Comparison of Cloud Application Development in Node.Js, Java, and Go

Computer Science Honours Project — COMP 4905

Carleton University, Ottawa, ON, Canada

Brandon Schurman (100857068)

Dr. D. Deugo

April 13, 2016

Acknowledgements

I would like to thank Professor Dwight Deugo for supervising me on this project, and collaborating with me on key design points with the three server implementations. Professor Deugo's teachings about Java EE server programming has helped me in many ways, and this project would not have been possible without his expertise in this domain. I would also like to thank the team at Materializecss for providing one of the most easy to use and best looking open source UI frameworks I've worked with, and for promptly responding to my (and the other developer's) posted issues on GitHub.

Abstract

This report contrasts the performance of three popular server programming languages, Node.Js, Java, and Go, when used in the development and deployment of modern Cloud Computing software. The three servers are measured in terms of their performance on Basic Operations (for example, reading/writing to a database), and Long Operations (which may benefit from the use of multithreading). As the Cloud Computing industry grows, with providers such as AWS, IBM Bluemix, and Microsoft Azure increasingly gaining more customers, the demand for high performance software becomes increasingly more important as well. These applications may run long running tasks, such as compiling and deploying applications on the web, building and running Docker containers, and even running test suites against a user's live web server. Furthermore, for SaaS customers of Cloud Computing platforms, high performance software can also reduce costs for server hosting. For example, when these users are billed by Gigabytes-per-hour of runtime usage, if their servers can handle their user's requests quicker, and can handle high amounts of user traffic better, it could in turn be cheaper to host their app in the long run.

Table of Contents

1. **Introduction**
    1.1. Problem
    1.2. Motivation
    1.3. Goals
    1.4. Objectives
2. **Background**
3. **Approach**
    3.1. UI Layer
    3.2. Server Design
    3.3. Test Suite Design
    3.4. Personal Experience
4. **Results**
    4.1. Basic Operations
    4.2. Long Operations
5. **Conclusion**
6. **References**

1. Introduction

This study contrasts the practical performance of Java, Node.Js, and Go, when used to develop and run modern Cloud based applications. The same basic server is implemented in each of these languages, and is designed to test Basic Operations, such as processing a request which accesses a database; as well as a more computationally heavy Long Operation (an inefficient sorting algorithm) which is used to simulate long running tasks which may be executed by servers in a Cloud Computing platform. Such tasks might require different demands from software, than traditional web applications have provided in the recent past. A User Interface is provided to interact with the three servers, to contrast their the performance in hands-on way so that users can experience the performance trade-offs themselves. In all of these operations, the response time is measured and analyzed at various levels of network traffic. A reduction in average response time can mean not only faster servers for users, but also reduce the runtime costs since some Saas (Software as a Service) and IaaS (Infrastructure as a Service) providers can bill their clients by Gigabytes-per-hour of usage [3]. Other factors, such as practical scalability, JVM performance, and simplicity to develop, will be touched-on briefly. As well as some personal experience after developing the exact same server application in these three languages.

1.1. Problem

Java, Node.Js, and Go are three popular server programming languages used in web and cloud development. The intention of this report is to run a practical

comparison on the performance of these three languages. These server programming languages are tested by running some typical tasks that are analogous to tasks that are run in a production system. Some tasks will involve simple read and write operations on a cloud hosted database, and another task will involve a more computationally intensive operation, which takes time for the server to complete. For the database, the exact same instance of IBM Cloudant is shared among the three servers [7]. The intensive operation may benefit from the use of concurrent threads. In all of these tasks, the performance will be measured as an average response time, under various loads. Contrasting the performance of these three servers when used to run the same simple application will provide valuable quantitative evidence of their advantages and or disadvantages when used to execute certain tasks.

1.2. Motivation

Intuition suggests that Java and Go may beat Node.js in performance [6] [14] [16], since they are both compiled, support static typing, and can run in multiple concurrent threads. Node.Js may have less complex code, but slower performance, since it is interpreted, dynamically typed, and single threaded by nature. But the Node.Js community, however, often argues that this JavaScript based server actually performs faster in practice than a traditional Java EE web server [6] [14]. Furthermore, Google's new Go language is quickly gaining traction and support from the developer community [18], with many users impressed by it's simple syntax and fast performance. It could be the case that this new contender is actually less complex and faster in practice than the other two. Nonetheless, the performance of these languages in a

Cloud Computing environment has been touched on less than in a traditional web environment. In a Cloud Computing environment, the types of operations requested by users can vary from quick lightweight tasks, to computationally heavy, long running tasks. Furthermore, costs of running servers in a Cloud Computing environment are different than on-site dedicated hardware, as these applications were often deployed in the past [19]. This report may better educate the developer community so that they can make more informed decisions when selecting which language to program their cloud application in, especially when performance and/or runtime costs are a concern.

1.3. Goals

The main goal of this report is to provide metrics on response time performance, under various loads, which will serve to educate the community on the tradeoffs between using Node.Js, Java, and Go in a Cloud Computing environment. Performance may vary between a single request for a lightweight operation, to high traffic with many requests for both lightweight,operations and to more computationally heavy operations. The response times of these requests can give insight into how servers written in these three languages react under these different circumstances. These metrics will provide valuable quantitative evidence suggesting the trade-offs between each language under different conditions and usages.

1.4. Objectives

This objective of this project is to deliver three different implementations of a basic Cloud based application in the three chosen programming languages. The applications are designed to test both Basic and Long operations, to contrast their performance on different tasks that are often executed in a Cloud Computing environment. In each implementation, a web server will present a simple user interface that allows a user to execute a number of tasks. The Basic Operations task will invoke a simple database query to read and write user data from an IBM Cloudant database (which is an implementation of Apache CouchDB). This database is selected since it operates through a simple web interface and REST API, which should be a fair choice between the three servers. The other Long Operation tasks will be designed to test the system under a computationally intensive load, especially one that may benefit from the use of concurrent threads. For an accurate comparison, the servers should be run on the same hardware. The web servers are thus either run on a laptop for local testing, and they are also deployed within the same Virtual Machine running on IBM Bluemix. On each of the implemented tasks, the response time will be measured and presented to the user in the UI layer when their request completes. The UI layer is intended to give users hands-on experience with the performance of the three servers. A more formal testing suite will be written to test the servers more quantitatively, and under various loads (e.g. moderate traffic, or high traffic). In this report, the average of the response times for each language ands each task will be compared, graphed, and contrasted through the use of tables. Another intention of this report is to provide insight into the cost of these applications when hosted in a Cloud environment like IBM Bluemix,

Microsoft Azure, or AWS, since applications on these platforms can sometimes be charged for runtime usage [1] [2] [3] [5].

