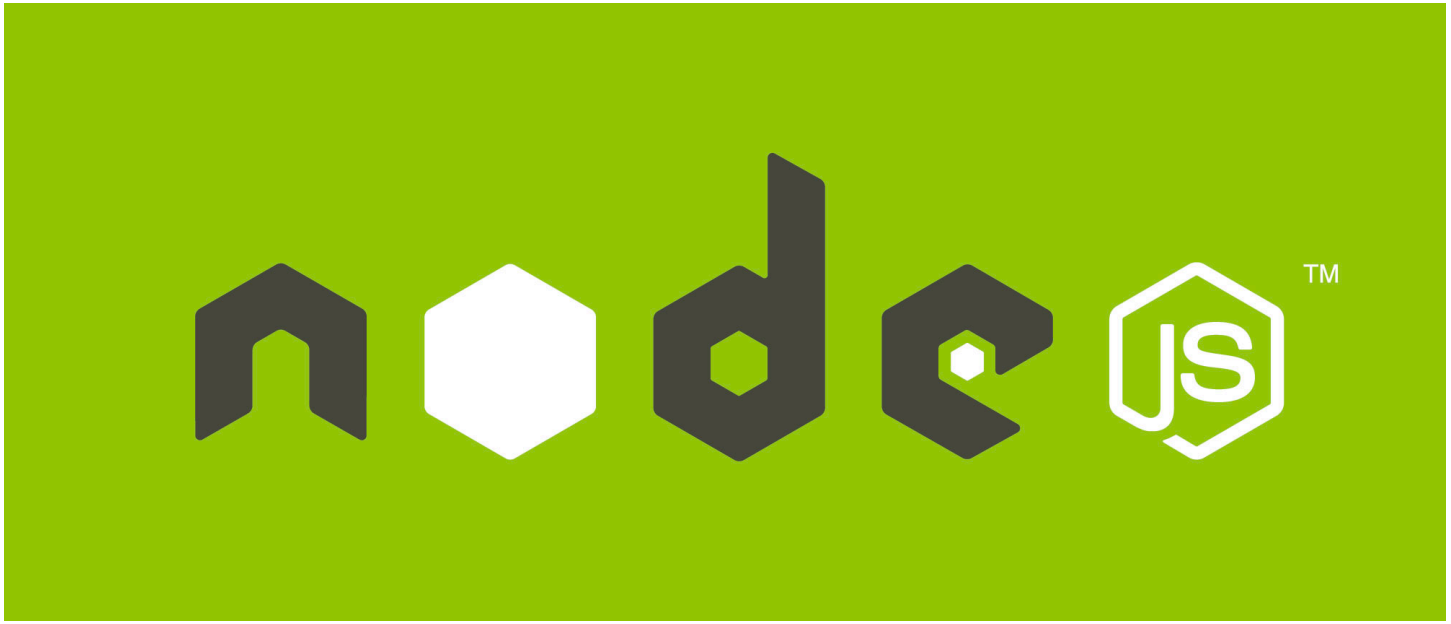
NodeJS

01

Presented by
Subhash EP

Subhash


node
JS
TM

Install NodeJS

Windows Installer (.msi)	32-bit	64-bit
Windows Binary (.exe)	32-bit	64-bit
Mac OS X Installer (.pkg)	Universal	
Mac OS X Binaries (.tar.gz)	32-bit	64-bit
Linux Binaries (.tar.gz)	32-bit	64-bit
SunOS Binaries (.tar.gz)	32-bit	64-bit
Source Code	node-v0.10.26.tar.gz	

Subhash

command-line tools

- node
- npm

REPL

- Read
- Evaluate
- Print
- Loop

Node REPL

- To test and experiment with code
- When you run node without any command line arguments,
  - it puts you in the REPL mode
- To view the options available,
  - type .help and press Enter in REPL mode

Node REPL Demo

Executing Node.js Scripts

- You can execute a JavaScript source file in Node.js
  - by simply passing the file to node on the command line
    - Create a new JS file helloworld.js in code directory
    - node code/helloworld

Node JS Execution Demo

Basics

- Variables
  - var
- console.log
- Data Types
  - Numbers – 1,2,3,..,..
  - Boolean – true / false
  - Arrays – [ ]
  - Object Literals – { }

Basics (cont.)

