A Middleware Implementation for a Novel Vertical Cloud Architecture

Gabriel Loewen and Xiaoyan Hong
Department of Computer Science
The University of Alabama
Tuscaloosa, Alabama
{*gloewen@crimson.ua.edu, hxy@cs.ua.edu*}

***Abstract*—One of the core components in building a cloud infrastructure is the development of a middleware solution, which allows for ease in resource management. Middleware solutions for IaaS cloud architectures are responsible for managing the instantiation of virtual machine images, constructing persistent storage volumes, providing node-to-node message passing for metadata retrieval, and more. Our reasoning for developing a new cloud middleware API is to address issues that we have encountered in current cloud middleware solutions, which are centered upon ease of deployment and ease interfacing with the system. Additionally, we have utilized our API to build a novel cloud middleware solution for vertical private cloud architectures. Specifically, this middleware solution is designed for management of compute resources, including instantiation of virtual machine images, construction of storage volumes, metadata aggregation, and other management tasks. In this paper we present a cloud middleware solution that we have developed to be used in building a private vertical IaaS cloud architecture.**

***Keywords*-API; middleware; resource management; IaaS; cloud architecture;**

I. Introduction

Management of resources is a key challenge in the development of a cloud architecture. Moreover, there is a necessity for minimizing the complexity and overhead in management solutions in addition to facilitating attributes of cloud computing, such as scalability, and elasticity. Another desirable quality of a cloud management solution is modularity. We define modularity as the ability to painlessly add or remove components on-the-fly without the necessity to reconfigure any services or systems. The field of cloud management exists within several overlapping domains, which include service management, system deployment, access control management, and others. In this paper we will address the requirements of a cloud management middleware API, which is intended to support the implementation of a private cloud architecture currently in development. Additionally, we compare our cloud management solution to solutions provided by freely available private IaaS cloud architectures.

When examining the current state of the art in cloud management, there are few options. We are confined to free and open source (FOSS) cloud implementations, such as Eucalyptus [1] , and Openstack [2]. Cloud management solutions used in closed-source, and often more popular cloud architectures, such as Amazon EC2 are out of reach from an academic and research perspective due to their closed nature. However, there has been an effort to make these private IaaS cloud architectures compatible with Amazon EC2 by implementing a compatible API and command line tools, such as eucatools [3] and Nova [4], respectively. The compatibility of API's makes it easy to form a basis of comparison between different architectures. Although, this compatibility may also serve as a downfall because if one API suffers from a bug, it may also be present in other API's.

A. *Eucalyptus*

The methodology for management of resources in Eucalyptus is predominantly reliant upon establishing a control structure between nodes, such that one cluster is managed by one second-tier controller, which is managed by a centralized cloud controller. In the case of Eucalyptus, there are five controller types: cloud controller, cluster controller, Block-based storage controller (EBS), Bucket-based storage controller (S3), and node controller. The cloud controller is responsible for managing attributes of the cloud, such as the registration of controllers, access control management, as well as facilitating user interaction through command-line and, in some cases web-based interfacing. The cluster controller is responsible for managing a cluster of node controllers, which entails transmission of control messages for instantiation of virtual machine images, and other necessities required for compute nodes. Block-based storage controllers provide an abstract interface for creation of storage blocks, which are dynamically allocated virtual storage devices that can be utilized as persistent storage. Bucket-based storage controllers are not allocated as block-level devices, but instead are treated as containers by which files, namely virtual machine images may be stored. Node controllers are responsible for hosting virtual machine instances and for facilitating remote access via RDP [5], SSH [6], VNC [7], and other remote access protocols.