2. Background

Intuition may lead one to believe that Node.Js is fast, but languages like Java and Go should outperform Node.Js on the Long Operation test. Since Node.Js is single threaded, and loosely-typed, it is natural to infer that Node.Js sacrifices fine-tuned performance in favor of code simplicity. Much of the Node.Js community, however, argues that Node is just as fast, if not faster than Java, in typical applications. In an article published by InfoWorld.com in 2015, Node.Js is claimed to outperform Java on speed [6]. In particular, the article states the following:

“People love to praise the speed of Node.js. The data comes in and the answers come out like lightning. Node.js doesn't mess around with setting up separate threads with all of the locking headaches. There's no overhead to slowdown anything.”

The quoted text claims that Node.Js performs faster, in part because it “doesn’t mess around with setting up separate threads”. This very fact, however, may be a significant reason that Node.Js can be significantly slower than Java (and Go), especially when developing modern Cloud applications. Cloud hosting software such as AWS, Microsoft Azure, or IBM Bluemix, are capable of running some very long tasks for their clients. For example, these hosting platforms are capable of compiling and deploying web applications, as well as building and running Docker containers. These operations can take minutes to run, and if their servers are single threaded like in Node.Js, a single request to execute one of these task may block an entire server instance until it completes. In the context of Cloud Computing, the above quote may in fact be false, and

for one of the very reasons it was claimed to be true. Furthermore, the fact that Node.Js is strictly single threaded can be problematic for applications under extreme load. Of course Node.Js apps (and any app for that matter) can be “scaled up” to run multiple instances so that load balancing can be used to reduce traffic from bottlenecking on a single node. The problem with doing this in a Cloud hosting environment is that running multiple instances of an application can be costly [3] [20], whereas if the server could naturally handle higher amounts of traffic, some running costs might be saved. Another article from DZone, published in 2013, claims that response times from a Node.Js server are recorded to be 20% faster than the response times from a Java EE server [14]. In fact, this article even claims that this is regardless of the use of multithreading in Java. The article quotes the following:

In this test the different concurrency models between single-threaded Node.js and multi-threaded Java EE made no difference. To test Node.js at higher concurrency levels – where it is supposed to outshine multi-threading – other problems like the number of open files need to be considered.

However, this claim is somewhat suspicious, since a single Node.Js server cannot process other incoming requests if it is currently processing some long computationally heavy task. In this report, we’ll be analyzing response times of these servers under our own Cloud Computing oriented tests, to find evidence that either confirms or denies these claims. Clearly, in our new Cloud-centric industry, the definition of a “fast” server programming language needs to be re-evaluated.

