3-2016

A Comparative Analysis of Node.js (Server-Side JavaScript)

Nimesh Chhetri
Saint Cloud State University, chni1201@stcloudstate.edu

A Comparative Analysis of Node.js (Server-Side JavaScript)

by

Nimesh Chhetri

A Starred Paper

Submitted to the Graduate Faculty

of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science in Computer Science

St. Cloud, Minnesota

February, 2016

Starred Paper Committee:

Andrew A. Anda, Chairperson

Bryant A. Julstrom

Dennis Guster

Acknowledgements

I would like to thank my committee members, Professor Andrew A. Anda, Professor Bryant A. Julstrom, and Professor Dennis Guster for their continuous support, feedback and guidance. My special thank goes to Professor Dennis Guster and Martin Smith for providing the test environment in BCRL (Business Computing Research Laboratory) without which one of the important section of this paper would not have been possible. Also, I would like to thank the Computer Science Department Faculty at St. Cloud State University for providing me high-tech education and technical skills required to write technical paper like this. Special thanks also go to my family for supporting me and encouraging me to do the best.

Abstract

Node.js (also termed Node) is a platform built on Google Chrome V8 JavaScript runtime engine for easily building fast, scalable, and lightweight applications.V8 and Node are mostly implemented in C and C++ focusing on performance and low memory consumption. In this paper, we provide an overview of Node by comparing it to a conventional server-side scripting programming language, PHP. Initially, we focus on Node's modularity, its in-built package manager labeled Node Package Manager and Node's working architecture. The main feature of Node is its use of non-blocking event-driven I/O with an asynchronous programming model to remain lightweight and efficient in handling concurrency. These comprise the underlying features of Node which we discuss in detail. Node differs from JavaScript which we describe by emphasizing some major deficiencies in JavaScript that Node remediates. Likewise, by introducing AJAX, and its pros and cons, we show how Node surpasses AJAX in real-time application development usability. With Node.js, complex real-time applications can be built that can scale to millions of client connections. We also discuss factors supporting choosing Node and why developers should use it. We describe some of the security holes in Node with solutions to handle them. In order to clarify where Node succeeds and where it fails, we present two different benchmarks comparing Node with PHP. We conclude by highlighting some of the limitations of Node and we discuss the current developments in process to remediate Node's deficiencies.

Table of Contents

Page

Chapter Page

Chapter Page

List of Tables

Table Page

List of Figures

Figure Page

Figure Page

Chapter 1: Introduction

Node.js (Node) [1] is a cross platform runtime environment originally developed in 2009 by Ryan Dahl for developing server-side applications. It can be regarded as server-side JavaScript. It was created to address the issues platforms can have with the performance in network communication time dedicating excessive time processing web requests and responses. "Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices [1]."

Node has become popular as it makes creating high performance, real-time web applications easy. Node allows JavaScript to be used end to end, both on the server and on the client. JavaScript has originally run only in the web browser, but the considerable demand has brought it to the server-side. JavaScript has developed very much and has exceled to dominate server-side scripting. We need to analyze the security issues in Node applications because of its use with JavaScript, which has security liabilities. Node is event-based rather than thread-based. Node uses an event loop within a single thread instead of multiple threads, and is able to scale to millions of concurrent connections. In Node, a single thread can accomplish a high concurrency. Every I/O operation in Node is carried out in an asynchronous fashion, meaning that the server can continue to process incoming requests while the I/O operation is taking place [2]. Because

Node is also asynchronous, AJAX (See Chapter 3) could be mistakenly considered equivalent to Node, though they are significantly different.

Objectives of our Study

JavaScript was created shortly after the World Wide Web (WWW) came into existence. JavaScript has played an integral role in adding interaction to the user interface of web applications and websites until the recent release of HTML5 (Hyper Text Markup Language) and modern JavaScript frameworks. JavaScript is also an integral part of AJAX which was introduced in late 1990's with the advent of Web 2.0 to add real-time like interactivity in the webpages. Despite all of this progress, JavaScript has been considered as the scripting language for client-side programming (that runs only from the browser). However, this approach has changed with the development of server-side JavaScript (among which Node is considered prominent). Node has not just strengthened server-side JavaScript, but also has been competitive with other popular server-side scripting languages with respect to performance and scalability. In this paper, we will describe the advantageous features of Node. The underlying features of Node: single-threaded, event-driven I/0, and asynchronous programming are discussed with sufficient examples to give better insight into the working architecture of Node that led to Node's success. In this paper, we also distinguish Node from JavaScript, which is the backbone on top of which Node has been developed. The same is true of AJAX, which is often time confused with Node. We perform an analysis of Node's performance with real-time data by implementing two well-known applications (Fibonacci number calculation and reading large text file). For further development and enhancement, we also highlight the existing limitations and deficiencies of Node.

Simple HTTP Server with Node.js

One of the common uses of Node is to build servers. Node can be used to create different types of servers [7]. A simple HTTP (Hyper Text Transfer Protocol) web server that responds "Hello Node!" to every request it receives can be created with very few lines of code. Type the following code in the text editor, save it as *hello_server.js* and execute it by typing `node hello_server.js` from the command prompt. Then, visit this URL: `http://localhost:8000` which should print the "Hello Node!" message in the browser.

```
// A HTTP Web Server
'use strict';                                                    // 0
var http = require('http');                                      // 1
var port = 8000;                                                 // 2
http.createServer(function (req, res) {                          // 3
  console.log(req.method + ' ' + req.url);                       // 4
  console.log(req.headers);
  console.log();
  res.writeHead(200, {'Content-Type': 'text/plain'});            // 5
  res.end('Hello Node!\n');                                      // 6
}).listen(port);                                                 // 7
console.log('Listening at port ' + port);                        // 8
```

Figure 1

Simple HTTP Server in Node.js

Description of Figure 1 is as follows [6]:

0. The `'use strict'` string is a directive that enables strict mode, which is a restricted subset of the language that fixes a few important language deficiencies and provides stronger error checking and increased security. For example, strict mode makes it impossible to accidentally create global variables.
1. Imports the *http* module and assign it to the *http* object.
2. Defines the port number from which the server will be accepting connections.

3. Create the web server by calling the factory method from the HTTP module and send it a callback function. This anonymous function takes *req* and *res* (HTTP request and response) objects as parameters. Every time a client makes a request, this function will get called.
4. Displays to standard output the request information (method, requested resource, and request headers) followed by a new line.
5. Writes the response line and header fields. A HTTP status code need to be specified (200, for example, when the request was successful) and an object with all the response headers.
6. Writes the response body before the end method closes the HTTP connection.
7. Tells the web server to start accepting connections on the specified port.
8. Call the `console.log` method to print the information to standard output.

Node.js Modules

Modules are plugins, add-ons, and extensions for Node to help with the development process. The Node module exposes a public API (Application Programming Interface) that one can use after the module is imported into the current script. Node modules can be categorized as *core modules*, *third party modules*, and *local modules*. Core modules are modules that come with Node's installation and are preloaded when a Node process starts. Core modules are referenced simply by name while local modules and third party modules maps into a file path. Third party modules are modules registered in NPM and installed using *npm* command. NPM by default dumps modules installed from NPM repository into *node_modules* local directory. And local modules are self-created modules [9].

To load a module of any type in Node, `require` function should be used like this:

```
var modulex = require ('module_name');
```

For instance, modules can be loaded in Node in following ways:

```
//loading http core module by directly referring to name
var http = require('http')
// loading local module named my_module using absolute path
var myModule = require('/home/nimesh/my_modules/my_module')
//loading local module named my_module referring by relative path
var myModule = require('./my_module')
//loading third party module express after installation
var express = require('express')
```

Sharing objects among files in a Node application is possible only by using the CommonJS module system. For a module to expose an API, *module* and *module.exports* are used, `module` is a variable representing the module currently in consideration and `module.exports` is the object that the module will export to other script that requires this module. For instance, a module can be created that exports a set of functions as shown in Figure 2 [9].

```
//myModule2.js
Function printA(){
    Console.log('A')
}

Function printB(){
    Console.log('B')
}

Module.exports.printA = printA; // expose function printA
Module.exports.printB = printB; // expose function printB
```

Figure 2

Local Node.js module exposing functions

Then, the client of this module uses this module like this:

```
var myModule2 = require('./myModule2');

myModule2.printA(); // prints A
myModule2.printB(); // prints B
```

NPM: Node Package Manager

Besides writing local modules oneself, and using default modules provided by Node, modules written by other people in the Node community can be used. Also, self-created local modules can be published for others. NPM which stands for Node Package Manager is the most common way to do so [10].

NPM is a built-in tool that is included by default with every installation of Node. NPM helps in easily managing modules in Node projects by downloading packages, resolving dependencies, running tests, and installing command line utilities [8]. The main purpose of the NPM modular system is to ease the availability and installation of bunch of publicly available, reusable components via an online repository, with version and dependency management. Modules are plugins, add-ons, and extensions for Node to help with the development process. A full list of packaged modules can be found on the NPM website https://npmjs.org/, or accessed using the NPM CLI (Command Line Interpreter) tool that automatically gets installed with Node. The Node's module ecosystem is open to all, and anyone can publish their own module to be listed in the NPM repository [11].

To ensure the successful installation of NPM, issue the following command which should display the NPM version [8]: `npm -version`

To install modules via npm, `npm install` command should be used which requires the name of the module package to be installed and its version. For instance, *mysql* module package can be installed by issuing this command [10]:

```
npm install mysql@2.0.0
```

NPM installs module packages to the *node_modules* subdirectory of the project. Thousands of modules in the registry can be explored using the search and view commands. The search command is useful when the name of the package to be installed is not known so it prints the name and description of all matching published modules [8, 10].

```
npm search sql
```

The properties and `package.json` of the package can be viewed by running the *npm view* command followed by the module name [8].

```
npm view sql
```

`package.json` is a JSON (JavaScript Object Notation) file that allows to locally manage installed npm packages. It serves as documentation for what packages the project depends on, allowing to specify the version of a package that the project can use [31].

How Node.js Works?

The main distinctive features of the Node architecture are the usage of non-blocking, event-driven, asynchronous I/O calls that operate in a single thread. Conventional web servers handle concurrency by spawning new threads for each new request, which can max out the available memory. Node is lightweight, efficient, and different. It is able to support tens of thousands of concurrent connections because of its unique features. Even with limited memory and a single

thread, Node can achieve high concurrency rate without having to perform context switching between threads [12].

The Node architecture and its working mechanism can be better clarified by understanding its underlying features and comparing it with previous approaches.

Non-Blocking Event Loop

Node is non-blocking in the sense that it is able to service multiple requests, and it doesn't waste clock cycles in I/O tasks as is the case in the conventional blocking model. The conventional blocking model tends to block subsequent requests sent to a server when it is performing I/O operations such as reading content from a database. In order to be non-blocking, Node uses an event loop, a software pattern that facilitates non-blocking I/O combined with event-driven I/O, a scheme where a registered event callback function is invoked when some action happens in the program [3].

Consider this blocking PHP (Hypertext Pre-processor) code and non-blocking Node.js code in Figure 3 and Figure 4 respectively [8].

```php
<?php
//Blocking PHP Code
print('Hello');
sleep(5);
print('Node');
print('Bye');

// this script will output:
// Hello
// Node (sleeps for 5 milliseconds before printing Node)
// Bye
?>
```

Figure 3

Example of Blocking PHP Code Example [8]

```
// Non-Blocking Node.js Code
console.log('Hello');
setTimeout(function () {
console.log('Node');
}, 5000);
console.log('Bye');

// this script will output:
// Hello
// Bye
// Node
```

Figure 4

Example of Non-Blocking Node.js Code [8]

In the first example, the PHP `sleep()` function blocks the thread of execution. While the program is sleeping, it does not perform any tasks but waits for the time specified. The execution is thus blocked as long as it is specified. And no other instructions are executed until the specified time elapses, indicating it's synchronous. Node on the other hand, leverages the event loop. So, even the use of blocking, `setTimeout()` is non-blocking in the latter case. It registers an event for the future and lets the program continue to run, therefore being asynchronous.

