Performance Evaluation of Node.js on Multi-core Computing Systems
Abstract

Since JavaScript code that is executed by the Node.js run-time environment is run in a single thread without really utilizing the full power of multi-core systems, fairly new approaches attempt to solve this situation. Some of these approaches are considered well publicly tested and are widely used at the time of writing this document. The objectives for this study are to check which ones of these approaches achieve the better scalability in accordance to the number of handled requests, and to what extent those approaches utilize the multi-core power compared to the raw Node.js environment with the normal CPU scheduling.

**Keywords:** Node.js, parallel computing, multi-core systems.
# Contents

Glossary

List of Figures/Tables

1 Introduction
   1.1 Background and Considered Attempts
   1.2 Related Studies
   1.3 Problem Formulation
   1.4 Motivation
   1.5 Objectives
   1.6 Scope/Limitation
   1.7 Target Group
   1.8 Outline

2 Method
   2.1 Test Code Example Implementations
   2.2 Experiment Environment Layout/Specifications
   2.3 Experiment Phases and Additional Variables
   2.4 Reliability and Validity
   2.5 Ethical Considerations

3 Results
   3.1 Setting Run-levels Phase Results
   3.2 Relaxed Phase Results
   3.3 Exhausted Phase Results

4 Analysis
   4.1 Analyzing Relaxed Phase
   4.2 Analyzing Exhausted Phase
   4.3 Utilization of the Multi-core Power

5 Discussion

6 Conclusion and Future Work

References
Glossary

The following terms and acronyms are used in this report:

- **Node.js**: is a JavaScript runtime environment which uses an event-driven, non-blocking I/O model that makes it lightweight and efficient [1].
- **‘Cluster’ module**: is a Node.js module that helps in creating child processes to take advantage of multi-core systems [2].
- **PM2**: is a popular process manager for Node.js [3].
- **JMeter**: full name is ‘Apache JMeter’, is a Java application designed to load test functional behavior and measure performance of the tested resource [4].
- **Htop**: is an interactive process viewer for Unix-like systems. It is used in this study, due to its reputation and popularity, to measure CPU performance [5].
- **NGINX**: is an open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more [6].
List of Figures/Tables

Here, we introduce a list of the figures and tables presented in this report:

**Figure 1.6.1:** Intel code name Cedarview Atom features. Source: [17]. ........................................... 5
**Figure 2.1.1:** Test code for the raw Node.js approach. ................................................................. 7
**Figure 2.1.2:** Test code for the ‘cluster’ module approach. ................................................................. 7
**Figure 2.1.3:** Test code for the PM2 approach. ....................................................................................... 8
**Figure 2.2.1:** Experiment setup overview. ............................................................................................ 9
**Figure 2.2.2:** Experiment environment’s deployment diagram. ............................................................ 9
**Table 2.2.1:** Experiment’s server hardware/software specifications. ..................................................... 9
**Table 2.2.2:** Experiment’s testing client hardware/software specifications ............................................. 9
**Table 2.3.1:** Experiment’s variable matrix. ......................................................................................... 10
**Figure 3.1.1:** Four CPU usage percentage readings (with 4 second interval) for first phase. Running at ‘relaxed run-level’. .......................................................... 13
**Figure 3.1.2:** Number of simulated users (threads) for first phase while running at ‘relaxed run-level’. .................................................................................................................. 13
**Figure 3.1.3:** Five CPU usage percentage readings (with 10 second interval) for first phase. Running at ‘exhausted run-level’. .................................................................................. 14
**Figure 3.1.4:** Number of simulated users (threads) while running at ‘exhausted run-level’. .................... 14
**Figure 3.2.1:** Four CPU usage percentage readings (with 4 second interval) for second phase raw Node.js, Running at ‘relaxed run-level’. .............................................. 14
**Figure 3.2.2:** Response times for second phase raw Node.js while running at ‘relaxed run-level’. .......... 15
**Figure 3.2.3:** Response count for second phase raw Node.js while running at ‘relaxed run-level’. ............. 15
**Figure 3.2.4:** Four CPU usage percentage readings (with 4 second interval) for second phase raw Node.js, Running at ‘relaxed run-level’ multiplied by 2. ................................. 15
**Figure 3.2.5:** Response times for second phase raw Node.js while running at ‘relaxed run-level’ multiplied by 2. .............................................................................................. 15
**Figure 3.2.6:** Response count for second phase raw Node.js while running at ‘relaxed run-level’ multiplied by 2. .............................................................................................. 16
**Figure 3.2.7:** Four CPU usage percentage readings (with 4 second interval) for second phase ‘cluster’ approach. Running at ‘relaxed run-level’. .................................................... 16
**Figure 3.2.8:** Response times for second phase ‘cluster’ approach while running at ‘relaxed run-level’. .................................................................................................................. 16
**Figure 3.2.9:** Response count for second phase ‘cluster’ approach while running at ‘relaxed run-level’. .................................................................................................................. 16
**Figure 3.2.10:** Four CPU usage percentage readings (with 4 second interval) for second phase ‘cluster’ approach. Running at ‘relaxed run-level’ multiplied by 2. ................................. 17
**Figure 3.2.11:** Response times for second phase ‘cluster’ approach while running at ‘relaxed run-level’ multiplied by 2. .............................................................................................. 17
**Figure 3.2.12:** Response count for second phase ‘cluster’ approach while running at ‘relaxed run-level’ multiplied by 2. .............................................................................................. 17
**Figure 3.2.13:** Four CPU usage percentage readings (with 4 second interval) for second phase ‘PM2’ approach. Running at ‘relaxed run-level’. ....................................................... 18
Figure 3.2.14: Response times for second phase ‘PM2’ approach while running at ‘relaxed run-level’.

Figure 3.2.15: Response count for second phase ‘PM2’ approach while running at ‘relaxed run-level’.

Figure 3.2.16: Four CPU usage percentage readings (with 4 second interval) for second phase ‘PM2’ approach. Running at ‘relaxed run-level’ multiplied by 2.

Figure 3.2.17: Response times for second phase ‘PM2’ approach while running at ‘relaxed run-level’ multiplied by 2.

Figure 3.2.18: Response count for second phase ‘PM2’ approach while running at ‘relaxed run-level’ multiplied by 2.

Figure 3.3.1: Five CPU usage percentage readings (with 10 second interval) for third phase with raw Node.js. Running at ‘exhausted run-level’.

Figure 3.3.2: Response times for third phase with raw Node.js while running at ‘exhausted run-level’.

Figure 3.3.3: Response count for third phase with raw Node.js while running at ‘exhausted run-level’.

Figure 3.3.4: Five CPU usage percentage readings (with 10 second interval) for third phase with ‘cluster’ approach. Running at ‘exhausted run-level’ multiplied by 2.

