Node.js: Event-driven Concurrency for Web Applications

Ganesh Iyer,

BTech.(Computer Engineering), SVNIT

1. Introduction

1.1 Introduction to Nodejs

Node(also known as Node.js, Nodejs, Node JS)[1] is an event driven I/O framework for the V8 JavaScript engine. It is intended for writing scalable network programs such as web servers. Programs are written in JavaScript, using event-driven, Asynchronous I/O to minimize overhead and maximize scalability. It's based on Google's runtime implementation — the aptly named "V8" engine. But, whereas V8 supports mainly JavaScript in the browser (most notably, Google Chrome), Node aims to support long-running server processes consumption.

Unlike other modern environments, a Node process doesn't rely on multithreading to support concurrent execution; it is based on an asynchronous, event-driven I/O model. Its event-driven, non-blocking I/O model makes it lightweight and efficient, ideal for data-intensive real-time applications that run across distributed servers.

1.2 Background

The Internet is continually evolving and presents a number of challenges to web application designers. Large internet services must deal with concurrency at an unprecedented scale. High traffic demands services to better handle concurrent sessions. As the need for Internet services grows, new system design techniques must be used to manage this load.

There are two prevalent strategies[2] for handling concurrency in web servers: *threads* and *events*. Web applications implement threading by assigning each incoming request to a separate thread of execution. For example Apache uses the approach in its multi processing module worker MPM[6]. Contrastingly, event-based systems, utilize a single thread to process events,

such as incoming request, from a queue. Each approach has some inherent advantages and disadvantages.

Building highly concurrent systems is inherently difficult. Structuring code to achieve high throughput is not well-supported by existing programming models. While threads area commonly used device for expressing concurrency, the high resource usage and scalability limits of many thread implementations has led developers to prefer an event-driven approach[2,5].

Node was created by Ryan Dahl in 2009, and its growth is sponsored by Joyent. The original goal of node was to create the ability to make websites with push capabilities as seen in several web applications(Gmail etc.) Node is influenced by systems like Ruby's *Event Machine* and Python's *Twisted*. Node takes the event model further by presenting the event loop as a language construct instead of as a library.

1.3 Motivation

Web servers must be able to handle multiple users at the same time. The traditional approach, using one thread per request, for example: Apache uses this approach in its multi processing module *Worker MPM,* has some drawbacks. Regardless of how well the threaded server is crafted as the number of threads in a system grows, the operating system overhead(scheduling and aggregate memory footprint) increases. This approach limits the number of clients that can simultaneously connect to the server. The *C10K problem*(Concurrent 10,000 connections)[7], which refers to the problem of optimizing web server software to handle a large number of clients at the same time, is most notable.

Event-driven systems tend to be robust to load, with little degradation in throughput as offered load increases beyond that which the system can deliver. The throughput exceeds that of their threaded counterparts, but more importantly does not degrade with increased concurrency[2].

Concurrency is explicit in the event driven approach, programmers can make use of application specific knowledge to reorder event processing for prioritization or efficiency reasons

Node has a single execution thread with no waiting on I/O or context witching. Instead, I/O calls set up request handling functions that work with the event loop to dispatch events when some things becomes available. Program execution is expected to quickly return to the event loop for dispatching the next immediately run-able task. JavaScript is an excellent fit for this approach because it supports event callbacks. For example, when a browser completely loads a document, a user clicks a button, or an AJAX request is fulfilled, an event triggers a callback. JavaScript's functional nature makes it extremely easy to create anonymous function objects that can be registered as event handlers. Also having the same language on both the server-side and client-side has immense benefits.

1.4 Outline

In Chapter 1, we provide an overview of Nodejs. In Chapter 2, a survey of the two most prevalent strategies for handling concurrency in modern systems: *threads* and *events,* is given. We first explore the thread based server architecture, highlighting the issues in scalability, such as the C10K problem of this approach, in large scale applications. We then provide an overview of the event-driven server architecture.

In Chapter 3, we discuss the JavaScript language, the features that make it suitable for event-drive I/O. An overview of Node's architecture is provided. The event-based concurrency model used in Node is explored. We also discuss the essential middleware that are most commonly used in a Node application. Furthermore, the advantages of using Node and server-side JavaScript are listed and later the challenges that Node faces are highlighted.