Single-Threaded Model

Node is a process that runs in an event loop making use of a single thread to service any requests. Whereas other web servers like Apache spawn a new thread per request, which starts with a fresh state every time [8]. Node is powerful considering the way it permits non-blocking I/O to occur in a single thread which makes the overhead of Node very small, because no new threads are created. When a Node application needs to perform operations, it sends an asynchronous task to the event loop, registers a callback function, and then continues to handle other operations. The event loop keeps track of the asynchronous operation, executes the given callback and when it completes, returns its result to the application. Node is able to handle a

large number of operations (even with a single thread) by managing the thread pool and optimizing the task execution, such as client connections or computations [13].

Consider the following `getLanguages()` function in Figure 5. This function gets executed every time the user makes a request to the `getLanguages()` function and returns a collection of languages in HTML form.

```
var languages = [
    'Node.js'
    ,'PHP'
    ,'Java'
    , 'C#'
];

function getLanguages () {
    var html = '<b>' + languages.join('</b><br><b>') + '</b>';
    // reset the array
    languages = [];
    return html;
}
```

Figure 5

Example of Single-Threaded Node Model [8]

The equivalent PHP code is shown in Figure 6.

```
<?php
$languages = array(
    'Node.js'
    ,'PHP'
    ,'Java'
    ,'C#'
);

function getLanguages () {
    $html = '<b>' . join($languages, '</b><br><b>') . '</b>';
    $languages = array();
    return $html;
}
?>
```

Figure 6

Example of Multiple Thread per requests in PHP [8]

A subsequent request to the `getLanguages()` function in Node and PHP gives different results. Node handles the first request and returns the concatenated string of languages in `html` variable. The second request returns nothing because the scope variable (`html`) is not affected as Node runs in the same process. PHP code returns the concatenated string of languages in both the cases because the `$languages` variable gets repopulated each time in a new thread per request.

In accordance with the above example we can draw the diagram as shown in Figure 7 and Figure 8.

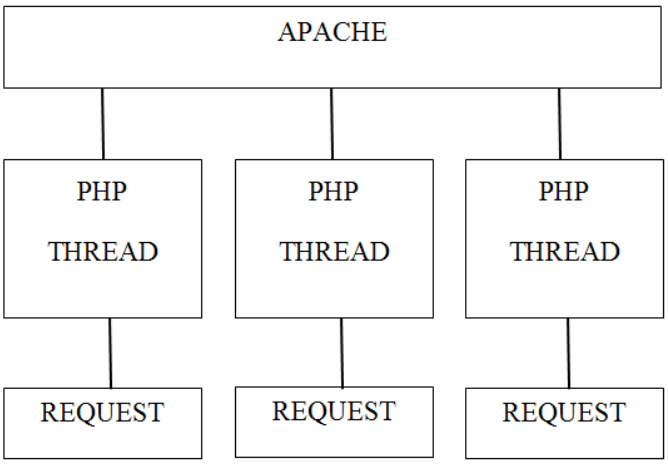

Figure 7

Threaded Model of Apache-PHP [8]

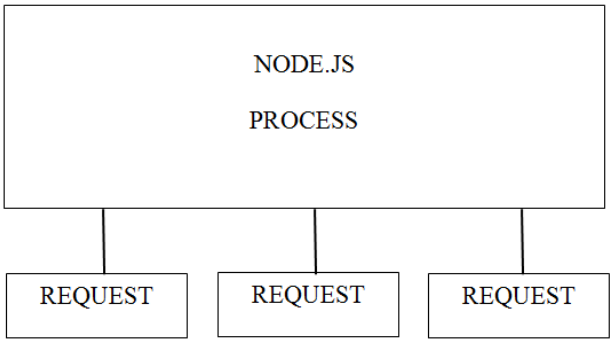

Figure 8

Single-Threaded Model of Node.js [8]

Asynchronous Programming

While the non-blocking part of Node makes it able to accept virtually all the requests made to it, its asynchronous programming makes it possible to handle the requests by effectively utilizing the limited clock cycles and memory available to its single-threaded architecture. Asynchrony is in the root of Node because almost all the APIs exposed through Node modules are asynchronous (although synchronous versions may exist). Node is able to achieve high concurrency by its asynchronous calls via a callback function to handle the tasks in its event loop. Node integrates asynchronous programming in its architecture by means of asynchronous APIs with callback function.

In order to understand the concept of an asynchronous call, consider these three examples to read a text file content as shown in Figure 9, Figure 10, and Figure 11.

```php
<?php
//PHP Version to read a text file that is synchronous

$file = fopen('info.txt', 'r');
// wait until file is open

$contents = fread($file, 100000);
// wait until contents are read

print($contents);
// do something with those contents like print it
?>
```

Figure 9

Reading Text File Using PHP Synchronously

This is obviously an inefficient version in Figure 9, which wastes lots of clock cycles, doing nothing, waiting for the computer file system to do its job.

```node
//Node version to read a text file with an incorrect asynchronous call

var fs = require('fs');

var file;
var buf = new Buffer(100000);
fs.open(
    'info.txt', 'r',
    function (handle) {
// this third argument is callback function
        file = handle;
    }
);

fs.read(   // this will generate an error.
    file, buffer, 0, 100000, null,
    function () {
        console.log(buf.toString());
        file.close(file, function () {  });
    }
);
```

Figure 10

Reading Text File Using Incorrect Node.js Asynchronously [10]

Figure 10 is a rewritten Node version of the synchronous PHP script in Figure 9. However, this code is incorrectly written and throws an error because the `fs.open` function runs asynchronously; it returns immediately, before the file has been opened. The `file` variable is

not set until the file has been opened and the handle to it has been received in the callback specified as the third parameter to the `fs.open` function.

```
//Node version to read a text file with correct asynchronous call and error handling

var fs = require('fs');

fs.open(
    'info.txt', 'r',
    function (err, handle) {
        if (err) {
            console.log("ERROR: " + err.code
                        + " (" + err.message + ")");
            return;
        }
        var buf = new Buffer(100000);
        fs.read(
            handle, buf, 0, 100000, null,
            function (err, length) {
                if (err) {
                    console.log("ERROR: " + err.code +
                                " (" + err.message + ")");
                    return;
                }
                console.log(buf.toString('utf8', 0, length));
                fs.close(handle, function () {  });
            }
        );
    }
);
```

Figure 11

Reading Text File Using Node.js Asynchronously [10]

This script version in Figure 11 takes the callback function passed as the third argument to an asynchronous function (`fs.open`). The first parameter in the callback indicates either the success or failure status of the last operation, and a second parameter indicates some sort of additional results or information from the last operation, such as a file handle [10].

Thus, a non-blocking event loop running on a single thread with asynchronous handling of tasks forms a Node architecture which can be visualized in the diagram shown in Figure 12.

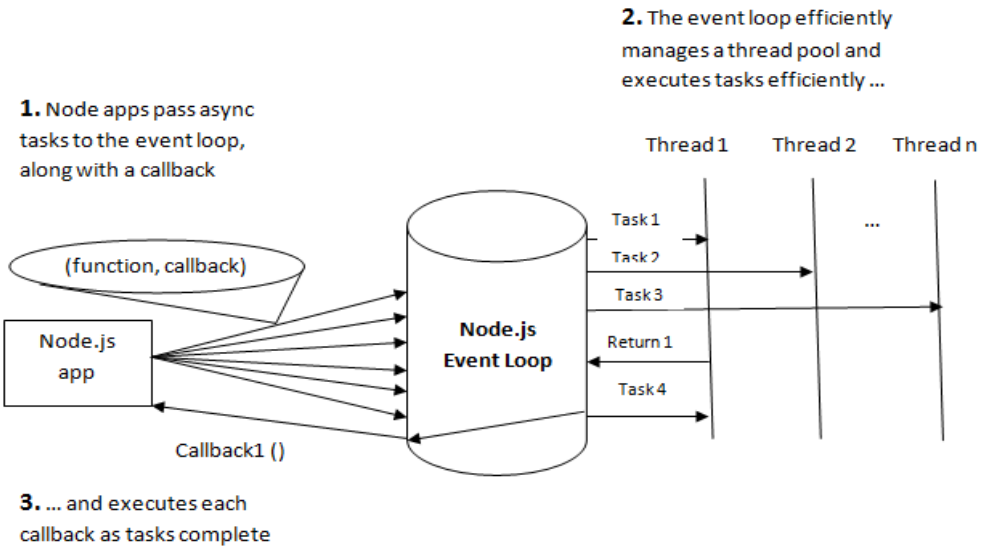

Figure 12

Node.js Architecture [12]

Chapter 2: JavaScript vs. Node.js

JavaScript is a prototype-based, object oriented, loosely-typed dynamic client side scripting language [37]. It is based on the implementation of the ECMAScript language standard. It sits and runs within the browser. It has been extensively used for adding interactivity to websites. Therefore, it requires help from another programming language if it has to perform any interactions with the server.

Although Node is based on JavaScript, and uses the construct of JavaScript for almost all of its functionality, Node offers an entirely different environment than JavaScript. Node can be regarded as superset of JavaScript. It has bundled additional functionalities and features on top of JavaScript.

Node is a wrapper on top of the High Performance Google Chrome V8 JavaScript runtime engine. As a result, most of the Node syntax is very similar to front-end JavaScript, including objects, functions and methods [9]. However, Node does have some features that are not available in a conventional browser-based JavaScript. Most of these features address the deficiencies that existed in JavaScript. Node took the basic JavaScript language and added different APIs on top of it to add more power for enhancing the network applications.

Module System

One of the deficiencies in browser JavaScript is lack of modularity. The only way to link together different scripts is by using a different language such as HTML. Even if including a module is possible in an inefficient way, dependency management is lacking [9].

It would not be any different to categorize JavaScript as an environment of globals, because all the APIs that are normally used are globally exposed in a JavaScript code. When any third-party modules are included, a global variable is required to expose and to make it accessible.

JavaScript doesn't describe an API for module dependency and isolation in its specification. As a result, including multiple modules is only possible by exposing a global variable. For example, the jQuery module can be included in a HTML document by including this line at the head tag `<script src="http://code.jquery.com/jquery-1.6.0.js">`. Then, refer to this module through the global jQuery object. This process pollutes the global namespace and can result in potential naming collisions [8].

Instead of defining a number of globals, Node has introduced a modular system (See Chapter 1). One can define their own module or can use the core modules or third party modules. Node ships with a lot of core modules such as `http, net,` and `fs`. These modules are the fundamental toolkits for building modern applications. Modularity in Node is made possible because of these three globals: `require`, `module`, and `exports` [8].

Global Object [8]

There is a global object in the browser JavaScript named `window`. Variables can be defined in the window object to make it globally available to all parts of the application code. Node implements globals with a clear separation. In Node, these two global objects are used for this purpose:

- **global**: Similar to a window object in JavaScript, any property can be attached to a global to make variables accessible from anywhere in the application code.

- **process**: There is a process object in Node that are assigned for everything that pertains to the global context of execution.

In JavaScript, there is one `window` object. While in Node, there is only one process at any given time. For instance, in JavaScript, the window name is `window.name`, and in Node, the name of the process is `process.title`.

Buffer

Another deficiency in JavaScript is its support for handling binary data [8]. Manipulation of binary data is poorly supported in JavaScript language, even though it is often necessary. Node's Buffer class resolved this deficiency by providing APIs for easy manipulation of binary data [14]. Buffer is a Node's addition to four primitive data types (boolean, string, number, and RegExp) and all-encompassing objects (array and functions are also objects) in a front-end JavaScript. It uses extremely efficient data storage [9].