Figure 3.3.5: Response times for third phase with ‘cluster’ approach while running at ‘exhausted run-level’ multiplied by 2.

Figure 3.3.6: Response count for third phase with ‘cluster’ approach while running at ‘exhausted run-level’ multiplied by 2.

Figure 3.3.7: Five CPU usage percentage readings (with 10 second interval) for third phase with ‘PM2’ approach. Running at ‘exhausted run-level’ multiplied by 2.

Figure 3.3.8: Response times for third phase with ‘PM2’ approach while running at ‘exhausted run-level’ multiplied by 2.

Figure 3.3.9: Response count for third phase with ‘PM2’ approach while running at ‘exhausted run-level’ multiplied by 2.

Figure 4.1.1: Response count comparison for all approaches while running at ‘relaxed run-level’.

Figure 4.1.2: Response time comparison for all approaches while running at ‘relaxed run-level’ multiplied by 2.

Figure 4.2.1: Response time comparison for ‘cluster’ and ‘PM2’ approaches while running at ‘exhausted run-level’ multiplied by 2.

Figure 4.2.2: Response count comparison for ‘cluster’ and ‘PM2’ approaches while running at ‘exhausted run-level’ multiplied by 2.

Table 4.3.1: Utilization of the multi-core power (percent truncated to max one decimal places).
1 Introduction

Node.js arrived as a single-threaded and event-loop driven JavaScript runtime solution [7] for small to middle sized web applications; and even for large sized web applications too, with the help of horizontal scaling that can be achieved with the help of a load balancer like NGINX. This essential feature from Node.js helps web applications grow easily when access demands grow, by simply allowing the addition of more hardware to meet these demands.

1.1 Background and Considered Attempts

The continuous architectural advancement in multi-core processors demands that thread level parallelism being exploited effectively and efficiently [8], but since Node.js processes are essentially single threaded [7], we will not enjoy the full power of our multi-core muscles when using Node.js. Thankfully, the Node.js team, and other interested software organizations, were aware of such situation, and prepared attempts to solve it.

There are several software organizations that are interested in multi-core utilization in Node.js. Here, we review some projects that have interests in this subject (discontinued projects will not be mentioned nor considered):

- **‘Cluster’ Module**: The ‘cluster’ module is a relatively new attempt that is developed to take advantage of multi-core systems by launching a cluster of Node.js single threaded processes of the same application. This module is considered in a stable stage (not experimental) now and it became one of the core modules of Node.js, and that’s why it was included in this study.

- **PM2**: PM2 is a relatively new process manager for Node.js. It is considered popular due to its many features, and it has its own attempt to utilize the full power of a multi-core CPU as one of those features. PM2 (and its utilization of the multi-core power) is also considered in a stable stage (not experimental) these days and that’s why it was included in this study too.

- **Napa.js**: Napa.js is a project led by Microsoft to provide a multi-threaded JavaScript runtime that complements Node.js [9]. It has its own way to utilize all CPU cores by introducing the concept of the ‘zone’. Every process can contain multiple zones, that in turn can contain several JavaScript workers that run the same JavaScript code [10]. Although this project is led and supported by the powerful Microsoft, it is still in its early stages (at the time of writing, the latest version had been v0.2.3); so, it was not fair to include it in this study.

1.2 Related Studies

This research falls into the broader topic of ‘parallel computing’; and in our case it deals with the way Node.js handles it on one machine, and to what
extent. Although there are several studies that handle several aspects of Node.js, the ones that deal with parallel computing and multi-core utilization are scarce. Some of the studies compare Node.js with other development technologies [11] which falls outside of the scope of this study, while others, inspired by the event-loop driven approach that Node.js adopted, presented a similar approach with a different programming language than JavaScript [12] (which also falls outside of the scope of this study). The following studies have some extent of relevance to this study because their focus is only on Node.js and JavaScript:

- **Node.js: Using JavaScript to Build High-Performance Network Programs** [13]: The authors of this research explained the code conventions used in Node.js due to its event-driven nature, and (among other subjects) the need to use non-blocking calls when programming for Node.js. The idea about the utilization of the multi-core power is also discussed; but, unfortunately, the proposed solution is a project that was discontinued long time ago (not to mention that it was in its early stages v0.2.3).

- **GEMs: Shared-Memory Parallel Programming for Node.js** [14]: Although this study deals with parallel programming for Node.js, its main focus is to propose a solution for message passing and shared-memory parallelism between running processes within a single multi-core machine. The study’s focus is not the same as ours; so, we cannot build on its results.

### 1.3 Problem Formulation

As was checked earlier, only the ‘cluster’ module and the ‘PM2’ approaches were part of this study, since they are the current stable/active ‘go-to’ solution when the utilization of the multi-core power is required. So, the three aspects that were studied and compared are:

1. The use of raw Node.js environment and how far it utilizes the multi-core power.
2. The use of ‘cluster’ module and how far it utilizes the multi-core power.
3. The use of PM2 and how far its approach utilizes the multi-core power.

Do the two attempts solve the problem of exploiting all the available CPU cores (compared to the raw Node.js environment) as those attempts advertise? And to what extent? This was what this study addressed.

### 1.4 Motivation

JavaScript has proven itself to be the most popular programming language [15]. Additionally, Node.js gave it more momentum by making it a server-
side programming language too. There is also the fact that even cheap hardware these days enjoys the multi-core power to a good extent. This raises the need for our Node.js based web applications to exploit every resource available, including every core of every CPU on every machine our application is running on. Additionally, with decently large web applications, spreading the load on a large number of small servers can be a hassle to maintain, and we might be better off spreading our web application on a smaller number of high end servers with state-of-the-art multi-core power.

1.5 Objectives

| O1   | Prepare the experiment environment and the test code example implementations that the experiments run with. |
| O2   | In the initial experiment phase, determine the level of utilization of the multi-core power the raw Node.js environment achieves, and make this level as a threshold for the other considered attempts/approaches. |
| O3   | Check how far the other attempts/approaches extend the utilization of multi-core power compared to the threshold set by objective O2. |
| O4   | In light of the experiment result, determine (up until the time of the experiment) the best approach (if any) and specify the best situation every approach excels at. |

1.5.1 Approach

“The commonly used techniques for the performance evaluation of computing systems include: measurement, mathematical modeling, and simulation [16].” This study used a measurement technique to achieve the objectives, where, first, some Node.js test code example implementations were developed that conform to the ‘cluster’ module and the PM2 conditions. Then, experiment phases were done to check if (and to what extent) they utilize the multi-core power; and then compared the results between them and, also, between them and the raw Node.js environment. The scenarios that were studied here (and applied on the aspects mentioned in ‘1.3 Problem Formulation’ subsection) are:

1- The level of utilization (are all CPU-cores utilized in a relatively equal manner) under average load on machines with real multi-core CPU (not considering hyper-threading).