B. *OpenStack*

Similar to the methodology used by Eucalyptus, OpenStack maintains a control structure based on the elements present in the Amazon EC2 cloud. However, there is a difference in terminology between Eucalpyuts and OpenStack. OpenStack consists of many “components”, whereas Eucalyptus consist of many “controllers”. The usage of these terms is consistent between the architectures and their meanings are taken to be the same. OpenStack maintains five components: compute component (Nova), object-level storage (Swift), block-level storage (Cinder), networking component (Quantum), and dashboard (Horizon). There are many parallels between the components of OpenStack and the controllers of Eucalyptus. The Nova component of OpenStack is similar to the node controller of Eucalyptus. Similarly we see parallels between Swift in OpenStack with the bucket-based controller in Eucalyptus, and Cinder in Openstack with the block-based storage of Eucalyptus. There seems to be a discretion in implementation between the highest level component in each architecture. OpenStack maintains a different component for interfacing and network management, while Eucalyptus maintains a single cloud controller, combining these functionalities. Additionally, OpenStack does not maintain a higher-level control structure for managing compute components, which is a deviation from the cluster controller mechanism present in Eucalyptus.

In the following sections we present our middleware solution for management of cloud resources in a private IaaS cloud architecture currently in development. Section II presents related work in cloud middleware and resource management solutions. Section III discusses the communications structure used for inter-node message passing. Section IV introduces our middleware API and presents a comparison of our solution with Eucalptus and OpenStack. Section V discusses interfacing with the API and presents an example usage in web interfacing with PHP. Sections VI and VII discusses the ability to support distributed services. in Section VIII we discuss our initial performance results by comparing latencies between our approach in authentication and the traditional approach with SSH. Section IX discusses future work and conclusion.

II. Related Work

The authors of [8] present their solution, SQRT-C, which is a light weight and scalable resource monitoring and dissemination solution using the publisher/subscribe model. The SQRT-C solution is designed specifically as a means to improve quality of service in real time systems. The approach considers three major design implementations as top priority: Accessing physical resource usage in a virtualized environment, managing data distribution service (DDS) entities, and shielding cloud users from complex DDS QoS configurations. SQRT-C can be deployed seamlessly in other cloud platforms such as Eucalyptus and OpenStack since it relies on the libvirt library for information on resource managment in the IaaS cloud.

In [9], the authors begin to build a cloud computing architecture for service oriented software development. Their SOARWare tool is used for software development, but users still need to install various tools on their local machine for development. MyCloud and App Engine provide the platform and interface for developers to begin work quickly on service oriented architecture projects.

In [10], the authors proposed a middleware for enterprise cloud computing architectures, which can automatically manage the resource allocation of services, platforms, and infrastructures. The middleware API set used in their cloud is built around servicing specific entities using the cloud resources. For end users, the API toolkit provides interaction for requesting services. These requests are submitted through a web interface. Internal interface APIs communicate between physical and virtual cloud resources to construct interfaces for users and determine resource allocation. A service directory API is provided for users based on user privileges. A monitoring API is used to monitor and calculate the use of cloud system resources.

The authors of [11] proposed a resource manager that handles users requests for virtual machines in a cloud environment. Their architecture deploys a resource manager and a policy enforcer module. First the resource manager decides if the user has the rights to request a certain virtual machine. If the decision is made to deploy the virtual machine, the policy enforcer module communicates with the cloud front-end and executes an RPC procedure for creating the virtual machine.

Authors of [12] describe how cloud platforms should provide a rich set of services on-demand that help the user complete their job quickly. Also mentioned is the cloud’s responsibility of hiding low-level technical issues such as hardware configuration, network management, and maintenance of guest and host operating systems. The cloud should also reduce costs by using dynamic provisioning of resources, consuming less power to complete jobs (within the job constraints), and by keeping human interaction to cloud maintenance to a minimum. We keep these thoughts in mind as we develop the middleware API for our IaaS cloud architecture.

Developing cloud APIs are discussed in [13]. The author mentions three goals of a good cloud API: Consistency, Performance, and Dependencies. Consistency implies the guarantees that the cloud API can provide. Performance is relatively considered in forms of decreasing latency while performing actions. Cloud dependencies are other processes that must be handled other than spawning virtual machines and querying cloud resource and user states. These three issues are considered in the development process of our own IaaS cloud architecture.