Buffer, a global object that represents a fixed memory allocation, behaves like an array of octets, effectively letting binary data to be represented in JavaScript. Most of the Node APIs that perform data I/O take and export data as buffers [8].

Chapter 3: AJAX vs. Node.js

Node is often confused with another technology, AJAX, but both of them are completely different and serve different purposes. The only similarity between AJAX and Node is that they both run on JavaScript. While Node is mostly used for server-side operations for developing a complete server-side application, AJAX is used for client-side operations for dynamically updating the content of the page without refreshing it. This can be more clarified by discussing what AJAX really is, why it is used, what its limitations are and how Node proves superior fulfilling those limitations that AJAX has.

AJAX

AJAX [18], an abbreviation for "Asynchronous JavaScript and XML", is a set of techniques for creating highly interactive websites and web applications [18]. AJAX is broadly used to refer to all the methods of communicating with a server from a client using JavaScript. Although, AJAX is mostly used for asynchronous communication and mostly involves XML for data transfer, it can be synchronous and can make use of other data formats like JSON [16].

AJAX has transformed the way users interact on the web. Applications no longer need to refresh the whole page in response to each user input. Using AJAX, application can call a specific procedure on the server and update only the specific section of the webpage. Before AJAX, interactivity on web pages was rather clumsy and expensive. Because, for any user interaction to happen, an updated version of the page was required to be generated on the server, sent back to the browser and rendered. Even if the required update was minute, the result was

always a whole new page refresh. This model wasted both bandwidth and resources. AJAX simplified this approach by modifying the process at a granular level [17].

In AJAX, JavaScript code uses a special object built into the browser: an `XMLHttpRequest` object to open a connection to the server and download data from the server [18]. AJAX is the mechanism for sending the data to, and retrieving the data from, the server with AJAX. The overall steps involved in making AJAX requests and getting the responses, are listed as follows [16]:

1. An `XMLHttpRequest` object is created using an `XMLHttpRequest()` constructor.
2. That object is used to make HTTP requests. To do so, the object is initialized with the `open()` method, which takes three arguments: Request Type (String: POST, GET, or HEAD), URL (String) and Asynchronous (Boolean).
3. The `xmlhttp` object's `readyState` property holds the current state of the response. There are five possible states (0-4): 0 refers to uninitialized, 1 refers to loading, 2 refers to loaded, 3 refers to interactive and 4 refers to complete.
4. Send the response back to the client.

An introduction of AJAX and its flow, compared to the classic web model is diagrammed in Figure 13.

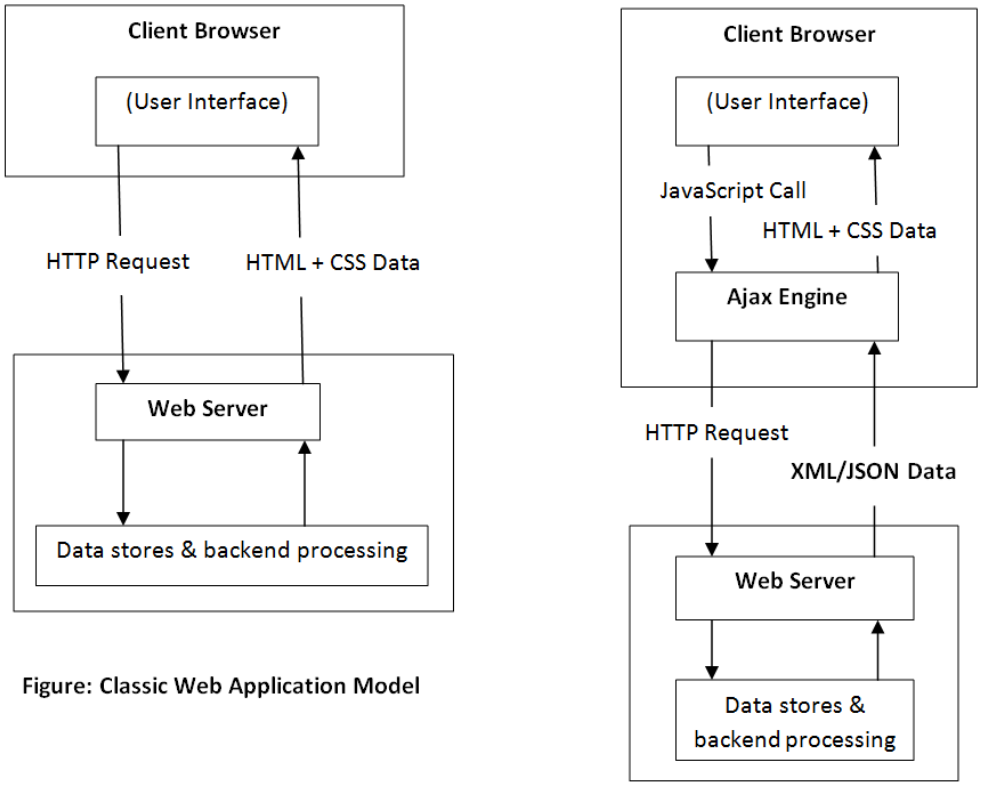

Figure 13

Classic vs. AJAX Web Application Model [32]

In Figure 14, we illustrate all the steps required for an AJAX call. In this example, using AJAX, the client updates the text with the weather forecast of the city within `divs` with the text read from a text file initiated by a windows `onload` event [18].

```
//This portion of script make a call to loadWeatherUpdate() //every 10 seconds
window.onload = function(){
    setInterval(function(){
        loadWeatherUpdate();
    }, 10000);
};

//This function makes call to server page processJson.php via //AJAX
function loadWeatherUpdate(){
var XMLHttpRequestObject = false;
    if (window.XMLHttpRequest) {
        //Step-1: Create XMLHttpRequest Object
        XMLHttpRequestObject = new XMLHttpRequest();
    }
    if(XMLHttpRequestObject) {
    var dataSource = "http://localhost/phpapp/processJson.php";
        //Step-2: Object initialized with the open method
        XMLHttpRequestObject.open("GET", dataSource);
        XMLHttpRequestObject.onreadystatechange = function()
        {
            //Step-3: Check the current status of the //response to display the result
            if (XMLHttpRequestObject.readyState == 4 &&
                XMLHttpRequestObject.status == 200) {
                var jsonObj = JSON.parse(XMLHttpRequestObject.responseText);
                document.getElementById("city").innerHTML = jsonObj.city;
                document.getElementById("max_temp").innerHTML = jsonObj.max_temp;
                document.getElementById("min_temp").innerHTML = jsonObj.min_temp;
            }
        }
    //Data sent already using GET in header so null is passed //in send
    XMLHttpRequestObject.send(null);
    }
}
```

```
<?php
//processJson.php reads content from data.json file and return
//back the data in JSON format
echo file_get_contents("data.json");
?>
```

Figure 14

Example of AJAX Call

AJAX enables applications to update pages, only in response to user actions on the page. It does not solve the problem of updates coming from the server. It does not offer a way to push information from the server to the browser [17]. With AJAX, the clients always have to query the server continuously for any new information or data that is available for the application. The server cannot push information to the client without being asked first. The consequence is an

application user will not see the data unless they ask for it from the server. In turn, the data which the applications gets will be outdated i.e. the data will not be real-time. This is where AJAX appears inferior to what Node has to offer. Although, there are some techniques using AJAX such as AJAX polling and AJAX long polling, to make application simulate real-time, there are caveats to these approaches.

AJAX Polling [32]

In AJAX polling, the request is sent from the client to the server at a regular interval of time to check for any new updates that are available. If there are any new updates available from the server, they are sent back. The drawback to this approach is that there will still be delays and will not replicate a real-time communication. Also, there will be lots of requests and responses to and from the client and server even if there are no updates. This is shown in Figure 15.

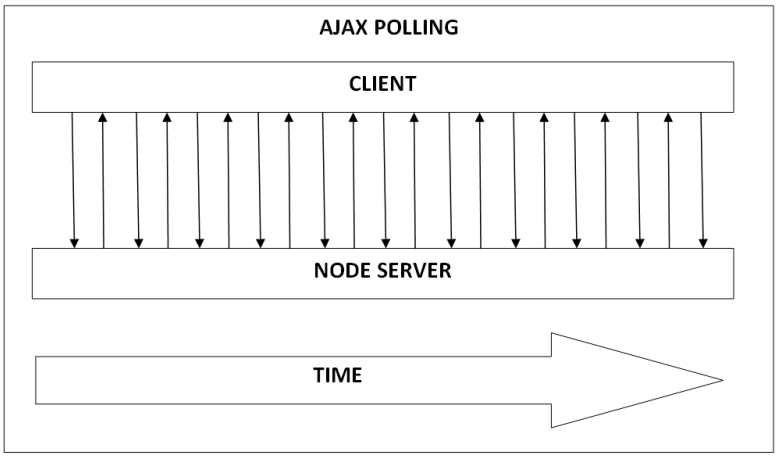

Figure 15

AJAX Polling [32]

AJAX Long Polling [32]

Another approach is AJAX long polling, which is just a slight variation to AJAX polling. Like regular polling, when the server receive requests, it immediately returns the new data if it's

available. However, if there are no new data to return, server keeps the connection open, and returns the data once it becomes available. Once the client receives data, the client immediately sends another request to the server again. This is shown in Figure 16.

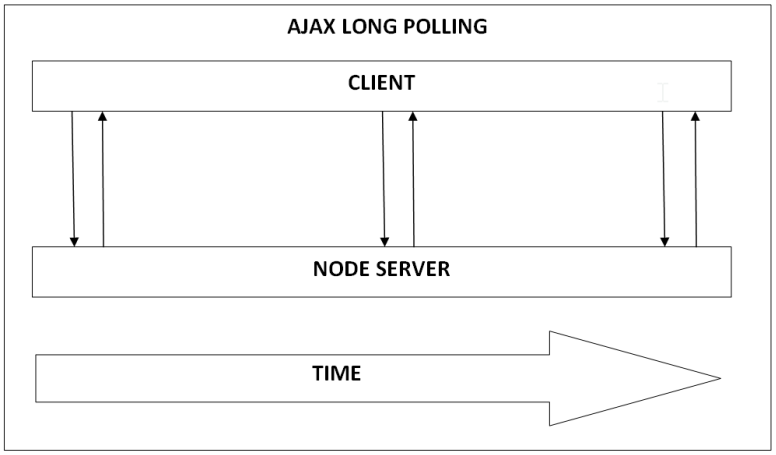

Figure 16

AJAX Long Polling [32]

Real Time Application Development with Socket.io [32]

There are more techniques besides AJAX polling and AJAX long polling, to establish a real time communication between the client and the server or at least resemble it like SSE (Server Sent Events) and Web Sockets. But nothing gets the job done for bi-directional communication between client and server in real-time as *Socket.io* does. SSEs allow servers to push data to their clients, but the client can't send data back to the server. Web Sockets allows simultaneous duplex communication in both directions, between client and server, but it's an entirely new protocol.

Socket.io is a library for real-time web applications. It is a module built in Node and it can be installed in Node application as:

```
npm install socket.io
```

Socket.io is event-driven and has both server-side and client-side components exhibiting similar APIs. Basically, Socket.io is made up of two parts:

- A server that mounts on or attaches to Node's HTTP server.
- A client-side library that allows interaction with the server.

Both the client and server parts essentially do the same thing: allow the sending (or emitting) of events and provide a way to define event handlers. In order to setup the Socket.io server components, the following steps should be followed:

- Create a Socket.io server and attach it to existing HTTP server.
- Define what the server will do on connection.
- Within that connection handler:
    - Define what the server will handle other custom events.
    - Send messages.

Likewise, in order to setup Socket.io on the client, following steps could be followed:

- Bring in the client side library.
- Create a socket object (an interface to the connection).
- Use that socket object to:
    - Send messages.
    - Define callbacks that get triggered on specific events.

