Abstract

Cloud computing delivers on-demand computing resources in a pay-as-you-go manner. One of the runtime environment provided to cloud-based applications is Node.js, which has gained significant popularity in cloud development due to its asynchronous, non-blocking and event-driven nature. However, the runtime environment Node.js fails to fully utilize the resource offered by a central processing unit (CPU), when operated in a multi-core cloud system. Such an operation limitation is a result of the single-threaded execution, as well as the scalability challenges. In order to overcome scalability challenges, a built-in cluster is required to spawn multiple worker processes. Moreover, certain cloud systems, including Platform-as-a-Service (PaaS), can resolve scalability challenges experienced by the runtime environment Node.js. When Node.js web applications are deployed in cloud systems, multiple instances of the Node.js web application can be spawned. This approach is commonly called horizontal scaling in cloud systems. Scalability challenges have to be considered to ensure good performance and support to achieve the maximum number of concurrent client requests.
For this reason, this thesis investigates scalability challenges and develops a benchmark suite used to define metrics for the Node.js scalability investigation. Furthermore, this thesis introduces the scalability-oriented benchmark suite as a tool to measure and evaluate the Node.js scalability in the cloud system.
Dedication

To my parents

Zhu, Xing and Jia, Lijun

and my grandma

Zhang, Rongzhen
Acknowledgements

Firstly, I would like to thank my supervisor Kenneth B. Kent for his continuous support of my whole Master’s study.

Besides my supervisor, my sincere thanks also goes to my colleague Panagiotis Patros for providing valuable technical suggestions and feedback; the CASA project manager Stephen MacKay for assisting in and improving my English technical writing; and all of my other colleagues at the Centre Advanced Studies-Atlantic for providing me with the necessary project background and technical support. I also would like to thank Michael Dawson, our IBM contact, for his technical assistance in the whole project.
Table of Contents

Abstract ii
Dedication iv
Acknowledgments v
Table of Contents vi
List of Tables xi
List of Figures xii
Abbreviations xvi

1 Introduction and Motivation 1

2 Background 4
  2.1 Cloud Computing 4
     2.1.1 Service Models 5
     2.1.2 Development Model 7
     2.1.3 Platform-as-a-Service Cloud 8
2.1.4 Scaling Service ........................................ 9
2.1.5 Load Balancing ........................................ 11
2.2 Docker and Docker Swarm ................................. 12
  2.2.1 Docker Architecture ................................. 12
  2.2.2 Virtual Machines and Docker Containers .......... 15
  2.2.3 Docker Swarm ......................................... 17
2.3 Node.js ................................................... 21
  2.3.1 JavaScript in Node.js ................................. 21
  2.3.2 Node.js Internal Architecture ....................... 22
  2.3.3 Single-threaded Event Loop Model in Node.js ...... 23
  2.3.4 Asynchronous I/O and Event-driven Node.js ....... 25
  2.3.5 Scalable Node.js ..................................... 27
2.4 Benchmark ............................................... 28
2.5 Scalability ................................................ 30
  2.5.1 Scalability Metrics .................................. 30
  2.5.2 Categories of Scalability ......................... 32
  2.5.3 Mathematical Models for Scalability .......... 32
    2.5.3.1 Linear Scalability Model .................... 33
    2.5.3.2 Sub-linear Scalability Models ............... 34
2.6 Related Work ............................................ 39
2.7 Summary ................................................ 42

3 Problem Refinement ......................................... 44
5.1.2.2 Scalability Patterns .......................... 90
5.2 Experimental Results and Analysis .................... 93
  5.2.1 S/MISN Scalability Pattern Analysis ................. 94
    5.2.1.1 HS Analysis Using CPU-intensive Application Benchmark .......................... 94
    5.2.1.2 HS Analysis Using Network-intensive Application Benchmark ......................... 100
    5.2.1.3 Comparison Analyses ........................ 103
  5.2.2 MIMN Scalability Analysis ........................ 112
    5.2.2.1 Using CPU-intensive Application Benchmark .................. 112
    5.2.2.2 Using Network-intensive Application Benchmark .................. 116
  5.3 Summary ............................................ 121

6 Conclusion and Future Work .......................... 123

Bibliography ........................................... 132

A Private Cloud Configuration Files .................. 133
  A.1 Dockerfile File ................................... 133
  A.2 Vagrant File .................................... 134

B docker-compose.yml Files .......................... 135
  B.1 Using S/MISN Scalability Pattern ................. 135
  B.2 Using MIMN Scalability Pattern ................. 136
Vita
List of Tables

3.1 Hardware Environment ........................................ 49
3.2 Software Environment .......................................... 50

4.1 Parameters in the Benchmark Summary .......................... 56
4.2 Summary of Resource Usage Collection .......................... 67
4.3 Summary of Resource Usage Calculation Formulas ............... 69

5.1 Scalability Variables Classification .............................. 89
5.2 Experimental Configurations ...................................... 92
5.3 Coefficient Values of Scalability Models for CPU-intensive Application with HS and CM Strategy ................. 109
5.4 Coefficient Values of Scalability Models for CPU- and Network-intensive Application .............................. 111
5.5 Total available virtual CPU cores and RAM for various cluster size ................................................................. 112
5.6 Maximum throughput of CPU-intensive test application achieved for various cluster size ........................................... 114
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Cloud Computing Service Model</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Horizontal Scaling and Vertical Scaling</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Docker Architecture</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>VM and Docker Container Architecture</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>Docker Swarm Networks</td>
<td>19</td>
</tr>
<tr>
<td>2.6</td>
<td>Node.js Async. I/O Event-driven Model</td>
<td>26</td>
</tr>
<tr>
<td>2.7</td>
<td>Linear Scalability and Sub-linear Scalability</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Benchmark Suite Architecture</td>
<td>53</td>
</tr>
<tr>
<td>4.2</td>
<td>Workflow of the Executor</td>
<td>55</td>
</tr>
<tr>
<td>4.3</td>
<td>Schema of Benchmark Run Report</td>
<td>60</td>
</tr>
<tr>
<td>4.4</td>
<td>Sample Final Report</td>
<td>61</td>
</tr>
<tr>
<td>4.5</td>
<td>Sample Data of Resource Usage in JSON</td>
<td>68</td>
</tr>
<tr>
<td>4.6</td>
<td>Benchmark Report Schema</td>
<td>72</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview of Private Cloud</td>
<td>85</td>
</tr>
<tr>
<td>5.2</td>
<td>Overview of Two Scalability Patterns</td>
<td>90</td>
</tr>
<tr>
<td>5.3</td>
<td>CPU-intensive Application Total Cluster Throughput (worker3)</td>
<td>95</td>
</tr>
</tbody>
</table>
5.4 CPU-intensive Application Total Cluster Response Time (worker3) 95
5.5 CPU-intensive App. Total Cluster CPU Utilization (worker3: maximum CPU utilization of total cluster is 400%) 97
5.6 CPU-intensive Application Total Cluster Incoming Network Bandwidth (worker3) 97
5.7 CPU-intensive Application Total Cluster outgoing Network Bandwidth (worker3) 97
5.8 CPU-intensive Application Total Cluster Throughput (worker6) 98
5.9 CPU-intensive Application Total Cluster Response Time (worker6) 98
5.10 CPU-intensive App. Total Cluster CPU Utilization (worker6: maximum CPU utilization of total cluster is 800%) 98
5.11 Network-intensive Application Total Cluster Throughput 100
5.12 Network-intensive Application Total Cluster Response Time 100
5.13 Network-intensive Application Total Cluster CPU Utilization (Maximum CPU utilization of total cluster is 800%) 102
5.14 Network-intensive Application Total Cluster Outgoing Network Bandwidth 102
5.15 CPU-intensive Application Throughput of HS and CM Strategy 104
5.16 CPU-intensive Application Response Time of HS and CM Strategy 105
5.17 CPU-intensive Application Total Cluster CPU Utilization of HS and CM Strategy (Maximum CPU utilization of total cluster is 400%) 105
5.18 USL Models of CPU-intensive Application Maximum Throughput with HS and CM Strategy ..................... 108
5.19 Amdahl’s Law Models of CPU-intensive Application Maximum Throughput with HS and CM Strategy ............. 108
5.20 USL Model of CPU- and Network-intensive Application Maximum Throughput ........................................ 110
5.21 Amdahl’s Law Model of CPU- and Network-intensive Application Maximum Throughput .............................. 110
5.22 CPU-intensive Application Total Cluster Throughput ............. 113
5.23 CPU-intensive Application Total Cluster Response Time ........... 113
5.24 CPU-intensive Application Total Cluster CPU Utilization (Maximum CPU utilization of one-node cluster is 400%, two-node cluster is 800%, three-node cluster is 1200%, four-node cluster is 2000%, and five-node cluster is 2800%) ............... 115
5.25 CPU-intensive Application Total Cluster Incoming Network Bandwidth ...................................................... 116
5.26 CPU-intensive Application Total Cluster outgoing Network Bandwidth ...................................................... 116
5.27 Network-intensive Application Total Cluster Maximum Throughput .......................................................... 117
5.28 Network-intensive Application Total Cluster Minimum Response Time ..................................................... 118
5.29 Network-intensive Application Total Cluster Maximum CPU
Utilization (Maximum CPU utilization of one-node cluster is
400\%, two-node cluster is 800\%, three-node cluster is 1200\%,
and four-node cluster is 2000\%) .................................. 119

5.30 Network-intensive Application Total Cluster Maximum Out-
going Network Bandwidth ............................................. 119
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASA</td>
<td>Centre Advanced Studies Atlantic</td>
</tr>
<tr>
<td>CLI</td>
<td>Command Line Interface</td>
</tr>
<tr>
<td>CM</td>
<td>Cluster Module</td>
</tr>
<tr>
<td>HS</td>
<td>Horizontal Scaling</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure-as-a-Service</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter-process Communication</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>MIMN</td>
<td>Multiple Instance in Multiple Node</td>
</tr>
<tr>
<td>PaaS</td>
<td>Platform-as-a-Service</td>
</tr>
<tr>
<td>RC</td>
<td>Resource Collector</td>
</tr>
<tr>
<td>RSS</td>
<td>Resident Set Size</td>
</tr>
<tr>
<td>RX</td>
<td>Receive</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software-as-a-Service</td>
</tr>
<tr>
<td>S/MISN</td>
<td>Single/Multiple Instance in Single Node</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>TX</td>
<td>Transmit</td>
</tr>
<tr>
<td>USL</td>
<td>Universal Scalability Law</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction and Motivation

Node.js is one of the popular runtime environment utilized in cloud systems. It is widely utilized for server-side development due to its nature of being a JavaScript runtime that is executed on the Google Chrome V8 engine [1]. The combination of the fast execution and single-threaded nature, can make Node.js in the asynchronous non-blocking I/O, event-driven nature, and scalability. In addition, Node.js is also scalable, even though it is single-threaded. The cluster module [2] is a Node.js built-in module, which enables Node.js to fork multiple worker processes that work in parallel with the master process. The master process is responsible for load balancing and worker process maintenance, when Node.js works as a web server. The application of the cluster module is one of the approaches to scale Node.js, and ensure utilizing all CPU cores in a multi-core platforms.

Among the services defined by cloud computing, is the Platform-as-a-service

1
(PaaS), which provides with a platform to maintain a complete software development lifecycle. Moreover, the PaaS service model can be incorporated in the development, deployment, testing, and maintenance of software lifecycle. Docker Swarm is a popular example of such a PaaS cloud service model, which provides a clustering functionality for Docker and turns a set of distributed Docker hosts into a large single cluster [3]. Docker Swarm also provides horizontal scaling, which enables a single web application to be replicated to multiple instances. A Node.js web application can also be deployed in Docker Swarm and spawned to multiple instances. The cluster module can also be utilized by Node.js in Docker Swarm.

Both scalability strategies, application of the cluster module (CM) and horizontal scaling (HS), help Node.js to achieve high scalability in clouds. Therefore, a Node.js scalability investigation, based on different scalability strategies, is a crucial research project to determine the maximum number of concurrent requests from clients and find performance bottlenecks. This thesis will investigate the characteristics of Node.js scalability and identify performance bottlenecks depending on different types of Node.js web applications. The main contributions of this thesis are:

- Develop and implement a scalability-oriented benchmark suite that contains a set of resource-intensive test applications and several benchmark infrastructure components.
- Build a private cloud using Docker Swarm technique stacks, which pro-
vides a distributed, scalable and multi-core experimental environment, aiming to perform a scalability-oriented investigation.

- Perform a number of scalability experiments utilizing the benchmark suite. Based on experimental results, we analyze scalability effects of different types of the test application; quantify and characterize the scalability of Node.js using a regression analysis; identify the associated performance bottlenecks; and make some predictions about the scaling failure of the network-intensive test application.

The first part of this thesis introduces some relevant background in Chapter 2, such as various aspects of Node.js, and its relationships with JavaScript. It also introduces cloud computing, the scalability concept, benchmarking and several related works. Chapter 3 presents a brief project design, including the goal of this research, approaches, and the software and hardware environment in this project. The scalability-oriented benchmark suite implementation, including the resource-intensive test applications and associated infrastructure components, is documented in Chapter 4. Chapter 5 introduces the scalability investigation, including the scalability experimental setup and design, and analyses based on experimental results. Chapter 6 draws several conclusions and proposes future work for this thesis.
Chapter 2

Background

2.1 Cloud Computing

Cloud computing abstracts and virtualizes computing, networking, and storage resources, and it provides the end-user with elastic and on-demand services in a pay-as-you-go manner. Furthermore, cloud computing enables users to access service-oriented and virtualized resources through the internet. The National Institute of Standards and Technology, Information Technology Laboratory (NIST) defines cloud computing as:

“Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., network, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”[4]
NIST also defines three service models and four development models. The following Sections 2.1.1, and 2.1.2 will introduce the cloud computing background according to definitions from NIST. Section 2.1.3 will introduce one of the cloud service models, PaaS; Section 2.1.4 will introduce two scaling solutions of PaaS clouds to handle the increased loads. Section 2.1.5 will present load balancing techniques in PaaS clouds. Section 2.1.6 will introduce a representative PaaS cloud, Docker.

2.1.1 Service Models

Cloud computing defines three service models, Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) [4]. Figure 2.1 shows the architecture of cloud computing service models.
• **Infrastructure-as-a-Service**: The IaaS service model is placed at the bottom of the cloud system stack. The IaaS service model abstracts the physical or virtual hardware infrastructure and associated software as the services, for instance, computing power, networks, and storage. IaaS offers on-demand hardware and associated software, and administrative services to users, which allows users to deploy and maintain their applications, data, and platforms that run their applications. This is a significant cost saving for the end user, because there is no need to purchase networking, computing power, and storage equipment. Instead, the end user can purchase and utilize services provided by an IaaS provider in a contract or pay-as-you-go basis. OpenStack and Amazon EC2 are representative IaaS cloud providers.

• **Platform-as-a-Service**: The PaaS service model lies in the middle of the cloud system stack. PaaS provides users with a platform that allows users to deploy and maintain their applications. PaaS cloud providers include Cloud Foundry, Docker Swarm, and Google Apps Engine. A more detailed background of the PaaS service model will be introduced in Section 2.1.3.

• **Software-as-a-Service**: The SaaS service model is hierarchically at the top of the cloud system stack. It delivers a complete software solution and provides an on-demand software service to users. The SaaS provider has full control over the infrastructure, performance,
scalability, privacy and so on. The most common way for the end user to use their software service is a web-based user interface. In this way, the end user does not need to purchase a software license and install it in the individual local machine, but subscribing to the software and remotely accessing it in the cloud through the internet with a browser or other mobile device. Facebook, Office Online, etc. fall into SaaS clouds.

2.1.2 Development Model

NIST defines four basic cloud development models, which are:

- **Private cloud**: The cloud infrastructure is operated by and limited to a single organization. It may be owned, operated or managed by the organization, a third party, or different combinations. Private clouds centralize different levels of computing resources and provide services at the organizational level, which is easier to manage, maintain, deploy and use.

- **Community cloud**: When several organizations jointly construct and share the cloud infrastructure, supporting a specific organization that has shared concerns, such a model is referred to as a community cloud. A community cloud is operated and managed by one of the organizations in the community or a third party.

- **Public cloud**: Public cloud infrastructure is constructed, owned, oper-
ated and managed by a business, academic, or government organization, which is available to general public users. General public users can access the cloud using web browsers and utilize the resources as needed. They pay for the amount that they use.

- Hybrid cloud: Hybrid cloud infrastructure contains two or more clouds, such as a private cloud, a community cloud or a public cloud. An individual cloud in the combination remains a unique entity, but it is linked together by standardized or proprietary technology that enables data and application portability.

2.1.3 Platform-as-a-Service Cloud

NIST defines a general PaaS service model as:

“The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure, including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.” [4]

This definition shows that a PaaS cloud delivers a managed platform and a solution stack that allows users to deploy and run their applications without being concerned about the underlying infrastructure in the cloud. A
PaaS cloud provides a ready-to-use execution environment supporting the full software lifecycle, which enables the users to utilize these services and the platform to develop, test, deploy and maintain their applications in the cloud.

One example of a PaaS cloud, Cloud Foundry [5], provides a managed platform that enables users to run their applications in an OS container and also provides users with the operating system, programming language runtime, middleware and database as a service that supports users’ development processes. Another example of a PaaS cloud, Docker Swarm, allows users to upload their own OS images and then applications will be deployed and run by using these customized OS images.

2.1.4 Scaling Service

The workload on the cloud is driven by client requests. As the number of concurrent clients and their requests increase, PaaS clouds provide two forms of scaling, horizontal scaling and vertical scaling (Figure 2.2), to help applications handle the increasing loads.

- Horizontal scaling: also called scaling out, adds a number of instances of a single application to increase the overall application capacity. Each additional instance generally has the equivalent capacity, such as the same amount of memory and the same CPU power. High-performance computing is more easily implemented within the limited hardware
resources by utilizing the scaling-out solution, which is cost-efficient and there is less downtime when it upgrades. Users can use the capacity of the applications on demand and automatically as well.

