Self Tuning for Elastic Storage in Cloud Environment

MOHAMMAD AMIR MOULAVI

KTH Information and Communication Technology

Master of Science Thesis
Stockholm, Sweden 2011

TRITA-ICT-EX-2011:247

KTH Computer Science
and Communication

Self Tuning for Elastic Storage in Cloud Environment

Master of Science Thesis at
Swedish Institute for Computer Science (SICS) and
Kungliga Tekniska Högskola (KTH)
Stockholm 2011

M. AMIR MOULAVI

Supervisor:
Ahmad Al-Shishtawy - PhD Candidate
Per Brand - PhD
Examiner:
Vladimir Vlassov
Associate Professor - PhD

Acknowledgements

This thesis would not have been possible without the help and support of many people around me, only a proportion of which I have space to acknowledge here. I would like to start by expressing my gratitude to my examiner, Prof. Vladimir Vlassov, for his continuous support, ideas, recommendations and patience through out this master thesis project. I am grateful to my supervisor Ahmad Al-Shishtawy for sharing his knowledge and experience with me during all phases of this project. His insightful advices and contributions kept me focused on my goals. I am grateful to my supervisor Per Brand for his final comments on the report. I would also like to acknowledge Leif Lindbäck for his great comments and corrections. Thanks to my friends for keeping me sane during the course of this project. Finally, I owe my deepest gratitude to my parents for helping me to be where I am now.

Abstract

Elasticity, where a system requests and releases resources in response to a dynamic property, has been an important issue in Cloud computing. It can be handled manually or automatically. Efforts being made to make elasticity as automatic as possible. Autonomic computing has played a significant role in many computing fields including Cloud computing. In this master thesis, we have adopted control theory approach for automation of elasticity in key-value storage that is provided in a cloud environment and operates under dynamic workloads. Automation is achieved by providing a feedback controller that automatically grows and shrinks the number of nodes in order to meet Service Level Agreement (SLAs) under high load and reduces costs under low load. Every step of building a controller for elastic storage, including System Identification and controller design, is discussed in this thesis. We have evaluated our approach by simulation. We have implemented a simulation framework based on Kompics[1], in order to simulate an elastic key-value store in Cloud environment and to be able to experiment with different controllers. Finally, we have examined the implemented controller against specific SLA requirements and we have evaluated the controller behaviors in different scenarios. Our simulation experiments have shown the feasibility of our approach to automate elasticity of storage services.

[1]Kompics is a message-passing component model for building distributed systems by putting together protocols programmed as event-driven components

Referat

Elasticitet, när ett system beställer och släpper taget om resurser som svar på en dynamisk egenskap, har varit ett viktigt problemområde inom Cloud computing. Det kan hanteras manuellt och automatiskt. Ansträngningar har gjorts för att göra elastiskt beteende så automatiserat som möjligt. Autonoma system har spelat en viktig roll i många fält inom datavetenskapen, inklusive Cloud computing. I denna avhandling har vi använt oss av ett reglerteoretiskt angreppssätt för att automatisera elasticitet i ett key-value lagringssystem som erbjuds i molnet och som verkar under varierande belastning. Automatisering åstadkoms genom en återkopplande kontrollmekanism som tillåter antalet noder att automatiskt bli fler eller färre för att möta krav på avtalad servicenivå (SLA) under hög belastning och för att reducera kostnader under låg last. Varje steg för att bygga kontrollmekanismen för den elastiska lagringslösningen, inklusive systemidentifikation och kontrollmekanismdesign, diskuteras i detalj. Vi har utvärderat och verifierat vårt angreppssätt genom simulering. Vi har implementerat ett simuleringsramverk baserat på Kompics[2]. Detta för att simulera en elastisk nyckel/värde-lagring i en molnmiljö och för att kunna experimentera med olika kontrollmekanismer. Slutligen har vi utvärderat den implementerade kontrollfunktionen mot givna SLA-krav och dess beteende under olika lastscenarier. Vår simulering visar att den föreslagna strategin för automatiserad elasticitet av lagring är genomförbar och bör generera det eftersträvade resultatet.

[2]Kompics är en meddelandebaserad modell som används för att bygga distribuerade system genom att sammanfoga protokoll som programmeras som händelsestyrda komponenter

Contents

Acknowledgements **i**

Abstract **ii**

Referat **iii**

List of Figures **1**

List of Tables **2**

Listings **2**

1 Introduction **1**
- 1.1 Background 1
  - 1.1.1 Objectives 1
  - 1.1.2 Thesis Outline 3
- 1.2 Autonomic Computing 5
  - 1.2.1 Properties of Self-managing Systems 5
  - 1.2.2 Autonomic Computing Architecture 6
- 1.3 Design Methodology for Self-Management 6
  - 1.3.1 Steps in Designing Distributed Management 6
  - 1.3.2 Orchestrating Autonomic Managers 8
- 1.4 Cloud Computing 8
  - 1.4.1 Delivery Models 8
  - 1.4.2 What is Elastic Computing? 9

2 Survey and State of the Art **11**
- 2.1 Distributed Storage Systems 11
  - 2.1.1 Google Bigtable 11
  - 2.1.2 Amazon Dynamo 12
  - 2.1.3 Hypertable 13
  - 2.1.4 Yahoo! PNUTS 15
  - 2.1.5 Apache HBase 17
  - 2.1.6 Terrastore 18

2.1.7 Self-* in Distributed Storage Systems Context 20
2.1.8 Summary 21
2.2 Application of Control Theory in Computing Systems 22
2.2.1 System Identification 22
2.2.2 Automated control of multiple virtualized resources 22
2.2.3 MIMO Control of Apache Web Server 23
2.2.4 Controller Analysis and Design 24
2.2.5 Desired Objectives 26
2.2.6 Summary 27

3 Control Analysis and Design 29
3.1 Problem Definition and System Description 29
3.2 System Identification 30
3.2.1 Basic Steps of System Identification 31
3.2.2 State Space Model 31
3.2.3 State Variables 32
3.2.4 State Space Model 33
3.2.5 Parameter Estimation 35
3.2.6 Solving State Dynamics 35
3.2.7 Model Evaluation 36
3.3 Control Analysis and Design 36
3.3.1 Dynamic State Feedback 37
3.3.2 LQR Optimal Control Design 38
3.3.3 Controllability 38
3.3.4 Stability 39
3.3.5 Fuzzy Controller 39
3.3.6 Method of designing controller for elastic storage in cloud . . 40
3.4 Summary 42

4 Simulation Framework Implementation 43
4.1 Introduction 43
4.2 Implementation 43
4.2.1 Cloud Instance Component 43
4.2.2 Cloud Provider Component 50
4.2.3 Elastic Controller 56
4.3 Summary 61

5 Experiment 63
5.1 System Identification 63
5.1.1 Preprocessing of collected data 66
5.1.2 Model Evaluation 66
5.1.3 Estimating Controller gains 67
5.2 Controller Design 68
5.2.1 Experimenting with Controller 68

5.3 Summary 74

6 Conclusion and Future Works **75**

6.1 Summary and Conclusion for the thesis 75

6.1.1 Study Distributed Data Storage (DSS) Systems 75

6.1.2 Study the application of control theory in Computing Systems 75

6.1.3 System Identification and appropriate controller design . . . 76

6.1.4 Method of designing controller for elastic storage in cloud . . 76

6.1.5 Implementation of a Simulation Framework for Cloud 76

6.1.6 Experiment with the controller with simulation framework . . 77

6.2 Future work 77

Appendices **78**

A Simulation Framework Usages Described **79**

Bibliography **87**

List of Figures

1.1 IBM Autonomic Computing Reference Architecture (Adopted from Fig. 2.1 in [11] 7

2.1 Amazon's Service Oriented Architecture 14
2.2 Hypertable System Architecture (Figure is taken from Hypertable wiki page) 15
2.3 PNUTS System Architecture (Figure is taken from [21]) 16
2.4 HBase System Architecture 18
2.5 Terrastore System Architecture 20
2.6 AutoControl - Logical Controller Architecture 23
2.7 Apache Web Server - Logical Controller Architecture 24

3.1 Storage Cloud Architecture 30
3.2 Scalar Block Diagram 32
3.3 State Space Model 33
3.4 Controllers Architecture 39

4.1 Overall System Architecture 44
4.2 Cloud Instance Component Architecture 45
4.3 OS Component Architecture 45
4.4 CPU Component Architecture 47
4.5 Memory Component Architecture 48
4.6 Disk Component Architecture 49
4.7 Cloud Provider Component Architecture 50
4.8 Cloud API Component Architecture 51
4.9 Health Checker Component Architecture 52
4.10 Elastic Load Balancer Component Architecture 53
4.11 Request Generator Component Architecture 55
4.12 Elastic Controller Component Architecture 57
4.13 Controller Component Architecture 57
4.14 Sensor Component Architecture 58
4.15 Actuator Component Architecture 60
4.16 Modeler Component Architecture 60

5.1 Number of Instances & Total Cost ($) 64
5.2 Average CPU Load & Average Bandwidth per download 64

5.3 Average Response Time & Average Load 65
5.4 Model Evaluation - one step prediction for Average CPU Load 67
5.5 Experiment1 Workload 69
5.6 SLA Experiment - Average CPU Load 71
5.7 SLA Experiment - Average Response Time 71
5.8 SLA Experiment - Interval Total Cost 72
5.9 SLA Experiment - Average Bandwidth per download (B/s) 73
5.10 SLA Experiment - Number of Nodes 73
5.11 Experiment2 workload 74

A.1 Simulation Framework - Cloud Provider Window 82
A.2 Simulation Framework - Instance Window 83
A.3 Controller Window - System Identification 85

List of Tables

2.1 Summary of Distributed Storage Systems 21
2.2 Summary of Computing Systems that exploit Control Theory 27

3.1 Summary of State-space feedback control architectures [26] 36

5.1 SLA Violations 70
5.2 Total Cost for each SLA experiment 72
5.3 Total Cost for Cost experiment 74

Listings

A.1 A sample scenario to create cloud environment 80
A.2 A sample Elastic IP addresses definition 81

Chapter 1

Introduction

1.1 Background

Web-based services frequently experience high work loads during their life time. A service can get popular in just within an hour and the occurrence of such high workloads has been observed more and more recently. Cloud computing (section 1.4) has brought a great solution to this problem: requesting and releasing instances that provide the service on-the-fly. This has helped to distribute the loads among more instances. However the high load state typically does not last for long and keeping resources in a cloud costs money. This has led to Elastic computing (section 1.4.2) where a system can scale up and down based on a dynamic property that is changing from time to time.

Elastic computing requires automatic management that can be used from the results obtained in Autonomic Computing area. Systems that exploit autonomic theory to enable automatic management of themselves are called Self-managing systems (section 1.2.2) In this way complex system such as a cloud system can be automatized without the need of human supervision. One common and proven way to apply automation to computing systems is control theory (section 2.2).

1.1.1 Objectives

This master thesis project aims at studying the cloud systems from data storage point of view and automating the elasticity (scaling up/down) property of cloud systems. In this section we go through a list of objectives that are solved by this master thesis project:

Study the Distributed Data Storage (DSS) Systems

Distributed Data Storage (DSS) systems construct a major portion of current storage systems. They gain more popularity every day because of their unique properties that target scalability problems in enterprise systems. This project studies the properties of such systems to obtain a better understanding of automation and how each property can affect it. It should be noted also that the target of this thesis is storage cloud that are inherently DSS systems.

Study the application of control theory in Computing Systems

Control theory has been exploited by electrical and mechanical engineers for automation of physical systems for a long time. However the idea of applying such theory to computing system is still new. This project studies the related computing systems that are availing from control theory and select the best practices and approaches for system identification and controller design.

System Identification and appropriate controller design

Based on the study of controller-enabled computing systems and control theory itself, this project identifies the system which is a key-value store and builds a model that can be used for capturing interesting properties from the system. The system identification should only capture the relevant system behavior that are required for automation and controller design and ignore the rest. Based on this system identification approach and performed studies on control-enabled computing systems, an appropriate controller type is selected, implemented and tested against the target system. The goals in designing the controller are:

- **Performance Assurance**: controller scales up/down the number of instances in a cloud to meet Service Level Agreements (SLA)
- **Adaptation**: controller is able to adapt to dynamic workloads
- **Automation**: all decision regarding scaling up/down in the number of instances is made automatically

Method of designing controller for elastic storage in cloud

The project proposes a method for designing controller for elastic storages in cloud environments. This method includes all the steps needed to such a design. The method is general enough to cover storage cloud systems.

Implementation of a Simulation Framework for Cloud

In order to simplify the system identification and controller design for the cloud environment, a simulation framework is implemented in great details that enables a better understanding and monitoring of the cloud system. This simulation framework is capable of the following items:

- A complete simulation of cloud resources including:
  - Cloud Provider that administers the cloud resources

  - Storage instances that serve requests
  - Elastic Load Balancer (ELB)
  - Eventual Perfect Failure Detector (EPFD) that is responsible for checking for failures
  - Request generator that is responsible for sending request to instances
  - Cloud API that enables remote launching and shutting down of storage instances

- System identification component which enables the identification and monitoring of different systems with various configurations

- Controller implementation and testing against the cloud

- Custom load distributions

In addition to feature implementation, simulation framework has GUI for better overview of cloud/controller activities.

Experiment the controller with simulation framework

The implemented controller is tested by the simulation framework and is evaluated against expected system behaviors. Different system properties are also examined and checked to see how they can influence the controller's decisions.

1.1.2 Thesis Outline

This master project is going to be in a number of chapters that each one is shortly described in the following:

- **Chapter 1**: the rest of this chapter is an introduction to three concepts that are used in this master thesis project: Autonomic Computing, Self-management systems and Cloud Computing.

- **Chapter 2**: consists of a survey for two related systems that this master project is based on: Distributed Storage Systems (DSS) and control theory enabled computing systems. a number of well designed systems in each category is briefly described. These systems are categorized based on different properties at the end of each section.

- **Chapter 3**: it is started with problem definition and a simple case scenario. In this chapter, the system Identification approach and steps is discussed in detail. Based on the properties of target system that is a cloud environment, we have chosen a controller design approach that is also described in the second half of this chapter in details.

- **Chapter 4**: implementation of the simulation framework is described in this chapter. The description is performed according to original Kompics documentation in which the architecture of each component is described together with ports, types of messages, channel subscriptions and component connections. We encourage the reader to get familiar with Kompics first before trying to understand the details of this implementation.
- **Chapter 5**: this chapter is where theory meets practice. We try to apply the theory that is covered in Chapter 3 and use the implemented framework to experiment with target environment that is a cloud environment.
- **Chapter 6**: conclusion and future work are discussed in this chapter. There are some extensions that can be done based on this master project that are mentioned in this chapter.
- **Appendix A**: includes necessary details to obtain, modify, build and run the simulation framework.

1.2 Autonomic Computing

In 2001, Horn from IBM [27] marked the new era of computing as Autonomic Computing. He pinpointed that the software complexity would be the next challenge in Information Technology. Growing complexity of IT infrastructures can cover the benefits of information technology aims to provide. One traditional approach to manage this complexity is to rely on human intervention. However, considering the expansion rate of softwares, there would not be enough skilled IT staff to tackle the complexity by their managements. Moreover, most of the real-time applications require immediate administrative decision-making and response times. Another drawback of the growing complexity is that it forces us to focus on management issues rather than improving the system itself.

The new approach, however, would be the autonomic computing. In other words, designing and building systems that are capable of managing themselves. These systems can adjust themselves to the changes of the environment. They must anticipate the needs and allow users to concentrate on what they want to accomplish.

1.2.1 Properties of Self-managing Systems

IBM proposed main properties that any self-managing system should have to be considered as autonomic systems [32]. These properties are usually referred to as *self-** properties. The four main properties are:

- **Self-configuration**: Autonomic systems should be able to configure themselves in accordance with defined high-level policies. When a new component is introduced in the system, it will incorporate itself seamlessly and the rest of the system will adopt to its presence. The system should be able to continuously reconfigure itself and adopt to the news changes in the environment.

- **Self-optimization**: Autonomic systems should continually monitor themselves and find new ways to identify and seize opportunities to make themselves efficient in performance and cost. They will proactively seeking to upgrade their functions by finding, applying and verifying the latest changes.

- **Self-healing**: Autonomic systems should detect, diagnose and repair localized problems resulting from bugs and/or failures.

- **Self-protection**: Autonomic systems should defend the system as a whole against large scale problems arising from malicious attacks or cascading failures. They should also anticipate problems perceived by their sensors and take steps to avoid them.

1.2.2 Autonomic Computing Architecture

IBM proposed a reference architecture for autonomic computing [28] consists of the following building blocks:

- **Manageability Endpoints (touchpoints)**: A manageability endpoint is a component in the system that provide the access to states and management operations for resources in the system. The manageability interface for monitoring and controlling a managed resource is organized into its sensor (that is used to perceive data from the resource) and effector (that is used to perform operation on the resources).
- **Autonomic Managers**: An autonomic manager is an implementation that automates the management functionality according to the behavior defined by management interface. In other words, it is an intelligent control loop. This control loop consists of four stages: monitor, analyze, plan and execute. It interacts with the managed resources with the touchpoints.
- **Knowledge Source**: is an implementation of database or repository that provides access to the knowledge defined in the interface (e.g. architectural information, monitoring history, management data) between the autonomic managers..
- **Manual Managers**: is an implementation of a user interface that enables an IT professional to perform some management function manually.
- **Enterprise Service Bus**: is an implementation that provides the integration between other building blocks. It can be used to connect various autonomic computing building blocks.

This architecture is demonstrated in Fig. 1.1

1.3 Design Methodology for Self-Management

A methodology for self management of distributed applications should include methods for management decomposition, distribution an orchestration. In [12] authors have discussed a comprehensive approach for designing such system that we will briefly address in the following.

1.3.1 Steps in Designing Distributed Management

Designing of a self-* application can be divided into three separate parts: functional part, touchpoints and the management part. The functional and management part are designed based on the requirements. For the management part there is an option to choose single or multiple managers. Obviously having multiple managers can remove the problem of single point of failures. Moreover, different context/concept

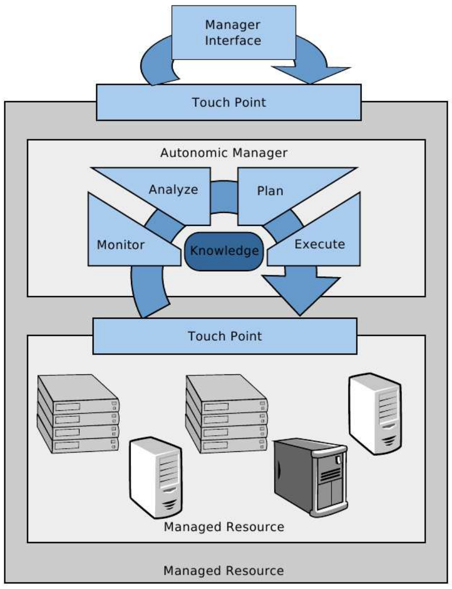


Figure 1.1. IBM Autonomic Computing Reference Architecture (Adopted from Fig. 2.1 in [11]

can be assigned to different managers as well that can lead to a cleaner design at the end. The following iterative steps should be performed when designing the management part:

Decomposition The first step is extracting the task out of management. Decomposition can be done either functional or spacial. The major design issue to be considered in this step is the granularity of tasks with this assumption that a task or a group of them can be performed by a single manager.

Assignment The extracted tasks can then be assigned to managers. Each manager can be responsible for one or more tasks. Assignment can be done based on a specific self-* properties or based on which area of the application task belongs to.

Orchestration Since the managers are not independent of each other the assigned tasks also have dependencies between themselves. These dependencies can cause conflicts and interferences. In order to avoid such cases, we need to have some kind of orchestration/coordination.

Mapping The set of managers are then mapped to the resources. The major design issue at this step is optimized placement of the managers on functional components on nodes to increase performance.

1.3.2 Orchestrating Autonomic Managers

Autonomic managers can coordinate their operations in the following four ways:

Stigmergy is a way of indirect communication and coordination between agents [15]. Agent makes changes and modifications in its environment that these changes can be sensed by other Agents. Hence the agent can be reactive to these changes and do other actions. In the context we are talking, agents are autonomic managers and the environment is the managed application. Generally Stigmergy makes it very difficult and challenging to design a self-managing system.

Hierarchical Management Some managers can control and monitor other managers. Lower level managers are considered as managed resource for the higher levels. Higher level managers can sense and affect lower level ones. Lower level managers are often speedy so they can catch up with the frequent changes of the environment however the higher level managers are often slower and used to orchestrate the system by monitoring global properties and tuning lower level managers.