To understand the real usage of Socket.io, consider the following script in Figure 17, which implements the same weather update scenario, already discussed for AJAX.

The server-side Node.js script is shown in Figure 17. The functionalities of important statements are described in a comment section.

```
var app = require('http').createServer(handler),
        io = require('socket.io').listen(app),
        fs = require('fs'), //file module
jf = require('jsonfile'); //jsonfile module
app.listen(1234);

function handler(req, res) {
    fs.readFile(dirname + '/index-socket.html',
        function (err, data) {
            if (err) {
                res.writeHead(500);
                return res.end('Error loading index.html');
            }
            res.writeHead(200);
            res.end(data);
        });
}

io.sockets.on('connection', function (socket) {
    fs.watch("data.json", function(event, fileName) {
    //watching data.json file for any changes
        jf.readFile('data.json', function(err, data) {
        //if change detected read the data.json
        var data = data; //store in a var
        socket.emit('weatherUpd', data);
        //emit to current client
        socket.broadcast.emit("weatherUpd", data);
        //emit to all clients
        });
    });
});
```

Figure 17

Example of Socket.io Server Side Script

In the example, the *data.json* file used for data interchange appears as:

```
{"city": "Minneapolis", "max_temp": "42", "min_temp": "10"}
```

The client-side script is shown in Figure 18.

```
var socket = io.connect("http://localhost:1234");
    socket.on("weatherUpd", function (data) {
    // jsonObj variable now contains the data structure and can
    // be accessed as jsonObj.name and jsonObj.country.
    document.getElementById("city").innerHTML = data.city;
    document.getElementById("max_temp").innerHTML =  data.max_temp;
    document.getElementById("min_temp").innerHTML = data.min_temp;
});
```

Figure 18

Example of Socket.io Client Side

The Socket.io model diagram can be depicted as shown in Figure 19.

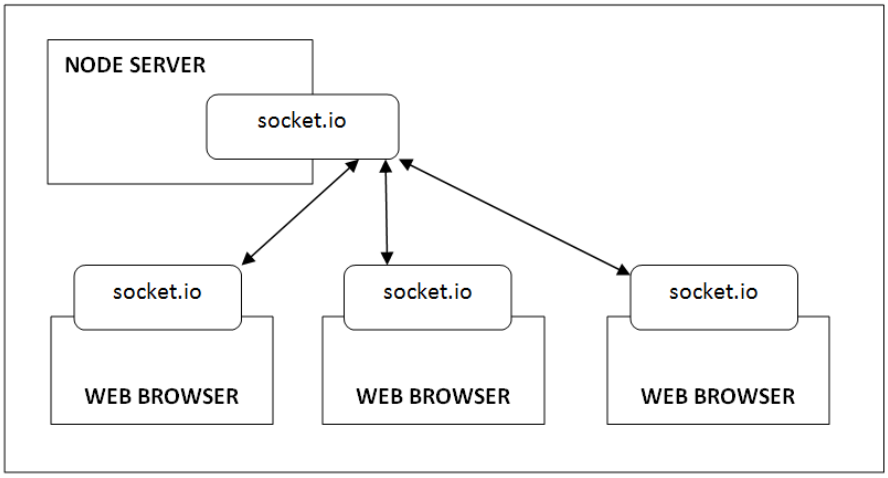

Figure 19

Block Diagram of Socket.io [32]

Chapter 4: Why Node.js?

Node has been popular among developers and with its success has attracted many high-tech companies who have adopted Node replacing their existing technologies. There are many reasons for Node's popularity and why one should use Node for their application development.

High Performance Web-Servers [5]

With the emergence of Web 2.0 and the Internet connectivity in different devices: cell phones, tablets, desktop, and laptops; the scope of application has highly scaled up. Aligning with the demands to support higher numbers of users and deliver a real-time experience in the application has become the major challenge. While installing new hardware adds power to continually increasing demands for speed and faster connectivity in the applications, which is not the optimal solution as it more expensive. Node attempts to solve this problem by introducing the architecture termed event-driven programming for web servers. Node is much more efficient with memory than conventional servers and can keep providing a very fast response time despite many concurrent users. This is because Node runs in a single thread, and whenever Node has to do something slow, like wait for a confirmation, it simply moves on to service another request. Conventional approaches can accomplish this by multi-threading, which requires more memory. At the same time conventional approaches are complex and involve context switching. Node is preferable in this aspect, providing a less expensive, scalable, and high performance application environment.

Popularity of JavaScript [19]

Since early in the evolution of WWW, JavaScript has been there in the browser. Even available when AJAX emerged, JavaScript was vital. This has led to the popularity of JavaScript among developers, despite some criticism. No matter which server-side scripting language is used, JavaScript has been the choice for client-side scripting. Familiarity with JavaScript and adherence of Node to JavaScript, with capabilities to code in the server-side and numerous other features has developers to adopt Node. By leveraging the best features of JavaScript as a language and nurturing a vibrant community, Node has become a popular platform and framework, with continued adoption growth.

One Language Multiple Functionality [19]

Node allows running JavaScript code on the server-side as well as the client-side. Node has elevated JavaScript to a new height of application development. Any system developed in Node will run from just about anywhere-on a local or on a client's platform, or from a high-end Node server hosted elsewhere. In addition, there are thousands of Node modules available for free. Applications can be developed locally using Node's built-in web server. So, unlike other programming languages, where a separate web server is required for it to get hosted or even tested, Node has everything it requires-a web server, client-side scripting, and server-side scripting.

Simple Development Environment [19]

Conventionally, setting up the development environment for new projects has been cumbersome for developers. It requires time and effort, the first step involves getting the development environment right, making sure all the software packages are installed with

required versions, and then putting the code in the repository to test. Many times a conventional environment requires developer to look back and check if something was missed. Node simplifies this process, increasing the developer's productivity. In the Node environment, developers simply download Node, pull their code from the repository, and go from there. Node installers are available for all the major OSes: Mac, Linux, Windows and SunOS. The source code can also be downloaded and built from the ground up. So, setting up the Node's development environment is comparatively much simpler (See Appendix).

Good Reputation [19]

Node has earned a good reputation in the tech industry. Many big players in the technology industry are using Node. Node plays a critical role in the technology stack of many high-profile companies who depend on its unique benefits. Node gives Microsoft Azure users the first end-to-end JavaScript experience for the development of a whole new class of real-time applications. Node's I/O event model freed eBay from worrying about locking and concurrency issues common with multi-threaded asynchronous I/O. On the server-side, the entire mobile software stack of LinkedIn is completely built in Node [30].

PayPal, after making a move from Java to Node for their existing projects, saw significant improvement over Java. Using Node, the re-written app was delivered in half the time with fewer developers, using fewer lines of code but with ability to handle twice as many requests each with one-third less latency. Hence, they saw their development and product performance increase dramatically after the switch [19].

Chapter 5: Node.js Security

The surge of demand for JavaScript in the programming field has expanded in scope from client-side to server-side. As a result, SSJS (Server-Side JavaScript) features are available almost everywhere, be it in database servers (like Mongo DB), file servers, and web servers (like Node). This move of JavaScript to SSJS has brought many benefits, but also bundled together some drawbacks in terms of security. Client-side script injection, that has existed for a long time can be exploited to execute on the server. Server-side script injections are equally easy to accidentally introduce into server-side application code as they are for client-side code. Comparatively, the effects of SSJS injection are far more severe. One of such vulnerability is *cross site scripting*. Since, Node is based on JavaScript it is also vulnerable to cross site scripting.

Cross Site Scripting (XSS)

XSS is an attack that allows the attacker to inject malicious script in the web application. XSS vulnerabilities are caused by a failure in the web application to properly validate user input. By subverting client-side scripting languages, an attacker can take full control over the victim's browser [20].

XSS vulnerabilities are not only extremely dangerous; they are extremely widespread as well. The Open Web Application Security Project (OWASP) currently ranks XSS as the second most dangerous threat to web applications (behind SQL injection), and the 2011 CWE/SANS Top 25 Most Dangerous Software Errors ranks XSS as the #4 threat (down from #1 in the 2010 list) [21].

Consider this block of client-side JavaScript code in Figure 20, intended to process weather forecast requests. The code uses JSON as the message format and `XMLHttpRequest` as the request object.

```
<script>
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
   if ((xhr.readyState == 4) && (xhr.status == 200)) {
       var weatherInfo = eval('(' + xhr.responseText + ')');
       alert('The current maximum temperature of ' + weatherInfo.city +
       ' is ' + weatherInfo.maxtemp);
   }
}

function makeRequest(cityCode) {
    //forecast_service is URL for the service
    xhr.open("POST","forecast_service",true);
    xhr.send("{\"city\" : \"" + cityCode + "\"}");
}
</script>
```

Figure 20

Example of XSS in Client-Side JavaScript

The block of code in Figure 20 makes a call to the `eval` function that potentially introduces a serious vulnerability. This function takes a string argument which can represent an expression, statement, or a series of statements, and it is executed as any other JavaScript source code [22]. An attacker can modify the JSON response like this statement to leak the cookie information of the client.

```
{"city":  "MSP", "maxtemp": "<script>document.cookie</script>"};
```

Again consider a very similar block of server-side JavaScript code in Figure 21 designed to parse JSON requests, executing on the server to implement a Node web server.

```
var http = require('http');
http.createServer(function (request, response) {
    if (request.method === 'POST') {
        var data = '';
        request.addListener('data', function(chunk) { data += chunk; });
        request.addListener('end', function() {
            var weatherInfo = eval("(" + data + ")");
            getForecast(weatherInfo.city);
        });
    }
});
```

Figure 21

Example of XSS in Server-Side JavaScript

The code snippet in Figure 21 exposes a similar kind of vulnerability caused by the `eval` function as in the client-side example. However, in this case, the effects of the vulnerability are much more severe than a leakage of a victim's cookies. For example, a normal JSON message to the forecast service looks like this:

```
{"city": "MSP", "maxtemp": "65"}
```

However, an attacker can modify JSON message with malicious content like this:

```
{"city": "MSP", "maxtemp": "process.exit()"}
```

The server would execute above injected command to potentially kill the process and program as a whole. One should ensure that all user inputs are parsed and filtered properly to ensure such vulnerability are not exposed. The best heuristic to follow regarding user input filtering is to deny all but a pre-selected element set of benign characters in the web input stream [20].

Another solution is running JavaScript in strict mode that reduces most of the harmful practices in JavaScript caused by the culprit `eval` and makes the compiler throw errors in its bad usage. Simply place `user strict` directive at the top of JavaScript. The most important

vulnerability addressed by strict mode is that the `eval()` function cannot be renamed and hence prevents an attacker from introducing new variables to the global scope. As shown in Figure 22, in strict mode, use of `eval` throws errors.

```
"use strict";

//All generate errors
Obj.eval = '…';
Obj.foo = eval;
var eval = '';
for (var eval in obj) {}
function eval(){}
function test (eval){}
function (eval) {}
new Function ("eval")
```

Figure 22

Use of Strict Mode in Node.js

Denial of Service (DoS)

DoS is an attack which makes information or data unavailable to its intended hosts [23]. It is one of the simplest forms of network attack. Instead of trying to steal or modify information, the aim of this attack is to prevent access to the service or resource. This is usually achieved by flooding the server with a large amount of requests, tying up the server's resources and preventing legitimate requests from being fulfilled [5].

This means of DoS attack may be not be quite effective in the Node platform as it would be in conventional web servers, but CPU intensive tasks can succumb to DoS victim [22]. This is because the Node architecture uses very few system resources and bombarding it with tons of requests comprising the DoS attack becomes ineffective in Node but it might exhibit bandwidth problems.

