International Journal of Application or Innovation in Engineering & Management (IJAIEM)
Web Site: www.ijaiem.org Email: editor@ijaiem.org, editorijaiem@gmail.com
Volume 2, Issue 11, November 2013 ISSN 2319 - 4847

ASYNCHRONOUS AND NON-BLOCKING INPUT/OUTPUTUSING BACKBONE.JS AND NODE.JS

Alok Adhao[1], Anuradha Gaikwad[2]

Abstract

The most obvious problem with any form of synchronous replication is the hit on application performance. Every commit requires a round-trip to transfer and acknowledge receipt at the remote site, which in turn reduces single-thread transaction rates to the number of pings per second between sites.

Backbone.js, which is one of the best client side JavaScript architectural frameworks, normally used for single page web application developmentwhere only a single HTML page is sent to the browser at the beginning and every interaction thereafter is handled in asynchronous way usingJavaScript logic that transforms the page, Over a RESTful JSON interface.

Node.js is the server side JavaScript architectural frameworks 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 built on Google's V8 JavaScript Engine and uses asynchronous approach for building applications.

So, using Backbone.js on client side and Node.js on the server side makes it possible to optimize the data transfer time in between client and server due to the asynchronous nature of both the technologies.

Additionally, all the temples are managed by Backbone.js, and not by node.js as server there is no template data transfer from server to client which not only reduce the data to be transferred but also makes the server much computation free for the template rendering and which allows server to speed up the process of data transfer from database to client.

Another benefit in using this combination is both users JavaScript on the server side and the client side, so no need change code if at all you want to put your code form client side to server side.

Keywords:*Asynchronous, Input/output, Backbone.js, Node.js*

1. INTRODUCTION

There are many serial data transfer protocols. The protocols for serial data transfer can be grouped into two types: synchronous and asynchronous.

Synchronous data replication over long distances has the sort of seductive appeal that often characterizes bad ideas because conventional SQL applications interact poorly with synchronous replication over internet.

Problem with any form of synchronous replication is the hit on application performance. Every commit requires a round-trip to transfer and acknowledge receipt at the remote site, which in turn reduces single-thread transaction rates to the number of pings per second between sites.

With the same situation above, NodeJS does this by asynchronously calling the IO operation. This means that while the operation is ongoing on the database, NodeJS proceeds in doing other stuff like cater the next request. When the IO operation completes, NodeJS runs the handler that was set to handle the data when the operation completes.

To easily understand this process, think of it as this: *"I (NodeJS) will come back to you when you (IO operation) are done. For the meantime, I'll be doing something else."*

Backbone.js enforces that communication to the server should be done entirely through a RESTful API. The web is currently trending such that all data/content will be exposed through an API.

2. Backbone.js

Building single-page web apps or complicated user interfaces will get extremely difficult by simply using jQuery. The problem is standard JavaScript libraries are great at what they do - and without realizing it you can build an entire application without any formal structure. You will with ease turn your application into a nested pile of jQuery callbacks,

all tied to concrete DOM elements.

Backbone.js enforces that communication to the server should be done entirely through a RESTful API. The web is currently trending such that all data/content will be exposed through an API.Backbone is an incredibly small library for the amount of functionality and structure it gives you. It is essentially MVC for the client and allows you to make your code modular.

Event-driven communication,it's easy to create small and slick web applications with frameworks like jQuery. When a project grows, however, the jQuery declarations and callbacks get more and more complex and are distributed all over the place. The code becomes more and more cluttered and hard to read.

Backbone.js alleviates this by providing an event-driven communication between views and models (and other elements which we ignore for now for the sake of simplicity). You can attach event listeners to any attribute of a model, which gives you much nuanced control over what you change in the view

The backbone.js events build on top of regular DOM events, which makes the mechanism very versatile and extensible. With one line of code.

Syncing with a back-endthe models in Backbone.js can be easily tied to a back-end. Out-of-the box the framework provides excellent support for RESTful APIs in that models can map to a RESTful endpoint. If the API is designed correctly, backbone is already configured to access these directly for read, write, and delete operations (via GET, POST, and DELETE).

If a backend other than a RESTful API is used, backbone.js is still flexible enough to accommodate for that.

The application uses REST calls to retrieve data from the server.

