Evolution of Node.js with New Runtime Environment

Arjun D. Kori[1] Pratik Bhosale[2] Sachin Vaidya[3]
[1,2,3]Department of Master of Computer Application
[1,2,3]ACPCOE, University of Mumbai, Navi Mumbai, Maharastra, India

Abstract— One of the new Web Technology Node has its own quality Unlike JAVA or PHP, there is no separation between the web server and code, nor do we have to setup different configuration files to get the work we want. With Node.js, We can create our own web server, deliver content , customize and modify it. Very important part t is that all this can do at the code level by writing the lines of code. Here I am going to demonstrates how Node.js Runtime Environment is faster and more sophisticated as compare to other runtime environment.

Key words: npm, parse. com, json

I. Introduction

Requirement of new web runtime is increases day by day. The basic important of new runtime is to minimize the dependency to other server(environment).The Old Traditional environment takes the extra disk Space, Cost and Lots of Configuration .I have work on Different web technologies like Servlet, JSP,PHP etc. Those technologies are fully dependent on different server's ,here client send's request and server process those request ,after processing it sends response back result to the client request. There are large no of server's are present like apache, glass fish, Jboss server .These servers have mainly different syntax for both front and back side.

Node.js, as it is called to distinguish it from other "nodes" - is an event-driven I/O framework for the V8 Chrome engine. Node allows JavaScript to be executed on the server side, and it uses the wicked fast V8 JS engine which was developed by Google for the Chrome browser.

Here i have mostly compared Node with Apache. Node.js is a new runtime environment and, comparing to other scripting languages like Apache, has lacking support. With Node we easily create our http server which is not rely on Apache server like PHP or JSP does.

The basic philosophy of node.js is:

Non-blocking I/O - every I/O call must take a callback, whether it is to retrieve information from disk, network or another process.

Built-in support for the most important protocols (HTTP, DNS, TLS)

Low-level. Do not remove functionality present at the POSIX layer. e.g, support half-closed TCP connections.

Stream everything;In Node.js never force the buffering of web data.

Node.js is different from client-side Javascript in that it removes certain things, like adds support for evented I/O and DOM manipulation, processes, HTTP, SSL, streams, String and C/C++ addons and buffer processing.

The main idea of Node.js: Event-driven I/O to remain lightweight, use non-blocking, and efficient in the face of data-intensive real-time applications that run across distributed devices.

Node really shines is in scalable network applications, building fast, as it's capable of handling a huge number of simultaneous connections with high throughput, which equates to high scalability.

How node.js works under-the-hood is very interesting. Compared to old traditional web-serving techniques where each connection or request spawns a new small thread, taking up system RAM and eventually maxing-out at the amount of RAM available, Node.js work on a single-thread, by using non-blocking I/O calls, allows it to support lakhs of thousands of concurrent connections.

II. Background

"Node.js is a platform build 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.

A. Need of scalable network applications?

Network scalability means a fast network server that can handle a large amount of user requests to our server. This is important because the real-time data's other than overloading the CPU with the frequent requests towards the data provider will also be accessed by a large number of visitors. Sometimes the web application might be under different kind of web attacks such as distributed attacks which opens a many false http connections. If server is un-scalable it will not be able to support those kinds of attacks and consequently will not serve to our requests on time and server will crash or die.

B. What should we understand with event-driven programming?

Event-driven programming or event-based programming is a programming paradigm in which the flow of the execution is determined by events. In such kind of programming the server waits for an event to happen such as a key press, click, hover etc. When the server receives the event it fires the action related to event. In our case we create a loop on the client side which will fire an event every one second.

C. What is a non-blocking input/output model?

Asynchronous I/O, in CPU, is a form of input/output processing that permits other processing to continue before the transmission has finished. This model gives us the opportunity to run other processes while the server is loading our stock data. These is really important in our case because we need to do other processes like buying or selling trades and in the meanwhile the stock data's must be updated regularly. So if we chose a synchronous model we will not give our users the possibility to make these actions because the server would be busy updating our data.

III. RUNTIME ENVIRONMENT

A. *V8 JavaScript Engine*

V8 is a JavaScript engine build in the google’s development center. It is open source technology and written in C++ language. It is used for both client side and server side.

B. *JavaScript applications*

V8 was first designed to increase the performance of the JavaScript execution inside web browser. In order to obtain speed, V8 engine translates JavaScript code into more efficient machine code. It compiles JavaScript code into machine code at execution by implementing a JIT compiler like JavaScript engines such as Spider Monkey or Mozilla Firefox are doing. The difference with V8 is that it doesn’t produce bytecode or any intermediate code.