- Vertical scaling: also referred to as scaling up, adds additional hardware resources to the existing instances of an application, such as additional RAM, bandwidth and a number of CPU cores to increase the application capacity. The resource can be either a physical or a virtual resource that is dynamically added to a virtual machine or its application. The vertical scaling solution is relatively more straightforward than the horizontal scaling solution, because it essentially resizes the cloud system and increases the capacity without changing any code. However, this solution is expensive and produces downtime due to hardware changes involved. Vertical scaling sometimes cannot take effect in a single-threaded programming language, such as Node.js.

Most PaaS cloud providers, such as Cloud Foundry [5], Docker [6], and Ama-
zon EC2 [7], provide both the scale-out solution, and the scale-up solution. Moreover, they also provide the scaling down solution that can be used to release no longer needed resources, which can be applied to both horizontal scaling and vertical scaling.

2.1.5 Load Balancing

Load balancing is a mechanism that evenly distributes the dynamic incoming loads from clients to all hosts in the whole cloud system, which is one of the central issues in cloud computing [8]. Some hosts in the cloud system handle heavy workloads, whereas some are idle and process little work. Load balancing can avoid such uneven distribution of workloads across multiple hosts in the cloud system and avoid overloading a single host. In addition, load balancing can prevent bottlenecks that occur in the cloud system due to host failure. For example, if one or more services hosted in cloud nodes fail, load balancing can be utilized to distribute workloads to another node that hosts available services, which leaves time to re-provision or de-provision the services. Load balancing also enables scalability of the cloud system by evenly spreading workloads to cloud nodes and ensuring the resources are utilized fairly and efficiently. Scalability will be introduced in Section 2.4. There are several load balancing policies utilized to control traffic distribution to cloud nodes, such as the round-robin policy, and the least connection policy. The round-robin policy is a popular one and is utilized by many load balancers, such as Docker Swarm’s built-in load balancer. Under the Round-
Robin load balancing policy, the load balancer forwards a client request to every server in the group in turn. Once it reaches the end it then loops back, going through the group of cloud servers again.

2.2 Docker and Docker Swarm

Docker is an open platform for developers and system administrations to build, ship, and run distributed applications, whether on laptops, data center VMs, or the cloud [6]. Docker enables applications to run in containers that isolate them from other containers and host infrastructure, which provides a solution to facilitate the development, deployment, delivery and execution of applications [9]. Subsection 2.2.1 will give an introduction to the Docker architecture. Subsection 2.2.2 will present more details about Docker containers and compare them with virtual machines that are a traditional virtualization technique applied in cloud systems. Subsection 2.2.3 will introduce Docker in swarm mode, referred to as Docker Swarm.

2.2.1 Docker Architecture

Docker contains four components, Docker client, Docker daemon, Docker registry, and Docker objects. The Docker objects include the Docker image and the Docker container. The overview of the Docker architecture is displayed in Figure 2.3. As Figure 2.3 shown, Docker uses the client-server architecture, where the Docker client communicates with the Docker Daemon that does
Figure 2.3: Docker Architecture [10]

The heavy lifting of building, running and distributing the Docker container. The Docker client and the Docker daemon can run either on the same host or different hosts that the client connects to a remote Docker daemon [10].

- **Docker client**: implements a high-level representational state transfer (REST) API and Command Line Interface (CLI) that can be utilized by Docker users to interact with Docker. The CLI uses the REST API to control or interact with the Docker daemon. When a Docker user runs commands using the Docker API, the commands are sent to the Docker daemon. The Docker client can communicate with more than one Docker daemon. For example, docker build, docker run and docker pull can be utilized to build the container based on the image, run the container, and pull the image from the Docker registry respectively.

- **Docker daemon**: also referred to as dockerd. It is utilized to manage and maintain Docker objects, such as containers, images and networks.
They communicate through a network interface bridge, `docker0`. `Docker` also listens to Docker API requests from Docker clients.

- **Docker registry**: can be either the public or a private store for downloading and uploading Docker images. Docker hub and Docker cloud are public image stores that anyone can use. Docker users can setup a private image registry for local use.

- **Docker objects**: when users use Docker, several Docker objects, such as containers, images, networks, volumes, and plugins are involved. The following will briefly introduce Docker images and containers.
  
  - **Docker image**: is a read-only template with instructions to create Docker containers without any future configurations. Docker images are designed to be reusable and extensible to allow customizations, which define the applications and their related execution environment, such as binary files, dependencies, libraries, configuration files and so on. Once Docker images are uploaded to the Docker hub, a public registry, everyone can download them and customize them to meet specific needs, making them ready to be deployed in a container.
  
  - **Docker container**: is a runtime instance of an image, which means it can be created from a Docker image [10]. It holds everything that an application needs. Docker users can use Docker API or CLI to interact with Docker to create, start, stop, move, or delete
a container. It is isolated from other containers and the host environment by default, unless users configure it to access host files.

2.2.2 Virtual Machines and Docker Containers

VMs and containers are both core virtualization technologies, which are typically applied to the cloud system and aim to ensure isolation and resource control. Isolation means the execution of an application instance cannot affect another execution of the application instance in the same system. They are both isolations from the host infrastructure. Resource control refers to constraining execution of an application instance to a specific set of computing resources, such as CPU core and memory. A PaaS cloud runs the application instances inside either the VM or the container, and imposes resource constraints on them, thus achieving both isolation and resource control.

However, VMs and containers are different and solve different problems. The architectural difference between VMs and containers is displayed in Figure 2.4. A VM instance is built on the top of a hypervisor and runs a single large process. A hypervisor is the software that provides isolation for VMs running on the physical infrastructure and coordinates the host system, which allows different kernels to run on top of the physical host. The container-based virtualization solution could be used as an alternative to a VM in the cloud. A container is lightweight and provides a portable and fast approach to package the application and related binaries and libraries. The Docker
solution is built on the Linux Container (LXC) technique [12]. Docker allows building and running different containers directly on top of the host OS at the same time, which does not need the extra load of the hypervisor like the VM. Each Docker container could be managed and maintained by the Docker daemon. All Docker containers do not require or include a separate full OS, but instead, share the same underlying kernel and rely on the kernels functionality. In this way, Docker containers can only include the files required by the applications, which makes them lightweight and start much faster than VMs. They can be isolated from each other and are constrained to only utilize the allocated computing resource specified by users, which is achieved from Linux kernel features, Group Control (cgroups) and kernel namespaces.

- Namespace isolation can separate groups of processes, not allowing them to see and utilize resources in other groups. When a Docker
container is created and runs, a set of namespaces are created for that container and they have limited access from each other, such as, pid namespace (process isolation), net namespace (managing network interface), and mnt namespace (managing filesystem mount point) [10].

- Cgroups is another resource isolation and control technique, which manages and limits resource access for process groups. Cgroups enables a Docker container to share available resources and optionally enforce limits and constraints [10]. Cgroups also expose metrics about CPU, memory, network, and block I/O usage of a particular container through a pseudo-filesystem. The pseudo-filesystem is under /sys/fs/cgroup, where there are several sub-directories, corresponding to a different cgroups hierarchy. Once a Docker container is created, its one coordinated cgroup will be created as well in each hierarchy. For example, if users would like to track memory usage of a certain container, they can take a look at /sys/fs/cgroup/memory/docker/⟨docker container ID⟩.

2.2.3 Docker Swarm

Docker Swarm is a project that provides native clustering functionality for Docker [3]. It aims to turn a pool of multiple Docker hosts to one large single and virtual Docker cluster. The Docker host can be referred to as a Docker node, an instance of the Docker host that joins in the swarm. A Docker node
can play the manager role, called the manager node, or play the worker role, referred to as the worker node. The manager node is responsible for handling cluster management tasks, such as maintaining cluster states, and scheduling services; on the other hand, the worker node is in charge of receiving and executing tasks dispatched from the manager node.

When Docker Swarm users deploy their applications to the swarm, multiple containers will be created and run on single or multiple Docker nodes participating in the swarm. At this time, each container needs to communicate with each other and thus networks should be taken into account. Docker Swarm defines three types of networks: the overlay network, the ingress network, and the bridge network \texttt{docker\_gwbridge} [13]. Figure 2.5 presents the big picture of Docker Swarm networks. In this figure, there are three generic nodes, playing either the worker or the manager role, and several Docker containers are deployed in them; Docker containers and nodes are connected by various types of networks.

- Overlay network: the default overlay network facilitates communications among Docker daemons participating in the swarm. Docker Swarm users can also create and customize overlay networks for standalone containers, enabling container-to-container communications within the same or different swarm nodes. Swarm users can attach containers to overlay networks.

- Ingress network: is a special overlay network that manages the load bal-
Docker Swarm also provides a horizontal scaling service to handle increasing among containers. Within the ingress network, all swarm nodes participate in the ingress routing mesh and enable the load balancing functionality. It means that any swarm node can receive requests on a published port, and then it routes all these incoming requests to published ports on available nodes to an active container, even though this node has no tasks running.

- docker_gwbridge: is a bridge network that connects the overlay network to an individual Docker daemon's physical network. Each container by default is connected to its local Docker daemon hosts docker_gwbridge network.

Docker Swarm also provides a horizontal scaling service to handle increasing among containers.
ing loads. To achieve horizontal scaling, the swarm has two types of service deployment modes: replicated, and global. Under the replicated mode, Docker Swarm users can specify a number of configurable replicated containers. Docker Swarm supports a configuration file in YAML format, docker-compose.yml that defines the behavior of the Docker container in production. The docker-compose.yml file allows users to define how Docker containers should behave in the swarm, for example, defining service stack networks, volumes and so on. Docker Swarm users can define the number of replicated containers in the docker-compose.yml file to achieve horizontal scaling. In addition, Docker Swarm users can also utilize the CLI, “docker service scale”, to achieve the same result. In contrast, the global mode results in a single container deployed on every node participating in the swarm. Swarm users can either use the docker-compose.yml file or the CLI flag, “--mode global” to setup the global mode.

Once the number of replicated containers has been determined, the manager node will select nodes in the swarm to run containers according to a scheduling strategy, called spread. Under this scheduling strategy, the container placement is based on an assessment of resources available on nodes. For example, if nodes are allocated to the same resources or have the same ranking, the container is scheduled and distributed to the node that has the least number of containers. If nodes have different available resources, the spread scheduling strategy prefers the node that has more available resources.
2.3 Node.js

Node.js is a framework that provides a JavaScript runtime, running on Google’s open source V8 engine [1]. Node.js is most suited for I/O intensive tasks and promises a fast running speed in the cloud, because of its single-threaded, event-driven and asynchronous nature. This section will have a brief overview of Node.js, introducing the relationship between JavaScript and Node.js in Subsection 2.3.1, the Node.js internal architecture in Subsection 2.3.2, its single-thread with an event-loop model in Subsection 2.3.3, its asynchronous I/O event-driven model in Subsection 2.3.4, and its scalable characteristics in Subsection 2.3.5.

2.3.1 JavaScript in Node.js

JavaScript is a dynamic interpreted language, which supports functional programming and object-oriented programming. An object is utilized to store various data or more complicated entities in terms of a “key-value” pair that is also called a property. JavaScript is weakly typed and during runtime, properties can be created and added to an object dynamically. Everything in JavaScript is an object, including functions. A function is called a first-class object, which means a function can be used as an object that can be passed to another function (containing function) as an argument or returned from the containing function. Such functions that are passed to containing functions as arguments and to be used on the return value, are called callback
functions [15]. Callback functions will not be executed immediately when passed to the containing function, but will be executed later, which drives the asynchronous execution feature in Node.js. JavaScript is widely used as the front-end language, interacting with HTML through events. When a webpage has finished loading or user actions have taken place, they are regarded as events. JavaScript can detect such events by utilizing an event handler. The event handler listens and waits an event to occur. Once events happen, the handler processes them by executing callback functions.

2.3.2 Node.js Internal Architecture

Node.js follows a modular structure and can be divided into two main components: the core and its modules. The core is implemented in C and C++, which contains the V8 JavaScript engine, the thread pool with an event loop, and protocol bindings (sockets, HTTP, etc.).

- V8 is Google’s JavaScript engine written in C++, which compiles JavaScript code directly into assembly code ready for execution without further interpretation. Moreover, V8 also executes these compiled codes, handles memory allocation for objects, and collects unneeded objects referred to as garbage collection. The V8 runtime environment has three major components: a compiler, an optimizer and a garbage collector.

- Libuv is a C++ library that handles Node.js’s asynchronous I/O op-
erations and maintains an event loop. It has a fixed size POSIX asyn-
chronous thread pool to handle delegated time-consuming and blocking
I/O operations. A thread is allocated from the pool for each such I/O
operation. Once these I/O operations are completed, they notify the
event loop. In this way, the V8 and other Node.js components can
continue processing other requests.

The modules are functionality-organized JavaScript files that can be reused
by Node.js applications to avoid reinventing the wheel for every new applica-
tion. Node.js has some built-in core modules (cluster module, child process
module, etc.) that bind the JavaScript with C/C++ libraries. Apart from
these, there are thousands of external modules (express module, etc.) devel-
oped by the community.

2.3.3 Single-threaded Event Loop Model in Node.js

A web server handling multiple concurrent client connections can be either
utilizing a multi-threaded or a single-threaded model. There are a number
of web servers employing multithreaded programming techniques, such as
the Apache web server. A multi-threaded web server internally maintains
a thread pool that contains a limited number of threads. When it receives
a number of concurrent requests from the client, it assigns a thread from
the thread pool to a request. In this way, different threads simultaneously
process different requests with true parallelism, which is straightforward for
a modern multicore system running multiple concurrent threads. However, this approach does not work well when facing heavy blocking I/O operations, because limited threads can be blocked to prepare their responses. It results in remaining client requests waiting for a long time. Apart from those problems, there are also some management difficulties that accompany multithreaded programming techniques, such as deadlock and protection of shared resources among threads.

On the other hand, Node.js is designed to be single-threaded, which means there is only one thread sequentially executing JavaScript code and attached to an event loop. It uses non-blocking I/O calls, enabling a significant capability of processing tens of thousands of multiple concurrent client requests. The Node.js web server accepts requests from clients and places them into an event queue. The event loop utilizes an infinite loop that is maintained by the single thread to pick up client requests from the event queue. If no client request is in the queue, the event loop waits for incoming requests. This single thread not only maintains the event loop, but is responsible for handing off blocking I/O client requests to background threads and responding to the client. In contrast to the thread pool of the multi-threaded model, the thread pool of the single-threaded model only handles blocking I/O requests and there is no need to spawn more threads, which consumes fewer resources, such as memory.
2.3.4 Asynchronous I/O and Event-driven Node.js

The asynchronous, non-blocking I/O and event-driven features of Node.js derive from JavaScript’s event and callback function mechanisms explained in Subsection 2.3.1. When applying the event-driven and asynchronous I/O model, the applications rely on an event notification system where applications register interest in certain events (e.g. I/O operation completion). Once the events occur, the event notification system informs applications to handle events. The I/O operations are usually time-consuming and many processor cycles are needed, which results in applications blocking while waiting for I/O operations. The asynchronous I/O operation can address this problem by preventing the application from blocking, which wastes time waiting for entire I/O operation completion. In this way, the application registers post-I/O tasks with the event notification system and then can process something else. Once blocking I/O tasks are completed, the application will be notified by the event notification system and the post-I/O tasks will be triggered and processed. The post-I/O task is executed as a callback function by the main thread.

Node.js follows this model where every I/O operation occurs then the control is immediately returned to the caller and finally is handled by callback functions. Figure 2.6 presents the whole process under this model. The main thread in Node.js handles the incoming loads in terms of request connections and then delegates blocking I/O tasks, for instance, querying a database, and accessing a file, to the thread pool in libuv library within an OS scope.
after finding available threads. In other words, the background threads in the thread pool are in charge of processing blocking I/O tasks. The blocking I/O task completion is viewed as an event and an associated callback function is attached to it. Once the delegated blocking I/O is completed, the event will be triggered and the thread pool will re-introduce this event to the single thread and immediately the callback function will be called asynchronously. On the other hand, if the incoming loads do not require any blocking I/O, then the main thread will process it, prepare the response and send it back to the client.
2.3.5 Scalable Node.js

Section 2.3.3 introduced Node.js as a single thread; the program runs on a single thread executed by only one CPU core. Therefore, it cannot fully utilize CPU resources in a multicore system without utilizing a module that supports multi-threaded functionality. Even if a Node.js application is scaled up by adding more CPU cores or other hardware resources, the performance of Node.js does not improve proportionally. To resolve this problem, there are various approaches to achieve high scalability in Node.js applications to handle increasing workloads. By use of the `child_process` module, any number of Node.js processes can be created, which allows building distributed applications with many Node.js instances. Each child process of Node.js will be allocated to a given CPU and memory resource with its own V8 instance that the OS is responsible for. For example, if there is an eight-core machine, eight Node.js child processes can be spawned and each allocated to an independent processor with an extra master process out of eight. The Node.js child processes are automatically distributed across CPU cores by the OS, which fully take advantage of the multicore environment. There are multiple powerful functions and properties in the `child_process` module, allowing users to spawn, destroy, and communicate between parent and child processes.

The Node.js `cluster` module is built on top of the `child_process` module, which aims to help build scalable web applications easier and gain significant performance improvements by the creation of child processes. The `cluster` module formalizes the pattern where a master parent controls single or multiple
children. It exposes some functions that allow users to create an arbitrary number of worker processes. When the `cluster` module is applied to a web server, these worker processes work in parallel with the master process to handle requests. Essentially, the worker processes are spawned by using the `child_process.fork()` method in the `child_process` module, which means the worker processes can communicate with the master process via inter-process communication (IPC) [2]. The master process is in charge of initiating and maintaining worker processes. Moreover, the master process is also responsible for listening on a port and distributing incoming loads among worker processes in a round-robin policy.