- Functions
- Immediately Executing Functions
- Anonymous Functions
- High-Order Functions

Basics Demo

Closures

- In JavaScript, a closure is a function to which the variables of the surrounding context are bound by reference.

```
1. function getMeAClosure() {
2.     var seeMe = "here I am";
3.     return (function theClosure() {
4.         return {canYouSeeIt: seeMe ? "yes!": "no"};
5.     });
6. }
7.
8. var closure = getMeAClosure();
9. closure().canYouSeeIt; //"yes!"
```

Lexical Scope

- lexical scope of a function is statically defined by the function's physical placement within the written source code

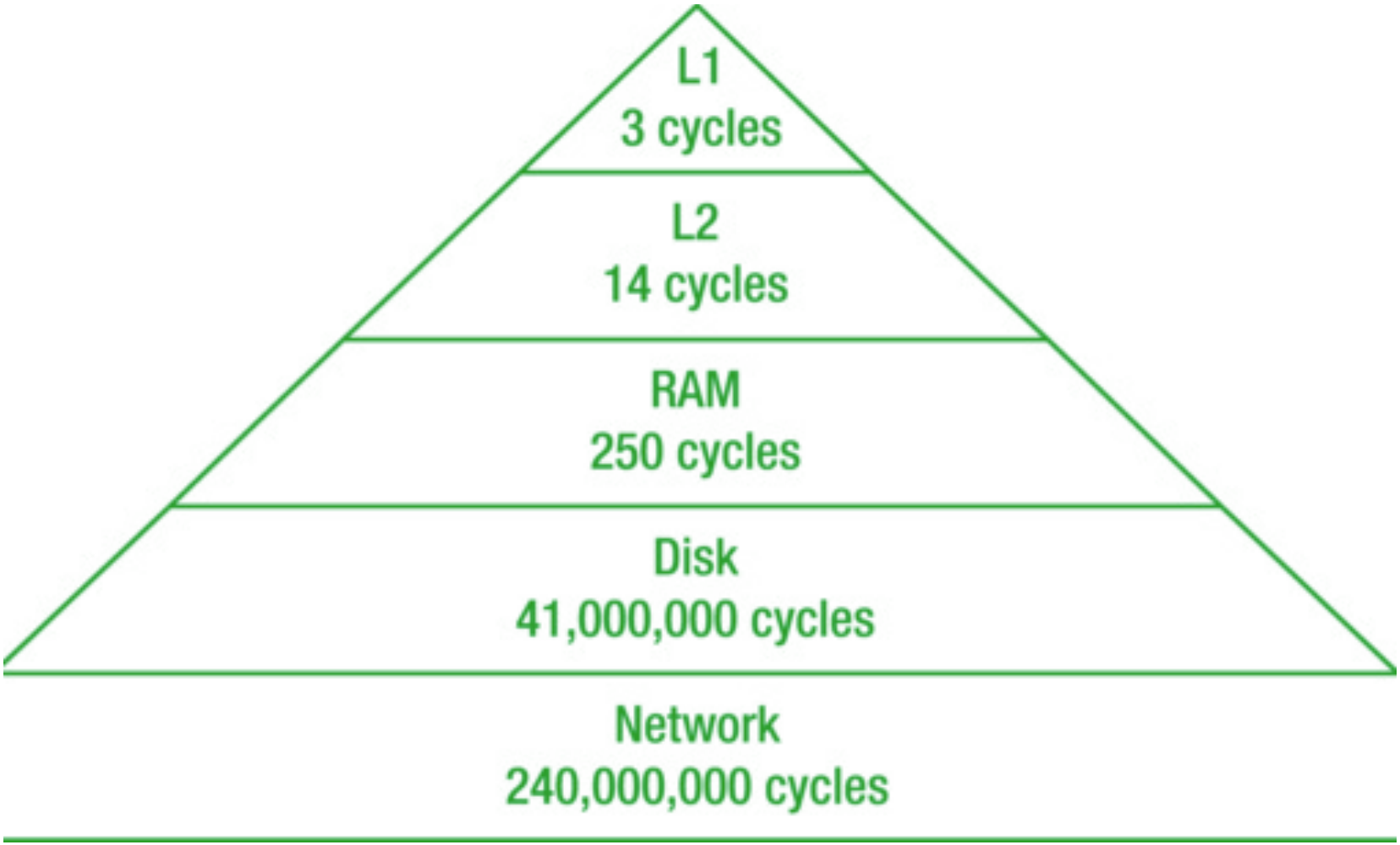
```
1. var x = "global";
2.  function outer() {
3.     var y = "outer";
4.      function inner() {
5.         var x = "inner";
6.     }
7. }
```

