Universität Stuttgart

Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis Nr. 3586

Energy-proportional Machines for Cloud Data Centers

Arturo Francato

Course of Study:	Information Technology/InfoTECH
Examiner:	Prof. Dr. Kurt Rothermel
Supervisor:	Dr. Frank Dürr
Commenced:	June 13th, 2013
Completed:	December 13th, 2013
CR-Classification:	C.2.1, C.2.3, C.4

Abstract

Today's concern is about the energy efficiency of servers and high power machines in a cloud datacenter infrastructure. According to Barroso et al. [1], an ideal machine consumes energy proportional to the work performed. In this case, an idle machine should consume no energy and a machine in operation should only consume energy proportionally to the number of tasks performed. Even though the energy efficiency of machines is constantly improving, they are still not perfectly energy-proportional. Therefore, Dürr proposed the concept of Elastic Tandem Machines Instances (ETMI) in [2] aims to improve the energy efficiency in particular for idle and weakly loaded instances.

In this thesis, we attempt to improve the concept of Elastic Tandem Machines. The original concept only integrated one low-power system on a chip (SoC) machine, which operates during low load on the datacenter, and exactly one high-power virtual machine(VM) instance, powered on when the traffic increases and needs to be redirected. However, if the performance of the SoC and the VM instance differed too much, the efficiency of the approach suffered since at the performance limit of the SoC, when the transferred occurred, the high-power would be almost idle. Therefore, we integrate different performance classes of VMs (e.g., small, medium, and large instances) into Elastic n-Instance Machines to further improve the efficiency and scalability of the system. We then design a predictive algorithm and integrate it with the ETMI to decide, in advance, when the best time is, before overloading any server, to switch among the instances.

The handover algorithm, based on a software-defined networking and the predictive algorithm, based on an Autoregressive Integrated Moving Average (ARIMA) model are presented. The performance of the system with respect to the energy efficiency and machine elasticity is evaluated using experiments and performance benchmarks. The evaluations of the model demonstrate the applicability of low and medium power instances serving low and medium loads efficiently, in addition to the scalability of the solution among n-instances. The predictive method shows satisfactory results when forecasting seasonal data, different models may have to be implemented for non-seasonal series.

Acknowledgements

This thesis would not have been possible without the contribution of many different people, each of whom played a very supportive and distinctive role. I would like to thank the following:

First, Prof. Dr. Kurt Rothermel for the opportunity he provided me to write my thesis in the Institute of Parallel and Distributed Systems.

My supervisor Dr. Frank Dürr for his guidance and patience during these six months of hard work.

My friends Alexandru Costinoaia, Ana Cristina Pintilie, Sukanya Bhowmik, Darshana Das, Sreedhar Mahadevan and Naresh Nayak for helping and motivating me when things got difficult, and also for creating a great atmosphere in the lab.

I would also like to thank Nicholas Mann for proof reading my thesis and giving me advice, and Noel Byrne for understanding and giving me time off work to complete this thesis.

And last but not least, I would like to thank my family for always being there for me in every moment of my life. Without their love, support, understanding and patience this would not have been possible.

I am very grateful.

Contents

Abstract i

Acknowledgements iii

Contents v

List of Figures vii

List of Tables ix

List of Algorithms xi

Acronyms xiii

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Thesis Structure . . . 2

2 Background Fundamentals 3
- 2.1 Cloud Data Centers . . . 3
- 2.2 Cloud Computing . . . 3
- 2.3 Energy . . . 4
  - 2.3.1 Energy Consumption . . . 4
  - 2.3.2 Energy Efficiency . . . 6
- 2.4 Software-Defined Networking and OpenFlow . . . 8
  - 2.4.1 SDN Concept . . . 8
  - 2.4.2 OpenFlow Protocol . . . 10
- 2.5 Time Series Analysis . . . 15
- 2.6 Related Work . . . 19

3 Problem Statement and System Model 21
- 3.1 Problem Statement . . . 21
- 3.2 System Model . . . 25

4 System Design 29
- 4.1 Overview . . . 29
- 4.2 ETMI and Handover Algorithm . . . 30
- 4.3 Predictive Algorithm . . . 35

5 Implementation **39**
5.1 Description of the Hardware 39
5.2 Description of the Software 41
5.3 Measurement Circuit 46

6 Evaluation **49**
6.1 System Performance 49
6.1.1 LPMI Performance 49
6.1.2 MPI Performance 50
6.1.3 ETMI Performance 51
6.2 Energy Efficiency 54
6.2.1 LPMI Power Consumption 55
6.2.2 MPI Power Consumption 56
6.2.3 HPI Power Consumption 57
6.2.4 ETMI Power Consumption 58

7 Summary and Future Work **59**
7.1 Summary 59
7.2 Future Work 60

Bibliography **61**

Author's Statement **63**

List of Figures

2.1 CPU Utilization to Power Consumption[3] 5
2.2 Analysis of a typical 5,000 square foot data center[4] 6
2.3 Server power usage and energy efficiency at varying utilization levels[1] 7
2.4 Power usage and energy efficiency in a more energy-proportional server[1] 7
2.5 Traditional network device 8
2.6 SDN Architecture[5] 9
2.7 SDN architecture with openflow enabled components 12
2.8 Floodlight controller with its interfaces and applications[6] 13

3.1 Server power consumption and energy efficiency at varying utilization levels, from idle to peak performance 23
3.2 System model with interfaces and key components 25
3.3 Raspberry Pi architecture and components[7] 26
3.4 BeagleBone Black board architecture and components[8] 27

4.1 System architecture 31

5.1 Test bed topology. 41
5.2 Sequence flow diagram 46
5.3 Circuits 47
5.4 Final circuit 48
5.5 Complete circuit 48

6.1 LPMI performance 50
6.2 MPI performance 51
6.3 Seasonal time series 52
6.4 Predicted load 52
6.5 ETMI performance 53
6.6 Stationary time series 54
6.7 LPMI power consumption 55
6.8 MPI power consumption 56
6.9 HPI power consumption 57
6.10 ETMI power consumption 58

List of Tables

2.1 Fields from packets used to match against flow entries[9] 11
2.2 Required list of counters for use in statistics messages[9] 11
2.3 Special ARIMA cases[10] 18

6.1 Time to predict 54

List of Algorithms

4.1 Handover Algorithm 33
4.2 Predictive Algorithm 35
5.1 Floodlight Controller Module 42
5.2 StatsThread 43
5.3 AutoArima application 44
5.4 PredictionController 45

Acronyms

AIC Akaike Information Criterion

AICc Corrected Akaike Information Criterion

API Application Programing Interface

AR Auto-Regressive

ARM Advanced RISC Machines

ARMA Autor-Regressive Moving Average

ARIMA Autoregressive Integrated Moving Average

ARP Address Resolution Protocol

CPU Central Processing Unit

DNS Domain Name Server

DDR3L Double Data Rate type three Low Voltages

DHCP Dynamic Host Configuration Protocol

DPID Open vSwitch Data Path Identifier

eMMC embedded Multi-Media Controller

ETMI Elastic Tandem Machines Instances

HPI High-Power Instance

HTTP HyperText Transfer Protocol

IaaS Infrastructure as a Service

IP Internet Protocol

JSON Java Script Object Notation

KPSS Kwiatkowski Phillips Schmidt Shin

LLDP Link Layer Discovery Protocol

LPMI Low-Power Micro Instance

MA Moving Average

MAC Media Access Control

MPI Medium-Power Instance

NIC Network Interface Card

NFS Network File System

OS Operating System

OVSDB Open vSwitch Database

PaaS Platform as a Service

QoS Quality of Service

REST REpresentational State Transfer

RISC Reduced Instruction Set Computer

SaaS Software as a Service

SARIMA Seasonal Auto-Regressive Integrated Moving Average

SDN Software Defined Networking

SDRAM Synchronous Dynamic Random Access Memory

SLA Service Level Agreement

SLO Service Level Objective

SoC System On a Chip

SSL Secure Socket Layer

ToS Type of Service

TCP Transmission Control Protocol

UDP User Datagram Protocol

URL Uniform Resource Locator

VIF Virtual Interface

VLAN Virtual Local Area Network

VM Virtual Machine

VN Virtual Network

XML Extensible Markup Language

Chapter 1

Introduction

A few years ago cloud computing was considered a tendency. As the concept developed and more companies started using and benefitting from it, a major concern was raised regarding energy efficiency. Since cloud computing systems, or cloud data centers, need a great number of servers in order to provide different services, such as infrastructure as a service (IaaS) and platform as a service (PaaS) among others, they have a high energy consumption. Nowadays one of the main concerns is about clean and renewable energy, as the data centers increase and more servers are required to satisfy customers demands, an increasing demand for power also creates growing energy costs, hence improving energy efficiency becomes a critical issue for datacenter management.

Most of the time, high power servers that consume a lot of energy do not even use their full potential and even when in idle mode consume a lot of energy. With a large amount of servers running at the same time, and usually in a limited space data center, a lot of heat is dissipated which means that a good cooling system needs to be implemented to cool down the machines so they can operate properly.

The use of virtualization, where multiple virtual machines (VM) can be created in one physical device, attempts to decrease the number of servers and the unused resources needed to provide specific services. However, to obtain more resources a cloud client needs to transfer its application from one VM to a more powerful one. Sometimes this transfer might not be as smooth as expected, since a new provisioning model from the service provider might only be through static allocation, it would be hard to constantly reallocate resources when the workload keeps fluctuating.

Energy efficient machines are improving, but they are still not entirely energy-proportional, meaning that the machine should consume energy proportional to the number of tasks performed [1]. For example, a typical server used in cloud data centers is energy efficient for medium and high load, but when under low load or even in idle mode it still consumes too much energy. Therefore, the proposed concept of Elastic Tandem Machines Instances (ETMI) in [2] improves the energy efficiency in particular for idle and weakly loaded instances. This concept only integrates one low-power system on a chip (SoC) machine and exactly one high-power virtual machine(VM) instance. The SoC machine serves low load and is always available, when the load increases the VM is powered on and the traffic transferred to it.

However, if the performance of the SoC and the VM instances vary too much, the efficiency of the approach suffers, since at the performance limit of the SoC, when the traffic redirection occurs, the high-power machine would be almost idle.

1.1 Motivation

The goal of this thesis is to improve the energy efficiency and the power consumption of cloud data centers by expanding the concept of Elastic Tandem Machines proposed by Dürr [2]. To overcome the limitations provided by the SoC and avoid the use of high power machines to perform during medium load, we integrate different performance instances to smooth out the scalability and improve the efficiency of the system. By adding more instances, such as Medium-Power Instances (MPI) we prevent underutilized resources and with the use of low power instances we decrease the energy consumption of the datacenter.

The previous system uses a relatively simple concept to decide when to switch between instances; it basically uses a threshold scheme based on the current load of the system. Therefore we integrate a more complex load and performance model. A predictive model is designed and integrated into the system; it forecasts the future load on the datacenter to decide in advance where to redirect the traffic.

1.2 Thesis Structure

Chapter 2 discusses the background concepts applied in the research and development of this thesis, such as cloud computing, software-defined networking (SDN) along with its elements and protocols, energy consumption in cloud data centers and the ARIMA model which was used to predict and decide in advance when to switch the traffic to the appropriate server. It also presents some related literature about this work. Chapter 3 gives a clear description of the problem statement and introduces the model of our system. Chapter 4 presents the system design and explains in detail the handover algorithm, based on Software-Defined Networking (SDN) technologies and the predictive algorithm used to forecast the load in the datacenter. The implementation of our system, together with the description of the hardware and software used in our solution, is described in chapter 5. Chapter 6 evaluates the performance benchmarks for the LPMIs and HPI in addition to the energy consumption measurements of the tested devices, as well as the performance analysis of the ETMI concept. Chapter 7 concludes the thesis and gives some suggestions for future work.

Chapter 2

Background Fundamentals

In this chapter, concepts such as cloud data centers and cloud computing, energy consumption and energy-proportional machines, along with Software-Defined Networking and OpenFlow will be introduced. The prediction model used to forecast the traffic load on the web server will also be presented.

2.1 Cloud Data Centers

A data center, also known as a server farm, is where the majority of an enterprise's servers, storage and network equipment are located, controlled and managed.

These data centers consist mainly of support infrastructure, IT equipment and maintenance operations. The support infrastructure refers to the equipment required to support data center operations, containing power distribution units (PDU), uninterruptible power supply (UPS), generators, computer room air conditioners (CRACs), remote transmission units (RTUs), chillers and air distribution systems among others. IT equipment includes the racks, cabling, servers, storage, network gear and management systems required to provide computing services. Maintenance operations ensure that both IT and infrastructure systems are properly operated, maintained, upgraded and repaired when necessary.

Typically a data center is on-premise hardware that stores data within an organization's local network, whereas a cloud data center is an off-premise form of computing that keeps the hardware infrastructure at a different location and the data can be accessed via the internet. Companies that provide cloud computing services outsource their infrastructure, platform or even software to third-party companies in the form of services.

2.2 Cloud Computing

Cloud computing is a technology in which the computing resources, such as servers, databases and applications, are distributed over a network and shared throughout the internet. The ability to segment this infrastructure and run different applications on many connected servers at the same time is ideal for organizations

to offer their resources, as a service, to other companies. Cloud providers offer their computing resources in various models, like infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS).

Since the users' requirements for cloud services are vast and diverse, the providers must be able to quickly adapt and reallocate their resources for a better utilization and efficiency of their equipment. Currently, the companies outsource their infrastructure in the form of virtual machines (VMs), where in a single physical machine, multiple virtual computing environments can coexist.

As cloud computing becomes more popular, reliable and efficient, the cloud providers need to update and improve their data centers and a major concern in doing so is the energy consumption and efficiency of their infrastructure.

2.3 Energy

Energy is the capacity of a physical system or a body to perform work. There are different forms of energy, such as electrical, mechanical, chemical, thermal, or nuclear. Energy can also be transformed from one form to another. It can be measured by the amount of work done in joules (J), according to the international System of Units (SI).

Power is described as the work done per unit of time, it is measured in watts (W) according to the SI and it is denoted by the letter P. Equation 2.1 shows the electrical power produced by an electric current I going through a voltage difference of V.

$$P = V \times I \tag{2.1}$$

Most of the electrical power that passes through a computer or server is converted into heat. Depending on the amount of energy the server consumes, the greater the heat dissipation. To avoid overheating these devices, heat sinks and fans are often used to increase their capacity to dissipate heat. The goal to reduce the amount of energy consumed by these servers, in other words, the efficient use of the energy, is the main cost-effective strategy today.

In this thesis, we are concerned about the amount of electrical energy that is being transferred by the servers, therefore the electric power produced when the servers are in use and in idle mode.

2.3.1 Energy Consumption

Energy consumption of electric energy is measured by $W \times h$ (Watt x hour). Electric devices use electric energy to produce a desired output, but not all of the energy is converted as desired. Therefore, the energy efficiency of a device is measured by the amount of work accomplished compared to the amount of energy it consumed to perform the operation.

The main source of power consumption in a server is the CPU [11]. The power consumption of the processor can vary depending on many aspects such as number of cores, technology used and also by its work load. Nowadays, multi-core processors are much more power-efficient than previous models, but they are still not 100 percent power-efficient. As shown in Figure 2.1, even when the server is idle the power consumption is still very high.

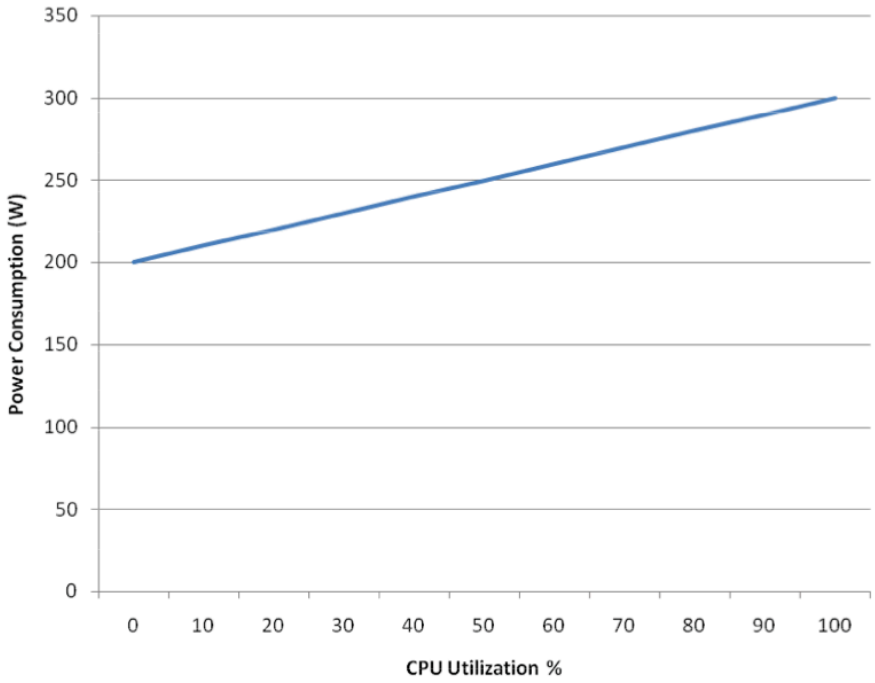
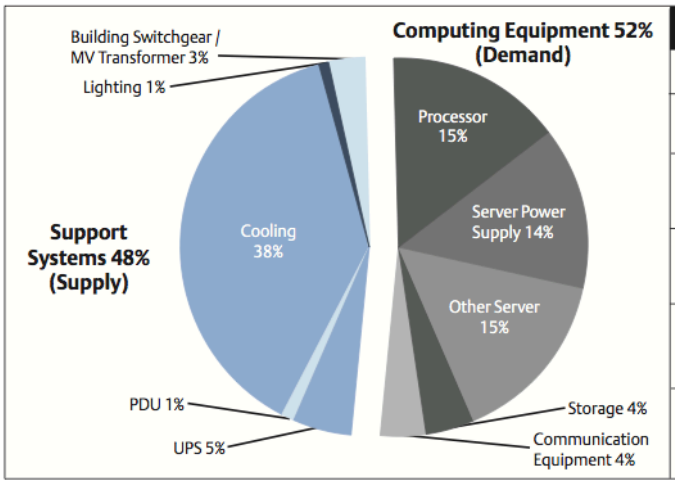


Figure 2.1: CPU Utilization to Power Consumption[3]

When evaluating the power consumption of a data center, not only the servers but many other elements should be taken into consideration. In an analysis made by [4] in Figure 2.2, where they modeled and examined the energy consumption of a 5,000 square foot data center, they categorized the energy use in either supply or demand, where demand systems are the servers, storage, network and other IT systems and the supply systems support the demand side, in this case UPS, cooling, PDUs etc.

Since much of the electrical energy that goes into the server is converted into heat, a lot of energy is also required to remove the generated heat from the server or a data center filled with servers. To cool down the servers and a data center a cooling system is required, which accounts for almost 40% of the power consumption of the datacenter.



Category	Power Draw*
Computing	588 kW
Lighting	10 kW
UPS and distribution losses	72 kW
Cooling power draw for computing and UPS losses	429 kW
Building switchgear/MV transformer/other losses	28 kW
TOTAL	1,127 kW

Figure 2.2: Analysis of a typical 5,000 square foot data center[4]

2.3.2 Energy Efficiency

Energy efficiency is the amount of work done compared to the amount of energy used. The exact way to measure and compare energy efficiency varies depending on the particular application. When considering the energy efficiency for services provided by a cloud or a server, it could be for example how many Joules are needed for completing a transaction or retrieving a file from a web server. Depending on the application, the metrics can greatly vary.

As stated by Barroso and Hölzle [1], servers usually operate most of the time between 10 and 50 percent of their maximum utilization levels. When measuring the energy efficiency of a typical server, the efficiency between those levels can be half of the efficiency at peak performance. To obtain the energy efficiency, a simple division between utilization and power value is performed, where utilization is a measure of an application's performance, for example the number of requests on a web server, normalized to the performance at maximum load levels.

Figure 2.3 shows the comparison of the power usage and the energy efficiency of a server varying the utilization levels from idle to maximum performance. When analyzing the efficiency where the servers spend most of their time (10 to 50 percent), it is evident that the power usage is too high compared to the utilization level.