There are several other ways to perform DoS attacks in Node unlike conventional request flooding which is not effective (as discussed above). One of the ways to attack is by using flaws

in the system such as a lack of error handling or other methods to make the server unresponsive. For instance, a DoS attack can be simulated by issuing this following `while` command [21]:

```
while(1)
```

The infinite `while` loop will consume all the processor time in the targeted server slowing it down. This will make the server unresponsive and thus will make it unable to handle any requests.

Another way to perform a DoS attack in Node would be to simply exit or kill the running process [21]:

```
process.exit()
process.kill(process.pid)
```

The appropriate approach to handle a DoS attack is by implementing a good error handling code. It's the developer's responsibility to handle the situation where the application will push the server in a stalemate situation. One of the possible solutions is to use the `try...catch` statement. Try-catch statements tries to execute good code inside `try` block and upon error will be directed to the `catch` statement to handle the error. An example is shown in Figure 23.

```
// Try to do something
try{
    //Defining some variable
    var a = 5;
        b = 7;
    //If the variable do not match a condition
    if((b-a)!=2){
        //Throw an error
        throw "Incorrect Result";
    }
}
//Catch the error
catch(err){
    console.log(err); //print the error to console
}
```

Figure 23

Example of Try Catch Block

Regular Expression DoS (ReDoS)

Although the responsiveness of the server and availability of resources are not quite considered as security issue, their impacts can be severe. Node is based on a single thread event loop architecture which makes it a suitable candidate with respect to loss of resource availability. There are many ways to block the event loop. One way an attacker can do that is with ReDoS [24].

"The Regular expression Denial of Service (ReDoS) is a Denial of Service attack, that exploits the fact that most Regular Expression implementations may reach extreme situations that cause them to work very slowly (exponentially related to input size) [25]."

If an application uses Regular Expressions containing vulnerable Regex, it is open to attackers who can prepare a well-crafted input to make the system unresponsive. Alternatively, if a Regex itself is affected by a user input, the attacker can inject a vulnerable Regex, and harm the system [25].

In Figure 24, a vulnerable RegEx example has been considered that attempts to validate an email address on the server.

```
validateEmail= function(string) {
var emailExpression = /^([a-zA-Z0-9_\.\-])+\@(([a-zA-Z0-9\-])+\.)+([a-zA-Z0-9]{2,4})+$/;
    return emailExpression.test(string);
}
```

Figure 24

Example of Vulnerable RegEx [24]

The impact of vulnerable Regex in Figure 24 can be checked with this test script shown in Figure 25 to analyze the responsiveness of the server.

```
start = process.hrtime();
console.log(validateEmail("chni1201@stcloudstate.edu"));

/* Matches on good input & takes less time less than a second */
console.log("Good Input Time: ");
console.log(process.hrtime(start));
start = process.hrtime();
console.log(validateEmail(
"ccccccccccccccccccccccccc@hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh.22222222222222222222222222222222222222211111111111111111{"
));

/* Bad input data takes just over 7 seconds */
console.log("Bad Input Time: ");
console.log(process.hrtime(start));
```

Figure 25

Analysis of Vulnerable RegEx [24]

Here is the output from the script in Figure 25:

```
true
Good Input Time:
[ 0, 13792252 ]
false
Bad Input Time:
[ 7, 935494039 ]
```

A check for bad regular expressions can be done in an automated way by using a module called safe-regex. Additionally, OWASP has a list of regular expressions for common validations that can be utilized for validations in our applications [26].

First, install the safe-regex module by using the `npm` command.

```
npm install safe-regex
```

Then check for good regular expression as shown in Figure 26.

```
var safe = require('safe-regex');
var emailExpression = /^([a-zA-Z0-9_\.\-])+\@(([a-zA-Z0-9\-])+\.)+([a-zA-Z0-9]{2,4})+$/;
if(safe(emailExpression)){
    console.log("Safe Regular Expression");
    // make a call to validateEmail function
}else{
    console.log("UNSAFE Regular Expression");
}
```

Figure 26

Check for safe regular expression [24]

File System Access [21]

Node provides the File I/O functionality in the built-in module that comes with the Node installation. Attackers can use the APIs exposed in this module to read the contents of the files from the local system. A file module can be imported by issuing this command:

```
var fs = require('fs');
```

All the methods in the `fs` module have two forms: asynchronous and synchronous. The asynchronous method always takes a completion callback as its last argument. Depending on the method, the arguments passed to the completion callback differs, but the first argument is always for an exception. In case the operation was completed successfully, the first argument will be either *null* or *undefined*. When using the synchronous form, any exceptions are immediately thrown [27].

Following are the details of some of the File I/O methods in the `fs` module that the attacker can utilize:

1) fs.readdirSync(path) [27]

   This is the synchronous `readdir` function that reads the contents of a directory specified in the path as argument. It returns an array of filenames excluding '.' and '..'.

2) fs.readFileSync(filename[, options]) [27]

   This is the synchronous version of `fs.readFile`. It returns the contents of the filename. If the encoding option is specified then this function returns a string otherwise a buffer.

It can take two arguments:

- filename String
- options Object
    - encoding String | Null default = null
    - flag String default = 'r'

3) fs.writeFileSync(filename, data[, options]) [27]

   This is the synchronous version of `fs.writeFile`. It returns undefined. It takes the following arguments:

   - filename String
   - data String | Buffer
   - options Object
       - encoding String | Null default = 'utf8'
       - mode Number default = 438
       - flag String default = 'w'

There is a way to add the file system access functionality in the currently running script even if it originally doesn't exist by including the appropriate `require` command such as the `fs` module. An attacker can utilize any of the methods described above to list the contents of the directory or even the file and even write to that. This can be accomplished by issuing the series of commands and methods chained together like this:

```
response.end(require('fs').readdirSync('.').toString())
response.end(require('fs').readdirSync('..').toString())
```

The preceding scripts will list the contents of the current directory and parent directory respectively. A complete directory structure of the entire file system can be built this way. The actual contents of a file can be listed by issuing the following command:

```
response.end(require('fs').readFileSync(filename))
```

More danger lies in the possibility of writing to the file as compared to just reading the contents of the file. This can be done and is demonstrated below:

```
var fs = require('fs');
var currentFile = process.argv[1];
fs.writeFileSync(currentFile, 'hacked' + fs.readFileSync(currentFile));
```

This attack shows how easy it is to write to a file by prepending the string "hacked" to the start of the currently executing file. The boundary is thus wide open for attackers to invoke more malicious attack than this.

Execution of Binary Files [21]

It has been noted that it is possible to create arbitrary files on the target server, including binary executable files:

```
require('fs').writeFileSync(filename,data,'base64');
```

In the preceding command, `filename` is the name of the resulting file (i.e. `foo.exe`) and `data` is the base-64 encoded contents that will be written to the new file. The attacker now only needs a way to execute this binary on the server.

Now that the attacker has written their attack binary to the server, they need to execute it by issuing this command:

```
require('child_process').spawn(filename);
```

Chapter 6: Benchmarking Node.js

In several section of this paper, we discussed the pros and cons of single-threaded Node by comparing it with a conventional threaded programming model. This has been limited to theory so far. So, in this section we verify those statements with testing. Tests were performed using programs in Node and PHP. Those programs were executed in two different servers which were configured prior to tests. The detailed benchmarking methodology and the results are discussed separately below.

Benchmarking Methodology

Benchmarking tests were performed with the objective to test Node and Apache Servers running PHP with increasing levels of concurrency and requests. These tests were intended to measure how well each framework handled varying server loads. Therefore, the purpose of the test was to compare Node with one of its competitor Apache-PHP combinations. Throughout this paper, we asserted that Node is able to attain higher levels of concurrency and is efficient at handling I/O while failing to repeat that trait when heavy computation is involved. Our experiments described in this section validate those statements with test results and analysis. For performing our tests, two test environments were set up inside a virtual machine, one running on Windows 7 and the other on Ubuntu. The detailed server configuration for our testing environment is shown in Table 1.

Table 1

Test Environment Server Configuration

Windows 7 Environment	
Hardware Configuration	**Software Configuration**
OS: Windows 7 Enterprise Service Pack 1, 64 Bit **CPU:** Intel (R) Xeon (R) CPU E5-2680 v2 @ 2.80 GHz 2.79 GHz **RAM:** 4 GB	**Node:** 0.12.7 **PHP:** 5.5.30 **Apache:** 2.4.17 **Apache Bench:** 2.3
Ubuntu Environment	
Hardware Configuration	**Software Configuration**
OS: Ubuntu 14.04.2 LTS **CPU:** Intel (R) Xeon (R) CPU E5-2680 v2 @ 2.80 GHz 2.80 GHz **RAM:** 2 GB **Linux Kernel:** Linux 3.13.0-65-generic x86_64	**Node:** 0.12.7 **PHP:** 5.5.9 **Apache:** 2.4.7 **Apache Bench:** 2.3

For gathering the hardware configuration info in Ubuntu, the following commands were used:

```
$ cat /proc/meminfo
$ cat /proc/cpuinfo
$ lsb_release –a
$ uname -mrs
```

Our experiments were conducted by running the Node and Apache web server locally, hosting 2 sample programs. For installing Apache-PHP in the Windows environment, the third-party tool named XAMP was used. For emulating the concurrent connections and multiple requests, a tool named Apache Bench was used (a command line utility that comes with default installation of Apache Server in XAMP). Apache Bench offers an array of configurations. For these experiments, the total number of server requests and number of concurrent requests were varied. Below is a sample command for benchmarking a server running locally on port 8080 with 10,000 total requests and 1,000 concurrent requests [28].

```
ab –n 10000 –c 1000 http://localhost:8080/
```

For installing Apache-PHP in Ubuntu following commands were used:

```
sudo apt-get install apache2
sudo apt-get install php5
sudo apt-get install libapache2-mod-php5
sudo /etc/init.d/apache2 restart
```

For installing Apache Bench in Ubuntu following command was issued:

```
sudo apt-get install apache2-utils
```

Test 1: Fibonacci number calculation [29]

The calculation of Fibonacci numbers is a common programming problem in teaching computer science and mathematics. Formally, the Fibonacci number F_n is the *nth* term of the series formed by the following recurrence relation:

$$Fn = Fn-1 + Fn-2$$

$$\text{for } n = 3, 4, \ldots, \text{with } F1 = F2 = 1$$

There are many iterative recursive solutions for calculating Fibonacci numbers. Among them the popular one is recursive version, the pseudo code of which is shown in Figure 27.

```
int Fib_Rec(int n){
    if((n==1)||(n==2))
        return 1;
    else
        return Fib_Rec(n-1) + Fib_Rec(n-2);
}
```

Figure 27

Recursive function to calculate Fibonacci number

The recursive Fibonacci function's main disadvantage, however, is its exponential time complexity (because of growing number of intermediate values) compared to the linear time complexity of the iterative versions. The Fibonacci calculation was chosen because it involves heavy computation and the objective was to test how Node performs doing heavy computation as compared to Apache-PHP. Both the Node program and the PHP program were coded to calculate the 20th Fibonacci number.

Test 2: Reading large text file with concurrent requests

File reading is an I/O operation. We discussed that Node is proficient at handling I/O requests. In order to test this, we coded in both Node and PHP to read a large text file having two column data delimited by a tab and with 500,000 lines. An excerpt of the text file is shown below:

```
//Dfile.dat
23279854 9
23215908 8
24666448 11
```

For monitoring, the CPU and the memory usage in the Windows environment during execution of 2 sample programs, Windows Task Manager was used. Likewise for Ubuntu, the command line tools named htop and mpstat were used. htop and mpstat were installed in Ubuntu by issuing the following commands:

```
sudo apt-get install htop

sudo apt-get install sysstat
```

Benchmarking Results and Findings

Node was quite fast in terms of execution time for Test 1 in the Windows environment as verified by the comparison between the response time readings in Table 3 and Table 2. Even with a significant increase in concurrent requests and number of requests, the response time was not as high for Node as it was for Apache-PHP. However, Node accomplished this task with a peak CPU utilization time of 100% (in a single thread) as compared to max 76% for Apache-

PHP. This has been shown in Figure 29 and Figures 28 for Apache-PHP and Node in Windows respectively.