Apart from the multi-process mechanisms mentioned above, there is another mechanism—horizontal scaling based on PaaS clouds. Section 2.1.4 introduced scaling services in PaaS clouds. Multiple identical instances can be created to form a cluster to handle increasing loads and the load balancing service is also taken into account by PaaS clouds. The load balancers distribute increasing loads among scaled instances.

### 2.4 Benchmark

A benchmark executes a set of test programs under a specific environment to measure and evaluate specific performance metrics for comparison purposes. Benchmarks can be classified into two categories: synthetic benchmarks, and real-world benchmarks. Synthetic benchmarks run a small piece of code to
measure and evaluate a specific aspect of the object. They are designed to test an individual component of the entire system through different isolated tests and obtain simply repeatable results for accurate comparisons and bottleneck identification. Real-world benchmarks give real workloads and run more complicated scenario test programs to measure and evaluate various aspects of objects.

Load testing is one type of benchmark test that aims to find out whether the system performs as intended under a specific load. External performance, such as traffic, and internal performance, such as the resource consumption, can be tested through load testing [16]. When a load test is conducted on a web application, workload sent to this web application is in terms of concurrent connections. Load testing can be performed in either the physical or virtual machine, by running a load testing tool in one machine and running the web application in a separate machine to avoid interference. The load testing tool generates a large amount of simulated HTTP traffic to the web application, and both internal and external performance of this web application can be measured.

JMeter [17] is such an open source load testing tool from the Apache Software Foundation. JMeter provides two modes to start load testing, the GUI mode and the non-GUI mode. The non-GUI mode has less overhead and increases JMeter capacities. When JMeter runs as the load testing tool, a test plan is built before starting JMeter. The test plan describes a series of steps JMeter will execute when run [17]. In addition, users can customize several load
testing parameters, such as the number and concurrency of HTTP requests, test duration, URL of web server, etc., passing them to the test plan. Once the test plan is ready, JMeter generates and fires workloads to web servers according to the test plan. At the end of load testing, JMeter can output a report in HTML or other formats.

2.5 Scalability

Scalability is a quality attribute of a system, which can be defined as a mathematical function, representing the relationship between dependent and independent variables (e.g., workloads, concurrency, and number of processors). Williams and Smith define scalability as: a measure of an application system’s ability to provide increased throughput, reduced response time and/or support more users when hardware resources are added [18]. This definition indicates the system is scalable if it uses additional resources to increase the throughput and reduce the response time. Section 2.5.1 will present frequently used metrics to measure the scalability, such as relative capacity. Section 2.5.2 will introduce categories of scalability, such as strong scalability and weak scalability. Scalability models will be introduced in Section 2.5.3.

2.5.1 Scalability Metrics

The following scalability metrics are most relevant for web applications.

- **Speedup**: is one of the most frequently used scalability metrics. Speedup
is a measure of the time reduction to complete a fixed workload on a multiprocessor compared to that required on a uniprocessor, when using the same algorithm or version of code. It is typically expressed as,

\[ S(p) = \frac{T(1)}{T(p)} \quad (1) \]

where \( T(1) \) is the execution time required to perform a fixed-size workload with a uniprocessor, and \( T(p) \) is the execution time that completes the fixed-size workload with a multiprocessor that has \( p \) processors.

- **Relative capacity**: measures the increase in the number of workloads that are processed in a fixed time when adding resources. Gunther et al. defined the relative capacity metric as the ratio (scaling factor) of the capacity with \( p \) processors to the capacity with one processor [19]. It indicates whether the system can process more workloads or support more concurrent users when adding more resources. When the relative capacity is measured using the maximum throughput, it is expressed as below,

\[ C(p) = \frac{X_{\text{max}}(p)}{X_{\text{max}}(1)} \quad (2) \]

Where \( p \) is the number of processors in the system; \( X_{\text{max}} \) is maximum throughput in terms of processed workloads within a fixed time-frame, which measures the capacity.

- **Efficiency**: can be defined as relative capacity normalized by the num-
number of processors. It indicates a ratio of useful work that is performed per processor.

\[ E(p) = \frac{C(p)}{p} \quad (3) \]

### 2.5.2 Categories of Scalability

Scalability can be classified into **strong scalability** and **weak scalability**. Strong scalability is when an increasing number of computing nodes are applied to a fixed-size problem. Achieving strong scalability will solve a fixed-size computation with shorter execution time, as the number of computing nodes increasing.

Weak scalability is when the problem size is fixed for each utilized computing node. Under a weak scalability, the execution time of system shows a constant behaviour, when the problem size increases accordingly with the number of computing nodes.

### 2.5.3 Mathematical Models for Scalability

The following subsections present a number of scalability models: the linear scalability model is introduced in Section 2.5.3.1; Amdahl’s Law [20] and Universal Scalability Law (USL) [21] are utilized to model the sub-linear scalability, and they are introduced in Section 2.5.3.2. All these scalability models are expressed as the relative capacity being a function of the number of processors.
2.5.3.1 Linear Scalability Model

The linear scalability model can be expressed formally as follows:

\[ C_L(p) = p \quad (4) \]

The relative capacity is equal to or is directly proportional to the processor count and the slope is equal to one in the linear scalability model. For example, if we double the number of processors, then it results in doubling the system’s throughput. When the linear scalability model is applied to cloud systems, throughput of cloud systems should scale linearly with an infinitely constant resource cost. The greater number of concurrent requests issued from the client side will all be answered by cloud systems with the addition of greater resources (hardware resource such as processors). The dashed line in Figure 2.7 represents linear scalability, where the throughput increases linearly with the addition of more processors. When cloud systems
fully take advantage of all these additional resources, it can achieve a linear scalability. However, linear scalability represents an ideal situation, and a system rarely achieves it. The following sections will introduce sub-linear scalability and its related models.

2.5.3.2 Sub-linear Scalability Models

With few exceptions, a system does not generally scale linearly due to various reasons, such as hardware limitations, service restrictions, network speed, and intra and inter-processor communication. Instead it follows sub-linear scalability whereby the relative capacity increases more slowly than linear scalability. The curve in Figure 2.7 represents sub-linear scalability, where the throughput does not always proportionally increase with greater numbers of processors. The following contexts introduce two sub-linear scalability models, Amdahl’s Law model and USL model respectively.

Based on the speedup, Amdahl’s Law shows the potential maximum speedup of parallel processing is limited by the serial portion of a computation. In other words, Amdahl’s Law states that if a single task that takes time $T_1$ is decomposed into even subtasks and they are processed in parallel using $p$ processors, but contains an irreducible fraction of the work $\sigma \in [0, 1]$ (serial fraction) that the task cannot be decomposed and executed in parallel, then only the execution time $(1 - \sigma) * T_1$ can be taken to execute those $p$ parallel subtasks on $p$ processors [23].

Williams and Smith [18] proposed an equivalence between the speedup and
the relative capacity for Amdahl’s Law and they verified the equivalent approximation between them in a case study. Amdahl’s Law can be written based on the relative capacity as follows,

\[ C_A(p) = \frac{p}{1 + \sigma(p - 1)} \]  

The Amdahl’s Law model is expressed as a rational function in terms of \( C(p) = \frac{P(p)}{Q(p)} \), where \( P(p) = p \) and the denominator \( Q(p) \) is a first-degree polynomial. As the number of processors \( p \) increases infinitely and the serial fraction is non-zero \((0 < \sigma < 1)\), the number of decomposed subtasks increases as well and the relative capacity asymptotically approaches the value \( 1/\sigma \). That is, \( \lim_{p \to \infty} C_A = \frac{1}{\sigma} \). Figure 2.7 also fits this situation; the curve can represent Amdahl’s Law and the relative capacity is less than the linear.

If the serial fraction is zero \((\sigma = 0)\), it means no portion of the workload is executed sequentially, then Amdahl’s Law predicts there is an infinite linear relative capacity \((C(p) = p)\).

Based on Amdahl’s Law, Gunther’s USL not only considers the serialization that is referred to as contention, but also the interprocessor communication overhead that is referred to as coherency cost. The USL model is expressed in terms of a rational function as well, \( C(p) = P(p)/Q(p) \), where \( P(p) = p \) is a linear function that models the linear scalability and \( Q(p) \) is a second-degree polynomial. The second-degree polynomial \( Q(p) \) contains two parts: one is derived from Amdahl’s Law, where the serial portion of the workload
is multiplied by the serial fraction $\sigma$, forming $1 + \sigma(p - 1)$; the other part is derived from USL, where the amount of coherency existing in most scalable systems is multiplied by a coefficient $\lambda$, forming $\lambda p(p - 1)$. The combination of the linear function and the second-degree polynomial as the denominator forms the formal USL model [21], and it can be defined as follows,

$$C_U(p) = \frac{p}{1 + \sigma(p - 1) + \lambda p(p - 1)} \quad (6)$$

The USL model explains that a scalable system follows sub-linear scalability, instead of linear scalability, due to contention and coherency that are parameterized by two coefficients $\sigma$ and $\lambda$ respectively. Contention is caused due to the serialization which means the workload cannot be processed in parallel, but in serial. For example, multiple instances of a web application spawned in a cloud system compete for the same CPU resource, and at this time some of them can obtain the CPU time to process requests, but the others have to wait in a queue until the CPU resource is available. Coherency occurs when a system, including different parts, needs to keep a coherent and consistent state, for example the data exchange between processors and the main memory.
The two coefficients, $\sigma$ and $\lambda$, can be interpreted and divided into four cases:

$$C_U(p) = \begin{cases} 
p, & \text{if } \sigma = 0, \lambda = 0; \quad (Case \ 1) \\
p \frac{p}{1+\sigma(p-1)}, & \text{if } \lambda = 0, \sigma>0; \quad (Case \ 2) \\
p \frac{p}{1+\lambda p(p-1)}, & \text{if } \sigma = 0, \lambda>0; \quad (Case \ 3) \\
p \frac{p}{1+\sigma(p-1)+\lambda p(p-1)}, & \text{if } \sigma, \lambda>0. \quad (Case \ 4) \end{cases}$$

- **Case 1**: there is unlimited speedup or relative capacity and follows a linear scalability manner.

- **Case 2**: there is contention-limited scalability and is regarded as a generalization of Amdahl’s Law for speedup or relative capacity; the maximum performance can be bounded only by queueing or serialization. Gunther in his book [21] proves such a case.

- **Case 3**: there is coherency-limited scalability and the maximum performance can be bounded only by the coherency cost; it occurs in an incoherent application state between cluster nodes.

- **Case 4**: USL states such a case and it is the worst case because both contention and coherency limit the achievable relative capacity or speedup. The denominator of the USL model can be algebraically rearranged to a standard form of a quadratic equation, $\lambda p^2 + (\sigma - \lambda)p + 1 - \sigma$. Therefore, it indicates that the coherency term grows as the square of the number of processors and causes the relative capacity $C_U(p)$.
or speedup $S_p(p)$ to reach a maximum and then decrease due to the quadratic characteristic of the coherency term. The specific maximum performance achieved for the number of processors is at $p^* = \lfloor \sqrt{\frac{1+\sigma}{\lambda}} \rfloor$. In other words, the throughput achieves the best at $p^*$, and then beyond $p^*$ the throughput is retrograde.

The independent variable $p$ in Equation (6) can represent the number of virtual users (workload). When the hardware configuration, such as the number of processors, remains fixed and the number of virtual users varies, it is referred to as software scalability [21]. The previous concept and theorem discussions can be extended to software scalability. Moreover, the equation of software scalability is identical to Equation (6). Software scalability can be mostly found in load testing environments. When the USL model is applies to software scalability within an online transaction system, contention is represented by $\sigma$; and coherency is denoted by $\lambda$. Contention refers to the situation where all requests from the client are launched simultaneously, and consequently these concurrent requests join the queue to wait to be processed by the server. On the other hand, coherency is referred to as interprocess communication overheads. When forking multiple processes in the system, it causes some communication overhead among processes.
2.6 Related Work

Lei et al. conducted benchmark tests and scenario tests to compare the performance of Node.js, Python-web and PHP [24]. Their tests followed a one-factor-at-a-time manner; they varied the number of concurrent clients and fixed the number of requests. They utilized the test tools ApacheBench (ab) [25] and LoadRunner [26] to make concurrent request loads on three test applications, Hello World, Fibonacci Calculation and Select Operation of DB, and two scenarios, Login and Encryption. They measured response time and throughput, and concluded Node.js significantly outperformed Python and PHP. This project will adapt their one-factor-at-a-time experimental design to the proposed work, investigating scalability in the cloud by varying only one factor and fixing other factors. However, this project will extend their work to cloud systems, focusing on Node.js application scalability issues.

Aronis et al. developed a scalability benchmark suite for Erlang/OTP, Bench Erl [27]. A set of synthetic benchmarks that measure the scalability of a specific aspect of Erlang and three real-world benchmarks are included in the benchmark suite. The architecture of the benchmark suite can be divided into four components, the coordinator, the executor, the sanity checker and the graph plotter. The coordinator finds out what and how to execute, by reading configuration files; the executor is responsible for executing a specific benchmark in a particular runtime environment; the sanity checker verifies the output produced by the benchmarks; and the graph plotter processes
the scalability measurements and visualizes the benchmarking results. The scalability-oriented benchmark suite is partially inspired by this design.

Patros et al. introduced Cloud Burners, a set of Java EE cloud tenants that target various specific basic hardware resources [28]. Cloud Burners can be classified into CPU Burner that targets CPU resource, Cache Burner that targets system’s cache and yields CPU, Resident Memory Burner that targets resident memory and yields CPU, Disk I/O Burner that targets disks I/O, and Network I/O Burner that targets network I/O. These Cloud Burners can be deployed in a cloud system as cloud tenants. Each Cloud Burner tenant has demand for a specific resource, which results in starvation to other tenants. Cloud Burners are utilized to propose a set of resource-slowdown and resource-intensiveness metrics. Similarly, test applications in the proposed benchmark suite are designed to target and stress specific resources. However, unlike Node.js, Java is multi-threaded and can directly leverage multi-cores.

Williams and Smith [18] presented a model-based approach that aims at scalability in web and distributed applications. Their paper reviewed four models of scalability: Linear, Amdahl’s Law, Super-Serial, and Gustafson’s Law. They also demonstrated a case study that a web application applied these models. When they demonstrated the vertical scaling characteristics of the web application, throughput versus workload was measured for configurations of one, two, four, and eight processors. They fitted the data throughput to the Amdahl’s Law Model and determined a $\sigma$ of 0.0881 ($R^2 = 0.992$); they
also fitted it to the Super-Serial Model and determined a $\sigma$ of 0.0787 and a $\lambda$ of 0.0164 ($R^2 = 0.993$). Based on the regression analysis of vertical scaling, they concluded vertical scalability is best fitted by either Amdahl’s Law or the Super-Serial Model and there is little difference between models. In addition, they also demonstrated the horizontal scaling characteristics of the web application by measuring throughput versus workload for configurations of one, two, three, and four nodes. They fitted the Gustafson’s Law model to the measured data and calculated $\sigma'(1) = 0.0477$ (approximately 4.8% of the work with one processor is performed sequentially) and $R^2$ of 0.9999; they also fitted the linear model to the measured data and determined the slope of the regression line is 1.039. Based on the regression analysis, they concluded horizontal scalability is best described by either Gustafson’s Law or Linear scalability. Further, Tsai et al. [29] proposed scalability metrics that can be utilized to test the scalability of SaaS applications in the cloud. The theoretical scalability metrics included, 1) processing time ($T$): reflects the traditional speedup, 2) resource consumption ($T \ast T_r$): defines the resource usage in the system, 3) performance resource ratio (PRR): reflects the relationship between performance and the resource consumption, and 4) metric variance. The cloud system is complex and metrics collected from each test run are varied, therefore, metric variance is defined as the standard variance of the performance metric in multiple runs of the same workload. Additionally, a scalability testing methodology was proposed. They briefly proposed data mining techniques that can be utilized to assist the scalabil-
ity testing. Also, Schwartz [30] described an approach to model scalability and performance using USL by observing system behavior and estimating coefficients. The author performed the load test on a system and collected the data throughput, response time or their combination. Afterwards, the dataset was fitted to the USL, using non-linear least squares regression. Finally, coefficients were determined. Based on these works, this thesis utilizes similar scalability models and performs regression analysis to theoretically model Node.js scalability.

2.7 Summary

The background chapter introduced the reader to cloud systems including a PaaS cloud and an instance of the PaaS cloud, Docker Swarm. The PaaS cloud provides a ready-to-use platform, in which users can deploy and maintain their applications. In addition, the PaaS cloud also allows horizontal scaling meaning multiple instances of a single application can be created. This chapter also introduced Node.js features. Node.js is single-threaded and it results in under-utilizing the CPU resource in a multi-core environment. However, Node.js is scalable by applying either its built-in module `cluster` or scaling solutions provided by the PaaS cloud. Moreover, this chapter presented the scalability concept and several models, the Amdahl’s Law model and USL model. Finally, related work was also introduced to readers. The next chapter will raise the problem statement and accordingly propose
approaches in this research project.
Chapter 3

Problem Refinement

This chapter describes the design of the research, starting by giving the problem statement in Section 3.1. Section 3.2 proposes approaches for this project. Section 3.3 describes the development environment.

3.1 Problem Statement

Node.js is scalable, because it can utilize two existing scalability strategies, either the \textit{cluster} module or a PaaS cloud’s horizontal scaling service, to improve scalability, as discussed in Section 2.3.5. The core idea behind those scalability strategies is creating multiple processes to fully take advantage of multi-core systems. This project will evaluate and compare those aforementioned scalability strategies.

Scalability has to be considered in the cloud system in order to support a
higher workload. This project will also perform a study of the scalability of Node.js in the cloud by varying different variables, such as workloads, and the number of spawned instances. In addition, Node.js web applications running in the cloud have to share the node’s limited resources, such as CPU, RAM, disk, network bandwidth, etc.; this project will measure their performance and detect the underlying resource bottlenecks that degrade the performance. In summary, the following research problems will be investigated in this project:

- Identify horizontal scaling characteristics (e.g. linear, sub-linear, and super-linear scalability) of Node.js CPU-intensive and network-intensive applications in the cloud.