Finally, we conclude the report by showing that the event driven, non-blocking I/O Model is efficient for building scalable web applications.

2. Concurrency in Web Servers

There are traditionally two major server architectures, based on: *threads* and *events*. In this chapter we introduce and evaluate these architectures from a scalability point of view.

2.1 Thread Based Server Architecture

The thread-based approach basically associates each incoming connection with a separate thread. In this way, synchronous blocking I/O is the natural way of dealing with I/O. It is a common approach that is well supported by many programming languages. it also leads to a straight forward programming model, because all tasks necessary for request handling can be coded sequentially. Moreover, it provides a simple abstraction by isolating requests and hiding concurrency. Real concurrency is achieved by employing multiple threads/processes at the same time.

The popular Apache web server provides a robust multi-processing module that implements a hybrid multi-process multi-threaded server. By using threads to serve requests, it is able to serve a large number of requests with fewer system resources than a process-based server. However, it retains much of the stability of a process-based server by keeping multiple processes available, each with many threads.

Conceptually, multi-processing and multi-threaded architectures share the same principles: each new connection is handled by a dedicated activity(process or thread).

2.1.1 Multi-Process Architecture

The traditional approach to network servers is the process-per-connection model, using dedicated processes for handling a connection[8]. This model was used in CERN httpd, the first HTTP server. The creation of process is a costly operation and servers often employ a strategy called

preforking. The heavyweight structure of a process limits then maximum number of simultaneous connections. The multi-process architecture provides only limited scalability for concurrent requests.

2.1.2 Multi-Threaded Architecture

With the advent of threading libraries, server architectures have emerged that have replaced the heavyweight processes with more lightweight threads. Effectively, they employ a thread-per-connection model. Multi-threaded approach, although follows the same principles, is different in terms of the lightweight structures of thread. Compared to the memory size of an entire process, a thread consumes limited memory. Also, threads require less resource for creating and termination.

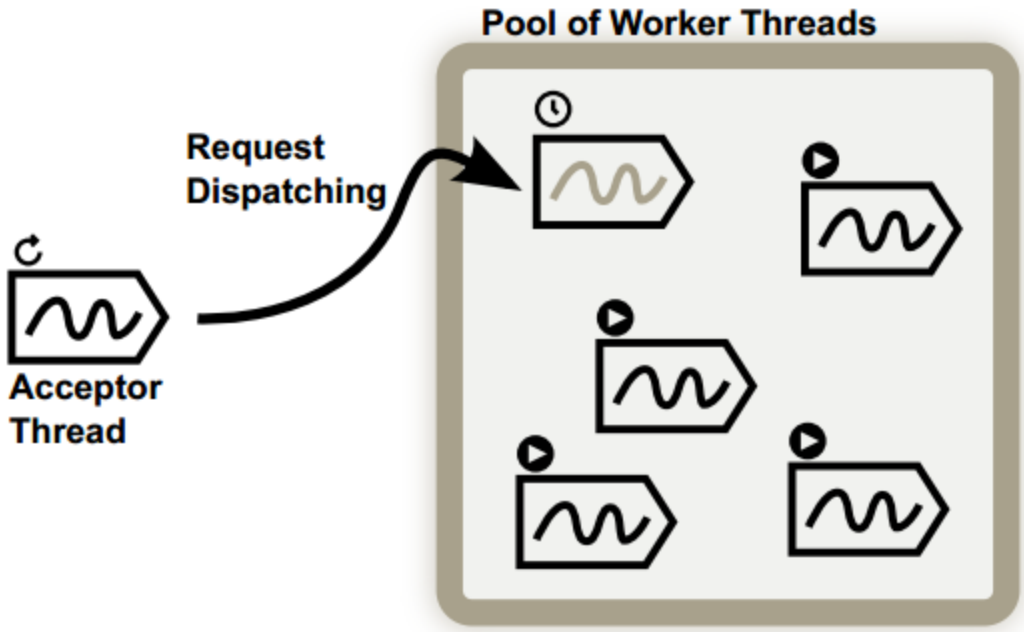

Figure 2.1: *A multi-threaded architecture that makes use of an acceptor thread[4].*

It is common practice to place a single acceptor thread in front of a pool of threads for connection handling[8], as shown in fig 4.2. Thread pools are a common way of bounding the maximum number of threads inside the server. The acceptor blocks for new socket connections, accepts connections and dispatches them to the worker pool and continues. The worker pool provides a set of threads that handle incoming requests. Worker threads are either handling requests or waiting for new requests to process.