Table 2

Test 1: Windows 7 Environment (Apache-PHP)

Execution Time in Seconds				
Concurrency\Requests	**500**	**1,000**	**5,000**	**10,000**
10	1.805	3.463	17.753	36.582
100	2.075	4.303	21.295	46.351
200	1.919	3.760	24.632	47.252

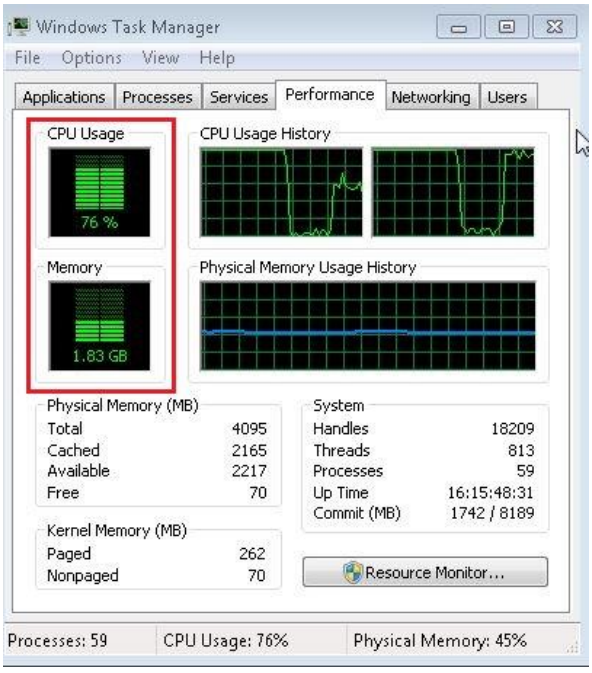

Figure 28

Test 1: CPU Utilization Windows7 Environment (Apache-PHP)

Table 3

Test 1: Windows 7 Environment (Node.js)

Execution Time in Seconds				
Concurrency\Requests	**500**	**1,000**	**5,000**	**10,000**
10	0.577	0.796	4.524	7.472
100	0.532	0.739	3.432	7.363
200	0.421	0.827	3.276	6.536

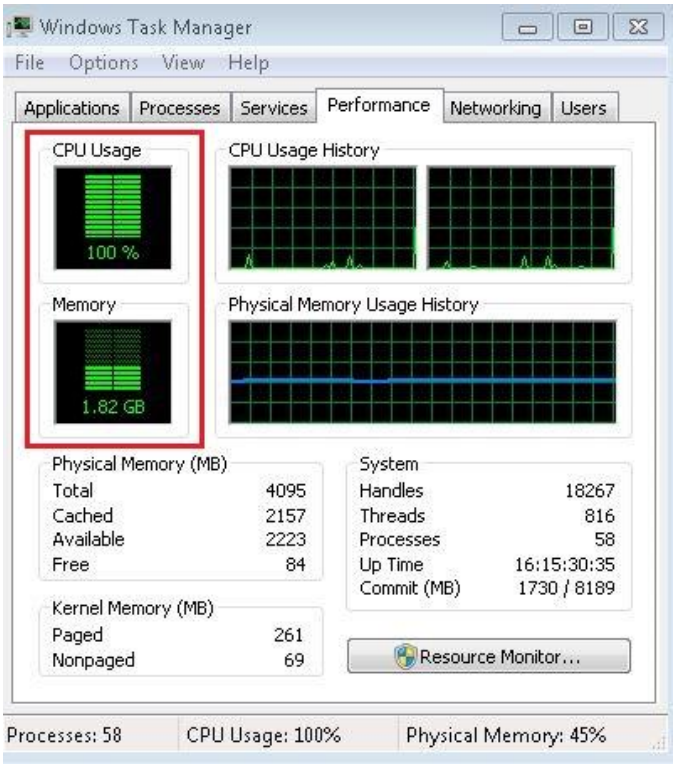

Figure 29

Test 1: CPU Utilization Windows7 Environment (Node.js)

Table 4

Test 1: Ubuntu Environment (Apache-PHP)

Execution Time in Seconds/CPU Utilization in %				
Concurrency/Requests	**500**	**1,000**	**5,000**	**10,000**
10	0.876 / 9.2%	1.777 / 21.2%	8.783 / 19.7%	17.459 / 17.9%
100	0.886 / 12.4%	1.740 / 19.7%	9.018 / 19.1%	18.041 / 5.2%
200	0.932 / 6.6%	1.762 / 17.1%	9.138 / 20.5%	17.946 / 37.9%

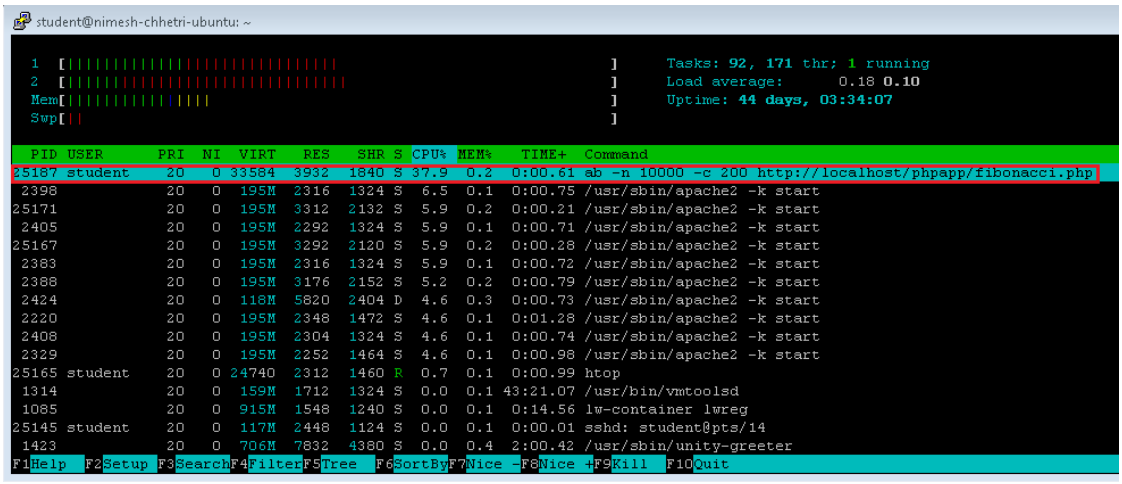

Figure 30

Test 1: CPU Utilization Ubuntu Environment (Apache-PHP)

Surprisingly, the execution time of the 20th Fibonacci number in Apache-PHP was quite low in the Ubuntu environment but so was the case for Node. The measurements of response

time along with CPU utilization time of Apache-PHP and Node in Ubuntu is listed in Table 4 and Table 5. The CPU activity was logged using the *htop* command as shown in Figure 30 and Figure 31. The peak CPU utilization ratio using Apache-PHP was 37.9% whereas Node had 97.6% (in a single thread). The comparative graph in Figure 33 shows that Node falls short in terms of CPU Utilization ratio but its response time performance is remarkably good for both the Windows and Ubuntu environments as shown in Figure 32. Hence, when heavy computation is involved Node is probably not the good choice for application development as it consumes way too much of CPU time in contrast to a conventional model.

Table 5

Test 1: Ubuntu Environment (Node.js)

Execution Time in Seconds/CPU Utilization in % (Single Thread)				
Concurrency/Requests	**500**	**1,000**	**5,000**	**10,000**
10	0.271 / 21.2%	0.457 / 25.3%	2.420 / 74.8%	3.934 / 98.6%
100	0.229 / 19.5%	0.443 / 24.5%	2.018 / 95.7%	3.787 / 98.0%
200	0.351 / 14.0%	0.509 / 21.3%	2.121 / 94.4%	4.337 / 99.7%

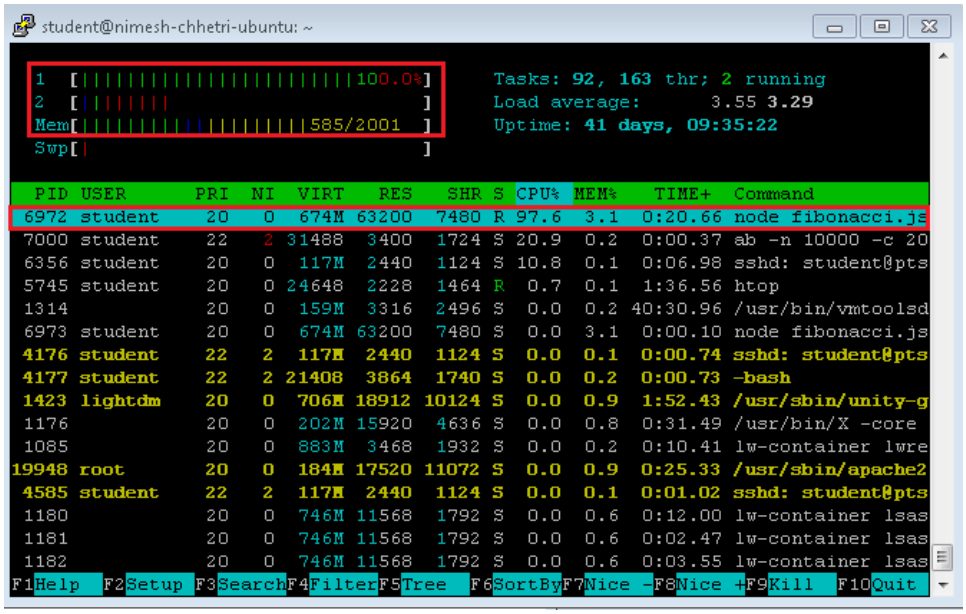

Figure 31

Test 1: CPU Utilization Ubuntu Environment (Node.js)