III. Communication Scheme

In contrast to the methodologies used by Eucalyptus, OpenStack, and presumably Amazon, our cloud middleware API addresses resource management in a simplified and more direct manner. The hierarchy of controllers used in Eucalyptus introduces extra complexity that we have deemed unnecessary. For this reason, our solution utilizes a simple publisher/subscriber model by which compute, storage, and image repository nodes may construct a closed network. The publisher/subscriber system operates in conjunction with event driven programming, which allows events to be triggered over the private network to groups of nodes subscribed to the controller node. Figure 1 shows the logical topology and lines of communication constructed using this model. In

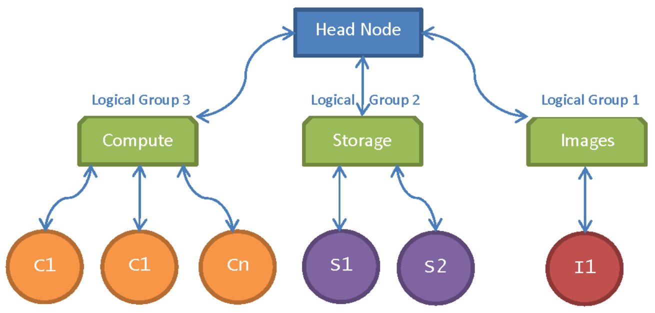


Figure 1. Logical topology - logical groups represent group-wise membership in publisher/subscriber model

constructing the communication in this manner we are able to broadcast messages to logical groups in order to gather metadata about the nodes subscribed to that group. Message passing is useful for retrieving the status of nodes, including virtual machine utilization, CPU and memory utilization, and other details pertaining to each logical group. Additionally, we are able to transmit messages to individual nodes in order to facilitate virtual machine instantiation, storage allocation, image transfer, and other functions that pertain to individual nodes.

A. *Registration of Nodes*

Communication between nodes utilizes non-persistent socket connections, such that the controller node maintains a static pre-determined port for receiving messages, while other nodes may use any available port on the system. Thus, each node in the cloud, excluding the controller node automatically selects an available port at boot time. Initial communication between nodes is done during boot time to establish a connection to the controller node. We utilize a methodology for automatically finding and connecting to the controller node via linear search over the fourth octet of the private IP range (xxx.xxx.xxx.0 to xxx.xxx.xxx.255). Our assumption in this case is that the controller node will exist on a predefined subnet, which allows us to easily establish lines of communication without having to manually register nodes. Once a communication link is established between a node and the controller node, the node will request membership within a specific logical group, after which communication between the controller node and that logical group will contain the node in question.

The registration methodology used in our middleware solution differs from the methodology used by Eucalyptus and OpenStack. For example, Eucalyptus relies upon command line tools to perform RSA keysharing and for establishing membership with a particular controller. We do not perform key sharing, and instead rely upon a pre-shared secret key and generated nonce values. This approach is commonly known as challenge-response, and it ensures that nodes requesting admission into the cluster are authentic before communication is allowed. When a node wishes to be registered as a valid and authentic node within a cluster a nonce value is sent to the originating node. The node will then encrypt the nonce with the pre-shared key and transmit the value back to the controller. We validate the message by comparing the decrypted nonce produced by the receiver and the nonce produced by the sender. Thus, we do not rely upon manual sharing of RSA keys beforehand, and instead we eliminate the need for RSA keys altogether and utilize a more dynamic approach for validation of communication during the registration process. Figure 2 presents the registration protocol.