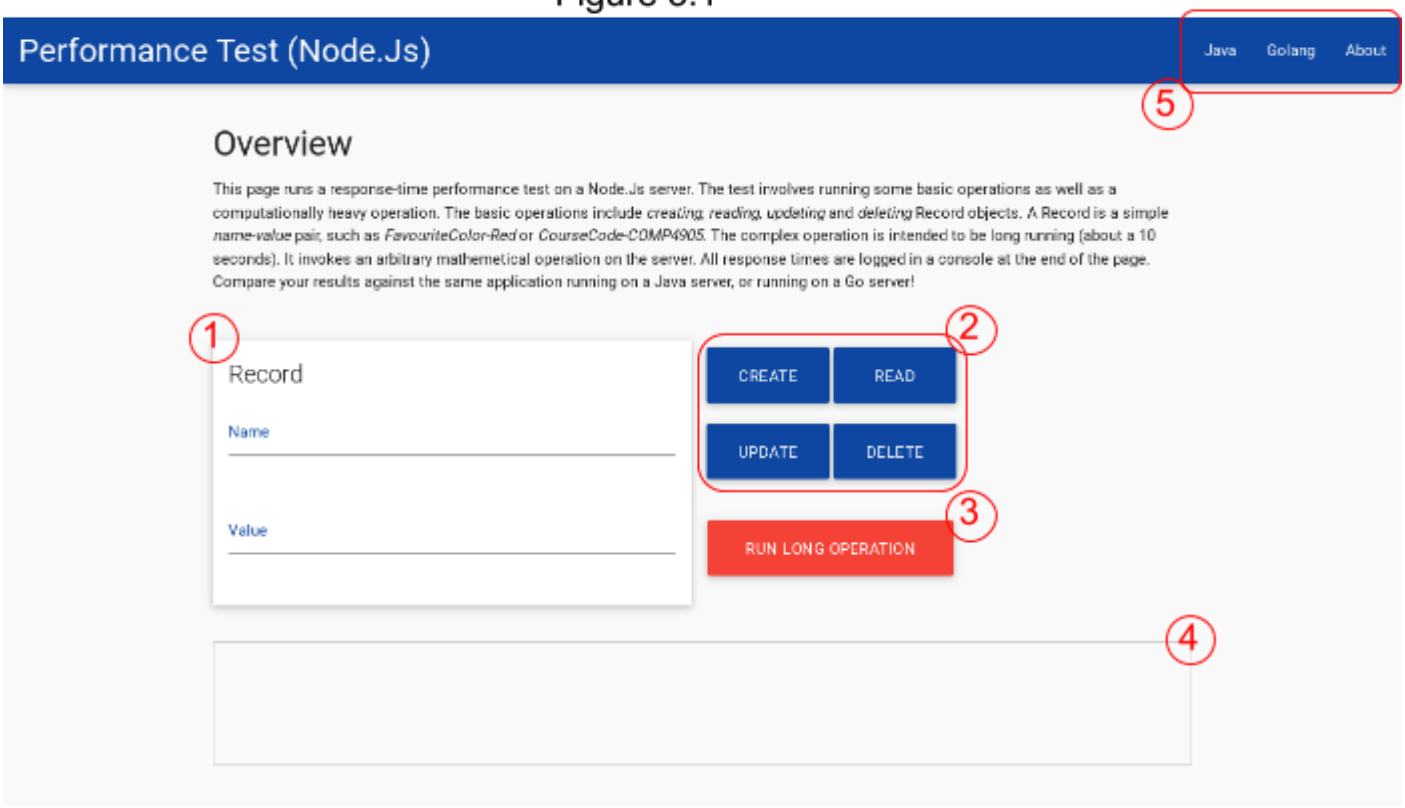
3. Approach

Our approach involves developing 3 nearly identical servers in Node.Js, Java, and Go. Each of these servers will implement the same RESTful API, and operate on the same instance of a Cloudant database. The RESTful API operates on basic Record entities, which are just simple key-value pairs. A long operation is also available, which invokes a $O(n^2)$ bubble-sort algorithm on the exact same array of floating point numbers. The three servers offer the same user interface (UI) to interact with their RESTful APIs. Furthermore, all three of the servers are run on the exact same hardware, for consistency. Formal quantitative testing involves two groups of tests, Basic and Long Operation. There are 3 Basic tests which are designed to simulate some typical tasks like reading and writing to a database. The 2 Long Operation tests are designed to show how the servers work when running long computationally heavier tasks.

3.1. UI Layer

First off, let's a take a look at the UI layer, which interacts with the servers we are testing. In Figure 1.1, component (1) shows the Record area, where values can be entered and written to the database, or vice-versa, where values are displayed after reading them from the database. The Record must have a unique name and an arbitrary value. The Record is not designed to be useful in any way, but it is a simple data structure that is written, read, updated, and deleted from the database. It is representative of many simple data resources that are handled in real production systems (e.g. a user profile).

Figure 3.1



Component (2) is a set of operations available on the Record. They each make a different call to the server's REST API using the Record data. Component (3) is the Long Operation button, which initiates the long running, and purposefully inefficient sorting algorithm on the server. The long operation is not intended to be useful, but is rather representative of other long processing task which may be seen in production cloud applications. Component (4) shows all responses as they are returned from the server, with their response time in milliseconds, after the server returns the request. Lastly, the tabs in component (5) are used to switch between the same UI, but setup to interact with the other two servers, or show an About page.

3.2. Server Design

Each of the three servers implement the exact same procedures, and use the exact same Cloudant database instance. So for example in the REST API, the Update procedure involves first checking if a Record with the given name is in the Cloudant database, and if it is, it is then updated with the new value. This procedure involves two database calls and roughly the same amount of logic-checking code in each of the three servers in Node, Java and Go. The same Bubble Sort algorithm is also implemented in three servers for the Long Operation, and the sorting is always done on the same array of 50,000 floating point numbers. For request routing, the Express NPM package is used in Node.Js, the gorilla/mux router is used in Go, and Jersey is used for Java [4]. These frameworks are selected as they commonly used in each of the three languages. Here's an example of the Read Basic Operation, which is executed on the server when the user clicks the "READ" button from component (2) in the UI.

In Node.Js the GET request is implemented as follows:

```
app.get('/record', function(req, res) {
        res.header('Content-Type', 'application/json');
        var name = req.query.name;
        if ( validator.trim(name) == '' ) {
            res.status(400).send('{"error": "name is required"}');
            return;
        }

        findRecord(name, function(error, response, body) {
            if ( error ) {
                res.status(500).send(error);
                console.log(error);
            } else {
                var docs = JSON.parse(body).docs;
                if ( docs.length > 0 ) {
                    res.send(docs[0]);
                } else {
                    res.status(404).send('{"error": "No Record with the
name \''+name+'\' was found"}');
                }
            }
        });
```

```
    });
```

Where the `findRecord` function is defined separately as:

```
 function findRecord(name, callback) {
        var options = {
            url: db+'/records/_find',
            headers: { 'content-type': 'application/json' },
            body: JSON.stringify({
                selector: {
                    _id: { $gt: 0 },
                    name: name,
                }
            })
        };
        request.post(options, callback).auth(dbUser, dbPass);
    };
```

In Go, the request handler is actually implemented entirely in a single funciton, as follows:

```
func GetRecord(w http.ResponseWriter, r *http.Request) {
      // get name param from request
      name, err := url.QueryUnescape(r.URL.Query().Get("name"))
      if err != nil {
            fmt.Printf("error parsing url paramater: routes.GetRecord: %s",
err)
            return
      }
      name = strings.Trim(name, " ")

      // request record from Cloudant and parse the JSON
      res, err := find_record(name)
      if err != nil {
            fmt.Printf("error reading cloudant json, routes.GetRecord:
%s\n", err)
            w.WriteHeader(500)
            fmt.Fprint(w, `{"error":"unable to read cloudant json
response"}`)
            return
      }
      buf := new(bytes.Buffer)
      buf.ReadFrom(res.Body)
      js := buf.String()
      var response Response
      err = json.Unmarshal([]byte(js), &response)

      // check if a document was found in db
      if len(response.Docs) > 0 {
            // response contains a document with the given name
            str, err := json.Marshal(response.Docs[0])
            if err != nil {
```

```
                  fmt.Printf("error converting documents to json,
routes.GetRecord: %s\n", err)
                  w.WriteHeader(500)
                  fmt.Fprint(w, `{"error":"unable to form json response"}`)
            } else {
                  fmt.Fprint(w, string(str))
            }
      } else {
            // no doc matching the given name was found
            w.WriteHeader(404)
            s := fmt.Sprintf(`{"error":"no Record found with name: ` + name
+ `"}`)
            fmt.Fprint(w, s)
      }
}
```

And in Java, the same request handler is written as follows:

```
 @GET
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    @Produces(MediaType.APPLICATION_JSON)
    public String getRecord(@QueryParam("name") String name) throws
Exception {
        Record r = findRecord(name);
        if ( r == null ) {
            return "{\"error\":\"no record found for name='"+name+"'\"}";
        }
        return r.toString();
    }
```