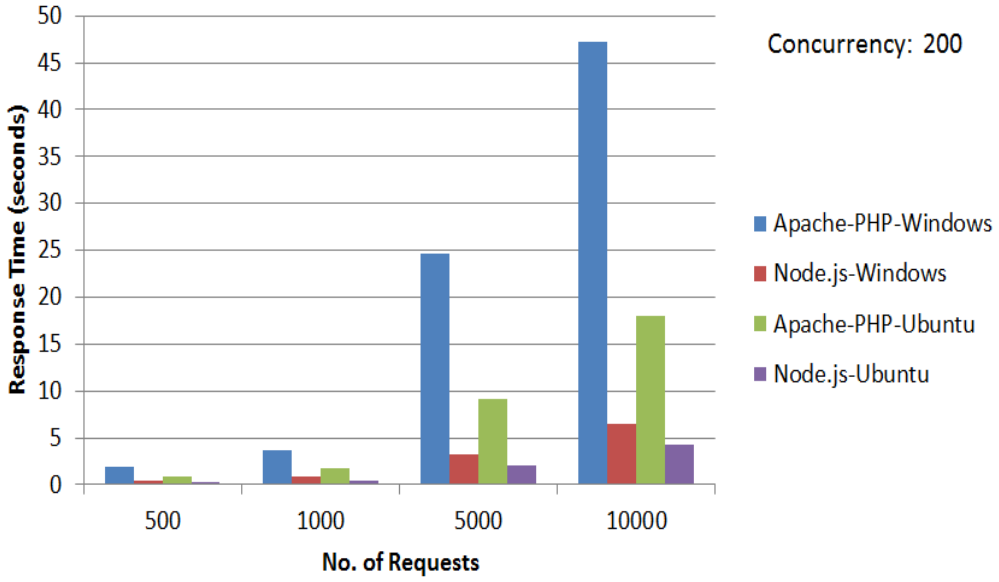

Figure 32

Test 1: Response Time Graph with HTTP Requests & 200 Concurrent

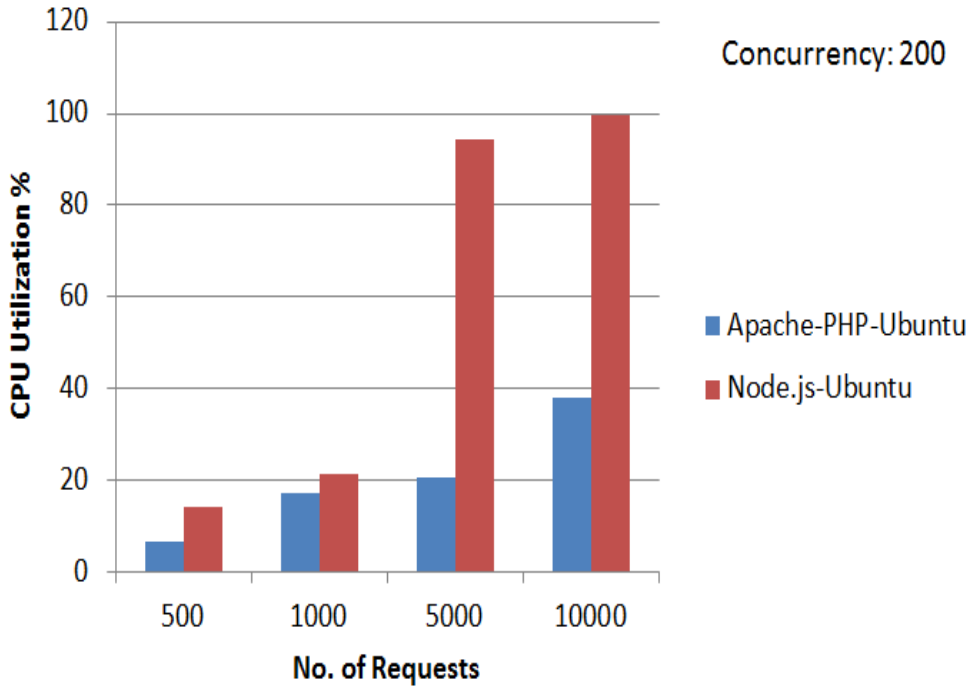

Figure 33

Test 1: CPU Utilization ratio with HTTP Requests & 200 Concurrent (Single Thread)

Test 2 is the test for I/O operations and as discussed, Node was expected to perform very well in this experiment and the results were as good as expected. The measurements of response time for Apache-PHP and Node in the Windows environment are shown in Table 6 and Table 7. From these two tables we observe that Node was almost twice as fast for several sets of metrics in the Windows environment. We observe that if sets of 100, 1000 and 200, 1000 combinations of concurrency, and requests are compared then Node is twice as much as fast as Apache-PHP.

Table 6

Test 2: Windows 7 Environment (Apache-PHP)

Execution Time in Seconds				
Concurrency\Requests	**500**	**1,000**	**5,000**	**10,000**
10	18.542	43.646	163.940	342.763
100	24.726	49.577	241.738	485.893
200	21.824	44.382	208.962	422.542

Table 7

Test 2: Windows 7 Environment (Node.js)

Execution Time in Seconds				
Concurrency\Requests	**500**	**1,000**	**5,000**	**10,000**
10	12.043	31.903	119.573	306.478
100	12.745	25.771	112.694	264.592
200	11.918	21.403	130.790	245.263

Table 8

Test 2: Ubuntu Environment (Apache-PHP)

Execution Time in Seconds				
Concurrency\Requests	**500**	**1,000**	**5,000**	**10,000**
10	1.999	3.838	19.796	38.426
100	2.003	4.169	22.190	39.922
200	1.990	6.823	21.134	42.091

Table 9

Test 2: Ubuntu Environment (Node.js)

Execution Time in Seconds				
Concurrency\Requests	**500**	**1,000**	**5,000**	**10,000**
10	1.379	2.601	13.066	20.235
100	1.531	3.201	12.988	22.346
200	1.533	3.065	14.253	29.648

The measurements of response time for both the Node and Apache-PHP for Test 2 in the Ubuntu environment is shown in Table 8 and Table 9. The results are quite similar to what it was in the Windows environment. For several instances such as for 100, 10000 and 100, 5000

combination of concurrency and requests, Node is essentially two times faster than Apache-PHP. The comparative graph for both environments for both Node and Apache-PHP is shown in Figure 34. From these results we assert that Node is a high performer for any I/O operations. Test 2 involved experiments on the I/O operation task for reading content from a large text file but this can be emulated for other I/O operation tasks as well such as data fetching from a database.

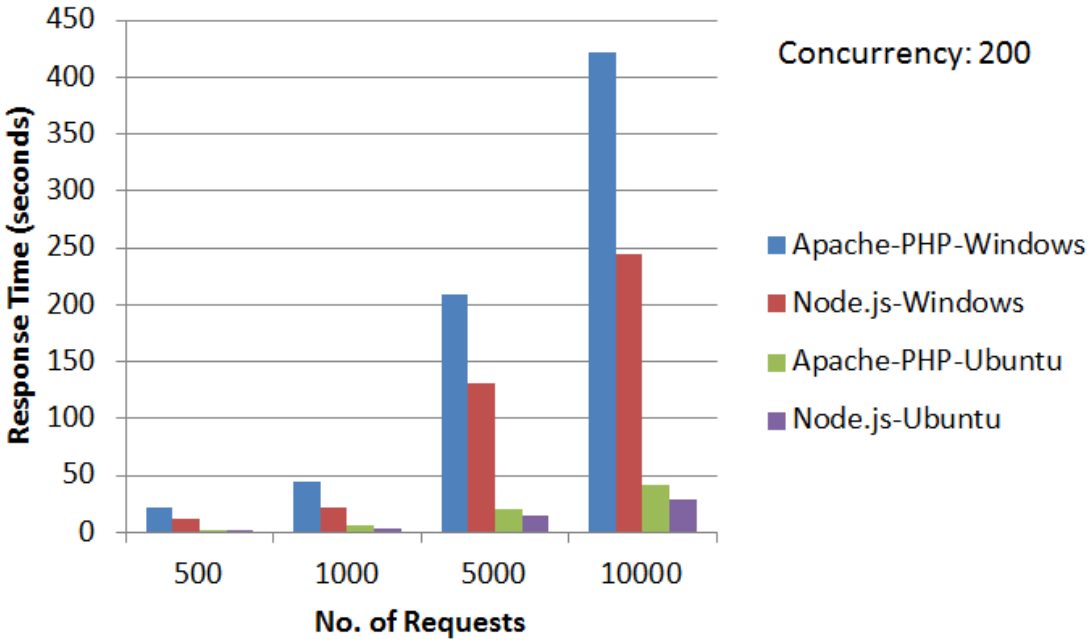

Figure 34

Test 2: Response Time Graph with HTTP Requests & 200 Concurrent

Chapter 7: Limitations of Node.js

We established Node's benefits in terms of performance and scalability. In this paper, we established Node as the superior and neat platform for application development. Our benchmarking results in Chapter 6 clearly shows Node's true potential in handling concurrency with limited resources. However, some of Node's benefits have become the reasons for its weaknesses. Part of the utility of Node is that there are limited weaknesses in the typical sense of the word [12]. Having been developed only in 2009, Node is currently still in the development life cycle. Perhaps the largest problem with the platform at its current state of development is the lesser developed ecosystem [12]. Comparatively, Node is at infancy stage with small development community and support.

Poor handling of heavy server-side computation

In general, any CPU intensive operation nullifies any of the throughput benefits Node offers with its event-driven, non-blocking I/O model because any incoming requests will be blocked, while the thread is occupied serving requests [12]. Node struggles in handling of very high computationally intensive tasks, because whenever it executes long-running task, Node will queue all remaining incoming requests, because it implements single-threaded architecture with an event loop [36]. As illustrated in Test1 of our benchmarking tests, Node utilized almost 100% of CPU (in a single thread) for calculating the 20^{th} Fibonacci number. Practically, the load ratio scales up higher than that, and undoubtedly Node will not be able to cope with it. For Node to be an effective programming language, it should be able to handle any sort of applications including heavy computation tasks. Node is single-threaded and uses only a single CPU core which limits

that possibility. This can probably be resolved by adding concurrency on a multicore server, in the form of a cluster module. A cluster module creates child processes (workers) which share all the server ports with the main Node process (master). Thus, the elegant solution for Node to deal with heavy computation and for scaling up the applications would be to split a single process into multiple processes or workers [33].

Server-side application with relational database

Node integrates quite well with NoSQL databases which are non-relational in nature. However, relational database tools for Node are still in their early stages and are rather immature [12]. Node is very popular with NoSQL databases but is seldom used in combination with relational databases. The integration of Node with relational databases is still at an early stage and is not so solidly established. While NoSQL is becoming more popular, relational databases are not going to be phased out anytime soon. Considering that, it becomes necessary for Node to become well-integrated with the relational databases.

Complexity with callback function

Asynchronous I/O combined with a callback function is the salient feature of Node that allows it to handle multiple concurrencies. A callback function in Node is an anonymous function that is usually nested together with some other factory methods. When the logic of the code becomes complex, Node might suffer from the problem termed *callback hell* [34]. Callback hell is the occurrence of an ugly nesting of multiple asynchronous JavaScript functions that go to multiple level of nesting in depth. The extensive use of asynchronous threading can make complicated logic very difficult to write. Anything requiring interaction between more than three

external APIs ends up causing code to nest many levels deep; also making the code very difficult to read and document [35].

Ecosystem in Development

Node's ecosystem is currently in its development stage. Although, the number of public libraries or modules in Node online repository (See Chapter 1) is increasing, and the community and support is expanding, there is still a long way to go, before its ecosystem is sufficiently established. Most of the APIs provided by Node may not be stable and they cannot be assumed to be ready in production. However, the Node ecosystem is expected to expand and become solid.

Adherence to JavaScript

JavaScript is the core of Node. Node is a JavaScript language with additional functionalities and features with both the client-side and the server-side scripting capability. It is discussed in previous section (See Chapter 4) how Node is benefited with its base on JavaScript and how Node suffers in terms of the security aspects (See Chapter 5) that it inherited from JavaScript. Additionally, JavaScript has not been developed for use on servers until recently. JavaScript is still very new in the server-side environment. Many solutions that would be otherwise easy to code in JAVA or .NET are far more difficult or even impossible in Node (or JavaScript in general). One specific example is with XML schema validation and XML transformations. There are not any modules with more than a basic level of support for XML schema validation, missing functionality for namespaces, and modularized schemas. Likewise, there is not any suitable module for transforming XML [35].

Chapter 8: Limitations and recommendations for further Study

Node is a very new application platform. Node's features and benefits have attracted many developers worldwide. As a result, the scope of Node has increased significantly. Due to the wide scope of Node issues, we covered a limited number of aspects of Node in this paper that includes, what Node is, how it is installed, how it works, what a Node Module and NPM is, how Node is different from JavaScript and AJAX, what security holes it has, and what its limitations are.

There are some limitations and shortcomings in our study. First of all, we don't provide detailed information on popular in-built Node modules available in NPM repository. There are many popular Node modules that are used in Node application development. For instance, modular frameworks such as Express.js and Jade are not discussed in this paper. Apart from that, application development has not been the prime focus. Therefore, Node application development is not discussed. Node is the significant part of the modern all JavaScript based web application development framework termed the MEAN stack. A further investigation of Node application development using MEAN would provide a better understanding of the application development process in Node. The differences and similarities described between JavaScript and Node in this paper is at a very high level. More emphasis on conventional JavaScript can illustrate those differences more clearly. Node is more similar to the JavaScript library termed jQuery than to conventional JavaScript. An emphasis on the jQuery library, in addition to classical JavaScript, will help in understanding Node and its syntax. Additionally, the real-time application development module of Node: Socket.io is only explained by us briefly. Socket.io is very powerful and can be used to develop several sorts of real-time applications

including real-time analytics, instant messaging, binary streaming, and document collaboration. A further study of Socket.io would reveal the power of Node to facilitate real-time application development that is not possible with AJAX. Lastly, the benchmarking results in this paper are based on the comparison of Node with Apache-PHP. A fair benchmarking trial would be to compare Node with the measurements made using the asynchronous programming feature available in other programming languages such as the SignalR library available in C# which has similar functionalities to Node.