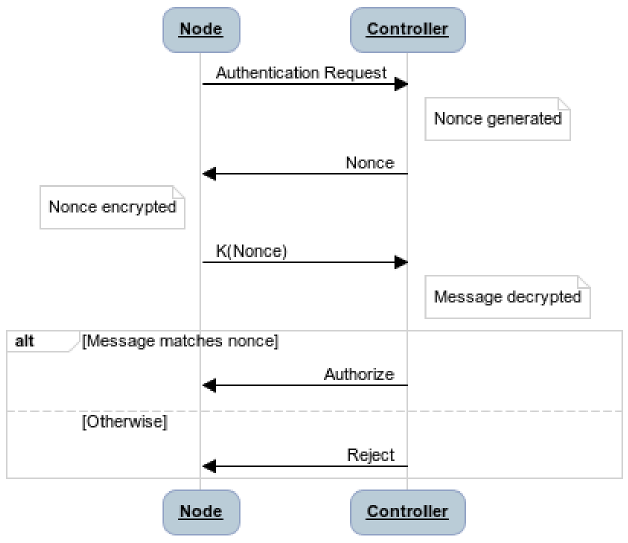


Figure 2. Authentication protocol

IV. Middleware API

As stated in the introduction, our methodology for constructing a middleware API for cloud resource management centers around the notion of simplicity. In order to facilitate a simple middleware solution, our API was designed to provide a powerful interface for cloud management while not introducing excessive code overhead. We have titled our API “NetEvent”, which is indicative of it’s intended purpose as an api for triggering events over a network. This API is utilized within our private IaaS cloud architecture as a means for communication, management of resources, and interaction with our cloud interface. Figures 3 and 4 illustrate the manner in which the API is accessed. Although, the code examples presented here are incomplete, they illustrate the simplicity of creating events to be triggered by the system for management of resources.

```
from NetEvent import *

def main():
  # Instantiate NetEvent object
  # Listen on port 6667 with a
  # role of ADMIN
  netEvent = NetEvent(6667, 'ADMIN')

  # Register an event for invoking commands
  netEvent.registerEvent('INVOKE', invoke)

# Invoke a command within the system.
# Allows for the retrieval of information
# and triggering of events in other nodes.
def invoke(params):
  if (params[0] == 'GET_CLIENT_LIST'):
    return netEvent.getClientList()
  elif (params[0] == 'GROUP'):
    groupName = params[1]
    event = params[2]
    res = netEvent.publishToGroup(groupName, event)
    return res
  else:
    # ...

main()
```

Figure 3. Example controller node service.

In Figure 3 we present sample code for the creation of a controller node, which is responsible for issuing commands to the rest of the cloud. The controller node establishes a special event, which is triggered by the cloud interface for gathering data and interacting with the resources. In Figure 4 we present a skeleton for the creation of a compute node with events written for instantiation of virtual machine images and for retrieving the status of the node. Although, we do not present code for the implementation of storage or image repository nodes, the implementations are similar to that of the compute node. In addition, the code examples presented in this paper show only a subset of the functionality contained within the production code.

The API presented in Figures 3 and 4 provides a powerful interface for implementing private cloud architectures. By means of event triggering over a private network we are able to instantiate virtual machine images, mount storage volumes, retrieve node status data, transfer virtual machine images, monitor activity, and more. The implementation of the system is completely dependent upon the developer's needs and may even be used in distributed systems, which may or may not be implemented as a cloud architecture.

```
from NetEvent import *
import libvirt

def main():
  # Create NetEvent object
  netEvent = NetEvent()

  # Register events
  netEvent.registerEvent('INSTANTIATE', instantiate)
  netEvent.registerEvent('STATUS', status)

  # Set the logical group name
  netEvent.associateGroup('COMPUTE')

  # Find and connect to controller node
  netEvent.findController()

# Instantiate virtual machine, return
# private IP address of instance
def instantiate(params):
  # params[0] -> name of image
  ...
  return ip_addr

main()
```

Figure 4. Skeleton for compute node service.

V. Interfacing

In Section IV we introduced our middleware API for managing cloud resources. However, another important component is a reasonable way to interface with the middelware solution. Although, the middleware API solution is completely independent from the interface, we have chosen to use a message passing approach. In this approach our web interface, which is written in PHP connects to the controller node in order to trigger the "INVOKE" event (see fig. 3). By interfacing with the controller node we are able to pass messages to groups, or individual nodes in order to manage the resources of that node and receive responses. The ability to interface in this manner allows our interface to remain decoupled from the logical implementation, while allowing for flexibility in the interface and user experience. Figure 5 shows an example PHP script for interfacing with the resources in the manner described in this section.