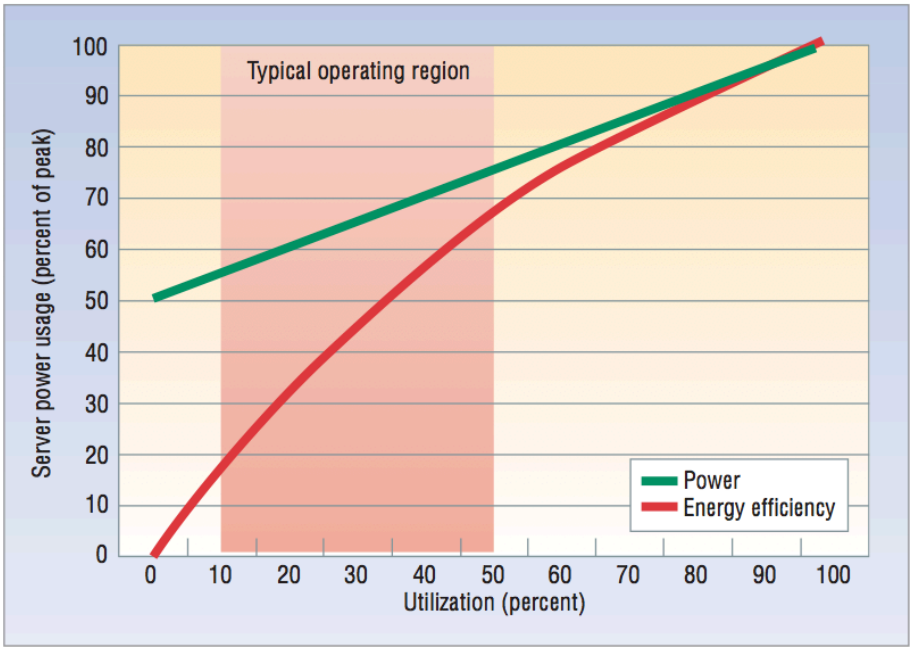


Figure 2.3: Server power usage and energy efficiency at varying utilization levels[1]

For a better power usage of the servers, Barroso and Hölzle [1] proposed the idea of energy-proportional machines, where energy-efficient servers consume energy proportionally to their utilization or amount of work done. With energy-proportional servers, the energy consumed in a datacenter could decrease considerably. As seen in Figure 2.4, the energy efficiency of the server is above 50 percent in the typical operating region.

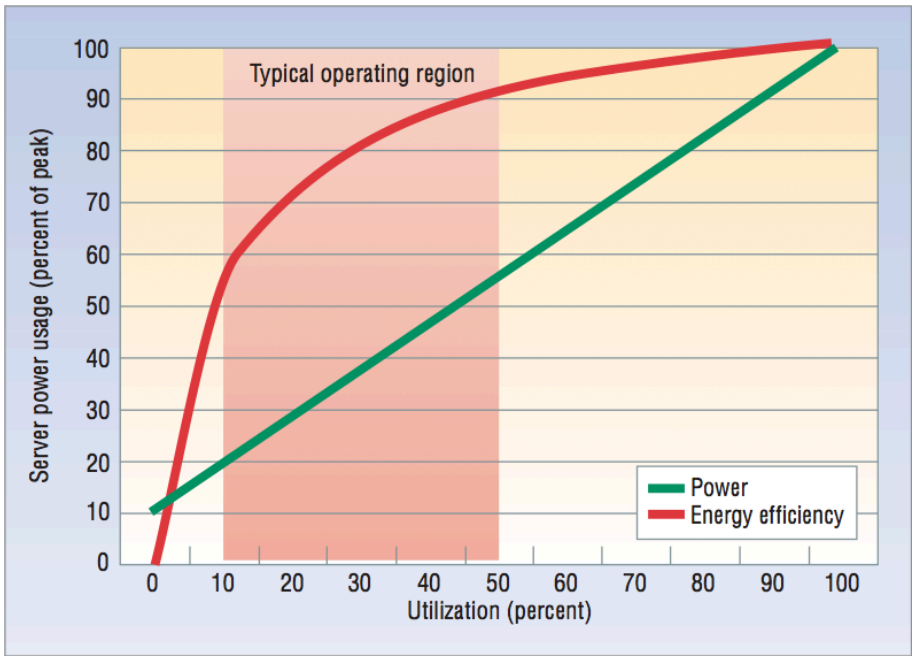


Figure 2.4: Power usage and energy efficiency in a more energy-proportional server[1]

2.4 Software-Defined Networking and OpenFlow

2.4.1 SDN Concept

Software Defined Networking (SDN) is a developing network architecture designed to innovate and address some issues in today's traditional network. The traditional network architecture is formed by three planes of operation, which are the management, the control and the data planes, all combined in a single network device.

The management plane is responsible for maintaining, configuring and monitoring the devices, through protocols such as Secure Shell (SSH), Simple Network Management Protocol (SNMP) and Telnet among others.

On the control plane the network devices run routing protocols, like Open Shortest Path First (OSPF), Border Gateway Protocol (BGP), etc, to learn about the neighbor devices and the complete network topology or reachability information. The routing protocols then load the data into the Routing information Base (RIB), which will populate the Forwarding Information Base (FIB).

The data plane contains the FIBs and once a packet/frame is received through an input port, it will be then checked on the the forwarding table and will be dispatched to an output port. Figure 2.5 shows the design of a traditional network device.

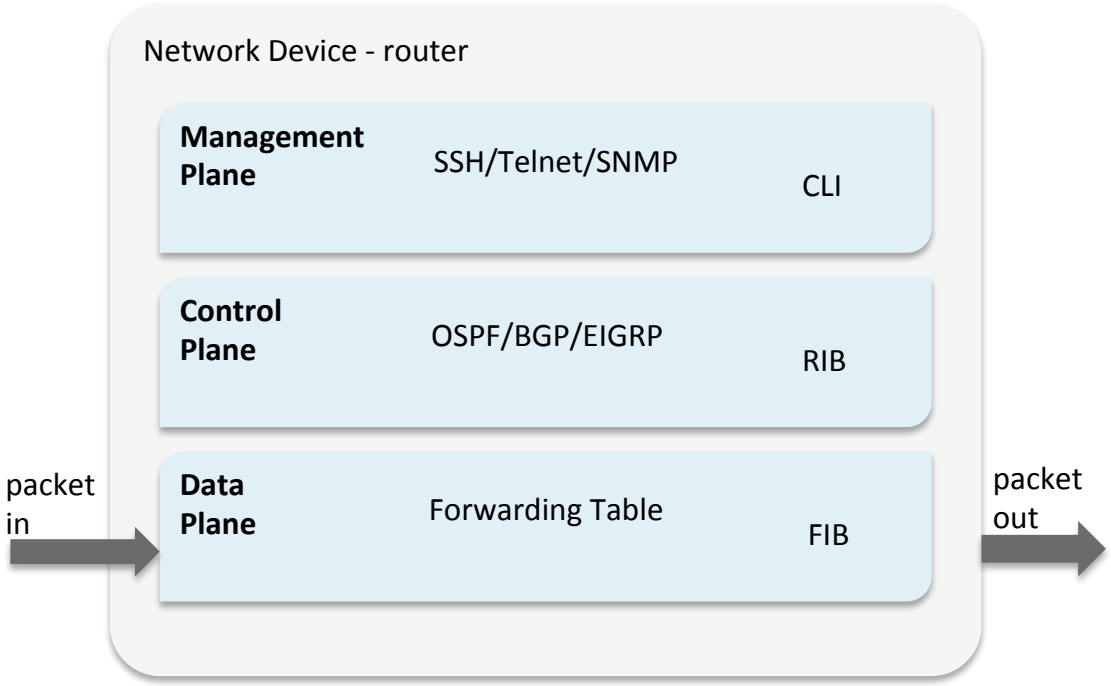


Figure 2.5: Traditional network device

The rapid increase in cloud data centers and services, server virtualization, mobile devices and content are some trends motivating the networking industry to reevaluate its traditional architectures. Many conventional networks are built with

layers of Ethernet switches arranged in a tree structure. This architecture was suitable when client-server computing was dominant, but today's needs in the enterprise data centers, universities campuses and carrier's environments for a new dynamic computing make the traditional static architecture unsuitable [5].

According to the OpenFlow Networking Foundation (ONF)[5], SDN is the new emerging design that is dynamic, convenient, cost-effective and adaptable, which makes it ideal for today's networking needs. This new architecture separates the control and the data planes, giving the control plane programmable features and the data plane more focus in the forwarding of packets.

As illustrated in Figure 2.6, the SDN architecture shows the division of the control and data planes. Now all of the intelligence of the network is centralized in the control layer, where software-based SDN controllers keep the general view of the whole network. The infrastructure or data layer receives and accepts instructions from the controllers and acts accordingly.

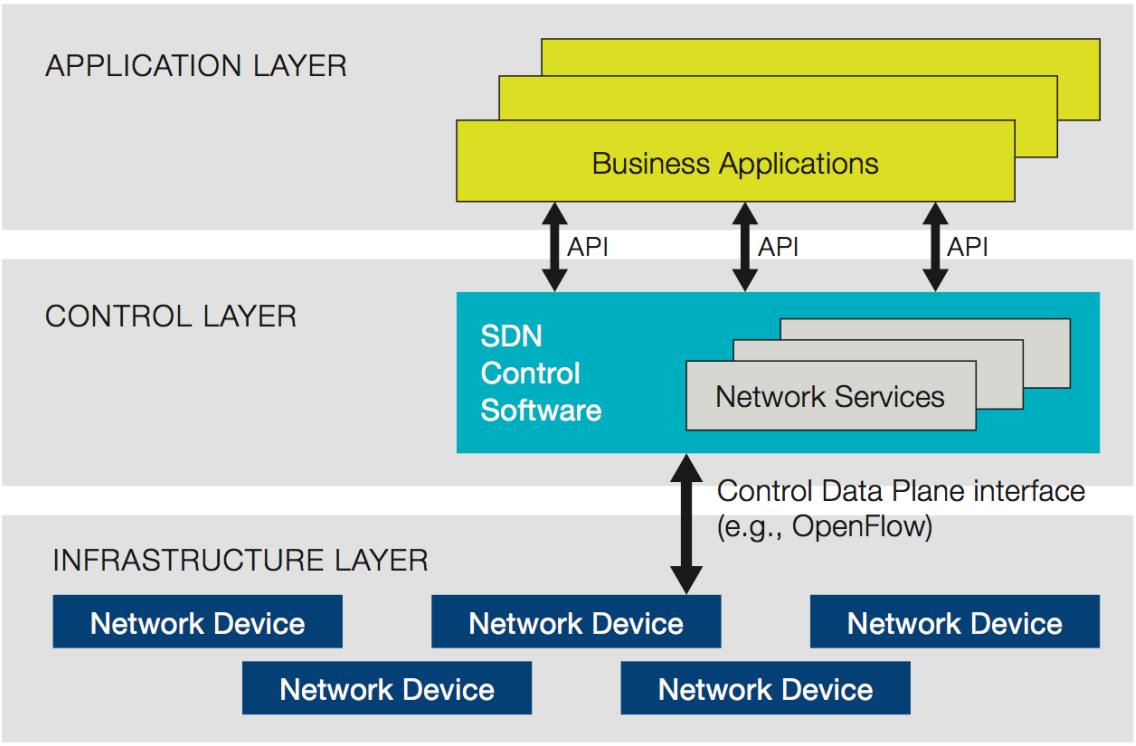


Figure 2.6: SDN Architecture[5]

This design simplifies the expansion and configuration of the network, when now network managers just programmatically configure the controllers that will then update the network, instead of having to configure each network device by hand.

SDN also has an application layer and with a set of supported APIs it is possible to develop applications according to each business and network strategy. Routing, traffic engineering, QoS, security and management are examples of common network services that can be implemented and customized to meet customer requirements.

Communication between the control layer and the data layer takes place through the standard OpenFlow protocol, as shown in Figure 2.6.

2.4.2 OpenFlow Protocol

OpenFlow is the standard protocol responsible for the interface between the control and the forwarding layers deployed in an SDN architecture. It provides access to the forwarding layer of the network devices, such as switches and routers, enabling traffic management and control.

The OpenFlow protocol is present at both ends of the interface. The control plane operation, which previously resided in the network device, is now shifted to run on external servers, so-called controllers. The network devices can be designed to support not only the traditional forwarding but also OpenFlow-based forwarding.

OpenFlow Controller

The controller is an application that should be connected to all network devices in order to send updates and receive data from these devices. The controller is responsible for managing the data flow between servers and network devices in an SDN environment. It creates a topology of the network and runs an algorithm to decide how the forwarding table will appear; as soon as the algorithm is performed across the network topology, the forwarding tables are dispatched to the switches or routers through the OpenFlow protocol.

The OpenFlow controller maintains an encrypted Secure Socket Layer (SSL) connection with the OpenFlow switches in order to communicate through the OpenFlow protocol. There is also a link discovery protocol so the Controller can discover what the network devices look like and the switches can signal back link/port state to the controller.

OpenFlow Switch

The OpenFlow switch or a traditional switch that is OpenFlow-based contains a secure channel, connected to an external controller, and a flow table, responsible for packet lookup and forwarding. Compared to the traditional switch, the OpenFlow switch's functionalities have been reduced as it only has to deal with packet handling, which it does using flow tables.

The flow table consists of a set of rules or flow entries; each flow entry contains a matching field or header field, activity counters or statistics, and a set of actions. When a packet ingresses the switch, it is compared against the switch's flow table based on prioritization. An entry that specifies an exact match (e.g. it has no wildcards) is always the highest priority. Higher priority entries must match before lower priority ones. If multiple entries have the same priority, the switch is free to choose any ordering. If no match is found for that particular packet, the switch sends the packet to the controller, which will handle the packet. When a match

is found, the action determined for that specific match will be performed and the statistics counters will be updated [9].

As defined in the Openflow Switch Specification version 1.0.0 [9], the header fields that the packets are matched against can be seen in Table 2.1.

Ingress Port	MAC src	MAC dst	Ether Type	VLAN ID	VLAN Prior	IP src	IP dst	IP Port	IP Tos	TCP UDP src port	TCP UDP dst port

Table 2.1: Fields from packets used to match against flow entries[9]

The action field consists of the instruction that the switch will apply in the packet; some of these actions are "forward", which sends the packet to a specific output port; it can also send the packet to multiple ports or even to the controller and "drop", where the switch drops the packet. Other optional actions may also be performed, such as forward the packet to a queue attached to a port, flood the packet to all ports and other switch's ports, and also modify a field, including setting a VLAN ID, changing the MAC address, modifying the IPv4 destination address and so on.

Statistics are also provided by the switch; whenever a rule is reached the switch will increment a counter. These counters can be per-table, per-flow, per-port and per-queue [9]. Table 2.2 shows the counters that can be retrieved by statistics messages.

Counter	Bits	Counter	Bits
Per Table		Per Port	
Active Entries	32	Received Packets	64
Packet Lookups	64	Transmitted Packets	64
Packet Matches	64	Received Bytes	64
Per Flow		Transmitted Bytes	64
Received Packets	64	Receive Drops	64
Received Bytes	64	Transmit Drops	64
Duration (seconds)	32	Receive Errors	64
Duration (nanoseconds)	32	Transmit Errors	64
Per Queue		Receive Frame Alignment Errors	64
Transmit Packets	64	Receive Overrun Errors	64
Transmit Bytes	64	Receive CRC Errors	64
Transmit Overrun Errors	64	Collisions	64

Table 2.2: Required list of counters for use in statistics messages[9]

Figure 2.7 shows the SDN architecture with openflow enabled components.

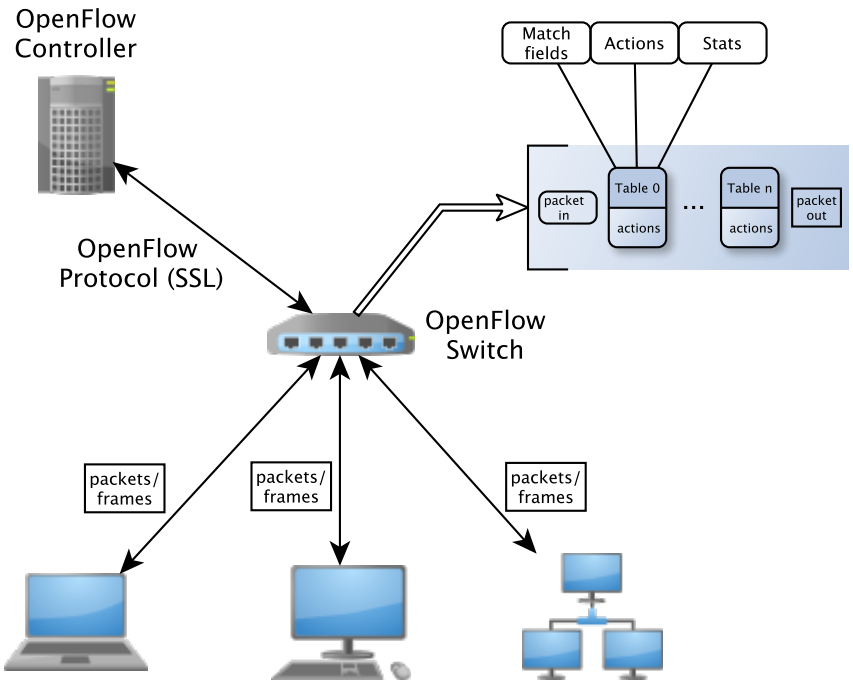


Figure 2.7: SDN architecture with openflow enabled components

Reactive and Proactive Approaches

There are two methods which the OpenFlow controller follows to set up the flow tables in the OpenFlow switches: One is the reactive flow instantiation and the other is the proactive flow instantiation.

The reactive flow happens when a packet received by a switch is not matched to any flow table and the switch reacts to it by encapsulating it in a so-called PACKET-IN and sending it to the controller in order to receive instructions on what to do with the packet. In other words, upon receipt of a packet from the switch the controller will determine the correct traffic for it and it will update the flow entries in that switch.

Even though this approach saves memory by not storing several flow tables in the switch and efficiently using the flow tables, it has its drawbacks. The controller is essential in this approach: if the connection is lost between the switch and the controller, the switch will have limited utility. It also uses a great amount of the controller's CPU, for example in a large network where several switches are connected and requesting instructions at the same time from the controller.

With proactive flow instantiation rather then reacting to the packet and sending it to the controller, the controller populates the flow tables of the switches ahead of time. In this case the controller must know the topology of the network and all the addresses of the end devices in order to download the flow entries into the switches.

In this approach, if the connection to the controller is lost, the traffic will not be interrupted and it also removes any latency produced by requesting the controller for flow entries.

Floodlight Open SDN Controller

Floodlight is a software implemented in Java which is not only an OpenFlow controller but also a set of applications built on top of the Floodlight controller [6].

Through a collection of functionalities, the Floodlight controller can analyze and control an OpenFlow network and, with applications built on top of it, users can develop different features to fulfill their needs.

By adopting a modular architecture, besides implementing its core network features through its controller modules, Floodlight also has application modules that implement other applications for various purposes. These applications are developed based on Java and REpresentational State Transfer (REST) application programming interfaces (API), which serves as the interface between the applications and the features supported by Floodlight.

Figure 2.8 shows in detail the Floodlight controller architecture, together with the controller built-in modules and the Java and REST API interfaces connected to the application modules.

