Startup Engineering
Balaji S. Srinivasan
Stanford University

Class: class.coursera.org/startup-001
Signup: coursera.org/course/startup
Forums: bit.ly/startup-eng-forum

Node.js: Asynchrony, Flow Control, and Debugging

In this lecture we will discuss latency, asynchronous programming with the async flow control library, and basic debugging with the Node debugger. To begin, it's useful to understand at a high level what problems Node is trying to solve, what it's good for, and what it's not suited for.

Motivation: reduce the impact of I/O latency with asynchronous calls

To understand the motivation behind Node's systematic use of asynchronous functions, you need to understand latency. The first thing to know is that different computer operations can take radically different amounts of time to complete, with input/output (I/O) actions such as writing to disk or sending/receiving network packets being particularly slow (Table 1).

Table 1: *Latencies associated with various computer operations as of mid-2012. Here, 1 ns is one nanosecond (10^{-9} s), and 1 ms is one millisecond (10^{-3} s). (Source: Jeff Dean and Peter Norvig.)*

Operation	Time (ns)	Time (ms)	Notes
L1 cache reference	0.5 ns		
Branch mispredict	5 ns		
L2 cache reference	7 ns		14x L1 cache
Mutex lock/unlock	25 ns		
Main memory reference	100 ns		20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000 ns		
Send 1K bytes over 1 Gbps network	10,000 ns	0.01 ms	
Read 4K randomly from SSD*	150,000 ns	0.15 ms	
Read 1 MB sequentially from memory	250,000 ns	0.25 ms	
Round trip within same datacenter	500,000 ns	0.5 ms	
Read 1 MB sequentially from SSD*	1,000,000 ns	1 ms	4X memory
Disk seek	10,000,000 ns	10 ms	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000 ns	20 ms	80x memory, 20X SSD
Send packet CA -> Netherlands -> CA	150,000,000 ns	150 ms	

To illustrate why these latency numbers are important, consider the following simple example, which downloads a number of URLs to disk and then summarizes them:

```
var results = [];
for(var i = 0; i < n; i++) {
  results.push(download(urls[i]));
}
summarize(results);
```

In this code we wait for each download to complete before we begin the next one. In other words, the downloads are *synchronous*. This means that each line of code executes and completes in order. If we conceptually unrolled the for loop it would look like this:

```
var results = [];
results.push(download(urls[0])); // block till urls[0] downloaded
results.push(download(urls[1])); // block till urls[1] downloaded
results.push(download(urls[2])); // block till urls[2] downloaded
// ...
results.push(download(urls[n])); // block till urls[n] downloaded
summarize(results);
```

Each line here completes before the next begins. That is, the script hangs or *blocks* on each line. Now let's define the following mathematical variables to analyze the running time of this simple code block:

- V: time to initialize a variable like `results`
- D: time to download a url via `download`
- A: time to append to an array like `results`
- S: time to calculate `summary(results)`

Then our total running time for the synchronous version (T_s) is:

$$T_s = V + D_1 + A_1 + D_2 + A_2 + ... + D_n + A_n + S$$

Crucially, from the latency numbers in 1, we can assume that $D_i \gg V$ and $D_i \gg A_i$, though S might be quite long if it's doing some heavy math. Thus we can simplify the expression to:

$$T_s \approx D_1 + D_2 + ... + D_n + S$$

Now, the reason that D_i takes so long is in part because we are waiting on a remote server's response. What if we did something in the meantime? That is, what if we could enqueue these requests in parallel, such that they executed *asynchronously*? If we did this, we would still start the downloads in order from 1 to n, but they could complete all at different times, and this would be a bit confusing. However, our new asynchronous running time (T_a) would now look like this:

$$T_a = \max(D_1, D_2, ..., D_n) + S$$

The reason it's the max with the asynchronous implementation is because we are only waiting for the slowest download, which takes the maximum time. And because $D_i > 0$ (as delay times must be positive), this is in theory strictly faster than the synchronous version:

$$\max(D_1, D_2, ..., D_n) < D_1 + D_2 + ... + D_n$$

In other words, it could be faster (potentially much faster) if we could kick off a number of asynchronous requests, where we wouldn't *block* on the completion of each download before beginning the next one. Our wait time would be dominated by the maximum delay, not the sum of delays. This would be very useful if we were writing a web crawler, for example. And this is exactly what Node is optimized for: serving as an orchestrator that kicks off long-running processes (like network requests or disk reads) without blocking on their completion unless necessary. As a concrete example, take a look at this script, which can be run as follows:

```
wget https://d396qusza40orc.cloudfront.net/startup/code/synchronous-ex.js
npm install async underscore sleep
node synchronous-ex.js
```