The PHP interface presented in Figure 5 illustrates the methodology behind how we may capture and display data about the nodes, as well as provide a means for user interaction in resource allocation and management. Although, we do not present the full source code in this paper, additional functions could be written. For example, a function could be written, which instruct compute nodes to instantiate a particular virtual machine image. One important aspect of this system is that at all levels the mode of communication remains consistent. Every message sent is implemented via non-persistent socket connections. This allows for greater data consistency without modifying the semantics of messages between the different systems. Figure 6 shows an example interface for metadata aggregation of a logical compute group.

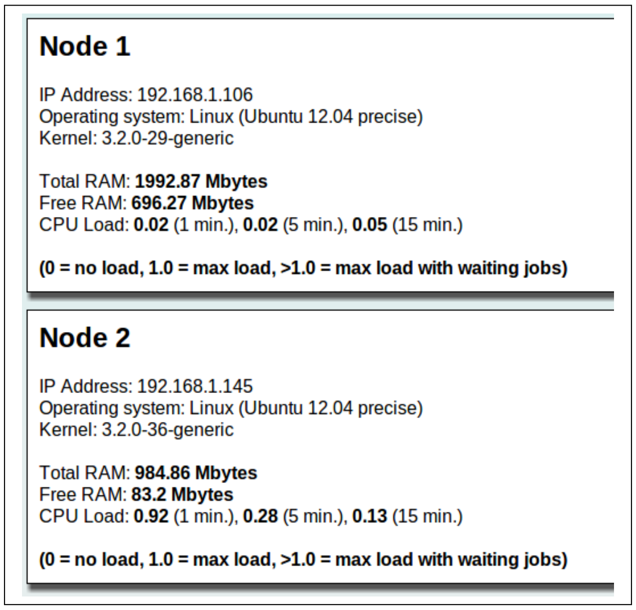
```
class Communication {
  var $sock;

  /* Insert socket connect/send/receive code here */

  function getStatus($group) {
    $this->connect();
    $this->send('INVOKE GROUP ' .
                $group .
                ' STATUS');
    $response = $this->receive();
    $this->close();
    if ($response == '') {return array();}
    return explode('\;',$response);
  }

  function getClusterList() {
    $this->connect();
    $this->send('INVOKE CONTROL GET_CLUSTER_LIST');
    $response = $this->receive();
    $this->close();
    if ($response == '') {return array();}
    return explode('\;',$response);
  }
}
```

Figure 5. Example communication interface in PHP.



Node 1

IP Address: 192.168.1.106
Operating system: Linux (Ubuntu 12.04 precise)
Kernel: 3.2.0-29-generic

Total RAM: **1992.87 Mbytes**
Free RAM: **696.27 Mbytes**
CPU Load: **0.02** (1 min.), **0.02** (5 min.), **0.05** (15 min.)

(0 = no load, 1.0 = max load, >1.0 = max load with waiting jobs)

Node 2

IP Address: 192.168.1.145
Operating system: Linux (Ubuntu 12.04 precise)
Kernel: 3.2.0-36-generic

Total RAM: **984.86 Mbytes**
Free RAM: **83.2 Mbytes**
CPU Load: **0.92** (1 min.), **0.28** (5 min.), **0.13** (15 min.)