- Quantify and assess Node.js CPU-intensive and network-intensive application scalability in the cloud by best fitting several existing mathematical scalability models.

- Detect and study performance bottlenecks of CPU-intensive and network-intensive test applications.

- Compare two scalability strategies applied to cloud systems, horizontal scaling and the `cluster` module of Node.js for CPU-intensive and network-intensive applications.
3.2 Approach

The work in this project can be divided into two parts, scalability-oriented benchmark suite development and scalability investigation utilizing this benchmark suite.

- **Scalability-oriented benchmark suite**: a set of test applications as benchmarks and the related benchmark infrastructure are developed to form a scalability-oriented benchmark suite. These test applications are implemented in JavaScript on Node.js and can be utilized to benchmark different scalability strategies, such as the `cluster` module and horizontal scaling. Scalability-oriented metrics are collected by this benchmark suite and they can be utilized to perform an investigation of scalability. More details regarding this benchmark suite are presented in Chapter 4. Once this benchmark suite is developed, the second part of this project can be launched.

- **Node.js scalability investigation in the cloud**: several scalability experiments will be conducted within a private cloud system. Afterwards, this project processes the scalability investigation. The investigation can be divided into three parts:
  
  - Identify the scalability characteristics of horizontal scaling: we explore several scalability models and fit these scalability models to a dataset from scalability experiment results, using nonlinear
least squares regression. The optimal coefficient values can be determined with a best-fit line or curve. By observing and calculating the R-squared value, we can find the value of fit to identify scalability types.

- Identify the underlying performance bottlenecks: the benchmark suite collects resource consumption data that can be utilized to find the performance bottlenecks.

- Compare two scalability strategies: the Cluster module and horizontal scaling strategies will be benchmarked and analyzed.

3.3 Development Environment

The development environment can be divided into hardware environment and software environment. Subsection 3.3.1 presents the hardware environment, such as server CPU, RAM information; Subsection 3.3.2 shows all software information involved in this development, including software versions.

3.3.1 Hardware

Five physical servers are utilized to build a Docker Swarm and they play manager and worker roles. The hardware and software environment are summarized in Table 3.1. Each server connects to the Centre for Advanced Studies-Atlantic (CASA) laboratory network and is allocated a static IP address. All servers have a multi-core environment suitable to investigate
Node.js scalability. Multiple Node.js instances can be spawned and each of them utilizes one core, and we can see scalability effects or none on Node.js applications.

### 3.3.2 Software

A number of software packages are utilized to implement a scalability-oriented benchmark suite and build a private cloud environment in this project. They are programming language runtimes, related modules, private cloud infrastructure and so on. The summary is shown in Table 3.2.

### 3.4 Summary

In summary, this chapter raised problem statements for the whole research project: 1) what the scalability characteristic (linear or sub-linear scalability) of Node.js in the cloud is and what affects scalability; 2) what the performance bottlenecks are; and 3) how two different scalability strategies, the HS and CM, applied in Node.js affect scalability and which one achieves good performance. To resolve these problems, this chapter proposed approaches: 1) develop a scalability-oriented benchmark suite; and 2) investigate Node.js scalability in the cloud using the developed benchmark suite. Finally, this chapter introduced the research project development environment, including hardware and software. The next chapter will present the detailed implementation of the benchmark suite.
<table>
<thead>
<tr>
<th>Host Name</th>
<th>CPU</th>
<th>RAM</th>
<th>OS Version</th>
<th>Role</th>
</tr>
</thead>
<tbody>
<tr>
<td>casaloaner1</td>
<td>Intel® Core™ i5-2400 CPU @3.10GHz</td>
<td>8GB</td>
<td>Ubuntu 14.04.5 LTS</td>
<td>Worker</td>
</tr>
<tr>
<td></td>
<td>4 Cores (1 thread per core)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>casahost2</td>
<td>Intel® Core™ i5-2400 CPU @3.10GHz</td>
<td>8GB</td>
<td>Ubuntu 14.04.5 LTS</td>
<td>Worker</td>
</tr>
<tr>
<td></td>
<td>4 Cores (1 thread per core)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>casahost3</td>
<td>Intel® Core™ i5-2400 CPU @3.10GHz</td>
<td>8GB</td>
<td>Ubuntu 14.04.5 LTS</td>
<td>Worker</td>
</tr>
<tr>
<td></td>
<td>4 Cores (1 thread per core)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>casa6</td>
<td>Intel® Core™ i7-2600 CPU @3.40GHz</td>
<td>8GB</td>
<td>Ubuntu 14.04.5 LTS</td>
<td>Worker</td>
</tr>
<tr>
<td></td>
<td>4 Cores (2 threads per core)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>casa16</td>
<td>Intel® Core™ i7-3770 CPU @3.40GHz</td>
<td>8GB</td>
<td>Ubuntu 14.04.5 LTS</td>
<td>Manager</td>
</tr>
<tr>
<td></td>
<td>4 Cores (2 threads per core)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>casa20</td>
<td>Intel® Core™ i7-3770 CPU @3.40GHz</td>
<td>8GB</td>
<td>Ubuntu 14.04.5 LTS</td>
<td>Worker</td>
</tr>
<tr>
<td></td>
<td>4 Cores (2 threads per core)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 3.2: Software Environment

<table>
<thead>
<tr>
<th>Software Name</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node.js</td>
<td>v6.12.3</td>
<td>A JavaScript runtime</td>
</tr>
<tr>
<td>Docker</td>
<td>v17.06.0-ce</td>
<td>It is utilized to build a private cloud</td>
</tr>
<tr>
<td>Vagrant</td>
<td>v1.9.7</td>
<td>Build and maintain virtual machines</td>
</tr>
<tr>
<td>VirtualBox</td>
<td>v4.3.36</td>
<td>A open-source hypervisor; provides VM</td>
</tr>
<tr>
<td>Express.js</td>
<td>v4.15.3</td>
<td>A Node.js third-party module; a web framework for Node.js</td>
</tr>
<tr>
<td>promise.js</td>
<td>v8.0.1</td>
<td>A Node.js third-party module; it is utilized to handle Node.js’s asynchronous operations</td>
</tr>
<tr>
<td>simple-ssh.js</td>
<td>v1.0.0</td>
<td>Node.js third-party module; it is utilized to establish SSH connections and execute remote BASH commands</td>
</tr>
</tbody>
</table>
Chapter 4

Scalability-oriented Benchmark Suite

This chapter will introduce a benchmark suite created in this project. This benchmark suite is scalability-oriented and contains a set of benchmarks, which is utilized to evaluate and measure different scalability strategies applied in Node.js based on cloud systems. The benchmark suite is built using various components that cooperate with each other to perform benchmarks. Section 4.1 will present an overview of the whole benchmark suite architecture. Section 4.2 and Section 4.3 will introduce some individual benchmark suite components.
4.1 Architecture of Benchmark Suite

The architecture of the benchmark suite follows a two-tier architectural model, which consists of the client side and the server side. The client side launches simulated concurrent requests referred to as workload to the server side. The client side also post-processes measurements. The server side runs benchmark test application as web servers to handle request from the client side. The overview of the benchmark suite architecture is shown in Figure 4.1. The client side consists of an executor component and an analyzer component. Users can configure a benchmark run, fire concurrent requests and launch the benchmark run from the client side. In addition, the client side is responsible for post-processing the captured raw data, such as the resource usage data and load testing results, and generating several relevant scalability-oriented metrics. All of these component codes can be run either in bare metal or in the container from the client side.

The server side contains a benchmark component and a resource collector (RC) component. The benchmark component wrapping test applications are deployed in server side by the PaaS cloud. They run as a web server on the server side, and process concurrent requests when receiving them from the client side. Afterwards, they start to prepare responses to the client side. Meanwhile, the RC component also runs on the server side, executed either in bare metal or in the container, and starts to collect computing resource usage data during the whole benchmark process. Once the benchmark pro-
cess ends, the captured computing resource usage data is sent back to the analyzer component running in the client side. A single entire benchmark run process is listed below:

1. Users deploy the benchmark component to the server cluster according to their configurations. Users also start the RC running on the server side.

2. Users start the benchmark run. The RC starts its initial resource usage data collection and sends various resource consumption data back to the client side. Afterwards, the concurrent requests are generated and sent to the server side.

3. Benchmark component running on the server side accepts these requests until finishing all of them. The RC captures the final resource usage data and sends results back to the client side when all workloads
have been finished and the benchmark ends.

4. The client side extracts, parses and post-processes the raw data. Finally, it generates a report containing scalability related metrics.

4.2 Benchmark Suite Infrastructure

The following subsections will introduce more details about each infrastructure component in this benchmark suite.

4.2.1 Executor Component

The executor runs on the client side, which is designed to launch a complete benchmark run. It is implemented using a set of bash scripts that run various types of benchmarks based on users’ configurations. Basically, the executor has the following responsibilities: 1) Control the entire benchmark process; 2) Maintain the service stack in the server cluster; 3) Generate and fire concurrent requests to the server side; 4) Trigger the RC running on the server side; and 5) Handle and process the raw data, and generate the final report. The workflow of the executor is displayed in Figure 4.2.

The executor allows users to configure benchmark parameters, such as running time, concurrency of workloads, server IP addresses, requested endpoints, load balancer options, and server port numbers. These parameters will be utilized by JMeter workload generation, benchmark launch and Docker service stack maintenance. The description and summary for each
Figure 4.2: Workflow of the Executor

parameter is listed in Table 4.1. This benchmark suite runs on top of Docker in swarm mode. Therefore, the executor enables users to configure and deploy the benchmark component as a stack of services in the server cluster. The customized configurations of deployment, such as node placement constraints, number of worker processes, resource constraints, etc. will be passed to the "docker-compose.yml" file.

Apart from the deployment configuration by generating the "docker-compose.yml" file, the executor allows users to deploy and remove their specific service stacks. They both are implemented by sending remote Docker commands through SSH. A pair of authentication keys should be generated by users to enable an automatic login to the remote server cluster for the purposes
Table 4.1: Parameters in the Benchmark Summary

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Time</td>
<td>The total time spent running one benchmark</td>
</tr>
<tr>
<td>Workload Concurrency</td>
<td>The number of parallel connections fired to the server side</td>
</tr>
<tr>
<td>URL</td>
<td>It contains server IP, port, and endpoint, which is used to access the server</td>
</tr>
<tr>
<td>Load Balancer Option</td>
<td>Nginx or Docker Swarm default load balancer can be selected. If Nginx is selected, then an Nginx configuration file &quot;nginx.conf&quot; should be set up and a related Nginx service will be configured in the “docker-compose.yml” file as well</td>
</tr>
<tr>
<td>Log File Location</td>
<td>Users should specify both the file path to JMeter log files and test plans, and final report files</td>
</tr>
<tr>
<td>Node Placement</td>
<td>It specifies the nodes that involve the benchmark run from the server cluster. It can be setup to either a single one or multiple. It is configured in the “docker-compose.yml” file and takes effect when users deploy their test applications</td>
</tr>
<tr>
<td>Number of Worker Processes</td>
<td>It specifies the number of worker processes running the Node.js test applications. The worker processes can be forked in a single Docker container by the Cluster module. It is configured in “docker-compose.yml” file as an environment variable</td>
</tr>
<tr>
<td>Recourse Constraints</td>
<td>Users can constrain the resource usage to the containers, such as the number of virtual CPU cores and memory. It takes effect when users deploy their applications to the server cluster. It is configured in “docker-compose.yml” file</td>
</tr>
<tr>
<td>Number of Docker containers</td>
<td>It specifies the replica number of Docker containers running Node.js test applications in the server cluster. It is configured in “docker-compose.yml” file</td>
</tr>
</tbody>
</table>

of security. For the deployment operation, the deployment commands will be sent and executed on the server side remotely, and a status code will be returned. If the status code is less than 0, then it will re-deploy the service stack until the status code is greater than 0. An operation that checks the number of service replicas is also performed to ensure all services are de-
Algorithm 1 Deploy Service Stack

function DEPLOY_STACK(num_of_containers)
    deployment_counter, check_counter ← 0
    status_code ← Deploy Service Stack
    while status_code < 0 do
        deployment_counter ++
        if deployment_counter > LIMITE then
            exit 1
        end if
        sleep 2
        status_code ← Deploy Service Stack
    end while
    sleep 5
    stat_num ← CheckNumberOfDeployed Services
    while stat_num < num_of_containers do
        check_counter ++
        if check_counter > LIMITE then
            exit 1
        end if
        stat_num ← CheckNumberOfDeployed Services
    end while
end function

Algorithm 2 Remove Service Stack

function REMOVE_STACK
    status_code ← Remove Service Stack
    while status_code < 0 do
        sleep 2
        status_code ← Remove Service Stack
    end while
end function

deployed successfully. Algorithm 1 is shown to describe such a procedure. The remove operation has a similar procedure with the deployment operation and its algorithm is shown in Algorithm 2.
Once users complete the service stack deployment on the server cluster, the executor triggers data collections, implemented by sending a single HTTP request with a GET method using the `curl` command. The `curl` tool is utilized to transfer data from or to a server using various protocols (FTP, HTTP, HTTPS, etc.)[31]. The data collection procedure has three operations: a handshake with the RC, the first data collection and the second data collection. Each operation can be performed by accessing its associated endpoint. For example, the endpoint `/startperf?flag=0` refers to trigger the first data collection; the endpoint `/startperf?flag=1` refers to trigger the second data collection; and an empty endpoint without passing any values refers to the handshake operation. The endpoint is attached to a URL and a sample URL is shown as, “curl -s managerIP:port/startperf?flag=0”. The flag “-s” enables a silent mode that does not show the progress or error messages. The RC will return an “OK” flag, when each operation is processed successfully, otherwise, the specific operation failed to be executed. When the failure happens, the executor will repeat the operation until a counter reaching a specific number and terminating the benchmark run. When the second data collection finishes successfully, the resource usage data in JSON format is returned and is processed by the analyzer later.

The executor starts the load testing tool, JMeter, to generate and fire several concurrent requests to the server side. This benchmark suite utilizes the non-GUI mode, because it is easy to adapt and wrap to the bash script to be executed. We create a test plan file that defines a thread group, a
HTTP request sampler and a listener in this benchmark suite. Several user-defined variables are also declared in the test plan file, such as the number of threads and duration under the thread group element, and server IP, port and path under the HTTP request sampler element. These variables will be assigned to specific values under the non-GUI mode. We configure several fields that are reported in a JMeter log file, including latency, code, timestamp, etc. We execute a command to start the JMeter in the non-GUI mode, and Listing 4.1 shows such a command with several command line options.

Listing 4.1: Run JMeter Using Non-GUI Mode

```
jmeter -n -t test_plan_path -l jmeter_output_path -
   JnThreads=concurrency -Jtime=duration -Jip=server_ip
   -Jport=server_port -Jpath=url
```

The flags starting with “-J” are user-defined variables in the test plan, and they can be assigned from this command. Therefore, the variable “-JnThreads” matches the benchmark parameter “workload concurrency”; “-Jtime” matches the “running time”; the combination of the “-Jip”, “-Jport” and “-Jpath” matches the benchmark parameter “URL”. Apart from the flags in terms of user-defined variables, the flag “-n” specifies the non-GUI mode; “-t” specifies the location of the test plan; “-l” allows users to locate the load testing result. Once JMeter finishes a load testing, a final report will
be generated and stored in the specified location and it will be post-processed by the analyzer later.

After the executor finishes the load testing and the second-time resource usage data collection, the executor triggers the analyzer component to parse and post-process the raw data. The JMeter final report and resource usage data in JSON format are passed to the analyzer component by the executor. Finally, the benchmark run final report is printed out. The final report is CSV format and each record is in one physical line. The schema of the final report is shown in Figure 4.3. Figure 4.4 presents a sample final report. In this sample final report, a benchmark runs three times, and five concurrent requests are issued to the server running a CPU-intensive test application and the number of Node.js instances are varied from one to three.

4.2.2 Resource Collector Component

The RC component runs on the server side and is designed to take a snapshot of resource usage before and after a benchmark run. The RC component is coded using the JavaScript programming language that runs in the Node.js framework, thus it uses the asynchronous programming style and an event-driven model that derives from Node.js features. It can be deployed and
run in any node of the server cluster, but it usually runs in the manager node as a web server, publishing an open port and waiting for requests. The executor component can access it by sending a HTTP request discussed in the previous section, in turn, the RC component captures the snapshot of current resource usage data and then responds to the executor component with collected data.

The RC component collects various types of data from each worker node participating in the swarm. Therefore, multiple remote SSH connections are established to execute a set of shell commands remotely, which is achieved by a Node.js module "simple-ssh". The "simple-ssh" module wraps an "ssh2 client" module and allows users to run a sequence of commands over SSH. The "simple-ssh" module enables users to make SSH connections to remote nodes by creating SSH objects, called "ssh"; insert one or multiple shell
commands into a queue by calling function “ssh.exec()”; execute all shell commands stored in the queue by calling function “ssh.exec().start()”. All the SSH related operations are processed asynchronously in Node.js, therefore, a “promise” module is also utilized to maintain these asynchronous operations. A “promise” represents the results of asynchronous operations wrapped in callback functions, and it maintains two basic states, fulfilled and rejected. The fulfilled state means a successful operation; the rejected state is referred to as a failed operation. Two different decisions, either executing the next asynchronous operations or error handling, can be made according to the state. In the RC component, all asynchronous operations are wrapped into promises. The sequence of building an SSH connection, inserting shell commands to the queues, and executing shell commands, as promises applies the following control flow:

```javascript
buildSSHConn(ssh_ip)
  .then(() => {return enqueueSSHCmd(ssh_ip, cmd)})();
  .then(() => {return execSSHCmd()})();
  .then(() => { // handle the final result })
  .catch(() // error handling);
```

Before the data is collected from each worker node, the RC component parses and extracts worker node IP addresses and hostnames from a JSON file that works like a DNS service that stores worker node information in a key-value pair format. The worker node hostname is the key and its IP address is
the value. This JSON file is created by end users. The RC component captures the container runtime information as well, such as container IDs, and its process IDs. Docker provides CLI to check such information, for instance, “docker ps -f status=running –format {{.ID}}” can be used to find all running containers’ ID; “docker inspect –format {{.State.Pid }} {{.Id}} $ContainerID” can be used to find process IDs of specified containers. The container runtime information will be utilized to capture resource usage of containers.