2.1.3 Issues of Scalability in Multi-Threaded Architectures

Regardless of how well the threaded server is crafted, as the number of threads in a system grows, operating system overhead(scheduling and aggregate memory footprint) increases, leading to a decrease in the overall performance of the system. There is typically a maximum

number of threads *T'* that a given system can support, beyond which performance degradation occurs[2].

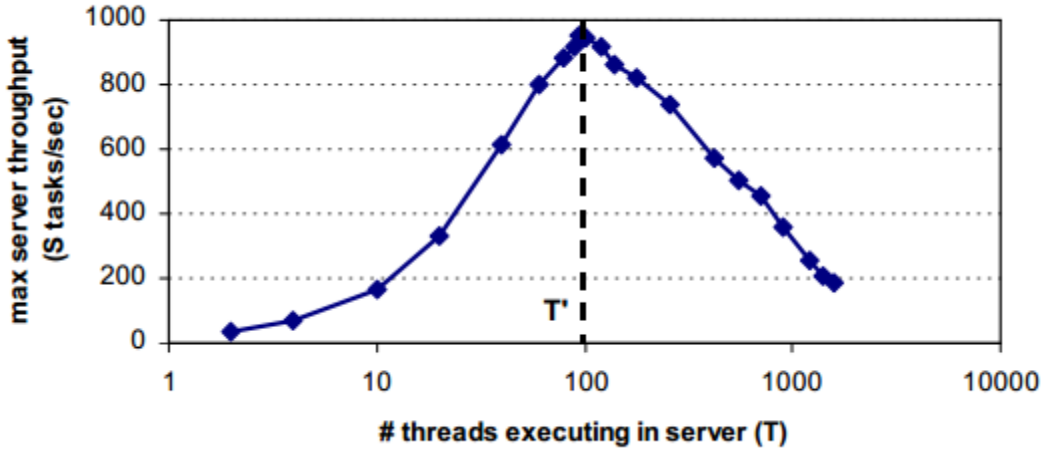

Figure 2.2: *Threaded server throughput degradation: (This benchmark has a very fast client issuing many concurrent150-byte tasks over a single TCP connection to a threaded server with per task latency = 50ms on a 167 MHz Ultra-SPARC running Solaris 5.6). As the number of concurrent T increases, throughput increases until T >= T', after which the throughput of system degrades substantially[2].*

Under heavy load, a multi-threaded server consumes large amounts of memory(due to single thread stack for each connection), and context switching causes considerable losses of CPU time. An indirect penalty thereof is increased chance of CPU cache misses. Reducing the absolute number of threads increases per-thread performance, but limits the overall scalability in terms of maximum simultaneous connections.

2.1.4 C10K Problem

The C10k problem refers to the problem of optimizing web server software to handle a large number of clients at the same time (hence the name C10k - concurrent ten thousand connections) [7].

The problem was published by Kegel in seminal article in 1999, proclaiming that "it is time for web servers to handle ten thousand clients simultaneously". It is observed that the thread based

approach has the disadvantage of using a whole stack frame for each client, which costs memory. This can lead to *out-of-memory* errors. Several asynchronous I/O methods are then surveyed with their effect on scaling for solving this problem.

2.2 Event-Driven Server Architecture

As an alternative to synchronous blocking I/O, the event-driven approach is also common in server architectures. Due to the asynchronous/non-blocking call semantics, other models than the previously outlined thread-per-connection model is needed. A common model is the mapping of a single thread to multiple connections. The thread then handles all occurring events form I/O operations of these connections and requests. As shown in Fig 2.3, new events are queued and the thread executes a so-called event loop - dequeuing events from the queue, processing the event, then taking the next event or waiting for new events to be pushed .