(0 = no load, 1.0 = max load, >1.0 = max load with waiting jobs)

Figure 6. Interface showing metadata aggregation of a logical compute group

VI. Supporting Compute Services

The NetEvent API allows for services to be written and distributed to nodes within a private cluster. These services utilize the NetEvent API as a means for triggering events remotely. Within cloud architectures there are a few important events that must be supported. Firstly, the instantiation of virtual machine images must be supported by all cloud architectures. Compute services may be supported by combining the flexibility of the NetEvent API and a hypervisor, such as KVM. A proper compute service should maintain an image instantiation event, which invokes the hypervisor and instructs it to instantiate a specific virtual machine image. Figure 7 shows an example of an event for instantiation of virtual machine images.

```
def instantiate(params):
  vmName = params[0]
  conn = libvirt.open("qemu:///system")
  if (conn == None):
    return -1

  try:
    vm = conn.lookupByName(vmName)
    vm.create()
  except:
    return -1

  # Parse the MAC address from the virtual machine
  # connection and then use ARP to get the IP address
  return getIPAddr(vm)

main()
```

Figure 7. Implemented instantiation event

VII. Supporting Metadata Aggregation

Metadata aggregation refers to the ability to retrieve data about each node within a specific group. This data may be used for informative purposes, or for more complex management tasks. Example metadata include the nodes IP address, operating system, and kernel. Additionally, dynamic data may be aggregated as well, including RAM availability and CPU load. We can support metadata aggregation in each service by introducing events that retrieve the data and transmit it to the controller node. Figure 8 presents the implementation of metadata retrieval events. Note that the return value for the events is comprised of colon separated values, which are passed by the NetEvent API to the service requesting the data.

VIII. Performance Results

One of the many reasons for not using SSH, which seems to be the industry standard approach for inter-node communication in general purpose cloud architecture, is that SSH produces excessive overhead. The communication approach used by NetEvent is very simplified, and does not introduce data encryption or a lengthy handshake protocol. The downfall of simplifying the communication structure is that the system becomes at risk for loss of sensitive data being transmitted between nodes. However, in the case of this system no sensitive data is ever transmitted, and instead only simple commands are ever sent between nodes. For this reason encryption is unnecessary. However, authentication is still required in order to determine if nodes are legitimate. In testing the performance of NetEvent we compared the elapsed time for authenticating a node with the controller and establishing a connection with the elapsed time for SSH to authenticate and establish a connection. We gathered data

```
def  utilization (params):
    # Get memory utilization
    memInfo = open('/proc/meminfo','r')
    memtotal = memInfo.readline(). split (':')[1]. strip (). split () [0]
    memfree = memInfo.readline(). split (':')[1]. strip (). split () [0]
    memInfo.close()

    # Get CPU load averages
    loadAvg = open('/proc/loadavg', 'r')
    data  = loadAvg. readline (). split ()
    loadAvg. close ()
    oneMin = data[0]
    fiveMin  = data [1]
    fifteenMin  = data [2]

    retVal  = memtotal + ":" + memfree + ":" + oneMin + ":" + fiveMin + ":
        ↪" + fifteenMin
    return  retVal

def  sysInfo (params):
    fullOsName = ''
    for  i  in  platform . dist () :
        fullOsName += i + '  '
    fullOsName = fullOsName[:−1]
    shortOsName = os.uname()[0]
    kernel  = os . uname()[2]
    arch  = os . uname()[4]

    retVal  = shortOsName + "  (" + fullOsName + "):" + kernel + ":" + arch
    return  retVal
```

Figure 8.   Metadata aggregation events

Trial #	SSH (ms)	NetEvent (ms)
1	298	3.1
2	301	3.2
3	302	9.2
4	298	3.1
5	299	3.0

Table I
ELAPSED TIME COMPARISON FOR AUTHENTICATION BETWEEN SSH AND NETEVENT

over five trials, which is presented in Table I. Additionally, Figure 9 presents the average latencies between SSH and NetEvent.