Closures Demo

Node.js Performance

- Node.js is focused on creating highly performing applications.
- How?
- First let us introduce the I/O scaling problem.
- Then let us see how it has been solved
  - traditionally,
  - followed by Node.js

I/O Scaling Problem – typical scene



Subhash

I/O Problem Explained

- You can clearly see that Disk and Network access is in a completely different category from accessing data that is available in RAM and CPU cache

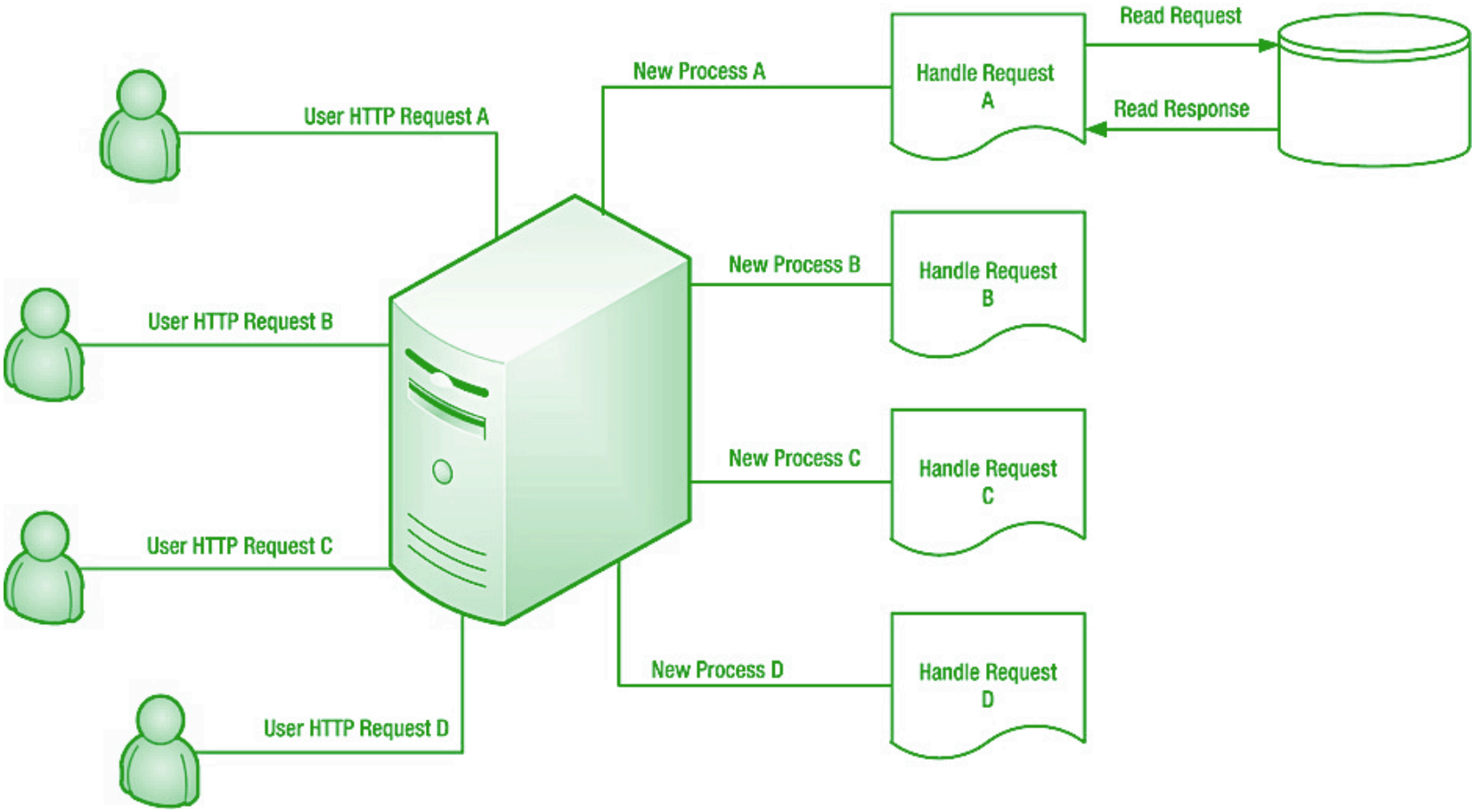
I/O Problem Explained

- Most web applications depend on reading data from disk or from another network source (for example, a database query).
- When an HTTP request is received and we need to load data from a database, typically this request will be spent waiting for a disk read or a network access call to complete.