The aim of this article is to show and to understand working of V8 engine, in order to optimized process for both client and server side applications. If you are worrying that asking yourself “Should I care about JavaScript performance?” then I will answer with a citation, from Daniel Clifford: “It’s not just about making your current application run faster, it’s about enabling things that you have never been able to do in the past”.

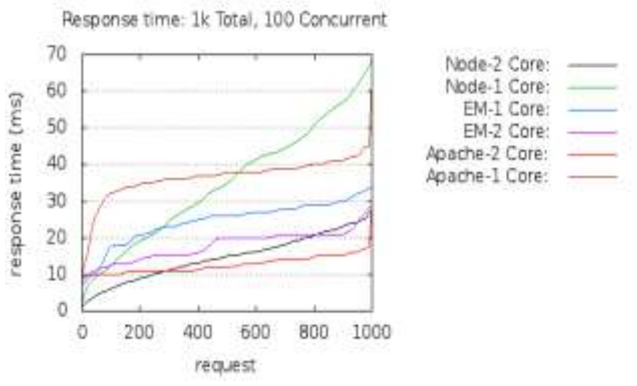
C. *B.Nginx*

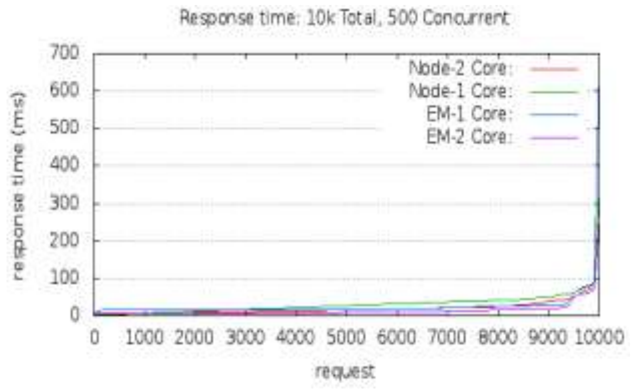
Nginx and Node are similar, since they both have a low number of processes. They both are evented and asynchronous, so their way of handling request and process is fundamentally different from Apache in that perspectt. Apacheis try to make new process, new thread for every requests in every connection. Nginx, like Node.It has little numbers of processes and will take new requests as they arrive, but since it’s asynchronous it can just start to process a request. In other words we can say , if app is taking a long time so it can do another request and then again come back to another one when something has finished happening.

If you are familiar with coding in Node, you know that it has call backs, which something can happen and you do not know when it gets finished you get a call back from the thing that is finishing. Even though it’s not actually make your application faster but it is better for handling a high number of requests because it can handle and process operations in a Un-blocking method. In diffrent words, it does not make web resources wait for a file operation or a network before it moves onto another pass. Therefore, in that respect it’s best than Apache. Apache server can do a many things such as handle a many traffic, but Nginx and Node.js server are a combination because they both have a really similar way of handling responses request and processing HTTP.

D. *Apache*

While quick and efficient at serving lower traffic, Apache underperformed relative to the other two frameworks. Threading may better serve more computationally-intensive web applications, so benchmarking of more complex applications may reveal work load is better served by Apache. But still Apache tomcat displayed less improvement with an one extral processor. Easy to reconfiguring the default or common thread limits could result in improved performance, minding that increasing threads count consumes more memory than a comparable number of events.





IV. ANALYSIS

A. *Performance Tests To Show How Nodes Server Behaves As Compared To Apache When Serving Very Simple Pages.*

Tests were executed on dual-core Intel T4200 2 GHZ machine with 4 GB RAM running on Ubuntu 10.04 Lucid.

For comparison I have used node.js server 0.1.103 on one side, and Apache server 2.2.14 with prefork MPM and PHP 5.2.10 on the other, hitting them with ApacheBench 2.3 and total of 100,000 request with 1,000 concurrent requests during first test:

```
ab -r -n 100000 -c 1000 <url>
```

and then with total of 1,000,000 requests and 20,000 concurrent requests during the second one:

```
ab -r -n 1000000 -c 20000 <url>
```

Basic "Hello World" node.js server used for testing:

```
http = require('http');
 http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<p>Hello World</p>');
  res.end();
}).listen(3000);
```

and equally basic "Hello World" PHP file for Apache:

```
<?php
echo '<p>Hello World</p>';
?>
```

B. *Test Results*

1) *Total request: 100,000; concurrency level: 1,000*

node.js results:

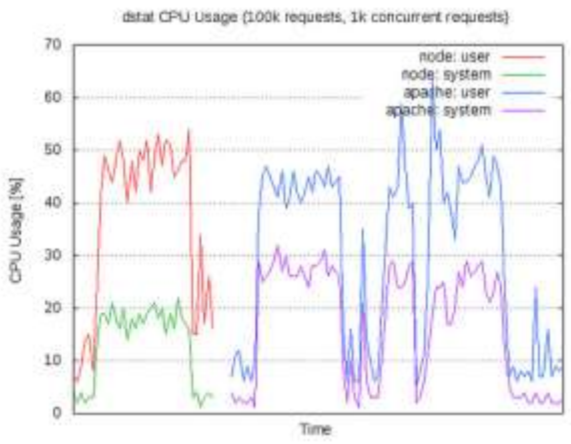
```
Concurrency Level:      1000
Time taken for tests:   21.162 seconds
Complete requests:      100000
Failed requests:        147
```

```
(Connect: 0, Receive: 49, Length: 49, Exceptions: 49)
Write errors:          0
Total transferred:      8096031 bytes
HTML transferred:       1799118 bytes
Requests per second:4725.43 [#/sec] (mean)
Time per request:   211.621 [ms] (mean)
Time per request:   0.212 [ms] (mean, across all concurrent
requests)
Transfer rate:      373.61 [Kbytes/sec] received
Connection Times (ms)
         min  mean[+/-sd] median   max
Connect:   0  135 821.9      0    9003
Processing:1   40 468.5     25   21003
Waiting:   1   30  64.1     25   12505
Total:     2  175 949.1     26   21003
Percentage of the requests served within a certain time (ms)
  50%     26
  66%     33
  75%     36
  80%     39
  90%     55
  95%     94
  98%   3030
  99%   3090
 100%  21003 (longest request)
```

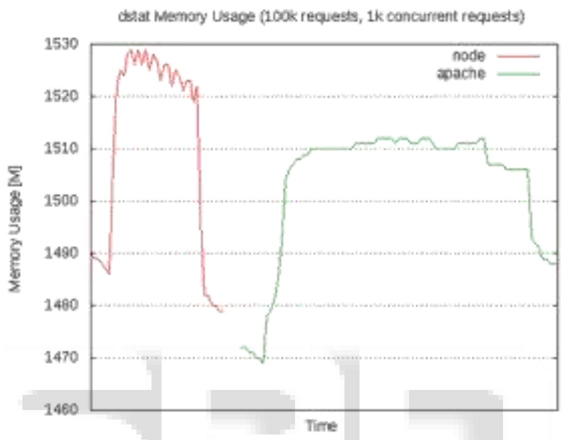
Apache results:

```
Concurrency Level:      1000
Time taken for tests:   121.451 seconds
Complete requests:      100000
Failed requests:        879
   (Connect: 0, Receive: 156, Length: 567, Exceptions: 156)
Write errors:           0
Total transferred:      29338635 bytes
HTML transferred:       1889607 bytes
Requests per second:    823.38 [#/sec] (mean)
Time per request:       1214.510 [ms] (mean)
Time per request:       1.215 [ms] (mean, across all
concurrent requests)
Transfer rate:          235.91 [Kbytes/sec] received
Connection Times (ms)
          min  mean[+/-sd] median   max
Connect:    0   38 321.8     20    9032
Processing: 0  565 5631.0    51  121380
Waiting:    0  262 2324.1    41   52056
Total:     29  603 5641.7    73  121431
Percentage of the requests served within a certain time (ms)
  50%     73
  66%     78
  75%     82
  80%     83
  90%     89
  95%    105
  98%   4251
  99%  13205
 100%  121431 (longest request)
```

To illustrate CPU and memory load changes with following results:



CPU Usage: node.js vs Apache/PHP in ApacheBench test - 100k requests, 1k concurrent requests