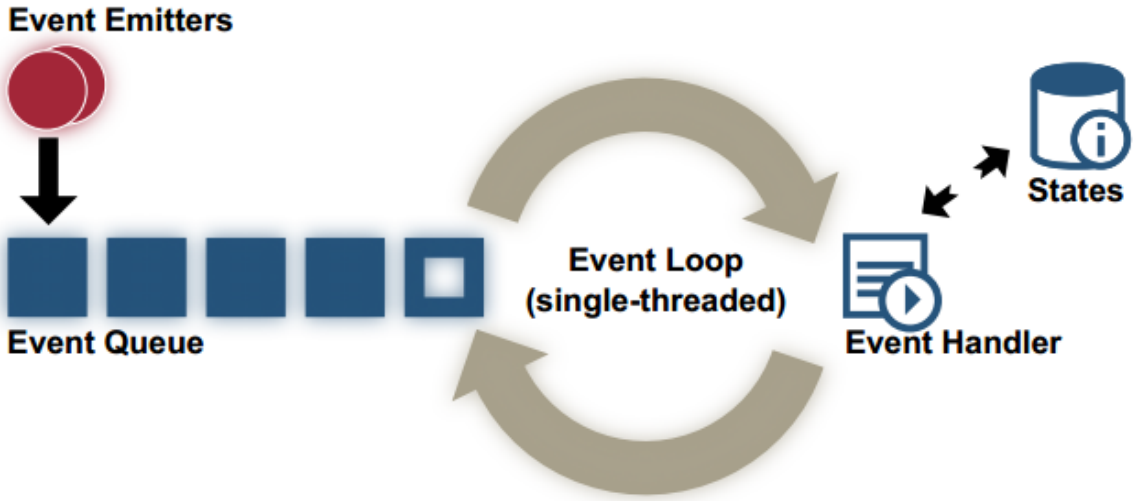

Figure 2.3: *This conceptual model shows the internals of an event-driven architecture. A single threaded event loop consumes events after event from the queue and sequentially executes associated event handler coded[4].*

Processing an event either requires registered event handler code for specific events, or it is based on the execution of a callback associated to the event in advance. The different states of the connections handled by a thread are organized in appropriate data structures - either explicitly using finite state machines or implicitly via closures of callbacks. As a result, the control flow of an application following the event-driven style is somehow inverted. Instead of sequential operations, an event-driven program uses a cascade of asynchronous calls and callbacks that get executed on events.

Having a single thread running an event loop and waiting for I/O notifications has a different impact on scalability than the thread-based approach. Not associating connections and threads does dramatically decrease the number of threads of the server - in an extreme case, down to the single event-looping plus some OS kernel threads for I/O. We thereby get rid of the overhead of excessive context switching[5].

Event driven systems tend to be robust to load, with little degradation in throughput as offered load increases beyond that which the system can deliver[2]. If the handling of events and bundling of task state is efficient, the peak throughput can be high. Figure 2.4 shows the throughput achieved on an event-driven implementation of the network service as a function of the load. The throughput exceeds that of the threaded server, but most importantly does not degrade with increased concurrency.

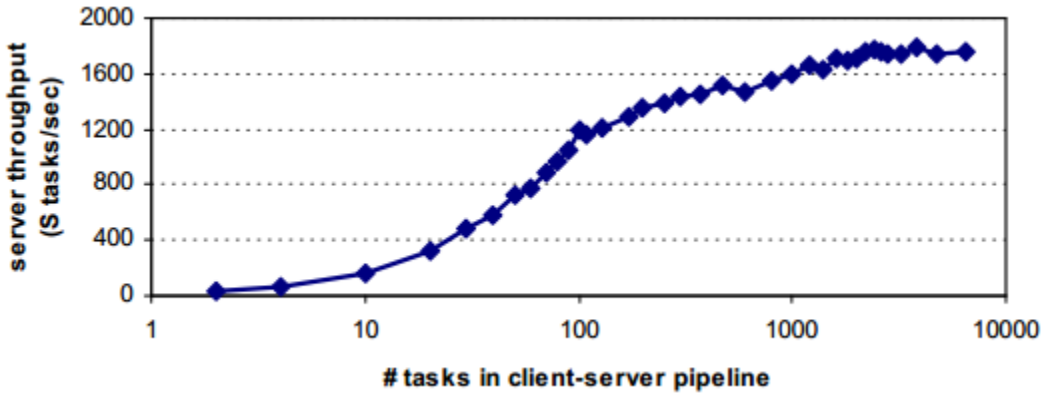

Figure 2.4: *Event-Driven Server throughput:(Using same benchmark setup as of Figure 2.2) This figure shows the event-driven server's throughput as a function of the number of tasks in the pipeline. The throughput fattens in excess of that of the threaded server as the system saturates, and the throughput does not degrade with increased concurrent load[2].*

