Apache webserver has a threshold point where the error, as determined by any request taking longer than 5 seconds to respond, will start to increase as the number of simultaneous connections are increased. What we wanted to investigate was whether this threshold point was severely affected as a result of our method of virtualization.

B. Benchmark Testing

Benchmark testing was performed using the popular httpperf[10] and autobench[11] Open Source tools. To simulate a real world LAMP application we used the popular Open Source CMS, Drupal.

For this investigation, we started by benchmarking a single instance of Apache and MYSQL running on the same physical hardware server to obtain a baseline to compare against. This was performed by continuously loading the home page of a default Drupal installation that had just a single article post. The command used to benchmark is shown in snippet 5 and the baseline results can be seen in figure 3. Based on the command in Snippet 5, we send the URL dom0.site.com/drupal a series of HTTP connections starting from a rate of 100 connections/sec, where each has 10 requests to a maximum of 1000/sec connections! The total number of connections is capped at 5000.

```

autobench --single_host --host1 "dom0.site.com" \
--uri1 /drupal --port1 80 \
--quiet --low_rate 100 --high_rate 1000 \
--rate_step 25 --num_call 10 --num_conn 5000 \
--timeout 5 --file results.tsv
  
```

Snippet 5: Command used to test main server

Our next test was to launch 41 virtual instances, each running a process of Apache and MYSQL, where 40 of the instances are idle while the 41st is being stress tested using a similar command to that of Snippet 5. Figure 4 shows the resulting graph. We next proceed to apply a moderate amount

of stress on the 40 virtual Apache processes by using 3 external laptops, each running the httpperf tool, to send a constant flow of HTTP traffic at a rate of 10 requests per second using the command shown in Snippet 6.

```
httpperf --hog --server=site2.com --port=9102 \
--uri=/drupal --rate=10 --num-conns=1000000
```

Snippet 6. Command to send constant traffic to tenant web servers

While this constant traffic is sent, we begin to stress test the 41st Apache process similar to the previous test using a command similar to Snippet 5. Its results are shown in Figure 5.

Finally we perform the first baseline test on a single Apache instance, but this time while receiving constant request rate that was equal to all 40 previous Apache instances for a whopping 400 requests per second. The resulting graph is shown in Figure 6.

The reason we sent the HTTP requests directly to the VM layer, by passing the Proxy layer was to eliminated the effects of caching at the front-end proxy server.