If we execute this script, you will see output similar to that in Figure 1. As promised by theory, the synchronous code's running time scales with the sum of all individual delays, while the asynchronous code has running time that scales only with the single longest delay (the max).

```
[ubuntu@ip-172-31-28-87:~]$node synchronous-ex.js
Synchronous start at Thu Aug 01 2013 04:05:29 GMT+0000 (UTC)
  http://www.bing.com/search?q=0 start at Thu Aug 01 2013 04:05:29 GMT+0000 (UTC)
  http://www.bing.com/search?q=0 end at Thu Aug 01 2013 04:05:30 GMT+0000 (UTC)
  http://www.bing.com/search?q=0 elapsed time: 862
  http://www.bing.com/search?q=1 start at Thu Aug 01 2013 04:05:30 GMT+0000 (UTC)
  http://www.bing.com/search?q=1 end at Thu Aug 01 2013 04:05:30 GMT+0000 (UTC)
  http://www.bing.com/search?q=1 elapsed time: 838
  http://www.bing.com/search?q=2 start at Thu Aug 01 2013 04:05:30 GMT+0000 (UTC)
  http://www.bing.com/search?q=2 end at Thu Aug 01 2013 04:05:30 GMT+0000 (UTC)
  http://www.bing.com/search?q=2 elapsed time: 103
  http://www.bing.com/search?q=3 start at Thu Aug 01 2013 04:05:30 GMT+0000 (UTC)
  http://www.bing.com/search?q=3 end at Thu Aug 01 2013 04:05:31 GMT+0000 (UTC)
  http://www.bing.com/search?q=3 elapsed time: 765
  http://www.bing.com/search?q=4 start at Thu Aug 01 2013 04:05:31 GMT+0000 (UTC)
  http://www.bing.com/search?q=4 end at Thu Aug 01 2013 04:05:31 GMT+0000 (UTC)
  http://www.bing.com/search?q=4 elapsed time: 214
Sum of times: 2774.613959947601 ms
Max of times: 860.0957898888737 ms
Synchronous end at Thu Aug 01 2013 04:05:31 GMT+0000 (UTC)
Synchronous elapsed time: 2793
Asynchronous start at Thu Aug 01 2013 04:05:31 GMT+0000 (UTC)
  http://www.bing.com/search?q=0 start at Thu Aug 01 2013 04:05:31 GMT+0000 (UTC)
  http://www.bing.com/search?q=1 start at Thu Aug 01 2013 04:05:31 GMT+0000 (UTC)
  http://www.bing.com/search?q=2 start at Thu Aug 01 2013 04:05:31 GMT+0000 (UTC)
  http://www.bing.com/search?q=3 start at Thu Aug 01 2013 04:05:31 GMT+0000 (UTC)
  http://www.bing.com/search?q=4 start at Thu Aug 01 2013 04:05:31 GMT+0000 (UTC)
  http://www.bing.com/search?q=2 end at Thu Aug 01 2013 04:05:32 GMT+0000 (UTC)
  http://www.bing.com/search?q=2 elapsed time: 101
  http://www.bing.com/search?q=4 end at Thu Aug 01 2013 04:05:32 GMT+0000 (UTC)
  http://www.bing.com/search?q=4 elapsed time: 213
  http://www.bing.com/search?q=3 end at Thu Aug 01 2013 04:05:32 GMT+0000 (UTC)
  http://www.bing.com/search?q=3 elapsed time: 763
  http://www.bing.com/search?q=1 end at Thu Aug 01 2013 04:05:32 GMT+0000 (UTC)
  http://www.bing.com/search?q=1 elapsed time: 838
  http://www.bing.com/search?q=0 end at Thu Aug 01 2013 04:05:32 GMT+0000 (UTC)
  http://www.bing.com/search?q=0 elapsed time: 860
Sum of times: 2774.613959947601 ms
Max of times: 860.0957898888737 ms
Asynchronous end at Thu Aug 01 2013 04:05:32 GMT+0000 (UTC)
Asynchronous elapsed time: 862
```

Figure 1: *The red boxes show that sync code scales with the sum, while async with the max. The blue boxes show the start times of the first and last sync vs. async downloads. The key is that async is non-blocking: each download does not wait for the next to complete before beginning.*

