A View of WSN-facilitating Application's Design and a Cloud Infrastructure in Academic Environment and Research

Alexander Novinskiy
Fb2, Informatik und Ingenieurwissenschaften
Frankfurt University of Applied Sciences
Frankfurt am Main, Hessen
Email: a.novinskiy@fb2.fra-uas.de

***Abstract*— This paper presents a view of seamless integration of Wireless Sensor Networks and server-side applications working with sensor data. The aim of this paper is to show a vector toward organizing a flexible and scalable platform for educational and research institutions or small organizations which demand privately running "Internet of Things" solutions when having restricted computational resources.**

I. Introduction

During the recent years the area of Internet of Things has arisen and gained increasing attention. Being facilitated by the use of multiple wireless network protocols such as ZigBee, Bluetooth, wirelessHART and sensor technologies IoT is gradually penetrating into humans lives. The interconnection of two concepts, Internet of Things and Wireless Sensor Networks, made them almost interchangeable. Wireless Sensor Networks significantly differ from traditional networks. First, communication goes via radio based signal exchange. Therefore, concepts such as operational frequency, number of channels available, range and power consumption go into the first plan when designing or choosing appropriate hardware solutions for your network. The need to operate long in the autonomous mode imposes multiple restrictions to hardware. First of all it must be power efficient. That led to the dramatical degradation of computational capabilities compared to nodes of traditional networks with permanent power supply. Having that limited computational power WSN-nodes are being designed to be able to read sensor data, transmit and receive signals via wireless networks. Most of the computationally intensive tasks such as statistical analysis, numerical data processing, decision making routines moved to the regular networks. That made engineers think about the architectural design of wsn-driven applications. Having omitted basic client/server approach developers and researchers came to the use of publish/subscribe architecture. This approach allowed software developers to integrated as many WSNs and applications as they needed into a single communicational system, thus, increasing scalability and simplifying development process. Nevertheless, economical needs made researchers and software developers seek for a better way of resources utilization and scalability possibilities, higher level of informational security and easier ways of software deployment. It has been 10 years since the concept of Cloud Computing had come into life and brought significant opportunities for developers and end-users. This paper covers how cloud computing core technologies and publish/subscribe communicational protocols make a suitable platform for Internet of Things solutions.

II. Problem definition

Due to the gaining popularity and promising prospects of the Internet of Things concept many organizations, companies and universities work in this area to deliver prototypes and/or complete solutions for home automation [1], weather monitoring, human behaviour recognition, motion tracking and many other areas of human life. Nevertheless, research institutions and universities have to deal with highly restricted financial, infrastructural and therefore computational resources. Being supported by highly volatile teams consisting of students, research assistants and simply enthusiasts these institutions deal with a great variety of approaches toward architectural design which are no longer supported after the lifespan of a project has come to its end. In order to manage risks caused by these specifics universities and research institutions need to be able to utilize available computational resources at its maximum and offer a simple and unified approach toward software design which would facilitate both software modules communication and seamless integration of WSNs with server-side business logic.

III. Communicational System Design

A. Naive approach

As long as most data processing jobs take place on the server side due to restrictive nature of devices comprising WSNs it has become a major issue to find out a way to organize data exchange between sensor nodes and server-side applications. Since most WSN solutions are being driven by network protocols different from those facilitated by the TCP/IP protocol stack such as Wi-Fi the concept of a gateway arose. Gateways are devices which establish communication between different media and network protocols. In the most

straight-forward approach a gateway establishes a direct connection with a server-side application and exchange data via a TCP/IP socket. Server side applications accept incoming data and store it in databases.

Nevertheless, as the amount of projects increases this approach shows its major disadvantages. Server-side applications and corresponding clients are strongly coupled. Having established a socket connection server-side applications bind themself to a certain port number. In this case replication of the server would require significant administrative efforts. Furthermore, real-time data access is being provided for a single application only. Could we need to add an alternative business logic to process the same data in real-time we might have to multicast these data among different server-side applications. That would make software development process more tangled and intolerant to architectural changes.

The use of databases on the other hand would destroy advantages of real-time communication which is crucial for many applications.

B. Publish/subscribe-based approach

The scale of distributed systems has dramatically increased with the development of Internet and networks in general. The publish/subscribe communicational model is dedicated to provide a very loosely coupled form of interaction for entities of distributed systems. The idea is that subscribers can register their interest in a particular event or a pattern of events, when publishers generate events on a software bus or an event manager [2], which are being asynchronously propagated to subscribers according to their interest. Many industrial systems and research prototypes support this style of interaction. Regardless the great variety of these systems all of them aim to achieve time, space, and synchronization decoupling of subscribers and publishers.