3. Fundamentals of Nodejs

Node is a platform for building event-driven networking programs, e.g. web servers. It provides the developer with a JavaScript runtime and libraries to write applications. The JavaScript-runtime is V8; the same engine that is used by Google's web browser Chrome. V8 uses dynamic machine code generation to achieve high performance[9]. Listing 3.1 shows a trivial hello world server implemented in Node.

```
1  var http = require("http");
2  var server = http.createServer(function (req, res) {
3      res.writeHead(200, {"Content-Type": "text/plain"});
4      res.end("Hello World\n");
5  });
6  server.listen(1337);
7  console.log("Server running at http://localhost:1337");
```

Listing 3.1: Hello World web server in Node

Unlike other modern environments, a Node process doesn't rely on multithreading to support concurrent execution of business logic; it's based on an asynchronous I/O eventing model. The Node server process can be thought of as a single-threaded daemon that embeds the JavaScript engine to support customization[10]. This is different from most eventing systems for other programming languages, which come in the form of libraries. Node supports the eventing model at the language level.

3.1 JavaScript

JavaScript, constructed by Brendan Eich at Netscape in 1995, was conceived as the scripting language of Netscape's browser *Navigator*. JavaScript has since then become the client-side

scripting language of the web and is one of the most wide spread languages, all modern browsers with a JavaScript-engine and almost all PCs have atleast one browser installed.

Javascript's syntax is inspired by C and java, but the inner working of the language are inspired by scheme[11]. It features prototype-based inheritance model and functions are first class objects. It also features lambdas i.e. anonymous functions[12].

3.1.1 Closures

Functions may exist inside of functions and be returned as "*first class objects*" to some variable outside the function it was first constructed in. When this happens a new *closure* is created. A closure is a data structure containing a function and a referencing environment for the non-local variables for that function[13].

```
1  function create_adder(x){
2      return function(y){
3          return x+y;
4      }
5  }
6  add_two = create_adder(2);
7  add_four = create_adder(4);
8  add_two(7);      //returns 9
9  add_four(10);    //returns 14
```

Listing 3.2: *Closure*

Listing 3.2 shows the function `create_adder` which creates an inner(anonymous function) and returns it. The inner function can be invoked later, when `create_adder` has returned and it will still have access to `x.`

3.1.2 Server-side JavaScript

Netscape introduced an implementation of the language for server-side scripting with Netscape Enterprise Sever, in December 1994. Although, JavaScript is the most popular language on the client side, it isn't dominant on the server side. Since the mid-2000s, there has been a proliferation of server-side Javascript implementations. Rhino, an open source JavaScript engine, managed by Mozilla Foundation, is intended to be used in server-side applications. Several other implementations like 10gen, Apache Sling etc exist. Node is one recent notable example of server-side JavaScript.

3.2 Nodejs Architecture

Figure 3.1 provides a high level overview of the components of Node. V8 is the JavaScript engine, *libeio* handles an internal thread pool which is used to make synchronous POSIX-calls asynchronous to the event loop. Asynchronous file system is not used, instead blocking I/O is performed in its own thread, so it is non blocking to the event loop in Node. *libev* is the event loop. Node bindings are some thin C++ bindings that expose the API of the underlying components to JavaScript.

Node's standard library exposed operating system features such as handling file system, sockets, processes and timers. The standard library also has functionality for events, buffers and simple DNS and HTTP requests.

Node uses techniques like `kqueue, select, epoll` to get notifications from the operating system when new connections are incoming and then dispatch new events to the programmer to handle with handlers.