We start to see why async could be handy. The synchronous code is considerably slower than the equivalent asynchronous code here, because (1) there was no need for each download to *block* on the results of the previous download and (2) there is nothing we could have done within the Node process itself to significantly speed up each individual download to file. The IO bottleneck is on the remote server and the local filesystem, neither of which can be easily sped up (Table 1). Now let's take a look at the source code to see how this example works:

```
/*
   Illustrative example to compare synchronous and asynchronous code.

   We demonstrate in particular that synchronous code has a running time
   that scales with the sum of individual running times during the parallel
   section of the code, while the equivalent async code scales with the
   maximum (and thus can be much faster).

   We also show examples of object-oriented programming (via Timer) and
   functional programming (via build_insts).

   To install libraries:

     npm install async underscore sleep
*/
var async = require('async');
var uu = require('underscore');
var sleep = require('sleep');


/*
   Illustrates simple object-oriented programming style with JS. No
   inheritance here or need for prototype definitions; instead, a fairly
   self explanatory constructor for a simple object with some methods.

   As shown below, we pass new Timer instances into asynchronous code via
   closures so that they can track timings across the boundaries of async
   invocations.
*/
var Timer = function(name) {
    return {
        name: name,
        tstart: null,
        tend: null,
        dt: null,
        start: function() {
            this.tstart = new Date();
            console.log("%s start at %s", this.name, this.tstart);
        },
        end: function() {
```

```
            this.tend = new Date();
            console.log("%s end at %s", this.name, this.tend);
        },
        elapsed: function() {
            this.end();
            this.dt = this.tend.valueOf() - this.tstart.valueOf();
            console.log("%s elapsed time: %s ms", this.name, this.dt);
        }
    };
};

/*
   Illustrates the functional programming (FP) style.

   We create a set of dummy URLs by using underscore's map and range
   functions.

   Then we create a set of delays, in milliseconds, to use consistently
   across both the sync and async implementations.

   Finally, we use underscore's zip function () a few times to help
   us create a final data structure that looks like this:

    [ { url: 'http://www.bing.com/search?q=0',
        delay_ms: 860.4143052361906 },
      { url: 'http://www.bing.com/search?q=1',
        delay_ms: 91.59809700213373 },
      { url: 'http://www.bing.com/search?q=2',
        delay_ms: 695.1153050176799 },
      { url: 'http://www.bing.com/search?q=3',
        delay_ms: 509.67361335642636 },
      { url: 'http://www.bing.com/search?q=4',
        delay_ms: 410.48733284696937 } ]

   The reason we like a list of objects like this is that each individual
   element can be passed to a single argument function in a map invocation,
   which can then access the object's fields and do computations on it.
*/
var build_insts = function(nn) {
    var bingit = function(xx) { return 'http://www.bing.com/search?q=' + xx;};
    var urls = uu.map(uu.range(nn), bingit);
    var delays_ms = uu.map(uu.range(nn), function() { return Math.random() * 1000;});
    var to_inst = function(url_delay_pair) {
        return uu.object(uu.zip(['url', 'delay_ms'], url_delay_pair));
    };
    return uu.map(uu.zip(urls, delays_ms), to_inst);
};
```

```

/*
   Simple code that uses underscore's reduce to define a sum function.
   As we'll see, synchronous code scales with the sum of times while
   async code scales with the max.

*/
var summarize = function(results) {
    var add = function(aa, bb) { return aa + bb;};
    var sum = function(arr) { return uu.reduce(arr, add);};
    console.log("Sum of times: %s ms", sum(results));
    console.log("Max of times: %s ms", uu.max(results));
};

/*
    A straightforward synchronous function that imitates (mocks) the
    functionality of downloading a URL. We take in an inst object of the
    form:

      inst =  { url: 'http://www.bing.com/search?q=1',
                delay_ms: 91.59809700213373 }

    For illustrative simplicity, we fake downloading the URL here by simply
    doing a synchronous sleep with the sleep.usleep function. That is, we
    halt the process for delay_us seconds. The reason we do a fake download
    is that this way we can do an apples-to-apples comparison of an async
    version of the code in which each download took exactly the same time.

    We add two spaces to the beginning of the Timer invocation for
    formatting purposes in STDOUT, like indenting code.
*/
var synchronous_mock_download = function(inst) {
    var tm = new Timer('  ' + inst.url);
    tm.start();
    var delay_us = inst.delay_ms * 1000;
    sleep.usleep(delay_us);
    tm.elapsed();
    return inst.delay_ms;
};

/*
    A straightforward synchronous way to start a time, iterate over a bunch
    of URLs, download the files to disk, accumulate the times required to
    download those files, summarize the results and then stop the timer.
*/
var synchronous_example = function(insts) {
    var tm = new Timer('Synchronous');
```

```
    tm.start();
    var results = [];
    for(var ii = 0; ii < insts.length; ii++) {
        results.push(synchronous_mock_download(insts[ii]));
    }
    summarize(results);
    tm.elapsed();
};


/*
   Functionally identical to synchronous_example, this version is
   written to be structurally more similar to asynchronous_example
   for comparative purposes. Note that the loop is replaced with a
   map invocation and sent directly to summarize, which is
   the equivalent of a callback (i.e. directly acting on the results
   of another function).
*/
var synchronous_example2 = function(insts) {
    var tm = new Timer('Synchronous');
    tm.start();
    summarize(uu.map(insts, synchronous_mock_download));
    tm.elapsed();
};

/*
    Like ths synchronous_mock_download, we start the timer at the beginning.
    However, for the async version, we do the following:

    First, we replace the return statement by a callback invocation. Normally
    null would be an error message but we're not doing any error checking here
    just yet.

         return inst.delay_ms -> cb(null, inst.delay_ms)

    Then we pull the tm.elapsed() call into the delay function. It is exactly
    this call (along with the callback) which is delayed by inst.delay_ms in the
    setTimeout.

    The big difference from the synchronous_mock_download function is that
    we need to explicitly engineer certain lines of code to complete before
    other lines of code.
*/
var asynchronous_mock_download = function(inst, cb) {
    var tm = new Timer('   ' + inst.url);
    tm.start();
    var delayfn = function() {
```

```
        tm.elapsed();
        cb(null, inst.delay_ms);
    };
    setTimeout(delayfn, inst.delay_ms);
};

/*
    Restructures the synchronous_example2 to be asynchronous.  Note that the
    whole trick is in structuring code such that you ensure that certain
    lines occur after other lines - e.g. tm.elapsed() should not occur
    before summarize(results) can occur.
*/
var asynchronous_example = function(insts) {
    var tm = new Timer('Asynchronous');
    tm.start();

    var async_summarize = function(err, results) {
        summarize(results);
        tm.elapsed();
    };

    async.map(insts, asynchronous_mock_download, async_summarize);
};


/*
   Finally, the main routine itself is just a simple wrapper that
   we use to group and isolate code.
*/
var main = function() {
    var nn = 5;
    var insts = build_insts(nn);
    synchronous_example(insts);
    asynchronous_example(insts);
};

main();
```