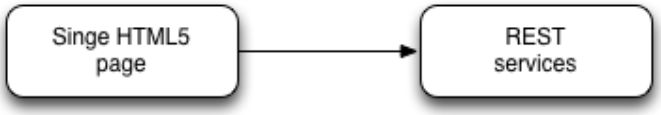
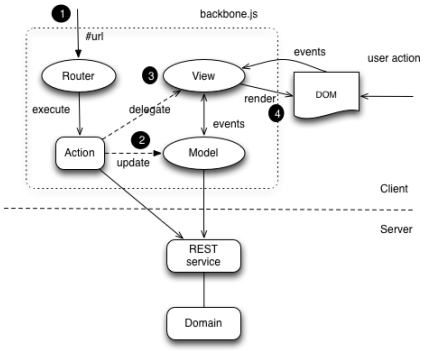


Figure 1 Single page application

Client-side MVC Support, because this is a moderately complex example, which involves multiple views and different types of data, we will use a client-side MVC framework to structure the application, which provides amongst others:

- routing support within the single page application;
- event-driven interaction between views and data;
- Simplified CRUD invocations on RESTful services.

In this application we use the client-side MVC framework "backbone.js".



Modularity, In order to provide good separation of concerns, we split the JavaScript code into modules. Ensuring that all the modules of the application are loaded properly at runtime becomes a complex task, as the application size increases.

Templating, Instead of manipulating the DOM directly, and mixing up HTML with the JavaScript code, we create HTML markup fragments separately as templates which are applied when the application views are rendered.

3. Asynchronous/Non-Blocking Input/output

Now that you have a refreshed and updated idea of what JavaScript programming is really like, it's time to get into the core concept that makes Node.js what it is: that of non-blocking IO and asynchronous programming. It carries with it some huge advantages and benefits, which you shall soon see, but it also brings some complications and challenges with it.

3.1 The Old Way of Doing Things

In the olden days (2008 or so), when you sat down to write an application and needed to load in a file, you would write something like the following (let's assume you're using something vaguely PHP for example purposes):