I/O Problem Explained

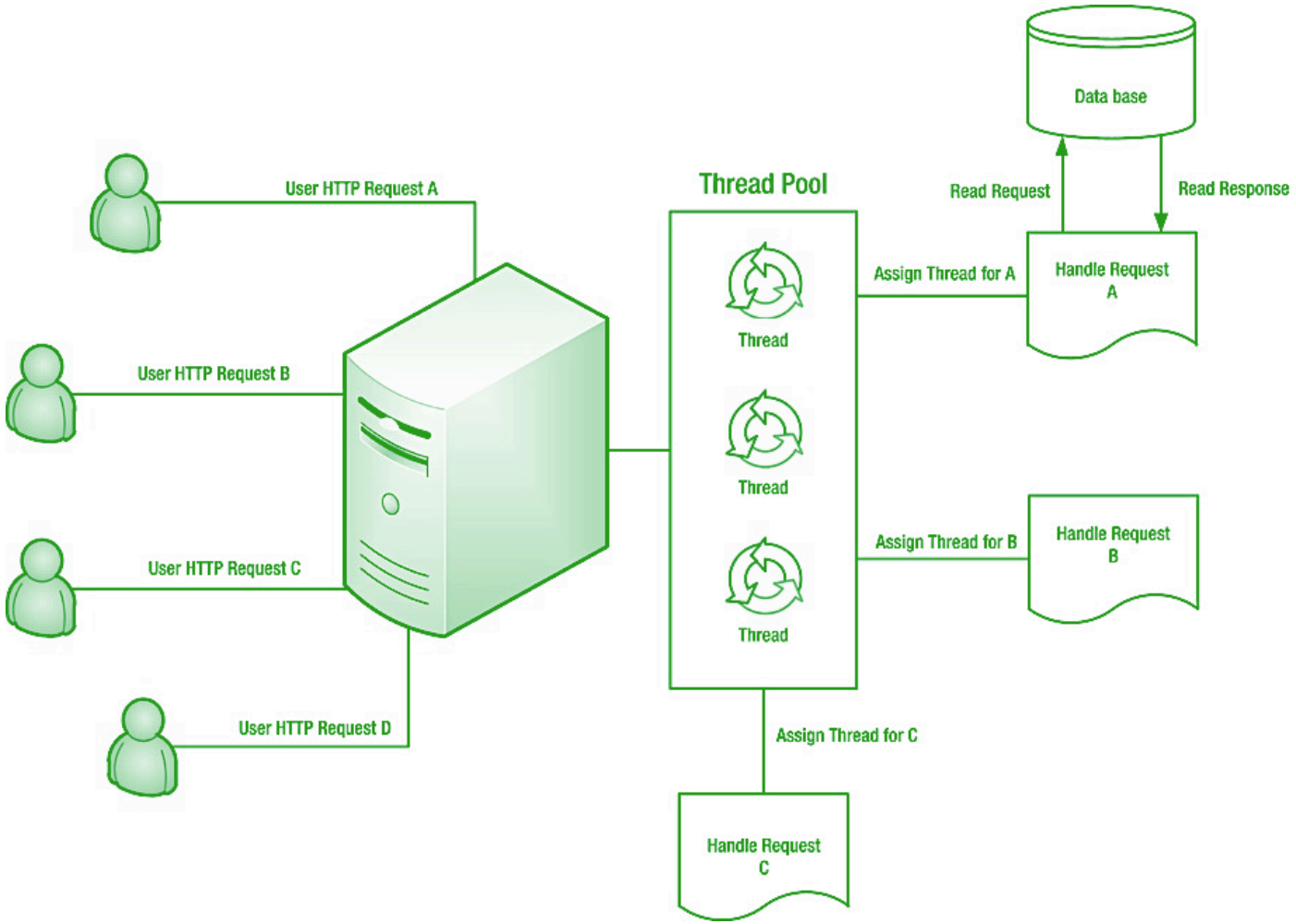
- These open connections and pending requests consume server resources (memory and CPU).
- In order to handle a large number of requests from different clients using the same web server, we have the I/O scaling problem.