2- The level of utilization (are all CPU-cores utilized in a relatively equal manner) under high load on machines with real multi-core CPU (not considering hyper-threading).

So, the criteria that were checked against were the utilization of cores, and the level of load that the application/test-code endured.
1.5.2 Expected results

After giving the test example implementations, and after every aspect tested and measured according to the specified scenarios, the expected results:

- An evaluation of averages for every aspect/approach (like CPU-core usage and response time) that would be presented in charts, which would help in comparing the utilization of the multi-core power for every aspect/approach.

- Other recommended steps (if any) that would help in making the utilization better when tests show that they give better results.

1.6 Scope/Limitation

To keep the report’s result beneficial and fair, it should study the use of the Node.js environment the way it is meant to be used, which is to execute JavaScript code on the server-side while providing services to clients. Additionally, due to how powerful the CPU power had become these days (even with entry level CPUs), and the maximum number of ephemeral TCP ports (short-lived ports allocated automatically for client communications), this study had the following limitations:

- The test code example implementations did not contain any CPU intensive code, because this study’s purpose was to simulate the wide use of Node.js to run server-side web applications that majorly deliver HTML content.

- The CPU used in this study was an entry level dual-core CPU (the Intel code name Cedarview - Figure 1.6.1 (below on page 5) - Atom D2700), because a more powerful CPU with higher number of cores would require the amount of several testing hardware clients (PCs) for testing the performance of even a single core of the CPU, while the number of testing hardware clients will multiply when the number of tested cores increase (which would have exhausted the resources of this study). To explain, every operating system assigns an ephemeral TCP ports for every outbound client communication. The maximum number of ports for IP communication is 65535 (port number 0 is reserved in TCP), and the operating system can be configured to use up-to this number of ports (less is the default for Windows OS). A single testing hardware client could barely exhaust a single core of the Intel Atom D2700 CPU running a test example process; and using a more powerful CPU with more cores would require a multiplied number of testing hardware clients. Nevertheless, the results of this study can be easily reproduced on a more powerful CPU but needs some extra resources.

- This study used a Linux distribution as an operating system environment for running the test code example implementations, because Linux is more flexible in controlling the running process and
disabling the hyper-threading feature for the CPU. The following command was used to disable the hyper-threaded CPU-core virtual devices in Linux:

```
echo 1 > /sys/devices/system/cpu/cpu#/online
```

where ‘#’ would be the index of the hyper-threaded CPU-core.

1.7 Target Group

Target group for this study would be small to medium sized companies and serious developers seeking to achieve a better utilization of their CPU power when using Node.js.

1.8 Outline

In the following sections; at first, the method used in this study was presented in the next section, with details about how it was organized and conducted, including how the reliability and validity of this study was achieved. After that, the result of the conducted experiment was presented in the ‘Results’ section, followed by the analysis based on those results and what could be seen in them. The ‘Discussion’ section followed, where it was discussed how the objectives of this study were achieved. After that, the collected conclusions of this study were presented, concluding the findings of this study.
2 Method

After the initial necessary literature review that was done to check the previous studies that relate to the scope of this study, this study used the controlled experiment method by conducting a series of tests on Node.js running some test code example implementations in a controlled environment (according to the aforementioned aspects in the ‘1.3 Problem Formulation’ subsection).

For our experiment to be scientifically conducted (and worthy of being a controlled experiment), it should be done under controlled conditions, where only one affecting factor (independent variable) would be changed for every result measurement done; while other affecting factors would remain unchanged/constant. Additionally, a control group should be elected (where the changing independent variable would be in a state that is considered initial) to set it as a threshold to compare against, while the other experiment targets (that would hold other values for the changing independent variable) would serve as the experimental group. As for the threats to the validity of the controlled experiment, precautions should be taken to eliminate the effect of any extraneous variable (more on that in the ‘2.4 Reliability and Validity’ subsection below).

The independent variable in this experiment was the ‘approach used to utilize the multi-core power’ (‘approach used’ for short) which would be the raw Node.js environment approach (that would serve as the control group), the ‘cluster’ module approach, or the PM2 approach.

The dependent variables, on the other hand, were the following:
- CPU-core usage percentage: this dependent variable was important to measure the level of every CPU-core usage when the ‘approach used’ independent variable was under test. This variable was provided by the ‘htop’ command line utility for Linux.
- Average CPU usage percentage: represents the average usage of all CPU-cores in the test subject. This variable was also provided by the ‘htop’ command line utility for Linux.
- Average response time (measured in milliseconds): this variable was provided with the help of the ‘JMeter’ testing tool; and was considered as a supporter for the previously mentioned variables.
- Average number of responses/second: also, was provided with the help of the ‘JMeter’ testing tool; and was considered as a supporter for the previously mentioned variables.

2.1 Test Code Example Implementations

To perform the experiment, an application (in the form of a JavaScript code) had to be provided for every instance of the ‘approach used’ independent variable.
Figure 2.1.1 (below) shows the code used to test the raw Node.js approach. This code was based on the standard recommended convention used by most of the web applications that use Node.js [18] and serves a standard HTML page to the client.

```javascript
// Load the 'express' module and create an express app object
let tmpApp = require('express')()

// Routes HTTP GET requests to root path handled by our arrow function that responds with '200 OK' HTML page
tmpApp.get('/', (req, resp, next) => resp.status(200).end(
  '<!DOCTYPE html><html><head><title>OK</title></head><body><h1>200 OK Random=' +
  (Math.random() * 1000 | 0) + ' // Generate a random (Gives CPU something to do to imitate real life page work)
  ')/h1></body></html>
'))

// Routes other requests and other path choice than root (responses with a '404 Not Found' HTML page)
tmpApp.use((req, resp, next) => resp.status(404).end(
  '<!DOCTYPE html><html><head><title>404</title></head><body><h1>404 Not Found</h1></body></html>
'))

// Final route in case there was an error to prevent application crash ('500 Internal Error' HTML page response)
tmpApp.use((err, req, resp, next) => resp.status(500).end(
  '<!DOCTYPE html><html><head><title>500</title></head><body><h1>500 Internal Server Error</h1></body></html>
'))

tmpApp.listen(4000) // Start listening
```

Figure 2.1.1: Test code for the raw Node.js approach.