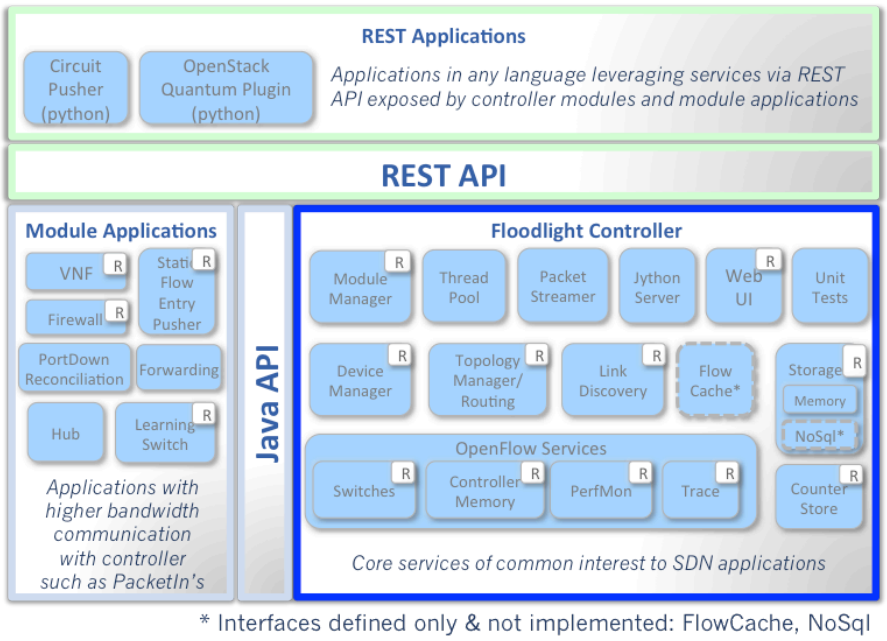


Figure 2.8: Floodlight controller with its interfaces and applications[6]

Controller Modules

The controller core modules are a set of common functions already implemented and that are used by a great number of applications, such as discovering and monitoring that state of the network, capturing events and building the network's topol-

ogy and flows, allowing the communication between the controller and the switches connected to the network through the OpenFlow protocol, managing the application modules and the resources shared between the modules, besides providing a graphical web user interface.

The following paragraphs describe some of these implemented controller modules.

FloodlightProvider

The "FloodlightProvider" module monitors the connections between the Floodlight controller and the OpenFlow switches, collects the OpenFlow messages transferred, such as PACKET-IN, PortStatus notification, FlowRemoved, etc., and adapts them into events that can be used by other modules. It is also responsible for determining the order in which these messages are transmitted to the modules.

DeviceManagerImpl

The "DeviceManagerImpl" module monitors the trajectory of end-host devices in the network by collecting information from packet-in events and matching their attachment points to switches, and it defines the destination device for a new flow.

LinkDiscoveryManager

The "LinkDiscoveryManager" module uses Link Layer Discovery Protocol (LLDP) and broadcast packets to detect and manage the status of links in the OpenFlow network.

TopologyService

Besides keeping track of the network topology for the controller, the "TopologyService" module can also discover routes in the network. The module depends on the results of the LinkDiscoveryManager and the FloodlightProvider modules to operate.

RestApiServer

The "RestApiServer" module permits that other modules present their services through the REST API. It uses the HTTP protocol and its methods such as GET, POST, PUT and DELETE to perform different operations. It can be invoked by the four methods using the custom Uniform Resource Locator (URL) and by modeling the data structures in the form of Extensible Markup Language (XML) or Java Script Object Notation (JSON) objects.

Application Modules

Floodlight already implements a few application modules that allow the controller to perform the reactive and proactive approaches to set up the flows in the OpenFlow switches.

Forwarding

The "Forwarding" module is the default reactive approach application of the Floodlight controller. Upon receipt of a packet-in, it will calculate the shortest path and forward the packets to the designated devices; if the controller does not know the destination of a particular packet, it will flood the packet onto the network.

Static Flow Pusher

The "Static Flow Pusher" module is the proactive approach that allows the manual insertion and extraction of static flows into the switches' flow tables. These flows will neither age out nor be removed automatically from the switches.

Open vSwitch

Open vSwitch is an open source project licensed by Apache 2.0 and mostly developed and deployed in Linux. It is a production quality, multilayer virtual switch that also supports the OpenFlow Protocol. It allows substantial network automation using programmatic extension and supports standard protocols and interfaces such as 802.1ag link monitoring, Spanning Tree Protocol (STP) and IPv6 among others [12].

Open vSwitch acts as an Ethernet physical switch but deployed in software. Logical ports can be created and configured, attaching either virtual network interfaces (VIFs) to perform virtual machine connectivity or network interface controllers (NICs) to emulate a physical interface. To distinctively identify an Open vSwitch instance bridge, a Data Path Identifier (DPID) is given and each logical port is assigned a port number.

The Open vSwitch can perform traffic forwarding between hosts and VMs connected to a physical network or forward traffic among VMs inside the same physical host. It supports standard management interfaces, for example sFlow, NetFlow, CLI and the Open vSwitch Database (OVSDB) management protocol to administer and query port status and configurations.

2.5 Time Series Analysis

Time series is the collection of data observed sequentially over a period of time. In order to obtain significant characteristics and statistics of the data, methods of time series analysis are used for analyzing time series data.

There are two main purposes for analyzing a time series: One is to comprehend and model a sequence of random variables that gives rise to an observed series and the other one is to predict or forecast the future values of a series based on previously observed values and other factors [13].

Time series data models may contain different forms and reproduce distinctive stochastic processes. When modeling variations in the level of a process, three main

classes, which depend linearly on previous data points, are presented: the Auto-Regressive (AR) models, the Integrated (I) models, and the Moving Average (MA) models. New models can be generated by combining these classes, such as Auto-Regressive Moving Average (ARMA) and Autoregressive Integrated Moving Average (ARIMA) models.

Time Series Concepts

- Stationarity - a time series is called stationary if its properties do not depend on the time at which the series is observed. For example, if y_t is a stationary time series, then for all s, the distribution of $(y_t, \ldots, y_{t+s})$ does not depend on t. Time series containing trends or seasonality are considered non-stationary, while a white noise is stationary [10].

- Differencing - is the calculation of the difference between consecutive observations. This method is used in order to make a time series stationary and remove its seasonality and trend features.

- Akaike Information Criterion (AIC) - for a given set of data AIC measures the appropriate quality of a statistical model. It helps in the selection of a model from a set of models. When selecting a forecasting model, the model with the lowest AIC is considered the most suitable. A second-order variant of AIC, also called AICc, was derived by Sugiura 1978, Sakamoto et al. (1986), as when the number of parameters in relation to the size of the sample is too high the AIC could perform weakly [14].

- Backshift operator - also known as the lag operator, is a convenient notation in time series analysis when an element of a series is operated to produce the previous element. The backshift operator can be denoted as B or L(for lag), as seen in equation 2.2 , B operating on y_t moves the data back k period(s).

$$B^k y_t = y_{t-k} \tag{2.2}$$

  The use of the backshift operator is also convenient when describing the method of differencing; in a first order difference the backshift notation can be written as

$$y'_t = y_t - y_{t-1} = y_t - By_t = (1 - B)\, y_t \tag{2.3}$$

  in a more general view, a difference of order k can be written as

$$(1 - B)^k\, y_t \tag{2.4}$$

- White Noise - In statistical forecasting, white noise refers to a time series that has no correlated variables, with zero mean and finite variance.

ARIMA Model

An ARIMA model is the combination of differencing with autoregression (AR) and moving average (MA). The differencing of the time series, causing the removal of its trend and seasonality, is integrated into an Auto-Regressive Moving Average (ARMA) model, thereby creating the ARIMA model. The same conditions, such as stationarity and invertibility, used for AR and MA are applied to the ARIMA model.

In the autoregressive (AR) model the prediction of the output variable is generated using a linear combination of the previous values of the variable, as indicated by the name autoregression or the regression of the variable against itself [10]. An autoregression model of order p is denoted by $AR\,(p)$and it is defined as

$$y_t = c + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \cdots + \phi_p y_{t-p} + e_t \tag{2.5}$$

where $\phi_1, \ldots, \phi_p$ are the parameters of the model, c is a constant and e_t is white noise. Applying the backshift notation the equation can be abbreviated to

$$y_t = c + \sum_{i=1}^{p} \phi_i B^i y_t + e_t \tag{2.6}$$

The moving average (MA) model, instead of depending on the past values of the forecast variable in a regression, uses a linear regression of the current and previous white noise errors. As seen in equation 2.7, where e_t is white noise and $\phi_1, \ldots, \phi_q$ are the parameters of the model

$$y_t = c + e_t + \theta_1 e_{t-1} + \theta_2 e_{t-2} + \cdots + \theta_q e_{t-q} \tag{2.7}$$

The equation can be abbreviated the equation with the backshift operator B

$$y_t = c + (1 + \theta_1 B + \cdots + \theta_q B^q)\, e_t \tag{2.8}$$

The ARMA model, combining AR and MA, assumes that the process y_t is stationary, but to deal with non-stationary series the differencing can help to make the process stationary, thus the ARIMA model extends ARMA. The series y'_t, as seen in equation 2.9, is the differenced series and the other parameters are the lagged values of y_t and lagged errors

$$y'_t = c + \phi_1 y'_{t-1} + \cdots + \phi_p y'_{t-p} + \theta_1 e_{t-1} + \cdots + \theta_q e_{t-q} + e_t \tag{2.9}$$

A non-seasonal ARIMA model is defined as an $ARIMA\,(p, d, q)$ model, where

$$\underset{\substack{\uparrow \\ AR(\mathrm{p})}}{(1 - \phi_1 B - \cdots - \phi_p B^p)} \quad \underset{\substack{\uparrow \\ I(\mathrm{d}\ differences)}}{(1 - B)^d y_t} \quad = \quad \underset{\substack{\uparrow \\ MA(\mathrm{q})}}{c + (1 + \theta_1 B + \cdots + \theta_q B^q) e_t}$$

The ARIMA model can also represent other models according to the selected values of p, d and q. Table 2.3 shows some examples of special ARIMA cases.

White Noise	$ARIMA\ (0,0,0)$
AutoRegression	$ARIMA\ (p,0,0)$
Moving Average	$ARIMA\ (0,0,q)$

Table 2.3: Special ARIMA cases[10]

The ARIMA model may experience some variations and extensions to forecast different types of series. When a seasonal effect is suspected in the model, an extension of the ARIMA model can include seasonal components to capture the periodic variations in the series. This extension of the ARIMA model is also called Seasonal ARIMA (SARIMA) and the only difference from the non-seasonal model is that it adds backshift components of the seasonal period. The seasonal ARIMA is defined as

$$ARIMA\ (p,d,q)\ (P,D,Q)_m$$

where (p,d,q) is the non-seasonal part of the model and (P,D,Q) is the seasonal part with m being the number of periods per season.

The equation involving the seasonal periods, without differencing, can be written as

$$\Phi\left(B^m\right)\phi\left(B\right)\left(y_t - c\right) = \theta\left(B\right)\Theta\left(B^m\right)e_t \tag{2.10}$$

The non-seasonal part is:

AR: $\phi\left(B\right) = 1 - \phi_1 B - \cdots - \phi_p B^p$
MA: $\theta\left(B\right) = 1 + \theta_1 B + \cdots + \theta_q B^q$

The seasonal part is:

SAR: $\Phi\left(B^m\right) = 1 - \Phi_1 B^m - \cdots - \Phi_p B^{pm}$
SMA: $\Theta\left(B^m\right) = 1 + \Theta_1 B^m + \cdots + \Theta_q B^{qm}$

Statistical software and programming languages are widely used to automate the forecasting of time series. R programming languages is an example of software and programming language used by statisticians to develop statistical software and data analysis.

R provides a variety of packages, containing functions and extensions, to facilitate the implementation of statistical and graphical techniques, such as the forecast, which provides forecasting functions, tseries, for time series analysis and so on.

2.6 Related Work

In this section, we review some literature related to our approach to improving the energy efficiency and resource utilization of cloud infrastructures.

In [15], the authors present a model of elastic resources framework for IaaS to address the need for a more flexible resource allocation mechanism to benefit the variable workload on servers and VM. They developed a forecasting engine that predicts the expected demand which a resource manager uses to allocate the resources and minimize the service level agreement (SLA) violations.

In their approach, they predict the VM CPU requirement based on the requested rates and they use the response time to measure SLA violations. After verifying their prediction with the pricing policy and SLA penalties, they choose the appropriate resources.

Another approach similar to [15] is [16], where an original scheme named PRedictive Elastic reSource Scaling (PRESS), to automatically adjust the resource allocation to applications' demands, is presented. They developed a signal processing technique that identifies patterns in the system. Then a discrete-time Markov chain is applied to predict the demand for the near future. According to the prediction, PRESS tries to avoid under-estimated values to prevent service level objective (SLO) violations.

Even though their approach is similar to ours in the sense of predicting the load and reallocating resources, they are concerned specially with preserving performance SLA and SLO, and not with energy efficiency. Since they use a homogeneous system with VM allocation, meaning the resource even if idle will still be powered on and consume energy. This is different from our approach, which utilizes heterogeneous hardware to efficiently support idle and weakly loaded VMs without sacrificing availability.

Based on energy-proportional systems, [17] also attempts to reduce energy consumption in data centers. The approach of using a hybrid datacenter design, combining low and high power platforms to balance the workload, is proposed with an Intel Atom SoC and a Xeon platform. The traffic is transferred from one platform to the other as soon as the load increases or decreases. If there is no traffic on either device, that device will be in idle mode and will be awakened by the other device when the traffic needs to migrate.

This approach uses a complete and discrete design to integrate their system, whereas we propose a network-based mechanism to integrate our system.

With the emergent OpenFlow standard, alternative network solutions are enabled. For example, in cases where balancing the traffic load is necessary, the use of OpenFlow-enabled switches and OpenFlow controllers could avoid dedicated load balancers, which can become a bottleneck on the network. An approach which takes advantage of the SDN architecture is presented in [18], where the openflow network

switches together with the controller are responsible for dividing the traffic among the servers by creating wildcard rules to handle the packets and automatically balance the load without disrupting current connections.

Even though this approach relates to ours, as it uses an SDN solution to balance the load across the datacenter, it has some weaknesses when redirecting the traffic. The system uses the TCP SYN flag to distinguish between new and existing connections, which can only be matched by the openflow controller. In the case of a centralized controller it might lead to a bottleneck in the system. Another problem is when the system tries to identify the end of a connection, which it does by assuming 60 second of inactivity. This might lead to broken connections if packets are transferred after the 60 second timeout. We prevent this scenario by keeping a connection available between the servers and the controller that maintains the open connections while still transferring the new ones to another instance.

This thesis attempts to improve a previous approach proposed by [2], where the concept of Elastic Tandem Machines Instances (ETMI) was introduced. It integrates two servers to simulate an energy-proportional machine and it aims to improve the energy consumption of cloud data centers. One server would be a SoC machine, referred to as a Low-Power Micro Instance (LPMI), and the other a commodity PC host, called High-Power Instance (HPI).

The basic idea of the approach is to keep the LPMI constantly running, since it only consumes a few watts, and leave the HPI in a sleeping mode. During low load conditions only the LPMI will be running to improve the energy efficiency; as soon as the load rises beyond a certain predefined threshold, where the performance of the LPMI drops, the HPI is turned on and the traffic is transferred to the HPI. The same process happens when the load drops, when the traffic is transferred back to the LPMI and the HPI returns to a dormant to consume less energy.

SDN technology is used to seamlessly perform the handover from one instance to the other. This ensures that currently established connections are not broken and new connections can be forwarded to the new instance in a way that is transparent for the clients.

To achieve a more energy efficient system, we try to fill the gap between an LPMI and an HPI by adding more instances to bring the energy efficiency curve as close as possible to an ideal energy-proportional machine, as described in [1].

Besides increasing the number of instances, a predictive model is used to improve the efficiency and scalability of the system, determining in advance when to transfer the load among the machines.

Chapter 3

Problem Statement and System Model

In this section, the intended system model and the assumptions for our approach are presented. Based on the current data centers' architecture and infrastructure we show their disadvantages and we attempt to efficiently improve them with our model.

Our system design is focused on specific data centers where IaaS providers offer their infrastructure resources to the customers. In order to improve efficiency and resource utilization, these providers use methods such as virtualization combined with server consolidation and overbooking of resources. These methods can be very beneficial in some sense, but as we show in the next paragraphs some disadvantages also prevent them from being cost and energy efficient.

3.1 Problem Statement

Server Virtualization

When virtualizing physical servers, IT organizations partition these servers into many virtual machines. Distributing resources through VMs reduces the number of physical hosts needed in a data center. Since most of these servers do not utilize their full capacity, using only between 10 to 50 percent of their potential [1], this method relieves the problem of underutilized physical hardware. In order to overcome this issue, operators sometimes overbook their servers by creating more VMs on the same host. However, this solution also has a side effect as overbooking the server with too many VMs may cause the server to overload, consequently decreasing its performance. In addition, overbooking also increases the complexity of the system.

Even when the resources are not being used, the operators must keep them available in case of a rise in the load, for example in a web or application server, when the amount of requests suddenly increases these VMs need to quickly respond with the minimum delay possible. Keeping the resources available even in idle mode not only wastes resources but also energy. Although being an effective approach,

it requires complex adaptation mechanisms. When migrating to different virtual machine to avoid overloading under dynamic circumstances, it must also keep the VM available while being idle or slightly loaded.

To avoid these issues, we propose a system based on heterogeneous physical hardware which adapts itself according to the scenario. It uses low power consumption machines, for low load and performance demands, and it gradually scales up to more powerful instances when the load increases and higher performance is required. With this approach, we try to approximate as much as possible to an ideal machine which when in idle mode consumes almost no power and when requested can be promptly available thereby proportionally consuming energy according to the amount of work performed (see Hölzle et al.) [1].

Server Adaptation

Another concern when creating VMs concerns resource allocation and how much should be provided in order to fulfill the SLA and at the same time avoid wasting resources. Some providers, such as Amazon [19] offer different instance types and features, such as small, medium or large with specific processor, memory, network performance etc. This type of mechanism is suitable for companies in which the requirements match one of the proposed types. However, for businesses in which the workload varies constantly and which need to suddenly either increase or decrease their resource, this method would not be appropriate, since every time more resources are needed, they either have to buy another instance or switch the old one to a larger one. Further, to book new instances and adapt them to the specific traffic also takes some time, therefore a mechanism to trigger the adaptation in time without overloading the instance and avoiding transition periods is needed.

With our heterogeneous approach combined with a predictive model, this issue could be resolved. We propose a predictive mechanism based on the ARIMA model using past values to forecast the future resources needed and according to those predictions act in advance by transferring the load to the specific machine capable of performing the workload at the right time, avoiding the machine becoming overloaded. If the predictions are either under- or over-estimated, another mechanism is also implemented in which each instance can communicate to the main controller warning the controller of its load, thus causing the controller to automatically respond by transferring the load to a more appropriate instance. This method will be detailed in the system architecture and in the implementation chapters.

Energy Consumption and Energy Efficiency

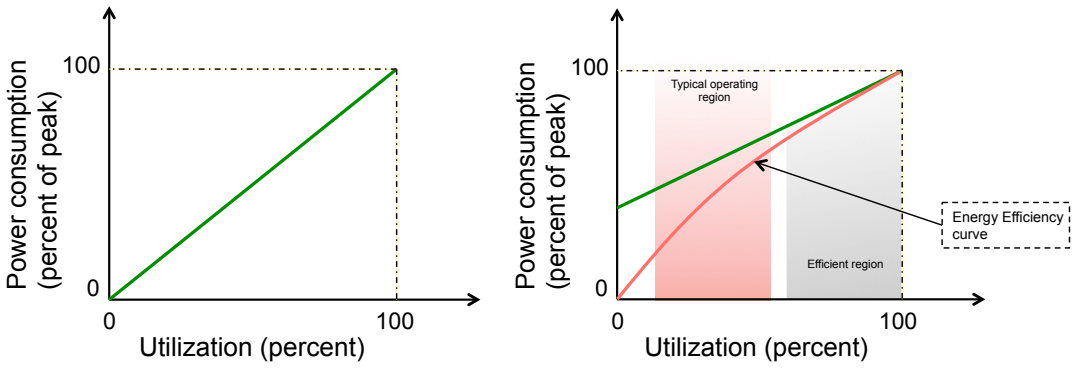
Cloud data centers also suffer as a result of the great amount of energy consumption. With more servers been stacked closely together in racks and inside closed warehouses, more electricity is required not only to power these machines but also to cool the equipment and infrastructure. Most of the electrical energy consumed by the servers is transformed into heat. During operations, a large amount of heat

is dissipated by the various components of the machine, such as power supply, memory, and mainly the processor. In order to keep these devices from overheating and within their safe operating temperatures, cooling methods need to be implemented.

Several methods are available today to cool the servers' components. The use of heat sinks is one of the methods that uses a thermal conductor to transport the heat away from the components to the larger surface of the sink, thus dissipating the heat and cooling the device. Other methods such as fans and water cooling are also implemented. Besides cooling the servers' components, larger and more powerful equipment is required to cool the entire infrastructure. Combined, these cooling systems may account for almost 40% of the energy consumption of a data center, for example in a 5,000 square foot data center research investigated by Emerson [4].