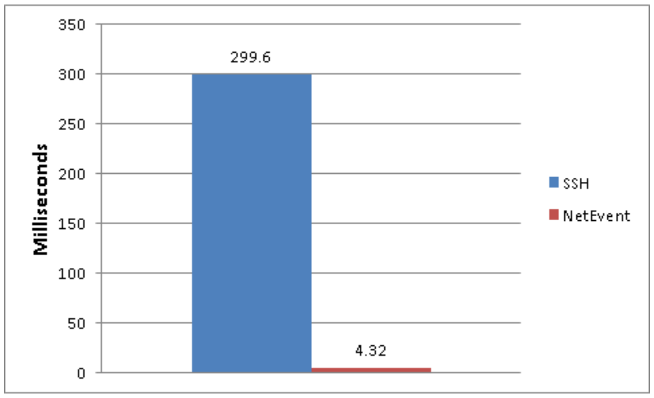


Figure 9.   Average elapsed time for authentication between SSH and NetEvent

From the performance comparison we draw the conclusion that general purpose cloud architecture that utilize SSH connections, such as Eucalyptus, sacrifice up to a 99% loss in performance when compared to traditional sockets. However, this comparison is being made at optimal conditions, because the servers are under minimal load. More data needs to be gathered to determine how much the performance is affected when the servers are under load.

IX. Future Work and Conclusions

One of the key challenges in building a cloud infrastructure is the development of a middleware solution, which allows for ease in resource management. The work presented in this paper demonstrates that a middleware solution does not have to be as complex as those found in the popular cloud architectures, Eucalyptus and Openstack. In Sections I, IV and V we introduce the model by which our API offers communication between nodes, namely utilizing event driven programming and socket communication. We have developed our API to be efficient, light weight, and easily adaptable for the development of vertical cloud architectures. Additionally, we show the manner in which a web interface may interact with the middlware API in order to send messages and receive responses from nodes within the cloud. For future work we would like to investigate approaches for fault tolerance in this architecture. Additionally, we would like to perform an overall system performance benchmark and make comparisons between other cloud architectures.

References

[1] Eucalyptus Systems. (2013, Feb.) Eucalyptus cloud. [Online]. Available: http://www.eucalyptus.com/

[2] OpenStack Foundation. (2013, Feb.) Openstack cloud software. [Online]. Available: http://www.openstack.org/

[3] Eucalyptus Systems. (2013, Feb.) Ec2 tools. [Online]. Available: http://www.eucalyptus.com/eucalyptus-cloud/tools/ec2

[4] OpenStack Foundation. (2013, Feb.) Openstack nova. [Online]. Available: http://nova.openstack.org/

[5] L. Surhone, M. Timpledon, and S. Marseken, *Remote Desktop Protocol*. VDM Publishing, 2010.

[6] D. J. Barrett, R. E. Silverman, and R. G. Byrnes, *SSH, The Secure Shell: The Definitive Guide*. O'Reilly Media, 2005.

[7] T. Richardson, "The rfb protocol," Tech. Rep., Nov 2010. [Online]. Available: http://www.realvnc.com/docs/rfbproto.pdf

[8] K. An, S. Pradhan, F. Caglar, and A. Gokhale, "A publish/subscribe middleware for dependable and real-time resource monitoring in the cloud," in *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management*, ser. SDMCMM '12. New York, NY, USA: ACM, 2012, pp. 3:1–3:6. [Online]. Available: http://doi.acm.org/10.1145/2405186.2405189

[9] H. Sun, X. Wang, C. Zhou, Z. Huang, and X. Liu, "Early experience of building a cloud platform for service oriented software development," pp. 1 –4, sept. 2010.

[10] S.-Y. Lee, D. Tang, T. Chen, and W.-C. Chu, "A qos assurance middleware model for enterprise cloud computing," in *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*, july 2012, pp. 322 –327.

[11] E. Apostol, I. Baluta, A. Gorgoi, and V. Cristea, "Efficient manager for virtualized resource provisioning in cloud systems," in *Intelligent Computer Communication and Processing (ICCP), 2011 IEEE International Conference on*, aug. 2011, pp. 511 –517.

[12] Y. Khalidi, "Building a cloud computing platform for new possibilities," *Computer*, vol. 44, no. 3, pp. 29 –34, march 2011.

[13] B. Cooper, "The prickly side of building clouds," *Internet Computing, IEEE*, vol. 14, no. 6, pp. 64 –67, nov.-dec. 2010.