---

```javascript
// Load the 'cluster' module and create a cluster object that handles the different processes
let tmpClust = require('cluster')()

// Check if the current process is the master that spawns a worker process for
// every CPU-core (and then sleeps).
if (tmpClust.isMaster) {
  // If it is the master process
  let tmpCnt = require('os').cpus().length // Get the number of CPU-cores

  for (let i = 0; i < tmpCnt; i++) {
    // Loop and spawn a worker process for every CPU-core
    tmpClust.fork()
  }

  // Add an event listener for the 'exit' event that indicate a worker dying
  // and spawn a new worker process if that happens
  tmpClust.addListener('exit', () => tmpClust.fork())
} else {
  console.log('Worker [' + tmpClust.worker.id + '] Started') // To indicate a worker start

  // Load the 'express' module and create an express app object
  let tmpApp = require('express')()

  // Routes HTTP GET requests to root path handled by our arrow function that responds with '200 OK' HTML page
  tmpApp.get('/', (req, resp, next) => resp.status(200).end(
    '<!DOCTYPE html><html><head><title>OK</title></head><body><h1>200 OK from process [' +
    tmpClust.worker.id + '] Random=' +
    (Math.random() * 1000 | 0) + ' // Generate a random (Gives CPU something to do to imitate real life page work)
    ')/h1></body></html>
  '))

  // Routes other requests and other path choice than root (responses with a '404 Not Found' HTML page)
  tmpApp.use((req, resp, next) => resp.status(404).end(
    '<!DOCTYPE html><html><head><title>404</title></head><body><h1>404 Not Found</h1></body></html>
  '))

  // Final route in case there was an error to prevent application crash ('500 Internal Error' HTML page response)
  tmpApp.use((err, req, resp, next) => resp.status(500).end(
    '<!DOCTYPE html><html><head><title>500</title></head><body><h1>500 Internal Server Error</h1></body></html>
  '))

  tmpApp.listen(4000) // Start listening
}
```

Figure 2.1.2: Test code for the 'cluster' module approach.
Figure 2.1.2 (above on page 7) shows the code used to test the ‘cluster’ module approach. In its ‘worker’ part (the part inside the ‘else’ block) it is basically the same as the code for the raw Node.js approach, but a part where the ID of the process that handles the current request was added to the response. This ID helped in checking that every process handled at least 90% of its load share in normal situations (results of this check were used to initially check the test subject and were not related to the study/results directly).

Figure 2.1.3 (below) shows the code used to test the ‘PM2’ approach. It is almost identical to the code for Node.js approach, but, as in the ‘cluster’ module approach, a part where the ID of the process that handles the current request was added to the response for the same purpose.

// Load the ‘express’ module and create an express app object 
let tmpApp = require('express')()

// Routes HTTP GET requests to the root path handled by our arrow function that responses with ‘200 OK’ HTML page
// '200 OK from process [' + process.env.pm_id + ']' + // To display the ID for the process that handles this request
// 'Random=[' + (Math.random() * 1000 | 0) + ']' + // Generate a random (Gives CPU something to do to imitate real life page work)
// '<h1>200 OK from process [<process.env.pm_id>]</h1> Random=[Math.random() * 1000 | 0]'

// Routes other requests and other path choice than root (responses with a ‘404 Not Found’ HTML page)
tmpApp.use((req, resp, next) => resp.status(404).end('404 Not Found'))

// Final route in case there was an error to prevent application crash (‘500 Internal Error’ HTML page response)
tmpApp.use((err, req, resp, next) => resp.status(500).end('500 Internal Server Error'))

tmpApp.listen(4000) // Start listening

Figure 2.1.3: Test code for the PM2 approach.

2.2 Experiment Environment Layout/Specifications

An overview of the experiment setup can be seen on Figure 2.2.1 (below on page 9) that shows how JMeter works by creating a specific number of simulated users that sends and receives requests to and from the tested subject (the tested server with Node.js running the test code example implementation). Figure 2.2.2 (below on page 9) shows the deployment diagram of the experiment environment, where we deployed the Node.js runtime on the server hardware to be tested, while the JMeter testing tool was deployed on a client computer. The server’s hardware/software specifications can be seen in Table 2.2.1 (below on page 9), while the testing client hardware/software specifications can be seen in Table 2.2.2 (below on page 9).
Figure 2.2.1: Experiment setup overview.

Figure 2.2.2: Experiment environment’s deployment diagram.

<table>
<thead>
<tr>
<th>Table 2.2.1: Experiment’s server hardware/software specifications.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor</strong></td>
</tr>
<tr>
<td><strong>RAM</strong></td>
</tr>
<tr>
<td><strong>Operating System</strong></td>
</tr>
<tr>
<td><strong>Drive</strong></td>
</tr>
<tr>
<td><strong>LAN</strong></td>
</tr>
<tr>
<td><strong>Node.js</strong></td>
</tr>
<tr>
<td><strong>PM2</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 2.2.2: Experiment’s testing client hardware/software specifications.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Processor</strong></td>
</tr>
<tr>
<td><strong>RAM</strong></td>
</tr>
<tr>
<td><strong>Operating System</strong></td>
</tr>
<tr>
<td><strong>LAN</strong></td>
</tr>
<tr>
<td><strong>JMeter</strong></td>
</tr>
</tbody>
</table>
2.3  Experiment Phases and Additional Variables

The JMeter test tool provides a technique, that was used in our experiment, where it simulates a specific number of users (threads) making requests to the test subject, and every user makes requests to the test subject continuously with a specified interval (for this controlled experiment, the interval was set to one millisecond). This gives the flexibility to induce any desired load on the test subject. So, to achieve determined results where we can replicate this experiment on different hardware (and with the help of JMeter features), two new independent variables were added to this experiment:

- Relaxed run-level: represents the number of users firing requests to the raw Node.js test subject while the CPU-core usage percentage for the highest used core is optimally between 75% to 85%.
- Exhausted run-level: represents the maximum number of users firing requests to the raw Node.js test subject without any sign of the test subject failing to respond to any test request-sample (a request-sample is a single request made by a single simulated user in JMeter).

These two variables were considered independent variable because they only change when the hardware, that the test is conducted on, changes. Every ‘approach used’ will endure the test experiment based on these two variables. So, with the introduction of these two additional variables, we ended up with the variables for this experiment that we can see in Table 2.3.1 (below); and our experiment would be conducted in three phases (described in the following three subsections).