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

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

// do something with those contents
```

If you were to analyze the execution of this script, you would find that it spends a vast majority of its time doing nothing at all. Indeed, most of the clock time taken by this script is spent waiting for the computer's file system to do its job and return the file contents you requested.

Let me generalize things a step further and state that for most IO-based applications—those that frequently connect to databases, communicate with external servers, or read and write files—your scripts will spend a majority of their time sitting around waiting.

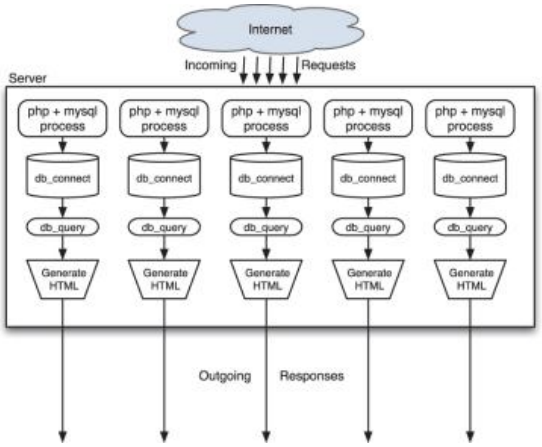


Figure 3.1. Traditional blocking IO web servers

The way your server computers process multiple requests at the same time is by running many of these scripts in parallel. Modern computer operating systems are great at multitasking, so you can easily switch out processes that are blocked and let other processes have access to the CPU. Some environments take things a step further and use threads instead of processes.

The problem is that for each of these processes or threads is that there is some amount of overhead. For heavier implementations using Apache and PHP, I have seen up to 10–15MB per-process memory overhead in the past—never mind the resources and time consumed by the operating system switching that context in and out constantly. That's not even 100 simultaneously executing servers per gigabyte of RAM! Threaded solutions and those using more lightweight HTTP servers do, of course, have better results, but you still end up in a situation in which the computer spends most of its time waiting around for blocked processes to get their results, and you risk running out of capacity to handle incoming requests.

It would be nice if there were some way to make better use of all the available CPU computing power and available memory so as not to waste so much. This is where Node.js shines.

4.Node.js

4.1 Node.js is an event-driven, server-side JavaScript environment. Node runs JavaScript using the V8 engine developed by Google for use in their Chrome web browser. Leveraging V8 allows Node to provide a server-side runtime environment that compiles and executes JavaScript at lightning speeds. The major speed increase is due to the fact that V8 compiles JavaScript into native machine code, instead of interpreting it or executing it as byte code.

Node.js is functionally similar to the PHP + Apache

Though JavaScript has traditionally been relegated to menial tasks in the web browser, it's actually a fully-functional programming language, capable of anything that more traditional languages like C++. Ruby, or Java, are. Furthermore, JavaScript has the advantage of an excellent event model, ideal for asynchronous programming. JavaScript is also a ubiquitous language, well known by millions of developers. This lowers the learning curve of Node.js, since most devs won't have to learn a new language to start building Node.js apps.

4.2 Asynchronous Programming, In addition to lightning fast JavaScript execution, the real magic behind Node.js is something called the Event Loop. To scale to large volumes of clients, all I/O intensive operations in Node.js are performed asynchronously.

The traditional threaded approach to asynchronous code is cumbersome and creates a non-trivial memory footprint for large numbers of clients. To avoid this inefficiency, as well as the known difficulty of programming threaded applications, Node.js maintains an event loop which manages all asynchronous operations for you.

When a Node application needs to perform a blocking operation (I/O operations, heavy computation, etc.) it sends an asynchronous task to the event loop, along with a callback function, and then continues to execute the rest of its program.

The event loop keeps track of the asynchronous operation, and executes the given callback when it completes, returning its results to the application. This allows you to manage a large number of operations, such as client connections or computations, letting the event loop efficiently managing the thread pool and optimize task execution. Of course, leaving this responsibility to the event loop makes life particularly easy for Node.js developers, who can then focus on their application functionality.

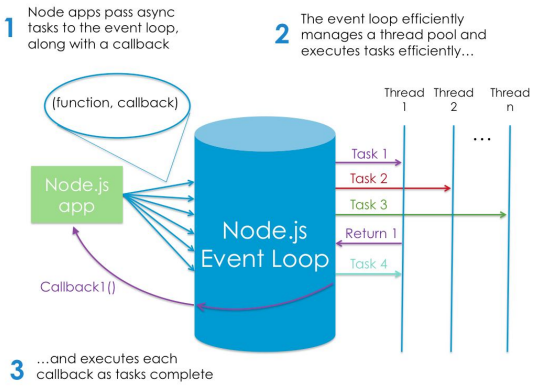


Figure: Node.js Event Loop Lifecycle

This capability to simplify asynchronous programming is what makes Node.js such a powerful tool for developers. With Node.js, you can build complex applications that can scale to millions of client connections because the application handling client requests is passing off all of the time-intensive work of managing I/O and computation to the event loop.

4.3 Asynchronous I/O, or non-blocking I/O, is a form of input/output processing that permits other processing to continue before the transmission has finished, in node.js it is like follows

What this means is, if a process wants to do a read() or write(), in a synchronous call, the process would have to wait until the hardware finishes the physical I/O so that it can be informed of the success/failure of the I/O operation.

On asynchronous mode, once the process issues a read/write I/O asynchronously, the system calls is returned immediately once the I/O has been passed down to the hardware or queued in the OS/VM. Thus the execution of the process isn't blocked (hence why it's called non-blocking I/O) since it doesn't need to wait for the result from the system call, it will receive the result later.

Following is the similar example as we have taken in **Asynchronous/Non-Blocking Input/output**part in php language now let us have an example in node.js

Code

```
console.log ("Before the read file function ");

fs.readFile('info.txt', function (err, data)
{
if (err) throw err;

console.log ("Data is" +data);
});

console.log ("After the read file function");
```

Output

```
Before the read file function   // This is the normal execution of the console log.

//    No waiting in between

After the read file function   //this log did not wait until the file read is completed and executed.

//   After some delay

Data is (the data of the file)   // this log executes once the read file process is completed.
```

The output shows us the Asynchronous nature of node.js the area of the non-blocking Input/output.

4.4 Within a performance test in comparison with JAVA we can see that the performance of node.js is much better even than the strict object oriented language like JAVA which is demonstrated below:

The same performance tests against both a Node.js application and a Java servlet application. Both applications used the same backend as our original Node.js application: CouchDB.CouchBase Single Server version 1.1.3. I created 10.000 sample documents of 4KB each with random text was used, The test machine was a iMac with 2.4 GHZ Intel Core 2 Duo, 4 GB RAM, and Mac OS X.

Apache JMeter running on a separate machine as a test driver. The JMeter scripts fetched random documents from each application at various levels of concurrency.

Java

The following Java code is a servlet that fetches a document from CouchDB by id and forwards the data as a JSON object.