The servers not only consume a great amount of energy but they are also not energy efficient. The ideal host, according to Barroso et al. in [1], is energy-proportional when the energy consumption is proportional to the utilization or work done. Our approach attempts to get as close as possible to an ideal machine. As seen in Figure 3.1a, in an ideal system the power consumption would be proportionally distributed according to the utilization, but Figure 3.1b shows a real system, where it can be seen that even when the utilization is low the power consumption is very high.

Typically, a server starts becoming efficient above around 60 percent of its capacity utilization [1]; before that it consumes a lot of energy even though it is not being significantly utilized. From Figure 3.1b we can also see that the typical operating region of servers is between 10 to 50 percent of capacity utilization, therefore in order to avoid that gap of inefficiency we designed a model that uses low-power instances to fill that region.



(a) Ideal system

(b) Real system and energy efficiency curve

Figure 3.1: Server power consumption and energy efficiency at varying utilization levels, from idle to peak performance

Our system's approach implements an energy-proportional system by using low power machines such as SoCs, which consume only a few watts even in high load circumstances, and since it consumes low energy, the heat dissipation of these devices

is also very low, rendering the use of heavy cooling equipment redundant. Another advantage of our design relies on the fact that the high-power instances remain in a dormant mode when not being used, saving energy and dissipating less heat. In Chapter 6, we evaluate and demonstrate the power consumption of these devices and the energy efficiency of our system.

Network Elements

A network element of common use is load balancers. These devices are used to direct the traffic to specific servers according to the load, in order to avoid overloading a particular server. It can also be used for redundancy: if one of the machines fails the traffic can be directed to another device. The problem with load balancers is that besides increasing the amount of devices in the system, and thereby also increasing the energy consumption of the datacenter, they also add another point of failure to the system. In this case redundant load balancers are necessary to avoid this issue, which brings us back to the problem of too many devices.

Our approach involves the use of an SDN architecture, which helps to resolve some network issues. The OpenFlow controller together with OpenFlow-enabled switches can not only decrease the amount of devices in the network but also facilitate the configuration and maintenance of the network. The OpenFlow protocol interfacing the controller and the switches using a SSL connection facilitates communication among the devices connected to the network and helps to keep the controller informed of the network topology with information such as link and port status, switch availability, etc.

Transparency

The integration of many devices can be difficult, inefficient and sometimes cause gaps in system availability. In order to design a transparent system where the data center can be accessed by clients through a single IP address and a range of heterogeneous hardware can be integrated, we utilize SDN technology to perform the redirection of the traffic to different instances, a handover protocol to avoid unavailability during transitions and a predictive algorithm to determine the best time to switch before overloading the machine.

In the next section, we present the complete model of our system, combining the heterogeneous hardware solution with an SDN architecture and taking advantage of a predictive mechanism to improve the data center's resource allocation and energy efficiency.

3.2 System Model

To model our system, we extended Dürr's [2] design, where implementation is carried out by having one low-power SoC hardware accommodating the Low-Power Micro Instance (LPMI) and one commodity computer hosting the High Power Instance (HPI). To further improve this approach we add more instances, such as low, medium and high power, assuming the performance of the high-power hosts would be considerably better than the performance of the low-power ones and the medium-power hosts scale up in power and performance. Figure 3.2 provides an overview of the proposed model. In the next paragraphs we describe the main elements of the model.

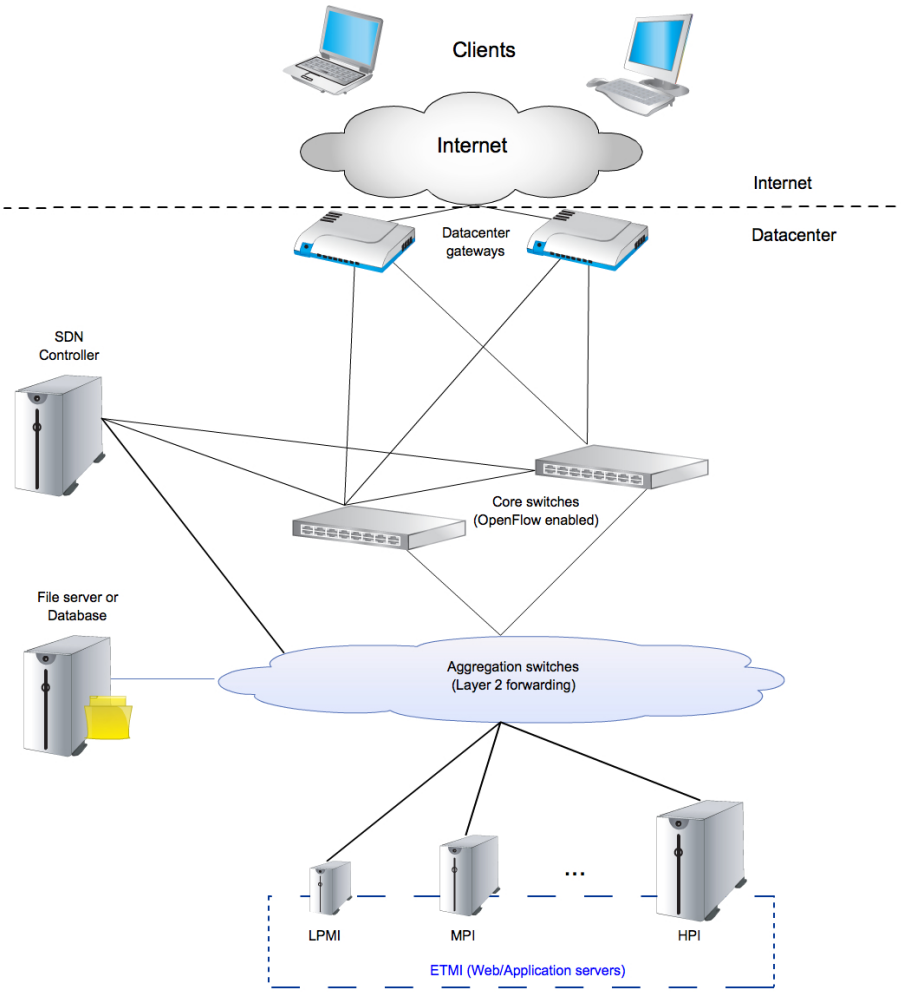


Figure 3.2: System model with interfaces and key components

Our approach considers a typical single data center acting as an IaaS provider and consisting of physical devices such as servers and network equipment. The servers which run the applications of the datacenter include: LPMIs, MPIs and HPIs. HPIs can host virtual machines (VMs), depending on the demand of the customer, since their performance is higher than the LPMIs. On the other hand,

in our case the LPMIs are not virtualized since their resources are limited. LPMIs have an advantage over HPIs when comparing their power consumption, as LPMIs are expected to consume only a few watts. However, their resources are reduced, with fewer memory, lower CPU and network speed, but still capable of hosting a web or application server, as shown in the evaluation section. Medium-Power Instances (MPIs) are machines that fall between LPMIs and HPIs, with not too many resources as compared to an HPI but at the same time more powerful than LPMIs.

For the SoC-based LPMI we take for example, the Raspberry pi, which is a low cost credit-card-sized single-board computer that has a Broadcom BCM2835 system on a chip and is based on the ARM 11 processor running at 700 MHz. The main storage device is an SD card and has a 512 MB SDRAM at 400 MHz. There is a 10/100 Mbps USB to Ethernet chipset and the board is powered by a 5V power source via a MicroUSB with a 322mA at idle and rated at 700mA [20]. Figure 3.3 shows the structure of the Raspberry pi and its components.

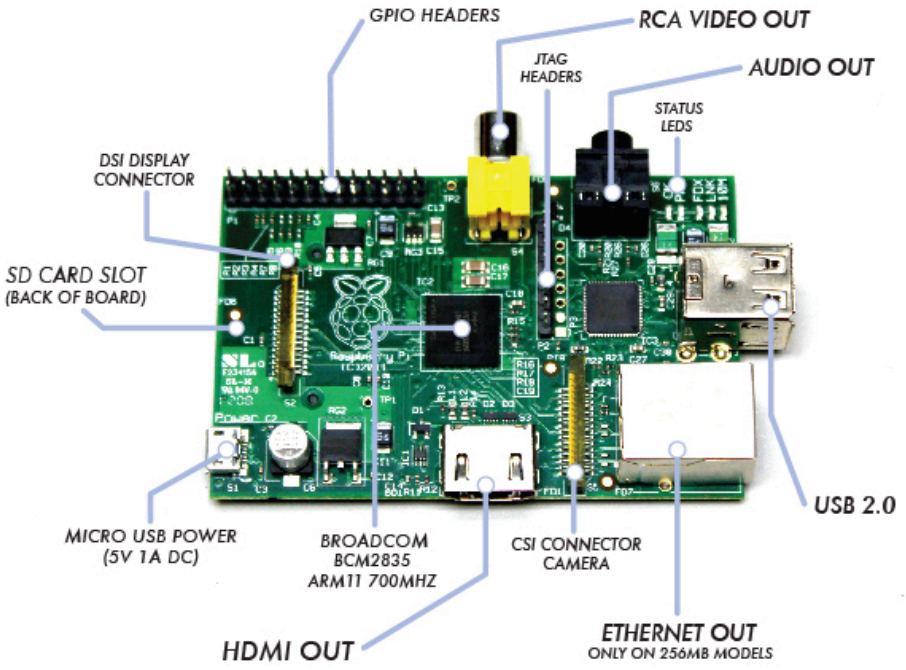


Figure 3.3: Raspberry Pi architecture and components[7]

Another SoC-based LPMI that the data center may contain is the BeagleBone Black board, which is designed with a low-cost ARM Cortex-A8 based processor running at 1GHz.

The BeagleBone Black offers an onboard 2GB eMMC and a microSD card as its main storage. A single 512MB DDR3L memory device is used and it operates at a clock frequency of 303MHz yielding an effective rate of 606MHZ on the DDR3L bus and allowing for 1.32GB/S of DDR3L memory bandwidth. A 10/100 Mbps Ethernet is the connection to the network. The board can be powered from four different sources: a USB port on a PC limited to 500mA, a 5VDC 1A power supply

plugged into the DC connector, a power supply with a USB connector and through expansion connectors [8]. A detailed structure and the key components of the board can be seen in Figure 3.4.

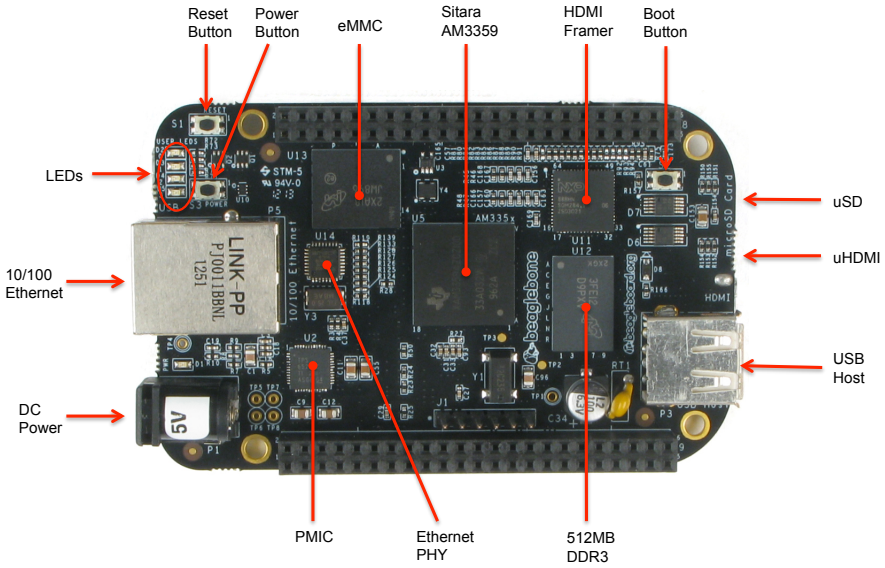


Figure 3.4: BeagleBone Black board architecture and components[8]

All of the instances (LPMI, MPI and HPI) are connected to the same data center network, which contains switches, routers, the OpenFlow controller and the file and database servers. In our network topology, we assume that the routers are the gateways connected to the Internet, the switches provide connectivity among the machines inside the data center and clients outside the data center and an OpenFlow controller is connected to the core switches, which are OpenFlow enabled. The other switches can be any multi-layer switch capable of forwarding packets between layers 2 and 4 through source and destination MAC or IP addresses and port numbers.

Since a number of network switch and router vendors already support the OpenFlow protocol, including Big Switch Networks, Brocade Communications, Cisco and IBM among others, our assumption of having our core switches being OpenFlow-enabled is plausible.

Our solution is focused on a typical three-tier architecture, where we use the middle tier services, such as web and application servers, to implement our approach. All of the instances, LPMI, MPI and HPI run these servers. Separate machines dedicated to store persistent data and state information, such as databases and file servers, are always running and are disregarded in the measurements of our optimization approach.

Since the hardware platform of the SoCs may differ from the high-power hosts, as the first uses ARM architecture and the last one x86, the transfer of state information between these servers can be complex. Accordingly, we assume that all information is stored in the backend servers (file servers or database) or in the client's machine, except for specific individual requests which the middle tier service will use as volatile

state information. For example, when clients want to access dynamic content on the web, in order to do so a server-side scripting, such as PHP or ASP, might request information from the database, handle it and write the results back to the database, all within the same request from the client. Other information such as HTTP cookies is stored on the client's machine. These assumptions are very realistic for web-based applications, such as LAMP (Linux, Apache, MySQL and PHP) or servlet engines connected to a persistent storage.

Chapter 4

System Design

This thesis extends the concept introduced by Dürr [2] which implements efficient and scalable machine instances. His approach uses a single LPMI and a single HPI to provide an Elastic Tandem Machine Instance (ETMI) and a non-disruptive handover protocol based on SDN technologies to transfer the traffic to the specific instance, either the LPMI or the HPI. In order to decide when to switch between devices, it uses a load monitor to indicate when the machines are either under- or overloaded. Our approach improves the ETMI by adding more instances between the LPMI and HPI, which we call Medium-Power Instances (MPI), in order to make the scale up or down smoother. We also add a predictive algorithm to determine in advance when the machines will be under/overloaded, switching between devices before the performance of the system decreases. In this chapter, we present the entire architecture of our system, combining some implementations performed by Dürr [2] and our extended achievements to the system.

4.1 Overview

As previously mentioned, we are interested in offering a system which is energy efficient and at the same time always available and capable of scaling up and down depending on demand. Through the ETMI we incorporate different instances, such as LPMIs, MPIs and HPIs to achieve that goal. We assume that every instance is connected to the same data center and they have the middle tier software installed and configured in the same way. They also share the same file server and databases. For example, in a web server, a request from a client to a certain service will have the same behavior regardless of the instance which responds to the request. A predictive algorithm will be running on the background to forecast the load and transfer the prediction to the controller, which will decide when and to which instance it should forward the traffic.

Since past values related to the load of the datacenter are needed in order to predict future loads, the prediction algorithm must wait until enough values are saved in order to best estimate the forecast. During that time, the transfers are made according to the current load on the instances. At first, assuming the load on the servers is low, only the LPMIs will be running, benefiting from the low power

consumption of the SoCs. As soon as the load increases beyond the performance of the LPMIs, an MPI is automatically booted; once the instance is ready new connections are transferred to the MPI, releasing the load on the LPMI. The same procedure will take place if the MPI becomes overloaded, but this time scaling up to an HPI (in case of multiple MPIs, it will be sent to a more powerful MPI). During the transition between instances, requests are still forwarded to the previous host until the next instance is available.

Traffic statistics are periodically collected from the OpenFlow switches, saved and used by the predictive algorithm. Once enough data is available, the prediction will be performed and sent to the controller for analysis. After the prediction has been analyzed, the transition between instances will occur according to the forecasted values. The original system will still be running in case a sudden change occurs which was not captured by the prediction. The transitions should occur as smoothly as possible, so for the client outside the data center they would be imperceptible. The control logic, which decides the path the traffic should flow, is implemented in software by the SDN controller, which configures the core switches to perform the forwarding of the packets in the communication network.

Even though the ETMI is formed by n instances, it is accessible by only one public IP address, so for the outside world it looks like a single machine. One of the difficulties in making this transparent to the client is during the handover process, where depending on the protocol used to communicate with the data center, e.g. HTTP requests over TCP, connections may still be established between one of the instances and the client. In such a case, to avoid breaking these connections and since our approach does not implement VM migration, we have to make sure these connections are still sent to the original instance until they are naturally disconnected, while the new connections are forwarded to the new host. In the next section, we will describe in detail the complete architecture, the predictive algorithm and the handover protocol implemented in our approach.

4.2 ETMI and Handover Algorithm

Our system is designed as a single data center, consisting of network equipment such as switches and routers, backend servers such as file servers and databases, the ETMI and a server running the predictive algorithm. Part of the network is based on SDN architecture, which contains the main element of our approach, the SDN controller. The core switches are OpenFlow-enabled, so they can communicate with the SDN controller through the OpenFlow protocol. Aggregation switches connect the ETMI to the core switches; these may not be OpenFlow enabled since they only need to perform layer 2 forwarding. Virtual switches connecting the VMs may also be implemented. The SDN controller is not only connected to the core switches but also the aggregation ones for the internal communication. The backend servers are connected to the network and are accessible by every instance of the ETMI. The LPMIs, MPIs and HPIs form the ETMI; these servers are connected to the network through the aggregation switches, therefore they do not need to be physically close

to each other. Since the predictive algorithm is implemented in software, it does not necessarily need to be on a separate server; in our implementation it runs on the same machine as the SDN controller. Figure 4.1 illustrates the architecture of our system, the key components and the interfaces that connect them.

The core switches are responsible for forwarding the traffic to the appropriate instance in the ETMI. They do it according to their forwarding tables updated by the SDN controller through the OpenFlow protocol, which decides, based on the prediction, the best instance to redirect the load to. In order to keep the network consistent, the SDN controller configures all of the core switches with the same forwarding table entries. The SDN controller is also responsible for periodically collecting traffic statistics, e.g. bytes and/or packets received/transmitted, from the switches. This information is used by the predictive algorithm to forecast the load.