<table>
<thead>
<tr>
<th>Independent Variables</th>
<th>The Variable</th>
<th>Short description</th>
<th>Unit (if applicable)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Approach used</td>
<td>One of the approaches: raw Node.js, ‘cluster’ module, or PM2.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Relaxed run-level</td>
<td>The number of simulated users firing requests to the ‘raw Node.js’ while the ‘CPU-core usage’ for the highest used core is optimally between 75% to 85%.</td>
<td>No. of users</td>
</tr>
<tr>
<td></td>
<td>Exhausted run-level</td>
<td>The maximum number of simulated users firing requests to the ‘raw Node.js’ without any sign of ‘raw Node.js’ failing to respond.</td>
<td>No. of users</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependent Variables</th>
<th>The Variable</th>
<th>Short description</th>
<th>Unit (if applicable)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU-core usage</td>
<td>Measures the level of every CPU-core usage for every ‘approach used’.</td>
<td>Percent</td>
</tr>
<tr>
<td></td>
<td>Average CPU usage</td>
<td>The average usage of all CPU-cores in the test subject for every ‘approach used’.</td>
<td>Percent</td>
</tr>
<tr>
<td></td>
<td>Average response time</td>
<td>The average response time for the requests sent by the ‘JMeter’ testing tool to every ‘approach used’.</td>
<td>Millisecond</td>
</tr>
<tr>
<td></td>
<td>Average number of responses/second</td>
<td>The average number of responses/second for the requests sent by the ‘JMeter’ testing tool to every ‘approach used’.</td>
<td>Resp/Second</td>
</tr>
</tbody>
</table>

Table 2.3.1: Experiment’s variable matrix.
2.3.1 Setting Run-levels Phase

In this phase, we initiated the experiment by setting the run-level variables that the rest of the experiment would be based upon.

For the ‘relaxed run-level’: it should give the same consistent CPU-core usage (75% to 85%) for 20 second test session (with additional 5 seconds to give the simulated users an initial 5 seconds to start firing) on the raw Node.js approach; and should give similar results five times in a row.

As for the ‘exhausted run-level’: to get this number, we kept increasing the number of users with 50 user increments and ran the test on the raw Node.js approach for 60 seconds (the actual JMeter test was at least 70 seconds to give the simulated users an initial 5 seconds to start firing in the beginning and 5 seconds to die at the end) until we reached a level where we got at least a single error/no-response from the tested subject. We then stepped back with 50 user decrement and tested that final level five times to be sure that no error/no-response on that final level.

2.3.2 Relaxed Phase

Every instance of the ‘approach used’ independent variable would be tested, once according to the ‘relaxed run-level’ and another according to the multiplication of the ‘relaxed run-level’ with the number of CPU-cores, while extracting the results according to the dependent variables. Each test in this phase would run for 25 seconds (10 seconds for the simulated users start and stabilize + 10 second for collecting data + 5 seconds for the test to finalize).

2.3.3 Exhausted Phase

The raw Node.js approach would be tested according to the ‘exhausted run-level’ and the other two approaches would be tested according to the multiplication of the ‘exhausted run-level’ with the number of CPU-cores; while extracting the results according to the dependent variables. Each test in this phase would run for 80 seconds (15 seconds for the simulated users start and stabilize + 60 second for collecting data + 5 seconds for the test to finalize).

2.4 Reliability and Validity

As a controlled experiment, this study took into consideration the threats to internal and external validity. For the internal validity, and since there was no natural or human participants/test-subjects, some factors (like mortality, maturation, or design contamination) were already considered a non-threat. In our case, a considerable threat to internal validity was ‘instrumentation’, where a change in the testing or measuring instrumentation during the study could affect the results. To avoid that, we used the same testing and measuring instrumentation during the whole study. It was also considered to eliminate the effect of any ‘extraneous variable’ (which could also be
considered a ‘history’ threat to validity) by running the tested subjects on a
stripped Linux distribution that runs nothing but the tested subject and
without any scheduled tasks (Node.js settings were also used to prevent the
garbage collection from running).

As for the threats to external validity, the previously mentioned
‘relaxed run-level’ and ‘exhausted run-level’ independent variables were
introduced to ensure that the ‘interaction of history and treatment’ threat to
external validity (whether we could get the same results today and in the
future) is avoided. Those two independent variables would help in
reproducing similar results later on a different experiment environment
(mainly, different/future hardware with different/future CPU) by setting
thresholds that the rest of the experiment would build on (this would also
help in achieving the maximum reliability by helping in reproducing similar
results when replicating the experiment). All that, without affecting other
aspects of the experiment.

2.5 Ethical Considerations
This experiment did not touch on any aspect that may directly require any
ethical consideration. There were no human participants in the experiment;
but since its results may lead to a lower number of hardware used (severs
running a web application) to achieve the same or better results, this in turn
may lead to a lower number of human resources required for administration.
This might be an indirect reason to consider the ethical aspect in this case,
because it might lead to a lower number of human resources hired.
3 Results

In this section, the variables resulted from each phase of the experiment would be shown independently for every phase, because the second and third phases were based on the results of the first phase.

3.1 Setting Run-levels Phase Results

In this phase, to reach the condition set for ‘relaxed run-level’, the needed number of simulated users in JMeter was ‘4’ with one millisecond interval between every request a simulated user fired (over and under this level would not achieve the previously set condition). Figure 3.1.1 (below) and Figure 3.1.2 (below) show the stable level of simulated users (threads) with the consistent CPU-core usage.

As for the ‘exhausted run-level’, the needed number of simulated users in JMeter according to its condition was ‘1450’ (one millisecond interval between every simulated user’s request is always used in this experiment). Figure 3.1.3 (below on page 14) and Figure 3.1.4 (below on page 14) show the level of simulated users (threads) with the inconsistent CPU-core usage.

<table>
<thead>
<tr>
<th>Number</th>
<th>CPU Usage %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>80.6%</td>
</tr>
<tr>
<td>2</td>
<td>2.0%</td>
</tr>
<tr>
<td>Avg</td>
<td>40.5%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number</th>
<th>CPU Usage %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>76.6%</td>
</tr>
<tr>
<td>2</td>
<td>3.3%</td>
</tr>
<tr>
<td>Avg</td>
<td>39.8%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number</th>
<th>CPU Usage %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>78.9%</td>
</tr>
<tr>
<td>2</td>
<td>2.0%</td>
</tr>
<tr>
<td>Avg</td>
<td>39.8%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number</th>
<th>CPU Usage %</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>75.0%</td>
</tr>
<tr>
<td>2</td>
<td>2.6%</td>
</tr>
<tr>
<td>Avg</td>
<td>38.0%</td>
</tr>
</tbody>
</table>

Figure 3.1.1: Four CPU usage percentage readings (with 4 second interval) for first phase. Running at ‘relaxed run-level’.

Figure 3.1.2: Number of simulated users (threads) for first phase while running at ‘relaxed run-level’.
3.2 Relaxed Phase Results

Figure 3.2.1 (below), Figure 3.2.2 (below on page 15), and Figure 3.2.3 (below on page 15) show the CPU-core usage and the responses count and times when running the raw Node.js environment at ‘relaxed run-level’.

Figure 3.2.1: Four CPU usage percentage readings (with 4 second interval) for second phase raw Node.js. Running at ‘relaxed run-level’.
Figure 3.2.2: Response times for second phase raw Node.js while running at 'relaxed run-level'.

Figure 3.2.3: Response count for second phase raw Node.js while running at 'relaxed run-level'.