You might think the async examples are more complicated. And it is true that for many simple scripts, the use of asynchronous coding can be considered a performance improvement that adds unnecessary complexity. You now need to think in terms of callbacks rather than return statements, you don't know when a given function will return for certain, and you generally need to give much more thought to the order of operations in your program than you did before. Maybe this just isn't worth it when you just want to download some URLs to disk.

However, it's much easier to turn an asynchronous program into a synchronous one than vice versa. Moreover, for any application that involves realtime updates (like chat, video chat, video games, or monitoring) or sending data across the network (like a webapp!), you'll find it anywhere from helpful to critical to think asynchronously from the very beginning. Additionally, in other languages like Python, while it's possible to write asynchronous code with libraries like Twisted, all the core libraries are synchronous. This makes it more difficult to write new asynchronous code in other languages as you would need to launch said synchronous code in background processes (similar to the bash ampersand) to prevent your main routine from blocking. Finally, in Node the use of flow control libraries (like async), implementations of Promises/A+ and Functional Reactive Programming, and the forthcoming yield statement allow cleanup and organization of asynchronous code so that it's much easier to work with; see Amodeo's excellent talk for more perspective if you want to go further than async.

All that said, it should be clear at this point why a server-side async language is the right tool at least some of the time. Ryan Dahl's rationale for doing Node specifically in JS was several-fold: first, that Javascript was very popular; second, that its asynchronous features were already heavily used in client side code (e.g. via AJAX); third, that despite this popularity and there was no popular server-side implemention of JS and hence no legacy synchronous library code; and fourth, that Google's then-new v8 JS interpreter provided an engine that could be extracted from Chrome and placed into a command line utility. Thus, Node was born, with the concept that Node libraries could be used in both the client and server and could be built from the ground up to be async by default. That's exactly what has happened over the last few years.

What are the advantages and disadvantages of Node?

Now that we understand the motivation for Node's asynchronous programming paradigm (namely, to minimize the impact of IO latency), we can start thinking through the advantages and disadvantages of Node:

- Advantages
  - Node.js is in Javascript, so you can share code between the frontend and backend of a web application. The potential for reducing code duplication, increasing the pool of available engineers, and improving[1] conceptual integrity is perhaps the single most important reason Node is worth learning. It is likely to become very popular in the medium-term, especially once new frameworks like Meteor (which are built on Node) start gaining in popularity. (1, 2, 3).
  - Node is inherently well-suited for:
    - Working with existing protocols (HTTP, UDP, etc.), writing custom protocols or customizing existing ones. The combination of the Node standard library

[1] For example, in the Django framework for Python, to implement something seemingly simple like an email validation, you would be checking that the syntax of the email satisfies a regular expression (often in JS on the client side for responsiveness) and that the email itself is not already present in a database (in Python code on the server side). Either of these would raise a different error message which you would return and process in JS to update the page, displaying some kind of red error message. Try signing up for a Yahoo account to see an example. The issue here is that something which is conceptually simple (validating an email) can have its logic split between two different languages on the client and server. But with the new full-stack JS frameworks like Meteor, built on top of Node, you can simply do something like `Meteor.isClient` and `Meteor.isServer` (example) and put all the email validation logic in the same place, executing the appropriate parts in the appropriate context.

and the support for asynchrony/event-based programming make it natural to think of your app in terms of protocols and networked components.