Chapter 9: Conclusion

In this paper we showed that Node has transformed the usability of JavaScript, making Node a complete programming language. From browsers to server-side scripting outside of browsers, Node has made possible the availability of a runtime environment, a library full of free useful modules that can be imported by using an in-built tool named NPM. Node uses non-blocking, event-driven I/O asynchronous programming to be lightweight and be efficient. We showed that setting up a Node environment is simple, and Node is available to all major operating systems.

Node is based on the familiar syntax of JavaScript, but differences do exist. Node can be confused with AJAX, but both of them are completely different tools with both using JavaScript as the base. Although AJAX was introduced to provide real-time interaction in the web application, AJAX failed to do so while exhibiting a significant wastage of bandwidth and resources. Node, via its socket.io module was shown to overcome that deficiency in AJAX by introducing an efficient real-time interaction in the application. Node's performance with respect to another server-side scripting language PHP is quite remarkable as shown by our benchmarking test results. Apart from its benefits, Node does have some security holes. If Node applications are not programmed with good error handling and input validation then those applications can be vulnerable to attacks. Therefore, it becomes the responsibility of Node developers to make Node applications secure.

With all of its advantages, Node plays a critical role in the technology stack of many high-profile companies who depend on its unique benefits. Node was created to solve the I/O scaling

problem. So, if a use case does not contain CPU intensive operations nor access any blocking resources, one can exploit the benefits of Node while experiencing fast and scalable applications development with the power of Node.

References

[1] Kurniawan, A. (2014). Node.js Succinctly. Synfusion Inc.

[2] Govett, D. (2010, March). Learning Server-Side JavaScript with Node.js. Retrieved from Envato Tuts+: http://www.webcitation.org/6ePoNkZwD

[3] Teixeira, P. (2013). Hands-on Node.js. In P. Teixeira, Hands-on Node.js. Lean Publishing.

[4] Young, A. (2012, May). Windows and Node: Getting Started. Retrieved from Dailyjs: http://www.webcitation.org/6ePozY7jz

[5] Tom Hughes-Croucher, M. W. (2012). Node: Up and Running. In M. W. Tom Hughes-Croucher, Node: Up and Running. O'Reilly Media Inc.

[6] Ortiz, A. (2013, March). Server-side Web Development with JavaScript and Node.js. Retrieved from http://webcem01.cem.itesm.mx:8005/node/node.html

[7] Mike Cantelon, T. H. (2013). Node.js in Action. Manning Publications.

[8] Rauch, G. (2012). Smashing Node.JS JavaScript Everywhere. John Wiley & Sons Inc.

[9] Teixeira, P. (2013). Professional NodeJs: Building JavaScript-Based Scalable Software. John Wiley & Sons Inc.

[10] Wandschneider, M. (2013). Learning Node.js A Hands-On Guide to Builiding Web Applications in JavaScript. Pearson Education.

[11] Capan, T. (n.d.). Toplal. Retrieved from Why The Hell Would I Use Node.js? A Case-by-Case Introduction: http://www.webcitation.org/6ePpS09lg

[12] Joseph Delaney, C. G. (n.d.). Node.js at a glance. Whale Path Inc.

[13] Cois, C. A. (2013, June). Why You Should Learn Node.js Today. Retrieved from Udemy Blog: http://www.webcitation.org/6ePpd1ib8

[14] Wilson, J. R. (2013). Node.js The Right Way Practical, Server-Side JavaScript That Sales. The Pragmatic Programmers LLC.

[15] Node.js. (n.d.). Retrieved from Node.js: http://www.webcitation.org/6eRYp2n8a

[16] Ajax Basics. (2015, August). Retrieved from Webucator: http://www.webcitation.org/6ePpmPP3W

[17] Godde, A. (2015, January). Why AJAX Isn't Enough. Retrieved from Smashing Magazine: http://www.webcitation.org/6ePptxaNV

[18] Holzner, S. (2009). AJAX A Beginner's Guide. McGraw-Hill Companies.

[19] Anderson/differential.io, D. (2014). How Node.js Can Accelerate Enterprise Application Development. Modulus.

[20] Endler, D. (2002). The Evolution of Cross-Site Scripting Attacks. iDefense Inc.

[21] Sullivan, B. (2011). Server-Side JavaScript Injection. Adobe Secure Software Engineering Team.

[22] Barnes, D. (2013). Node.js Security. Packt Publishing.

[23] Subramani Rao, S. R. (2011). Denial of Service Attacks And Mitigation Techniques: Real Time Implementation With Detailed Analysis. The SANS Institute.

[24] Baldwin, A. (2014, November). Regular Expression DoS and Node.js. Retrieved from Lift Security: https://blog.liftsecurity.io/2014/11/03/regular-expression-dos-and-node.js?utm_source=nodeweekly&utm_medium=email

[25] Regular expression Denial of Service - ReDoS. (2015, November). Retrieved from OWASP: http://www.webcitation.org/6ePqK8gbl

[26] OWASP Validation Regex Repository. (2014, July). Retrieved from OWASP Validation Project: http://www.webcitation.org/6ePqRiV4I

[27] Node.js v5.1.0 Documentation. (n.d.). Retrieved from Node.js: http://www.webcitation.org/6ePqYdtwX

[28] McCune, R. R. (2011). Node.js Paradigms and Benchmarks. University of Notre Dame.

[29] Rubio, M. (n.d.). Fibonacci numbers using mutual recursion. Rey Juan Carlos University, Department of Computer Science.

[30] Martonca, E. (2015, June). Why all the hype about Node.js? Retrieved from http://www.webcitation.org/6ePrX7R2R

[31] Using a package.json. (n.d.). Retrieved from npm: http://www.webcitation.org/6eaIdaR9F

[32] Pasquali, S. (2013). Mastering Node.js. Packt Publishing.

[33] Kamali, B. (2015, July). How to Create a Node.js Cluster for Speeding Up Your Apps. Retrieved from Sitepoint: http://www.webcitation.org/6ew832ZZA

[34] Callback Hell (n.d.). Retrieved from Callback Hell: http://www.webcitation.org/6ew8DXVK0

[35] McIlvenna, S. (January, 2014). Retrieved from Evolving Software: http://www.webcitation.org/6ew8PX8Dh

[36] Posa, R. (March, 2015). Retrieved from JournalDev: http://www.webcitation.org/6ew9SMBVa

[37] Flanagan, D. (2006). JavaScript The Definitive Guide. O'Reilly Media Inc.

Appendix

Node.js Installation

Node's installation is fairly simple. It can be downloaded and installed easily and then get it up-and-running in a matter of minutes. Node can be installed out of the box on Windows, Linux, Macintosh, and Solaris. Depending upon the platform, a package installer can be downloaded for Windows or Mac OS that can be executed to install Node. For Linux distribution, the latest stable source code can be downloaded and built [3].

Installation on Windows

Installing Node in the Windows environment is as easy as installing any other Windows application. In order to begin the installation, download the Windows Installer (MSI) file from the official website of Node. Click on the download file to initiate the Windows installer with a wizard which is pretty easy to follow. It's just like installing any other Windows program - the Node binaries will end up in `C:\Program Files (x86)\nodejs\` (in 32 bit Windows) and will be accessible from `cmd.exe` [4].

Alternatively, Node can be installed using package instead of installer. For that, a command line installer for Windows such as the scoop or the chocolatey should be installed. In order to install scoop, Powershell 3 should be installed in the machine and ensure to change the execution policy (i.e. `enable Powershell`) with `set-executionpolicy unrestricted -s cu`. Then from the command line, Node can be installed directly using this command:

```
scoop install nodejs
```

After the installation, Node executable can be run from the command line to check if it is installed successfully by typing `node -v`. The command should show the version of the Node, if the installation is successful [5].

A Node shell or *REPL* (*read-eval-print loop*) can be run to interactively test the JavaScript code. Node REPL is an interactive Node programming environment great for testing out and learning about it [5]. At the terminal window type: `node` and that will allow entering any JavaScript expression after the shell prompt "`>`".Type `.exit` followed by *Enter* to quit the shell or press *Ctrl-C* twice [6]. For the production development, any text editor can be chosen to write a Node program, save it with a *.js* extension anywhere in the machine. Then, in order to execute the program from the command line type [5]:

```
node program_name.js
```

Installation on Ubuntu

Installation of Node in Linux is fairly simple. In this paper, we used the Ubuntu for Linux. Therefore, this part discusses the steps involved in setting up the Node in the Ubuntu that requires following 2 primary tasks. At first is the installation of pre-requisite packages and then the compilation of Node [7].

There are some pre-requisite packages that need to be installed in the Ubuntu before installing Node. This can be done by executing the single line command given below:

```
sudo apt-get install build-essential libssl-dev
```

After compiling the Node, some more steps are required. First, create a temporary folder by entering the following command from the command line:

```
mkdir tmp
```

Then, navigate into the directory and enter the following commands to get the *tar* of Node setup files:

```
cd tmp
```

```
curl –O http://nodejs.org/dist/node-latest.tar.gz
```

Once the download is complete, enter the following command to extract the tar:

```
tar zxvf node-latest.tar.gz
```

Next, enter the following command in sequence to move inside the extracted directory, to run a configuration script and finally to compile Node respectively:

```
cd node-v*
```

```
./configure
```

```
make
```

Once the text stops scrolling, and after the command prompt comes in, enter the final command in the installation process:

```
sudo make install
```

This will install Node in the Ubuntu. The successful installation of the Node can be verified by issuing the following command to show the version of Node in the terminal like it was shown in the Windows installation:

```
node -v
```

Source Code

The following script was used for calculating the 20th Fibonacci number using Apache-PHP.

```php
<?php
ini_set("precision",50);
$fibnum = 20;
function fibonacci($number) {
    if ($number < 2) {
        return 1;
    } else {
        return (fibonacci($number-2) + fibonacci($number-1));
    }
}
echo $fibnum.'th Fibonacci Number is: ';
echo fibonacci($fibnum);
?>
```

The following script was used for calculating the 20th Fibonacci number using Node.js.

```javascript
//fibonacci.js
var fibnum = 20;
var resFibNum;
'use strict';
var http = require('http');
var port = 8000;
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    //print the result of the function
    resFibNum = fibonacci(fibnum);
    console.log(fibnum+'th Fibonacci number is: ' + resFibNum);
    res.end(fibnum+'th Fibonacci number is: ' + resFibNum);
}).listen(port);
console.log('Listening at port ' + port);
// function for calculating fibonacci number recursively
function fibonacci(n) {
  if (n < 2)
    return 1;
  else
    return fibonacci(n-2) + fibonacci(n-1);
}
```

The following script was used for reading the large data file content using Apache-PHP.

```php
<?php
$fname = 'DFile.dat';
$result = file_get_contents($fname);
echo $result;
?>
```

The following script was used for reading the large data file content using Node.js

```
//filereader.js
var http = require('http');
var server = http.createServer(handler);
var fs = require('fs');
function handler(request, response) {
    response.writeHead(200, {'Content-Type': 'text/plain'});
    fname = 'DFile.dat'
    //read from the file
    fs.readFile(fname, function (err, data) {
        if (err) throw err;
        result = data;
        response.end(result);
    });
}
server.listen(8124);
console.log('Server running at http://127.0.0.1:8124/');
```