Traditional - Process Per Request



Subhash

Traditional – Thread Pool



Subhash

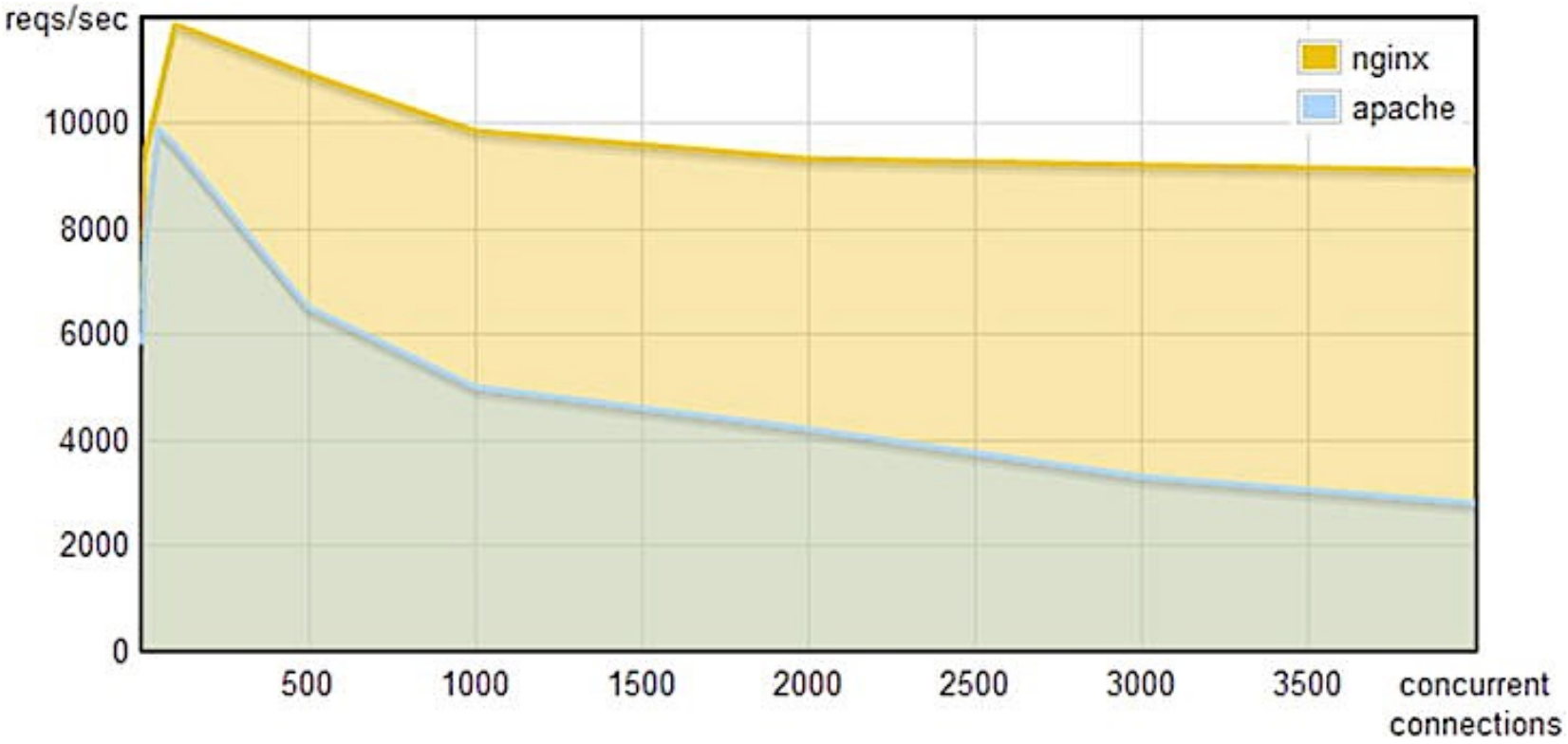
The Nginx Way

- We have seen that creating separate processes and separate threads to handle requests results in wasted OS resources.
- The way Node.js works is that there is a single thread handling requests.
- The idea that a single threaded server can perform better than a thread pool server is not new to Node.js

The Nginx Way

- Nginx is built on this principle.
- Nginx is a single-threaded web server and can handle a tremendous amount of concurrent requests.