Direct Interaction Managers may interact with each other directly. Binding can be used to coordinate managers and avoid undesired behaviors.

Shared Management Elements In this way the managers usually share their state (knowledge) to synchronize their actions.

1.4 Cloud Computing

Cloud computing is a model for enabling ubiquitous, scalable, on-demand network access to a shared pool of configurable computer resources that can be rapidly provisioned and released with minimal management effort [43].

1.4.1 Delivery Models

The National Institute of Standards and Technology (NIST) definition of cloud computing categorizes three delivery models [43]:

- **Software as a Service (SaaS)**: provides the capability to the consumers to use the provider's applications running on a cloud infrastructure. The consumer does not manage or control the underlying cloud infrastructure.
- **Platform as a Service (PaaS)**: provides the capability to the consumers to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages and tools supported by the provider. The consumer does not manage the underlying cloud infrastructure.

- **Infrastructure as a Service(IaaS)**: provides the capability to the consumers to provision processing, storage, networks and other fundamental computing resources where the consumer is able to deploy and run arbitrary software.

1.4.2 What is Elastic Computing?

First we need to understand what elasticity is in physics. According to Wikipedia:

> *"In physics, elasticity is the physical property of a material that returns to its original shape after the stress (e.g. external forces) that made it deform is removed"*

Similarly this can be applied to computing. It can be thought as the amount of strain an application or infrastructure can bear while either expanding or collapsing to meet specific requirements. In the area of storage systems, this can be bringing in new nodes of storage and releasing the unused ones when they are not needed anymore.

Thus in simple words, elasticity is defined as the ability to scale resources both up and down as needed. [20] has defined Elastic Computing as following:

> *"The quantifiable ability to manage, measure, predict and adapt responsiveness of an application based on real time demands placed on an infrastructure using a combination of local and remote computing resources."*

Chapter 2

Survey and State of the Art

2.1 Distributed Storage Systems

New generations of applications require processing terrabytes of information. Storage plays a significant role in computations. This is mainly achieved by distributed computing. Distributed computing means distributed data. Combining networking with storage and distributed computing provides us Distributed Storage Systems (DSS). DSSs are capable of achieving various things, from spanning a global network of users providing rich set of services, file sharing and high performance to global federation and utility storage [47].

With the advances of networking infrastructure and distributed computing, new DSSs were emerged through the time and they continued to evolve. However, new generation of applications have faced with many challenges such as longer delays, unreliability, unpredictability and potentially malicious behavior. This is due to the nature of public shared environments. To cope with this challenges, innovative architectures and algorithms have been proposed and developed, providing improvements mainly to consistency, security and routing.

In this section we will review some of the existing DSSs from different perspectives.

2.1.1 Google Bigtable

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size such as petabytes of data across thousands of commodity servers [18]. Bigtable has achieved several goals like wide applicability, scalability, high performance and high availability. Bigtable does not support a full relational data model instead it provides client with a simple data model that supports dynamic control over data layout and format.

Data Model

A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key and a timestamp. Each value in the map is an uninterpreted array of bytes. The data model is optimized for storing multiple versions of a content.

Building Blocks

Bigtable is a distributed hash mechanism built on top of Google File System (GFS). The underlying file format is SSTable. Queries (distributed computations) such as filtering, aggregation, statistics are done using Sawzall language [46].

A Bigtable cluster typically operates in a shared pool of machines that run a wide variety of other distributed applications. Bigtable depends on a cluster management system for scheduling jobs, managing resources on shared machines, dealing with machine failures and monitoring machine status.

Bigtable relies on a highly-available and persistent distributed lock service called Chubby [16]. A Chubby service consists of 5 active replicas. One of them is selected to be master and actively serves requests. The service is live when the majority of replicas are running and can communicate with each other. Chubby uses Paxos algorithm [17], [35] to keep its replicas consistent in the case of failures. Bigtable uses Chubby for variety of tasks: to ensure that there is at least one active master at any time; to store the bootstrap location of Bigtable data; to discover tablet servers and finalize tablet server deaths; to store Bigtable schema information and to store access control lists.

Architecture

Bigtable consists of one master and many tablet servers. Tablet servers can be dynamically added or removed from a cluster thus providing elasticity. The master is responsible for assigning tablets to tablet servers, detecting the addition and expiration of tablet servers, balancing load between tablet servers and garbage collection of files. Moreover, it manages schema changes. The tablet server handles read and write request to the tablets and also split the tablets that has grown too large.

Client data does not move through the master node. Instead clients communicate directly with tablet servers for reads and writes. As a result the master is lightly loaded in practice.

2.1.2 Amazon Dynamo

Dynamo is a high available key-value storage system that some of Amazon's core services use to provide constant availability experience. Dynamo is used to manage the state of services that require high reliability and tight control over the trade offs between availability, consistency, cost-effectiveness and performance [22].

Dynamo uses a variety of well-known techniques to achieve scalability and availability: data is partitioned and replicated using consistent hashing [30], and consistency is provided by object versioning [34]. The consistency among replicas is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo exploits a gossip-based distributed failure detection and membership protocol.

Data Model

Dynamo has a primary-key-only interface. Data are stored as key-value pairs and the only interface to access data is the key. Data is hashed and replicated. The hashed key range can be distributed among the available machines. Data can be requested from a random machine. Each machine has enough routine information to forward the request if it does not have the value corresponding to the key.

Building Blocks

In Dynamo each storage node has three main components: request coordination, membership and failure detection, and a local persistence engine. The request coordination is built on top of an event-driven messaging that is very similar to Staged Event Driven Architecture (SEDA) [50]. The coordinator executes request for reading and writing on behalf of the client. Each client request leads to creation of a state machine on the receiving node that is responsible for handling the request.

Dynamo is designed to be highly available for writing as opposed to reading, since failure of writing inconveniences the end-user of the application. So, any data conflicts are resolved at the time of reading rather than writing.

The system is completely decentralized with the least need for manual administration. However, new machines have to be manually added since downtimes of systems are considered to be usually temporary which means it is wasteful to redistribute that machine's data to other machines in the meanwhile. However, when a new machine is added to the ring, the system automatically starts sharing part of the responsibility by handing over a part of the key range. Dynamo's SOA is demonstrated in Fig. 2.1.

2.1.3 Hypertable

Hypertable is a scalable high performance, distributed storage and processing system for structured and unstructured data. It is mainly designed to handle the storage and information processing on a huge cluster of commodity servers, providing resilience to machine and component failures [3].

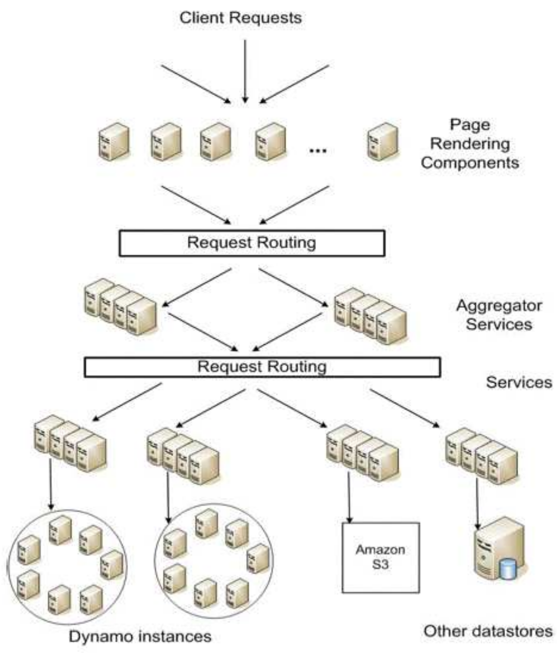


Figure 2.1. Amazon's Service Oriented Architecture

Data Model

Hypertable consists of multi-dimensional table that can be queried using a single primary key. The first dimension is the row key that acts as the primary key. It defines the physical order of data to be stored. The second dimension is the column that is very similar to traditional databases. The third dimension is the column qualifiers. Within each column family there can be infinite number of qualified instances in theory. The last dimension is the time dimension. This dimension consists of a timestamp that is assigned by the system providing versioning of the content. The data model is very similar to the Google Bigtable that is described in section 2.1.1.

Architecture

Hypertable is designed to run on top of a third-party distributed filesystem such as Hadoop DFS. However, it can be run on top of a local filesystem. The general system architecture of Hypertable is depicted in Fig. 2.2. In the following we will describe each component briefly.

Hyperspace is analogous to Chubby [16] from Google. Hyperspace provides a filesytem for storing small portion of metadata. It also acts as a lock manager. At

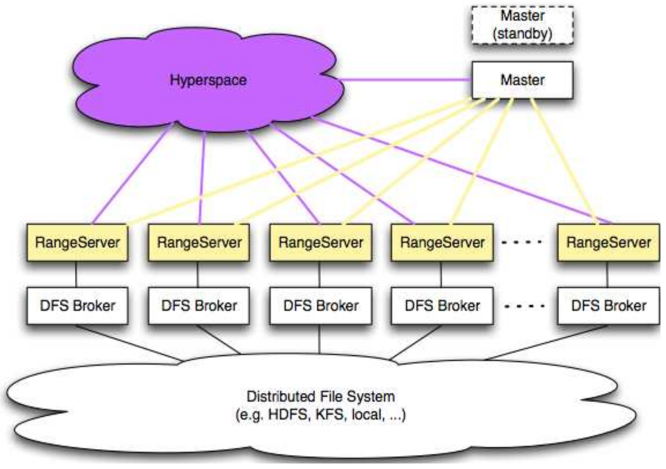


Figure 2.2. Hypertable System Architecture (Figure is taken from Hypertable wiki page)

the time of writing this section, it is implemented as a single server, but will be made distributed in future.

Range server Tables are broken into a set of continuous low ranges, each of which is managed by a range server. The splitting process is continues for all the ranges as they keep on growing. Each range server handles all reads and writes for the data table it is responsible for.

Master handles all meta operations such as creating and deleting of tables. Like Bigtable, client data does not move through the master. Hence the master can be down without even the clients are aware of it. The master is also responsible for failure detection of range servers and reassign them the ranges. Master is also responsible for load balancing.

DFS broker Hypertable is capable of being run on top of any filesystem. In order to do that, it has abstracted the interface to filesystems through something called DFS brokers. The DFS broker is a process that translates standardized filesystem protocol messages into the system calls that are unique to the specific filesystem

2.1.4 Yahoo! PNUTS

PNUTS [21] is a massively parallel and geographically distributed database system for Yahoo! web applications. Data is stored as hashed and ordered tables. It provides low latency for large numbers of concurrent requests including updates and queries. Moreover PNUTS facilitates automated load balancing and failover to reduce operational complexity.

Data Model

PNUTS presents a simplified relational data model that data is organized into tables of records with attributes. Schemas are flexible which means that new attribute can be added at any time. It is mainly designed for online serving workloads that consist mostly of queries that read and write single records or small group of records.

System Architecture

PNUTS is divided into regions that each region contains a full complement of system components and a complete copy of each table. The system architecture is depicted in Fig. 2.3. The regions are usually geographically distributed. In order to provide reliability and replication, PNUTS uses publisher/subscriber mechanism. The replication of data to multiple regions provides additional reliability.

Storage units store tables, respond to get and scan requests by retrieving and returning matching records and respond to set requests by processing update. Storage units can use any physical storage layer that is proper. In order to decide which storage unit is responsible for a given record and also for finding the corresponding tablet that has that record, router is used to facilitates the system for such functionalities. Routers contain only a cached copy of the internal mapping. They poll the tablet controller to get any changes to the mapping. Tablet controller is also responsible for moving tablets between storage units for load balancing or recovery when a large tablet must be split.

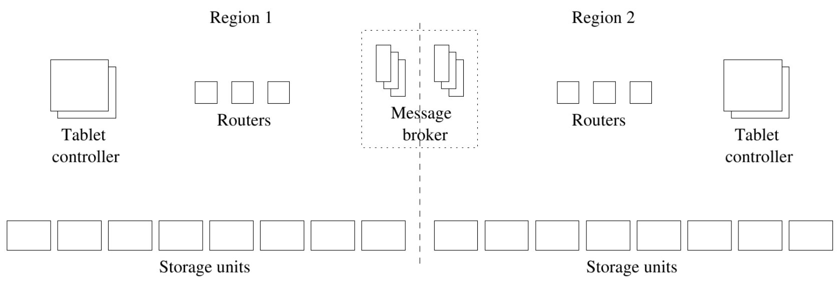


Figure 2.3. PNUTS System Architecture (Figure is taken from [21])

If a router fails, a new one will be started and there is no recovery. The controller is not a bottleneck since it does not sit on the data path. The primary bottleneck in the system is disk seek capacity on the storage units and message brokers.

Yahoo! Message Broker PNUTS uses asynchronous replication to ensure low latency updates. Yahoo! message broker is a topic-based publisher/subscriber mechanism that acts as a replacement for redo log and replication mechanism. Data updates are considered commited when they have been published to this service and the update will be propagated to different regions for various replicas.

2.1.5 Apache HBase

Apache Hbase [2] is an open source answer to Google Bigtable ([18]). It is built on top of Hadoop which implements functionality similar to Google Filesystems and MapReduce systems.

Data Model

HBase uses data model that is very similar to Bigtable. Data is logically organized into tables, rows and columns. Applications and different services store data rows in labeled tables. A data row has a sortable row key and an arbitrary number of columns. The table is stored sparsely, so those rows in the same table can have varying numbers of columns. HBase stores the column families close on disk and provides good physical locality. By default only one row may be locked at a time. Row writes are always atomic. But it is possible to lock a single row and perform both read and write operations. However, there is possibility to lock multiple rows but at the time writing this section, it is not the default behavior and should be enabled.

From an application perspective, a table seems to be a list of tuples sorted by row key ascending, column name ascending and timestamp descending. However, tables are broken up physically into row ranges called regions (analogous to Bigtable tablets). Obviously a set of regions that are sorted forms a table. Each column family in a region is managed by an HStore. Each HStore may have one or more MapFiles (a Hadoop HDFS file type) that is very similar to a Google SSTable. Like SSTables, MapFiles are immutable once closed. MapFiles are stored in the Hadoop HDFS

Architecture

There are three main components within HBase that we will discuss in this section. The system architecture is demonstrated in Fig. 2.4.

HBaseMaster is responsible for assigning regions to HRegionServers. The first region to be assigned is the ROOT region which locates all the META regions to be assigned. Each META region maps a number of user regions which comprise the multiple tables that a particular HBase instance serves. Once all the META regions have been assigned, the master will then assign user regions to the HRegionServers, attempting to balance the number of regions served by each HRegionServer. The HBaseMaster also monitors the health of each HRegionServer, and if it detects one is no longer reachable, it will split the HRegionServer's write-ahead log so that there is now one write-ahead log for each region that the HRegionServer was serving.

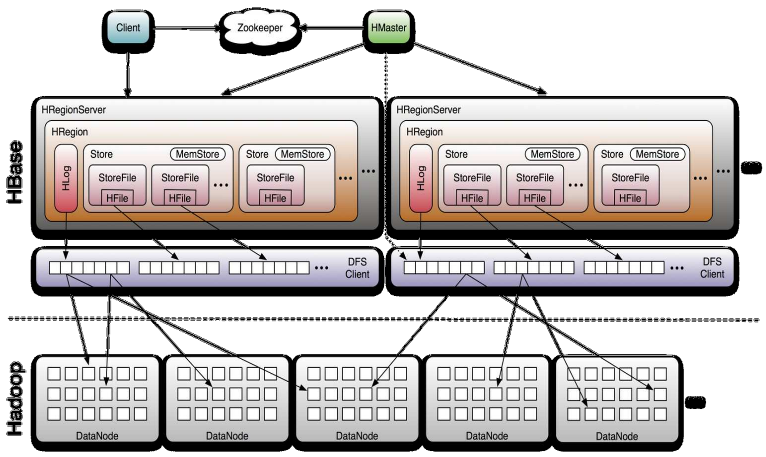


Figure 2.4. HBase System Architecture

After it has accomplished this, it will reassign the regions that were being served by the unreachable HRegionServer. At the time of writing this section, when the HBaseMaster dies the whole cluster will shut down.

HRegionServer is responsible for handling client read and write requests. It communicates with the HBaseMaster to get a list of regions to serve and to tell the master that it is alive.

HBase client is responsible for finding HRegionServers that serve the particular row range of interest. On instantiation, the HBase client communicates with the HBaseMaster to find the location of the ROOT region. This is the only communication between the client and the master. Once the ROOT region is located, the client contacts that region server and scans the ROOT region to find the META region that will contain the location of the user region that contains the desired row range. It then contacts the region server that serves that META region and scans that META region to determine the location of the user region. After locating the user region, the client contacts the region server serving that region and issues the read or write request.

2.1.6 Terrastore

Terrastore is a distributed, scalable and consistent document store supporting single-cluster and multi-cluster deployments. It provides advanced scalability support and elasticity feature without loosening the consistency at data level. Terrastore provides ubiquity by using universally supported HTTP protocol.

Data is partitioned and distributed among the nodes in the cluster(s) with automatic and transparent re-balancing when nodes join and leave. Moreover, it distributes the computational load for operations like queries and updates to the nodes that actually hold the data. In this way Terrastore facilitates with scalability at both data and computational layers.

Terrastore is based on an industry-proved clustering solution called Terracotta [7]. Terracotta is used as a distributed lock manager for locking single document access during write operations, as an intra-cluster group membership service, and for durable document storage (and replication).

Data Model

Data model is pure JSON [4] which is stored in documents and buckets which are analogous to table row and table correspondingly in relational DBs. Data (documents and buckets) is partitioned according to the consistent hashing schema [30] and is distributed on different Terrastore servers.

Building Blocks and Architecture

Terrastore system consists of an ensemble of clusters that in each cluster can exist one Terrastore master and several Terrastore servers. The system architecture is shown in Fig. 2.5

Master is responsible for managing the cluster membership: hence it notifies when the servers join/leave, changing the group view. In addition to this membership management, Master is also responsible to durably store the whole documents. It is also responsible for replicating the data to server nodes but it does not partition the data itself and partitioning strategy is decided by the server nodes which is either the default consistent hashing or a user defined one. Replication is a pull strategy performed by server nodes from the master node. Hence each server requests its own partition from the master. All the writes go through the master but only the first read request goes through the master and later requests will be read from the server memory.

Each server owns a partition to which a number of documents are mapped. Each document is only own by one server node. If a request is sent to server that does not own the document, then the request is routed to the corresponding server. All the write requests go to both the server that owns the document and the master node.

The role of ensemble is to join multiple clusters and make them work together. It provides better scalability by providing multiple active masters. It also facilitates the whole system partition-tolerance behavior. Thus in the case of partitioning the data will be available locally but it can not be seen by other clusters except the cluster owns the data.

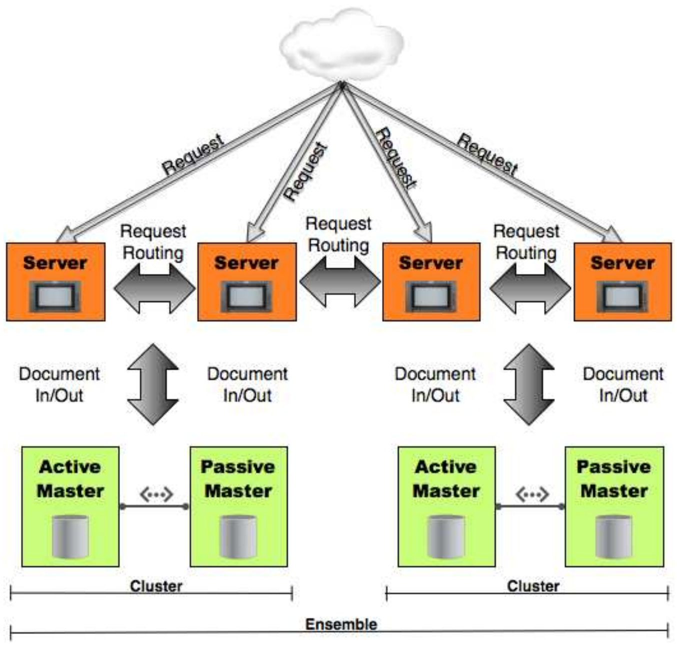


Figure 2.5. Terrastore System Architecture

2.1.7 Self-* in Distributed Storage Systems Context

In this section we will briefly describe what each of the properties defined in section 1.2.1 means in the context of Distributed Storage Systems by some examples and scenarios.