Figure 3.2.4 (below), Figure 3.2.5 (below), and Figure 3.2.6 (below on page 16) show the CPU-core usage and the responses count and times when running the raw Node.js environment at 'relaxed run-level' multiplied by the number of cores (two in our case).

Figure 3.2.4: Four CPU usage percentage readings (with 4 second interval) for second phase raw Node.js. Running at 'relaxed run-level' multiplied by 2.

Figure 3.2.5: Response times for second phase raw Node.js while running at 'relaxed run-level' multiplied by 2.
Figure 3.2.6: Response count for second phase raw Node.js while running at ‘relaxed run-level’ multiplied by 2.

Figure 3.2.7 (below), Figure 3.2.8 (below), and Figure 3.2.9 (below) show the CPU-core usage and the responses count and times when running the ‘cluster’ approach at ‘relaxed run-level’.

Figure 3.2.7: Four CPU usage percentage readings (with 4 second interval) for second phase ‘cluster’ approach. Running at ‘relaxed run-level’.

Figure 3.2.8: Response times for second phase ‘cluster’ approach while running at ‘relaxed run-level’.

Figure 3.2.9: Response count for second phase ‘cluster’ approach while running at ‘relaxed run-level’.
Figure 3.2.10 (below), Figure 3.2.11 (below), and Figure 3.2.12 (below) show the CPU-core usage and the responses count and times when running the ‘cluster’ approach at ‘relaxed run-level’ multiplied by the number of cores (which is 2 in our case).

![Figure 3.2.10](image1)

Figure 3.2.10: Four CPU usage percentage readings (with 4 second interval) for second phase ‘cluster’ approach. Running at ‘relaxed run-level’ multiplied by 2.

![Figure 3.2.11](image2)

Figure 3.2.11: Response times for second phase ‘cluster’ approach while running at ‘relaxed run-level’ multiplied by 2.

![Figure 3.2.12](image3)

Figure 3.2.12: Response count for second phase ‘cluster’ approach while running at ‘relaxed run-level’ multiplied by 2.

Figure 3.2.13 (below on page 18), Figure 3.2.14 (below on page 18), and Figure 3.2.15 (below on page 18) show the CPU-core usage and the responses count and times when running the ‘PM2’ approach at ‘relaxed run-level’.
Figure 3.2.13: Four CPU usage percentage readings (with 4 second interval) for second phase ‘PM2’ approach. Running at ‘relaxed run-level’.

Figure 3.2.14: Response times for second phase ‘PM2’ approach while running at ‘relaxed run-level’.

Figure 3.2.15: Response count for second phase ‘PM2’ approach while running at ‘relaxed run-level’.

Figure 3.2.16 (below on page 19), Figure 3.2.17 (below on page 19), and Figure 3.2.18 (below on page 19) show the CPU-core usage and the responses count and times when running the ‘PM2’ approach at ‘relaxed run-level’ multiplied by the number of cores (which is 2 in our case).
### 3.3 Exhausted Phase Results

Figure 3.3.1 (below on page 20), Figure 3.3.2 (below on page 20), and Figure 3.3.3 (below on page 20) show the CPU-core usage and the responses count and times when running the raw Node.js environment at 'exhausted run-level' (we cannot run at 'exhausted run-level' multiplied by the number of core in this experiment on raw Node.js environment, because we reached the exhaustion level that the raw Node.js environment can handle without errors/no-responses). It is worth mentioning that the results in the Figure
3.3.1 (below) are taken with 10 seconds interval; and the CPU-core usage for every core had alternated between around 2% and 98% between the cores (with always half the total CPU usage around 54%).

Figure 3.3.1: Five CPU usage percentage readings (with 10 second interval) for third phase with raw Node.js. Running at ‘exhausted run-level’.

Figure 3.3.2: Response times for third phase with raw Node.js while running at ‘exhausted run-level’.

Figure 3.3.3: Response count for third phase with raw Node.js while running at ‘exhausted run-level’.
Figure 3.3.4 (below), Figure 3.3.5 (below), and Figure 3.3.6 (below) show the CPU-core usage and the responses count and times when running the ‘cluster’ approach at ‘exhausted run-level’ multiplied by the number of cores (which is 2 in our case).

Figure 3.3.4: Five CPU usage percentage readings (with 10 second interval) for third phase with ‘cluster’ approach. Running at ‘exhausted run-level’ multiplied by 2.

Figure 3.3.5: Response times for third phase with ‘cluster’ approach while running at ‘exhausted run-level’ multiplied by 2.

Figure 3.3.6: Response count for third phase with ‘cluster’ approach while running at ‘exhausted run-level’ multiplied by 2.
Figure 3.3.7 (below), Figure 3.3.8 (below), and Figure 3.3.9 (below) show the CPU-core usage and the responses count and times when running the ‘PM2’ approach at ‘exhausted run-level’ multiplied by the number of cores (which is 2 in our case).

Figure 3.3.7: Five CPU usage percentage readings (with 10 second interval) for third phase with ‘PM2’ approach. Running at ‘exhausted run-level’ multiplied by 2.

Figure 3.3.8: Response times for third phase with ‘PM2’ approach while running at ‘exhausted run-level’ multiplied by 2.

Figure 3.3.9: Response count for third phase with ‘PM2’ approach while running at ‘exhausted run-level’ multiplied by 2.
4 Analysis

When analyzing the results gathered from the relaxed phase, it was made sure to ignore the first 10 seconds (to give the simulated users the chance to take off and stabilize request firing) and the last 5 seconds (to avoid any disturbance when simulated users die), giving pure 10 seconds to base the results and analysis upon. Similar action was taken for the exhausted phase, but with 15 second at the beginning and 5 at the end, giving pure 60 seconds to base the results and analysis upon.

4.1 Analyzing Relaxed Phase

For this phase, beginning with ‘relaxed run-level’ applied: by comparing the CPU-core usage percentage (for single cores and their average) from Figure 3.2.1 (above on page 14), Figure 3.2.7 (above on page 16), and Figure 3.2.13 (above on page 18), while matching that with their related response times in Figure 3.2.2 (above on page 15), Figure 3.2.8 (above on page 16), and Figure 3.2.14 (above on page 18), and their related response count in Figure 3.2.3 (above on page 15), Figure 3.2.9 (above on page 16), and Figure 3.2.15 (above on page 18), it was noticed that while the CPU-core usage is focused on the first core in the case of the raw Node.js approach, the other two approaches show a decent usage of the second core with focus still on the first core (but the average CPU-usage is the same in all approaches which is roughly 40%). It was noticed also that, at this level of relaxed load, the raw Node.js approach showed a slight advantage response-count-wise; averaging at 1.3 milliseconds and 1875 resp/sec (while the other two approaches averaged at 1.3 milliseconds and 1660-1700 resp/sec). Although this difference (in responses/second) was not big, it indicates that, in a relaxed load on a single core, the raw Node.js approach showed a slight advantage (while there was no noticeable difference between ‘cluster’ and ‘PM2’ approaches). Figure 4.1.1 (below on page 24) gives a clear way to compare the response count results (where there is a difference, and raw Node.js showed a slight advantage) between the three approaches running at ‘relaxed run-level’.