The RC component collects the CPU usage, the number of disk I/O operations, the network traffic bytes, and the memory usage. Docker allows users to obtain resource usage metrics by either executing the Docker CLI “docker stats” or relying on cgroups (pseudo-filesystem), referred to as “cgroupfs”. Users can execute the command “docker stats” to live stream a particular container’s resource usage metrics. The way our RC collects resource usage metrics is by accessing cgroupfs, or executing shell commands to check. The reason is 1) our RC is designed to capture a snapshot of resource usage metrics instead of the real-time data; 2) “cgroupfs” works not only on Docker containers, but also LXC containers, which is more generic and easy to extend the RC’s functionality in the future; and 3) accessing “cgroupfs” has lower overheads due to executing system calls than executing the command “docker stats”. Section 2.2.2 introduced cgroups that expose metrics about CPU, memory, network and block I/O usage of a particular container through the pseudo-filesystem. The details for each resource usage collection
is as follows:

- **CPU usage:** the RC component finds the CPU usage in terms of accumulated CPU usage under the pseudo-filesystem, “/sys/fs/cgroup/cpuacct/docker/⟨containerID⟩/cpuacct.stat”. The accumulated CPU usage metric contains a user time and a system time that is expressed in ticks of 1/100 second (user jiffies). The user time is defined as the time that the processes are in direct control of the CPU; the system time refers to the time that the processes using the CPU execute system calls.

- **Disk:** the disk usage refers to the number of bytes read and written from the disk by block I/O operations. They are found under “/sys/fs/cgroup/blkio/docker/⟨containerID⟩/blkio.throttle.io_service_bytes”.

- **Network:** the Docker official documentation [32] clarifies that when we capture network metrics, network usage data cannot be directly collected from cgroups. The reason is that the network interfaces exist within the context of network namespaces and the processes in a single cgroup belong to multiple network namespaces. That causes a single cgroup of process mapping to multiple network interfaces, such as lo, eth0, etc., which is difficult to interpret data from cgroups. However, as the documentation shown, there is a way to run an executable from the host environment within a network namespace of
a container using the shell command “ip-netns”. In this way, the host can access the network namespace of the container and capture the incoming and outgoing network bandwidth data of the specified container. The shell command “ip netns exec ⟨containerID⟩ netstat -i” implements that. The shell command “ip netns” finds a specific container under “/var/run/netns/container” that belongs to one of the namespace pseudo-files “/proc/⟨pid⟩/ns/”. Therefore, before executing “ip netns”, a symlink should be built first by using “ln -sf /proc/⟨pid⟩/ns/net /var/run/netns/⟨containerID⟩”, which ensures a correct network namespace of the container can be entered by the host.

In summary, two shell commands can be executed to capture network bandwidth data, “ln -sf /proc/⟨pid⟩/ns/net /var/run/netns/⟨containerID⟩” and “ip netns exec ⟨containerID⟩ netstat -i”. Finally, the network data in bytes, transmitted (TX) and received (RX), of each network interface can be collected.

- Memory: Node.js runs on the V8 engine that compiles the JavaScript code into machine language and executes it. V8 also manages the allocation and freeing of memory during execution. V8 divides the memory into three segments: 1) code segment (stores all code to be executed); 2) stack segment (stores all local variables with pointers referencing objects on the heap); and 3) heap segment (contains all objects referenced by pointers on the stack). The combination of all four memory segments form a memory block called the resident set.
The size of the resident set is called resident set size (RSS), which represents the total memory allocated in a single Node.js application. The RC component collects the metric RSS that shows the amount of RAM allocated to a container process. RSS can indicate RAM usages of Node.js test applications running in containers. The shell command “ps -o rss ⟨pid⟩” can be executed to capture RSS. The “⟨pid⟩” is associated with running containers. RSS is collected periodically, because this is a real-time metric. Garbage collection (GC) usually involves dynamic memory management to free memory following a particular GC policy. This benchmark suite does not report the GC effects but the overall memory usage RSS. The JavaScript function setInterval() enables a function that is passed to as an argument to be executed at specified intervals. The RSS collection is implemented using such a function.

The summary of resource usage data collection is listed in Table 4.2. In the process of first-time data collection, the RC writes the captured snapshots to a JSON file and stores it on disk, afterwards it responds to the client side with a flag “OK”; moreover, when the RC is triggered to perform the second-time data collection, it captures the required data; reads the JSON file stored on disk; converts the contents in the JSON file to a JSON object; adds the captured data at this data collection round to the JSON object; and finally, responds to the client side with this JSON object. The file writing and reading operations are involved, which utilize the Node.js synchronous functions such as, writeFileSync(), readFileSync(). The JSON file I/O operations are
Table 4.2: Summary of Resource Usage Collection

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Collect from cgroup pseudo-filesystems, /sys/fs/cgroup/cpuacct/</td>
<td>System time and user time in jiffies</td>
</tr>
<tr>
<td></td>
<td>docker/⟨containerID⟩/cpuacct.stat</td>
<td></td>
</tr>
<tr>
<td>Disk</td>
<td>Collect from cgroup pseudo-filesystems, /sys/fs/cgroup/blkio/</td>
<td>Total bytes of read and write operations from the disk</td>
</tr>
<tr>
<td></td>
<td>docker/⟨containerID⟩/blkio.throttle.io_service_bytes</td>
<td></td>
</tr>
<tr>
<td>Network</td>
<td>Cannot be captured directly from cgroups. Instead run an executable from the host within a network namespace of a container to extract data, using shell command “ip netns exec ⟨containerID⟩ netstat -i”</td>
<td>RX and TX in bytes for each interface mapping to containers</td>
</tr>
<tr>
<td>Memory</td>
<td>Execute shell commands “ps -o rss ⟨pid⟩”</td>
<td>Resident Set Size (RSS), includes all stack and heap memory allocated in RAM</td>
</tr>
</tbody>
</table>

performed in the manager node and isolated from the entire benchmark run, therefore, there are few overheads in a benchmark run.
4.2.3 Analyzer Component

The analyzer component is responsible for post-processing collected data, which is implemented using Node.js JavaScript as well. There are two data sources as inputs that are passed by the executor component. One is resource usage data that is collected from the RC component; the other one is load testing results that are generated from JMeter. The original resource usage data is formatted in JSON and its data structure is presented in Figure 4.5. The

```
{
  "one":
  {
    "cpu": ["192.168.1.185": ["user 22 system 3"],
            "192.168.1.179": ["user 25 system 2"]],
    "disk": ["192.168.1.185": ["Total 0"],
             "192.168.1.179": ["Total 0"]],
    "net": ["192.168.1.185": ["Kernel Interface table eth0
      RX bytes:0 (0.0 B)
      TX bytes:0 (0.0 B)
    ],
            "192.168.1.179": ["Kernel Interface table eth0
      RX bytes:0 (0.0 B)
      TX bytes:0 (0.0 B)
    ]],
  
  "two":
  {
    "cpu": ["192.168.1.185": ["user 4363 system 2809"],
             "192.168.1.179": ["user 4428 system 3114"]],
    "disk": ["192.168.1.185": ["8:10 Read 4096 8:10 Write 120414208 8:0
        Sync 0 8:0 Async 120418304 8:0
        Total 120418304 Total 120418304"],
             "192.168.1.179": ["8:10 Read 0 8:0 Write 128077824 8:0
        Sync 0 8:0 Async 128077824 8:0
        Total 128077824 Total 128077824"],
    "net": ["192.168.1.185": ["Kernel Interface table eth0
      RX bytes:22567774 (22.5 MB)
      TX bytes:31340947 (31.3 MB)
    ],
            "192.168.1.179": ["RSS 37532"],
    "rss": ["0": ["192.168.1.185": ["RSS 36684"],
                  "192.168.1.179": ["RSS 36684"]],
            "1": ["192.168.1.185": ["RSS 75604"],
                   "192.168.1.179": ["RSS 75604"]],
            "2": ["192.168.1.185": ["RSS 76072"],
                   "192.168.1.179": ["RSS 76072"]],
            ...],
    "interval": 108.103,
    "placement": ["worker3"]
  }
```
<table>
<thead>
<tr>
<th>Type</th>
<th>Metric</th>
<th>Formula</th>
</tr>
</thead>
</table>
| CPU  | CPU usage percentage (%)      | \[
|      | \[total\_{time\text{first}} = \text{user\_time\text{first}} + \text{system\_time\text{first}}\][\text{total\_{time\text{second}} = \text{user\_time\text{second}} + \text{system\_time\text{second}}}\]
|      | \[CPU\_Usage\% = \frac{\left(\text{total\_{time\text{second}} - total\_{time\text{first}}}\right)}{\left(\text{Hertz} \times \text{time\_interval}\right)} \times 100\] |
| Disk | Disk throughput (MB/sec.)     | \[DiskT. = \frac{\left(\#\text{ofIOops\_{second}} - \#\text{ofIOops\text{first}}\right)}{\text{time\_interval}}\] |
| Network | Network Bandwidth (MB/sec.)  | \[RXT. = \frac{\left(\text{RXinMB\_{second}} - \text{RXinMB\text{first}}\right)}{\text{time\_interval}}\]  \\
|      |                              | \[TXT. = \frac{\left(\text{TXinMB\_{second}} - \text{TXinMB\text{first}}\right)}{\text{time\_interval}}\] |
| Memory | Average Resident Set Size (MB) | \[\text{AvgRSS} = \frac{\text{totalRSS}}{n}\] |

JSON file contains results of two data collections and the key names are “one” and “two” for each data collection respectively. The total benchmark execution time, RSS data collected in a period of specific seconds, and container placement among nodes information are attached to the end of the file. The rest of the data is resource usage in the key-value pair format. The worker node IP addresses are keys and they map to various types of resource usage. The analyzer component parses JSON files and extracts diverse types of resource usage data from all worker nodes, and calculates the relevant resource usage metrics according to the formula. The formulas involved are listed in Table 4.3. The executor runs JMeter in non-GUI mode and this mode does not produce a summary report, but a log file containing all request information. The analyzer component post-processes this
log file and determines the relevant metrics. This JMeter load testing log file is generated from the JMeter listener component. It is a JTL file and JTL is also the default extension. The JTL file as JMeter load testing reports is formatted in CSV with headers. Each line represents a single request and each column of the request is separated by commas. There are sixteen columns and we only consider the columns: 1) time stamp; 2) response code (e.g. 200 indicates a successful request); and 3) latency (the time from just before sending the request to just after the first response has been received [33]). Node.js has a built-in module “ReadLine” that provides an interface for reading data from a Readable stream one line at a time [34]. The analyzer reads the JTL file line-by-line by utilizing the “ReadLine” module. Each line can be split into an array of sub-lines by a delimiter comma using a JavaScript function “split()”. The function “split()” is from String class. The array of sub-lines is for easy data extraction purposes and the specific column data can be fetched by specifying an index of the array, for example, if we extract the response code from a single request, then we just specify its index of the array. The data processing implementation follows the asynchronous programming style, using the promise module. The analyzer extracts and parses data from the JTL file and calculates the metrics response time and throughput. Algorithm 3 shows the whole procedure to determine these two metrics. If the response code is “200”, it means the current request is valid and expected, otherwise it is an invalid request. The analyzer component parses the JMeter report and counts the total number of valid requests, and finally divides it by the time elapsed to determine the throughput of this benchmarking run. To calculate the response time of the current benchmark execution, the analyzer component sorts all latency data.
Algorithm 3 JMeter Load Testing Report Processing

\[
\begin{align*}
&\textbf{count} \leftarrow 0 \\
&\textbf{minTS} \leftarrow -1, \textbf{maxTS} \leftarrow -1, \textbf{ts} \leftarrow -1 \\
&\textbf{resp} \leftarrow [] \\
&\textbf{function} \ \text{READ\_DATA}(\text{report\_path}) \\
&\quad \textbf{while} \ \text{to the end of line of the file do} \\
&\quad \quad \textbf{dataArr} \leftarrow \text{line\_split}(,) \\
&\quad \quad \textbf{if} \ \textbf{dataArr}[\text{indexOfStatus}] \leftarrow 200 \ \textbf{then} \\
&\quad \quad \quad \textbf{resp}[\text{count}++] \leftarrow \textbf{dataArr}[\text{indexOfRT}] \\
&\quad \quad \quad \textbf{ts} \leftarrow \textbf{dataArr}[\text{indexOfTimestamp}] \\
&\quad \quad \quad \textbf{if} \ \textbf{count} \leftarrow 1 \ || \ \textbf{ts} < \textbf{minTS} \ \textbf{then} \\
&\quad \quad \quad \quad \textbf{minTS} \leftarrow \textbf{ts} \\
&\quad \quad \quad \textbf{end if} \\
&\quad \quad \textbf{if} \ \textbf{count} \leftarrow 1 \ || \ \textbf{ts} > \textbf{maxTS} \ \textbf{then} \\
&\quad \quad \quad \quad \textbf{maxTS} \leftarrow \textbf{ts} \\
&\quad \quad \quad \textbf{end if} \\
&\quad \textbf{end if} \\
&\quad \textbf{end while} \\
&\textbf{end function} \\
\end{align*}
\]

\[
\begin{align*}
&\textbf{function} \ \text{CALCULATE\_THROUGHPUT} \\
&\quad \textbf{throughput} \leftarrow 1000 \times \text{count} / (\text{minTS} - \text{maxTS}) \\
&\quad \textbf{return} \ \text{throughput} \\
&\textbf{end function} \\
\end{align*}
\]

\[
\begin{align*}
&\textbf{function} \ \text{CALCULATE\_RT} \\
&\quad \textbf{sort}(\text{resp}) \\
&\quad \textbf{n} \leftarrow \text{resp\_length} \\
&\quad \text{resp95} \leftarrow \text{resp}[0.95 \times \text{n}] \\
&\quad \text{resp100} \leftarrow \text{resp}[\text{n}] \\
&\quad \textbf{return} \ [\text{resp95}, \text{resp100}] \\
&\textbf{end function} \\
\end{align*}
\]
in descending order and finds the latency in the 95th percentile out of the total number of latency data.

Once the analyzer component finishes parsing, extracting and post-processing operations on different data sources, final metrics are determined, including several scalability related metrics and container placement information. Figure 4.6 describes the schema of metrics. Each column is separated by a comma; the resource consumption data is from column 1 to 5 and the load testing result is from column 6 to 8. The rest of the columns are container placement.

4.3 Benchmarks

The benchmark component is the main part of the whole benchmark suite, which contains a benchmark server and a set of test applications implemented in Node.js JavaScript. The benchmark server with test applications can be treated as a black box and is deployed in swarm worker nodes based on users’ configurations. The deployed benchmark component acts as a web server, waiting and processing workloads from the client side. The benchmark server utilizes a third-party module, Express. It is a Node.js web application framework that provides a set of features for web applications [35]. A routing is a basic feature for a web application, which enables a request to connect and communicate to the server’s endpoints from the
client side. Just as Section 4.2.1 discussed, the load testing tool JMeter launches HTTP requests from the client side to the server side. In turn, the benchmark server needs to handle HTTP requests from the client side, which is implemented by utilizing a method of the Express module app object that corresponds to HTTP methods. For instance, the app.get() method handles HTTP’s GET requests, and the app.post() method processes HTTP’s POST requests. These routing methods of the Express module have two arguments passed, one is a defined endpoint, and the other one is a callback function. When the benchmark server receives a request to a specific endpoint from clients, the callback function is executed. In this case, the callback function corresponds to a specified test application, and the test application runs once a request accesses its associated endpoint. Listing 4.2 shows a code snippet of routing methods with endpoints and callback functions.

The benchmark component deployment is highly configurable, and its deployment configuration is based on a Docker compose file “docker-compose.yml”. Its deployment depends on different scalability strategies. The benchmark server that handles various types of requests is viewed as a service in the Docker’s perspective. A specific number of duplicates of a benchmark service can be created, and users can control the placement of these services as well, which can be specified in the “docker-compose.yml”. Listing 4.3 describes partial configurations for specifying the number of replica of benchmark services and their placement constraints. In this sample, there are five benchmark service (named “ibench”) replicas deployed in nodes excluding the manager node and the worker1 node. Each benchmark service runs in a single Docker container. This configuration corresponds to the horizontal scaling strategy.
Listing 4.2: Code to Routing Methods

```javascript
const app = require('express')();

// process CPU-intensive test application
app.get('/getcpubound', cpuBound.isPrime);

// process disk-intensive test application
app.get('/getdiskbound', diskBound.diskIntensive);

// process network-incoming test application
app.post('/gethelloworldbound', netInBound.netInIntensive);

// process network-outgoing test application
app.get('/getnetoutbound', netOutBound.netOutIntensive);

// process memory-intensive test application
app.get('/getcachehitbound', cacheHitBound.cacheHitIntensive);

...
```

The benchmark component deployment is highly configurable, and different scalability strategies can be configured, based on a Docker compose file “docker-compose.yml”. For example, the benchmark server that handles various types of requests is viewed as a service in the Docker’s perspective. A specific number of duplicates of a benchmark service can be created and users can control the placement of these services as well, which corresponds to the horizontal scaling strategy. Listing 4.3 describes such a configuration. In this sample, there are five benchmark service (named “ibench”) replicas deployed in nodes excluding the manager node
and the \textit{worker1} node. Each benchmark service runs in a single Docker container.

\begin{verbatim}
Listing 4.3: Docker Compose File Sample

... services:  
  ibench: 
    replicas: 5  
    placement: 
      constraints: 
        - node.hostname != manager 
        - node.hostname != worker1 

...
\end{verbatim}

In addition, another scalability strategy can be applied by specifying the number of worker processes running in a single container. Listing 4.4 shows such a configuration in the \textit{docker-compose.yml} file. The number of worker processes is specified by defining an environment variable called \textit{CPU_COUNT}. The benchmark server spawns some particular worker processes according to the value of the environment variable \textit{CPU_COUNT} to handle requests. Section 2.3.5 discussed the \textit{cluster} module can be utilized to spawn worker processes. Apart from the environment variable \textit{CPU_COUNT}, there are three more: the \textit{SERVER_PORT} and the \textit{SERVER.IP} are utilized to assign a benchmark server port and IP respectively; and the \textit{NODE_SERVER_PORT} defines the container’s port mapped to the benchmark server port. The container’s port is an open port and the client can access it through this port. The benchmark server can obtain these environ-
ment variables by calling `process.env.ENV_VAR_NAME`; the environment variable name is specified behind the `process.env`.