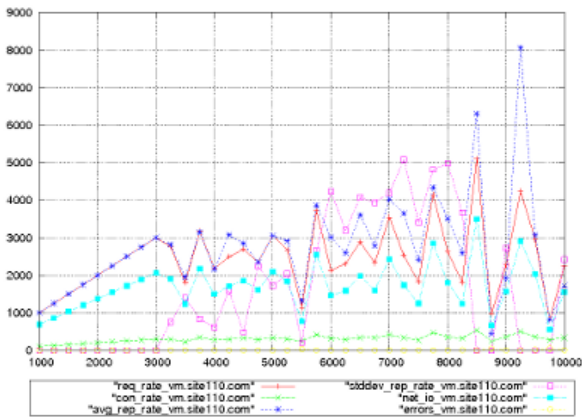


Figure 3. Threshold for single Apache with no other load

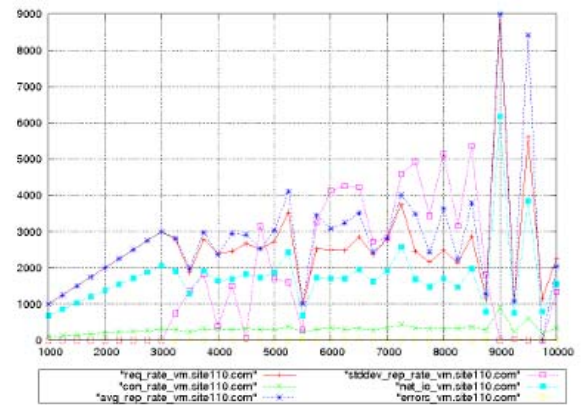


Figure 4. Threshold for single Apache with 40 idle tenants

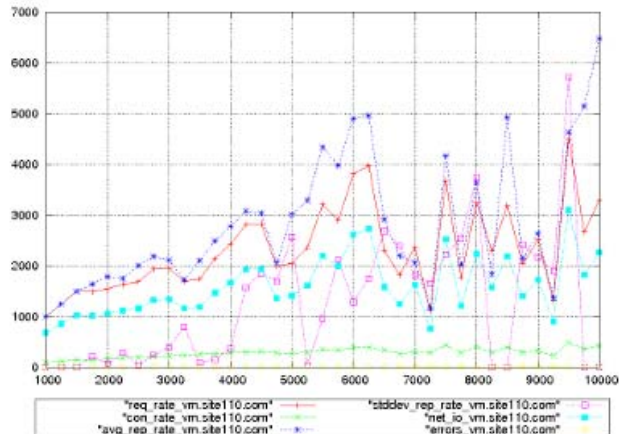


Figure 5. Threshold for single Apache with 40 loaded tenants

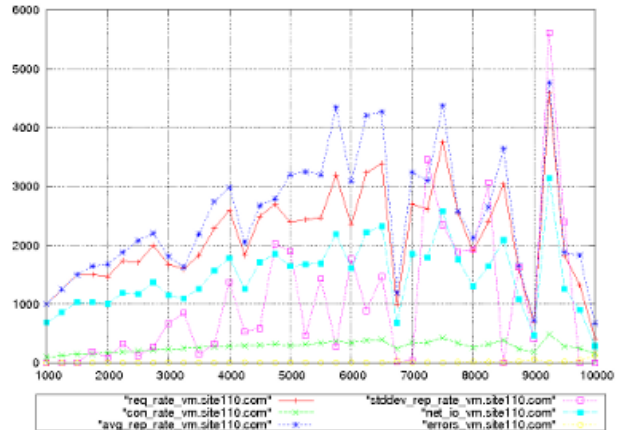


Figure 6. Threshold for single Apache while being constantly loaded

C. Benchmark Results

From the graph shown in figure 4 and figure 5, we can see that regardless of whether we're running a single Apache/MYSQL or 40 idle Apache/MYSQL, there is hardly a difference. In other words the performance hit by using our virtualization method has negligible impact on the system. This is further emphasized in figure 6 and figure 7 when the system is put under constant load. We get a similar threshold value regardless of whether we're benchmarking a single Apache instance having a constant load sent or using 40 instances where the same constant load is equally spread.

Furthermore the system was able to easily cope with a 41 active connections under moderate traffic conditions with less a less than 1.0 Linux load average. Obviously these result can be further improved by having the front-end proxy and optimally tuning Apache for both the proxy and tenants.

IV. CONCLUSIONS

In the paper we presented an approach to application virtualization using multi-tenant concept. Unlike traditional multi-tenant virtualization we stressed the need for a platform that worked with existing applications without the need for any modification. We also focused on security features provided by our platform to safeguard data and isolate processes. Finally we concluded by sharing some benchmark testing results we obtained.

In the near future we will be extending the framework to support HTTPS and we have already completed the necessary proof of concept ground work. We are also looking into increase support for other more recent file systems such as BTRFS[12]. We also hope to conduct much more rigorous testing that cover a higher number of simultaneous tenants and analyzes other aspects such as network and security impact. While the framework is currently based on the LAMP stack, we would like to extend it to other popular web development platforms such as “Ruby on Rails” or “Jango” and even other databases such as PostgreSQL instead of MYSQL within the PHP platform.

ACKNOWLEDGMENT

We would like to thank all our colleagues at Thinkcube and AccessTelecom for extending their fullest support and to our families for their support, understanding and patience.

REFERENCES

- [1] Xue Liu, Xing Jin, WenBo He, QingBo Wang, Ying Chen, "GreenCloud: A New Architecture for Green Data Center", http://portal.acm.org/ft_gateway.cfm?id=1555319&type=pdf, 2009.
- [2] Amazon, "Amazon EC2", <http://aws.amazon.com/ec2>
- [3] Citrix Systems, "Comparison of virtualizing Citrix XenApp 5.0 with XenServer 5.5 on Intel Xeon X7350 & X5570 host servers", http://www.citrix.com/site/resources/dynamic/partnerDocs/XenApp_XenServer_Intel_Xeon_x5570_WP.pdf, 2009
- [4] Chang-Hao Tsai , Yaoping Ruan , Sambit Sahu , Anees Shaikh , and Kang G. Shin, "Virtualization-based Techniques for Enabling Multi-tenant Management Tools," EECS Department The University of Michigan, 2007
- [5] Wikipedia, "Greek mythology - Uranus", [http://en.wikipedia.org/wiki/Uranus_\(mythology\)](http://en.wikipedia.org/wiki/Uranus_(mythology))
- [6] J. Lee, B. Ware. "Open Source Web Development with LAMP", Addison-Wesley, December 2002
- [7] Apache Software Foundation, "Apache mod_proxy documentation", http://httpd.apache.org/docs/2.2/mod/mod_proxy.html.
- [8] Peter A. Loscocco, Stephen D. Smalley, "Meeting Critical Security Objectives with Security-Enhanced Linux", Proceedings of the Ottawa Linux Symposium, 2001.
- [9] Oracle Corp., "mysqld_multi - Manage multiple MySQL Servers", <http://dev.mysql.com/doc/refman/5.0/en/mysqld-multi.html>
- [10] David Mosberger, Tai Jin, "httpperf - A Tool for Measuring Web Server Performance", HP Research Labs, <http://www.hpl.hp.com/research/linux/httpperf/wisp98/httpperf.pdf>, 1998.
- [11] Autobench, <http://www.xenoclast.org/autobench>
- [12] Morgan Janes, "The Progression of Linux File Systems", http://mjan.es.public.iastate.edu/Engl314/indiv_doc.pdf, 2009.
- [13] Ralf Spenneberg, AppArmor, "Protective armor: Shutting out intruders with AppArmor", Linux Magazine, http://www.linux-magazine.com/w3/issue/69/Shutting_out_Intruders_with_AppArmor.pdf, 2006
- [14] Lewis, AJ, "LVM HOWTO", <http://tldp.org/HOWTO/LVM-HOWTO>, 2006
- [15] Wikipedia, "ZFS", <http://en.wikipedia.org/wiki/ZFS>