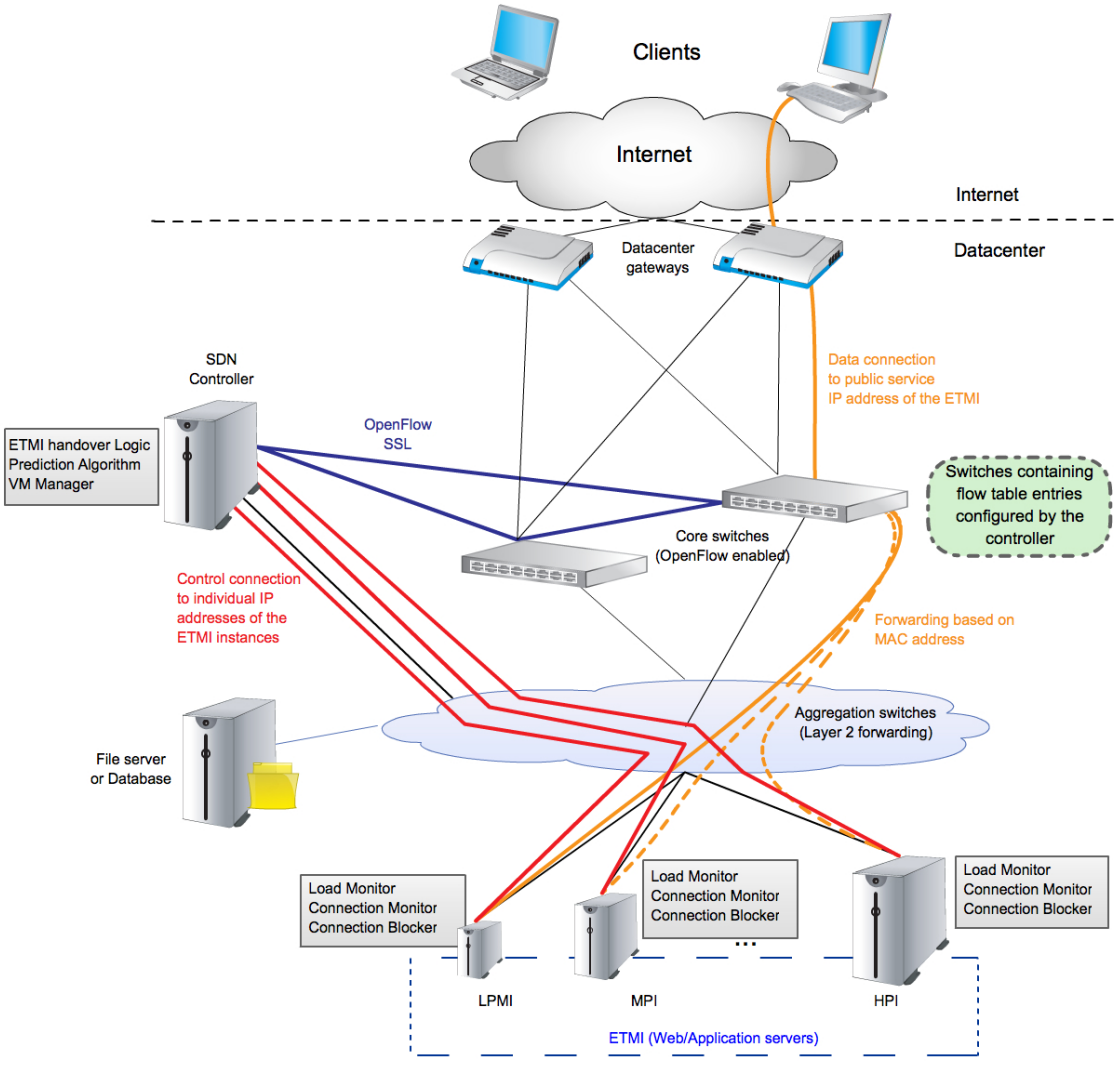


Figure 4.1: System architecture

The ETMIs are each configured with two different IP addresses. One is a private and distinct IP address, which is used for internal communication among the backend

servers and the SDN controller. The other one is the public IP address of the service, which is the same for every instance of the ETMI. Since the SoCs only have one network interface, both IPs are assigned to the same NIC, applying a method named IP aliasing. To redirect the traffic from the client to the appropriate instance, the core switches perform a MAC address rewriting, which is previously configured by the controller. This process is explained in detail further on.

According to the traffic statistics collected by the controller, the predictive algorithm reads these statistics and applies an ARIMA model to forecast the traffic load for the next period. It then, according to the prediction, decides which instance is more suitable for the load at each specific time. After making the decision, the algorithm sends the result to the controller. The controller will then apply the handover algorithm to update the flow table entries of the core switches to forward the traffic to the specific instance.

As previously mentioned, if the prediction is not accurate and the load is either higher or lower than predicted, the system can still adapt based on the approach proposed by Dürr [2] with a few modifications. A *load monitor*, installed and executed on each instance of the ETMI, determines the load of the particular instance and constantly updates the controller about its status by sending either under- or overloaded messages. To decide when a server is under/overloaded, threshold values are defined for each instance. Different metrics can be implemented to measure the load on the instances, for example CPU load, request rate etc. Upon receiving the messages from the instances, the controller decides where to redirect the flow. Mechanisms to avoid too many changes in a short period of time are also implemented. Since the HPIs and possibly the MPIs (depending on the machine used) need to boot every time the traffic is forwarded to them, we added a function that first verifies if the instance is underloaded in case old connections are still loading the host; if not we check if the redirection has happened within a 120-second interval. A Virtual Machine Manager is responsible for starting the MPIs and HPIs after receiving the command from the controller. The manager also informs the controller when the instance is ready to receive requests.

To clarify in more detail how the handover and the prediction algorithms work, we present them in the next sections.

Handover Algorithm

Since the new model contains more than two instances and a prediction process has been added to the design, the new handover algorithm contains a few modifications from the previous handover protocol implemented by Dürr [2]. First, the algorithm must identify to which device it should redirect the traffic, according to the prediction received from the predictive algorithm. Second, it should transfer the traffic at a specific time before the instance is overloaded. This last step is performed by the prediction controller, which will be introduced in the implementation chapter. The handover algorithm is formally presented in Algo. 4.1 as a pseudo-code, followed by the detailed explanation of the steps required to perform the operations.

Algorithm 4.1 Handover Algorithm

```
1: procedure ONHANDOVERALGORITHM
2:   newInstance ← prediction
3:   if newInstance ≠ previousInstance
4:     VMManager.bootAndWaitForResponse()
5:     ConnectionBlockerPreviousInstance.blockSynRequests()
6:     C ← ConnectionMonitorPreviousInstance.getConnections()
7:     for all c ∈ C do {pin established connections}
8:       addFwdTableEntry:
9:         match[c.asrc, c.adest, c.portsrc, c.portdst] →
           action[dstMac = mPreviousInstance]
10:    end for
11:    modifyFwdTableEntry: {forward new connection to newInstance}
12:     match{apublic} → action[dstMac = mNewInstance]
13:    ConnectionBlockerPreviousInstance.unblockSynRequests()
14:  end if
15: end procedure
```

The algorithm is responsible for redirecting the traffic to the correct instance. Since our approach is based on an SDN technology, the controller is able to statically configure the switches' flow table entries to specify the forwarding path that the packets should follow. As mentioned earlier, every instance of the ETMI is configured with the same public IP address; to distinguish among the instances we use their individual MAC addresses.

Whenever requested, the controller will push a static flow table to the switch with the action of rewriting the MAC address of incoming packets that match the destination public IP address of the data center. In this way, different flows will be installed according to the instance that should receive the traffic and the old flows will be deleted from the table. Thus, when a packet arrives at the switch with the destination public IP address of the data center, the core switch will write the MAC address of the corresponding instance and forward the packet to the aggregation switches; since the aggregation switches perform layer-2 forwarding the packets will be sent based on their MAC addresses. The action of rewriting the MAC addresses is only necessary for incoming packets, since every instance contains the same public IP address; through IP aliasing, outgoing packets will always be forwarded to the client from the same source IP address.

Before redirecting the traffic to a specific instance, the controller must identify established connections to avoid breaking them when forwarding the traffic to a new host. To accomplish that, two components installed on the instances help the controller to identify and block the new connections to the host. One is the *connection monitor*, which upon request from the controller will detect all of the established TCP connections between the client and the instance and send them to the controller. The other element is the *connection blocker*, which is responsible for blocking new requests to the instance during the handover between instances.

The following steps demonstrate the procedures of the handover algorithm once the controller is triggered by the ETMI.

Step 1 – Identifying the new instance:
The new instance is selected according to the prediction received from the predictive algorithm. Before going to the next step, it first checks if the new instance is the same as the old instance; if it is the next steps do not need to be performed.

Step 2 – Booting new instance:
If the new instance is not an LPMI, which should always be running, the controller either sends a request to the VM Manager to start the instance or it sends a Wake On LAN (WOL) packet to the desired machine. Once the machine has been booted and is ready, the controller is acknowledged.

Step 3 – Pinning established connections:
Before transferring the new connections to the new instance, the controller needs to make sure that old connection are still forwarded to the former instances and not destroyed. To achieve that, the controller requests from the *connection monitor*, of the former instance, all of the established TCP connections. The *connection monitor* replies to the controller the source and destination IP addresses and port numbers of each TCP connection. In order to maintain those connections, the controller pins them by updating the flow tables of the switches adding a new forwarding rule, with a higher priority to rewrite the MAC address of only those connections. By applying a higher priority, we assure that these packets will be transferred to the original instance even after the new flows are added in the next step.

Step 4 – Blocking SYN requests:
In order to avoid a race condition that could lead to broken connections during the handover, when connections might still be established between a client and the previous instance, after the controller has requested the *connection monitor* for opened connections and before the connections were pinned. To solve this issue, before pinning the old connections, the *connection blocker* applies a firewall rule to drop SYN messages received by the server; in this case no new connections will be established with the old instance. During the blocking period some packets could be lost, but it would not be a problem since IP packets are retransmitted after a small timeout, and by then it would be forwarded to the new instance as performed in the next step.

Step 5 – Redirecting new connections:
After blocking new connections and pinning the old ones to the old instance, the controller installs a new forwarding table entry on the switches that matches the public destination IP address and rewrites the MAC address to the new instance. This new flow will have a lower priority than the specific pinned flows of the old connections. In this way we assure that the packets are being forwarded to the correct instance. After forwarding the new connections to the new instance, the *connection blocker* can unblock the old instance to receive SYN messages.

As a final stage to prevent the switches from having a large amount of flow tables and to save space in their memories, the controller removes the flow table entries of the pinned connections of *step 2*. To do so, the controller checks the status of the connections by periodically querying the *connection monitor* of the related instance to identify the closed connections and remove its entry from the switches.

4.3 Predictive Algorithm

Our predictive algorithm uses past history of the system load to build a time-series model from it. An analysis of the time-series is then implemented to build a prediction model. To analyze the time series and predict it, we use an ARIMA model. For our analysis, we considered the bytes received per unit of time to define the metric of the load on the system. The bytes received are periodically captured by the controller from the core switches. The period can be defined manually, depending on the load of the system. After collecting the information from the switches, the controller saves it in a file. The predictive algorithm reads the file and creates a time-series object out of the saved information and according to the period desired. Once the time-series object is generated, an ARIMA model is constructed through a variation of the Hyndman and Khandakar algorithm [10, 21], combining unit root tests (to check the stationarity of the time-series), a minimization of the AICc and a Maximum Likelihood (MLE) method to estimate the parameters of the ARIMA model. An ARIMA model, as seen previously, can be of the type $ARIMA\,(P, D, Q)\,(p, d, q)_m$ for seasonal data, where m is the seasonal frequency, and $ARIMA\,(p, d, q)$ for non-seasonal data. The sequence of operations of the predictive algorithm is introduced in Algo. 4.2, followed by the description of each step.

Algorithm 4.2 Predictive Algorithm

```
1: procedure ONPREDICTIVEALGORITHM
2:   read file
3:   ts ← generateTimeSeriesObject()
4:   fit ← fitARIMAModel(ts, frequency)
5:   forecasts ← forecast.ARIMA(fit, periodsAhead)
6:   return forecasts
7: end procedure
```

Step 1 - Generating time series:

The saved file containing the load information is read and, according to the configured number of observations per unit of time, a time series is generated.

Step 2 - Fitting into an ARIMA model:

Once the time series is generated, it is time to fit it into an ARIMA model, following the procedures of the automatic Hyndman and Khandakar algorithm [21]:

***Selecting** d **and** D:* The Canova-Hansen test [22] is performed, even though it includes seasonal dummy terms it can still be applied to choose D in a strictly ARIMA framework, as tested by Hyndman et al. [21]. After selecting D, successive Kwiatkowski–Phillips–Schmidt–Shin (KPSS) unit-root tests are executed to determine the number of differences d. For models where $d + D < 2$, the constant c, in equation 2.10, can be different than zero.

***Selecting** p, q, P **and** Q:* To select p, q, P and Q, every potential model would have to be fit and the model with the lowest AICc would be the best. Since trying every single possibility would not be feasible and it would take a lot of time and processing, a set of the four most common models are tested first:

- $ARIMA\,(2, d, 2)$ if $m = 1$ and $ARIMA\,(2, d, 2)\,(1, D, 1)$ if $m > 1$
- $ARIMA\,(0, d, 0)$ if $m = 1$ and $ARIMA\,(0, d, 0)\,(0, D, 0)$ if $m > 1$
- $ARIMA\,(1, d, 0)$ if $m = 1$ and $ARIMA\,(1, d, 0)\,(1, D, 0)$ if $m > 1$
- $ARIMA\,(0, d, 1)$ if $m = 1$ and $ARIMA\,(0, d, 1)\,(0, D, 1)$ if $m > 1$

After testing with these models if $d + D \leq 1$ then constant $c \neq 0$, or else $c = 0$. The model with the smallest AICc value will be denoted the "current" model and if frequency $m = 1$ the model will look like an $ARIMA\,(p, d, q)$ and if $m > 1$ $ARIMA\,(P, D, Q)\,(p, d, q)_m$. Besides the four models, another set of thirteen alternatives from the current model is considered:

- where one of p, q, P and Q is allowed to vary by ± 1 from the current model
- where p and q both vary by ± 1 from the current model
- where P and Q both vary by ± 1 from the current model
- where the constant c is included if the current model has $c = 0$ or excluded if the current model has $c \neq 0$

After each variation, the AICc is verified and checked against the current model; if it is lower, then the model becomes the new "current" model. This procedure is repeated until the lowest model is found.

To avoid convergence and near unit-roots problems a few constraints are applied to the fitted models:

- Upper bounds are defined for p, q, P and Q, with p, q not being higher than 5 and P, Q not being higher than 2.
- The absolute values of the roots of $\phi\,(B)\,\Phi\,(B)$ and $\theta\,(B)\,\Theta\,(B)$ should be smaller than $1,001$, or else the model is rejected since it approximates to non-invertible or non-causal.
- The non-linear optimization routine used for estimation should have no errors otherwise the model is also excluded.

Step 3 - Forecasting:

After a model is finally selected, we use it to predict our workload. The number of periods for forecasting can be configured manually. Once the forecasted points are returned, they are used to decide the best instance to serve the predicted load. The forecast function also returns prediction intervals of 80% and 95% upper and lower limits, which can also be selected before choosing the instances.

For clients accessing the data center, for example a web server, they are interested in the time the server takes to respond to their requests. According to an acceptable response time, each server is given a threshold of how many bytes it can handle before affecting its performance. Based on the prediction and the threshold values of each server, the predictive algorithm decides which server will best perform for the specific time. After the decision is made the algorithm sends the information to the controller which will be in charge of changing the flow table entries of the switches at the specific time, redirecting the flow to the appropriate instance.

Chapter 5

Implementation

In Chapters 2 and 3 we modeled and designed the architecture of our approach, stating the components, their interfaces and the protocols they use to interact with each other, to provide a more efficient data center. In this Chapter, we describe the implementation of the components in a prototype system.

There are three separate parts of our testing system: One is the hardware part, consisting of the servers and switches; another is the software, involving the controller and its modules for the SDN topology, the operating systems (OSs) and the other software developed, such as the predictive algorithm, the prediction controller, connection monitor and blocker and the load monitor. Lastly, since we are concerned with providing an energy-efficient data center, we have designed an electronic circuit to measure the current flowing in the device in order to calculate its power consumption. In the next sections we describe each part of the system.

5.1 Description of the Hardware

In order to replicate a single data center, in a condensed version, we assembled a set of hardware in our implementation that is mainly formed by servers, which form our ETMI solution, core and aggregation switches and backend servers for service such as Network File System (NFS) and database. To compose our ETMI we use 3 types of devices, each of them to simulate the LPMI, the MPI and the HPI. Our backend service runs on the same machine as the SDN Controller and a dedicated machine runs software to simulate clients accessing the datacenter. The following list describes the technical specification of each server of our test bed:

LPMI
Machine: Raspberry Pi
CPU: ARM 11 at 700 MHz
Memory: 512 MB SDRAM
Network: 10/100 Mbps USB to Ethernet interface

MPI
Machine: BeagleBone Black
CPU: TI Sitara ARM Cortex-A8 at 1GHz
Memory: 512 MB DDR3
Network: 10/100 Mbps Ethernet NIC

HPI
Machine: Commodity computer
CPU: AMD Athlon 64 X2 dual-core processor 4200+ at 2.2 GHz
Memory: 2 GB RAM
Network: 2 × 1 Gbps Ethernet NICs

Controller
Machine: Commodity computer
CPU: Intel i5 quad-core at 2.67 GHz
Memory: 12 GB SDRAM
Network: 5 × 1 Gbps Ethernet NICs

Client
Machine: Commodity computer
CPU: AMD Athlon 64 X2 dual-core processor 4200+ at 2.2 GHz
Memory: 2 GB RAM
Network: 2 × 1 Gbps Ethernet NICs

For the LMPI we used the Raspberry Pi SoC machine. It is well suited to our solution as it can perform fairly well as a web server, as shown in our evaluation chapter, and it also consumes very little power, around 2W. The BeagleBone Black is our next instance, We used it as an MPI since it is more powerful than the Raspberry Pi and according to our tests it could handle more requests, but it could also be used as a secondary LPMI since it is an SoC with a very low power consumption. A more powerful computer was used for the HPI, which consumes more power than the other two instances. At first this machine will be in a sleeping state until the controller requests it to be awakened. The controller server is the main machine of our implementation, as it runs not only the SDN controller but the predictive algorithm, the backend services, the VM manager, and the Open vSwitch Bridge. The client host runs software to generate the load (requests) on the data center, either according to a Poisson or a uniform distribution.

Two different switches are used in our test bed. One is a basic Layer-2 forwarding switch that connects all of our devices in a private network, and the other is an Open vSwitch, which is installed on the same machine as the SDN controller. Since it has 5 network interfaces, we are able to use some interfaces for the Open vSwitch bridge. One interface connects the controller to the aggregation switch used for the internal communication among the servers and two other interfaces are configured for the Open vSwitch bridge, one connecting the client to the datacenter and the other connecting the core switch to the aggregation switch. Figure 5.1 shows the topology of our test bed.

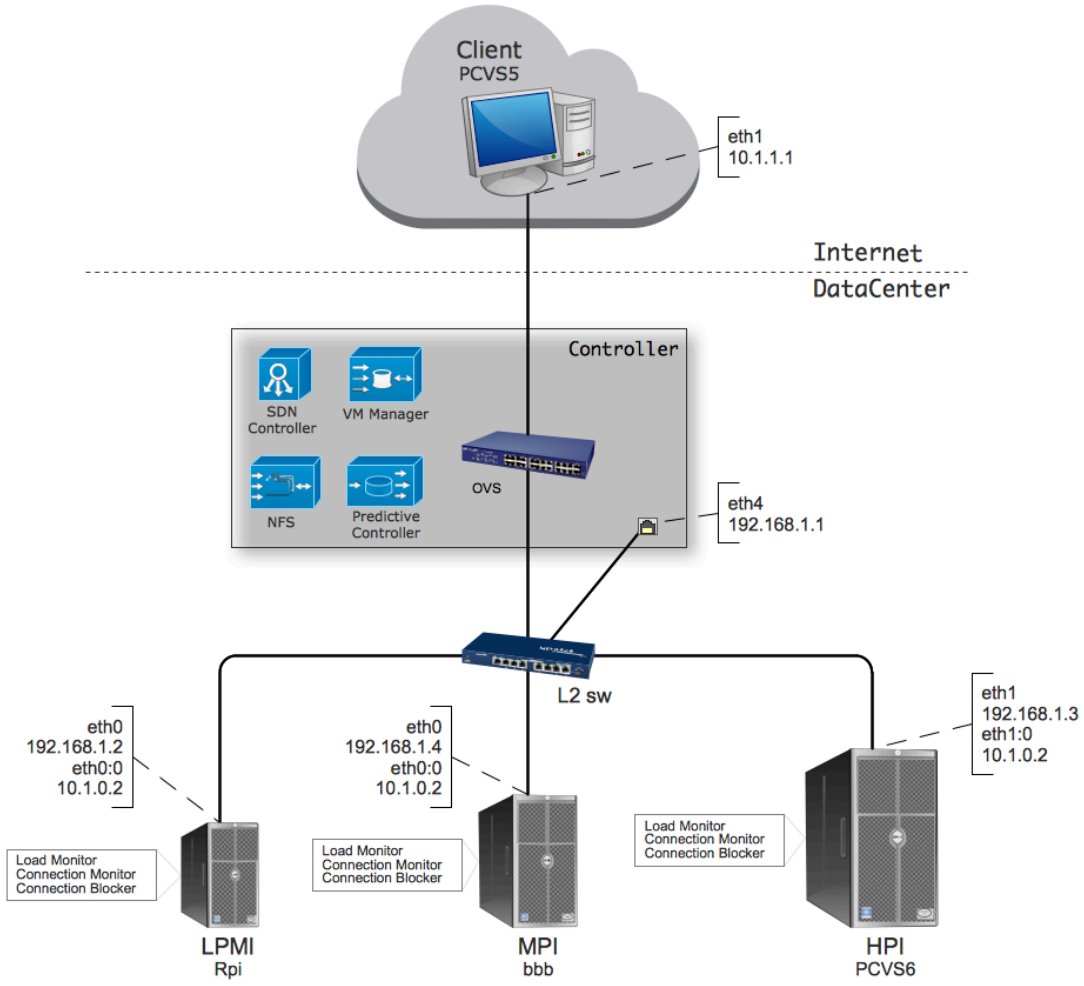


Figure 5.1: Test bed topology.