   * Soft realtime apps[2], like chat or dashboards. Combining Node on the backend with something like Angular.js on the frontend can enable sophisticated event-based apps that update widgets very rapidly as new data comes in.
   * JSON APIs that essentially wrap a database and serve up JSON in response to HTTP Requests.
   * Streaming apps, like transloadit's realtime encoding of video.
   * Full stack JS interactive webapps like Medium.com or the kinds of things you can build with the Meteor framework.

- Disadvantages
  - Node is not suitable for heavy mathematics (1, 2, 3); you'd want to call an external C/C++ math library via addons or something like node-ffi (for "foreign function interface") if you need low-latency math within your main program. If you can tolerate high-latency[3], then a python webservice exposing numpy/scipy could also work (example).
  - Node is fine for kicking off I/O heavy tasks, but you wouldn't want to implement them in Node itself; you probably want to use C/C++ for that kind of thing to get low-level access, or perhaps Go[4].
  - The idiomatic way of programming in Node via asynchronous callbacks is actually quite different from how you'd write the equivalent code in Python or Ruby, and requires significant conceptual adjustment (1, 2, 3), though with various flow control libraries (especially async) and the important new yield statement, you can code in Node in a manner similar to how you'd do functional programming in Python or Ruby.
  - It's maturing rapidly, but Node doesn't yet have the depth of libraries of Python or Ruby.

It's also useful to keep in mind that the `node` binary is more similar to the `python` or `ruby` binaries, in that it is a server-side interpreter[5] with a set of standard libraries. The the web framework on top of Node is a distinct entity in its own right. Indeed, it is likely that the

[2] You wouldn't want to use Node for hard realtime apps, like the controller for a car engine, due to the fact that JS is garbage collected and thus can cause intermittent pauses. It is true that the v8 JS engine that Node is based on now uses an incremental garbage collector (GC) rather than a stop the world GC, but it is still fair to say that real-time apps in garbage-collected languages are a subject of active research, and that you'd probably want to program that car engine controller in something like C with the PREEMPT_RT patch that turns Linux into a realtime OS (1, 2).

[3] By "tolerate high latency" we mean you can tolerate a delay between when you invoke the math routine and when you get results. For example, this would be acceptable for offline video processing, but would not be acceptable for doing realtime graphics.

[4] You should probably use C/C++, though, if you're already using Node. The reason is that you don't want to get too experimental. Node is vastly more stable than it was a year or two ago, and is used for real sites now, but you want to limit the number of experimental technologies in your stack.

[5] If we want to be precise: the `node` binary is not the same thing as the abstract Javascript programming language (specified by the grammar in the ECMAScript specification), in the same way that the CPython binary named `python` is not the same thing as the abstract Python programming language (which is specified by a grammar).

ultimate successor to Ruby/Rails or Python/Django will be based on Node.js. There are a few candidates for this successor:

- The so-called MEAN stack (MongoDB, Express, Angular, and Node), as a pseudo-successor[6] to the LAMP stack (Linux, Apache, MySQL, and PHP).
- A realtime JS framework like Meteor or Derby. Meteor in particular is backed by significant venture capital and is worth checking out (do try the examples here). When it attains full maturity, it will likely become quite popular.
- It's also worth mentioning the Matador framework from Medium.com by the founder of Blogger and Twitter; this is one of the first large sites built on Node (here's more)

For this class we'll be using three out of four components of the MEAN stack: E (Express), A (Angular), and N (Node). Rather than MongoDB and the associated Mongoose module, however, we'll use the somewhat more conservative choice of PostgreSQL and the sequelize module for our database. Overall, Node is maturing rapidly and being used in production at an increasing number of companies, so it's worth experimenting with while keeping in mind that it has more in the way of rough edges than older platforms.