benchmark comparing Nginx to Apache

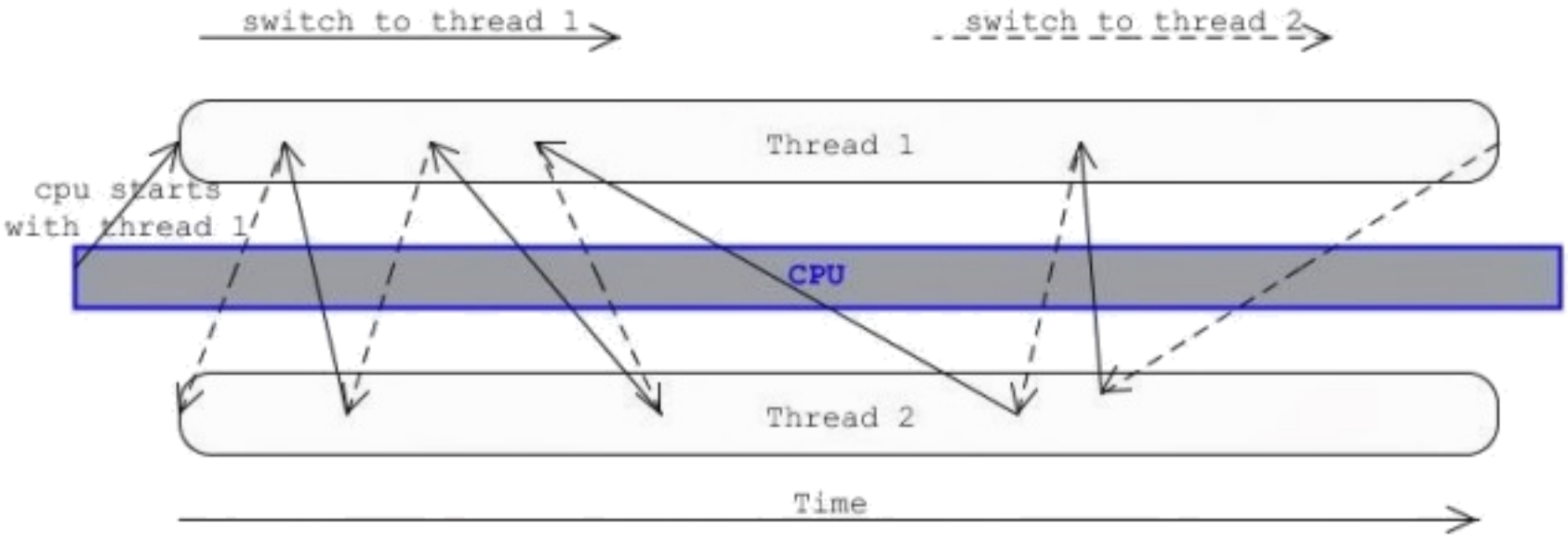


Subhash

Secret!!