Listing 4.4: Environment Variable in Docker Compose File

```yaml
... services:
  ibench:
    environment:
      - CPU_COUNT=0
      - SERVER_PORT=9000
      - NODE_SERVER_PORT=9001
      - SERVER_IP=0.0.0.0
...```

The test applications in the benchmark component are inspired by *Cloud Burners*, developed by Patros [28]. They are resource-intensive, because such programs stress resource usages in systems and easily causing scaling when reaching certain resource thresholds, for scalability evaluation purposes. In other words, when resource-intensive programs run and reach certain resource thresholds, scaling will be triggered in PaaS clouds. We can observe and analyze scalability effects in PaaS clouds. The test applications are classified into CPU-intensive applications, disk-intensive applications, network-intensive applications and memory-intensive applications. They are designed to target and stress one resource type. The following introduces each resource-intensive test application.

- **CPU-intensive test application**: It is designed to determine whether
Algorithm 4 Primality Test

function isPrime(val)
    num ← \sqrt{val}
    N ← 2
    while N<num do
        if num\%N ← 0 then
            return false
        end if
        N ← N + 1
    end while
    return true
end function

A large integer is a prime or not. It executes a tight loop to process the calculation and does not process any other operations, such as I/O. Users can specify an integer to be tested. By changing the examined value, the number of loop iterations execution can be varied accordingly. In other words, it affects how much a single request stresses the CPU resource. Algorithm 4 shows the procedure of the primality test.

- **Disk-intensive test application**: It writes a number, in terms of current milliseconds, to a file. This file is stored on disk when writing operations complete and the file is closed, which produces disk I/O operations to target disk resources. There are no other operations involved, excluding disk I/O operations. Node.js has the asynchronous and non-blocking I/O feature, therefore, we use promise module to make an appropriate control flow between I/O operations and client response. An asynchronous writing method, fs.createWriteStream(file_path), is utilized to perform the writing operation and it returns a write stream object “ws”. If the writing operation is per-

77
formed successfully, then a promise state “resolved” is returned and then the benchmark server prepares to respond an “OK” flag to the client; otherwise, a promise state “reject” is returned and the benchmark server responds to the client with an error message. Finally, the write stream is terminated by calling “ws.end()” to close the file, and then the content is stored on disk. The initial implementation of the disk-intensive test application writes a small file and does not significantly stress the disk resource. Users can only see the effect of targeting the disk resource, when compared with other types of test applications. In future work, this test application will be extended and realistic.

- **Network-intensive test application:** It targets two types of network traffic: one is the incoming traffic and the other one is the outgoing traffic. The network-intensive test application, targeting the incoming traffic, receives an image from the client side. However, processing operations are not involved and it actually works as a network drain. It consumes the incoming bandwidth of the network. The HTTP’s POST method is used to transmit the graph from the client to the benchmark server. A JMeter test plan for the POST method is created in the executor component, which is similar with the GET method test plan, excluding the HTTP method change and a file path to the graph added. Once the benchmark server receives the graph, a callback function referenced from the network-intensive test application is called to respond to the client with an “OK” flag. The network-intensive test application targeting the outgoing network traffic just sends fixed-size text to the client side, which consumes the outgoing network bandwidth. In
the initial implementation, the transferred data (an image or plan text) size can not be customized, but is fixed. Users can attempt a set of experiments and determine the proper data size for incoming and outgoing requests.

- **Memory-intensive test application:** The array is an object in JavaScript and its element positions are sparse. The array in JavaScript does not need to set a fixed length, because it grows automatically. For example, if an array `var arr = []` is declared, then two values, 1 and 100, are inserted to the index 0 and the index 999. The length of the array `arr` is 1000 at this time, instead of 2. This test application creates such an array without specifying a size and keeps inserting a number of objects to the array by a loop. It yields the CPU-intensive program stressing the CPU resource and consequently causes that the memory size can grow to be quite a large size, but it does not exceed the total memory size allocated to the container. Each request creates such a unique array and cannot be shared, thus it reduces the caching effect to mainly target the memory resource. Algorithm 5 shows the procedure to target the memory. Once the update operation is processed, the benchmark server responds to the client with an “OK” flag.

### 4.4 Summary

This chapter introduced the detailed implementation of the benchmark suite, including its infrastructure components and several benchmark test applications. The set of benchmark test applications are designed to target and stress specific resources. Additionally, infrastructure components coordinate with these bench-
Algorithm 5 Stress Memory Procedure

```plaintext
function STRESSMEM
    buff ← []
    loops ← LOOP
    for i ← 0; i<loops; i ++ do
        obj ← {}
        buff[i] ← obj
    end for
    return 1
end function
```

mark test applications to launch a benchmark run, post-process the captured raw data, and generate a final report. This benchmark suite can evaluate and measure different scalability strategies applied in Node.js. This thesis performs the Node.js scalability investigation by analyzing and modeling the data collected from this benchmark suite, introduced the next chapter.
Chapter 5

Scalability Investigation

Multiple instances of a single Node.js web application can be spawned to provide end users disparate services at the same time within a cloud system, which makes a cloud system able to provide a scalable solution to handle a multitude of concurrent requests. The scalability of Node.js web applications in cloud systems has to be taken into account to support the maximum number of concurrent requests. For this reason, this project will perform a set of scalability analyses and this chapter will present and discuss results.

This chapter is organized as follows: Section 5.1 describes scalability experiments that follow the two types of scalability patterns defined in this project; Section 5.2 introduces a series of scalability analyses based on the experimental results.
5.1 Scalability Experiments

We conduct several scalability experiments in a private cloud and collect related scalability metrics, using the benchmark suite discussed in Chapter 4, to address issues that are raised in Section 3.1. This section is organized as follows: Section 5.1.1 introduces the experimental setup; Section 5.1.2 shows a detailed design for the experiments.

5.1.1 Experimental Setup

The experimental setup stage has three tasks: 1) build a private cloud; 2) create a private registry that stores customized images; 3) containerize the benchmark server with test applications. The following subsections introduce the details of those three tasks.

5.1.1.1 Build a Private Cloud

The experimental setup begins with building a private cloud that provides a scalable, distributed, multi-core and virtualized cloud environment. To this end, we utilize Vagrant [36], a tool for building and managing VM environments, to launch and manage VMs in each physical server. VirtualBox VM is a default VM provider for Vagrant and it is free, cross-platform and less friction, therefore, we utilize Oracle VirtualBox VM, which is managed and maintained by Vagrant on each physical server. The benefits of using Vagrant to build a virtualized environment is: 1) we can simulate a cloud-based production environment; 2) hardware resource can be controlled on the fly; and 3) the whole environment can be re-produced easily by other researchers and have high portability. Vagrant enables users to specify the
OS version, the amount of virtual computing resources allocated, including the
number of virtual CPU cores and the virtual memory, and networking configura-
tions such as the IP address type (static/dynamic IP), hostname, etc., which can
be defined in a “Vagrantfile” file. A full “Vagrantfile” file is presented in Appendix
A.2. Listing 5.1 shows the code that allocates the virtual resource to a particular
VM in the “Vagrantfile” file. Four virtual CPU cores and 4096 MB of RAM are
allocated in this case. The entire environment simulates the cloud system using
the virtualization technique. However, in the current work, we do not consider
and measure the performance effects of this layer of virtualization. In future work,
we will perform a set of Node.js scalability investigation in the bare metal and
compare it with the virtualization environment, finding performance effects of the
layer of virtualization.

Once all VMs have been installed and run, the next step is installing the Docker
engine inside each VM. The swarm mode can be enabled directly by executing
several Docker CLI commands. Before starting swarm mode, protocols and ports
among VMs are ensured to be open, such as TCP port 2377 for cluster manage-
ment communication; TCP and UDP port 7946 for communication among VMs;
and UDP port 4789 for overlay network traffic.

Listing 5.1: Sample Code to allocate virtual resource to VM

```ruby
config.vm.provider "virtualbox" do |vb|
  vb.memory = "4096"
  vb.cpus = "4"
end
```
Listing 5.2 shows a public static IP 192.168.1.42 is assigned to the manager node in the Vagrant file. A hostname for each VM can be configured in the “Vagrantfile” file as well and all VMs are named: “manager”, “worker1”, “worker2”, “worker3”, “worker6”, and “worker20”.

Listing 5.2: Sample Code to assign IP to VM