Where the findRecord function is:

```
public Record findRecord(String name) throws MalformedURLException,
IOException, ParseException {
        String url = db + "/records/_find";
        DefaultHttpClient httpClient = new DefaultHttpClient();
        httpClient.getCredentialsProvider().setCredentials(
                new AuthScope(AuthScope.ANY_HOST, AuthScope.ANY_PORT),
                new UsernamePasswordCredentials(dbUser, dbPass));
        HttpPost postRequest = new HttpPost(url);

        JSONObject obj = new JSONObject();
        JSONObject selector = new JSONObject();
        JSONObject id = new JSONObject();
        id.put("$gt", 0);
        selector.put("_id", id);
        selector.put("name", name);
        obj.put("selector", selector);

        StringEntity input = new StringEntity(obj.toString());
        input.setContentType("application/json");
        postRequest.setEntity(input);
        HttpResponse response = httpClient.execute(postRequest);
        if (response.getStatusLine().getStatusCode() != 200) {
            throw new RuntimeException("Failed : HTTP error code : "
                    + response.getStatusLine().getStatusCode());
```

```
        }
        BufferedReader br = new BufferedReader(
                new
InputStreamReader((response.getEntity().getContent())));
        String output = "", line = "";
        while ((line = br.readLine()) != null) {
            output += line;
        }
        httpClient.getConnectionManager().shutdown();

        JSONParser parser = new JSONParser();
        JSONObject js = (JSONObject)(parser.parse(output));
        JSONArray arr = (JSONArray)js.get("docs");
        if ( arr.isEmpty() ) {
            return null;
        }
        JSONObject o = (JSONObject)arr.get(0);
        Record r = Record.parse(o);
        return r;
    }
```

By looking at these three implementations, we see that the GET request for a Record is nearly the same on each server. Each request handler uses a “name” URL parameter to lookup an existing record in the Cloudant database. If the record is found, it is returned, otherwise an error is returned. The implementation for the Update, Delete, and Create actions as seen in component (2) of the UI are similar. The differences being that the Update, Delete, and Create other actions invoke a POST, DELETE, or PUT request, respectively, on the same `/record` resource, and that the database operations with either update, delete, or create the records. Full implementation of the Basic Operation’s REST API can be found in the included source package with this report, or online at https://github.com/brandonSc/HonoursProject.

Looking next at the Long Operation, we see that the implementations of the bubble-sort algorithm are nearly identical on each of the servers as well.

In Node.Js:

```
app.get('/long-operation', function(req, res) {
          //console.log(factor(largeNumber));
        bubbleSort(array);
```

```
        res.header('Content-Type', 'application/json');
        res.send('{"message":"done"}');
    });

    function bubbleSort(a) {
        for ( i=0; i<a.length; i++ ) {
            for ( j=1; j<a.length-i; j++ ) {
                if ( a[j-1] < a[j] ) {
                    var temp = a[j-1];
                    a[j-1] = a[j];
                    a[j] = temp;
                }
            }
        }
        return a;
    }
}
```

And in Go:

```
// from route/handlers.go file
func LongOperation(w http.ResponseWriter, h *http.Request) {
      longops.BubbleSort()
      fmt.Fprintf(w, `{"message":"done"}`)
}

// from longops/sort.go
func BubbleSort(a []float64) []float64 {
      for i := range a {
            for j := 1; j < len(a)-i; j++ {
                  if a[j-1] < a[j] {
                        temp := a[j-1]
                        a[j-1] = a[j]
                        a[j] = temp
                  }
            }
      }
      return a
}
```

And lastly, the Java version:

```
    @GET
    public String get(@Suspended final AsyncResponse asyncResponse) {
            asyncResponse.register(new CompletionCallback() {
            @Override
            public void onComplete(Throwable throwable) {
                if (throwable == null) {
                    // no throwable - the processing ended successfully
                    // (response already written to the client)
                    numberOfSuccessResponses++;
                } else {
                    numberOfFailures++;
                    lastException = throwable;
```

```
            }
        }
    });

    new Thread(new Runnable() {
        @Override
        public void run() {
            if ( values == null )
                initValues();
            bubbleSort(values);
            asyncResponse.resume(values);
        }
    }).start();
    return "done";
}

public double[] bubbleSort ( double[] values ) {
    for ( int i=0; i<values.length; i++ ) {
        for ( int j=1; j<values.length-i; j++ ) {
            if ( values[j-1] < values[j] ) {
                double temp = values[j-1];
                values[j-1] = values[j];
                values[j] = temp;
            }
        }
    }
    return values;
}
```

Again, we see that the bubble sort algorithm and request handlers are implemented very similarly. The key differences being that in Java, the request is defined explicitly to by asynchronous and run in a new thread. In Go, this behavious is given by default, and in Node.Js, this behaviour is not technically possible.

3.3. Personal Experiences

It is worth noting some personal experiences with the implementation of the three servers. Keep in mind that these are *personal* experiences which may be biased, for example, due to past experiences or skill level. After completing the application in each of the languages, it was clear that it was quickest programming the app in Node.Js to completion. The syntax is simple (although arguably incoherent if you’re new to

Node/JavaScript), and required less time debugging, and of course no time spent compiling. Java also has a very simple and user friendly syntax, and has the advantage of being extremely popular and well supported. However, programming tasks which are often done in modern cloud based applications, such as reading/writing JSON, working with network requests and responses, or request authentication, felt more difficult than was necessary compared to Node.Js and Go. In my experience on this project, the Java version took the longest to develop due to time some extra time spent debugging and compiling. This could in-part be due to my choice in using Maven for package management, Jersey for the web server, or Tomcat for deploying and debugging. Go is an interesting new contender in the software world, and is said to be “as fast as C and intuitive and simple as python”. It’s syntax is coherent, and the process of writing, compiling, and debugging, felt quick. The learning curve can be steep, however, and it can be difficult to fully master the language, especially when delving into advanced features, such as Go’s concurrency constructions. In this project, it required longer than in Node.Js to complete the application in Go, but less than in Java.