```
packagecom.shinetech.couchDB;

importjava.io.IOException;
importjava.io.PrintWriter;

importjavax.servlet.http.HttpServlet;
importjavax.servlet.http.HttpServletRequest;
importjavax.servlet.http.HttpServletResponse;

import org.apache.log4j.Logger;

import com.fourspaces.couchdb.Database;
import com.fourspaces.couchdb.Document;
import com.fourspaces.couchdb.Session;

@SuppressWarnings("serial")
```

```
public class MyServlet extends HttpServlet {
  Logger logger = Logger.getLogger(this.getClass());
  Session s = new Session("localhost",5984);
  Database db = s.getDatabase("testdb");

  public void doGet(HttpServletRequestreq, HttpServletResponse res)
    throws IOException {
    String id = req.getPathInfo().substring(1);
PrintWriter out = res.getWriter();
    Document doc = db.getDocument(id);
if (doc==null){
res.setContentType("text/plain");
out.println("Error: no document with id " + id +" found.");
    } else {
res.setContentType("application/json");
out.println(doc.getJSONObject());
    }
out.close();
  }
}
```

theJMeter tests against this servlet at various levels of concurrency. The following table shows the number of concurrent requests, the average response time, and the requests that were served per second.

Concurrent Requests	Average Response time (ms)	Requests/second
10	23	422
50	119	416
100	243	408
150	363	411

Figure 4.4.1 showing response

What can be seen is that the response time deteriorates as the number of concurrent requests increases. The response time was 23 ms on average at 10 concurrent requests, and 243 ms on average at 100 concurrent requests.

Node.js

The Node.js application ran on Node.js 0.10.20 using the Cradle CouchDB driver version 0.57. The caching was turned off for the driver to create equal conditions.

The following shows the Node.js program that delivers the same JSON document from CouchDB for a given ID:

```
var http = require ('http'),
url = require('url'),
cradle = require('cradle'),
  c = new(cradle.Connection)(
 '127.0.0.1',5984,{cache: false, raw: false}),
db = c.database('testdb'),
port=8888;
process.on('uncaughtException', function (err) {
console.log('Caught exception: ' + err);
});
http.createServer(function(req,res) {
var id = url.parse(req.url).pathname.substring(1);
db.get(id,function(err, doc) {
if (err) {
      console.log('Error'+err.message);
res.writeHead(500,{'Content-Type': 'text/plain'});
res.write('Error' + err.message);
res.end();
    } else {
res.writeHead(200,{'Content-Type': 'application/json'});
```

```
res.write(JSON.stringify(doc));
res.end();
   }
  });
}).listen (port);
```

The numbers for Node.js system were as follows:

Concurrent Requests	Average Response time (ms)	Requests/second
10	19	509
50	109	453
100	196	507
150	294	506

Figure 4.4.2 showing response

As before the average response time has a linear correlation to the number of concurrent requests, keeping the requests that can be served per second pretty constant. Node.js is roughly 20% faster, e.g. 509 requests/second vs. 422 requests/second at ten concurrent requests.

Conclusion

The Node.js is 20% faster than the Java EE solution for the problem at hand. An interpreted language as fast as or faster a compiled language on a VM in which years of optimization have gone into. Not bad at all. Both Node.js and Java EE scale beyond what a normal server needs. 400-500 requests per second is quite a lot. Google, the largest website in the world, has about 5 billion requests per day. If you divide that by 24 hours, 60 minutes, and 60 seconds it comes out to 57870 requests/ second. That is the number of requests across all Google domains worldwide, so if you have a website running with 400 requests per second on one machine your website is already pretty big. 1 million requests per day on average means 11.5 requests per second.

So, as I am getting a 20% better response using node.js which is the recently developed technology(which is evolving day by day) with compare with the pretty best language like JAVA then I think this is the most amazing thing to use on a server with a heavy load for the fastest Input/Output.

References

[1] Hands-on Node.js by Pedro Teixeria
[2] https://www.udemy.com/blog/learn-node-js/
[3] http://www.sqa.org.uk/e-learning/NetTechDC01BCD/page_39.htm
[4] http://scale-out-blog.blogspot.in/2012/08/is-synchronous-data-replication-over.html
[5] http://en.wikipedia.org/wiki/Data_transmission#Asynchronous_and_synchronous_data_transmission
[6] http://www.informit.com/articles/article.aspx?p=2102373
[7] http://blog.shinetech.com/2013/10/22/performance-comparison-between-node-js-and-java-ee/

AUTHORS

Alok Adhao received the B.Edegree in Computer Science and Engineering from Guru Nanak Institute of Engineering & Technologyin 2013. After that working In an Information technology industry as Web Application developer.

Anuradha Gaikwad received the B.E degree in Computer Science and Engineering from Guru Nanak Institute of Engineering & Technologyin 2013. After that working as an Assistant Professor in Polytechnic college.