Self-configuration

- A new storage node joins the system and starts communicating with other nodes. Other nodes should be able to configure themselves to consider the newly joined node. The new node should be able to receive the latest configuration from any existing node and adopt itself to that. Nodes in the system should be able to adjust new configuration according to any join/leave in general.

- All nodes in the system have a reference configuration model that can be obtained from another node. The node is responsible for converging itself to this model and not deviating from it. The model can be updated at run-time and can be distributed through the system (e.g. by means of a gossip-based protocol)

Self-optimization

- Nodes optimize themselves to make better routing for upcoming read requests considering some critical factors (e.g. load balancing)
- Nodes would optimize the number of replicas for each data set. The number of replicas can be increased/decreased upon its usage.
- Optimization for load balancing as new data might come under more read-/write requests.

Self-healing

- In case of failure for a node, there should be a node that can restart the failed node from its latest stable state.
- In case of split brain[1], the inconsistencies should be resolved automatically and without the intervention of administrators

2.1.8 Summary

The studied Distributed Storage Systems (DSS) are summarized in Table 2.1.

Table 2.1: Summary of Distributed Storage Systems

DSS	DB Type	Architecture	Focus	Elasticity	Consistency
Google Bigtable [18]	Column-oriented	Cluster of master/server nodes	Scalability, Performance, Availability	Yes	Relaxed
Amazon Dynamo [22]	Key-value	Cluster of Node rings	Availability, Consistency, Performance	Yes	Eventual
Hypertable [3]	Column-oriented	Cluster of master/server nodes	Scalability, Performance, Failure resilience	Yes	Strong
Yahoo! PNUTS [21]	Key-value	Cluster of regions	Performance, load balancing, Scalability	Yes	Eventual
HBase [2]	Column-oriented	Cluster of master/server nodes	Scalability, Performance, Availability	Yes	Strong
Terrastore [8]	Document	Ensemble of clusters of master/server nodes	Consistency, Scalability, Availability	Yes	Eventual

[1]In clustering split brain means that all private links go down simultaneously, but the cluster nodes are still running

2.2 Application of Control Theory in Computing Systems

In this section we will investigate common approaches within control theory. Controller design consists of two important steps: *System Identification* and *Controller design* [45]. Each of which will be discussed in the next section.

2.2.1 System Identification

System identification deals about how to construct a model to identify a system. In this phase a transfer function is constructed. This transfer function connects past/present input values to past/present output values. This constructs a model for the system. Based on the transfer functions and desired properties and objectives a control law is chosen. System identification is mainly divided into the following approaches:

First principle approach is one of the de facto approaches to identification of computer systems [26]. it can be considered as a consequence of queue relationship. First principle approach is developed based on knowledge of how a system operates. In some studies and systems like [19], [33], [38], [29],[10], [48], [9], [37], [36], [42] and [49] this approach has been used. However there are some shortcomings with this approach that has been stated in [45]. It is very difficult to construct a first principle approach for a complex system. Since this approach considers detailed information about the target systems, it requires an on going maintenance by experts. This approach does not address model validation.

Empirical approach It starts by identifying the input and output parameters like the first principle approach. But rather than using a transfer function, an autoregressive, moving average (ARMA) model is built and common statistical techniques are employed to estimate the ARMA parameters [45]. This approach is also known as Black box [26]. This approach required minimal knowledge of the system. Half of the system in our studies have employed this approach for system identification like [31], [45], [24], [39], [44], [41].

2.2.2 Automated control of multiple virtualized resources

In [44] a resource control system called *AutoControl* has been designed that is able to adapt to dynamic workloads to meet certain Service Level Objectives (SLO). It is a combination of an online model estimator and a multi-input and multi-output (MIMO) resource controller. Model estimator captures the complex relationship between application performance and resource allocation. MIMO controller allocates the right amount of virtualized resources to achieve application SLO. Logical controller architecture is shown in Fig. 2.6

For each application, its AppController periodically polls measured performance. This performance is compared to target application performance. In the case of dis-

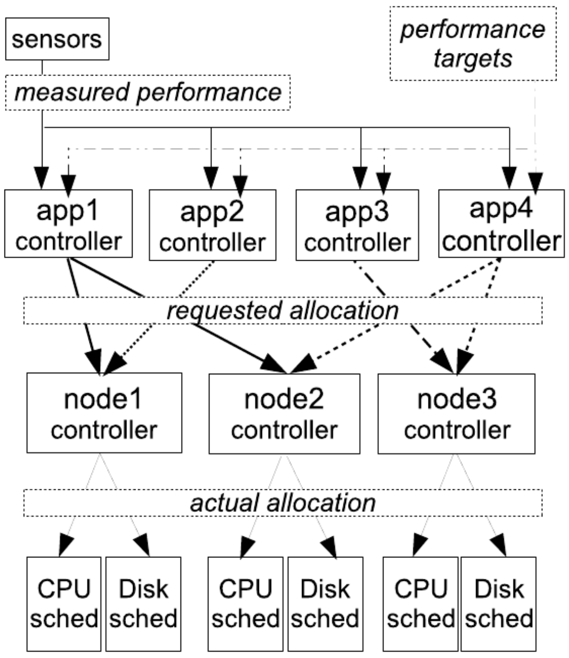


Figure 2.6. AutoControl - Logical Controller Architecture

crepancy, it determines the resource allocation needed for the next control interval and sends these requests to the NodeControllers for the node hosting and individual tiers of the application. Design of AutoControl allows the placement of AppControllers and NodeControllers in a distributed fashion.

For each node, based on the aggregated requests from all AppControllers, the corresponding NodeController determines whether it has enough resource of each type to satisfy all requirements.

AutoControl enables dynamic redistribution of resources between competing applications to meet their targets. It is evaluated using two testbeds consisting of varying number of Xen virtual machines. Experimental results confirms that AutoControl can detect dynamically-changing CPU and disk bottlenecks across multiple nodes and can adjust resource allocation according to defined application SLO .

2.2.3 MIMO Control of Apache Web Server

In [24] a multi-input multi-output controller is designed to handle performance trade-offs and external disturbances for Apache Web Server. Performance metrics include end-user response times, response times on the server, throughput, utilizations of various resources on the server. Control outputs are CPU and Memory utilizations. Control inputs are selected as MaxClients and KeepAliveTimeout. The former limits the the size of worker pool and the latter limits the user think time, the time between an HTTP reply and the receipt of the next client request.

Modeling of Apache web server is done by black box approach and ARX time variant model is chosen for System identification. Logical controller architecture is demonstrated in Fig. 2.7.

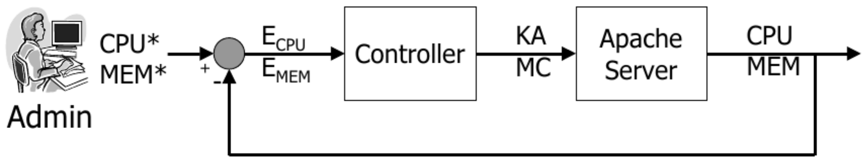


Figure 2.7. Apache Web Server - Logical Controller Architecture

System dynamics for Apache is as follows:

$$\begin{bmatrix} CPU_{k+1} \\ MEM_{k+1} \end{bmatrix} = A. \begin{bmatrix} CPU_k \\ MEM_k \end{bmatrix} + B. \begin{bmatrix} KA_k \\ MC_k \end{bmatrix} \tag{2.1}$$

Because of robustness and narrowing down the space needed for a decision, Proportional Integral controller is selected. Description regarding this type of controller will appear in the coming sections. The authors showed in their paper that using MIMO technique is beneficial for computing systems that acts in non-linear way. They have also compared Controller design with pole placement and LQR techniques. They showed using LQR technique will end up having less agressive controller with smaller gains. As a result of this technique the close loop system is less oscillatory.

2.2.4 Controller Analysis and Design

In this section we briefly go through different controller variation designs.

Properties of Feedback Control Systems

Below we will go through a list of properties of interest for computing systems that is stated in [26]:

- A system is said to be *stable* if for any bounded input, the output is bounded also.
- The control system is *accurate* if the measured output converges to the reference input. For a system that is in steady state, its inaccuracy or steady-state error is the steady state value of the control error.

- The system has *short settling times* if it converges quickly to its steady-state value.
- The system should achieve its objectives in a manner that *does not overshoot*.

These properties often referred as SASO (stable, accurate, settling time, overshoot) in the literatures.

Proportional Controller

A proportional control (PC) system is a type of linear feedback control system that simply can relate control error to control input. It is more complex than on-off control system but still simpler than proportional integral derivative (PID) that will be discussed in the next section. PC is inherently inaccurate for a step input. PC always tries to synchronize the actual monitored property to the target property according to the observed error. In other words, the controller output is proportional to the error signal, which is the difference between set points (target value) and the process variable (current status). This can be formulated as follows:

$$P_{out} = K_p e(t) \tag{2.2}$$

in which P_{out} is the output of the proportional controller, K_p is proportional gain, $e(t)$ is error at time t that is defined as $e(t) = SP - PV$. SP is the set point and PV is the process variable. Choice of K_p constitutes the design problem for proportional control and involves trade-off. Choosing larger K_p improves accuracy. However a sufficiently large K_p causes settling time and the maximum overshoot to increase and may cause instability.

There are four main desirable properties of controller in computer systems that we need to pay enough attention when designing a controller. First the controller requires to be *stable*. It should result in a stable closed-loop system. Second the controller needs to be *accurate*. The third property is the *settling time* of the system, the time needed for an output to reach new steady-state value after a change in one of the inputs. The final property is the *maximum overshoot* which is defined as the largest amount by which the transient response exceeds the steady state value as a result of a change in an input, scaled by the steady-state value [26].

Proportional Integral Derivative (PID) Controller

One drawback to PC is that the steady-state is unavoidable. Integral control can drive the steady-state to zero. Derivative control provides a way to respond quickly. Putting proportional, integral and derivative control gives us the PID. It is widely used in industrial systems [13] and it is the most commonly used feedback controller. In the absence of knowledge of the underlying process, PID controller is the best the controller [14]. It calculates an error value as the difference between a measured process variable and a desired set point. The controller then attempts to minimize the error by adjusting the process control inputs. This error is the sum of three control terms and can be formulated as bellow:

$$P_{out} = K_p e(t) \tag{2.3}$$

$$I_{out} = K_i \int_0^t e(x)dx \tag{2.4}$$

$$D_{out} = K_d \frac{d}{dt} e(t) \tag{2.5}$$

$$MV(t) = P_{out} + I_{out} + D_{out} \tag{2.6}$$

in which K_p, K_i and K_d are proportional, integral and derivative gains correspondingly. P_{out} is the output of proportional controller, I_{out} is the output for integral controller, D_{out} is the output for deviative controller and $MV(t)$ is the manipulated variable at time t. These value can be determined by empirical methods.

Self Tunning Regulators

Self tunning regulators can reduce the complexity of controller design by reducing manual tunning. They change controller parameters at each sample time based on updated estimates of the model of the target system. They adapt dynamically to the characteristic of the target system. Self tunning regulators composed of two loops, an inner loop which consists of the process and an ordinary linear feedback regulator, and an outer loop which is composed of a recursive parameter estimator and a design calculation, and which adjusts the parameters of the regulator.

The main drawback to STR is that it tends to be slow when it comes to rapid and abruptive changes in workloads or configurations. Moreover, adjusting control parameters at each sample time may not be effective for some computing systems since it is difficult to figure out if the environment has changed or not [26].

2.2.5 Desired Objectives

In this section we will briefly review the desired objectives and tunning parameters and some of the properties generally for selecting sensor parameters in different literatures for the target systems.

Properties

[39] and [24] have defined a practical set of properties for choice of metric for the target system that we bring them as stated, selected parameters should:

- be *dynamically* changeable
- affect the selected performance metric in a meaningful way

- be *easy* to measure
- be *stable*
- *correlate* to the measure of level of service

CPU Utilization

In [39] the authors showed that CPU Utilization strongly correlates with overall response time in the storage layer when the bottleneck is in the storage layer. Hence they have selected CPU Utilization as one of the sensors to get feedback for the controller.

Response Time

Response time is defined as how quickly an interactive system can respond to user input. In section 2.2.5 it is stated that [39] showed that CPU Utilization is correlated to response time. They have used response time also as one of their feedbacks from the target system.

In [29] the authors demonstrated that the response times can be bound to the requests while maintaining a high throughput under overload. As client load increases, the throughput increases until the load reaches a threshold after which the throughput drops and response time grow.

Latency

Latency is a measure of time delay experienced in a system, the precise definition of which depends on the system and the time being measured. In Triage system [31], the authors had latency goal in mind for all workload requests. They showed that latency depends mostly on the characteristic of the corresponding system.

Throughput

Throughput is the average rate of successful request delivery to a system in a specific period of time. In Triage [31] performance isolation between workloads that compete for system throughput has been investigated under different scenarios.

2.2.6 Summary

In Table 2.2 all the studied systems have been summarized based on three criteria: *system identification*, *controller design* and *desired objectives*.

Table 2.2: Summary of Computing Systems that exploit Control Theory

System	**System Identification**	**Controller Design**	**Desired Objectives**

Triage [31]	Black box	Direct self-tunning regulator adaptive controller	Latency, Max. Throughput
Yaksha [29]	First principle	Self-tunning proportional integral controller	Response time, Throughput
Automated Cloud Control [40]	Not mentioned	Integral controller	CPU Utilization
Lotus Notes [45]	Black box	Saturated integral controller	Max. Users, Queue Length
Apache Web Server [24]	Black box	PI controller	CPU, Memory
Elastic Storage [39]	Black box	Integral controller	CPU utilization, Response time
Apache Web Server [10]	First principle	PI controller	Response time, Service time
Automated Control [44]	Black box	MIMO Resource controller	Disk/CPU allocation, response time and throughput
Web Server [41]	Black box	Root-Locus Controller	Relative delays, Response time
Web Server [48]	First principle	PI controller	Queue length
Server [9]	First principle	PI Controller	Utilization
QoS Adaptation [37]	First principle	PID Controller	Responsiveness of controller
Mass Storage [36]	First principle	Feedback-controlled leaky Bucket	Throughput, Response time

Chapter 3

Control Analysis and Design

In this chapter we first define and describe the problem we try to approach. Then we show the methodology to identify the investigated system and its behavior. And finally this chapter will end by the analysis and design of the controller for the described system.

3.1 Problem Definition and System Description

In section 1.2.2 we defined briefly what Autonomic Computing is. The problem we are interested to solve is the management of Elastic Storage instances within a cloud computing environment. We have already gone through the benefits of autonomic computing. The type of management we are looking for is Autonomic. Cloud environment is very dynamic. Any system that is running in a cloud can scale-up and/or down in only few minutes. In-time and proper decisions against the changes in the environment is very critical when it comes to enterprise and scalable application.

The type of cloud system we are interested at is a distributed storage system that provides files to end users. The overall architecture of the system is depicted in Fig. 3.1.

Consumers (end users) request files that are located in one of the storage cloud node (instance). All the requests are arrived at the Elastic Load Balancer that sits in the front of all storage instances. Elastic Load Balancer decides to which instance the request should be dispatched. Simply Elastic Load Balancer tracks the CPU load and the number of requests sent previously to each instance and at each time it determines the next node that can serve the incoming request. In addition to performance metrics that it tracks, ELB has the file tables with the info showing where each file is located since more than one instance can have the same file in order to satisfy the replication degree.

The actual scaling up/down is performed by Cloud Provider. The main respon-

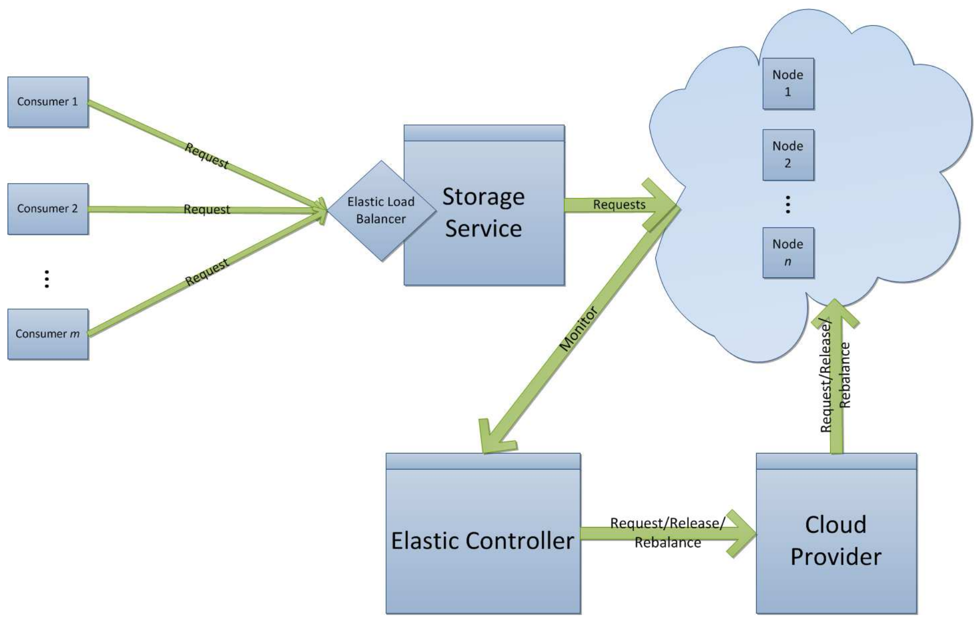


Figure 3.1. Storage Cloud Architecture

sibility of Cloud Provider is to launch a new instance or terminate an existing one. In addition it can issue rebalance of files from one instance to another.

A simple scenario is that consumer m sends a request to Elastic Load Balancer to download a file called `scala.pdf` that is located on three nodes namely n_1, n_2 and n_3. ELB decides which node the request should be dispatched to. Assume node n_2 is elected. It receives the request and start uploading the file to the consumer.

In order to add the autonomity functionality to this system we have chosen to use an elastic controller that is connected to cloud provider and monitors the instances currently running in the cloud. This controller is responsible for helping cloud provider in order to scale up and down the number of instances.

In the rest of this chapter we investigate how to identify the described system using control theory techniques and how to design a controller for such system.

3.2 System Identification

In this section we describe how we approach constructing a model that can identify the system behavior. This is the key to design a controller for such system.

System identification allows to build a mathematical model of a dynamic system based on measured data. Number of systems are studied in Chapter 2. These systems use two main approaches to system identification: *first-principle* and *black-box*. However most of the studied system have constructed a black-box model rather than a first-principle model. This is mainly because the relationship between inputs and outputs of the system is complex enough that first-principle system identification can not be used easily.

We also use black-box model to identify system behavior together with some knowledge of the system to build state-space model as is described in the next section. Before diving into state-space model we review the basic steps in any system identification regardless of the used method.

3.2.1 Basic Steps of System Identification

The steps can be simplified as following items:

- Design an experiment and collect input/output data
- Examine the data. Polish (preprocessing) it and remove trends. Select useful portion of original data.
- Select and define a model.
- Examine the obtained model's properties.
- Simulate the model and study the system behavior. If the model does not represent the system refine the data and model and repeat the steps from beginning.

3.2.2 State Space Model

State space model provides a scalable approach to model systems with a large number of inputs and outputs [26]. State-space model allows us to deal with higher order target systems without a first-order approximation. Since the studied system is a cloud environment and such environments are highly dynamic we prefer to choose state space model as the system identification approach. Another benefit of using state-space model is that it can be extended easily. Suppose that in time we find more parameters to control the system. This can be managed by state-space model without affecting the characteristic equations as will be seen later on in this chapter.

The main idea behind this approach is to characterize how the system operates in terms of one or more variables. These variables may not be directly measurable even. However they can be sufficient to express the dynamics of the system. These variables are also called *state variables*.

3.2.3 State Variables

In order to define the state variables for a system, first we need to define the measured inputs and outputs since the state variable is defined based on these two. Measured input and outputs for cloud are demonstrated in Fig. 3.2.

System input (number of nodes) is represented by $\texttt{NN}(k)$ at time k and measured system outputs are represented by:

- *average CPU load* $\texttt{CPU}(k)$: average CPU load of all instances currently running in the cloud system.
- *interval total cost* $\texttt{TC}(k)$: total cost for all instances at each interval the sesing is done.
- *average response time* $\texttt{RT}(k)$: time that is required to start a download on an instance.