```ruby
Vagrant.configure("2") do |config|
  config.vm.network "public_network",
  ip: "192.168.1.42"
end
```

The VMs hosted on physical servers are viewed as swarm nodes in this project. Once all node roles are determined, the command “docker swarm init --advertise-addr ⟨managerIP⟩” is executed in a manager node to create the swarm. The flag “--advertise-addr” and the IP address following it are used to let the manager node publish its IP address to worker nodes, which ensures all worker nodes can access the manager node at this IP address. This command is executed and its output contains a command used to enable worker nodes to join the newly created swarm. Finally, we execute the output command “docker swarm join --token ⟨TOKEN⟩ ⟨managerIP:2377⟩” on all five worker nodes. At this point, the entire swarm has been built, which contains five worker nodes and one manager node. The overview of the private cloud is displayed in Figure 5.1.
5.1.1.2 Setup a Private Docker Registry

Docker provides a public registry called a Docker Hub, where users can store their images and make them public. However, we do not need to upload our images to a public environment, therefore we set up a private registry that is limited to this project scope. Within this project scope, the customized images can be distributed to any node; in other words they can be pushed to and pulled from this private registry. In this private cloud, the manager node is responsible for maintaining and managing such a private registry service, therefore, the manager node works as a private registry server and the other worker nodes play a client role. A valid Transport Layer Security (TLS) protocol is maintained to ensure the Docker daemon does not reject connections from the registry client to the registry server. We setup the private registry in the manager node, therefore, we firstly create a self-signed certificate, using the OpenSSL tool on the manager node. Afterwards,
we run the registry with the local domain certificate and key file in the manager node. The certificate and key file path are passed to environment variables when we run the Docker registry container using Docker CLI, `docker run`. The generated certificate file in the manager node needs to be stored in other worker nodes as well, which ensures all nodes in the swarm have TLS support for the private registry. At this moment, customized images can be pushed to the private Docker registry server from any node as a client, and in turn any node as a client can access the server running the registry and pull images. Before the push operation, we tag the image in the form of `RegistryIP:port/ImageName:tag`. We tag the image used in this benchmark suite to `192.168.1.42:5001/ibench:latest`.

5.1.1.3 Containerize the Benchmark

Before conducting scalability experiments in this private cloud, the benchmark server with a set of test applications needs to be containerized so that benchmarks can be a service running in a Docker container. Just as Section 2.2.1 discussed, a Docker image is a read-only template with instructions to create a container and it is essentially a snapshot of the Docker container. A `Dockerfile` is first created to build a Docker image, in which it defines the environment inside the container. In the benchmark image, we 1) create a working directory inside the container; 2) install the Node.js runtime; 3) install necessary modules; and 4) publish an open port to the outside world from the container. The full `Dockerfile` is presented in Appendix A.1. Once the `Dockerfile` file is defined, we run CLI “`docker build -t ⟨ImageName⟩ -f ⟨DockerfilePath⟩ .`” to build the benchmark service and generate an image. The benchmark image can be pushed to the private registry using CLI
“docker push ⟨ImageName⟩”, and in turn the other nodes can pull it using “docker pull ⟨ImageName⟩” and create a container based on it later.

5.1.2 Experiment Design

This section presents the design for the scalability experiments: Section 5.1.2.1 discusses the selected scalability variables; Section 5.1.2.2 introduces two defined scalability patterns, applied in the experiments.

5.1.2.1 Scalability Variables

We selected several scalability variables shown below:

- **Number of nodes**: refers to the number of physical servers participating in the cluster. When computation is distributed, there will be more than one physical server. The physical servers run cloud infrastructures such as Docker in swarm mode, which forms a server cluster and enables them to communicate with each other in networks. The nodes can be classified into manager nodes and worker nodes according to their roles. Users can deploy and maintain their applications on top of these nodes.

- **Number of requests and concurrency**: is referred to as the workload, in terms of parallel connections from the client side. When Node.js test applications are deployed in the cloud and work as web servers, several users can access a server at the same time. Increasing or decreasing the request concurrency can test the ability of the entire system to handle workloads.

- **Number of Node.js processes/instances**: refers to the number of Node.js
instances created by different scalability strategies in the cloud. In other words, instances are replicas of a single Node.js web application to handle workloads. For example, the Cluster module creates multiple instances by forking worker processes, and PaaS clouds create multiple instances by scaling Node.js applications horizontally. Each Node.js instance will utilize one virtual or physical CPU core, which allows it to fully take advantage of CPU resources in the cloud system and make parallel computing possible.

- **Number of containers**: refers to the number of Docker containers that run Node.js applications hosted on the Docker swarm infrastructure.

- **Running time**: the total time spent running one benchmark. It starts when users fire requests, and ends when users receive all responses, excluding all coordination time among benchmark suite components. It indicates the total time that all requesting processes consume.

- **Response time (RT)**: the time elapsed between a user initially sending a request and receiving the response. In contrast to the running time, RT refers to the time that a single request process takes. Lower RT implies this system is more responsive and achieves high scalability.

- **Throughput**: the number of requests handled within a time-frame. A high value of throughput is expected, which indicates a high scalability for this system in handling increasing workloads.

- **Computing resource usage**: amount of computing resources used during the benchmark, such as CPU, memory, disk and networking bandwidth.
Various types of computing resource consumptions are crucial metrics to detect performance bottlenecks.

The scalability variables can be classified into independent variables and dependent variables. They are utilized to quantify and evaluate scaling effects in different scalability strategies. The independent variables represent the scaling aspect of the execution environment and they can be changed and controlled. For instance, the number of nodes can be varied and scaled. The dependent variables represent the aspects of the system behaviour that can be affected by changing independent variables, which can be manipulated for a scalability analysis. For example, a Node.js web application is scaled out, which results in one of the dependent variables, throughput, changing. Table 5.1 summarizes the classification of scalability variables, utilized in this project. The main goal of the scalability experiment is to observe and detect the relationship between these two types of variables to resolve the problem of how do the dependent variables behave or affect the independent variables under a given configuration. To achieve the goal, the scalability experiment varies different independent variables, following a one-factor-at-a-time experimental design.
5.1.2.2 Scalability Patterns

The experiments follow two scalability patterns according to the placement of test application instances deployed in Docker swarm. Figure 5.2 displays an overview of two scalability patterns applied in experiments.

1. **Single/Multiple Instance on Single Node (S/MISN) Pattern**: refers to single or multiple instances of a Node.js application deployed in one node participating in the Docker swarm. If the instance is created using the Docker swarm’s horizontal scaling service, that is horizontal scaling (HIS) strategy, then one instance runs in one Docker container. In this experiment, the Docker container resource constrains to one virtual CPU core and 128 MB memory, which enables one instance to run in one process, taking only one CPU core and partial memory from the underlying VM level. For example, if one experiment creates four instances and deploys them in worker3 (four-core CPU and eight GB memory in total), then there are four Docker containers hosting these four instances. All four cores of the CPU are taken and up to 512 MB of eight GB memory can be utilized in this experiment. If instances are created using the cluster module (CM), either one or multiple worker processes can be spawned into one Docker container.
For this reason, a larger Docker container with larger RAM size and an equal number of CPU cores to the underlying VM, is created. Single or multiple instances are spawned in this large Docker container, therefore we allocate much more RAM resource than HS experiments to support multiple instances and avoid out of memory. For example, an experiment specifies the number of instances is four and deploys in `worker3`; the resources of the large Docker container can be limited to four CPU cores and four GB of RAM; the spawned four instances run in this larger Docker container and fully utilize the four-core CPU and up to four GB of RAM resources under the Docker container level. In this scalability pattern, a comparison experiment between two scalability strategies, HS and CM is conducted; moreover, scalability experiments of benchmarking the HS strategy are also conducted.

2. **Multiple Instances on Multiple Nodes (MIMN) Pattern**: is referred to as multiple instances of Node.js application deployed in multiple nodes participating in a Docker swarm. This pattern extends the single node to multiple nodes, up to five, in the swarm. The specific combinations of node placement are varied and are denoted as a vector: `{[worker3], [worker2, worker3], [worker1, worker2, worker3], [worker1, worker2, worker3, worker6], [worker1, worker2, worker3, worker6, worker20]}. For example, when conducting the HS benchmarking experiment in the two-node cluster, it plans to create eight instances and deploys them in two nodes `worker2` and `worker3` (8 CPU cores and 8GB of RAM in total). The Docker swarm’s scheduler places these eight deployed instances in two nodes based on spread scheduling policy. Once all eight instances are deployed, the load balancer in the
Table 5.2: Experimental Configurations

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S/MISN</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Requests</strong></td>
<td>infinite</td>
<td>infinite</td>
</tr>
<tr>
<td><strong>Instances</strong></td>
<td>{1,2,3,4,5,6,7,8,9,10,...,20}</td>
<td>{1,2,3,...,12}</td>
</tr>
<tr>
<td><strong>Workload</strong></td>
<td>{5,15,25,35}</td>
<td>{5,15,25,35}</td>
</tr>
<tr>
<td><strong>Node</strong></td>
<td>{1}</td>
<td>{1}</td>
</tr>
<tr>
<td><strong>Duration</strong></td>
<td>100 sec</td>
<td>100 sec</td>
</tr>
<tr>
<td><strong>MIMN</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Requests</strong></td>
<td>infinite</td>
<td>infinite</td>
</tr>
<tr>
<td><strong>Instances</strong></td>
<td>{1,2,3,4,5,6,7,8,9,10,...,20}</td>
<td>{1,2,3,...,12}</td>
</tr>
<tr>
<td><strong>Workload</strong></td>
<td>{5,15,25,35}</td>
<td>{5,15,25,35}</td>
</tr>
<tr>
<td><strong>Node</strong></td>
<td>{1,2,3,4,5}</td>
<td>{1,2,3,4}</td>
</tr>
<tr>
<td><strong>Duration</strong></td>
<td>100 sec</td>
<td>100 sec</td>
</tr>
</tbody>
</table>

manager node distributes requests to worker nodes in a round-robin load balancing policy. In this experimental pattern, the scalability investigation of a HS strategy based on a multi-node experiment is conducted. The container resource constraints are similar with the S/MISN pattern, that is, one Docker container is only allowed to run one instance and the total CPU resource is limited to the same with its underlying VM.

Both patterns can be achieved by specifying a Docker compose file, `docker-compose.yml`, which was introduced in Section 2.2.3. A complete configuration for both patterns is displayed in Appendix B, Section B.1 and Section B.2 respectively. The Docker container is clean and only contains a necessary Node.js runtime and original test applications, which maintains isolation but not performance interference from other programs in the same Docker container.

In summary, we conduct a number of experiments varying the defined independent variables shown in Table 5.2 and narrow them down to the following groups,

- **HS strategy benchmark**: applies both the S/MISN and MIMN scalability
patterns. We conduct some strong scalability experiments by varying the number of instances and fixing the workload for each experiment run. The test applications, CPU-intensive and network-intensive (outgoing network traffic) are utilized.

- **CM benchmark:** applies only the S/MISN scalability pattern. The CM does not allow a worker process to be forked in a separate node, but the master process and the worker process must run together in the same node, which limits the CM to run in a single node. For this reason, the MIMN pattern that involves multiple nodes is not suitable to be applied to benchmark the CM strategy. The test applications, CPU-intensive and network-intensive (outgoing network traffic) are utilized as well.

## 5.2 Experimental Results and Analysis

In this section, we present several scalability investigations based on obtained experimental results. We benchmark the HS strategy applying both the S/MISN and MIMN scalability patterns, and the CM strategy only using the S/MISN scalability pattern. We collect diverse metrics to represent the system’s performance, such as the throughput and the response time; the others represent the characteristics of the system performance, such as the resource usage. This section is organized as follows: Section 5.2.1 demonstrates experimental results of benchmarking the HS and CM strategy applied to the S/MISN scalability pattern; Section 5.2.2 demonstrates experimental results of benchmarking the HS strategy after applying the MIMN scalability pattern.
5.2.1 S/MISN Scalability Pattern Analysis

This section performs a scalability analysis with the S/MISN scalability pattern applied to the HS and CM scalability strategies by demonstrating and discussing the obtained experimental results. Section 5.2.1.1 introduces the analysis of the HS scalability strategy using the CPU-intensive test application; Section 5.2.1.2 introduces the analysis of the HS scalability strategy using the network-intensive test application; Section 5.2.1.3 presents a comparison analysis of the HS and CM scalability strategies and two different types of test applications in the cloud.

5.2.1.1 HS Analysis Using CPU-intensive Application Benchmark

We utilized the VM “worker3” to run Docker containers. The “worker3” VM is allocated to four virtual CPU cores and four gigabytes of RAM from its host machine “casahost3”. The CPU-intensive test application stresses and targets the CPU resource, which processes heavy computing by maintaining a tight loop to judge the primality of a large integer. For this reason, the CPU resource is critical to the CPU-intensive test application to maintain a good performance.

According to Little’s Law, $L = \lambda W$, under a steady state, the number of queued requests ($L$) is equal to the rate at which requests arrive ($\lambda$) multiplied by the time a request takes to process ($W$) [37]. We apply it to our experiments. Specifically, the workload in terms of concurrent requests is denoted by $L$; throughput in terms of the number of requests processed per second is denoted by $\lambda$; and response time is denoted to $W$. Figure 5.3 illustrates the throughput as a function of the number of instances within Docker containers when workloads of 5, 15, 25, and 35 are launched; Figure 5.4 presents response time as a function of the number of
instances with varying workloads. From Figure 5.3 and Figure 5.4, it can be seen that when we increasing workloads, both throughput and response time become greater. It is consistent with Little’s Law. For example, throughput and response time at the workload of 5 are less than that at the workload of 15. The reason is that workloads (from five to thirty-five) do not saturate computing resources, which causes throughput is directly proportional to response time. When we fix the workload, we find the relationship between throughput and response time: when the number of instances reaches four, throughput reaches the maximum and
then becomes retrograde to start decreasing with the greater number of instances; while response time reaches the minimum when four instances are spawned; the response time starts to increase with a higher number of instances after four. We can see that response time begins to grow at the same point where throughput begins to decline. Such a pattern shows response time is in inverse proportion to the throughput. The reason is when we fix the workload and the workload does not saturate the system, the system has a higher ability to handle more requests within the shorter time and consequently resulting in a higher throughput, which is also consistent with Little’s Law.

Figure 5.5 demonstrates the CPU utilization percentage as a function of the number of instances within Docker containers when various workloads are launched. When there are four instances spawned in the VM, the CPU utilization reaches the peak (approximately 366% of 400%) and then starts to decrease flatly. There are four virtual CPU cores allocated to the VM, for this reason, once four instances are spawned and each instance utilizes a CPU core, all CPU cores are busy processing requests from clients, in other words the CPU resource is saturated. At this moment, instances running in processes compete for the CPU resource and a context switch occurs to handle workloads from clients. The context switch has a high overhead in this situation, which results in a retrograde performance of the system. This can be verified from the scalability models USL and Amdahl’s Law discussed later. Therefore, the CPU resource is the bottleneck in this situation.

Figure 5.6 and Figure 5.7 show the incoming and outgoing network bandwidth respectively as a function of the number of instances within Docker containers when different workloads are launched. When the number of instances exceeds
Figure 5.5: CPU-intensive App. Total Cluster CPU Utilization (worker3: maximum CPU utilization of total cluster is 400%)

Figure 5.6: CPU-intensive Application Total Cluster Incoming Network Bandwidth (worker3)

Figure 5.7: CPU-intensive Application Total Cluster outgoing Network Bandwidth (worker3)
Figure 5.8: CPU-intensive Application Total Cluster Throughput (worker6)

Figure 5.9: CPU-intensive Application Total Cluster Response Time (95%) (worker6)

Figure 5.10: CPU-intensive Application Total Cluster CPU Utilization (worker6: maximum CPU utilization of total cluster is 800%)
four, limited CPU resources cannot support more instances of the test application. Moreover, the network bandwidth is limited to the CPU resource and it causes the system to accept fewer requests and to reduce the network bandwidth. The CPU-intensive test application does not involve much data transmission using network bandwidth. For this reason, the overall incoming and outgoing network bandwidth is less than 1 Megabyte per second, therefore, the network bandwidth is not the performance bottleneck in this case.

Afterwards, we continue the scalability experiments using the CPU-intensive test application, but we conduct them in another VM “worker6” that has eight virtual CPU cores, a greater number of virtual CPU cores than the VM “worker3”. Figure 5.8, Figure 5.9 and Figure 5.10 report metrics throughput, response time and total cluster CPU utilization as functions of the number of instances within Docker containers when running on the VM “worker6”. It can be seen that the point where throughput and response time start to decline, and the CPU utilization reaches the maximum moves from four instances to eight instances. The reason is the “worker6” has greater number of virtual CPU cores and supports more number of instances of the Node.js test application running than the VM “worker3”. We can make the similar observation and conclusion that when the CPU resource has been saturated, the system cannot support more workloads and has retrograde performance, thus the CPU resource is still the performance bottleneck.
5.2.1.2 HS Analysis Using Network-intensive Application Benchmark

We utilized the VM “worker6” that has eight virtual CPU cores and four gigabytes of RAM to run Docker containers, when we utilize the network-intensive test application to conduct scalability experiments. The network-intensive test application stresses and targets the outgoing network bandwidth by responding to clients with a large data size. Network bandwidth is a critical resource in this case. The “worker6” VM has more CPU resources, because it ensures the bottleneck of CPU resources is eliminated and targets the network bandwidth resource. Fig-
ure 5.11 shows throughput as functions of the number of instances within Docker containers when launched with workloads of 5, 15, 25, and 35. It can be seen that throughput is maximized when the workload is 15, whereas, Workloads of 25 and 35 degrade throughput. Throughput starts to degrade when there are three instances within the workload of 15, 25 and 35. In contrast, at the workload of 5, throughput becomes retrograde at the point where two instances are spawned, because the client side does not provide more workloads for instances to stress the network bandwidth. Figure 5.12 shows response time as functions of the number of instances within Docker containers when launched with workloads of 5, 15, 25, and 35. It illustrates that response time becomes longer with a greater workload. We find that there is a directly proportional relationship between throughput and response time and is consistent with Little’s Law, when increasing workload from 5 to 15 is launched. Afterwards, the relationship of throughput and response time becomes inverse proportional. The reason is that workloads that exceed 15, overload the server and make the server unsteady, which causes hardware in the network, such as the network interface card (NIC) to reach its maximum capacity and the server accepts fewer requests from clients. When we monitor the real-time CPU utilization using the command “top -H”, we find quite a high CPU utilization (approximately 100%) in the ksoftirqd process [38]. It means the NIC receives a very large number of packets in a short time frame and reaches the bottleneck, causing a number of interrupt requests (IRQ). Consequently, the OS has to handle the IRQ and cause instances having a long delay to process requests. In addition, we find that this case violates Little’s Law, because the system does not keep a steady condition when it is overloaded.
Figure 5.13: Network-intensive Application Total Cluster CPU Utilization (Maximum CPU utilization of total cluster is 800%)

Figure 5.14: Network-intensive Application Total Cluster Outgoing Network Bandwidth

Figure 5.13 shows the total cluster of CPU utilization percentages as a function of the number of instances within Docker containers when different workloads are fired. Figure 5.14 demonstrates that the total cluster of outgoing network bandwidth is a function of the number of instances. From both figures, it can be seen that the CPU utilization of the network-intensive test application is smaller than that of the CPU-intensive test application and does not even exceed half of 800%, while its outgoing network bandwidth is greater. We find the CPU resource is not the performance bottleneck in this case, because the CPU utilization keeps increas-
ing with a greater number of instances. It means that this test application does not have much CPU resource contention and the CPU resource is not saturated when adding more instances in this system. In addition, we note that the performance does not start to degrade when eight instances are created in the system, even though all CPU cores are utilized by a number of instances. It implies the CPU resource suffices to support more than eight instances and removes the bottleneck of the CPU resource. Instead, the network is the performance bottleneck in this case. The network resource is saturated by instances and makes hardware in the network congested. The network congestion limits the data transferred to the client side, which results in retrograde overall performance. The network-intensive test application consumes much more network bandwidth and reaches the peak when approximately three instances are spawned. Afterwards, it accordingly keeps constant, which matches the trend of throughput and also verifies the network is the performance bottleneck. We also check other resource usage data collected by one of the benchmark suite infrastructure components RC, such as disk I/O throughput and RSS. We find the disk I/O throughput is zero, because the network-intensive test application does not target the disk I/O or contains file writing and reading operations. We also find the RSS is not the performance bottleneck, because we allocated enough RAM to Docker containers.

5.2.1.3 Comparison Analyses

We perform two groups of comparison analyses: one is comparing both the HS and CM scalability strategy; the other one is comparing the scalability effects of the CPU-intensive test application and the network-intensive test application applied
We first compare both scalability strategies based on metrics throughput, response time, and CPU utilization when launching different workloads. The comparison experiment is conducted in the VM “worker3” with four virtual CPU cores. We collect these metrics, when four instances are created. Figure 5.15 demonstrates a histogram where the maximum throughput is a function of the various workloads. Figure 5.16 shows a histogram where the minimum response time is a function of the various workloads. Figure 5.17 shows the total cluster’s maximum CPU utilization in function of different workloads. The red bar represents the CM strategy and blue bar denotes the HS strategy in all histograms.

These histograms indicate the HS strategy with higher throughput and shorter response time, outperforms the CM strategy. Moreover, the CM strategy utilizes more CPU resources to support its good performance than the HS strategy. The \textit{cluster} module is a built-in module that is implemented inside Node.js without
depending on other software. The master process is responsible for maintaining worker processes and the load balancing using a round-robin policy. Those tasks are not free and produce overheads, moreover it causes the cluster module to utilize the additional CPU resources to support the best performance. Apart from those tasks that the master process has to perform, the master process also works
in parallel with other worker processes to handle the workload. However, the master process handles less workload than worker processes due to its additional task overheads, which degrades the overall performance. In contrast, under the HS strategy, all the instances of the Node.js test application does not process the load balancing task and the worker process maintenance as the master process does. A load balancer is installed and utilized in the manager node, which does not compete for CPU resources with instances deployed in worker nodes. Consequently, it ensures all instances deployed in worker nodes focus on handling concurrent workload from clients without processing additional tasks. Therefore, the horizontal scaling can process more workload and achieve higher throughput than the cluster module.

Afterwords, we perform a regression analysis by building different scalability models for both scalability strategies to calculate coefficients and identify their scalability characteristics. We utilize the software scaling form for both scalability models. Figure 5.18 and Figure 5.19 respectively show the maximum throughput of the CPU-intensive application achieving, as a function of a varying workloads of the HS and CM scalability strategy. Dots denote measured data points; dashed lines denote the USL and Amdahl’s Law model fitted to sample data; Both plots demonstrate the effects of diminishing returns, which means the benefit of achieving good performance gets smaller for greater numbers of workloads.

Various coefficients are determined by performing regressions and their associated values are shown in Table 5.4. It summarizes coefficient values of the HS and CM scalability strategy obtained from the USL model and the Amdahl’s Law model regression analysis. Depending on coefficients and workloads, we can calculate
the predicted relative capacity $C(p)$ for both models. When we launch the workload of one, their values of predicted relative capacity $C(1)$ equals one by utilizing Equations (5) and (6). Finally, we can determine the modeled throughput data $X_{max}(p)$ by utilizing the Equation (2). From Table 5.4, we find the USL model provides a better fit to the data than the Amdahl’s Law model, depending on $R^2$ values. The $R^2$ value estimates how well the model fits the sample data. The Amdahl’s Law model has poor $R^2$ values. The reason is that Amdahl’s Law does not consider interprocess communication overheads that are significant for multiple instances of web-based applications, whereas, the USL model extends Amdahl’s Law with extra effects of coherency for the interprocess communication. From the USL modeling, we find that values of contention and coherency, represented by the coefficient $\sigma$ and $\lambda$ respectively, are non-zero for both scalability strategies. The reason for non-zero contention is that more workloads are launched on the system and the system cannot process them as they arrive, but queuing them for later processing. Therefore, contention is produced which is also consistent with Amdahl’s Law. Furthermore, we also find that values of the interprocess communication overhead, represented by the coefficient $\lambda$, for both scalability strategies are the same ($\lambda = 0.001$). This is reasonable, because we spawned the same number of instances or worker processes in both strategies, when we conduct the experiment. However, when we compare the values of $\sigma$ of both strategies, we note the HS strategy has a higher value ($\sigma = 0.0686$) than the CM strategy ($\sigma = 0.0547$). It indicates the CM strategy should have a better performance than the HS strategy due to less contention, which is against the previous comparison analysis. However, we find their difference is not very large and we also see the USL model does
not well fit to the CM strategy by observing the value of $R^2$. It explains why the coefficient values does not reflect and match the previous comparison analysis.
Table 5.3: Coefficient Values of Scalability Models for CPU-intensive Application with HS and CM Strategy

<table>
<thead>
<tr>
<th></th>
<th>HS Strategy</th>
<th>CM Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>USL Model</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\sigma_c$</td>
<td>0.0686</td>
<td>0.0547</td>
</tr>
<tr>
<td>$\lambda_c$</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>$R^2_c$</td>
<td>99.34%</td>
<td>98.75%</td>
</tr>
<tr>
<td><strong>Amdahl’s Law Model</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1062</td>
<td>N/A</td>
<td>97.73%</td>
</tr>
<tr>
<td>0.1062</td>
<td>N/A</td>
<td>97.73%</td>
</tr>
</tbody>
</table>

When we ignore the interprocess communication overhead and only consider contention, we can see both strategies have the same value, $\sigma = 0.1062$ from the Amdahl’s Law modeling. Essentially, they apply the multi-process technique and have the same number of instances or worker processes. Therefore, if we ignore communication overheads between instances or worker processes and launch the completely same workloads on them, contention is probably the same.

The next comparison we conduct is comparing the characteristics and scalability effects of the CPU-intensive and the network-intensive test application. Both test applications are run in the VM “worker6”. We collect the maximum throughput, when a particular number of instances are created with launching various workloads. We also perform a regression analysis on network-intensive test application by fitting the USL model, and the Amdahl’s Law model to experimental result datasets. Figure 5.20 and Figure 5.21 respectively illustrate the Amdahl’s Law and USL model of CPU- and network-intensive test application fitting to benchmark data.

We calculate their associated coefficient values of the USL and Amdahl’s Law model and summarize them, listed in Table 5.4. From Table 5.4, we can see coefficient values for both models are non-zero and they control the shape of functions.
From Figures 5.20 and 5.21, we can see shapes of functions are quite similar and conclude the performance scales not linearly but sub-linearly, launching a greater
Table 5.4: Coefficient Values of Scalability Models for CPU- and Network-intensive Application

<table>
<thead>
<tr>
<th></th>
<th>CPU-intensive</th>
<th>Network-intensive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>USL Model</td>
<td></td>
</tr>
<tr>
<td>$\sigma_c$</td>
<td>0.012</td>
<td>0.0282</td>
</tr>
<tr>
<td>$\lambda_c$</td>
<td>0.0012</td>
<td>0.002</td>
</tr>
<tr>
<td>$R^2_c$</td>
<td>98.88%</td>
<td>99.24%</td>
</tr>
<tr>
<td>Amdahl’s Law Model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\sigma_n$</td>
<td>N/A</td>
<td>0.1023</td>
</tr>
<tr>
<td>$\lambda_n$</td>
<td>N/A</td>
<td>93.43%</td>
</tr>
<tr>
<td>$R^2_n$</td>
<td>83.27%</td>
<td></td>
</tr>
</tbody>
</table>

number of workloads. That is determined by non-zero coefficient values; in other words contention and coherency affect the scalability of the system. It can be seen that the USL model provides a good fit for both test applications due to a high value of R-square. In contrast, the Amdahl’s Law model has a worse fit for the CPU-intensive test application and the network-intensive test application. Both types of test applications create multiple instances to support maximum throughput, causing significant communication overheads within the system. However, we have introduced earlier that Amdahl’s Law does not consider the interprocess communication overhead, which results in worse fitting than the USL model. Furthermore, we can find all coefficient values of both models for network-intensive test application are greater than which of the CPU-intensive test application. It means the network-intensive test application has higher contention and coherency than the CPU-intensive test application. Both terms affect performance of the network-intensive test application, which can be verified from Figure 5.20 and 5.21, the CPU-intensive test application achieving higher maximum throughput than the network-intensive test application.
Table 5.5: Total available virtual CPU cores and RAM for various cluster size

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>Available vCPUs</th>
<th>Available vRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4G</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>8G</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>12G</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>16G</td>
</tr>
<tr>
<td>5</td>
<td>28</td>
<td>20G</td>
</tr>
</tbody>
</table>

5.2.2 MIMN Scalability Analysis

We extend the number of nodes from a single node to multiple nodes to conduct a set of scalability experiments using both the CPU-intensive test application and the network-intensive test application. We collected the metrics throughput, response time and resource usage to investigate the scalability effects and determine the performance bottleneck. Table 5.5 shows all available virtual CPU cores and RAM for various number of nodes in the cluster. Section 5.2.2.1 and Section 5.2.2.2 respectively present and discuss the scalability issues based on the CPU-intensive test application and the network-intensive test application.

5.2.2.1 Using CPU-intensive Application Benchmark

We first perform an analysis of strong scalability, which means observing the performance and varying the number of instances (one to twenty) when the workload is fixed to 25. Figure 5.22 and 5.23 respectively illustrate throughput and response time of the cluster for different numbers of spawned instances when experiments run on various numbers of nodes (one to five). We can change the cluster size by removing or adding a node from the cluster, therefore the cluster size is varied.
Figure 5.22 demonstrates how throughput scales according to the varying number of nodes. Obviously, it does not scale linearly with respect to the number of instances, according to the different cluster sizes. For example, considering when...
Table 5.6: Maximum throughput of CPU-intensive test application achieved for various cluster size

<table>
<thead>
<tr>
<th>Number of Nodes</th>
<th>Thr.</th>
<th>Number of Inst. at Max Thr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3229.61</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5100.43</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>8348.39</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>9526.41</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>10187.80</td>
<td>14</td>
</tr>
</tbody>
</table>

there are ten instances created, the throughput obtained within one node is 3019 requests/second and within three nodes is 7340 requests/second. Thus, the ability to handle requests per second within three nodes is supposed to be triple that of one node, and the expected value should be 9057 requests/second. However, the real value is 7340 requests/second that is approximately twice greater than which of one node, 3229.61. Table 5.6 lists the maximum and achievable throughput for different cluster sizes from one to five nodes under a given workload 25. We can see such non-linear scalability effects from this table. Figure 5.23 illustrates that response time gives us a similar information to throughput. It can be seen that response time decreases at the beginning until reaching the particular lowest point, then it increases toward an exponential trend with a high number of instances. The point where response time starts to increase is the same as throughput starts to decline. In addition, we can also find how response time scales accordingly with the various cluster sizes. With a greater cluster size, response time becomes shorter under a fixed workload of 25. Response time also follows a non-linear scalability pattern.
We also measure and collect the CPU resource usage percentage (Figure 5.24), the network incoming bandwidth (Figure 5.25), and the network outgoing bandwidth (Figure 5.26) data to identify performance bottlenecks. All these figures show that multiple instances of the CPU-intensive test application, from one to twenty, runs on various cluster sizes under a fixed workload of 25. Considering the characteristics of the CPU-intensive test application, we assume the CPU resource is the potential performance bottleneck. Figure 5.22 verifies our assumption is correct. The CPU utilization grows until reaching a peak point where all created instances have saturated the CPU resource and the utilization reaches the maximum capacity of computation, as we increase the number of instances. For this reason, the network incoming and outgoing bandwidth begins to decline at the same point where the CPU utilization begins to decrease.
5.2.2.2 Using Network-intensive Application Benchmark

This section introduces some scalability analyses using the network-intensive test application benchmark in various cluster sizes (from one node to four nodes), when
varying workloads are launched (from workload of five to thirty-five). In addition, we also make a set of predictions of performance bottlenecks for this case.

Figure 5.27: Network-intensive Application Total Cluster Maximum Throughput

Figure 5.27 illustrate maximum throughput as a function of the number of nodes when launched with varying workloads of 5, 15, 25 and 35. From Figure 5.27, we can see throughput does not scale well with a greater number of nodes for each launched workload. In other words, the benefits of obtaining good performance are smaller with extending the cluster size. The system that runs in various numbers of nodes, achieves the maximum throughput when the workload of 15 is launched. When workloads that exceed 15 are launched, throughput becomes retrograde.

Figure 5.28 illustrates minimum response time as a function of the number of nodes when launched with varying workloads. From Figure 5.28, we fail to see an obvious scalability effect of response time with a greater number of nodes for different launched workloads. The relationship between throughput, response time, and the
workload is quite similar with that of S/MIMN scalability pattern. When the workload varied from 5 to 15, throughput and response time is consistent with Little’s Law and they are directly proportional to the workload: when workloads increase, both throughput and response time are greater. Once the workload exceeds 15, throughput is inverse proportional to response time with greater workloads. The reason is the system encounters the resource saturation after workloads reach 15, and in turn leading to performance bottlenecks. We make several predictions for those bottlenecks.

Figure 5.29 shows the maximum CPU utilization as a function of the number of nodes when different workloads are launched. From Figure 5.29, it indicates that the CPU resource is not the performance bottleneck, because this system has a quite low CPU utilization for each launched workload, when it runs in different cluster sizes. It can be seen that the maximum CPU utilization only has 263%
Figure 5.29: Network-intensive Application Total Cluster Maximum CPU Utilization (Maximum CPU utilization of one-node cluster is 400%, two-node cluster is 800%, three-node cluster is 1200%, and four-node cluster is 2000%)

Figure 5.30: Network-intensive Application Total Cluster Maximum Outgoing Network Bandwidth
of 2000%. It is determined by the characteristic of this test application. Figure 5.30 presents the maximum network bandwidth as a function of the number of nodes within various workloads. It shows a similar scalability effect with that of throughput. We find a network bandwidth bottleneck at the workload of 15, which matches the throughput trend. After the workload of 15, the network bandwidth starts to decline. We also check other computing resource usage data and the result is the same with that of S/MIMN scalability pattern: the disk I/O and RAM are not performance bottlenecks in this case. Node.js itself cannot be the performance bottleneck in this case. We have confirmed that this test application does not have contention of the CPU resources, but much contention of network resource. Node.js is best suited for handling network I/O due to its non-blocking and asynchronous nature. In turn, this model allows other requests from clients to be served while waiting for an I/O operation to complete. We make the following predictions about causes of the performance bottlenecks and in future work, these predictions will be verified:

- **Network configuration**: a poor network configuration of Linux TCP/IP connection in nodes also has a significant influence on network performance and in turn affects scalability of the whole system. For example, in a poor network configuration condition, there are fewer available ephemeral ports defining the maximum number of sockets that can be created from an IP address, which limits the launched number of concurrent requests. In addition, there is a long wait time at the TIME_WAIT state of TCP connection, which causes a long connection time and in turn produces greater network latency.
• **Docker swarm overlay network**: The concurrent requests from clients are dispatched by a built-in load balancer to different Docker containers hosting web servers over the overlay network. The internal load balancer can be a bottleneck and will not be able to serve beyond those concurrent requests from clients. We use the load balancer that is integrated in the manager node. When creating more Docker containers, extending the cluster size, or increasing workloads, the load balancer will be overloaded and degrade the network performance.

• **Hardware in the network**: Hardware in the network, such as switches, network interface cards (NICs) and links can affect the network performance with a greater workload. For example, transferring the data consumes much network bandwidth and in turn leads to exceeding the maximum capacity of links between the client and the server. The NIC and the multi-port switch can also overload and produce congestion with an increased workload, which results in retrograde network performance.

5.3 **Summary**

This chapter introduced how the Node.js scalability investigation is performed by conducting a set of scalability experiments using the benchmark suite. To this end, the thesis firstly defined several scalability variables classified into independent and dependent variables, and two scalability patterns, S/MISN and MIMN, according to deployment of benchmark test applications. Afterwards, this thesis investigated Node.js scalability on the HS strategy utilizing the CPU-intensive and
network-intensive test application, and all experiments were conducted applying both scalability patterns. Moreover, the Node.js scalability investigation on the CM strategy utilizing the CPU-intensive test application was performed as well and it is only applied the S/MISN pattern. Finally, this chapter compared the scalability effects of both scalability strategies, the HS and CM, applied in Node.js and two types of test applications. The regression analysis were also performed. The associated conclusion and future work are discussed in the next chapter.
Chapter 6

Conclusion and Future Work

The goal of this thesis was to investigate scalability of Node.js using different scalability strategies, horizontal scaling (HS) and cluster module (CM), in clouds and identify its associated performance bottlenecks, utilizing a scalability-oriented benchmark suite.

To achieve this goal, this research project first developed a scalability-oriented benchmark suite that contains a set of resource-intensive test applications and some benchmark infrastructure components. These resource-intensive test applications target and stress different types of computing resources; benchmark infrastructure components coordinate with these test applications to perform scalability-based benchmarks. A set of scalability-oriented metrics were also determined in order to perform a scalability analysis. This research project also built a private cloud using Docker Swarm, which provides a scalable, distributed and multi-core environment and allows us to run a number of scalability experiments in it.

Afterwards, this research project started to investigate Node.js scalability in clouds...
based on results of the experiments. Several scalability variables were defined and they can be classified into dependent and independent variables, which enables us to investigate and analyze the relationship between these two types of variables, and perform a scalability analysis. In addition, this research project also defined two types of scalability patterns depending on the deployment of containerized benchmarks in the cloud: S/MISN refers to deploying single or multiple instances of containerized benchmarks in a single node, and MIMN refers to deploying multiple instances of containerized benchmarks in multiple nodes.

This research project utilized a CPU-intensive and a network-intensive test application from the benchmark suite to conduct a number of scalability experiments, applying S/MISN and MIMN scalability patterns. The following observations and conclusions are drawn:

- For CPU-intensive test applications, the CPU resource is critical. When more instances of containerized CPU-intensive test applications are created in a single node, the CPU resource becomes the bottleneck and significantly affects performance. In other words, once the CPU resources are saturated by a greater number of instances, the system cannot support many more instances and in turn leads to a retrograde performance. When extending the cluster by adding more nodes, it can observe sub-linear scalability effects. The performance measured by throughput, response time, and CPU utilization percentage, has an obvious improvement, but the benefit of adding a greater number of nodes to the cluster becomes smaller due to a heavy network communication overhead between these additional nodes in the cluster. It can be concluded that adding more instances of a web application does
not always improve the performance, because these additional instances will saturate the limited computing resource. Once it reaches the maximum computing capacity, system users need to consider adding more hardware, such as new nodes. However, the additional network communication overhead has to been taken into account, and an optimized network environment is necessary to maintain good performance.

- For network-intensive test applications, it can be found the network resource is critical. With launched workloads and the number of instances increasing in a single node, much more network bandwidth is consumed until reaching a maximum capacity and becoming a performance bottleneck. When performed the scalability analysis based on the MIMN scalability pattern, it cannot be seen an obvious scalability effect. The performance of the whole cluster does not have a significant improvement accordingly by adding more nodes. Based on this non-scaling effect, several predictions in aspects of hardware in the network, Docker swarm overlay network, and network configuration of nodes are made. It can be concluded that building a good network environment, such as network hardware, network topology, and network configuration of TCP/IP connection, is necessary to achieve a performance improvement, instead of just adding more nodes or instances of applications.

- This research project compared both types of test applications, CPU-intensive and network-intensive by conducting a regression analysis and identified their characteristics of scalability. The Amdahl’s Law model and the USL model, in terms of software scalability were fitted to obtained experimental
result data set and calculated associated coefficients. Non-zero coefficients were obtained. Based on the result, it can be concluded that the performance of both test applications is sub-linear scaling with launching more workloads. The contention and coherency have an effect on their scalability. The network-intensive test application is designed to transfer large sized data, which produces a long wait time for data transmission. Moreover, when launching concurrent requests that transfers large sized data, the system cannot handle them as they arrive and becomes overloaded. Consequently, it causes higher contention and coherency than the CPU-intensive test application, and degrades the system performance when running the network-intensive test application.

- This research project also compared the HS and CM scalability strategies using CPU-intensive test applications based on the S/MISN scalability pattern. It can be concluded that the HS scalability strategy outperforms the CM, because HS obtains higher throughput, shorter response time and lower CPU utilization than CM. In the CM scalability strategy, the master process and associated worker processes must run in the same Docker container, and the master process has to be responsible for load balancing and worker-process maintenance that are not free and produce the extra overheads. The HS scalability strategy migrates the load balancing service to the manager node, which ensures the load balancing service and requests handling are separated and has less overhead than the CM scalability strategy.

In future work, This research project will extend our scalability investigation by utilizing the other type of resource-intensive test applications, such as memory-
intensive and disk-intensive test applications. A real-world benchmark will also
be performed and use different workloads, such as querying database concurrent
requests. In addition, predictions about the scaling failure of network-intensive
test applications will be verified and network bottlenecks will be narrowed down.
Bibliography


[27] Stavros Aronis, Nikolaos Papaspyrou, Katerina Roukounaki, Konstantinos Sagonas, Yiannis Tsiouris, and Ioannis E Venetis. A scalability benchmark


Appendix A

Private Cloud Configuration

Files

A.1 Dockerfile File