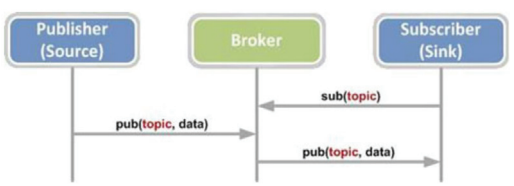

Fig. 1. Publish/Subscribe architecture

Communicational parties do not need to actively participate in data exchange. In fact, publishers can send their data even when subscribers are disconnected. And on the contrary, if a publisher has disconnected after the message was sent, corresponding subscribers will get notified about the event and perhaps about the disconnection of the publisher. This property is called time decoupling - something that we can not achieve with pure client/server architecture. Synchronization decoupling means that publishers are not blocked when generating an event and subscribers can get asynchronously notified while performing some concurrent activity. Moreover, participants of interaction do not have to know each other. This is called space decoupling which is achieved by the use of communicational middleware - a service responsible for messages redistribution.

By removing all unnecessary dependencies between communicational parties we can increase overall scalability of our systems. More information about publish/subscribe communicational paradigm can be found in the article ”Many faces of publish/subscribe” [3].

Nevertheless, an IoT facilitating feasible publish/subscribe system should meet following requirements.

- It should be lightweight in order to intercommunicate with tiny devices comprising WSNs.
- It should be reliable, but yet rapidly operating.
- It should be simple to learn and to use and yet be extendable. Rapid prototyping requires low entry costs for developers, but there must be room for improvements.
- It should be supported by a wide community of software developers from different application areas.
- It should provide security mechanisms such as authentication of clients and prevention of unauthorized access to published data.
- It should be scalable to support high scale deployment.

C. MQTT

MQTT stands for Message Queue Telemetry Transport and is a ”light weight” application layer communicational protocol based on publish/subscribe model. Even though, there is a great variety of publish/subscribe model implementations, MQTT is definitely one of those which hold many of properties desired by IoT developers [4] [5].

The MQTT protocol relies on communicational middleware called a message broker, which is responsible for orchestration of communicational flows. MQTT has a modification for smart sensor networks called MQTT-SN or MQTT-S [6]. It introduces a concept of a gateway - a framework which, when used, is responsible for data transfer between a smart sensor network and a message broker. It is usually being located on a physical gateway which is traditionally represented as a tiny device interconnecting different media and protocols.

MQTT supports basic end-to-end quality of service. The simplest one is the QoS level 0. It offers the best effort service where a message is either delivered once or not at all. Retransmission or delivery acknowledgement is not defined. QoS level 1 ensures delivery of a message but the message can be delivered more then once due to retransmission. QoS level 2 is the highest one and it ensures messages are being delivered exactly once.

MQTT supports username/password authentication and is capable of authorizing users to provide specified access rights to published data. Furthermore, MQTT clients can still transmit encrypted payload via SSH/TLS tunnels or by means of custom encryption [7].

MQTT is easy to use. It provides ”CONNECT” function to connect to a message broker, ”PUBLISH” function to publish sensor data or user controls, and ”SUBSCRIBE” function to register clients interest in certain data. Yet, many

broker implementations provide plugins and APIs to extend the default behaviour.

Some MQTT brokers support bridging - direct broker-to-broker connections, which makes scaling easier and provides higher level of flexibility.

IV. Cloud infrastructure

A. Motivation

There are multiple reasons to make use of ”clouds” in research projects and prototyping tasks. Among them are lower cost of entry, reduced risk of IT infrastructure failure, higher Return On Investment (ROI), quick responses to changes on demand, rapid deployment, increased security, and ability to focus on the core business of an organization [8]. By making use of cloud environment one can increase efficiency of resources utilization in comparison with the bare metal deployment. It can be achieved via the core cloud-comprising technology called virtualisation. Rather then having only one bare metal machine we can have several virtual ones with a certain amount of allocated resources. Moreover, we can save significant amount of administrative efforts as long as cloud infrastructure supports high level of automation.

B. General cloud architecture

A cloud infrastructure can be created based on computer clusters with the help of special orchestrating software. OpenStack and Ecalyptus are examples of such systems.