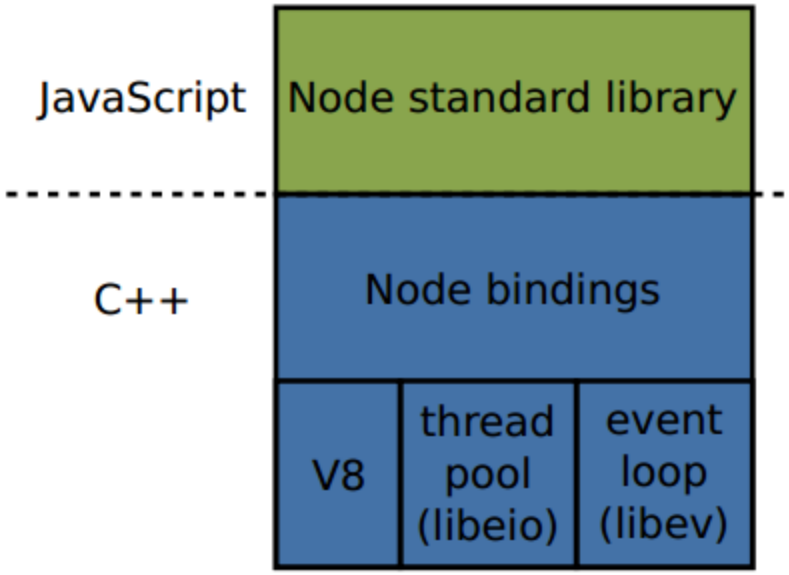

Figure 3.1: *Node Architecture[14]*

3.3 Concurrent Programming With Nodejs

A Node server process, runs single threaded, yet can serve many clients concurrently. As shown in Fig 3.2, There is an implicit main loop around the code. No actual I/O, let alone business-logic processing, happens in the loop body. I/O related events trigger the actual processing, such as a connection being made or bytes being sent or received from a socket, file or external system.

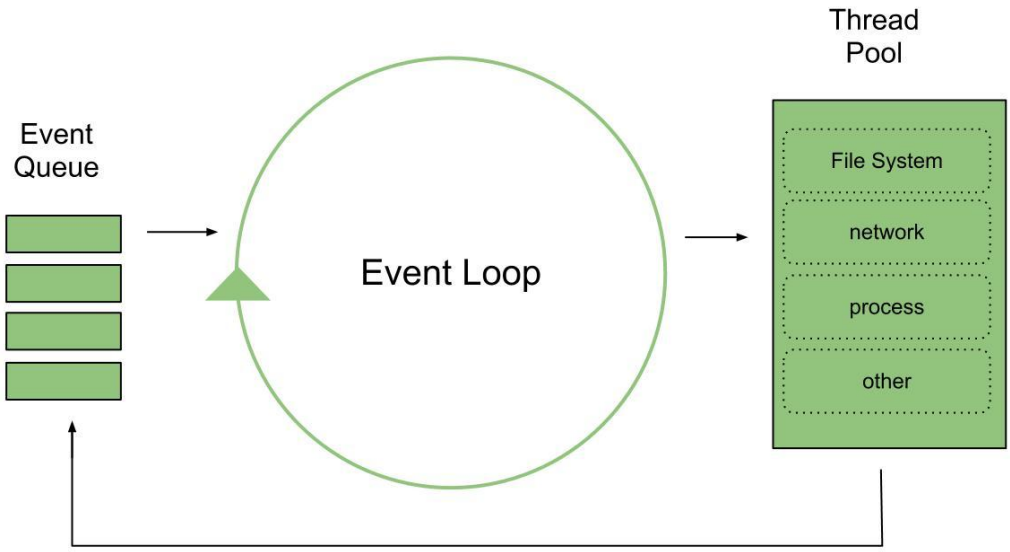

Figure 3.2: *The Node Event loop*

With node developers can easily build a high-performance, asynchronous, event-driven network server with modest resource requirements. The program's main flow is determined by the functions that are explicitly called. These functions never block on I/O, but rather register appropriate handlers. There is no explicit blocking call to invoke the event loop. The event loop concept is so core to Node's behavior that it's hidden in the implementation. In addition, making asynchronous I/O the default forces developers to adopt the asynchronous model from the start. This is one of the main difference between Node and using asynchronous I/O in other programming environments.

```
var sys      = require("sys"),
    http     = require("http"),
    url      = require("url"),
    path     = require("path"),
    fs       = require("fs");

http.createServer(function(request, response){
    var uri = url.parse(request.url).pathname;
    var filename = path.join(process.cwd(), uri);
    path.exists(filename, function(exists){
        if(exists){
            f = fs.createReadStream(filename);
            f.addListener('open', function(){
                response.writeHead(200);
            });
            f.addListener('data', function(chunk){
                response.write(chunk);
                setTimeout(function(){
                    f.resume()
                }, 100);
            });
            f.addListener('error', function(err){
                response.writeHead(500, {"Content-Type":"text/plain"});
                response.write(err + "\n");
                response.end();
            });
            f.addListener('close', function(){
                response.end();
            });
        }
    });
}).listen(1337);
```

***Listing 3.3**: A simple streaming HTTP file server. Chunks of the file are read from disk and sent to the client using HTTP's "chunked" transfer encoding[10].*

Listing 3.3 is a slightly more complex variant of the simplistic HTTP server. It parsed the URI from an HTTP request and maps the URI's path component to a filename on the server. The file is read in small chunks rather than all at once. In certain situations, the function provided for the scenario as callback is invoked. Example situations include when the file system layer is ready to

hand a number of bytes to the application, when the file has been read completely, or when some kind of error occurs.