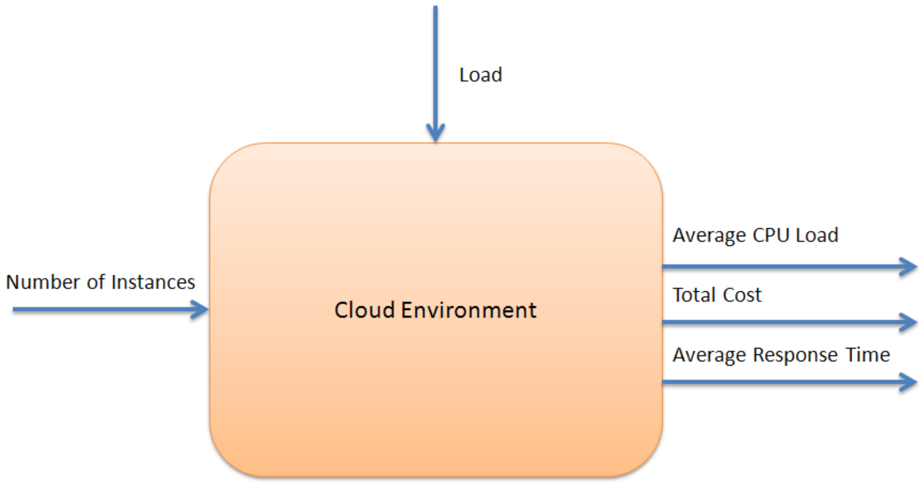


Figure 3.2. Scalar Block Diagram

This diagram shows that there are one input and five outputs for the system. We can consider four internal states for the system that are not equivalent to measured outputs.

We skip defining internal state for average load $\texttt{TP}$ since we are not interested in this value and it is only used for a better estimation of the internal states based on load as we will see later. Moreover load can not be controlled thus there would be no reason to include in our internal states and it can be considered as a noise to system.

The value of each state variable at time k is shown by $x_1(k)$, $x_2(k)$ and $x_3(k)$. The vector representation of the state variables is

$$\mathbf{x}(k) = \begin{bmatrix} x_1(k) \\ \vdots \\ x_3(k) \end{bmatrix}$$

The offset value for input is $u_1(k) = \texttt{NN}(k) - \widehat{\texttt{NN}}$ which $\widehat{\texttt{NN}}$ is the operating point for the input. The offset values for outputs are

$$y_1(k) = \texttt{CPU}(k) - \widehat{\texttt{CPU}} \qquad (3.1)$$
$$y_2(k) = \texttt{TC}(k) - \widehat{\texttt{TC}} \qquad (3.2)$$
$$y_3(k) = \texttt{RT}(k) - \widehat{\texttt{RT}} \qquad (3.3)$$

in which $\widehat{\texttt{CPU}}$, $\widehat{\texttt{TC}}$ and $\widehat{\texttt{RT}}$ are operation points for each output. Input and output offset vectors are represented like the following

$$\mathbf{u}(k) = \begin{bmatrix} u_1(k) \end{bmatrix}$$

and

$$\mathbf{y}(k) = \begin{bmatrix} y_1(k) \\ \vdots \\ y_3(k) \end{bmatrix}$$

3.2.4 State Space Model

State space model uses state variables in two ways. First it uses state variables to describe the dynamicity of the system and how $\mathbf{x}(k+1)$ can be obtained from $\mathbf{x}(k)$. Second it obtains the measured output $\mathbf{y}(k)$ from state $\mathbf{x}(k)$. Fig 3.3 shows how the input, output and state variable relate to the studied system.

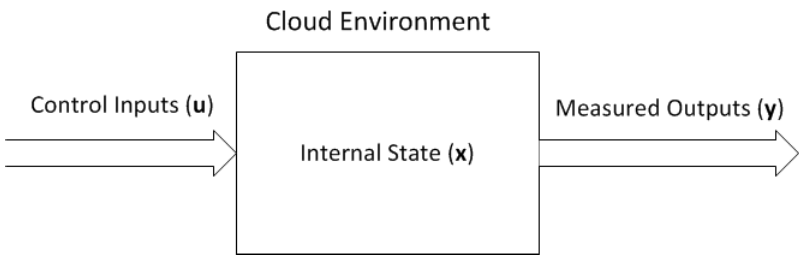


Figure 3.3. State Space Model

State-space dynamics for a system with n states are written as

$$\mathbf{x}(k+1) = \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k) \qquad (3.4)$$
$$\mathbf{y}(k) = \mathbf{C}\mathbf{x}(k) \qquad (3.5)$$

where $\mathbf{x}(k)$ is an $n \times 1$ vector of state variables, $\mathbf{A}$ is an $n \times n$ matrix, $\mathbf{B}$ is an $n \times m_I$ matrix, $\mathbf{u}(k)$ is an $m_I \times 1$ vector of inputs, $\mathbf{y}$ is an $m_O \times 1$ vector of outputs and $\mathbf{C}$ is an $m_O \times n$ matrix.

Having equations 3.4 and 3.5, we can discuss how outputs at time $k+1$ are related to internal states at time k. We can go through each state variable and write the state-space dynamics:

- Average CPU Load (`CPU`): is dependant on the number of nodes in the system and previous CPU load, thus it becomes

$$\begin{aligned} x_1(k+1) = \texttt{CPU}(k+1) &= a_{11}\texttt{CPU}(k) \\ &\quad + b_{11}\texttt{NN}(k) \\ &\quad + 0 \times \texttt{TC}(k) + 0 \times \texttt{RT}(k) \end{aligned}$$

- Total Cost (`TC`): is dependant on the number of nodes in the system (more nodes we have, more money we should pay) and previous `TC` hence it becomes

$$\begin{aligned} x_2(k+1) = \texttt{TC}(k+1) &= a_{21}\texttt{TC}(k) \\ &\quad + b_{21}\texttt{NN}(k) \\ &\quad + 0 \times \texttt{RT}(k) + 0 \times \texttt{CPU}(k) \end{aligned}$$

- Average Response Time (`RT`): is dependant on number of nodes in the system and CPU load, so it is

$$\begin{aligned} x_3(k+1) = \texttt{RT}(k+1) &= a_{31}\texttt{CPU}(k) + a_{33}\texttt{RT}(k) \\ &\quad + b_{31}\texttt{NN}(k) \\ &\quad + 0 \times \texttt{TC}(k) \end{aligned}$$

The last line in each equation is for those state variables that do not affect the corresponding state variable definition. Thus their coefficient is zero. This is to show that we are sure there is no relation between those state variables and their existence in the equations is for the sake of clarity. As a proof one should do a sensitivity analysis to investigate the lack of relation but this would be out of scope of this project.
The output for the system is defined like below:

$$y_1(k) = x_1(k) \tag{3.6}$$
$$y_2(k) = x_2(k) \tag{3.7}$$
$$y_3(k) = x_3(k) \tag{3.8}$$

The outputs are the same as internal state of the systems. That is why the matrix `C` is a diagonal matrix of 1's. The matrices of coefficients look like this:

$$\mathtt{A} = \begin{bmatrix} a_{11} & 0 & 0 \\ 0 & a_{22} & 0 \\ a_{31} & 0 & a_{33} \end{bmatrix} \tag{3.9}$$

$$\mathtt{B} = \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix} \tag{3.10}$$

$$\mathtt{C} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.11}$$

3.2.5 Parameter Estimation

In the previous section we found the dynamics for the system that are in form of equations 3.6 to 3.6. There are two matrices A and B with the coefficients for the equations. Before solving equation 3.4 we need first to calculate the coefficients namely matrices A and B.

Parameter estimation can not be done unless there exist experimental data. We have implemented a simulation framework of an entire cloud system that is explained in details in Chapter 4. Using the framework we can obtain experimental data (see Chapter 5). We use a complete range for the number of nodes in the system and we increase it from a to b and then back from b to a in a fixed period of time. In this way we can assure the coverage of input signal to the system is complete enough to cover the output signals in their operating regions.

Once training data is collected, they can be used by *multiple linear regression* method to compute the coefficients. Then `regress(y,X)` function can be used from Matlab to calculate the coefficients. This is discussed in Chapter 5 in details.

3.2.6 Solving State Dynamics

Now consider that the coefficients are calculated, in order to solve the state dynamics we use the following equations:

$$\mathtt{x}(k) = \mathtt{A}^k\mathtt{x}(0) + \sum_{i=0}^{k-1} \mathtt{A}^{k-1-i}\mathtt{Bu}(i) \tag{3.12}$$

$$\mathtt{y}(k) = \mathtt{Cx}(k) = \mathtt{CA}^k\mathtt{x}(0) + \mathtt{C}\sum_{i=0}^{k-1} \mathtt{A}^{k-1-i}\mathtt{Bu}(i) \tag{3.13}$$

in which we need to provide $\mathtt{x}(0)$ and $\mathtt{u}(0)$ as the initial condition values for each state variable and input respectively. Function `ss` from Matlab can be exploited to solve these equations.

3.2.7 Model Evaluation

Model evaluation explains how the model structure can describe the collected data. It is generally recommended to design a separate data set and evaluate the model with that. This provides better insight and guards against overfitting. One of the best approach for model evaluation is residual analysis. An example of this analysis would be a scatter plot of measured and predicted values. We will provide more on model evaluation in Chapter 5

3.3 Control Analysis and Design

In this section we describe how to analyse and design a feedback controller for our cloud system that is modeled in state-space model. There are three common architectures for state-space feedback control [26] that are briefly discussed in the following.

Static State Feedback is a multidimensional extension of proportional control that is discussed in Section 2.2.4 in which the reference input is fixed at the system's operating point. The idea behind this architecture is that the control input $u(k)$ should be proportional to the state but with an opposite sign.

Precompensated Static Control extends static state architecture by including a precompensator to accomplish reference tracking. The idea is to adjust the operating point of the control system.

Dynamic State Feedback can be viewed as a state-space analogous to PI (proportional integral) control that has good disturbance rejection properties. It both tracks the reference input and reject disturbances.

A brief comparison between the common architectures is presented in Table 3.1.

Table 3.1: Summary of State-space feedback control architectures [26]

Control Architecture	**Controller Complexity**	**Reference Tracking**	**Disturbance Rejection**	**Settling Time**
Static	Low	No	No	Short
Precompensation	Moderate	Yes	No	Short
Dynamic	Moderate	Yes	Yes	Moderate

A close investigation in this comparison reveals that Dynamic state feedback control is more suitable for cloud system although the settling time is larger than the other two. The benefit of dynamic over precompensation is the disturbance rejection. Disturbance is very likely to be observed in a dynamic environment such as cloud that elastic nodes can join and leave frequently and measured metrics collection can have delay phase and noise. Thus we choose dynamic state feedback control as our controller for autonomic management.

The design problem that needs to be solved is to select feedback gains that leads to desire controller properties specially settling times and maximum overshoot. In the next section we explain the Dynamic state feedback analysis and then we employ Linear Quadratic Regulation (LQR) which is an optimization technique to calculate the optimized feeback gains.

3.3.1 Dynamic State Feedback

In this section we describe a state-space architecture that both tracks the reference input and rejects disturbances. We need first to augment the state vector to include the control error $e(k) = r - y(k)$ in which r is the reference input. We use integrated control error which describes the accumulated control error. Integrated control error is shown by $x_I(k)$ and computed as

$$x_I(k+1) = x_I(k) + e(k) \tag{3.14}$$

The augmented state vector is $\begin{bmatrix} \mathtt{x}(k) \\ x_I(k) \end{bmatrix}$. The control law becomes

$$u(k) = -\begin{bmatrix} \mathtt{K}_p & K_I \end{bmatrix} \begin{bmatrix} \mathtt{x}(k) \\ x_I(k) \end{bmatrix} \tag{3.15}$$

where $\mathtt{K}_p$ is the feedback gain for $\mathtt{x}$(k) and K_I is the gain associated with $x_I(k)$. To understand the characteristic of dynamic state feedback control, we continue as follows. The augmented state-space model is

$$\begin{bmatrix} \mathtt{x}(k+1) \\ x_I(k+1) \end{bmatrix} = \begin{bmatrix} \mathtt{A} & 0 \\ -\mathtt{C} & 1 \end{bmatrix} \begin{bmatrix} \mathtt{x}(k) \\ x_I(k) \end{bmatrix} + \begin{bmatrix} \mathtt{B} \\ 0 \end{bmatrix} u(k) + \begin{bmatrix} 0 \\ 1 \end{bmatrix} r \tag{3.16}$$

we obtain the closed-loop model by substituting Equation 3.15 into this equation:

$$\begin{bmatrix} \mathtt{x}(k+1) \\ x_I(k+1) \end{bmatrix} = \left(\begin{bmatrix} \mathtt{A} & 0 \\ -\mathtt{C} & 1 \end{bmatrix} - \begin{bmatrix} \mathtt{B} \\ 0 \end{bmatrix} \begin{bmatrix} \mathtt{K}_p & K_I \end{bmatrix} \right) \times \begin{bmatrix} \mathtt{x}(k) \\ x_I(k) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} r \tag{3.17}$$

The characteristic polynomial is

$$det \left\{ z\mathtt{I} - \left(\begin{bmatrix} \mathtt{A} & 0 \\ -\mathtt{C} & 1 \end{bmatrix} - \begin{bmatrix} \mathtt{B} \\ 0 \end{bmatrix} \begin{bmatrix} \mathtt{K}_p & K_I \end{bmatrix} \right) \right\} \tag{3.18}$$

where *det* is the determinant function. Thus by properly choosing $\mathtt{K}_p$ and K_I we can determine the dynamics of the closed-loop system. With dynamic state feedback the measured output converges to the reference input.

3.3.2 LQR Optimal Control Design

An approach in controller design is to focus on the trade-off between control effort and control errors. *Control error* is determined by the squared values of state variables which are normally the difference from their operating points. *Control effort* is quantified by the square of $u(k)$ which is the offset of control input from operating point. By minimizing control errors we improve accuracy and reduce both settling times and overshoot and by minimizing control effort, system sensitivity to noise is reduced.

Least Quadratic Regulation (LQR) design problem is parametrized in terms of relative cost of control effort and control errors. This is determined by two matrices $\mathtt{R}$ and $\mathtt{Q}$. $\mathtt{R}$ defines the cost of control effort and $\mathtt{Q}$ defines the cost of state variables diverging from their operating point. The objective function for LQR to minimize is:

$$J = \frac{1}{2} \sum_{k=0}^{\infty} \left[\mathtt{x}^\top(k)\mathtt{Q}\mathtt{x}(k) + \mathtt{u}^\top(k)\mathtt{R}\mathtt{u}(k) \right] \tag{3.19}$$

In order for $J \geqslant 0$, $\mathtt{Q}$ must be positive semidefinite (eigenvalues of $\mathtt{Q}$ must be nonnegative) and $\mathtt{R}$ must be positive definite (eigenvalues of $\mathtt{R}$ must be positive). The steps of LQR is as follows:

1. Select the matrices $\mathtt{Q}$ and $\mathtt{R}$ in a way to satisfy the mentioned conditions
2. Compute feedback gain $\mathtt{K}$ using Matlab `dlqr` function
3. Run simulation to predict the performance based on the closed-loop system model.
4. Choose new $\mathtt{Q}$ and $\mathtt{R}$ and repeat the above steps if the performance is not appropriate

3.3.3 Controllability

In state-space model we are concerned about the relationship between the input $\mathtt{u}(k)$ and the state vector $\mathtt{x}(k)$. The term controllability means that for any reachable final state $\mathtt{x}_d$, there exists some sequence of input values $\{\mathtt{u}(0), \mathtt{u}(1), ..., \mathtt{u}(M-1)\}$ that will drive the system to state $\mathtt{x}_d$. In order to realize if a system is controllable or not, we have to construct the matrix $\mathcal{C}$ as follows:

$$\mathcal{C} = \begin{bmatrix} \mathtt{A}^{n-1}\mathtt{B} & \mathtt{A}^{n-2}\mathtt{B} & \ldots & \mathtt{AB} & \mathtt{B} \end{bmatrix} \tag{3.20}$$

where n is the number of states in the system. A linear time-invariant system is controllable if and only if $\mathcal{C}$ is invertible.

3.3.4 Stability

One of the most important properties of a system is stability. A system is called bounded-input bounded-output (BIBO) stable if there exist a positive constant M such that $|u(k)| \leq M$ for all k. This can be defined in terms of system poles. If all the poles are inside the unit circle then the system can considered as stable [23].

3.3.5 Fuzzy Controller

In this section we want to introduce a simple fuzzy controller. The main purpose for using such a fuzzy controller is optimization of the control input. Fuzzy controller uses heuristic rules to describe when the controller should take what actions. The controller we designed in the previous section is responsible for regulation. The output of Dynamic State Feedback Controller is redirected together with measured outputs to fuzzy controller. Then it decides if the control input should affect the system or not. The overall architecture for controllers is demonstrated in Fig. 3.4.

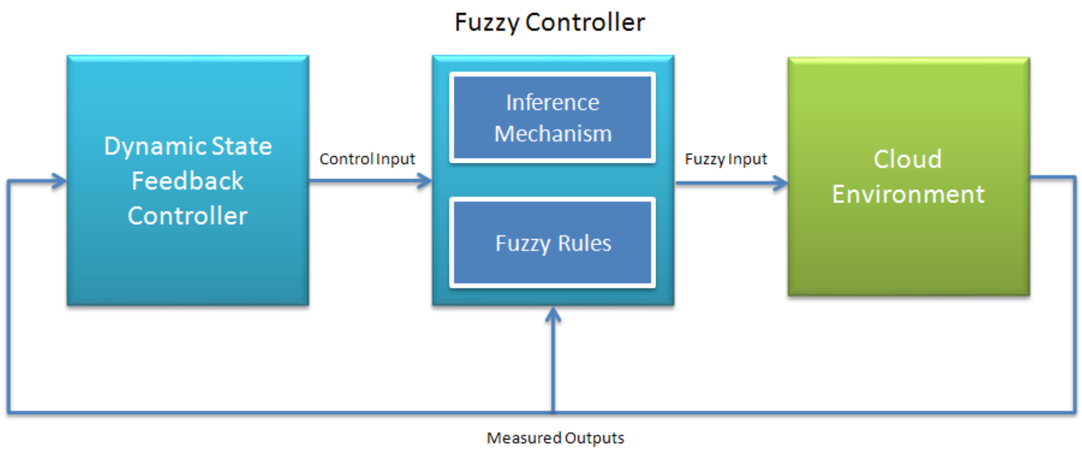


Figure 3.4. Controllers Architecture

There is one important case that dynamic state feedback controller can not act accordingly. Consider that there are some instances with high CPU load. Since the average is high, the controller will add some new instances. The new instances will be launched and started to serve requests. But at the beginning of their life cycle they have low CPU Loads so the average CPU load that is reported back to controller can be low. The controller then assumes that the situation is normal now and tries to remove some nodes.

A closer look at CPU loads reveals that we can not judge the system state by only average CPU load. Hence fuzzy controller also looks at CPU Standard Deviation. In this way if the controller orders to reduce the number of instances but there is high standard deviation for CPU loads then fuzzy controller does not allow that this control input affect the system, reducing unexpected results and confusions for the controller. This will lead to more stable environment without so many unncessary scaling up/down.

3.3.6 Method of designing controller for elastic storage in cloud

A method is given in this section to summarize the steps needed for designing a controller for elastic storages in a cloud environment. In other words, this is a summary for the detailed discussion in this chapter. The steps are in order and in the following we address them one by one:

1. Study system behavior to identify the inputs($\mathtt{u}$) and outputs($\mathtt{y}$) of the system.
2. Place inputs and outputs in $\mathbf{u}(k)$ and $\mathbf{y}(k)$ matrices respectively.
3. Select m system inputs that will be the outputs of your controller

$$\mathbf{u}(k) = \begin{bmatrix} u_1(k) \\ \vdots \\ u_m(k) \end{bmatrix}$$

   These inputs should have the highest impact in your system. In some systems there is only one input that has high impact and in other systems there are a couple of them that together have high impact.

4. Select n system outputs that will be the inputs to your controller and consider state variables matrix for them like

$$\mathbf{x}(k) = \begin{bmatrix} x_1(k) \\ \vdots \\ x_n(k) \end{bmatrix}$$

   These outputs should be related directly/indirectly to system inputs selected in previous step. Usually they are related to Service Level Agreements (SLA) and performance metrics.