2) *Total requests: 1,000,000; concurrency level: 20,000*

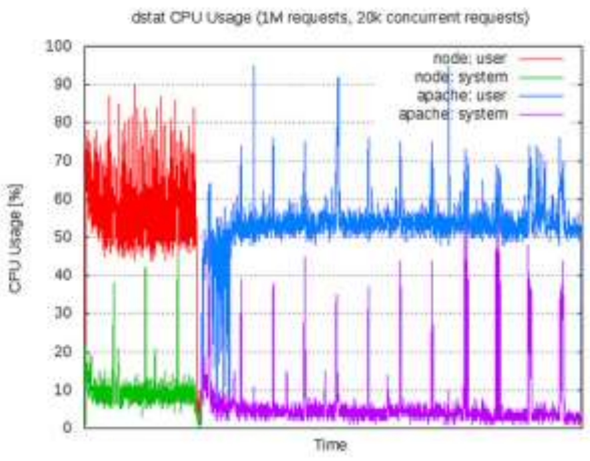
node.js results:

```
Concurrency Level:      20000
Time taken for tests:   1043.076 seconds
Complete requests:      1000000
Failed requests:        25227
   (Connect: 0, Receive: 8409, Length: 8409, Exceptions:
8409)
Write errors:           0
Total transferred:      81265680 bytes
HTML transferred:       18059040 bytes
Requests per second:    958.70 [#/sec] (mean)
Time per request:       20861.529 [ms] (mean)
Time per request:       1.043 [ms] (mean, across all
concurrent requests)
Transfer rate:          76.08 [Kbytes/sec] received
Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0 10201 2391.8  10840   20177
Processing:   595 10455 3239.1  10904   39809
Waiting:        0 8323 2331.0    8728   38740
Total:       1181 20656 4758.5  21795   44333
Percentage of the requests served within a certain time (ms)
  50%  21795
  66%  21929
  75%  22047
  80%  22135
  90%  22667
  95%  24252
  98%  24727
  99%  25942
 100%  44333 (longest request)
```

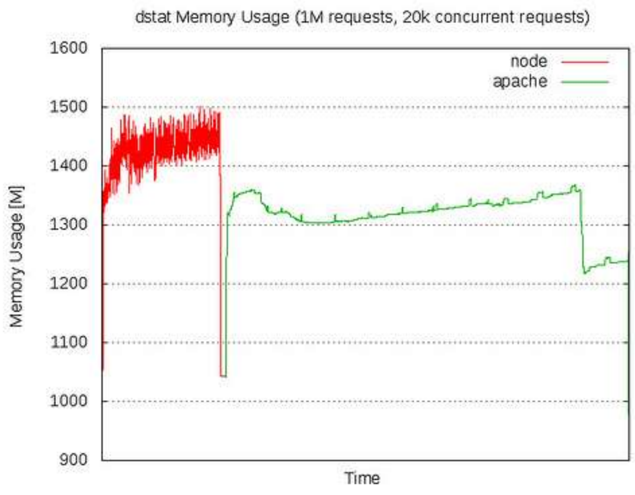
Apache results:

```
Concurrency Level:      20000
Time taken for tests:   3570.753 seconds
Complete requests:      1000000
Failed requests:        2617614
   (Connect: 0, Receive: 848121, Length: 886497,
Exceptions: 882996)
Write errors:           0
Total transferred:      36832520 bytes
HTML transferred:       2372264 bytes
Requests per second:    280.05 [#/sec] (mean)
Time per request:       71415.058 [ms] (mean)
Time per request:       3.571 [ms] (mean, across all
concurrent requests)
Transfer rate:          10.07 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0 4259 14734.0      0   79497
Processing:     4 64979 51442.2  65543  381910
Waiting:        0 2725 16784.2      0  249108
Total:         87 69238 56233.8  68138  426365

Percentage of the requests served within a certain time (ms)
  50%  68138
  66%  80099
  75%  84390
  80%  85475
  90%  91309
  95%  134983
  98%  303390
  99%  333308
 100%  426365 (longest request)
```

Again, CPU and memory usage comparison:



CPU Usage: node.js vs Apache/PHP in ApacheBench test - 1M requests, 20k concurrent requests.



Memory Usage: node.js vs Apache/PHP in ApacheBench test - 1M requests, 20k concurrent requests

V. Conclusions

As above tests show, node is fast. Really fast. Much faster than Apache server -If more requests per second then higher transfer rate with much smaller number of failed requests at the same time.

Node.js is more hungry for memory and system's CPU, but considering its performance is very high.

What needs to be kept in mind though is that everything really depends on what you want to use Node for. Hence Node.js is not something that should replace Apache everywhere because both technologies have different functionality.

When considering using Node.js for a new project absolutely first thing to do is to make sure that Node.js really is a good fit for it in different aspects. Also, depending on project's planned functionality, libraries, I/O connections with other systems and DB, Node performance could be completely different.

Finally main point is – folowing all tests are only point of reference - but to make sure that Node really meets syour own specific requirements, you need to test your own system yourself.

References

[1] Jim R. Wilson "Node.js the Right Way" The Pragmetic Bookself,Dallas,Texas ,California April 2015. (references)

[2] Etah Brown ,Web Development with Node.js & Express" Oreilly India May 2015.

[3] http://www.infoworld.com/ access 5/28/15

[4] http://zgadzaj.com/benchmarking-nodejs-basic-performance-tests-against-apache-php

[5] http://code.google.com/apis/v8/design.html accessed 5/28/15