3.4 Nodejs Ecosystem: Essential Packages

Node is one of the better-known frameworks and environments that support server-side JavaScript development. The community has created a whole ecosystem of libraries for, or compatible with, Node.

3.4.1 npm - A package manager for Nodejs

npm is a package manager for Node that is run through the command line and manages dependencies for an application. It is the predominant package manager for Node. Modules for Node can be downloaded as source and assembled manually for use. npm provides a simpler alternative, it is the standard package manager for Node and greatly simplifies the management of these modules.

3.4.2 Overview of Some Popular Packages

Table 3.1: An overview of some useful Node packages

Cluster	It is an extensible multi-core server manager for Node. It starts up a configurable set of child processes, restarting them in case of a crash, and has extensive logging, command-line control utilities, and statistics
Connect	Connect is a *middleware* framework for node. It includes over 18 essential middleware and a rich selection of 3rd-party middleware. Some essential middleware are ● *logger* - request logger with custom format support ● *basicAuth* - basic http authentication

	• *json* - application/json parser • *session* - session management support.
Express	Express is a web application framework for Node. It provides a skeleton on which to build, handling many of the mundane and ubiquitous aspects of development so as to focus on the functionality unique to the application. Its built on top of the *Connect* middleware.
Socket.IO	Websockets is a relatively new technology that enables bidirectional, real-time communication directly from within a client to a server application, and back again. Socket.IO aims to make realtime applications portable, blurring the differences between the different transport mechanisms.
Underscore	Underscore provides utility functions for Node. It provides a lot of extender JavaScript functionality that are found with other libraries such as jQuery. Underscore provides around 80 functions that support **`map, select, invoke`** - as well as specialized helpers.

3.5 Advantages of Nodejs

3.5.1 JavaScript

JSON(JavaScript Object Notation) is the native representation of objects in JavaScript and one of the most common formats when sending data via AJAX.

A programmer working on both front and back end will not have to switch between two languages. Switching between different tasks or languages is considered a high cost for a programmer and may decrease productivity[15].

Code reuse with the same code running both on the server-side and on the client side, in the browser is possible when the language on both sides is the same.

JavaScript comes with some language features that make it suitable to write event-driven applications. The first is *closures* which automates the saving of state on the stack. JavaScript can make callbacks because it has anonymous functions

3.5.2 Nodejs

One of the advantages of Node over other event-driven frameworks is that Node has a built-in event loop. All available modules and libraries use it. This makes working with multiple parts in an asynchronous program a lot easier[16].

When using Node in the event-driven paradigm is the natural way to do things. As long as the programmer as the programmer does not do anything blocking the performance and scaling will come from the platform.

3.6 Challenges

Node is a young platform, current version being v0.8.14, at the time of this writing. Thus it is constantly evolving. Deployment is an area where there are many alternatives, but none of them are well tested. The fact that it is new and untested in large environments can also be a risk factor for businesses.

In Node there is no loose coupling between the web server and the web application. Loosely coupled architecture has proved itself to be maintainable.

Asynchronous event-driven programming is a new concept to many developers and it can take to get used to and get fully productive with. Debugging becomes difficult, as stack traces no longer represent the control flow for the processing of a particular task.

4. Conclusions

We claim that concurrency is crucial for scalability, which in turn is inherently critical for large-scale architectures. The growth of prevalence of web-based applications thus require both scalable architectures and appropriate concurrent programming.

We provide a detailed overview of the techniques being used for concurrency, namely *threads* and *events*. We evaluate the different approaches by introducing the general architectures and the corresponding patterns in use. Multi-threaded server using a thread-per-connection model are easy to implement and follow a simple strategy. Under heavy load, a multi-threaded web server consumes large amounts of memory(due to a single thread stack for each connection), and constant context switching considerable losses of CPU time. Events, allows programmers to manage concurrency explicitly by structuring code as a single-threaded handler that reacts to events. Event-driven systems tend to be robust to load, with little degradation in throughput as offered load increases.