5. Each state variable can depend on one or more other state variables and system inputs. Find the relation between the next value for a state variable to other state variables and system inputs like:

$$x_1(k+1) = a_{11}x_1(k) + \ldots + a_{1n}x_n(k) + b_{11}u_1(k) + \ldots + b_{1m}u_m(k)$$

$$x_2(k+1) = a_{21}x_1(k) + \ldots + a_{2n}x_n(k) + b_{21}u_1(k) + \ldots + b_{2m}u_m(k)$$

$$\vdots$$

$$x_n(k+1) = a_{n1}x_1(k) + \ldots + a_{nn}x_n(k) + b_{n1}u_1(k) + \ldots + b_{nm}u_m(k)$$

6. Extract coefficients from the previous equations into two matrices **A** and **B**. Some of the coefficients can be zero:

$$\mathbf{A}_{n\times n} = \begin{bmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \ldots & a_{nn} \end{bmatrix}$$

$$\mathbf{B}_{n\times m} = \begin{bmatrix} b_{11} & \ldots & b_{1m} \\ \vdots & \ddots & \vdots \\ b_{n1} & \ldots & b_{nm} \end{bmatrix}$$

7. In order to simplify the design of controller, we assume that outputs of the systems at each time are equal to state variables, thus matrix **C** is:

$$\mathbf{C}_{n\times n} = \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \\ 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{bmatrix}$$

8. Design an experiment in which the system is fed with its inputs. Inputs in the experiment should be changed to cover their ranges at least one time. A Range for an input is the interval that the possible values for the input will most likely end up. The selection of an interval can be modeled from industry best practices. All inputs and outputs should be collected at a fixed time interval T. Put collected data for each equation in a separate file called x_i.

9. In Matlab, for each file x_i, load the file and extract each column of data in a separate matrix. Use function `regress` to calculate the coefficients. Repeate this for every file. At the end you will have all the coefficients that are required for matrix **A** and **B**.

10. Construct matrices **Q** and **R** as described in this chapter. Remember to put more weights for state variables that are of more importance in matrix **Q**.

11. Use matrices **A**, **B**, **Q** and **R** in Matlab function `dlqr` to calculate matrix **K** which is the controller gains.

3.4 Summary

This chapter covered problem definition and system description. A normal use case for the system is studied and some important components' responsibility of cloud system is investigated. Based on the description a number of metric is selected for system identification purpose.

The system identification methodology is discussed in details. All required steps for this process is explained and the theory behind each one is provided. Furthermore different architecture for controller design is briefly introduced. Based on provided information in this chapter, one has enough knowledge to collect experimental data, identify a system and design a controller for such system.

Finally a general method for designing controller for elastic storages is given that covers all the necessary steps from System identification to controller design based on state-space model.

Chapter 4

Simulation Framework Implementation

4.1 Introduction

We have selected Kompics as the implementation tool. Kompics [5] is a message-passing component model for building distributed systems using event-driven programming. Kompics components are reactive state machines that execute concurrently and communicate by passing data-carrying typed events through typed bidirectional ports connected by channels. For further information please refer to Kompics programming manual and tutorial available on its web site.

4.2 Implementation

In this section we will briefly explain the implemented simulator by Kompics. Implementation is done in Java and Scala languages [6]. We first give the component architecture diagram for each component in the system and then each sub-component and their communication ports. Each port is indicated either by + (`provides`) or − (`requires`) which the former means that the component provides a service to other components and the latter means that the component is dependant on the service of other components. An overall simulation system is shown in Fig. 4.1

4.2.1 Cloud Instance Component

Cloud instance component represents an entire storage instance within a cloud. The component architecture for Instance is demonstrated in Fig. 4.2.

This component consists of several sub-components which will be discussed in the following.

OS Component

The main component within cloud instance is the OS (Operating System). It is depicted in Fig 4.3.
Responsibilities:

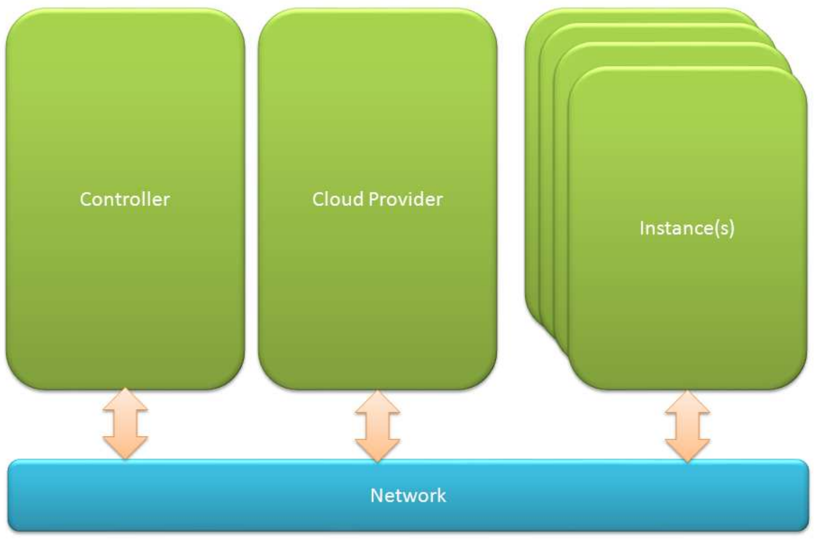


Figure 4.1. Overall System Architecture

- Handle incoming request for downloads and put them into `requestQueue` and process them.
- Manage process table
- Execute different operations on CPU component.
- Read/Write data blocks from/to Memory component.
- Read/Write data blocks from/to Disk component.
- Handle signals received from cloud provider and its sub-components

provides: `OS Port` which is a service to provide info for other components that are dependant on OS component
requires: communicate with CPU component by `CPUChannel`, with Memory component by `MemoryChannel`, with Disk component by `DiskChannel`, with other components in the system by `Network` and use `Timer` to schedule tasks.

Event handlers:

- subscribed to Control port
  `initHandler`: is responsible for initializing the component related variables

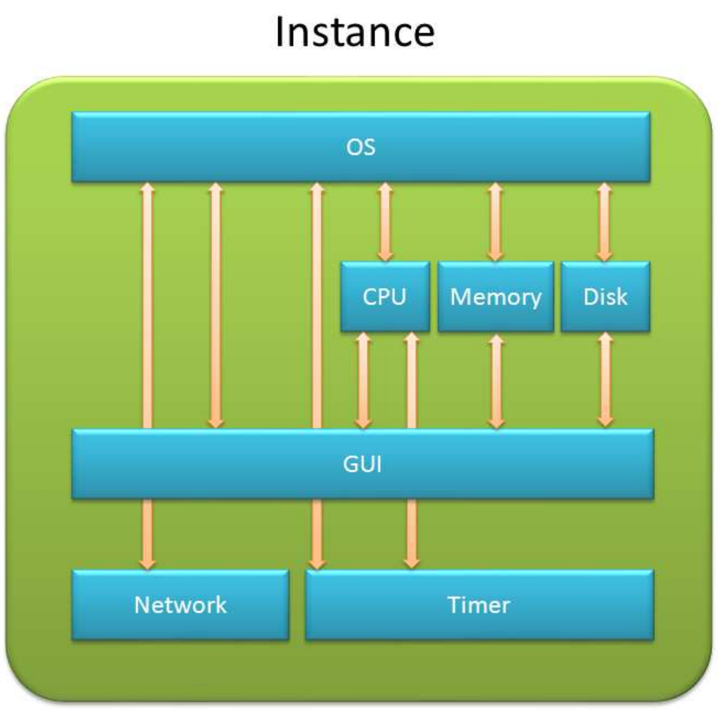


Figure 4.2. Cloud Instance Component Architecture

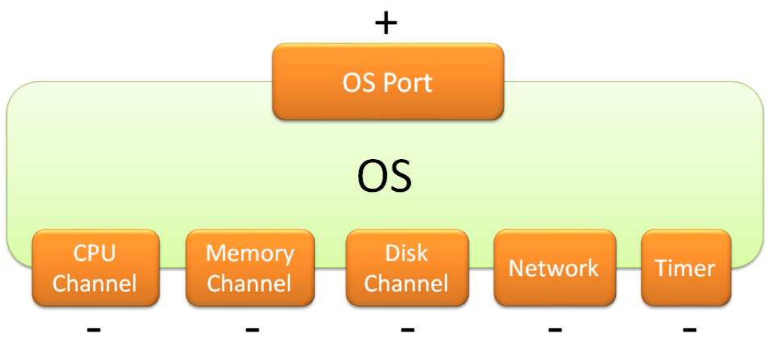


Figure 4.3. OS Component Architecture

- subscribed to CPU channel:
  `cpuReadySignalHandler`: executed when the CPU sends the `READY` signal
  `cpuLoadHandler`: executed when the CPU sends its load to OS

`snapshotRequestHandler`: executed when the user issues a snapshot (**??**) command

- subscribed to Memory channel:
  `memoryReadySignalHandler`: executed when Memory sends `READY` signal
  `ackBlockHandler`: executed when Memory acknowledges that it has the requested data block in the memory
  `nackBlockHandler`: executed when Memory acknowledges that it does not have the requested data block in the memory

- subscribed to Disk channel:
  `diskReadySignalHandler`: executed when Disk sends `READY` signal
  `blockResponseHandler`: a disk read operation (read from Disk and write into Memory

- subscribed to Timer channel:
  `processRequestQueueHandler`: triggered periodically to process incoming request queue
  `propagateCPULoadHandler`: periodically sends CPU load to Cloud Provider component
  `transferringFinishedHandler`: updates transfer table and computes the new bandwidth for remaining transfers
  `readDiskFinishedHandler`: executed when the read disk operation is done and the data block is in the Memory so the transfer can be started in the network
  `waitTimeoutHandler`: is executed if the system has not yet started until it starts successfully with all the hardware components
  `deathHandler`: executed to start a complete shut down of the running instance
  `calculateCostHandler`: is simple time out event handler for computing the cost for the current instance according to Amazon EC2 and S3 pricing

- subscribed to Network channel:
  `requestMessageHandler`: is in charge of handling the request that are sent by Elastic Load Balancer
  `heartbeatMessageHandler`: triggered when receives a heatBeat message from HealthChecker
  `monitorHandler`: triggered by Sensor component in Elastic Controller component
  `shutDownHandler`: triggered upon receiving a shutDown request from Cloud Provider component
  `restartInstanceHandler`: triggered when Cloud Provider sends a `RESTART` signal
  `rebalanceRequestHandler`: triggered when Cloud Provider suggest the name of blocks and its provider so the instance can send request to another instance asking for the recommended blocks

`rebalanceResponseHandler`: triggered when the other instance is responding with the requested block hence the download will start
`blockTransferredHandler`: triggered when the rebalancing of data block is finished from the other instance `closeMyStreamHandler`: triggered by dying instance to clean up open streams that can block accepting further requests

CPU Component

CPU Component represents the CPU unit within an instance in the cloud. It is shown in Fig. 4.4.

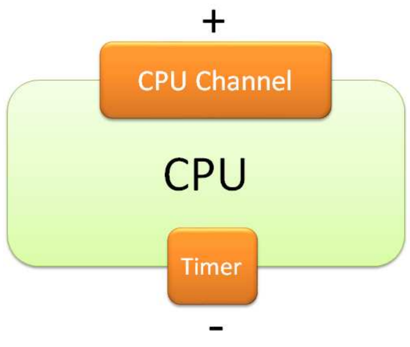


Figure 4.4. CPU Component Architecture

Responsibilities:

- Execute operations
- Compute load every 5 seconds
- Handle signals from OS component

provides: `CPUChannel` is used to communicate with the OS component
requires: `Timer` is used to schedule tasks.

Event handlers:

- subscribed to Control port
  `initHandler`: is responsible for initializing the component related variables
- subscribed to CPU channel:
  `startProcessHandler`: triggered when a new process starts in the OS
  `endProcessHandler`: triggered when a process ends in the OS
  `abstractOperationHandler`: triggered when a new operation is started on the CPU

`snapshotRequestHandler`: triggered when a snapshot is issued by the user from OS
`restartSignalHandler`: triggered when a `RESTART` signal is received from OS

- subscribed to Timer channel:
  `loadCalculationTimeoutHandler`: periodically executed and computes the load average
  `loadSamplerTimeoutHandler`: stores the current load of CPU
  `operationFinishedTimeoutHandler`: removes the corresponding operation from operation queue
  `restartHandler`: restarts the CPU

Memory Component

Memory component represents the Memory unit of an instance in the cloud. This component is depicted in Fig. 4.5.

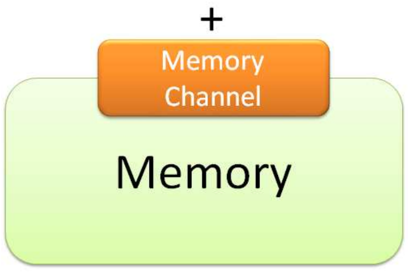


Figure 4.5. Memory Component Architecture

Responsibilities:

- Respond to read/write requests that are sent from OS component
- Manage access to current data blocks that are located in the memory
- Implements LFU[1] algorithm to manage data block eviction/admission

provides: `MemoryChannel` is used to communicate with the OS component

Event handlers:

- subscribed to Control port
  `initHandler`: is responsible for initializing the component related variables

[1]Least Frequently Used

- subscribed to Memory channel:
  `requestBlockHandler`: is responsible for checking to see if the Memory has the requested data block
  `writeMemoryHandler`: executed when a write into Memory command is issued by OS
  `restartSignalHandler`: restarts the Memory
  `startMemoryUnit`: starts the Memory

Disk Component

Disk component represents a physical disk within an instance in the cloud. The component architecture is shown in Fig. 4.6.

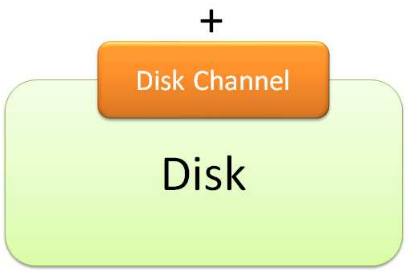


Figure 4.6. Disk Component Architecture

Responsibilities:

- Respond to read/write requests that are sent from OS component

- Manage access to current data blocks that are persisted in the disk

provides: `DiskChannel` is used to communicate with the OS component

Event handlers:

- subscribed to Control port
  `initHandler`: is responsible for initializing the component related variables

- subscribed to Disk channel:
  `loadHandler`: loads the Disk with initial data blocks
  `requestBlockHandler`: retrieves the requested data block from disk
  `startDiskUnitHandler`: starts Disk component
  `restartSignalHandler`: restarts Disk component

4.2.2 Cloud Provider Component

Cloud provider component represents an important unit in the implementation. It is the heart of a cloud computing infrastructure and provides vital services to manage and administer the nodes within the cloud. The overall component architecture is demonstrated in Fig. 4.7.

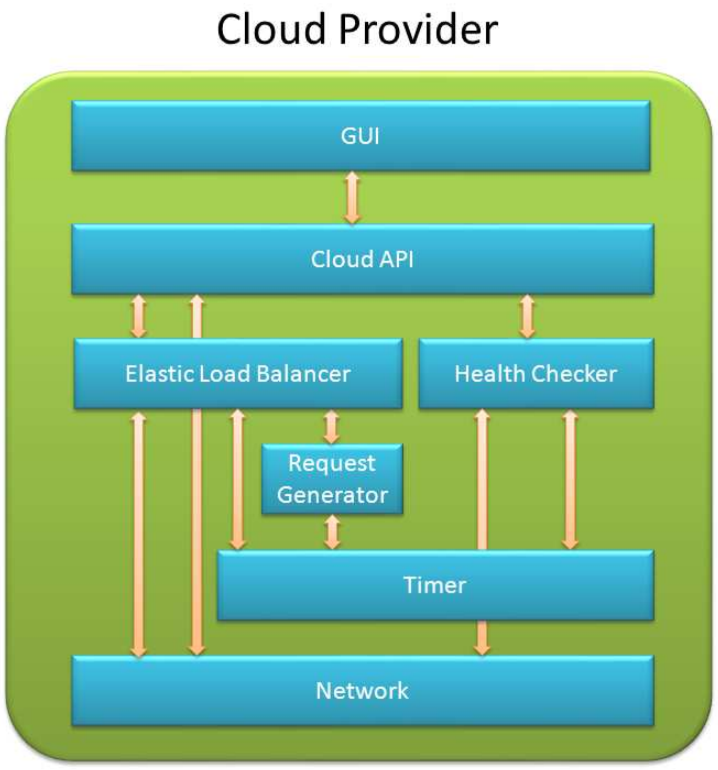


Figure 4.7. Cloud Provider Component Architecture

This component consists of several sub-components which each one will be discussed in the following.

Cloud API Component

Cloud API component implements all the functionalities that can be accessed by outside boundary of the cloud. In other words, it acts as a public API. The component architecture is shown in Fig. 4.8.
Responsibilities:

- Launch/shut down of storage instances
- Provide the rebalancing of data blocks between instances

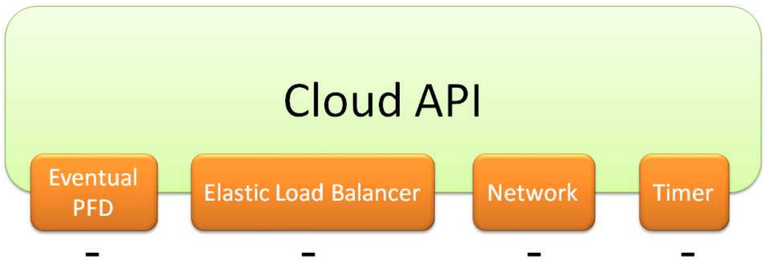


Figure 4.8. Cloud API Component Architecture

- Manage elastic IP addresses for storage nodes
- Collect and calculate the total cost for the running instances
- Respond back to sense and training data requests
- Show an overview of the running instances

requires: `EventualPFD` is used to communicate with Health Checker component, `ElasticLoadBalancer` is used to communicate with Elastic Load Balancer component, `Network` is used to communicate with other components in the system, `Timer` is used to schedule tasks.

Event handlers:

- subscribed to Control port
  `initHandler`: is responsible for initializing the component related variables

- subscribed to EPFD channel:
  `suspectHandler`: executed when receives a suspect signal from EPFD and informs ELB about the suspected node
  `restoreHandler`: executed when receives a restore signal from EPFD and informs ELB about the restored node

- subscribed to ELB channel:
  `replicasHandler`: triggered by ELB and sends back a set of blocks that the new joined node should start with
  `rebalanceResponseHandler`: triggered when ELB provides the Cloud API the map of data blocks that the newly joined node can retrieve data blocks from
  `nodesToRemoveHandler`: is triggered when the ELB returns the nodes to remove so CloudAPI can remove them

- subscribed to Network channel:
  `instanceStartedHandler`: executed when the requested instance is initialized
  `connectControllerHandler`: triggered when the controller tries to connect to cloud API
  `disconnectHandler`: triggered when the controller disconnects from the cloud API
  `newNodeRequestHandler`: triggered when the controller requests a new node
  `instanceCostHandler`: triggered when an instance sends its cost to cloudProvider for presentation purposes
  `requestTrainingDataHandler`: is triggered when the modeler requests training data from cloudAPI. It distributes this request to corresponding components
  `requestSensingData`: This handler is triggered when the sensor request data to sense. This data will be used by the controller to act accordingly
  `removeNodeHandler`: is triggered when the modeler from controller requests to remove a node

Health Checker Component

Health Checker component is simply an implementation of Eventual Perfect Failure Detector (EPFD) algorithm that is introduced in [25] with a minor modification. The difference is to consider the health of a changing group of instances that are under the control of cloud API component and informs other components within Cloud Provider about any suspicious on the health of nodes. The component architecture is depicted in Fig. 4.9.

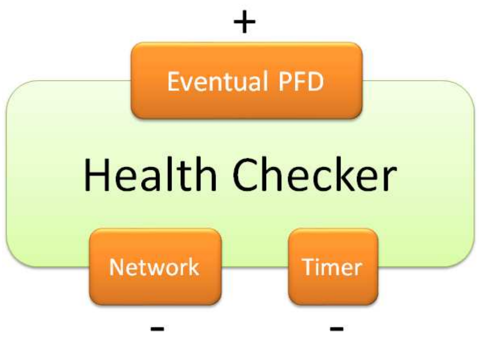


Figure 4.9. Health Checker Component Architecture

Responsibilities:

- Send out heart beat messages to every instance in the system

- Decide about the health of an instance and inform other components

provides: `EventualPDF` is used to communicate with Cloud API component.
requires: `Network` is used to communicate with other components in the system, `Timer` is used to schedule tasks.