3.4. Test Suite Design

Next, we’ll look at the response-time testing procedures. A simple test app is written in JavaScript to make requests to all three of the servers. The code is not modified for any of the response time tests, other than the URL the requests are sent to. The test program basically makes a certain number of requests at a given interval, then measures and averages the response times for each request. For example, to test how the server stands-up against 10 requests per second, we may configure the test

program to send requests at 100 millisecond intervals. In order to gather an accurate average of the data, we may send a total of 1000 requests at this rate. There are two classes of tests: basic testing involves testing the REST API on each of the servers, at 1 request per second, 10 requests per second, and at 50 requests per second. At 1 req./sec, the response time is basically the same as the average response time for a single request, since all three of the servers can answer a request well within 1 second. At 10 req./sec, however, we may see a slight drop in performance, since the server is likely accepting incoming requests requests before it finishes answering previous requests. When we increase to 50 req./sec, the servers are under a very high load, and this is where we'll see a huge performance drop. Under this load, we may see a significant contrast in performance between the three servers. The second class of tests involve the Long Operation. In this class, the frequency requests are sent is reduced, since each request can take about 10 seconds to complete. We measure the average response time at 2 request per second with 50 total requests, as well as at 10 requests per second at 50 total requests. The number of total requests is important here, since if a server does not support concurrent processing, then each request may slow down the next if it is not answered. In this case, the response time for requests may continue to slow down even more for each successive requests, as the total number of unanswered requests continue to pile on. The code for the test suite is really quite simple. The codeis exactly the same for each test on each server, only the global variables `url`, `numTests`, and `interval`. Here's the whole program:

```
var request = require('request');
var sleep = require('sleep');

var numTests = 50; // number of requests to send
var interval = 100; // how often in millis to send request
var url = "http://localhost:3000/record";
```

```
var responseTimes = new Array(numTests);
var numDone = 0;

for ( var i=0; i<numTests; i++ ) {
    runTrial(i);
}

function runTrial(i) {
    setTimeout(function() {
        testGet(i, "test");
    }, (i * interval));
}

function testGet(i, name) {
    var startTime = new Date();
    request(url, function(err, req, res) {
        if ( err ) {
            console.log('error '+err);
            return;
        }
        responseTimes[i] = new Date() - startTime;
        numDone++;
        console.log(i+": "+responseTimes[i]+" - "+res);

        if ( numDone == numTests )
            printStats();
    });
}

function printStats() {
    var total = 0;
    for ( var i=0; i<numTests; i++ ) {
        total += responseTimes[i];
    }
    console.log('\nAvg Resp. Time = '+(total/numTests));
}
```

The program simply sends a request to the given URL, at the provided interval, for the specified number of tests. It retains the response time for each test in an array, and averages the values in the array, after all the requests have been completed.

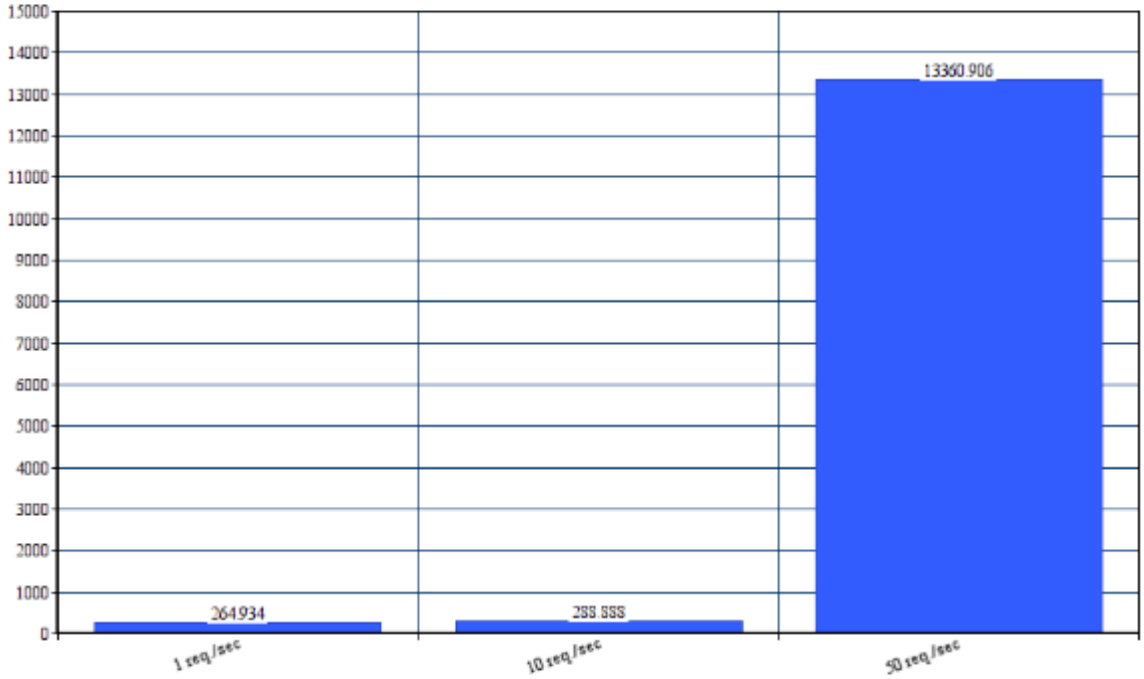
4. Results

Beginning with the basic class of tests, the average response times are analyzed for each of three servers in this section. We first examine the results for the Basic set of operations for Node.Js, Go, then Java, while summarizing the data using bar graphs and tables. Next we examine the results for the Long Operation set of tests for Node.Js, Java, then Go, then compare the data using a table.

4.1. Basic Operations

Begin with the Node.Js version of the server. With Node.Js, we expect a significant decrease in performance as the number of requests per second, since it is single-threaded by nature. The results confirm this expectation, with a massive drop in performance at 50 requests per second.