Asynchronous Programming and Flow Control

Now that we understand what Node is suited for, let's extend this and do a more in-depth example of asynchronous programming. We want to do something seemingly simple: hit the Node documentation site, get a list of all modules, and group these modules by their stability. Adapting an example from Takada's book, in pseudocode this would look something like this:

```
for(var i = 0; i < docurls; i++) {
  request(docurls[ii], function(err, resp, body) {
      // Parse the response and response body
  });
}
after_url_downloading();
```

This code is very simple when synchronous: loop over a bunch of URLs, download them, and then move on to the next thing. However, this seemingly simple `for` loop introduces at least three new things to keep in mind when the internal `request` calls become asynchronous:

1. First, we need to to wait for all the `request` calls to complete before executing `after_url_downloading`.

[6]We say pseudo-successor because the individual terms in the acronym don't map one-to-one. Specifically, (1) Linux and/or BSD are assumed today as the OS, whereas they weren't in the early 2000s when the LAMP acronym was coined (as many were still using Windows). That is, the MEAN stack is assumed to be deployed on Linux and often developed on a Mac. (2) Apache would potentially be replaced by something like `nginx` rather than Node directly. (3) PHP in the past was both a server-side and client-side language, whereas Node is the basic JS interpreter, Express is the backend web framework, and Angular is the frontend web framework. (4) MongoDB could be considered a replacement for MySQL, and the MongoDB/mongoose combination is indeed well debugged and allows you to use JS within the database as well, but you also want to consider using PostgreSQL/sequelize, because PostgreSQL is more stable/mature than MongoDB. That said, MEAN is an interesting acronym and tech stack that looks likely to gain some traction.

2. Second, we might need to limit the number of parallel `request` calls if `docurls` is a very long list (e.g. 1000s of URLs). You can do `ulimit -n` to determine the number of simultaneous filehandles that your OS supports; see here for more details.

3. Third, we will often want to pass the results of these downloads to the next function, and right now the callback in each `request` isn't populating a data structure.

To solve these issues we need:

1. A way to force all the `requests` to complete before `after_url_downloading`

2. A way to determine when all `request` invocations been completed (e.g. by setting a flag)

3. A way to accumulate the results of the requests into a datastructure for use by subsequent code

4. A way to limit the parallelization of the number of requests

Let's take a look at the following code, which solves these issues in order to download the Node documentation and output the list of modules grouped by stability. First, download and execute the script on an EC2 instance by doing this:

```
wget https://d396qusza40orc.cloudfront.net/startup/code/list-stable-modules.js
npm install underscore async request
node list-stable-modules.js
```

When you run the script it should look like Figure 2.

```
[ubuntu@ip-172-31-28-87:~]$node list-stable-modules.js
{
  "1": [
    "documentation",
    "cluster"
  ],
  "2": [
    "crypto",
    "domain",
    "punycode",
    "readline",
    "stream",
    "tty",
    "vm"
  ],
  "3": [
    "buffer",
    "child_process",
    "debugger",
    "dns",
    "fs",
    "http",
    "https",
    "net",
    "path",
    "querystring",
    "string_decoder",
    "tls",
    "dgram",
    "url",
    "zlib"
  ],
  "4": [
    "console",
    "events",
    "os"
  ],
  "5": [
    "assert",
    "modules",
    "timers",
    "util"
  ],
  "undefined": [
    "synopsis",
    "addons",
    "globals",
    "process",
    "repl"
  ]
}
```

Figure 2: *The script hits the nodejs.org website, downloads all the documentation in JSON, and outputs modules by their degree of stability.*

Now let's see how the code works. At a high level it does the following:

1. Downloads http://nodejs.org/api/index.json
2. Extracts all modules and creates a list of module URLs (e.g. http://nodejs.org/api/fs.json)
3. Downloads each module URL JSON asynchronously, launching dozens of simultaneous requests
4. Parses the JSON, determines the stability index of each module, and organizes modules by stability
5. Prints this `stability_to_names` data structure to the console