We introduce Node, a platform built on Google's V8 engine, used for building fast, scalable network applications. It uses JavaScript to write server-side code, we argue that Server-side JavaScript is a logical step, enabling the use of single programming language for all aspects of a Web-based distributed application. We discuss the event-driven, non-blocking I/O model that makes it lightweight and efficient, ideal for data-intensive real-time applications that run across distributed devices. Node is constantly evolving and has a large community that contributes with essential packages and modules.

References

1. Joyent Inc, 'Node's goal is to provide an easy way to build scalable network programs', http://nodejs.org/about/ (2012).
2. M. Welsh, S. D. Gribble, E. A. Brewer, and D. Culler. A design framework for highly con-current systems. Technical Report UCB/CSD-00-1108, U.C. Berkeley Computer Science Division, April 2000.
3. Lauer, Hugh C. and Needham, Roger M.: On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.* (1979), vol. 13:pp. 3-19
4. Benjamin Erb: Concurrent Programming for Scalable Web Architectures,(Diploma Thesis), Ulm University (2012).
5. Pai, Vivek S.; Druschel, Perter and Zwaenepoel, Willy: Flash: An efficient and portable Web server, in: *Proceedings of the annual conference on USENIX Annual Technical Conference,* USENIX Association, Berkeley, CA, USA, pp 15-15
6. The Apache Software Foundation, Module Index. http://httpd.apache.org/docs/2.2/mod/ (2012)
7. Kegel, Dan: The C10k Problem, Tech. rep., http://www.kegel.com/c10k.html (2012).
8. Stevens, W. Richard; Fenner, Bill and Rudoff, Andrew M: *Unix Network Programming, Volume I: The sockets networking API(3rd Edition)*, Addison-Wesley Professional (2003)
9. Google Inc, 'V8 design elements', https://developers.google.com/v8/design#mach_code (2012)
10. Tilkov, S., Vinoski, S.: Node.js: Using JavaScript to Build High-Performance Network Programs. Internet Computing, IEEE, 2010.
11. Eich, B. http://brendaneich.com/2008/04/popularity/ (2008)/
12. Ecma International, ECMA-262: ECMAScript Language Specification, 5.1 edn, ECMA (European Association for Standardizing Information and Communication Systems),

Geneva, Switzerland. URL:
http://www.ecmainternational.org/publications/standards/Ecma-262.htm (2011).

13. Sussman, G. J. & Steele Jr., G. L., Scheme: An interpreter for extended lambda calculus, in 'MEMO 349, MIT AI LAB', 1975.
14. Torstensson, D. & Eloff, E.: An Investigation into the Applicability of Node.js as a Platform for Web Services. (Student paper). Linköpings universitet, 2012.
15. Sparky 'Human task switches considered harmful' http://www.joelonsoftware.com/articles/fog0000000022.html (2001).
16. Elhage, N., http://blog.nelhage.com/2012/03/why-node-js-is-cool/ (2012).
17. Adya, A., Howell, J., Theimer, M., Bolosky, W. J. & Douceur, J. R., Cooperative task management without manual stack management, in 'Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference', ATEC '02, USENIX Association, Berkeley, CA, USA, pp. 289– 302, 2002. http://dl.acm.org/citation.cfm?id=647057.713851
18. Welsh, Matt; Culler, David and Brewer, Eric: SEDA: an architecture for well-conditioned, scalable internet services, in: Proceedings of the eighteenth ACM symposium on Operating systems principles, SOSP '01, ACM, New York, NY, USA, pp. 230–243
19. Welsh M, The staged event-driven architecture for highly concurrent server applications, 2000.
20. Robert Ryan McCune, Node.js Paradigms and Benchmarks(Student paper). University of Notre Dame, 2011.
21. Pedro Teixeira, Professional Node.js: Building JavaScript Based Scalable Software. Wrox, 2012.

Acknowledgement

I am grateful to my guide, Mrs. Nirali Nanavati for her systematic guidance and encouragement. I would like to acknowledge everyone else who have directly or indirectly helped me in preparing this seminar report.

Ganesh R Iyer