5.2 Description of the Software

Regarding the operating system, every server runs a Linux distribution. The HPI, the controller and the client run Ubuntu release 12.04. The Raspberry Pi and the BeagleBone Black run Debian wheezy version 7.0. Every server on the ETMI is configured in the same way and they all run a typical web-based application like LAMP, where the operating system is **L**inux, the http web server is **A**pache, **M**ySQL is the database management server and **P**HP is the scripting language. Besides the web application installed on the ETMI, a few other applications, based on Dürr's implementation [2], run on the servers and are described below:

Connection Monitor - is a c/c++ application designed to collect the set of TCP connections of the corresponding server. It is triggered by the controller and it not only replies to the controller the established TCP connections but also the ones where the synchronization message (SYN) has been received; it does that through the socket statistics (ss) command.

Connection Blocker - runs a bash script to either block or unblock connections to the corresponding server. When triggered by the controller, it creates a rule on the server's firewall IPTABLES to drop all the SYN packets received from the public IP address. If the controller needs to unblock the connections, it sends an unblock message and the rule is simply deleted.

Load Monitor - also runs a bash script; it constantly monitors the load on the server, creating a rule on the IPTABLES to count the packets and bytes received, where the destination is the public IP address.

The Controller is the main component of the system, as this machine has the most important elements installed on it. First, the SDN controller: We implemented an extension of the Floodlight OpenFlow controller [6], for which we use the REST API to set up the flow table entries on the OpenFlow switches, and also a separate module we developed to collect traffic statistics from the core switches. Second, the Predictive controller: a Java application responsible for performing the handover algorithm either based on the prediction of the workload sent by the *predictive algorithm* or based on the immediate workload sent by the *load monitor*. Thirds, the NFS file server, where all of the data is stored. Besides the three main components in our test bed, the *predictive algorithm* also runs on this machine.

To collect the traffic statistics from the Open vSwitch, we implemented a separate module in the Floodlight controller. It runs a distinct thread, named *StatsThread*, that is initialized when the floodlight controller starts running. The *StatsThread* runs a timer task that periodically collects the bytes received from the core switches and saves it in a separate file, called *"swstats"*. The timer task needs to be manually configured with a couple of parameters in order to provide the correct information that later will be used by the *predictive algorithm*. These parameters include, the starting time of the thread and the period in between successive task executions. This configuration is carried out according to a previous analysis of the workload of the data center. For example, a web server receives higher requests during the day than in the night, presenting signs of a 24 hours seasonality. In this case we could start the timer task at midnight and collect the traffic statistics every hour, building a time series with a frequency of 24. This way our *predictive algorithm* could forecast, for instance, the 24 hours of the next day, based on the past values saved by the *StatsThread*. Algorithm 5.1, shows the sequence of operations to start *StatsThread* on the Floodlight controller, and once it is started algorithm 5.2 describe its procedures.

Algorithm 5.1 Floodlight Controller Module

1: **procedure** ONFLOODLIGHTCONTROLLERSTART
2: load net.master_thesis.loadstats module
3: inside loadstats
3: **start**StatsThread
4: **continue** loading modules

Algorithm 5.2 StatsThread

```
1:  procedure ONSTATSTHREAD
2:    schedule Timertask
3:      start Timertask
4:        inside Timertask
3:          collect traffic statistics from SW
4:          save statistics in swstats file {save timestamp and bytes received}
2:      exit Timertask
2:      wait until next execution
2:  end procedure
```

The *AutoArima* is a Java application which runs the *predictive algorithm*. It predicts the workload of the data center, based on the traffic statistics collected from the file saved by the *StatsThread*, using an ARIMA model. The *AutoArima* needs to be configured with the same frequency value specified in the *StatsThread* in order to create an appropriate time series. *AutoArima* also runs a timer task which is configured with a starting time, at which the task is to be executed, and the period, which is the time between successive task executions. Once the task is started, it performs the *predictive algorithm* explained in Chapter 4. Upon completion of the prediction, the decision of the most suitable server takes place. To decide which server is the most suitable for the specific predicted value, we previously tested the servers to measure how many bytes they can handle before decreasing their performance. The application then compares the predicted values with the measured ones and creates a string array that contains the decision of the specific server for each predicted value; the length of the string array is the same size as the number of predicted values.

The *predictive algorithm* was implemented using the R programming language. With its variety of packages and functions, it improved our implementation of the statistical analysis. Packages such as the *"forecast"*, developed by Hyndman et al. [23], provides a variety of functions such as *forecast(), arima(), Arima(), auto.arima(), predict()* among others, which helped us to model, analyze and forecast time series. Another package, such as *"stats"* provides the *ts()* function, used to create time series objects. In order to integrate R into our Java application, we used the Java/R Interface (JRI) which allows us to run R inside Java applications as a single thread. It mainly loads the R dynamic library into Java and provides a Java API to R functionality. JRI comes bundled with an R package named rJava. To take advantage of the interface, we can simply import two main classes REngine, which is the interface between an instance of R and the Java VM; and REXP, used to encapsulate and cache R objects as returned from R [24].

To clarify the operation of the *AutoArima,* we can take our earlier example where we mentioned a web server with a 24 hours seasonality. Once we know that, we can configure the *AutoArima* to run at the beginning of the day and collect the values saved from the previous day(s) to forecast the 24-hour of the coming day. In this case, the algorithm will return 24 values which will then be compared to

the workload of each instance to decide which instance should serve at each hour. Algorithm 5.3 explains the steps of the procedures of the *AutoArima* application.

Algorithm 5.3 AutoArima application

```
1: procedure ONAUTOARIMA
2:   read configuration file {file containing over/underload info of every server}
3:   initialize R System
4:   start Timertask
5:    inside Timertask
6:     read swstats file
7:     prediction ← predictiveAlgorithm(swstats)
8:     makedecision(prediction) {select best server according to traffic load}
9:      i ← 0
10:       for each p[i] in prediction do
11:        if p < overloadA then
12:         p[i] = "A"
13:        else if p < overloadB then
14:         p[i] = "B"
15:        else if p >= overloadB then
16:         p[i] = "C"
17:        end if
18:        i++
19:       end for
20:       send p {send array including ordered servers to PredictionController}
21:      exit makedecision
22:   wait until next execution
23: end procedure
```

In line 20 of Algorithm 5.3, the string array containing the sequence of the predicted instances is sent to the PredictionController. Knowing the frequency of the time series and the number of predicted values, at a specific time the PredictionController sends the new flow table entry that directs the traffic to the predicted instance to the OpenFlow switches. Following our previous example, the PredictionController receives a string array containing 24 values, since the frequency is 24 hours and there is a change every hour. The PredictionController runs a timer task every hour and checks the instance the traffic is being sent to. If the instance is the same as the one running, the controller does not update the flow table entry of the switch(es), but if a new instance is required, the controller, after waking the new instance up (if necessary), sends the switch(es) the new flow table entry, transferring the traffic to the required server. The operations of the PredictionController can be seen in Algorithm 5.4.

Algorithm 5.4 PredictionController

```
1: procedure ONPREDICTIONCONTROLLER
2:   clear static flows
3:   state <- LPMI
4:   direct flows to state
5:   listen to socket {receive messages from the servers and AutoArima}
6:   if msgReceived = prediction
7:     start Timertask {run task periodically}
8:       p[] <- prediction
9:       n <- 0
10:      newServer <- p[n]
11:      if newServer = state
12:        return
13:      else
14:        handoverAlgorithm(newServer)
15:      end if
16:      n++
17:      wait until next execution
18:  else
19:      checkserver&message {check the message(over/underload) and the server
which sent the msg, identify if a switch is needed}
20:        if switchNeeded
21:          newServer <- server
22:          previousHandoverProtocol(newServer) {basic handover protocol with-
out prediction}
23:        end if
24:  end if
25: end procedure
```

A backup system will also be running in case the prediction was either under- or over estimated. Every server runs the *load monitor*, in case the server gets overloaded during its hour; it can also send a message to the controller saying it is overloaded. In that case, the controller will then switch the traffic to the more powerful server.

Figure 5.2 shows the sequence diagram of the proposed system running inside the controller host.

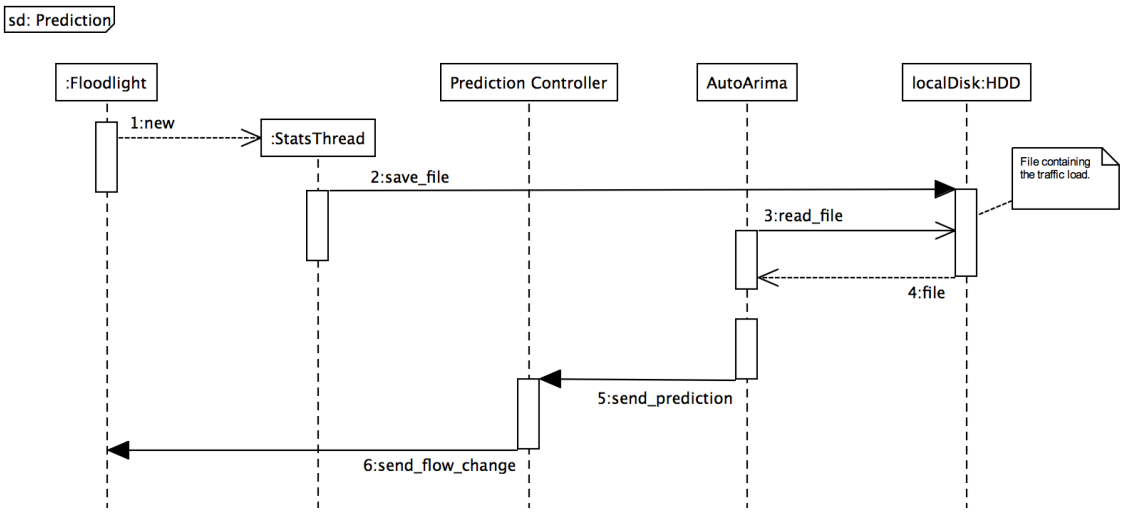


Figure 5.2: Sequence flow diagram

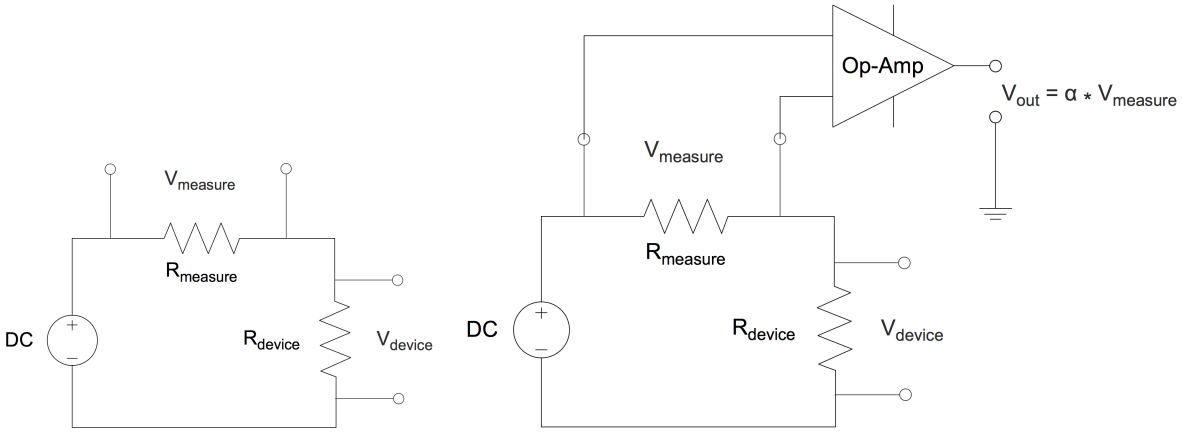
5.3 Measurement Circuit

There are a few methodologies to evaluate the power consumption of a device. One approach would involve simulations using the data given by the technical specification of the equipment. This approach would be suitable for large scale settings, but it could also neglect essential details of a real system. Another approach would be through real measurements utilizing real hardware; in this case a more realistic scenario can be implemented and tested providing the effects of the actual hardware. On the down side, only scenarios with limited devices can be implemented. Fortunately, for our test case, we could design our own circuit in order to make our measurements.

The parameter we need to calculate is the power consumption. To calculate it, we need to multiple the voltage and the current of the device. In our case the voltage is constant, therefore we just need to measure the current. By calculating the power of these devices we can determine a more realistic effect of the hardware we are utilizing.

Figure 5.3a shows the proposed current measurement circuit. By using Ohm's law, $I = V/R$, we can calculate the current flowing through the circuit, hence, calculating the power of the device using equation $P = V \times I$ [2.1]. Since a new resistor was added to the circuit, it becomes a voltage divider, where the voltage on the device is $V_{device} = V_{supply} - V_{measure}$. In order to keep the voltage on the device as low as possible, the resistance on $R_{measure}$ should also be as low as possible. In our design, we use a shunt resistor of $0.01\,\Omega$, $1\,W$ of power and a precision of 1% [25]. From the specification of the Raspberry Pi and the BeagleBone Black, the input voltage of these devices is $5\,V$ and the maximum current is $700\,mA$ for the Raspberry Pi [20] and $1\,A$ for the BeagleBone black [8]. Considering these values, the maximum voltage drop on the circuit for the Raspberry Pi would be $V_{measure} = 700\,mA \times 10\,m\Omega = 7\,mV$, and for the BeagleBone Black $V_{measure} =$

$1\,A \times 10\,m\Omega = 10\,mV$. According to these values, the voltage on $R_{measure}$ is too low, therefore we need to amplify $V_{measure}$, by adding an Operational Amplifier (Op-amp), which increases the voltage by a gain of α ($V_{out} = \alpha * V_{measure}$), Figure 5.3b shows the new circuit.



(a) Basic circuit (b) Op-Amp circuit

Figure 5.3: Circuits

For a better accuracy, we decided to use an instrumentation amplifier from Texas Instruments with a 3 op-amp design and only a single resistor that sets the gain from 1 to 10,000; the model of the op-amp is INA 114 [26]. From the technical specification of the op-amp, the gain is measured by the following formula:

$$G = 1 + \frac{50\,k\Omega}{R_{gain}} \tag{5.1}$$

We opted to offer two different gains, one of 100 and the other of 1,000, to provide a bigger range of options. By applying these gains to equation 5.1, we need two different resistors R_{gain}, one of $50.05\,\Omega$ and the other of $505.05\,\Omega$, respectively. To provide these resistances we use two types of potentiometers one that goes up to $100\,\Omega$ and the other until $1\,k\Omega$. The final circuit with the addition of a few other components, such as a DC/DC converter to provide $\pm 15\,V$ for the operational amplifier, a rectifier to have a polarity free input and a voltage regulator to have an input voltage range from 8 to 15 volts, is depicted in Figure 5.4.

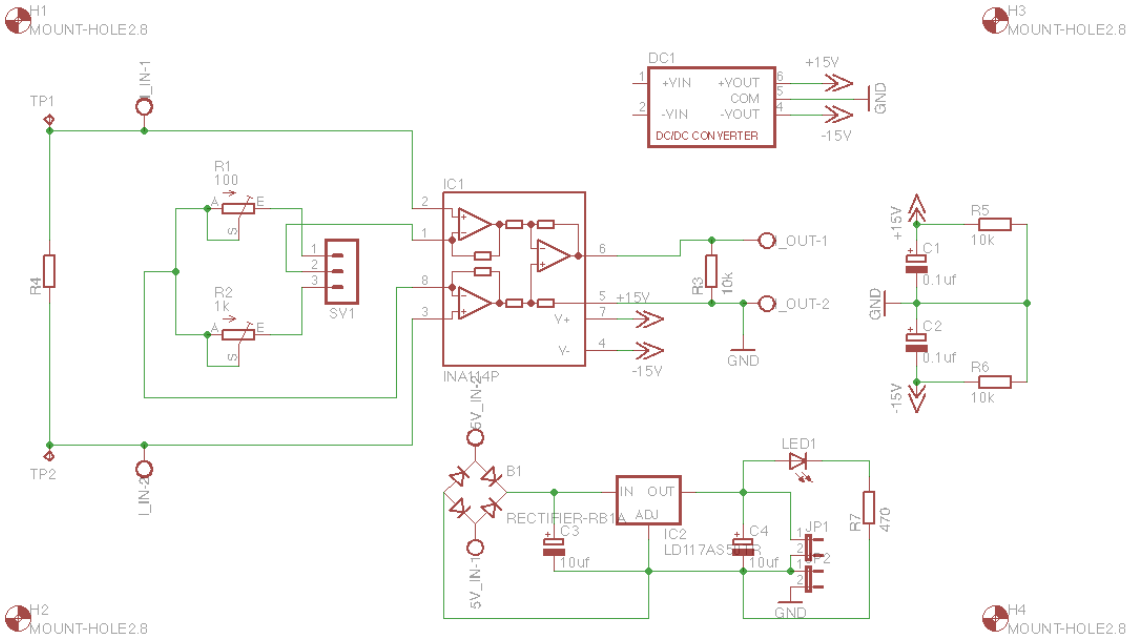


Figure 5.4: Final circuit

To complete our implementation, additional hardware and software were necessary, for example, a power supply to provide the necessary power to operate the circuit, a PC board and an I/O module, to connect the circuit to the computer, Using an application such as ME-Powerlab3 [27], we were able to automatically and more precisely collect the values measured. After collecting the values from the device, with a frequency of 1kHz, under different workloads, we inserted the values in a script to calculate the power of the device. Figure 5.5 shows the final implementation of the real circuit.

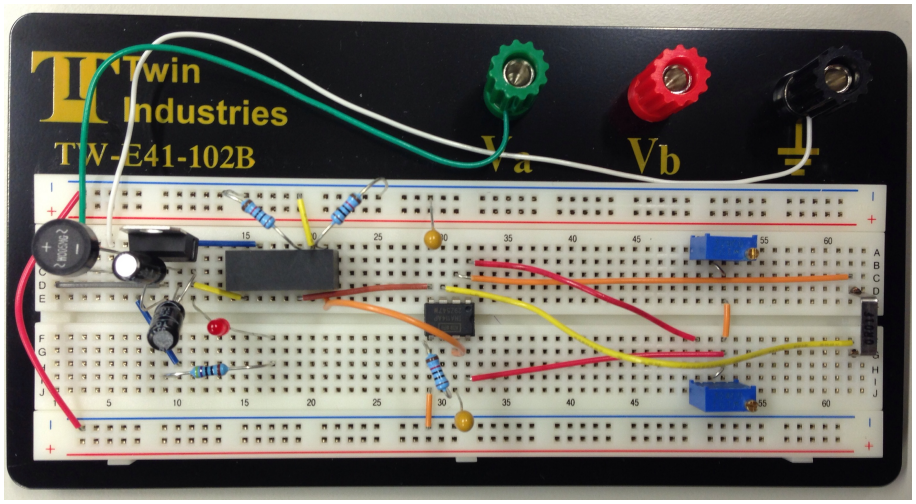


Figure 5.5: Complete circuit

Chapter 6

Evaluation

In this chapter, we first introduce the performance of the LPMI and MPI to confirm that the low power instances are capable of handling low and perhaps medium loads. Then we present the complete ETMI system. Finally, we evaluate the power consumption of the instances using our measuring circuit.

6.1 System Performance

To evaluate our instances, we use a scenario in which the servers deliver static web pages to clients. The webpages used in our tests (http://www.netsys2013.de/) were the same ones used in a real website for a Network Systems conference in March 2013. The website consists of 40 webpages, containing small images of average size 11 kB, style sheets, etc. The webpages are stored on a fileserver and are accessed through a network file system (NFS) over the internal network.

Clients retrieve content from the web server by generating requests according to a Poisson distribution $P_\lambda(k) = \frac{\lambda^k}{k!}e^{-\lambda}$ with λ being the requests per second on average. The requested pages are selected randomly from a set of webpages, for each request the complete webpage is downloaded, hence one request to a web page may generate more than one HTTP request. The performance of the instances is described below.