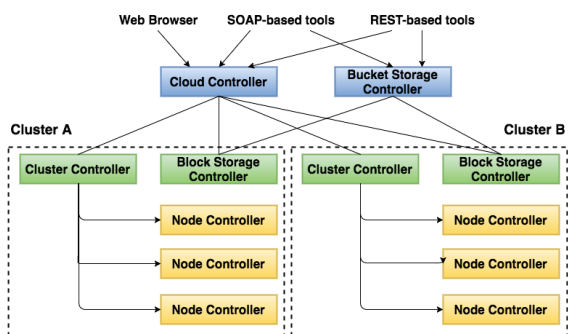

Fig. 2. Eucalyptus architectural layers and advanced setup according to Z. Pantic et al. [8]

A conventional cloud computing orchestration software system contains modules to elicit resources usage statistics, manage virtual instances, control available storage and provide a user interface. Cloud computing system of that level is called an Infrastructure as a Service and it aims to provide isolated computational resources on demand.

1) Node controller: This module is responsible for ”instances” - virtual machines with operational systems on them. Each node with a node controller installed may have zero, one or many simultaneously running instances with possible different OS on them. Node Controllers communicate with the hyper-visor running on the node, host operational system and with the corresponding Cluster Controller. It gathers data about utilization of available resources and about instances running on that node. [8]

2) Cluster controller: This module gathers information about nodes, resources available on them and about running instances by communicating with corresponding Node Controllers. It also communicates with the Cloud Controller propagating information about NCs to it, receiving requests for deploying instances, and deciding about where they should be put. It is also responsible for virtual networks available to instances.

3) Block level storage controller: Instances on the cluster level may need to get access to storage volumes just as any OS be it deployed on the bare-metal or inside of virtual machine requires storage devices to keep its files and programs on it. A conventional IaaS platform is supposed to encapsulate block level storage access as an abstraction from actual storage devices. Eucalyptus does it via ATA over Ethernet (AoE) or Internet SCSI (iSCSI) protocol to mount virtual storage devices. [8]

4) Bucket-based storage controller: This module is responsible for managing a put/get storage model (create and delete buckets, create objects, put or get those objects from buckets). [8]. This kind of service is being used to keep machine images and snapshots.

5) Cloud Controller: This is the top level service module and is an entry point to the cloud. It provisions both end-user and administrator web interfaces. All the relevant information about the cloud including the amount of available resources and running instances is collected here. Based on this information a Cloud Controller arbitrates available resources, dispatching the load to clusters.

6) Network organization: There are several ways how network infrastructure can be organized for a cloud. The simplest but not the safest option is to put all machines to the public network as shown on Fig. 3. Nevertheless, this approach imposes some inconveniences such as the need for more public IP’s as well as for additional security measures to protect publicly available nodes.

The alternative network configuration assumes that we have a public network with all the internet facing services, and a private network with internal services such as Node Controller and Block Level Storage Controller. Having set up firewalls both on the public and private networks entry points we can organize a so called demilitarized zone (DMZ). [9]

C. Deployment on limited amount of resources

Nevertheless, the actual network scheme significantly depends on the amount of machines available to build a network. In the ideal case we would have hundreds of computers distributed over different network broadcast domains, comprising a huge data centre with a vast cloudy infrastructure. In this case the deployment scheme would be approximately like depicted in Fig. 2.

Unfortunately, small organization and/or research groups with limited budgets can not afford large scale data centres.

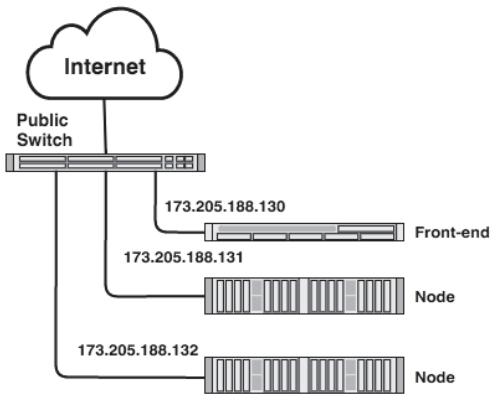

Fig. 3. Nodes connected directly to the public network according to Z. Pantic et al. [8]