Event handlers:

- subscribed to EPFD channel:
  `initHandler`: is responsible for initializing the component related variables
  `considerInstanceHandler`: triggered when a new node is started and cloud Provider discovers its existence
  `instanceKilledHandler`: triggered when one instance is killed or shut down so the EPFD would not consider checking its health
- subscribed to Timer channel:
  `heartbeatTimeoutHandler`: triggered periodically to check the health of all available instances and sends a Heatbeat message
- subscribed to Network channel:
  `aliveHandler`:triggered when a node responds back to a previously sent Heat-beat message

Elastic Load Balancer Component

Elastic Load Balancer component represents a load balancer that can route incoming traffic to different instances in the system. In addition to classic Load Balancer, the elastic implementation consider the joins and leaves of instances in the cloud. The component architecture is demonstrated in Fig. 4.10.

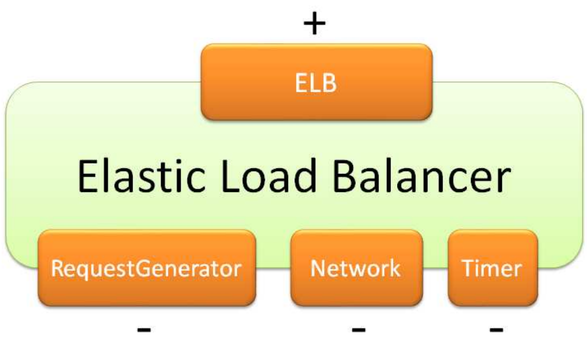


Figure 4.10. Elastic Load Balancer Component Architecture

Elastic Load Balancer implements least CPU Load algorithm and in addition to that it captures the number of request that it has sent to a particular instance.

The priority on routing the incoming request is first the least CPU load and then the least number of previously sent requests.

Responsibilities:

- Prepare replicas of data blocks for each instance
- Mark a particular data block as unavailable if the provider instance becomes unavailable and no other instance in the system provides that replica
- Monitor the CPU Load of instances
- Add/Remove replica
- Show a tree representation of current replicas in the GUI
- Provide replica map for newly joined instance thus the instance can start downloading data blocks from existing instances
- Help Cloud API to decide which instance should be removed

provides: `ELB` is used to communicate with Cloud API component.
requires: `RequestGenerator` is used to communicate with Request Generator component, `Network` is used to communicate with other components in the system.

Event handlers:

- subscribed to ELB Channel:
  `initHandler`: is responsible for initializing the component related variables
  `getReplicasHandler`: triggered by CloudAPI upon launching a new instance and calculate what data blocks it should have
  `suspectNodeHandler`: triggered when receives a suspect signal from EPFD so it marks the corresponding data blocks' replica as suspected
  `restoreNodeHandler`: triggered when receives a restore signal from EPFD so it marks the corresponding data blocks' replica as restored
  `removeReplicaHandler`: triggered when a node shuts down so its replicas become unavailable
  `rebalanceDataBlocksHandler`: triggered when Cloud API requests data blocks map according to least CPU load so the new instance can retrieve each data block from an existing instance
  `sendRawDataHandler`: triggered when ELB receives a request from cloud-Provider to send the training data to controller
  `selectNodesToRemoveHandler`: This handler is triggered when the cloudAPI ask for some nodes so he can remove. ELB selects the nodes with the least CPU load according to ELB algorithm that is running.

- subscribed to Generator channel:
  `requestHandler`: triggered by Request Generator and it is responsible for sending request to the chosen node with respect to LoadBalancerAlgorithm
  `responseTimeHandler`: triggered when RequestGenerator provides the average response time

- subscribed to Network channel:
  `blocksAckHandler`: triggered when the newly requested instance starts up completely and acknowledges back its start up so its data blocks' replica become available
  `requestDoneHandler`: triggered when a transfer starts
  `rejectedResponseHandler`: triggered when a transfer is rejected
  `myCPULoadHandler`: receives the CPU load from an instance and updates the `nodeStatistics` in `LoadBalancerAlgorithm`
  `activateBlockHandler`: triggered when the newly joined instance informs about a block that is available and can be served from that instance

- subscribed to Timer channel:
  `elbTreeUpdateHandler`: triggered when timer times out and it should update the ELB tree in the GUI

Request Generator Component

Request Generator component represents traffic on behalf of end users of the cloud. It generates request for different data blocks. The component diagram is shown in Fig. 4.11.

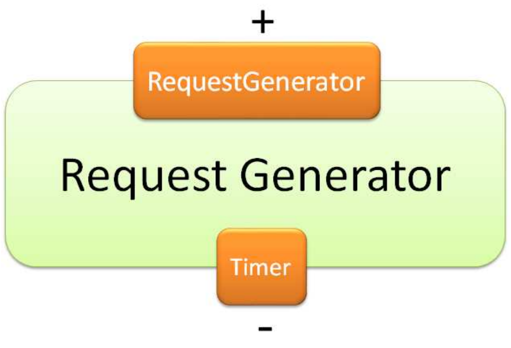


Figure 4.11. Request Generator Component Architecture

Responsibilities:

- Generate traffic to the storage instances based on different defined distribution.

- Collect responses from instances and generate response time scatter plot in the GUI.

provides: `RequestGenerator` is used to communicate with Elastic Load Balancer component.
requires: `Timer` is used to schedule different tasks (distribution).

Event handlers:

- subscribed to Generator Channel:
  `initHandler`: is responsible for initializing the component related variables
  `requestDoneHandler`: triggered when a transfer finishes and calculates the response time for further presentation
  `sendRawDataHandler`: triggered when cloudAPI request the average response time
  `blocksActivatedHandler`: triggered when the engine receives blocks that are activated from ELB
  `downloadRejectedHandler`: triggered when the engine receives a rejection for the requested download

- subscribed to Timer channel:
  `requestEngineTimeout`: triggered according to the current distribution and prepares and sends requests for all data blocks
  `RTCollectionTimeoutHandler`: triggered periodically to draw response time scatter plot in the GUI

4.2.3 Elastic Controller

Elastic controller represents the controller that can connect to cloud provider and retrieve information about the current nodes in the system. The main responsibility for controller component is to manage the number of nodes currently running in the cloud. In other words, it attempts to optimize the cost and satisfy some SLA parameters. The overall component architecture is demonstrated in Fig. 4.12.

Elastic controller component consists of several sub-components which each one will be discussed in the following.

Controller Component

Controller is the main component within Elastic Controller. Generally it controls the modeler, sensor and actuator components that are described in the upcoming sections. The component diagram is shown in Fig 4.13.
Responsibilities:

- Connect to cloud provider component and retrieve information about running nodes

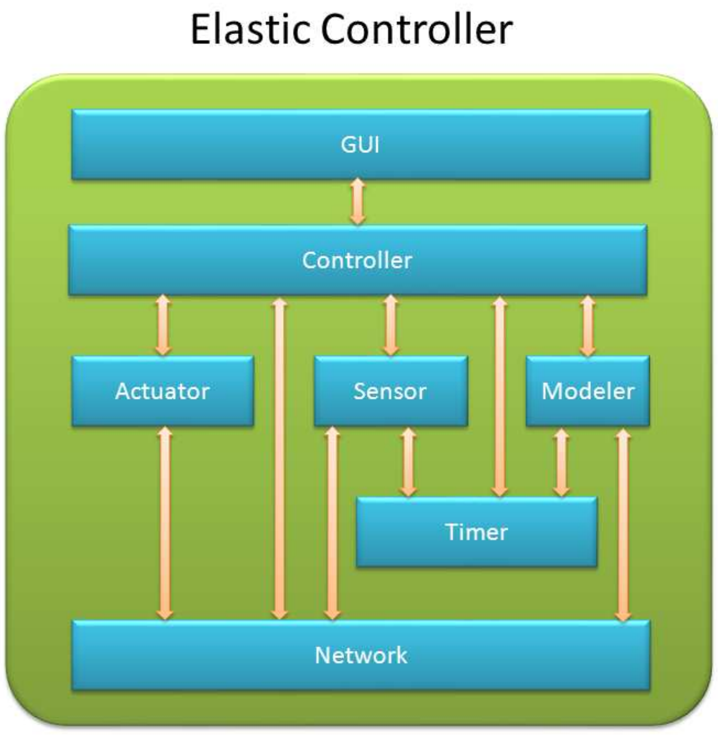


Figure 4.12. Elastic Controller Component Architecture

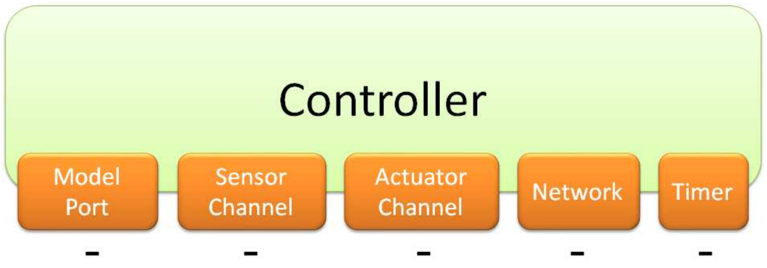


Figure 4.13. Controller Component Architecture

- Can switch between different controller implementations
- Run the current selected controller implementation
- Start/Stop sensor, actuator and modeler

requires: `ModelPort` is used to communicate with the Modeler component, `SensorChannel` is used to communicate with the Sensor component, `ActuatorChannel` is used to communicate with the Actuator channel, `Network` is used to communicate with cloud provider as well as instances in the system and `Timer` is used to schedule tasks

Event handlers:

- subscribed to Control channel:
  `initHandler`: is responsible for initializing the component related variables
- subscribed to Sensor channel:
  `monitorResponseHandler`: is triggered when it receives monitor information from sensor component
- subscribed to Timer component:
  `connectionTimeoutHandler`: is triggered to check if the controller is connected to cloud provider or not
  `actuateTimeoutHandler`: is triggered periodically to initiate the controller action
- subscribed to Network component:
  `connectionEstablishedHandler`: is triggered when the cloudProvider responds back to the connection establishment request

Sensor Component

Sensor component acts as a sensor to sense the environment (so the name). The component diagram is depicted in Fig. 4.14.

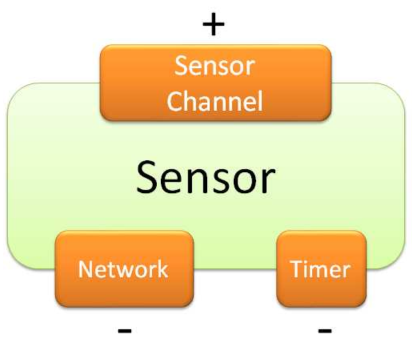


Figure 4.14. Sensor Component Architecture

Responsibilities:

- Start/Stop sensing the cloud environment and collect monitoring packets from each instance

provides: `SensorChannel` is used to communicate the monitoring packets to other components like Controller
requires: `Network` is used to communicate with running instances in the system and `Timer` is used to schedule tasks

Event handlers:

- subscribed to Control channel:
  `initHandler`: is responsible for initializing the component related variables
- subscribed to Sensor channel:
  `senseHandler`: is triggered when controller sends a set of instances that Sensor should sense
  `startSenseHandler`: is triggered when controller issues a START signal with a frequency of sensing
  `stopSenseHandler`: is triggered when controller issues a STOP signal to the Sensor
- subscribed to Timer component:
  `senseTimeout`: is triggered to send out monitoring message and schedule the next sensing
- subscribed to Network component:
  `monitorResponseHandler`: is triggered when the sensor receives monitor response from an instance
  `newNodeToMonitorHandler`: is triggered when a new instance joins the cloud environment

Actuator Component

Actuator component is responsible for ordering a new instance or shuting down a current instance to cloud provider. The component diagram is depicted in Fig. 4.15.
Responsibilities:

- Request to launch new instance to cloud provider
- Request to shut down an instance to cloud provider

provides: `ActuatorChannel` is used to communicate with the controller
requires: `Network` is used to communicate with running instances in the system

Event handlers:

- subscribed to Control channel:
  `initHandler`: is responsible for initializing the component related variables

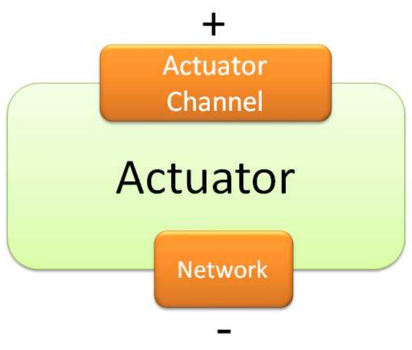


Figure 4.15. Actuator Component Architecture

- subscribed to Actuator channel:
  `nodeRequestHandler`: is triggered when it receives a signal from controller in order to request a new node

Modeler Component

Modeler component acts as the main component in order to identify the system. It collects the training data from cloud provider. The component diagram is demonstrated in Fig. 4.16.

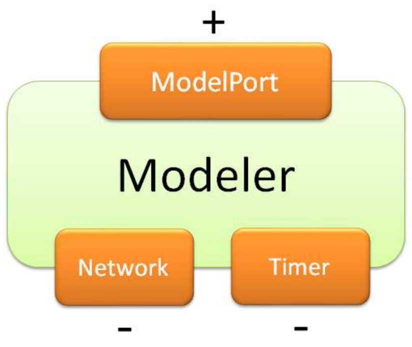


Figure 4.16. Modeler Component Architecture

Responsibilities:

- Request training data to cloud provider
- Collect and prepare training data for system identification purpose

- Plot diagrams based on received training data

provides: `ModelPort` is used to communicate with the controller
requires: `Network` is used to communicate with running instances in the system and `Timer` is used to schedule tasks

Event handlers:

- subscribed to Control channel:
  `initHandler`: is responsible for initializing the component related variables
- subscribed to ModelPort:
  `startModelerHandler`: is triggered when the Controller issues a START signal to Modeler
  `senseDataHandler`: is triggered only when the controller runs and it enables monitoring
- subscribed to Timer:
  `sampleTraingingDataHandler`: is responsible for requesting training data from cloudProvider
  `instanceCreationHandler`: is responsible for adding and removing instances so the modeler can have a range for the system inputs
- subscribed to Network:
  `trainingDataHanler`: is responsible for sorting out the response (raw data tuples) it receives from cloudProvider

4.3 Summary

This chapter covered the simulation framework that is implemented to simulate a cloud environment. All the implemented components are discussed and all responsibilities for a component is explained. The overall architecture is depicted and how these components are connected together to build the simulation framework is also investigated. Some components like EPFD implements a well known algorithm that the algorithm itself is skipped and is rather referenced for the reader.

Chapter 5

Experiment

In this chapter we first identify the system based on the description that is covered in Chapter 3. We then show how to configure and calculate the parameters for controller to automate the control of elastic storages in cloud using the simulation framework described in Chapter 4.

5.1 System Identification

A normal case within a cloud environment that provides storage services is to have at least two storage instances that one is acting as the fail-over node. In order to cover an acceptable range for system identification we start by 2 fresh and ready instances. By fresh we mean that the caches are empty and none of data blocks exists in the memory.

Using the simulator modeler component, we scale up the number of nodes from 2 to 10 and then scale down from 10 to 2. This is done one time. Sampling of training data is performed every 10 seconds and every 225 seconds a new node is either added or removed (depending on whether we scale up or down).

Using the request generator component, we schedule requests to load balancer component in the cloud provider component from `sin` distribution between $[1, 10]$ seconds. So the first request be sent after 10 seconds, the second after 9 seconds, ... After each scaling up/down the system will experience 2 `sin` loads of request between 1 to 10 seconds. The time needed to experience 2 `sin` is $2\sum_{i=1}^{10} i$ which is 220 seconds. That is the reason why we have selected 225 seconds as the action time.

The summary of properties that is used during system identification is as following:

- Distribution: `sin` distribution between 1 and 10 seconds.
- Sampling: every 10 seconds

- Action: every 225 seconds a node will be added or removed
- Number of nodes: one period between 2 to 10 nodes.

Figure 5.1 to 5.3 is the result of this system identification.

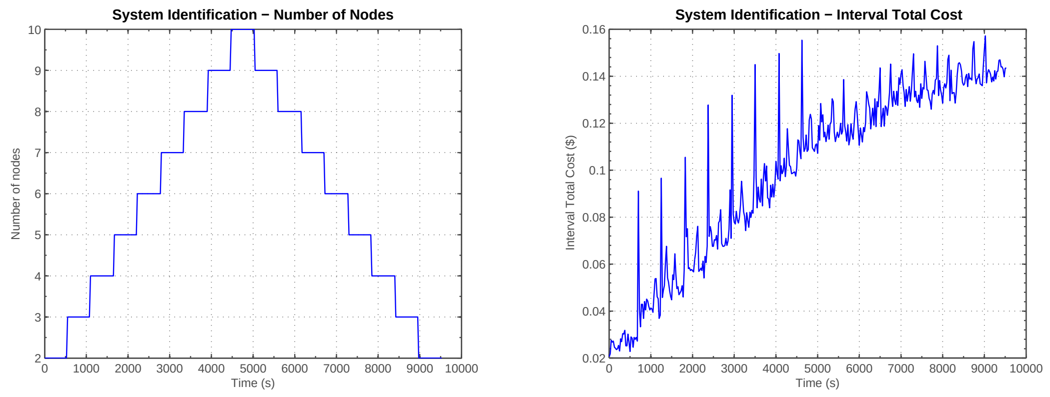


Figure 5.1. Number of Instances & Total Cost ($)

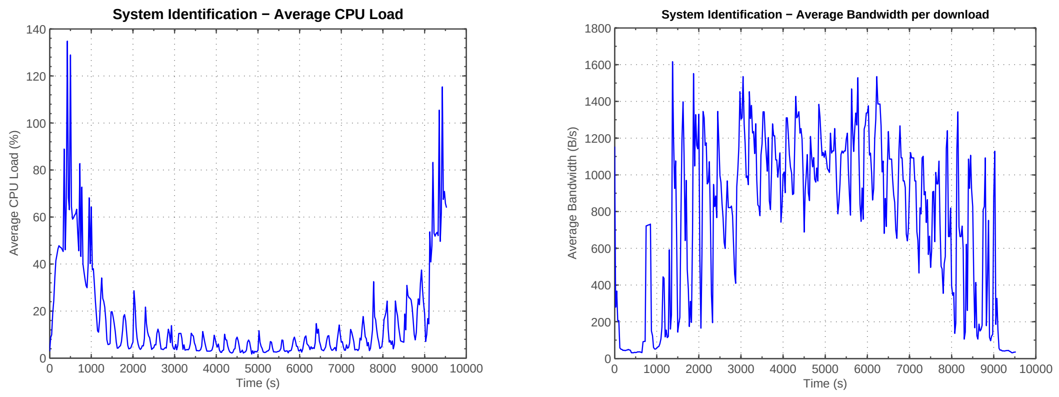


Figure 5.2. Average CPU Load & Average Bandwidth per download

The modeler component generates 4 files that contain the internal states for equations 3.6 to 3.6. Using Matlab `regress` function, the coefficient matrices are calculated as follows:

$$A = \begin{bmatrix} 1.02 & 0 & 0 \\ 0 & 0.724682978432349 & 0 \\ 5.927 & 0 & 0.295092571282931 \end{bmatrix} \tag{5.1}$$

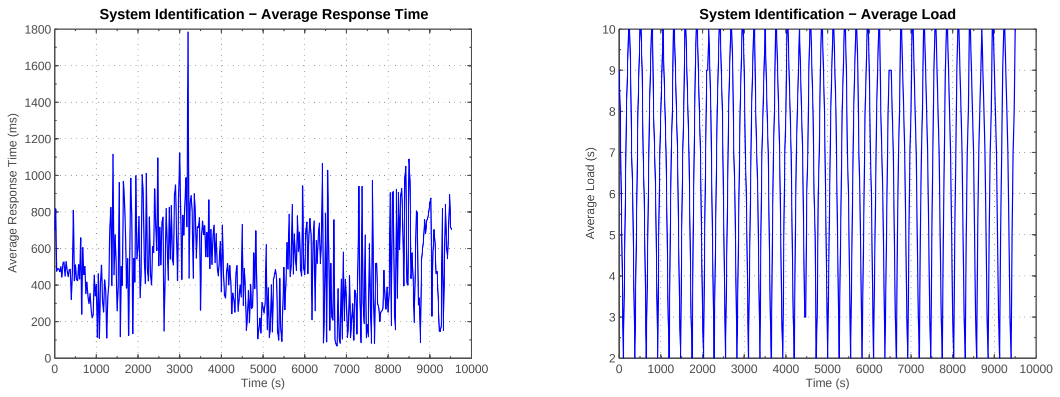


Figure 5.3. Average Response Time & Average Load

$$B = \begin{bmatrix} 2.30038938006101 \\ 0.0147684756564848 \\ 77.8759559716798 \end{bmatrix} \tag{5.2}$$

Having these matrices, we use `ss` function from Matlab to compute the equations 3.12 and 3.13.

```
>> C = [1 0 0 ; 0 1 0 ; 0 0 1];
>> D = [0;0;0];
>> sys = ss(A,B,C,D,-1);
```

The -1 is specified as the last argument for `ss` function to indicate that the modeling is in discrete time model.
It is interesting to know if the system is controllable or not. Using equation 3.20, we construct the matrix $\mathcal{C}$ like

$$\mathcal{C} = \begin{bmatrix} \mathtt{A}^3\mathtt{B} & \mathtt{A}^2\mathtt{B} & \mathtt{AB} & \mathtt{B} \end{bmatrix}$$

and we use Matlab to calculate it:

```
 >> C = [A*A*B A*B B]

C =

    2.3945    2.3470    2.3004
    0.0078    0.0107    0.0148
   24.7156   36.6153   77.8760

>> det(C)
```

```
ans =

    0.1848
```

Since the determinant of matrix $\mathcal{C}$ is non-zero then the system is controllable. The settling time and stability of a state-space system is determined by its poles and the poles of a state-space model are the eigenvalues of $\mathbf{A}$.

```
>> eig(A)

ans =

    0.2951
    1.0202
    0.7247
```

According to Section 3.3.4 since all the poles are in the unit circle the system is stable.