Read through the following commented source code for more details. Our implementation relies crucially on the use of the async library, perhaps the most popular way in Node to manage asynchronous flow control. The basic concept is to develop functions that accomplish the above five tasks in isolation. As we get each function to work, we save its output to a variable with the simple `save` debugging utility.

```
/*
  0. Motivation.

  Try executing this script at the command line:

    node list-stable-modules.js

  The node.js documentation is partially machine-readable, and has
  "stability indexes" as described here, measuring the stability and
  reliability of each module on a 0 (deprecated) to 5 (locked) scale:

    http://nodejs.org/api/documentation.html#documentation_stability_index

  The main index of the node documentation is here:

     http://nodejs.org/api/index.json

  And here is a sample URL for the JSON version of a particular module's docs:

     http://nodejs.org/api/fs.json

  Our goal is to get the node documentation index JSON, list all modules,
  download the JSON for those modules in parallel, and finally sort and
  output the modules by their respective stabilities. To do this we'll use
  the async flow control library (github.com/caolan/async).

  As a preliminary, install the following packages at the command line:

     npm install underscore async request;
```

```

  Now let's get started.

*/
var uu = require('underscore');
var async = require('async');
var request = require('request');

/*
  1. Debugging utilities: save and log.

  When debugging code involving callbacks, it's useful to be able to save
  intermediate variables and inspect them in the REPL. Using the code below,
  if you set DEBUG = true, then at any point within a function you can put
  an invocation like this to save a variable to the global namespace:

     save(mod_data, 'foo')

  For example:

    function mod_urls2mod_datas(mod_urls, cb) {
      log(arguments.callee.name);
      save(mod_urls, 'foo');   // <------ Note the save function
      async.map(mod_urls, mod_url2mod_data, cb);
    }

  Then execute the final composed function, e.g. index_url2stability_to_names, or
  otherwise get mod_urls2mod_datas to execute. You can confirm that it
  has executed with the log(arguments.callee.name) command.

  Now, at the REPL, you can type this:

    > mod_urls = global._data.foo

  This gives you a data structure that you can explore in the REPL. This is
  invaluable when building up functions that are parsing through dirty data.

  Note that the DEBUG flag allows us to keep these log and save routines
  within the body of a function, while still disabling them globally by
  simply setting DEBUG = false. There are other ways to do this as well, but
  this particular example is already fairly complex and we wanted to keep
  the debugging part simple.
*/
var _data = {};
var DEBUG = false;
var log = function(xx) {
    if(DEBUG) {
```

```
        console.log("%s at %s", xx, new Date());
    }
};
function save(inst, name) {
    if(DEBUG) { global._data[name] = inst; }
}

/*
  NOTE: Skip to part 6 at the bottom and then read upwards. For
  organizational reasons we need to define pipeline functions from the last
  to the first, so that we can reference them all at the end in the
  async.compose invocation

  2. Parse module data to pull out stabilities.

  The mod_data2modname_stability function is very messy because the
  nodejs.org website has several different ways to record the stability of a
  module in the accompanying JSON.

  As for mod_datas2stability_to_names, that takes in the parsed data
  structure built from each JSON URL (like http://nodejs.org/api/fs.json)
  and extracts the (module_name, stability) pairs into an object. The for
  loop groups names by stability into stability_to_names. Note that we use
  the convention of a_to_b for a dictionary that maps items of class a to
  items of class b, and we use x2y for a function that computes items of
  type y from items of type x.

  The final stability_to_names data structure is the goal of this script
  and is passed to the final callback stability_to_names2console at
  the end of async.compose (see the very end of this file).
*/
function mod_data2modname_stability(mod_data) {
    var crypto_regex = /crypto/;
    var stability_regex = /Stability: (\d)/;
    var name_regex = /doc\/api\/(\w+).markdown/;
    var modname = name_regex.exec(mod_data.source)[1];
    var stability;
    try {
        if(crypto_regex.test(modname)) {
            var stmp = stability_regex.exec(mod_data.modules[0].desc)[1];
            stability = parseInt(stmp, 10);
        }
        else if(uu.has(mod_data, 'stability')) {
            stability =  mod_data.stability;
        }
        else if(uu.has(mod_data, 'miscs')) {
            stability =  mod_data.miscs[0].miscs[1].stability;
```

```
        }
        else if(uu.has(mod_data, 'modules')) {
            stability = mod_data.modules[0].stability;
        }
        else if(uu.has(mod_data, 'globals')) {
            stability = mod_data.globals[0].stability;
        } else {
            stability = undefined;
        }
    }
    catch(e) {
        stability = undefined;
    }
    return {"modname": modname, "stability": stability};
}

function mod_datas2stability_to_names(mod_datas, cb) {
    log(mod_datas);
    log(arguments.callee.name);
    modname_stabilities = uu.map(mod_datas, mod_data2modname_stability);
    var stability_to_names = {};
    for(var ii in modname_stabilities) {
        var ms = modname_stabilities[ii];
        var nm = ms.modname;
        if(uu.has(stability_to_names, ms.stability)) {
            stability_to_names[ms.stability].push(nm);
        } else{
            stability_to_names[ms.stability] = [nm];
        }
    }
    cb(null, stability_to_names);
}

/*
  3. Download module urls and convert JSON into internal data.

  Here, we have a function mod_url2mod_data which is identical to
  index_url2index_data, down to the JSON parsing. We keep it distinct for
  didactic purposes but we could easily make these into the same function.

  Note also the use of async.mapLimit to apply this function in parallel to
  all the mod_urls, and feed the result to the callback (which we can
  leave unspecified in this case due to how async.compose works).

  We use mapLimit for didactic purposes here, as 36 simultaneous downloads
  is well within the capacity of most operating systems. You can use ulimit
  -n to get one constraint (the number of simultaneous open filehandles in
```

```
  Ubuntu) and could parse that to dynamically set this limit on a given
  machine. You could also modify this to take a command line parameter.
*/
function mod_url2mod_data(mod_url, cb) {
    log(arguments.callee.name);
    var err_resp_body2mod_data = function(err, resp, body) {
        if(!err && resp.statusCode == 200) {
            var mod_data = JSON.parse(body);
            cb(null, mod_data);
        }
    };
    request(mod_url, err_resp_body2mod_data);
}

function mod_urls2mod_datas(mod_urls, cb) {
    log(arguments.callee.name);
    var NUM_SIMULTANEOUS_DOWNLOADS = 36; // Purely for illustration
    async.mapLimit(mod_urls, NUM_SIMULTANEOUS_DOWNLOADS, mod_url2mod_data, cb);
}

/*
  4. Build module URLs (e.g. http://nodejs.org/api/fs.json) from the JSON
  data structure formed from http://nodejs.org/api/index.json.

  The internal modname2mod_url could be factored outside of this function,
  but we keep it internal for conceptual simplicity.
*/
function index_data2mod_urls(index_data, cb) {
    log(arguments.callee.name);
    var notUndefined = function(xx) { return !uu.isUndefined(xx);};
    var modnames = uu.filter(uu.pluck(index_data.desc, 'text'), notUndefined);
    var modname2mod_url = function(modname) {
        var modregex = /\[([^\]]+)\]\(([^\)]+).html\)/;
        var shortname = modregex.exec(modname)[2];
        return 'http://nodejs.org/api/' + shortname + '.json';
    };
    var mod_urls = uu.map(modnames, modname2mod_url);
    cb(null, mod_urls);
}

/*
  5. Given the index_url (http://nodejs.org/api/index.json), pull
  down the body and parse the JSON into a data structure (index_data).

  Note that we could factor out the internal function with some effort,
  but it's more clear to just have it take the callback as a closure.