```
FROM node:boron
# Create a user to run the benchmark
RUN useradd -s /bin/bash -d /home/bench -m bench

# Set the working directory
WORKDIR /home/bench/benchmarks

# Copy the current directory contents into the container at working directory
ADD . /home/bench/benchmarks

# Set user permissions for the files
RUN chown -R bench:bench /home/bench/benchmarks

# Install necessary modules
RUN npm install

# Switch user to bench
USER bench
```
# Make port 9000 available to the world outside this container
EXPOSE 9000

# Start benchmark server
CMD ["node", "benchmarks.js"]

## A.2 Vagrant File

```ruby
# ←← mode: ruby ←←
# vi: set ft=ruby :
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/xenial64"
  config.vm.box_version = "20170815.1.0"
  config.vm.hostname = "worker1"
  config.vm.network :public_network
  config.vm.provider "virtualbox" do |vb|
    vb.memory = "4096"
    vb.cpus = "4"
  end
  config.vm.provision "shell", inline: <<-SHELL
    apt-get update
    apt-get install linux-image-extra-$(uname -r) linux-image-extra-virtual -y
    apt-get install apt-transport-https ca-certificates curl software-properties-common -y
    curl -sSL https://get.docker.com | sh
    apt-get update
    docker --version
    iptables -A INPUT -p tcp --dport 22 -j ACCEPT
    iptables -I INPUT -p tcp --dport 2376 -j ACCEPT
    iptables -I INPUT -p tcp --dport 7946 -j ACCEPT
    iptables -I INPUT -p udp --dport 7946 -j ACCEPT
    iptables -I INPUT -p udp --dport 4789 -j ACCEPT
    docker swarm join --token SM0TKN-1-j37btk12neauocoesqnb7agvph8n3scsjch478004x8e uqsi0cy-2bs4gkg8sdwyjgh0rqkzai17d 192.168.1.42:2377
  SHELL
end
```

134
Appendix B

docker-compose.yml Files

B.1 Using S/MISN Scalability Pattern

```yaml
version: "3"
services:
  ibench:
    image: 192.168.1.42:5001/ibench
    deploy:
      replicas: 1
      resources:
        limits:
          cpus: "1"
          memory: 512M
      placement:
        constraints: [node.hostname=worker9]
    ports:
      - "9001:9000"
    environment:
      - CPU_COUNT=0
      - SERVER_PORT=9000
      - NODE_SERVER_PORT=9001
      - SERVER_IP=0.0.0.0
networks:
  webnet:
```

135
B.2 Using MIMN Scalability Pattern

```yaml
version: "3"

services:
  ibench:
    image: 192.168.1.42:5001/ibench
    deploy:
      replicas: 1
      resources:
        limits:
          cpus: "1"
          memory: 128M
    placement:
      constraints:
        - node.hostname != manager
    restart_policy:
      condition: on-failure
    ports:
      - "9001:9000"
    environment:
      - CPU_COUNT=0
      - SERVER_PORT=9000
      - NODE_SERVER_PORT=9001
      - SERVER_IP=0.0.0.0
    networks:
      - webnet
```

networks:
  webnet:
Vita

Candidate’s full name: Jiapeng Zhu

University attended: Bachelor of Computer Science, University of New Brunswick, 2016

Publications:

Conference Presentations: N/A