Figure 4.1 - Basic Results In Node.Js

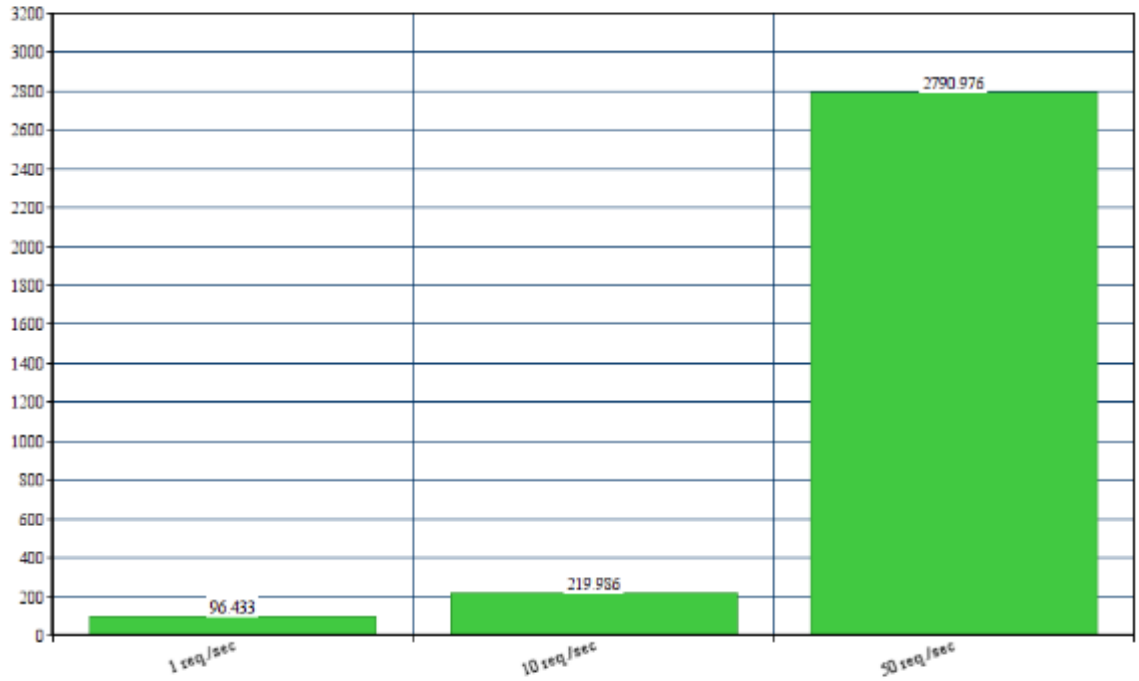


At the 50 req./sec rate, an average response time of 13360.906 milliseconds (or roughly 13.36 seconds) was recorded. This is a massive increase from the 288.888 milliseconds that was recorded from testing at 10 req./sec, or 264.934 milliseconds at 1 req./sec. A significant drop in performance under this extreme load is expected in any server programming language, but with an average response time of 13.36 seconds, Node.Js clearly suffers more than some other frameworks (such as Java or Go) as network traffic increases to high levels.

Next, let's take a look at the response time performance in Go. For the 1 req./sec test, Go performed 168.501 milliseconds faster than Node.Js, with a 96.433 millisecond average response time. On the second test, at 10 req./sec, the Go server

clocked in at 219.986 milliseconds response time. Under extreme load at 50 req./sec, Go only took an average of 2790.976 milliseconds (2.79 seconds).

Figure 4.2 - Basic Results in Go

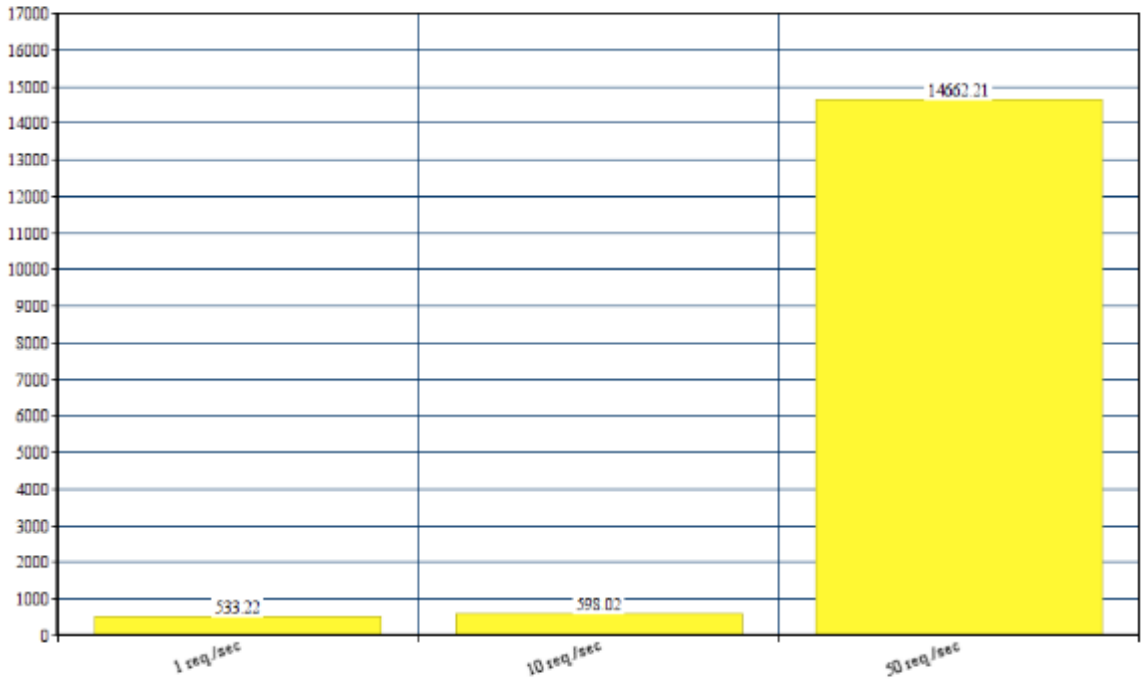


Using the popular `gorilla/mux` package, the Go version of the server utilizes multithreading to significantly improve response time as more requests are stacked on. Each request is handled in a separate goroutine, so requests are processed asynchronously. For these tests, Go performed 63.6% faster at 1 req./sec, 16.97% faster at 10 req./sec, and 79.11% faster at 50 req./sec. In summary, Go is much faster than Node, especially when dealing with high amounts of traffic.

But how does the Java version stack-up against these other two? Our assumption is that the performance of Java will be closer to that of Go, since Java can

also take advantage of multithreading. Here are the results after testing the Java server using our same testing suite.