As for when ‘relaxed run-level’ multiplied by the number of cores (2 in our case) was applied: in this level, we are pushing a single core out of its comfort zone. By comparing the CPU-core usage percentage (for single cores and their average) from Figure 3.2.4 (above on page 15), Figure 3.2.10 (above on page 17), and Figure 3.2.16 (above on page 19), while matching that with their related response times in Figure 3.2.5 (above on page 15), Figure 3.2.11 (above on page 17), and Figure 3.2.17 (above on page 19), and their related response count in Figure 3.2.6 (above on page 16), Figure 3.2.12 (above on page 17), and Figure 3.2.18 (above on page 19), it was noticed that, in raw Node.js approach, the second CPU-core is still hardly used averaging the total CPU usage to 50%. Additionally, in raw Node.js
approach, while the first core was being maximum used (and the second barely used), the response time for the raw Node.js approach had suffered a drop in the response time resulting in an average of 2.2 milliseconds compared to the average of both ‘cluster’ and ‘PM2’ approaches which averaged around 1.6 milliseconds for both. This was shown also in the average response count, where it was 2540 resp/sec for raw Node.js, while it was 3075 resp/sec for ‘cluster’ and 3025 resp/sec for ‘PM2’. This showed that, here too, the ‘cluster’ and ‘PM2’ approaches showed the same level of CPU-core power utilization, while the raw Node.js approach showed no utilization. Figure 4.1.2 (below on page 25) can be checked to compare the response times between the three approaches running at ‘relaxed run-level’ multiplied by the number of cores.

Figure 4.1.1: Response count comparison for all approaches while running at ‘relaxed run-level’.
4.2 Analyzing Exhausted Phase

This phase’s results were intended to check how the different instances of the ‘approach used’ independent variable handle the exhaustion level of load. Since the actual exhaustion level for the ‘cluster’ and ‘PM2’ cannot be achieved in our case (as described in ‘1.6 Scope/Limitation’ subsection), doubling the exhaustion level of the raw Node.js approach seemed as a decent level to measure the ‘cluster’ and ‘PM2’ approaches against.

By checking the CPU-core usage percentage shown in Figure 3.3.1 (above on page 20), it was noticed that, in the case of raw Node.js, there was no utilization of the full power of the CPU (whether single-core-wise or total-CPU-wise), while showing an inconsistency and volatile CPU-core usage. The average CPU usage was between 50% and 56% (and did not exceed that) all the time during this phase for raw Node.js. The inconsistency and volatility in raw Node.js CPU-core usage was reflected on the response times and count for raw Node.js, where the Figure 3.3.2 (above on page 20) and Figure 3.3.3 (above on page 20) show the inconsistency and volatility in the response times and count respectively. Although this was expected at this point, it was necessary to test and show the result for the raw Node.js when it reached the exhaust level; which ended the role of raw Node.js in this experiment, while the ‘cluster’ module and ‘PM2’ approaches were left to compete.

By comparing the CPU-core usage percentage for the ‘cluster’ module and ‘PM2’ approaches shown in Figure 3.3.4 (above on page 21), and Figure 3.3.7 (above on page 22), while matching that with their related response times in Figure 3.3.5 (above on page 21), and Figure 3.3.8 (above on page 22), and their related response count in Figure 3.3.6 (above on page 21), and Figure 3.3.9 (above on page 22), it was noticed that the response times results showed an advantage for the ‘cluster’ module approach; averaging to 2790
milliseconds compared to 3190 milliseconds for the ‘PM2’ approach. This was also backed by the response count results, where the ‘cluster’ module approach averaged around 1050 resp/sec, while the ‘PM2’ approach averaged around 910 resp/sec. This is, of course, due to the slightly more CPU usage as the figures show. Figure 4.2.1 (below) and Figure 4.2.2 (below on page 27) give a clear way to compare the response times and count results respectively between the ‘cluster’ and ‘PM2’ approaches running at ‘exhausted run-level’ multiplied by the number of cores.

The analysis of this phase shows that the raw Node.js has no utilization of the CPU-core power (in fact, it could not reach the average number of responses it already reached in the previous phase).

On the other hand, the ‘cluster’ and ‘PM2’ approaches reached an optimal level of CPU-core muscle utilization showing stable results under the ‘exhausted run-level’ multiplied by ‘2’. The ‘cluster’ module approach had an advantage over the ‘PM2’ (although, it used 5% to 8% more CPU power on every core) achieving an average of 1050 resp/sec compared to 910 resp/sec for ‘PM2’ (around 140 more responses in average). The response time for the ‘cluster’ approach was also better; averaging 2790 milliseconds while it was 3190 milliseconds for ‘PM2’ approach (around 400 milliseconds difference in average).

**Figure 4.2.1:** Response time comparison for ‘cluster’ and ‘PM2’ approaches while running at ‘exhausted run-level’ multiplied by 2.
4.3 Utilization of the Multi-core Power

To link the analysis with the aspects and questions in the ‘1.3 Problem Formulation’ subsection, the level of utilization of the multi-core power from the analysis of the second and third phases was categorized in the following three amounts of load (with the help of the ‘relaxed run-level’ and ‘exhausted run-level’ independent variables):

- At relaxed load: Here, our intention was to apply an amount of load that could be, in a relaxed manner, handled by a single CPU-core and check the utilization of the multi-core power for all approaches. All the approaches were exposed to the ‘relaxed run-level’ in this case.

- At average load (relaxed load multiplied by the number of CPU-cores): For this amount of load, our intention was to push a single core out of its comfort zone by applying an amount of load that a single CPU-core could not, in a relaxed manner, handle alone and induce a level of multi-core power utilization. All the approaches were exposed to the ‘relaxed run-level’ multiplied by the number of CPU-cores in this case.