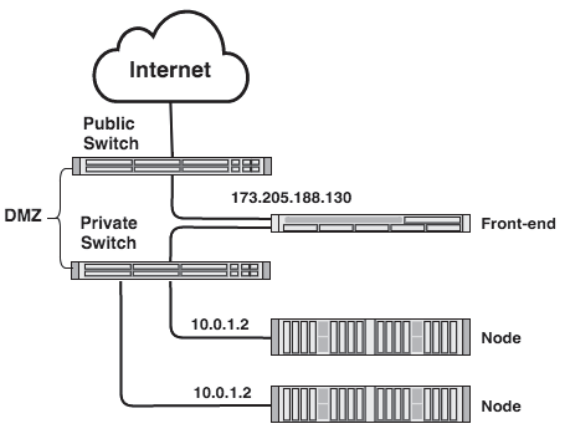

Fig. 4. Nodes on the private subnet, front-end on the public according to Z. Pantic et al. [8]according to Z. Pantic et al. [8]

In this case we should consider deployment of cloud infrastructure on limited amount of resources. Theoretically, all necessary services can be located on a single machine. But it is highly recommended having at least two machines to isolate computationally intensive services such as NCs from those, which are responsible mostly for monitoring and scheduling. There are also security considerations which dictate to split those services (See Fig. 5).

D. Linux Containers

Linux containers is a novel technology based on the Linux kernel addition called **cgroups**. Cgroups kernel module is responsible for managing resources for groups of processes. This allowed to make products which are capable of creating lightweigh Linux virtual machines. Linux Containers (LXC) provide operating-system-level virtualisation which is characterized by having multiple user-spaces when sharing a single kernel-space. By eliminating the necessity of having multiple kernels booted simultaneously, LXC provides better

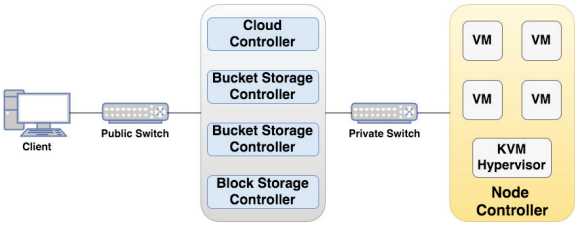

Fig. 5. Minimal set up according to Z. Pantic et al. [8]

productivity, [10] [11] [12] achieves higher density level when multiple Linux containers are deployed.

Linux containers gave birth to many different products such as LXD and Docker. LXD is a Linux containers management tool which is capable of controlling multiple container instances. Docker is also based on cgroups but it doesn't let users to change its configuration. Once created it can not be modified and therefore can not serve as a virtual machine. Instead, Docker serves as an applications wrapper and facilitates rapid deployment of software.

One of the most powerful and promising feature of LXC and other products based on it is the Software Defined Networking (SDN) [13] which makes it possible to connect different containers into a single network regardless of where they are located [14].

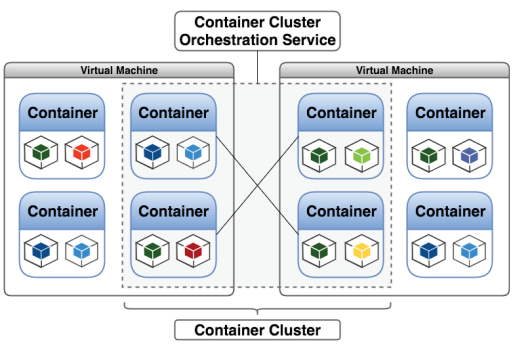

Fig. 6. Container Cluster Organization

This functionality makes it possible to consider a variety of communicational system architectures, characterized by message broker placement.

One of the obvious options is to place a message broker at the highest level on one of the internet facing servers. This broker would be publicly available to all clients and would be able to create a global communicational middleware. There are certain pitfalls related to this approach. High availability level might cause security and reliability issues. Clients which by chance might have gotten authentication credentials which they must not possess could gain access to data they would have not normally been authenticated for. For security sensitive applications such as processing medical data it can be a serious issue. Nevertheless, this approach can be suitable for

less critical applications and therefore shall be considered as a simple way of organizing communicational infrastructure. When used, one should plan a broker scaling strategy either by means of bridging or via third party software such as load-balancers and distributed message brokers like Apache Kafka.

Another option of organizing communicational systems is to place a message broker onto a subnetwork level or inside a demilitarized zone (see Figure 4) protected by an internet facing firewall. This configuration would create a private communicational system for services located inside a corresponding subnetwork.