Figure 4.3 - Basic Results In Java



At 1 request per second, the Java server clocked in at 533.22 milliseconds average response time. At 10 req./sec, Java averaged at 598.02 milliseconds. At 50 req./sec, Java's average response time was 14662.21 milliseconds (or 14.66 seconds). Surprisingly, the response times on the 50 req./sec test are similar to the results from the 50 req./sec in Node.Js, both of which were much slower than in Go. In our particular configurations for the Basic operations, however, each request is not handled in a separate thread. Using Jersey in Java, requests are not *automatically* handled in separate threads as they are when using the `gorilla/mux` package in Go. With Jersey, you must explicitly configure a request handler to run asynchronously, whereas in Go, no configuration was necessary, each request is non-blocking by default. In the

Java server, only the Long Operation was programmed to run asynchronously. Since the Basic tests are synchronous and “block” other incoming requests until the current request has been processed, it is certainly possible that this is the reason that Java performed similarly to Node.Js on the 50 req./sec test. We may see different results for Java when running the Long Operation test, which is asynchronous. In summary: Java performed 50.5% slower than Node.Js and 83.87% slower than Go at 1 req./sec, 51.69% slower than Node.Js and 63.21% slower than Go at 10 req./sec, and, 8.88% slower than Node.Js and 80.96% slower than Go at 50 req./sec.

The results from our Basic group of tests indicate that the Java version is the slowest amongst the three. With a basic REST API connected to a cloud hosted database like Cloudant, and when used with the popular web framework, Jersey, and deployed with the popular application server, Tomcat, Java took longer on average to handle client requests, than both Node.Js and Go. The performance of both Node.Js and Java on the Basic set of tests were similar, but their performance was significantly slower than that of Go. When developing a simple REST API application that will be Cloud hosted, if performance is a concern, or if cost is important (for example when charged by Gigabytes per hour of run-time usage), it may be of benefit to use Go instead of Node.Js or Java. But that’s not to say that Node.Js or Java should not be used in this sort of environment, as there are other significant factors to consider such as simplicity and comfortability with developing and testing in the language, scalability, formal complexity (e.g. Cyclomatic Complexity), or availability and support for third party libraries [9]. Figure 4.4 shows a table which summarizes the results on the three servers for each test.

Figure 4.4 - Comparison of Basic Results

	1 req./sec	10 req./sec	50 req./sec
Node.Js	264.934 ms	288.888 ms	13,360.906 ms
Go	96.433 ms	219.986 ms	2,790.976 ms
Java	533.22 ms	598.02 ms	14,662.21 ms

4.2. Long Operations

We'll next look at the performance of these three servers on the Long Operation test, which is designed to test how quickly each server can react to incoming client requests, while it is already processing some long running task. An inefficient bubble-sort algorithm is chosen as an arbitrary long running task. The bubble-sort is executed on the exact same array of 1000 floating point numbers each time. Although the algorithm has no actual use, it is intended to simulate what might happen if a server is used to run a task that takes a similar amount of time, such as compiling code or deploying an application.

Beginning with the Long Operation test in Node.Js, requests are sent at an interval of 2 requests per second (average load), and then at 10 requests per second (high load). After calling sending 50 requests to the Long Operation task at an interval of 2 requests per second, Node.Js responded at an average of 88,831.08 millisecond (or 88.8 seconds). Clearly Node.Js's performance drops significantly as additional requests are added. Since it only takes about 4 seconds to answer a single request to the Long Operation, the fact that the average response time in this test is 88.8 seconds

indicates that each request “blocks” other incoming requests until it has been completed, which is as expected, given the single threaded nature of Node.Js. This evident by looking at the logs from this test, where the first request sent only took 3,961 milliseconds, and the last request took 172,735 milliseconds (nearly 3 minutes!), since the last request had to wait for many preceding requests to be answered first. On the 10 request per second test, after sending 50 consecutive requests, Node.Js performance dropped to 101,163.96 milliseconds (1.68 minutes) average response time, due to the same issues from running on a single thread. The last request took 3.31 minutes to complete because of the high number of preceding requests which needed to be processed by the server first.

Next up for the Long Operation test is Go. At 2 requests per second for 50 requests, Go clocked in with an average response time of 16,944.58 milliseconds (16.9 seconds). This number is significantly lower than the 88.8 seconds average response time recorded in Node.Js, indicating that Go benefits greatly from the use of concurrent threads. As consecutive requests are continually added on, however, the average response time for each request does still increase, since the first request in the test was logged at 4,113 milliseconds, and the last request was logged at 29,977 milliseconds. When the interval is increased to 10 requests per second, after a total of 50 requests have been sent, the average response time for Go was 28,299.66 milliseconds (28.29 seconds). Again, the performance does drop as many requests are added, since the first request took only 4,100 milliseconds, and the last request took 50,885 milliseconds.

For the last test, we analyze the performance of the Long Operation task in Java. Recall that Java's Long Operation task is programmed to be asynchronous and non-blocking, as is the case Go. At an interval of 2 requests per second for 50 requests, the average response time was 44,078.7 milliseconds (44 seconds). In Java, the first request took 42,881 milliseconds and the last request took 59,602 milliseconds. When the interval is increased to 10 requests per second, after 50 requests, Java's average response time was 36,842.84 milliseconds (36.8 seconds). On this test in Java, the first request took only 6,907 milliseconds, while the last request was logged at 37,566 milliseconds. Interestingly, the performance after running the test a second time with 10 req./sec was faster than the first time it was at 2 req./sec. Also, the difference between the first and last requests on the 10 req./sec test were much greater than the difference between the first and last requests at 2 req./sec. Furthermore, the first request on the 10 req./sec test took only 6.9 seconds, while the 1 req./sec test which was run first took 42 seconds, which is very close to the average response time of 44 seconds on that test. In later versions of Java (we're using Java 8 in this project), optimizations such as Adaptive Optimization and Inline Expansion have been added to improve efficiency when the same procedure are run [12] [13]. These performance features are a benefit for tasks such as the Long Operations test, and their benefits on the performance of these tests are clear in the results. Furthermore, our findings seem to contradict the claim discussed in the Background section, which stated that different concurrency constructs between Java and Node.Js did not affect response time performance.

In summary, Go is still the fastest among the three on the Long Operation group of tests. The tables have turned for Java and Node.Js, however, with Java performing significantly faster than Node.Js on all of these tasks, due to it's availability of features like multithreading and advanced optimization features built into the JVM. The response times from all of the Long Operation tests are contrasted in the table in figure 4.5 below. This concludes the testing of the three servers.

Figure 4.5 - Comparison of Long Operation Results

	2 req./sec	10 req./sec
Node.Js	88.8 sec	172.7 sec
Go	16.9 sec	28.3 sec
Java	44.1 sec	36.8 sec

In Java, the 10.req/sec test was run immediately after the 1 req./sec test, which allowed the JVM to use advanced optimization features.