6.1.1 LPMI Performance

To analyze the performance of the LPMI, in our case the Raspberry Pi, we first applied a load of $\lambda_{min} = 1\ request/second$ increasing it by 1 request every 50 seconds, up to a maximum load of $\lambda_{max} = 30\ requests/s$. We then measured the time taken by the server to deliver the complete web page, the response time for each request. We also measured the average rate of successfully delivered web pages (throughput). The result can be seen in Figure 6.1, where the average of 10 runs is plotted. We noticed that after 1,000 seconds the response time increases above the upper threshold that was considered to deliver web pages of 150 ms, indicating that the LPMI was overloaded. At this time the number of requests that the LPMI was processing was about 20 requests/s.

In a further analysis, when the test reached 1,350 seconds the number of requests remained at a rate of around 26 request/s meaning that the network connection of the LPMI in this case was limiting the load. From the experiments, we can determine that for the LPMI, its processor was the main bottleneck of this setting, with a throughput of 20 requests/s and a response time of 150 ms.

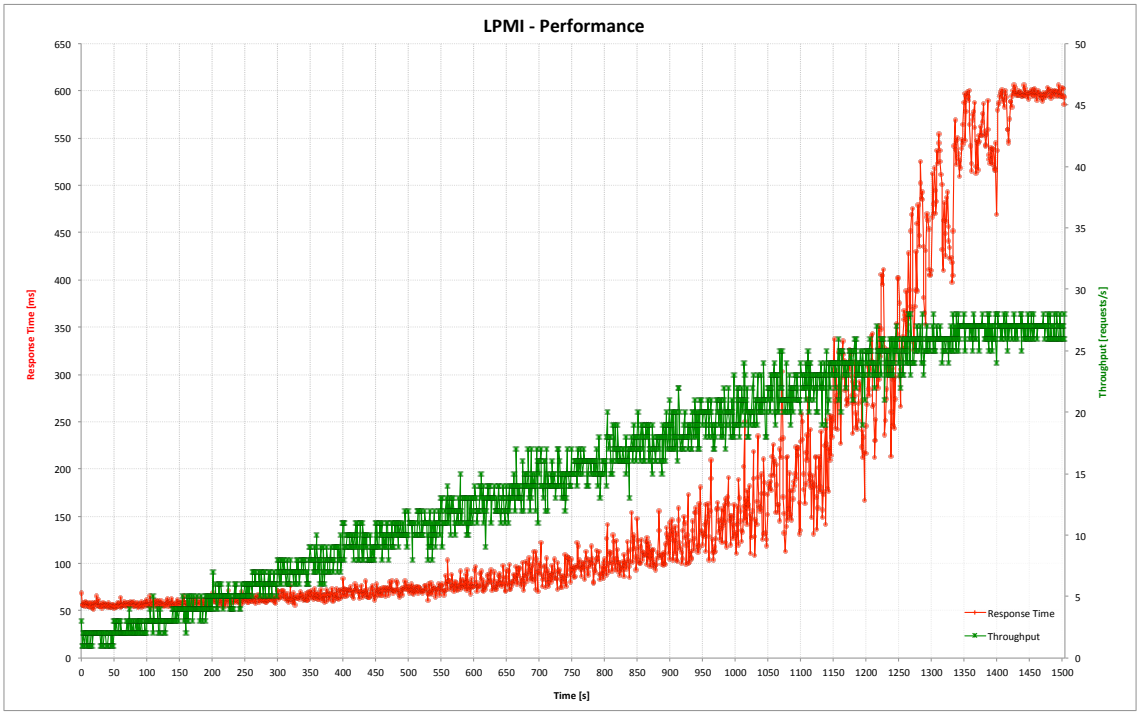


Figure 6.1: LPMI performance

6.1.2 MPI Performance

To evaluate the MPI's performance, in our case the BeagleBone Black board, since its processor is faster than the Raspberry Pi (1GHz versus 700MHz) and it has Ethernet built into the chip instead of via USB, we increased λ_{max} to 50 requests/s instead of 30 requests/s. As a result, the request rate in this experiment goes from $\lambda_{min} = 1\ request/s$ up to $\lambda_{max} = 50\ request/s$. We also averaged the results over 10 runs. Figure 6.2 illustrates the performance of the MPI. The response time in this case only increases above 150 ms after 1,900 seconds, where at this time the throughput is around 35 requests/s, 1.75 times higher than the throughput of the LPMI. At around 2,350 s we notice that the network connection starts limiting the load with a throughput around 45 requests/s. Both processors of the SoCs in our experiments were the bottleneck, limiting the throughput below 40 requests/s for the maximum constraint response time of 150 ms.

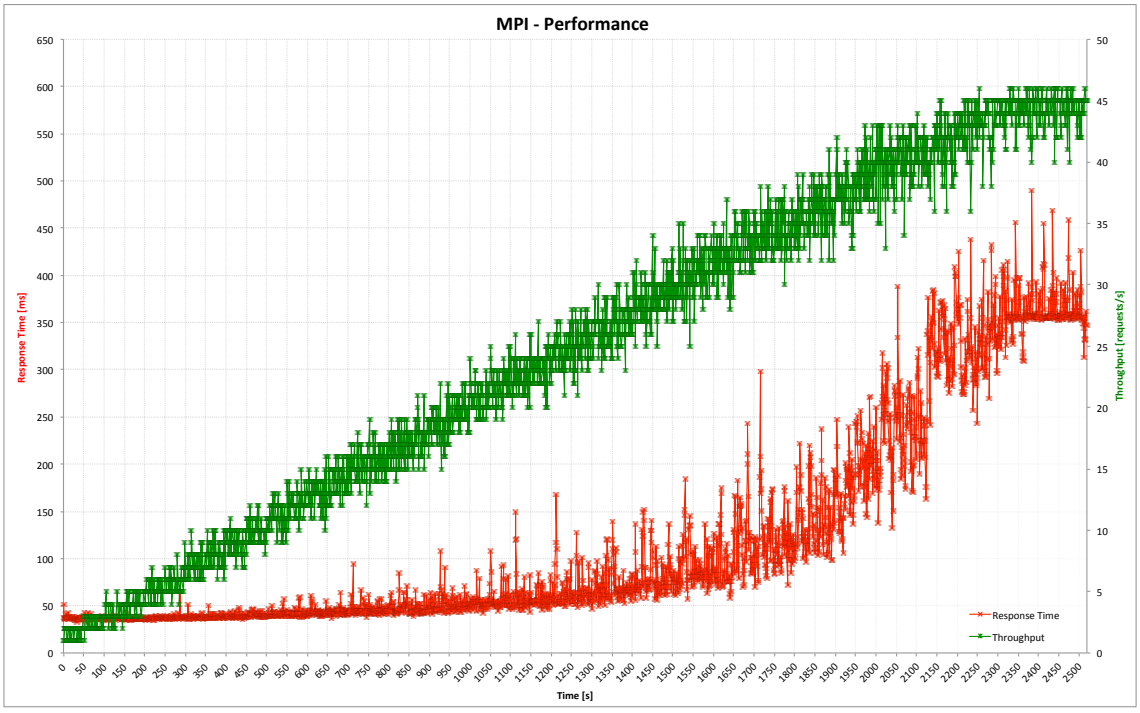


Figure 6.2: MPI performance

6.1.3 ETMI Performance

Both SoCs' performance were satisfactory, considering their size and processing power. They can be used in a realistic small data center with moderate demand and workload. In order to overcome their limitations for higher loads, we introduced our ETMI solution, which scales up to more powerful instances to take care of those restrictions. This section shows the performance of our implemented ETMI solution.

We used the test bed topology, shown in Figure 5.1, to evaluate the ETMI performance. In order to analyze our predictive algorithm, we performed two separate tests. Since it requires past records in order to forecast future values, we first provided a non-stationary time series with a seasonal pattern to perform some tests and analyze the data. We then applied a stationary time series, where no seasonality or trend was observed.

To generate the seasonal data, we used a Poisson process with a $\lambda_{min} = 5\ requests/s$ and we increased the rate every 60 seconds by 5 requests/s up to a $\lambda_{max} = 55\ requests/s$. Once it reached 55 requests/s we started decreasing the rate by 5 requests/s until the load was back at 5 requests/s. By running this process a few times we eventually created a seasonal series as shown in Figure 6.3.

Using the generated time series in the *predictive algorithm* we forecasted the load of the next 21 values. These values were then compared against an offline benchmark, where we set the overload threshold of the LPMI to $T_{overload/LPMI} = 80\ kB/s$ and the overload of the MPI as $T_{overload/MPI} = 138\ kB/s$. After analyzing the data against the thresholds, the sequence of the servers to be used for each of the 21 predicted

values was defined. This sequence was then sent to the *prediction controller*. The *prediction controller* receives the forecasted values and is responsible for switching the traffic to the correct instance by applying the *handover algorithm*.

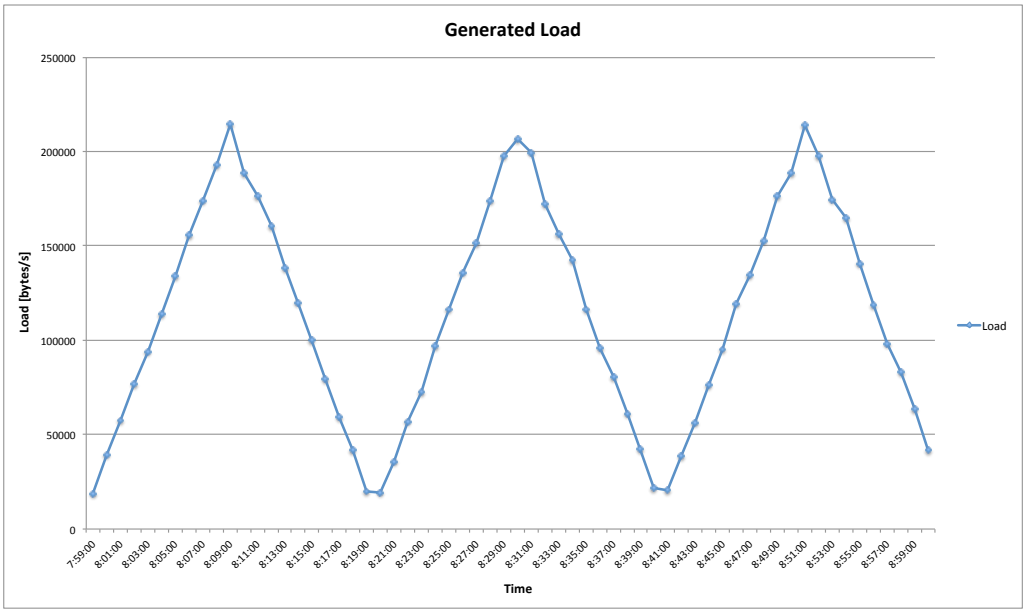


Figure 6.3: Seasonal time series

Using the Poisson process of $\lambda_{min} = 5\ requests/s$ and $\lambda_{max} = 55\ requests/s$, we increased the load by 5 requests/s every 60 seconds and then decreased it back to 5 requests/s. The result of the load and the prediction is illustrated in Figure 6.4.

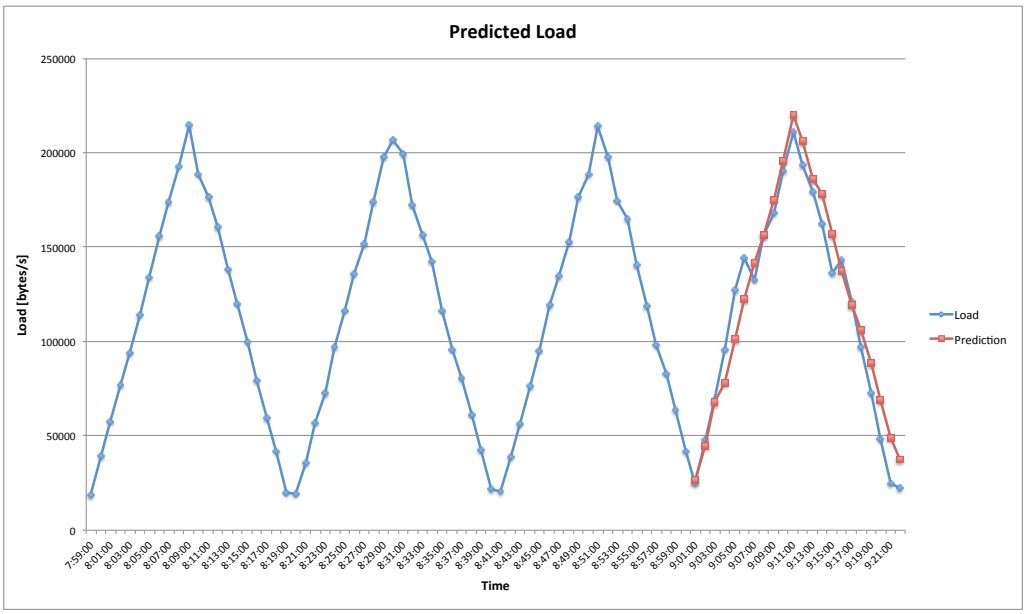


Figure 6.4: Predicted load

To analyze the performance of the ETMI we measured the throughput and the response time during the load. Figure 6.4 shows the ETMI's performance over a time

$t = 1260$ *seconds*. Analyzing the graph we notice that when the throughput reaches around 20 requests/s the controller switches the traffic to the MPI and the response time drops below our limit of 150 ms. When $t = 360$ s, where the request rate is around 30 requests/s, the controller switches the traffic again but now to the HPI. The same procedure can be observed after $t = 900$ s, when the load is decreased and the controller transfers the traffic back to the MPI, and later at around $t = 1140$ s the LPMI receives the traffic, all based on the prediction calculated by the *predictive algorithm*.

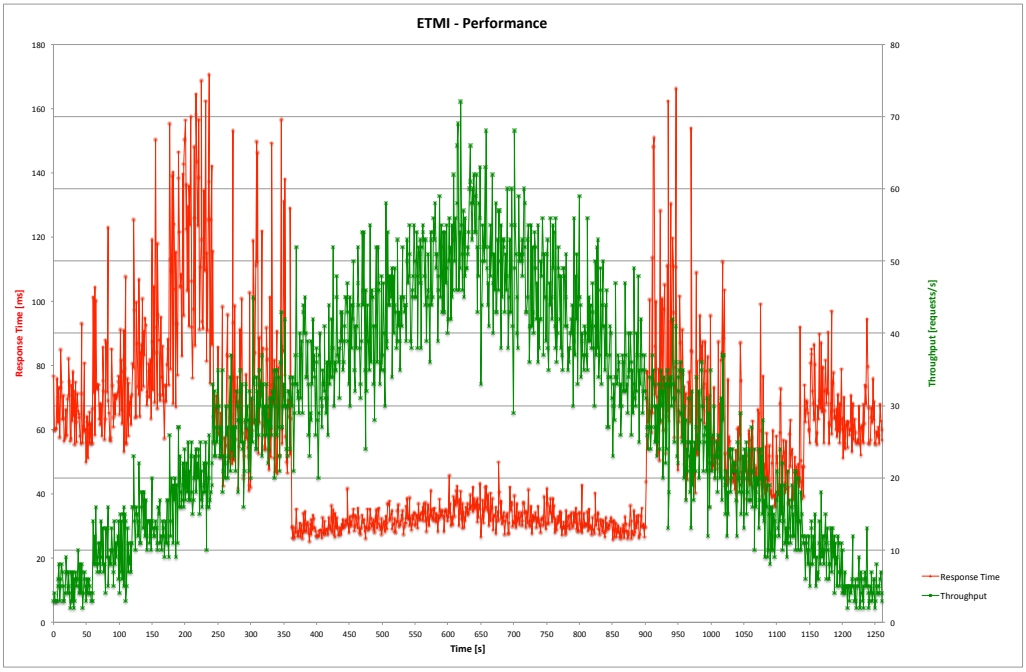


Figure 6.5: ETMI performance

The experiments demonstrate the ability of the ETMI to scale up and down according to predicted values calculated from past seasonal data of the workload an the data center. The response time remains below the 150ms limit and the smooth transition between instances makes it transparent to the client, who sees the web server and the ETMI as a single machine accessed through one public IP address.

An issue when using an ARIMA model is the processing power and time required to predict. Depending on the number of past values and the frequency of the data, the calculation of the prediction may take some time. We measured the time spent to calculate the prediction with different values and as can be seen in Table 6.1 the time increases quickly. To prevent a delay in the prediction when the past traffic data is too long, only a certain amount of the data can be used or the prediction should be done less often, perhaps on a daily or weekly basis. Also, in our experiments the same machine was used for the controller and the predictive algorithm, whereas perhaps a dedicated server would provide a faster response.

Number of values	Time (s)
21	0.729
42	0.755
63	1.169
84	2.913

Table 6.1: Time to predict

When predicting a stationary time series with a non-seasonal data, the results were less satisfactory as seen in Figure 6.6. Therefore, a different model has to be designed in order to provide a more accurate prediction, which is not described in this paper and will be part of a future work.

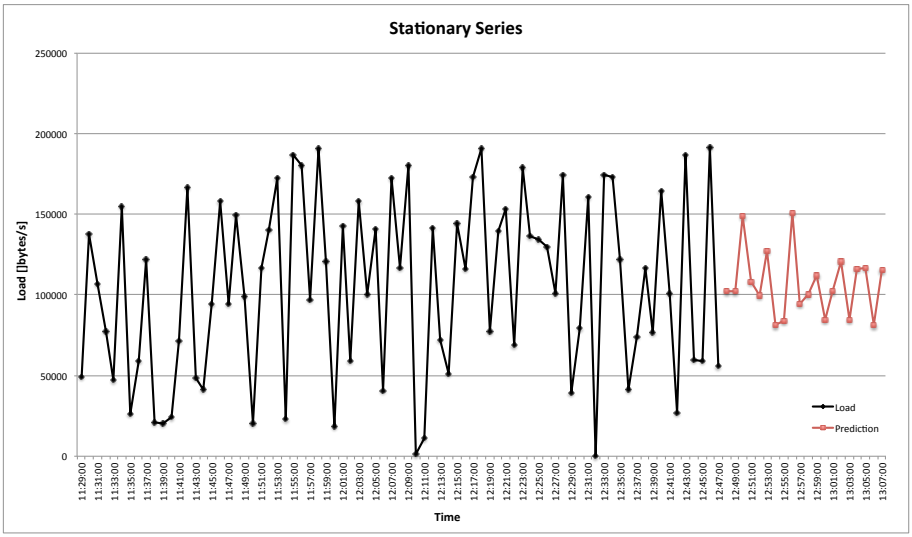


Figure 6.6: Stationary time series

6.2 Energy Efficiency

Besides offering an Elastic Tandem Machine Instance (ETMI) which is always available and is able to transparently scale up and down according to predicted data of the workload, we proposed to efficiently use the energy to approximate our system to an energy-proportional machine, as discussed in the previous chapters. Therefore, we use SoC hardware to provide the service to customers in low and medium load situations. We have shown, in the earlier experiments, that the performance of these SoCs is acceptable and reasonable to be used for low and moderate load in a realistic scenario. We now evaluate their power consumption in different loads to prove their capacity of performing those tasks consuming low energy.

To measure their power consumption, we use the developed measuring circuit introduced in Chapter 5. In the next sections we show the power consumption of each instance of our ETMI solution, then we analyze the results to check the efficiency of our ETMI solution.

6.2.1 LPMI Power Consumption

The LPMI in our solution is the Raspberry Pi SoC. To measure the current flowing through the Raspberry Pi, we connected it in series with our measuring resistor $R_{measure}$ (from Figure 5.3b in Chapter 5) and using the ME-Powerlab3 [27] tool we could export the measured voltage on $R_{measure}$, hence calculating the current flowing through the circuit using Ohm's law. We start our measurements in an idle state where no load is being generated, we then start loading the instance with a rate of 1 request/s, increasing the load by 1 request every 20 seconds. During each request rate we export the measurement to calculate the power consumption experienced during each load of the system. We apply a maximum load of 50 requests/s.

Figure 6.7 shows the power consumption of the Raspberry Pi. In an idle state the power consumption of the Raspberry Pi is $P_{idle/LPMI} = 1.82\,W$; the power consumption increases as the load increases, reaching a maximum power consumption of $P_{max/LPMI} = 2.04\,W$ when the load reaches around 26 requests/s.