5.1.1 Preprocessing of collected data

Data preprocessing is one of the most important part in system identification. There are various techniques to preprocess the data. Removing means is one of the preprocessing technique that is used widely to detrend the data. It is recommended to remove mean values at least before the estimation phase.

5.1.2 Model Evaluation

So far we have calculated the matrices and before continuing to controller gains estimation, we evaluate the model. As described in Chapter 3, residual analysis can be employed to get a better view of the designed model. In order to do residual analysis, a scatter plot can be drawn showing actual versus predicted data. We design a simple scenario in which number of instances raises from 4 to 5 and then back to 4. This technique is called *one-step prediction*. At each sampling time k the actual $y(k)$ is collected and based on the sampling at time $k-1$, the predicted $y(k)$ is calculated. 630 data points have been collected for `CPU` and the result is demonstrated in Fig. **??**.

As can be seen from the scatter plot, most of the data points are very close to unit slope and this mean that the constructed model is suitable enough.

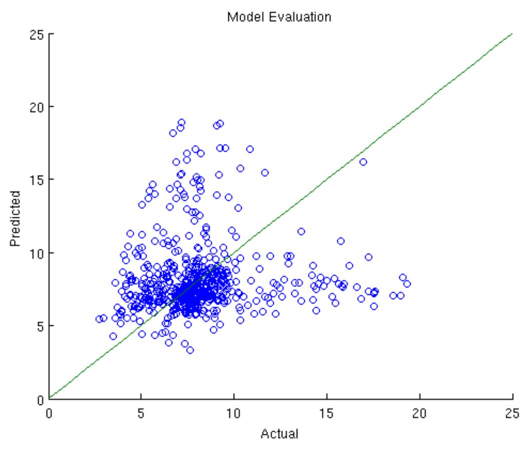


Figure 5.4. Model Evaluation - one step prediction for Average CPU Load

5.1.3 Estimating Controller gains

Now the data is ready to be used for estimation. It is the time to compute the feedback gain K. First we need to select matrices Q and R as described earlier:

```
 >> Q = diag([100 1 1 1])

Q =

   100      0      0
     0      1      0
     0      0      1

 >> R = 1

R =

     1
```

We have given 100 to the element that is corresponding to CPU Load. In this way we emphasize that this state variable is of more importance for us comparing to others. Now we can exploit `dlqr` function from Matlab to compute the K:

```
 >> K = dlqr(A, B, Q, R)

K =
     0.134         1.470162e-06         0.00318
```

5.2 Controller Design

These `K` parameters now can be used in the controller to compute the next number of nodes for the system simply by multiplying it by state variables matrix. The result is `NN`.

5.2.1 Experimenting with Controller

A `DynamicStateFeedbackController` is written with the calculated parameters. At each time the controller is asked for the next `NN` value. It would be interesting to know how to interpret this value. The controller output is not a natural number but a real number. This number should be rounded to a natural integer. It can be rounded up or down based on the design issue. We select to round down to save total cost the cloud generates. One can assume two boundaries that we address in the following:

- L (Lower boundary): minimum number of instances that should exist in the cloud at all times.
- U (Upper boundary): maximum number of instances that is allowed to exist in the cloud at all times.

Hence if even the controller outputs a value that is either smaller than L or greater than U, that value should be disregarded. If the output of controller is Θ The possible cases are as the following:

$$\mathtt{NN} = \begin{cases} L & \text{if} \quad \Theta \leqslant L \\ \Theta & \text{if} \quad L < \Theta < U \\ U & \text{if} \quad U \leqslant \Theta \end{cases} \tag{5.3}$$

if the number of current nodes in the system is $\mathtt{NN}'$ then:

$$\text{Next action} = \begin{cases} \text{scale up with } \mathtt{NN} - \mathtt{NN}' \text{ nodes} & \text{if} \quad \mathtt{NN}' < \mathtt{NN} \\ \text{scale down with } \mathtt{NN}' - \mathtt{NN} \text{ nodes} & \text{if} \quad \mathtt{NN} < \mathtt{NN}' \\ \text{no action} & \text{otherwise} \end{cases} \tag{5.4}$$

In the rest of this chapter we investigate two experiments that each one will examine an aspect of the provided controller. The instance configuration for these experiments are as the following:

- CPU: 2 GHz
- Memory: 8 GB
- Bandwidth: 2 MB/s
- Number of simultaneous downloads: 70

They are 10 data blocks in the experiments with sizes between 1 to 5 MB. It should be noted that this configuration is used through the system identification also.

SLA Experiment

In order to get a better sense that how the controller operates, a simple scenario can be conducted. One of the common cases for cloud environment is to test against specific Service Level Agreements (SLA). SLA is defined as the minimum criteria a provider promises to meet while delivering a service.

Two experiments can be considered: one with controller and the other without controller. In the coming results and figures, they are denoted by `w/ controller` and `w/o controller` respectively. Each experiment starts with tree warmed up instances. With warmed up nodes we mean that each data block is requested at least once thus it resides in the memory all instances.

Workload that is used for this experiment consists of two states: normal and high. It is started with normal state which the timeout for sending request is selected from a uniform random distribution between [10, 15] seconds. After 500 seconds at this state, the workload changes to high state. In this state the timeout is selected from the uniform random distribution between [1, 5] seconds. This is demonstrated in Fig 5.5.

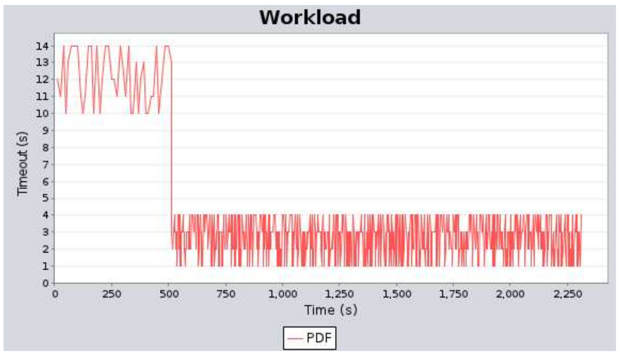


Figure 5.5. Experiment1 Workload

Sensing of data is done every 25 seconds. In the case of controller, actuation is performed every 100 seconds. Thus there are 4 sets of data at each actuation time that controller should consider. In order to decide what the state variables are, average of data sets is calculated and used by the controller. The duration of each experiment is 2000 seconds with warm up of 100 seconds. SLA requirements are as the following:

- Average CPU Load: $\leqslant 55\%$

Table 5.1. SLA Violations

SLA Parameter Violation (%)	w/ Controller	w/o Controller
CPU Load	17.94	72.28
Response Time	2.12	7.073
Bandwidth	35.89	74.69

- Response Time: $\leqslant 1,5$ seconds
- Average Bandwidth per download: > 200000 B/s

For each experiment the percentage of SLA violations are calculated per parameter based on Equation 5.5. The result is shown in Table 5.1.

$$Percentage\ of\ SLA\ \ Violations = 100 \times \frac{Number\ of\ SLA\ \ Violations}{Total\ Number\ of\ SLA\ \ Checking} \tag{5.5}$$

SLA checking is done at each data sensing for *Average CPU Load* and *Average Bandwidth per download* and is performed at each request response for *Response Time*.

Results

This experiment gives us interesting results that is discussed in this section. `NL` and `HL` that exist in the result figures correspond to *Normal Load* and *High Load* respectively.

Figure 5.6 depicts Average CPU Load for the aforementioned experiments. Average CPU Load is the average of all nodes' CPU Loads at each time the sensing is performed. As it is observed from the diagram, CPU loads for experiment with controller is generally lower than the same experiment without controller. This is due to controller that launch new instances under high workloads causing a huge drop in average CPU Load.

Figure 5.7 demonstrates Average Response Time for the experiments. By response time we mean the time that an instance responds to a request that download is started afterward and not the actual download time. As it is seen from the diagram, average response time for experiment with controller is generally lower than the experiment without controller. This is because in case of having fixed number of instances (in this experiment 3), there would be congestion by the number of requests an instance can process. This increases the responsivity of an instance. However, in the case that controller launches new instances, no instance will actually go under high number of requests.

Figure 5.8 shows the Total cost for the experiments. Interval Total cost means that total cost is calculated in each interval in which the senses are done. As can

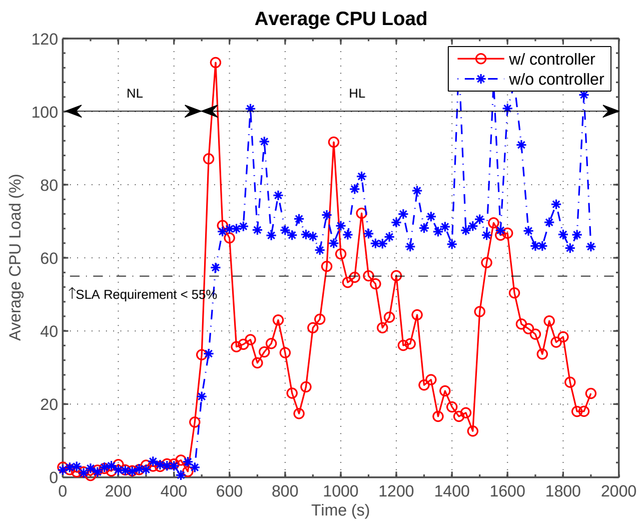


Figure 5.6. SLA Experiment - Average CPU Load

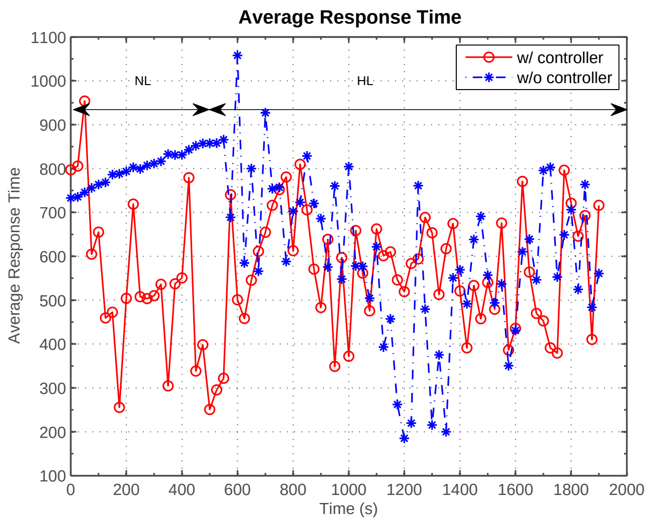


Figure 5.7. SLA Experiment - Average Response Time

be observed from the diagram the interval total cost for controller experiment is much higher than the experiment without controller. This is because launching new instances will cost more money than having fixed number of instances available in the cloud. This experiment has high load of requests for the system in which controller is more likely to scale up and resides in that mood than scale down. It should be noted that costs are computed according to Amazon EC2 price list.

Total cost for each experiment is calculated in Table 5.2.

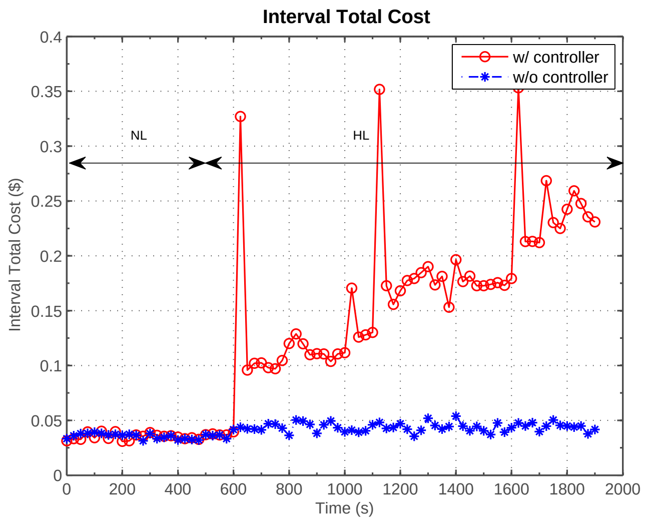


Figure 5.8. SLA Experiment - Interval Total Cost

	w/ controller	w/o controller
Total Cost ($)	14.4528	8.6779

Table 5.2. Total Cost for each SLA experiment

Figure 5.9 depicts Average bandwidth per download. If an instance has a bandwidth of 4 Mb/s and has two current downloads running, the bandwidth per download is 2 Mb/s. As can be seen from the diagram, experiment that uses controller has significant higher bandwidth per download. This is mainly because the instances receive less number of request and bandwidth is divided among less requests also. This will end up having higher bandwidth available on each instance.

Figure 5.10 demonstrates number of nodes for each experiment. As we discussed earlier the number of nodes is constant for experiment without controller. However, for the other experiment that the controller is used, number of nodes is changed through time hence the SLA requirements can be met.

Cost Experiment

The purpose of this experiment is to show that total cost is not always higher when using controller. For this experiment we start initially by 7 instances. Duration of the experiment is 2000 seconds.

For this experiment a different workload is used. The workload consists of two states again: high and low. Unlike previous workload, it starts in high state and then goes into low state after 500 seconds. The workload is shown in Fig. 5.11. Timeouts in high state is selected from uniform random distribution between [1, 3]

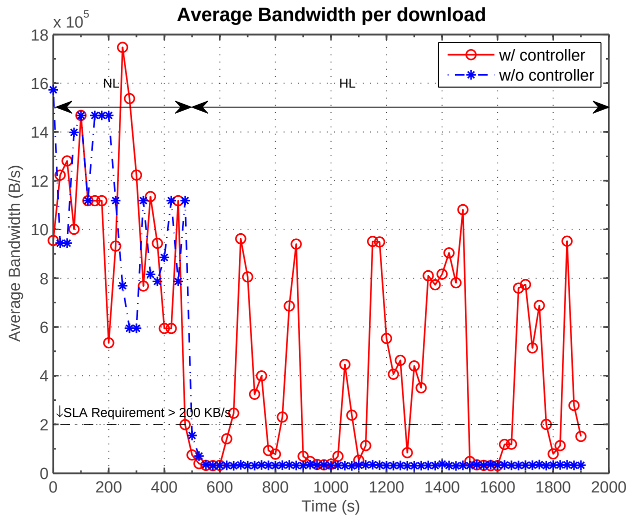


Figure 5.9. SLA Experiment - Average Bandwidth per download (B/s)

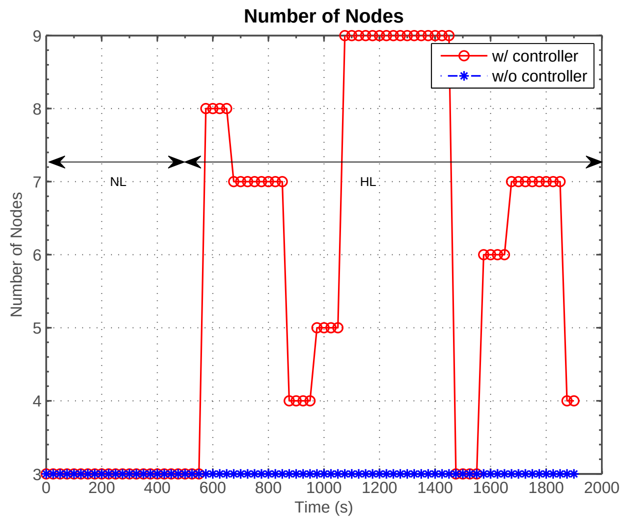


Figure 5.10. SLA Experiment - Number of Nodes

seconds and in low state between [15, 20] seconds.

The result that is shown in Table 5.3 is interesting since this time the total cost for the experiment with controller is actually lower than the experiment without controller unlike previous experiment. This because of low load in the system. Controller basically will remove instances that this will save cost in this experiment. The reason that this experiment has lower cost than the previous one is that L (lower bound on number of nodes) is not equal the initial number of nodes and it is smaller.

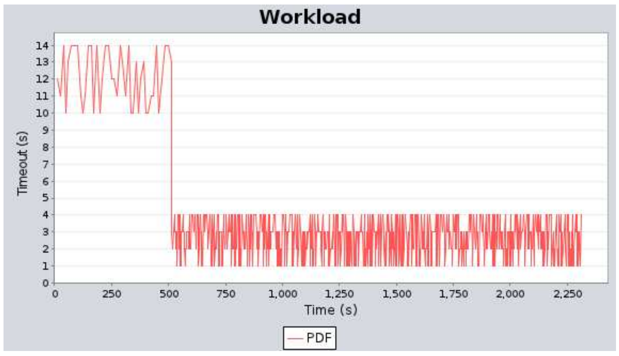


Figure 5.11. Experiment2 workload

	w/ controller	w/o controller
Total Cost ($)	10.509	16.5001

Table 5.3. Total Cost for Cost experiment

Hence controller can scale down in number of nodes to L.

5.3 Summary

A practical insight has been given in this chapter for how to perform System Identification and controller design using the implemented simulation framework. Two experiments are done: First an experiment against Service Level Agreement (SLA) in abscense and prescense of controller was conducted. SLA violations are calculated and the results are discussed. Second an experiment is done to compare total cost in the case that workload is transfered from high load to low load and the controller scales down in number of nodes in the cloud.

Chapter 6

Conclusion and Future Works

This chapter summarizes and concludes the thesis and presents future works. We start with concluding what has been done through this master thesis project. This chapter will be finalized with some ideas for extension and future work.

6.1 Summary and Conclusion for the thesis

For this section we review the defined objectives in the introduction chapter.

6.1.1 Study Distributed Data Storage (DSS) Systems

Six different Distributed Data Storage (DSS) systems are briefly studied in this project. Common approaches and mechanisms are discussed and a summary for all of them is provided so that one can compare these systems based on *DB Type*, *Architecture*, *Focus* and *Consistency*. The main purpose of this study was to get familiar with DSS systems and understand how a key-value store can operate in context of a cloud environment. Although all of these systems claim to provide elasticity, there is no information provided in their publications regarding the details of this elasticity.

6.1.2 Study the application of control theory in Computing Systems

This master thesis project is based on the idea of using control theory to provide automation for elasticity in cloud environments. An introduction was given in Chapter 2 which covered basic concepts in this field such as System Identification and controller design. We have studied 13 computing systems that based on control theory for different purposes (mostly performance). Common performance objectives are discussed also and a summary for all systems is provided at the end of chapter. This summary categorizes the studied systems based on *System Identification*, *Controller* and *Desired Objectives*. The purpose of this study was to understand the common approaches and strategies of employing controller theory in computing systems. We realized that *Black-box model* is widely used as one of the promising and approved way to identify systems since most of the computing systems are non-linear.

Two systems ([44] and [24]) are discussed briefly. We provided the overall controller architecture and how the system dynamics are constructed. This short description was mainly to introduce the reader with the concept of control theory when it applies to computing systems.

6.1.3 System Identification and appropriate controller design

The introduction to control theory is brought to next level by a discussion on the system and problem description. We defined the key-value store system that we aim to implement in a cloud environment in Chapter 3. The overall architecture of such a system is presented and common usage scenarios are given. This description then followed by a common proved model for system identification: State-space model. This model is explained in details and we showed how to apply this model to our desire system step by step. This explanation involves the extraction of relevant properties of the system and system dynamics construction also.

In the second half of the chapter, we discussed common controller architectures in State-space model and we provided a quick comparison between these architectures. We investigated how to do controller design step by step.