5. Conclusion

The choice for the top performing server between Java and Node.Js has long been debated amongst the developer community. In our findings, we have shown that both sides have fair arguments: Node.Js is simple and perhaps has less overhead than Java, however, Java benefits from huge performance gains when multithreading or powerful JVM optimizations are appropriate. In the context of Cloud computing, some new dimension are added to the argument. Long running operations like compiling code, deploying applications, running tests, etc, are common in the competitive industry of Cloud Computing and hosting. The competition for dominance in the Cloud Computing market continues between providers such as Amazon Web Services (AWS), Microsoft Azure, Google Cloud, and IBM Bluemix . It is highly important that these platforms provide top software performance to their customers, while also reducing runtime costs on their own servers. Furthermore, for developers and businesses running their Cloud applications on one of these platforms, their choice in server programming environment can have an impact on their costs, especially when they are billed by Gigabytes-per-hour in runtime usage by these providers. Go is a new contender in the scene, and it's impressive performance, syntax, and concurrency features have quickly made it a very appealing choice in the Cloud Computing industry. Large business such as Dropbox and Docker have made the switch to Go, claiming significant performance increases [16]. Programming a server in Go may also lead to a reduction in runtime costs as well.

Our results from building nearly identical servers in Node.Js, Go, and Java show the tradeoffs in performance between these three languages. When running a lightweight REST API that reads/and writes some basic data to a Cloud database like IBM Cloudant, and which does not expect very high web traffic, it is evident that Node.Js and Go beat Java slightly in performance. As traffic increases to much higher levels, however, we quickly see the performance of Node.Js drop, since incoming requests can be “blocked”, due it’s single threaded design. At higher loads, Java and Go can take advantage of multithreading to significantly improve performance. In our particular tests, the Basic set of operations was not asynchronous in Java as it was in Go, however, it is certainly possible to have programmed the Basic operations to run asynchronously in Java, whereas in Node.Js, such a feature is just not possible. With the Long Operation tasks, we saw some interesting results with Java. Advanced optimizations provided by the JVM at runtime meant that Java actually performed faster on the 10 requests per second test than on the 1 request per second test. The JVM is able to take advantage of features such as Adaptive Optimization and Inline Expansion. For Long Operations, Node.Js proved to be the poorest of the three choices, compared to Java and Go. Although Node.Js is a very capable language for Basic tasks with moderate amounts of traffic, it suffered from the same issues at high amounts of traffic and with Long Operations, which is it's single threaded design. Our findings seemed to contradict some claims in other publications, which have stated that Node.Js is faster than Java, and even that Node.Js is still faster, regardless of the fact that Java can be multithreaded. Furthermore, Go recorded the fastest response times for both the Basic and Long operations tasks, and is evidently the best choice to increase server performance, and may even reduce runtime costs, when charged by usage by a Cloud

hosting platform such as IBM Bluemix. For real-world tasks similar to both the Basic and Long Operation tasks, Java has also demonstrated that it is a solid choice, especially when the application is engineered to take advantage of the JVM's optimization features and multithreading. I hope that this report has raised awareness within the developer community about the tradeoffs in Cloud Computing performance between Node.Js, Java, and Go.

6. References

[1] Microsoft, *Microsoft Azure: Cloud Computing Platform & Service* **https://azure.microsoft**.com (accessed 04/12/2016)

[2] IBM, *IBM Bluemix - Next-Generation Cloud App Development* https://ibm.com/**Bluemix** (accessed 04/12/2016)

[3] IBM, *Pricing - IBM Bluemix* https://console.ng.bluemix.net/pricing/ (accessed 04/12/2016)

[4] Jersey *Jersey* https://jersey.java.net/ (accessed 04/12/2016)

[5] Amazon *Amazon Web Services (AWS) - Cloud Computing Services* https://aws.**amazon**.com/ (accessed 04/12/2016)

[6] Info World *Java vs. Node.js: An epic battle for developer mind share* http://www.infoworld.com/article/2883328/java/java-vs-nodejs-an-epic-battle-for-Developer-mindshare.html (accessed 04/12/2016)

[7] IBM Cloudant *Cloudant* https://cloudant.com/ (accessed 04/12/2016)

[8] Gorilla Toolkit *mux - Gorilla, the golang web toolkit* http://www.gorillatoolkit.org/pkg/mux (accessed 04/12/2016)

[9] Wikipedia *Cyclomatic complexity* https://en.wikipedia.org/wiki/Cyclomatic_complexity (accessed 04/12/2016)

[10] Apache Tomcat *Welcome!* http://tomcat.apache.org/ (accessed 04/12/2016)

[11] Stack Overflow *What are the advantages and disadvantages of Go programming language?* http://stackoverflow.com/questions/2198529/what-are-the-advantages-and-disadvantages-of-go-programming-language (accessed 04/12/2016)

[12] Wikipedia *Adaptive optimization* https://en.wikipedia.org/wiki/Adaptive_optimization (accessed 04/12/2016)

[13] Wikipedia *Java performance* https://en.wikipedia.org/wiki/Java_performance (accessed 04/12/2016)

[14] DZone *Performance Comparison Between Node.js and Java EE* https://dzone.com/articles/performance-comparison-between (accessed 04/12/2016)

[15] Google *Google Cloud Computing, Hosting Services & APIs* https://cloud.google.com/ (accessed 04/12/2016)

[16] Hosting Advice *Node.js vs Golang: Battle of the Next-Gen Languages* http://www.hostingadvice.com/blog/nodejs-vs-golang/ (accessed 04/12/2016)

[17] Materialize *Documentation* http://materializecss.com/ (accessed 04/12/2016)

[18] InfoWorld *Java reigns, but Go language spikes in popularity* http://www.infoworld.com/article/2981872/application-development/java-reigns-go-language-spikes-in-popularity.html (accessed 04/12/2016)

[19] Sahara Net *Cloud Computing VS Traditional Computing* http://www.sahara.com/blog/en/cloud-computing-vs-traditional-computing/ (accessed 04/12/2016)

[20] Amazon *Cloud Services Pricing* https://aws.amazon.com/pricing/services/ (accessed 04/12/2016)