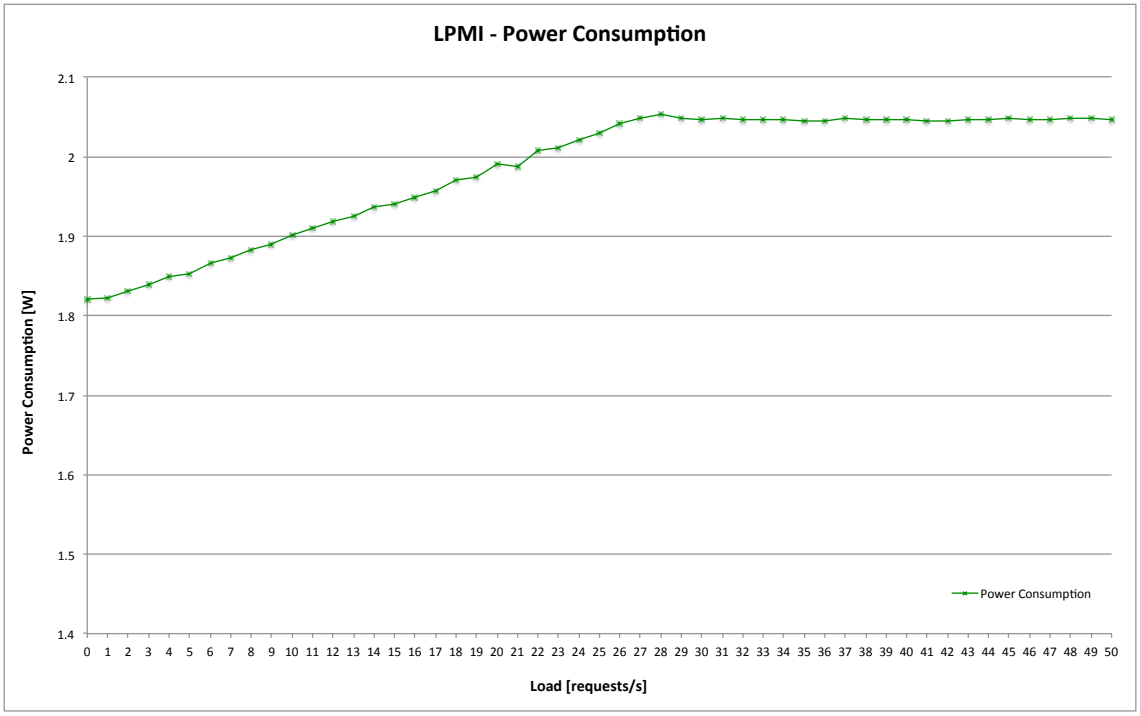


Figure 6.7: LPMI power consumption

6.2.2 MPI Power Consumption

The BeagleBone Black board is used as the MPI in our experiments. Using the same technique presented for the Raspberry Ri, we measured the power consumption of the BeagleBone Black under the same load, from idle to 50 requests/s. In our previous tests we showed that the performance of the Beaglebone Black was almost 1.5 times better than the Raspberry Pi. While the Raspberry Pi was overloaded at a request rate of 20 requests/s, the Beaglebone Black reached overload at around 35 requests/s. Figure 6.8 shows the power consumption of the MPI, from the graph we notice that the energy efficiency is better than the Raspberry Pi. While idle the BeagleBone board's power consumption is $P_{idle/MPI} = 1.52\ W$, and the maximum power consumption was $P_{max/MPI} = 1.77\ W$, below $P_{idle/LPMI}$.

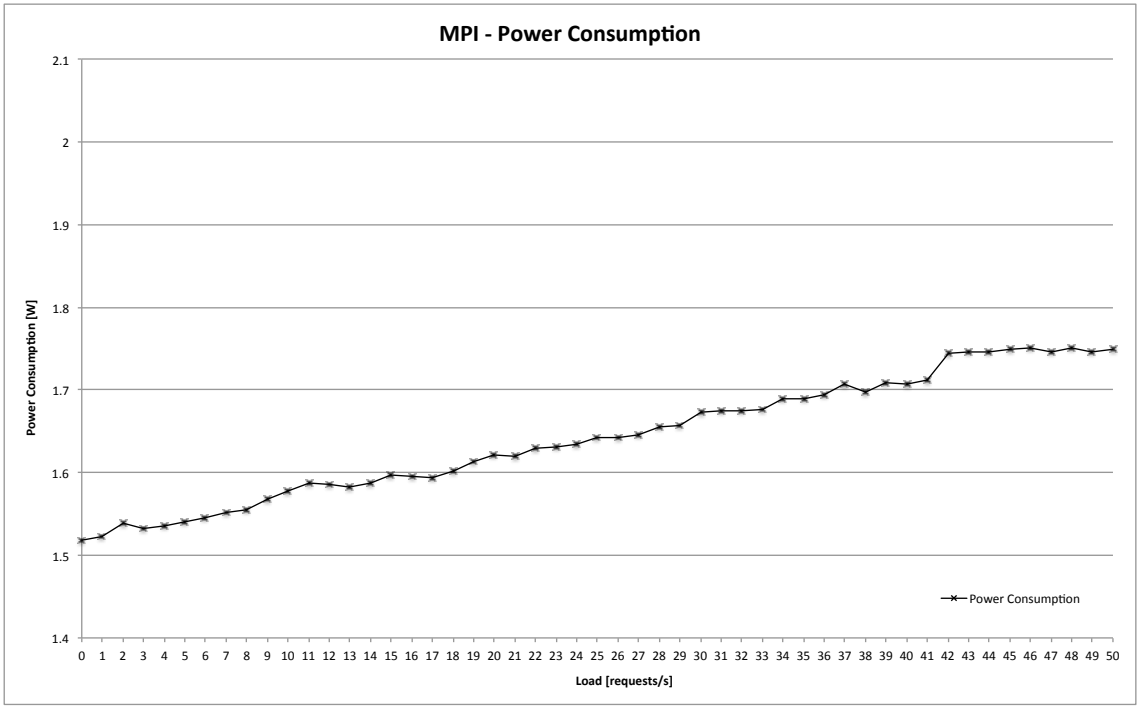


Figure 6.8: MPI power consumption

6.2.3 HPI Power Consumption

The HPI is a VM running on a physical host. To calculate its power consumption we used a multimeter to measure the current flowing in the machine. Figure 6.9 displays the power consumption of the host from an idle state up to a load of 400 requests/s. While idle the host consumes $P_{idle/host} = 141.22\ W$, and the consumption increases to a maximum of $P_{max/host} = 184.46\ W$ when the load reaches 300 requests/s.

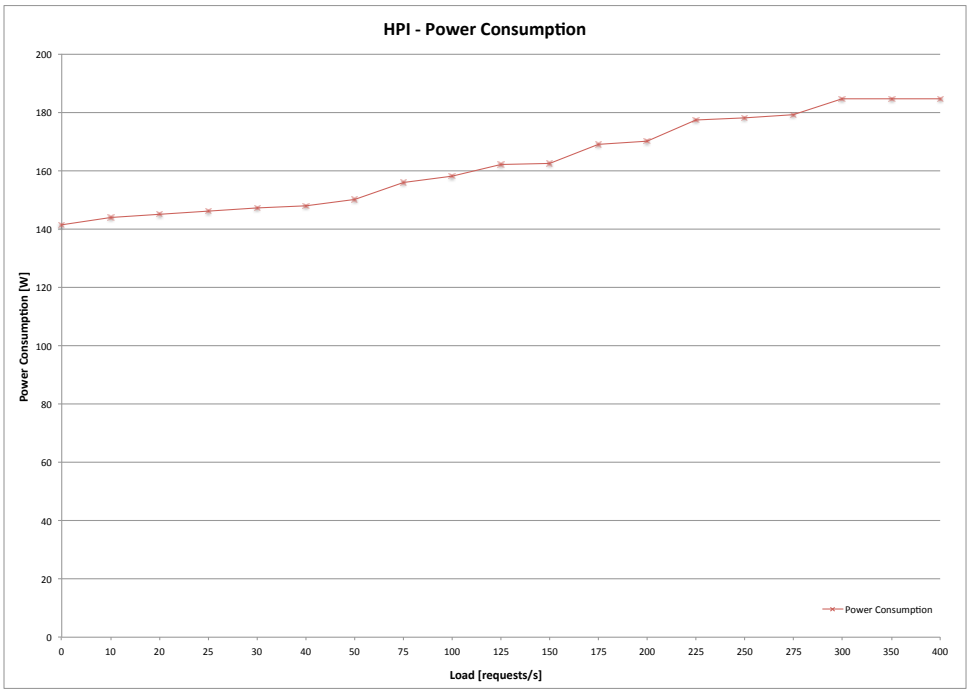


Figure 6.9: HPI power consumption

When we compare the energy efficiency of the instances: $\frac{P_{idle/host}}{P_{idle/LPMI}} \approx 77$, which means that the host would need 77 VMs to accomplish the same energy efficiency of 77 LPMIs in idle mode, and for the MPI, $\frac{P_{idle/host}}{P_{idle/MPI}} \approx 93$ VMs. Considering that our physical host can support a limit of around 300 requests/s, thus $300\ requests/s/77 \approx 4\ requests/s$ on each VM, loading 4 requests/s on each LPMI, their power consumption, taken the load of static web pages shown in the previous scenario, goes to $P_{4req/LMPI} = 77 \times 1.85\ W = 142.45\ W$ and calculating for the MPI the request rate would be 3 requests/s, therefore $P_{3req/MPI} = 93 \times 1.53\ W = 142.29\ W$. The host with a load of 300 requests/s consumes $184,46W$, consequently it consumes 29% more power than 77 LPMIs and almost 30% more than 93 MPIs.

If we consider the upper limit request rate (20 requests/s) of the LPMI, the improvement in energy efficiency increases significantly. For 76 LPMIs at 20 requests/s, it would require at least 5 physical hosts to handle the same load (1540 requests/s). At the load of 20 requests/s the Raspberry Pi consumes $1.99\ W$, thus $77 \times 1.99\ W = 153.23\ W$ for 77 LPMIs. When we compare with 5 physical hosts serving the same load ($5 \times 184.46\ W = 922.3\ W$), the power consumption is more

than 600 % higher than 77 LPMIs. Employing the same comparison with the MPI, we have a maximum request rate of 35 requests/s, therefore 11 hosts would be necessary to execute the same load of 93 MPIs (3255 requests/s), thus leading to a power consumption of $11 \times 184.46\ W = 2029.06\ W$, which is 1291 % higher than 93 MPIs ($93 \times 1.69\ W = 157.17\ W$).

6.2.4 ETMI Power Consumption

In Chapter 2, we mentioned the concept of an energy-proportional machine, consuming power according to the workload performed [1]. Our ETMI solution attempts to provide a more energy-efficient system to approximate to an ideal system. We compare the power consumption of an ideal system, shown in Figure 3.1 in Chapter 3, against our ETMI solution, depicted in Figure 6.10. Despite the abrupt spike when the load reaches around 40 requests/s, due to the fact that in our tests we only used three instances to create our ETMI, we notice that when the load is low the power consumption of the system is also very low. As the load increases beyond the limits of our LPMI and MPI, the HPI boots and the power consumption suddenly increases from 3.34 W (LPMI + MPI) to above 150 W, where all instances are on. The sudden spike in the power consumption of the system could be smoothed by the addition of more MPIs.

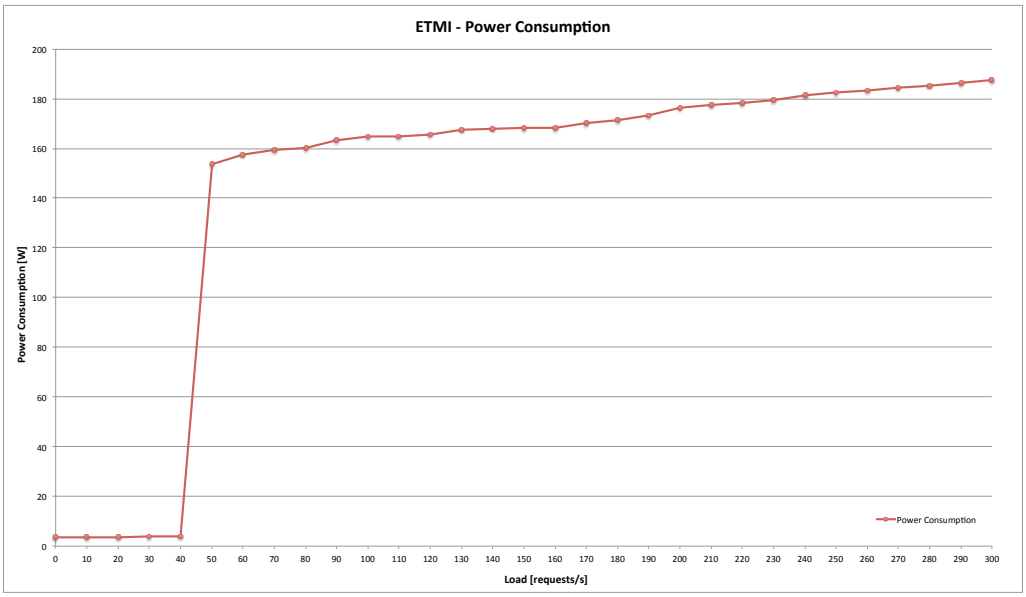


Figure 6.10: ETMI power consumption

Chapter 7

Summary and Future Work

7.1 Summary

This thesis presented an enhancement to the concept of Elastic Tandem Machine Instances (ETMI) proposed by Dürr [2], by integrating different performance instances, such as small, medium and large, to improve the scalability and efficiency of the system. The previous model made use of only two instances to compose the ETMI. By adding more instances to the system, instead of switching directly from a low to a high power device, which could waste resources since the performance of the instances varied significantly, intermediate instances were added to smooth out the transitions between devices and improve the efficiency by having more adequate instances to serve different loads.

To provide a transparent integration among the devices and avoid unavailability during transitions from one instance to another, a handover protocol was implemented, which utilizes software-defined networking technologies to seamlessly switch among instances without interrupting any existing connections. A predictive algorithm, based on an ARIMA model, was designed to decide in advance without overloading the instances the best time to switch. Besides preventing resource waste it also focusses on energy efficiency by only utilizing the server when required and saving energy once in idle mode. The predictive algorithm also takes advantage of the SDN technology. By the support of an SDN controller and statistics acquired from the open vSwitch, traffic information could be retrieved and used to provide a forecast of future activities.

Furthermore, we demonstrated the benefits of using low power SoC hardware to improve the energy consumption of the data center and also serve as a realistic three-tier system such as a web service. The SoCs demonstrated themselves to be efficient when serving moderate load, whilst consuming very low power. The design and implementation of an ETMI and the use of SoC hardware, for small and perhaps medium loads, proved convenient to improve the energy efficiency of data centers. With a proof of concept of the ETMI and the power consumption of the instances, the experiments showed the possibility of utilizing the system in a real scenario.

7.2 Future Work

Even though the results obtained in this thesis were satisfactory, as in most system design and implementation, there are always possibilities for improvements and enhancements.

One way to improve the system concerns the predictive algorithm. Currently, it is able to predict non-stationary time series, where a trend or a seasonal pattern is noticed. Further work could also support stationary time series, in this case the use of a second prediction model could be applied. By supporting different prediction models, a more efficient and accurate prediction can be generated, hence a larger variety of data centers can be supported.

The presented design scales up and down according to the prediction of the load on the data center, once the traffic increases the controller transfer the traffic to a more powerful instance, leaving the previous instance idle. If the traffic increases again, another instance is powered on and the traffic is redirected again, and the previous server is either kept on and in idle mode, in case of an SoC, or is shut down, in case of a VM. Instead of leaving the instance idle and transferring the traffic to a new high power instance, perhaps two low power instances could balance the load between each other and manage the traffic without the need for powering up a high power instance.

Moreover, the test bed used two SoCs and a commodity server to form the ETMI solution. Even though the results regarding scalability and energy efficiency were effective, future experiments which add more devices to smooth the energy efficiency curve and add more options to the controller to switch the traffic would be more convenient. Furthermore, more extensive tests with real data from a web server, for example, could be performed in order to provide more accurate results from a real data center.

Bibliography

[1] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. December 2007. [pages i, vii, 1, 6, 7, 20, 21, 22, 23, and 58]

[2] Frank Dürr. Improving the efficiency of cloud infrastructures with elastic tandem machines. June 2013. [pages i, 1, 2, 20, 25, 29, 32, 41, and 59]

[3] Mark Blackburn. Five ways to reduce data center server power consumption. 2008. [pages vii and 5]

[4] Emerson Network Power. nergy logic: Reducing data center energy consumption by creating savings that cascade across systems. 2009. [pages vii, 5, 6, and 23]

[5] Open Networking Foundation. Software defined networking: The new norm for networks. April 2012. [pages vii and 9]

[6] Floodlight OpenFlow Controller. `http://www.projectfloodlight.org/floodlight`, October 2013. [pages vii, 13, and 42]

[7] Raspberry pi. `http://www.maxoverpro.org/archives/752`, November 2013. [pages vii and 26]

[8] Gerald Coley. Beaglebone black system beaglebone black system reference manual. Revision A5.2, April 2013. [pages vii, 27, and 46]

[9] Dan Talayco David Erickson Glen Gibb Guido Ap-penzeller Jean Tourrilhes Justin Pettit KK Yap Martin Casado Masayoshi Kobayashi Nick McKeown Peter Balland Reid Price Rob Sherwood Yiannis Yiakoumis. Ben Pfaff, Brandon Heller. Openflow switch specification version 1.0.0. December 2009. [pages ix and 11]

[10] Rob J Hyndman and George Athanasopoulos. Forecasting: principles and practice. `https://www.otexts.org/fpp`, November 2013. [pages ix, 16, 17, 18, and 35]

[11] Intel. Power management in intel architecture servers. 2009. [page 5]

[12] Open vSwitch - An Open Virtual Switch. `http://openvswitch.org`, October 2013. [page 15]

[13] Jonathan D. Cryer and Kung-Sik Chan. *Time series Analysis With Applications in R*. Springer, 2 edition, 2008. [page 15]

[14] Kenneth P. Burnham and David R. Anderson. *Model Selection and Multimodel Inference*. Springer, second edition, 2002. [page 16]

[15] S. K. Nandy K. Gopinath Mohit Dhingra, J. Lakshmi and Chiranjib Bhattacharyya. Elastic resources framework in iaas, preserving performance slas. 2013. [page 19]

[16] Xiaohui Gu Zhenhuan Gong and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. 2010. [page 19]

[17] Giuseppe Iannaccone Randy Katz Gunho Lee Byung-Gon Chun, Gianluca Iannaccone and Luca Niccolini. An energy case for hybrid datacenters. October 2009. [page 19]

[18] Dana Butnariu Richard Wang and Jennifer Rexford. Openflow-based server load balancing gone wild. March 2011. [page 19]

[19] Amazon ec2 instance types. `https://aws.amazon.com/ec2/instance-types/`, November 2013. [page 22]

[20] Raspberry pi technical documents. http://`http://www.raspberrypi.org/technical-help-and-resource-documents`, November 2013. [pages 26 and 46]

[21] Rob J. Hyndman and Yeasmin Khandakar. Automatic time series forecasting: The forecast package for r. *Journal of Statistical Software*, 27(3), July 2008. [pages 35 and 36]

[22] Canova F and Hansen BE. Are seasonal patterns constant over time? a test for seasonal stability. *Journal of Business and Economic Statistics*, (13):237–252, 1995. [page 36]

[23] Rob J Hyndman. forecast: Forecasting functions for time series and linear models. `http://cran.r-project.org/web/packages/forecast/index.html`, November 2013. [page 43]

[24] Jri package. `http://www.rforge.net/org/docs/`, November 2013. [page 43]

[25] Bourns. *PWR4412-2S Series Bare Metal Element Resistor*, number REV. 03/09, January 2003. [page 46]

[26] Burr-Brown. Precision instrumentation amplifier. Technical report, Texas Instruments, April 2013. [page 47]

[27] Me-powerlab3. `http://www.meilhaus.de`, November 2013. [pages 48 and 55]

Author's Statement

Hereby I certify that I have realized this work on my own and that all the sources that I have used or consulted are duly noted herein.

Furthermore, I certify that I know and accept that I have no right to exploit the results of my Master Thesis by any means without the written permission of the Institute of Parallel and Distributed Systems(IPVS).

Stuttgart, December 13th, 2013

Arturo Francato