- At exhausting load: Here, the intention was to apply a maximal load (without any sign of failure to respond) on all approaches. In this case, the ‘raw Node.js’ approach was exposed to the ‘exhausted run-level’, while the other two approaches were exposed to the ‘relaxed run-level’ multiplied by the number of CPU-cores.
Table 4.3.1 (below) shows if and how far every approach utilizes the multicore power according to the experiment results and the analysis of the second and third phases.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Multi-core Utilization Exists</th>
<th>Higest Used CPU-core Usage (avg.)</th>
<th>Lowest Used CPU-core Usage (avg.)</th>
<th>Higest to Lowest Difference (avg.)</th>
<th>Overall CPU usage (avg.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relaxed Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raw Node.js</td>
<td>No (practically, only the first core used)</td>
<td>77.7% (core 1)</td>
<td>1.9% (core 2)</td>
<td>75.8%</td>
<td>39.8%</td>
</tr>
<tr>
<td>‘cluster’ module</td>
<td>Yes (considerable and steady usage of both cores)</td>
<td>68.6% (core 1)</td>
<td>11.7% (core 2)</td>
<td>56.9%</td>
<td>40.1%</td>
</tr>
<tr>
<td>‘PM2’ approach</td>
<td>Yes (considerable and steady usage of both cores)</td>
<td>73.9% (core 1)</td>
<td>18.9% (core 2)</td>
<td>55%</td>
<td>46.4%</td>
</tr>
<tr>
<td>Average Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raw Node.js</td>
<td>No (practically, only the first core used with half total CPU usage)</td>
<td>98.7% (core 1)</td>
<td>2.4% (core 2)</td>
<td>96.3%</td>
<td>50.5%</td>
</tr>
<tr>
<td>‘cluster’ module</td>
<td>Yes (considerable and steady usage of both cores)</td>
<td>89.9% (core 1)</td>
<td>56.2% (core 2)</td>
<td>33.7%</td>
<td>73%</td>
</tr>
<tr>
<td>‘PM2’ approach</td>
<td>Yes (considerable and steady usage of both cores)</td>
<td>88.7% (core 1)</td>
<td>59.2% (core 2)</td>
<td>29.5%</td>
<td>73.9%</td>
</tr>
<tr>
<td>Exhausiting Load</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raw Node.js</td>
<td>No (CPU usage only alternated between the CPU-cores with always half the total CPU usage)</td>
<td>Intense differences</td>
<td>Intense differences</td>
<td>N/A</td>
<td>54.4%</td>
</tr>
<tr>
<td>‘cluster’ module</td>
<td>Yes (considerable and steady usage of both cores)</td>
<td>97.5% (core 2)</td>
<td>97.1% (core 1)</td>
<td>0.4%</td>
<td>97.3%</td>
</tr>
<tr>
<td>‘PM2’ approach</td>
<td>Yes (considerable and steady usage of both cores)</td>
<td>91.2% (core 2)</td>
<td>89.9% (core 1)</td>
<td>1.3%</td>
<td>90.5%</td>
</tr>
</tbody>
</table>

Table 4.3.1: Utilization of the multi-core power (percent truncated to max one decimal places).
5 Discussion

In this section, we discuss the results of this experiment (and their analysis), our research objective, and how our findings could help our target group.

The objective: “Prepare the experiment environment and the test code example implementations that the experiments run with” was achieved in the preparation stage of this experiment (in the ‘2.1 Test Code Example Implementations’ subsection). It was taken care of that the prepared source code followed the recommended convention used by most of the web applications that run on Node.js, while making sure that the code reflects the real-life usage. On the other hand, the second objective: “determine the level of utilization of the multi-core power the raw Node.js environment achieves, and make this level as a threshold for the other considered attempts/approaches” was answered using the data in the ‘3 Results’ section. Particularly, at the beginning of the second and third phases’ results and their analysis (relaxed and exhausted phases). This objective was essential to the rest of the experiment because the test subject running the ‘raw Node.js’ approach is the control group for this controlled experiment (against which the experimental group was compared). Here, whether the load was relaxed, average, or exhausting, the ‘raw Node.js’ approach showed no utilization of the multi-core power (which also affected the accomplished response time and count when the average and exhausting loads were applied). As for the third: “Check how far the other attempts/approaches extend the utilization of multi-core power compared to the threshold set by objective O2” and fourth: “In light of the experiment result, determine the best approach (if any) and specify the best situation every approach excels at” objectives; the ‘4 Analysis’ section answers those objectives (with the help of the collected results and Table 4.3.1 (above on page 28)). Here, the ‘cluster’ and ‘PM2’ approaches showed matching behaviors in general; but under exhausting load, the ‘cluster’ module showed an advantage over ‘PM2’, while using slightly more CPU-power than ‘PM2’ (this might be a feature in ‘PM2’ to not cram the CPU, but this study did not consider such a feature, and there is no option to disable such a feature in PM2 if it exists). So, the ‘cluster’ module appears as the best approach in this experiment.

Since the target group for this study is concerned about achieving a better utilization of the CPU power when using Node.js, the ‘raw Node.js’ approach is not a good candidate in this case, while the other two approaches can be appealing in most of the cases. Based on what this study showed, we can give an idea about the scenarios/cases that every approach might be used in:

- Raw Node.js approach: Since the ‘raw Node.js’ does not utilize the multi-core power, it is better to avoid it in any serious, business, or commercial usage. It can be used in the development phase of a web
application, or for personal web applications on entry-level web servers that runs other applications.

- ‘cluster’ module approach: Since this approach gives the chance for a single application to utilize the full power of the CPU-cores under heavy load, a good scenario for this approach is a serious, business, or commercial web application running on a server dedicated only for this web application.

- ‘PM2’ approach: While not taking into consideration all the other features it has as a process manager for Node.js, ‘PM2’ never crammed the CPU (even under heavy load where it showed around 90% average CPU usage). This can be advantageous when running a serious, business, or commercial web application on a server that runs several other applications (where even on heavy load, there is still some CPU power margin to be used to run other applications).
6 Conclusion and Future Work

Letting the raw Node.js environment handle the utilization of the multi-core power alone, seems to be a bad approach even for an entry level commercial/serious application that uses the Node.js environment.

The two studied approaches (in addition to the raw Node.js) show a great capability to handle the utilization of the multi-core power with an advantage to the ‘cluster’ module. PM2 is widely used and popular, which might affect the decision about the approach that might be used and considered these days and in the future; but if the choice is to take advantage of the multi-core power as far as possible, the ‘cluster’ module seems to have the advantage, but it needs an extra step to do that. The initial master process will be responsible for the initial forking of new worker processes that will actually do the work. In the ‘PM2’ approach, the programmer does not have to deal with such forking of processes, ‘PM2’ can handle that automatically (even distributing the need for the communication port between the working processes if the application depends on a single communication port).

Future work may focus on a more CPU-intensive approach and maybe include some ‘NoSQL’ database operations in the experiment and check how those ‘NoSQL’ operations are handled between the processes; because ‘NoSQL’ database management systems had become a popular choice when working with Node.js. Additionally, the ‘Napa.js’ project [9], even though it is still in its early stages, seems to be advancing steadily. An experiment on ‘Napa.js’ might be an approach worth studying.
References


10, pp. 531-547, 2016.