6.1.4 Method of designing controller for elastic storage in cloud

At the end of Chapter 3, a general method is given for designing a controller for storage cloud systems. The method consists of all the necessary steps for system identification and controller design. This method is the summary of this master project and it can be used for the purpose of identifying a system and designing a controller which totally provides automation for similar systems. it is generic enough to cover systems with arbitrary number of inputs and outputs.

6.1.5 Implementation of a Simulation Framework for Cloud

A complete simulation framework based on Kompics is implemented to model a key-value store in a cloud environment. Detailed architecture and implementation are discussed in Chapter 4. This implementation consists of several components that each is responsible for a set of specific tasks. Each component and its relation to other components is described in details. This framework enables the modeling of cloud environments targeting possible storage strategies in cloud. It is flexible enough to cover a wide range of services that are deployed in a cloud environment and model detailed interactions between instances and end-users. This framework is one the major contribution of this master project. A step-by-step guide for checking out, building, running and using of this framework is given in Appendix A.

6.1.6 Experiment with the controller with simulation framework

In Chapter 5 we showed how to do a system identification with the implemented simulation framework and then we walked through the steps for designing the controller based on the provided description earlier. This chapter consists two important experiments to prove the effect of using controllers in cloud environments. First experiment examines SLA violations in presence and absence of controller. We showed that using controller can reduce SLA violations when the workload for a system is transited from normal state to high state experiencing a flash crowd. Second experiment investigate Cost in presence and absence of controller. We showed that using controller can reduce total cost when the workload is transited from high to low state.

6.2 Future work

In this section we try to propose some ideas to the reader for further extension of this project.

Fuzzy Controller

One can replace the controllers introduced in this project (Dynamic State Feedback controller and Fuzzy controller) with a Fuzzy controller. This means that instead of solving the system dynamics by statistical approaches, we can use fuzzy logic to define rules and tune the controller to employ these rules for its decisions. This needs intelligent system identification and a thorough and detailed understanding of the system. The rules can increase in number and complexity though.

Constraint Programming

Through out our System identification, controller design and experiments, we used a fixed hardware configuration for instances. This means that all launched instances had the same hardware configurations. One possible extension is to exploit constraint programming in order to calculate the best hardware configuration for specific situation. One can define a number of spaces that are desirable and based on current state of the system provide a suitable hardware setup that for example will minimize the total cost or decrease response time. This can be implemented as a separate component that is connected to the output of Actuation component 4.2.3 in simulation framework and system states can be redirected into it on every sense. This method fits perfectly where the system tries to meet specific SLAs that are changing over time.

Appendix A

Simulation Framework Usages Described

In this section a quick tour is given to demonstrate how to use the simulation framework to simulate cloud environment, write your own controller and identify a system.

Check out and build

The code for simulation framework is distributed under Apache Software License [1]. Check out the source code from `Github` repository:

```
# git clone git://github.com/amir343/ElasticStorage.git
```

You should have `Maven` 2 installed in order to be able to build the source code:

```
# mvn clean install
```

This will download the required libraries from Maven repositories and builds the source code.

How to construct the cloud?

A cloud environment consists of several elements that each one is described in the following can be passed to `Cloud` class:

- `cloudProviderAddress(address, port)*`: binds the cloud provider to this address and port. The address should refer to `localhost`.
- `data(name, size, size unit)*`: defines the data blocks that will be used during the simulation and their corresponding sizes. The names must be unique.

- `replicationDegree(number)*`: defines the replication degree to meet by Elastic Load Balancer.
- `addressPoll(file name)*`: points to address ranges that are available for this simulation.
- `node(name, address, port)`: defines a storage instance with a name and address, port to bind to. The name should be unique.
  - `cpu(n)`: defines the CPU for the instance with frequency of `n` GHz.
  - `memoryGB(n)`: defines the memory for the instance with size of `n` GB.
  - `bandwidthMB(n)`: defines the bandwidth for the instance with capacity of `n` MB.
- `sla()`: enables the SLA violation calculation according to the parameter that will be defined as the following:
  - `cpuLoad(n)`: SLA requirements for percentage of average CPU load in the system.
  - `responseTime(n)`: SLA requirements for average response time (`ms`) in the system.
  - `bandwidth(n)`: SLA requirements for average bandwidth per download (`B/s`) in the system.

Using these elements a complete cloud environment can be built. Note that elements related to instances are not mandatory. Instances can be launched by cloud provider as well. After this definition, the object that holds theses elements should be started by calling the method `start()`.

Controller also should be defined but separately. The class that is responsible for controller is `ControllerApplication` and has the following element:

- `controllerAddress(address, port)`: defines the address and port that controller will be bind to.

Note that the address should be different from address that are included in `addressPoll` and cloud provider address. Controller is started with a method called `start()`. These definition can be put in a source code with desire name for class inside a `main` method. It is a good practice to put your class in package `scenarios.executor`. To start the simulation it is enough to run this class. An example scenario can be like the following:

Listing A.1. A sample scenario to create cloud environment

```
package scenarios.executor;

```

```java
import cloud.CloudProvider;
import econtroller.ElasticController;
import instance.Instance;
import instance.common.Size;
import org.apache.log4j.PropertyConfigurator;
import scenarios.manager.Cloud;
import scenarios.manager.ControllerApplication;

public class TestScenario {

    public static final void main(String[] args) {
      Cloud cloud = new Cloud(CloudProvider.class, Instance.class) {
        {
          cloudProviderAddress("127.0.0.1", 23444);
          node("node1", "127.0.0.1", 23445).
            cpu(2.2).
            memoryGB(8).
            bandwidthMB(2);
          data("block1", 2, Size.MB);
          data("block2", 4, Size.MB);
          data("block3", 3, Size.MB);
          data("block4", 1, Size.MB);
          data("block5", 4, Size.MB);
          replicationDegree(2);
          addressPoll("addresses.xml");
          sla()
            .cpuLoad(30)
            .responseTime(1000);
        }
      };

      cloud.start();

      ControllerApplication controller =
    new ControllerApplication(ElasticController.class) {
        {
          controllerAddress("127.0.0.1", 23443);
        }
      };

      controller.start();
    }
```

addressPoll

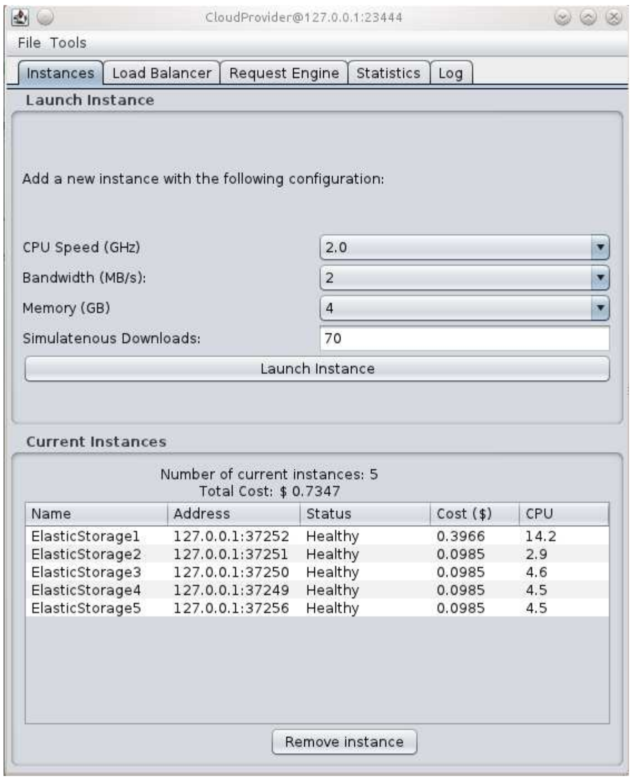
addressPoll is an XML file that defines the elastic IP address ranges that can be used by cloud provider. A simple example is like:

Listing A.2. A sample Elastic IP addresses definition

```xml
<addressPoll>
  <addresses>
    <addressRange>
```

```xml
            <ip>127.0.0.1</ip>
            <startPort>37000</startPort>
            <endPort>38001</endPort>
        </addressRange>
        <addressRange>
            <ip>127.0.0.1</ip>
            <startPort>47000</startPort>
            <endPort>48005</endPort>
        </addressRange>
        <addressRange>
            <ip>127.0.0.1</ip>
            <startPort>51000</startPort>
            <endPort>51857</endPort>
        </addressRange>
    </addresses>
</addressPoll>
```

This definition includes 2866 unique elastic addresses. Note that `startPort` can not be greater than `endPort`. The cloud provider window is demonstrated in Fig. A.1



Add a new instance with the following configuration:

CPU Speed (GHz)	2.0
Bandwidth (MB/s):	2
Memory (GB)	4
Simulatenous Downloads:	70

Number of current instances: 5
Total Cost: $ 0.7347

Name	Address	Status	Cost ($)	CPU
ElasticStorage1	127.0.0.1:37252	Healthy	0.3966	14.2
ElasticStorage2	127.0.0.1:37251	Healthy	0.0985	2.9
ElasticStorage3	127.0.0.1:37250	Healthy	0.0985	4.6
ElasticStorage4	127.0.0.1:37249	Healthy	0.0985	4.5
ElasticStorage5	127.0.0.1:37256	Healthy	0.0985	4.5

Figure A.1. Simulation Framework - Cloud Provider Window

Taking snapshot

At any point during the running of the simulation, you can take snapshots from any of cloud provider, instance or controller. This can be done by using `ALT + s` keys. You can take as many snapshots as you may need. These snapshots are saved into *Snapshots* panel in controller/instances windows and *Statistics* panel in cloud provider. You can save any of the snapshots to disk later on by selecting and right clicking on your desire one in the corresponding panel. If you need only the log messages to be saved you can right click on the log panel and select `save to file`.

Each snapshot includes the set of all diagrams and also log messages for that specific node in the system.

Headless Run

In order to consume less resources on the target machine, you can specify `headless()` in the cloud class of your scenario. This will prevent any construction of GUI for storage instances and save a lot of JVM heap memory. An instance windows is shown in Fig A.2.

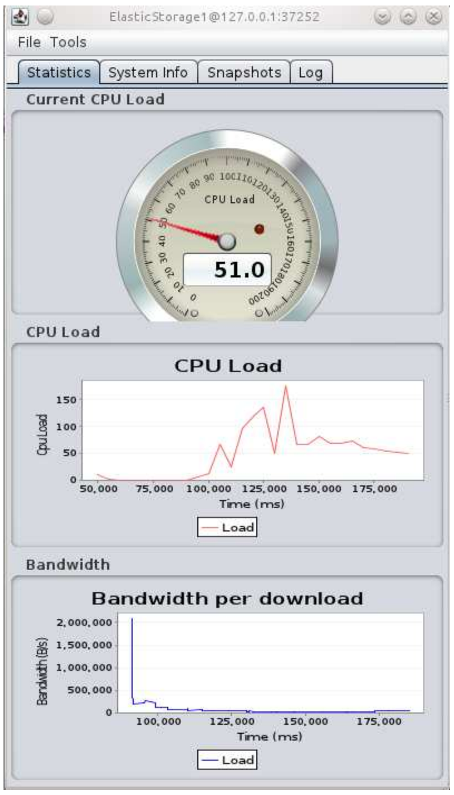


Figure A.2. Simulation Framework - Instance Window

How to add your own controller

If you require to write your own controller, you can do so by implementing the `ControllerDesign` interface. When you run the simulation, your controller will be listed with the name you have chosen for the class that implements the interface. It is a good practice to put the implemented class in package `econtroller.design.impl`. The name should not be used before and should be unique.

How to define custom distributions

You can define your own custom distribution to be used by Request Generator when sending request to cloud provider. Simply create a file and put your timeouts one in each line. So if you have something like

```
2
3
10
```

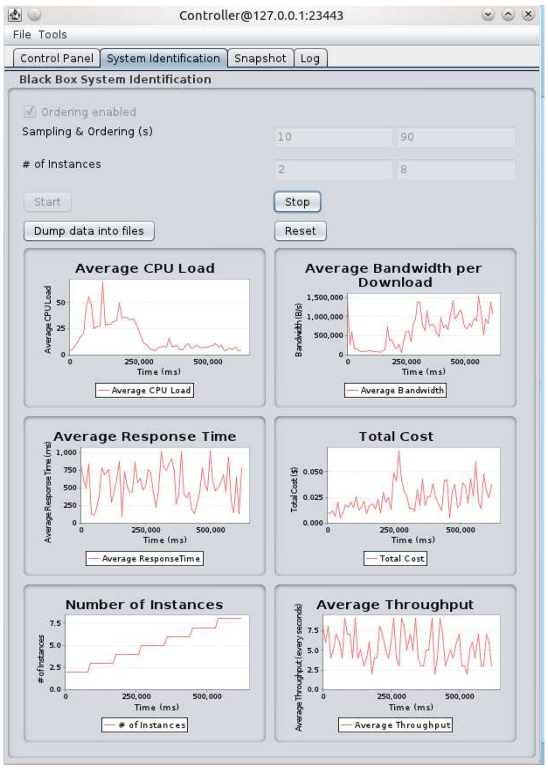
it means that the first request will be sent after 2 seconds, second request after 5 and the third one after 15 seconds. When the request generator reaches the end of the file, it repeats the distribution from the beginning. After you create the file, choose *Custom* from drop down list in Request Generator panel in cloud provider window.

How to do System Identification?

First define your scenario with Cloud and controller classes. Run your scenario. Try to launch at least two instances by cloud provider. Then connect the controller to cloud provider. In the meantime the instances are launching, select the distribution you want the Request Generator to send request and start it. When the instances are launched and ready to use, browse to System Identification panel in controller window. Make sure that settings are the ones you need start the modeler.

At any time during the system identification, you can stop the modeler or dump the collected training data into files. After one period that instances are scaled up and down, the modeler dumps the collected data into files and takes a snapshot as well which is listed in snapshot panel. It is recommended to do the system identification in headless mode that is described earlier. A sample system identification is depicted in Fig. A.3.

By default `Ordering enabled` check box is checked and it means that modeler acts as a System Identifier and orders new to launch/shut down instances. However, if you need to just monitor the system without affecting the number of instances, you can disable this option.



Black Box System Identification

☑ Ordering enabled

Sampling & Ordering (s) 10 90

# of Instances 2 8

Figure A.3. Controller Window - System Identification

Bibliography

[1] Apache software license. URL `http://www.apache.org/licenses/LICENSE-2.0.html`.

[2] Hbase. URL `http://hbase.apache.org/`.

[3] Hypertable. URL `http://code.google.com/p/hypertable/`.

[4] Json. URL `http://www.json.org`.

[5] Kompics. URL `http://kompics.sics.se/`.

[6] Scala language. URL `http://www.scala-lang.org/`.

[7] Terracotta. URL `http://www.terracotta.org/`.

[8] Terrastore. URL `http://code.google.com/p/terrastore/`.

[9] T. Abdelzaher and N. Bhatti. Web content adaptation to improve server overload behavior. In *WWW8 / Computer Networks*, pages 1563–1577, 1999.

[10] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: a control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, August 2002.

[11] Ahmad Al-Shishtawy. *Enabling and Achieving Self-Management for Large Scale Distributed Systems*. PhD thesis, KTH, 2010.

[12] Ahmad Al-Shishtawy, Vladimir Vlassov, Per Brand, and Seif Haridi. A design methodology for self-management in distributed environments. *Computational Science and Engineering*, 2009.

[13] K. J. Astrom and T. Hagglund. *PID Controllers: Theory, Design, and Tuning*. Instrumentation, Systems, and Automation Society, Research Triangle Park, NC, 1995.

[14] Stuart Bennett. *A history of control engineering*. 1993. ISBN 9-780863412998.

[15] E. Bonabeau. Editor's introduction: Stigmergy. *Artificial Life*, 5(2):95–96, 1999.

[16] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. *7th Operating System Design and Implementation,*, pages 335–350, Nov 2006.

[17] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *ACM Symposium on Principles of Distributed Computing*, pages 398–407, 2007.

[18] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *Seventh Symposium on Operating System Design and Implementation*, 2006.

[19] D. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN*, 17(1):1–14, 1989.

[20] Reuven Cohen. Defining elastic computing, September 2009. URL `http://www.elasticvapor.com/2009/09/defining-elastic-computing.html`.

[21] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo! 's hosted data serving platform. *Very Large Data Bases (VLDB)*, pages 1277–1288, August 2008.

[22] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, 2007.

[23] G. F. Franklin, J. D. Powell, and A. Emami-Naeini. *Feedback Control of Dynamic Systems*. Addison-Wesley, 3rd edition, 1994.

[24] N. Gandhi, D. M. Tilbury, Y. Diao, J. Hellerstein, and S. Parekh. Mimo conttrol of an apache web server: Modeling and controller design. *American Control Conference*, 2002.

[25] Rachid Guerraoui and Luis Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[26] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, September 2004. ISBN 978-0-471-26637-2.

[27] Paul Horn. Autonomic computing: Ibm's perspective on the state of information technology. Technical report, October 2001.

[28] IBM. An architectural blueprint for autonomic computing. Technical report, June 2006.

[29] Abbinav Kamra, Vishal Misra, and Erich M. Nahum. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. *In International Workshop on Quality of Service (IWQoS*, pages 47–56, 2004.

[30] David Karger, Eric Lehman, Tom Leighton, Mathhew Levine, Daniel Lewin, and Rina Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *In ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[31] Magnus Karlsson, Christos Karamanolis, and Xiaoyun Zhu. Triage: Performance isolation and differentiation for storage systems. In *In International Workshop on Quality of Service (IWQoS)*, pages 67–74, 2004.

[32] J. O. Kephart and D. M. Chess. The vision of autonomic computing. Technical report, January 2003.

[33] Srinivasan Keshav. A control-theoretic approach to flow control. In *ACM SIGCOMM*, September 1991.

[34] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, pages 558–565, July 1978.

[35] Leslie Lamport and Keith Marzullo. The part-time parliament. *ACM Transactions on Computer Systems*, 16:133–169, 1998.

[36] H. D. Lee, Y. J. Nam, and C. Park. Regulating i/o performance of shared storage with a control theoretical approach. *NASA/IEEE conference on Mass Storage Systems and Technologies (MSST)*, April 2004.

[37] Baochun Li and Klara Nahrstedt. A control theoretical model for quality of service adaptations. In *In Proceedings of Sixth International Workshop on Quality of Service*, pages 145–153, 1998.

[38] Baochun Li and Klara Nahrstedt. A control-based middleware framework for quality of service adaptations. *IEEE Journal on Selected Areas in Communication*, 1999.

[39] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated control for elastic storage. *International Conference on Autonomic Computing*, pages 1–10, 2010.

[40] Harold C. Lim, Shivnath Babu, Jeffrey S. Chase, and Sujay S. Parekh. Automated control in cloud computing: Challenges and opportunities. *Automated Control for Datacenters and Clouds*, pages 13–18, 2009.

[41] C. Lu, T. Abdelzaber, J. Stankovic, and S. Son. A feedback control approach for guaranteeing relative delays in web servers. In *Real-Time Technology and Applications Symposium, 2001. Proceedings. Seventh IEEE*, pages 51–62, 2001.

[42] Saverio Mascolo. Classical control theory for congestion avoidance in high-speed internet. In *Decision and Control*, December 1999.

[43] Peter Mell and Timothy Grance. The nist definition of cloud computing (draft). Technical Report 800-145, National Institute of Standards and Technology, January 2011.

[44] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *4th ACM European conference on Computer systems*, pages 13–26, 2009.

[45] Sujay S. Parekh, Joe Hellerstein, T. S. Jayram, Neha Gandhi, Dawn Tilbury, and Joe Bigus. Using control theory to achieve service level objectives in performance management, 2001.

[46] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal, Special Issue on Grids and Worldwide Computing Programming Models and Infrastructure*, 13(4):277–298, January 2006.

[47] Martin Placek and Rajkumar Buyya. A taxonomy of distributed storage systems. Technical Report GRIDS-TR- 2006-11, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, July 2006.

[48] Anders Robertsson, Björn Wittenmark, and Maria Kihl. Analysis and design of admission control in web-server systems. In *In American Control Conference (ACC)*, 2003.

[49] David Steere, Ashvin Goel, Joshua Gruenberg, Dylan Mcnamee, Calton Pu, and Jonathan Walpole. A feedback-driven proportion allocator for real-rate scheduling. In *Operating Systems Design and Implementation (OSDI)*, Feburary 1999.

[50] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. *18th ACM Symposium on Operating Systems Principles*, pages 230–243, October 2001.

TRITA-ICT-EX-2011:247