The third option is to let users place message brokers inside their virtual machines, Linux containers (LXCs) and application wrappers such as Docker. This highly adjustable approach would let developers achieve the highest level of privacy for their communicational flows. Moreover by placing a custom message broker developers would be able to extend brokers default functionality on demand.

There are also examples when mixed approaches may take place. The concept of computational pipelines may need up to three brokers installed on different levels in order to be fully implemented.

V. Pipelines

Most WSN and IoT applications are data centric. It means they make use of existing network- and computing infrastructure to collect, transfer, process and store data about real world. Data processing jobs can be generalized in a form of data transforms. Each transform modifies incoming data according to some business logic. Transforms may be chained together to make a pipeline. A similar concept has been used by Google in their public cloud solution [15].

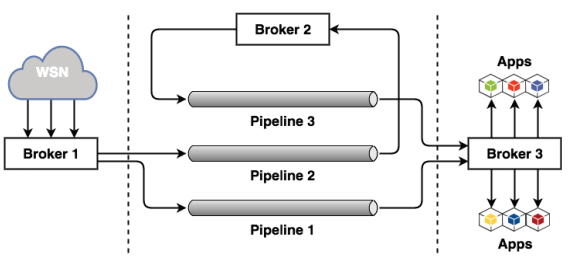

Fig. 7. Pipelines

In computer science a pipeline is a set of data processing elements connected together so that an output of one element can be an input of another. Having implemented a set of standard routines such as getting a mean value, generating distributions, transforming data from one format to another, generating key-pairs for data elements, calculating hash-values, ciphering and/or deciphering incoming data, one can build a sequential chain of transformations or make them work in parallel.

There are multiple ways of organizing such functionality, but there is a need in a simple and yet flexible design for small organizations, universities, research institutions and groups.

On Figure 7 there is a pipelines design pattern depicted. In this case each data processing job is represented as a running instance of a routine and/or executable, which performs a simple transformation.

In this case brokers 1, 2 and 3 act as middleware between the outside world represented by WSNs, data transforms, and user applications. Even though the same functionality could be implemented having only one broker which might be feasible for some very tiny data aggregating centres or for data-centric software prototyping, having three brokers helps organize logical isolation of system components and apply different security rules on every level.

```
public interface DataTransform{
  DataSourceDescriptor ↩
      getOutputDataDescriptor();
  void setInputDataDescriptor(↩
      DataSourceDescriptor);
  void setTParameterList(TParameterList);
  void addParameter(TParameter);
  void cleanTParameterList();
  void run();
  void stop();
}
```

Listing 1. DataTransform Class

A certain SDK is needed in order to build and manage pipelines. A core element of such SDK can be represented as an abstract class or an interface containing descriptions of major methods necessary to operate with data transforms. Listing 1 shows an example of such interface which defines core methods to manage data transforms such as

- ***void*** *setInputDataDescriptor(****DataSourceDescriptor****)* to define a subscription topic and user credentials to get authorized access to input data,
- ***DataSourceDescriptor*** *getOutputDataDescriptor()* to get a corresponding data object for published data,
- ***void*** *setTParameterList(****TParameterList****)* to define a set of parameters for the corresponding data transformation routine be it a software module or an instance of an executable,
- ***void*** *run()* and ***void*** *stop()* to start and terminate data transformation routines.

In the Listing 2 there is an example of a common data transform use case. First, two classes, MeanValue and NormalDistribution, are being declared as an implementation of DataTransform interface. The corresponding objects are being created. Second, a pipeline is built by binding MeanValue output with NormalDistribution input via a corresponding topic name and under common user credentials represented as a DataSourceDescriptor object. When the pipeline is built, both transformations comprising it are started in the order opposite to how they are connected. This is required in order to avoid data losses. I.e. NormalDistribution transform shall start listening for incoming data before MeanValue transform begins publishing them.

```
//create a Mean Value transformation
class MeanValue implements DataTransform{
...
}
//create a Normal Distribution ↩
    transformation
class NormalDistribution implements ↩
    DataTransform{
...
}
//create an input values source descriptor
DataSourceDescriptor mySensorVal = new ↩
    DataSourceDescriptor(username,password,↩
    topicname);
...
//create a Mean Value transform object
DataTransform meanVal = new MeanValue();
meanVal.addParameter("-samplesize 50")
meanVal.setInputDataDescriptor(sensorVal);
...
//create a Normal Distribution transform ↩
    object and link it with the MeanValue ↩
    transform
DataTransform normalDist = new ↩
    NormalDistribution();
normalDist.addParameter("-stdDev 1.5");
normalDist.setInputDataDescriptor(meanVal.↩
    getOutputDataDescriptor());
...
//launch both transforms
normalDist.run();
meanVal.run();
...
//stop both transforms
meanVal.stop();
normalDist.stop();
```

Listing 2. Pipeline Use-Case

Simplicity of the presented architecture favours rapid development and deployment, encourages small teams and individual enthusiasts to contribute to its state, and facilitates data centric IoT projects with useful tools and infrastructure.