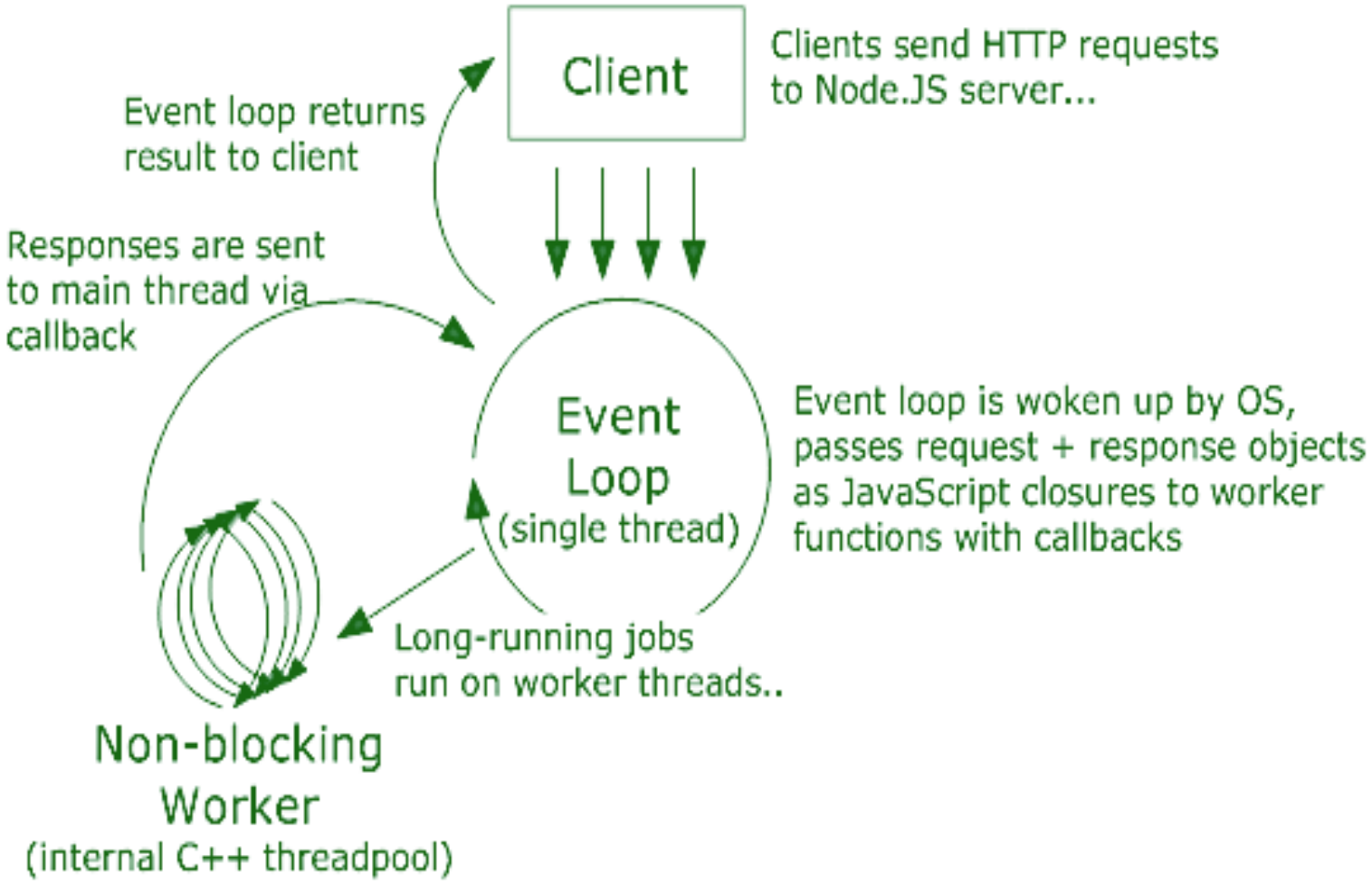
Subhash

Threads require context switching



They require evils such as synchronization and create headaches with maintaining concurrency

node.js has an Event Queue



Why use JavaScript?

- Web developers are already a little familiar with it
- JavaScript has never needed more than one thread because it's entirely event-driven
- JavaScript never had any type of blocking I/O because it has no built-in I/O API's

Blocking I/O

- Syncronous function call in C#

```
public static void getConfig() {
    StreamReader sr = new StreamReader("cfg.txt");
    using (("TestFile.txt")) {
      String line = sr.ReadToEnd();
      Console.WriteLine(line);
    }
}
:
function will wait for results..
```

Asynchronous I/O

- Asyncronous function call in node.js

```
function getConfig(req, res) {
    fs.readFile ("c:\\config.xml", function(err,
data) {
        if (!err) res.send(200, JSON.stringify(data));
    });
}
:
execution falls through and continues..
```

What about Error handling?

- If function is synchronous, simply return an error
- If function is asynchronous, then first arg returned is err
- If flow is more complex, then add an event listener and emit an error event from anywhere within your code as opposed to using try / catch / throw which really only work on synchronous code.

node.js uses Google's V8

- Compiles JavaScript to machine language, as opposed to bytecode, for speed
- Uses advanced optimization techniques, such as copy elision to avoid duplication of objects
- Made from Scratch, its Open Source, and its used in used in MongoDB and Chrome

Why look into node.js

- Very active Open Source community
  - socket.io express uglify-js less-css async jade mongo
- Designed for building fast and scalable real-time mobile applications and websites. *Its coming to Desktop too!*
- Dynamic language that is consistently asynchronous and compatible with browser code libraries.

Performance Demo

That's all

End of Session

©

subhash.ep@gmail.com
linkedin.com/in/subhashep