VI. Conclusion

In reply to multiple challenges encountered during IoT and WSN related projects an analysis of modern technologies which can be leveraged to boost effectiveness of computational resources' utilization and developers productivity has been made. There has been discovered what modern data-centric communicational protocols, such as publish/subscribe-based MQTT, offer in order to provide higher decoupling level for applications working with sensor data in real-time.

Having multiple projects of different levels of complexity and duration we were also looking toward an optimal solution for rapid software prototyping and deployment. Our aim was to utilize existing limited hardware and network infrastructure in such way, that multiple software developers and researchers could make use of isolated environment to safely manage their data and run their prototypes independently from other participants. It was concluded that a minimal cloud infrastructure can provide this level of comfort when solving a number of other problems such as rapid scaling up and down when demanded, providing appropriate security level, and automation of administrative tasks.

Also, a concept of computational pipelines as a chain of data transform has been presented. It was concluded that a corresponding SDK might be required to manage pipelines and data transforms as well as data flows between them. A simple implementation and usage example have been presented in order to show suggested simplicity of pipelines and to demonstrate how such SDK could be implemented.

These means are supposed to facilitate rapid prototyping, development and deployment of private IoT solutions.

References

[1] X. Li, L. Nie, S. Chen, D. Zhan, and X. Xu, "An iot service framework for smart home: Case study on hem," in *2015 IEEE International Conference on Mobile Services*, June 2015, pp. 438–445.

[2] D. Ajitomi, H. Kawazoe, K. Minami, and N. Esaka, "A cost-effective method to keep availability of many cloud-connected devices," in *2015 IEEE 8th International Conference on Cloud Computing*, June 2015, pp. 1–8.

[3] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, Jun. 2003. [Online]. Available: http://doi.acm.org/10.1145/857076.857078

[4] J. L. Espinosa-Aranda, N. Vallez, C. Sanchez-Bueno, D. Aguado-Araujo, G. Bueno, and O. Deniz, "Pulga, a tiny open-source mqtt broker for flexible and secure iot deployments," in *Communications and Network Security (CNS), 2015 IEEE Conference on*, Sept 2015, pp. 690–694.

[5] A. Antoni, M. Marjanovi, P. Skoir, and I. P. arko, "Comparison of the cupus middleware and mqtt protocol for smart city services," in *Telecommunications (ConTEL), 2015 13th International Conference on*, July 2015, pp. 1–8.

[6] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-s - a publish/subscribe protocol for wireless sensor networks," in *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*, Jan 2008, pp. 791–798.

[7] D. Lee and N. Park, "Security through authentication infrastructure in open maritime cloud," in *2016 International Conference on Platform Technology and Service (PlatCon)*, Feb 2016, pp. 1–2.

[8] Z. Pantic and M. A. Babar, "Guidelines for building a private cloud infrastructure," IT University of Copenhagen, Tech. Rep. TR-2012-153, 2012.

[9] A. Babar and B. Ramsey, "Tutorial: Building secure and scalable private cloud infrastructure with open stack," in *Enterprise Distributed Object Computing Workshop (EDOCW), 2015 IEEE 19th International*, Sept 2015, pp. 166–166.

[10] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, March 2015, pp. 171–172.

[11] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*, March 2015, pp. 342–346.

[12] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb 2013, pp. 233–240.

[13] C. Costache, O. Machidon, A. Mladin, F. Sandu, and R. Bocu, "Software-defined networking of linux containers," in *2014 RoEduNet Conference 13th Edition: Networking in Education and Research Joint Event RENAM 8th Conference*, Sept 2014, pp. 1–4.

[14] M. Hausenblas, *Docker Networking and Service Discovery*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: OReilly Media, Inc., 2016.

[15] (2016) Dataflow programming model. [Online]. Available: https://cloud.google.com/dataflow/model/programming-model