```

```
  Note also that we define the function in the standard way rather than
  assigning it to a variable, so that we can do log(arguments.callee.name).
  This is useful to follow which async functions are being executed and
  when.

*/
function index_url2index_data(index_url, cb) {
    log(arguments.callee.name);
    var err_resp_body2index_data = function(err, resp, body) {
        if(!err && resp.statusCode == 200) {
            var index_data = JSON.parse(body);
            cb(null, index_data);
        }
    };
    request(index_url, err_resp_body2index_data);
}

/*
  6. The primary workhorse async.compose (github.com/caolan/async#compose)
     sets up the entire pipeline in terms of five functions:

     - stability_to_names2console    // Print modules ordered by stability to console
     - mod_datas2stability_to_names  // List of modules -> ordered by stability
     - mod_urls2mod_datas     // nodejs.org/api/MODULE.json -> List of module data
     - index_data2mod_urls    // Extract URLs of module documentation from index_data
     - index_url2index_data   // Get nodejs.org/api/index.json -> JSON index_data

  The naming convention here is a useful one, especially at the beginning of
  a program when working out the data flow.  Let's understand this in terms
  of the synchronous version, and then the async version.

  Understanding the synchronous version
  -------------------------------------
  If this was a synchronous program, we would conceptually call these
  functions in order like so:

    index_data = index_url2index_data(index_url)
    mod_urls = index_data2mod_urls(index_data)
    mod_datas = mod_urls2mod_datas(mod_urls)
    stability_to_names = mod_datas2stability_to_names(mod_datas)
    stability_to_names2console(stability_to_names)

  Or, if we did it all in one nested function call:

     stability_to_names2console(
      mod_datas2stability_to_names(
       mod_urls2mod_datas(
```

```
      index_data2mod_urls(
       index_url2index_data(index_url)))))

That's a little verbose, so we could instead write it like this:

  index_url2console = compose(stability_to_names2console,
                              mod_datas2stability_to_names,
                              mod_urls2mod_datas,
                              index_data2mod_urls,
                              index_url2index_data)
  index_url2console(index_url)

This is a very elegant and powerful way to represent complex dataflows,
from web crawlers to genome sequencing pipelines. In particular, this
final composed function can be exposed via the exports command.

Understanding the asynchronous version
--------------------------------------
The main difference between the synchronous and asynchronous versions is
that the synchronous functions would each *return* a value on their last
line, while the asynchronous functions do not directly return a value but
instead pass the value directly to a callback. Specifically, every time
you see this:

  function foo2bar(foo) {
      // Compute bar from foo, handling errors locally
      return bar;
  }

You would replace it with this, where we pass in a callback bar2baz:

  function foo2bar(foo, bar2baz) {
      // Compute bar from foo
      // Pass bar (and any error) off to to the next function
      bar2baz(err, bar);
  }

So to compose our five functions with callbacks, rather than do a
synchronous compose, we use the async.compose function from the async
library:

  https://github.com/caolan/async#compose

The concept is that, like in the synchronous case, we effectively
generate a single function index_url2console which represents
the entire logic of the program, and then feed that index_url.
```

```
*/
function stability_to_names2console(err, stability_to_names) {
    log(arguments.callee.name);
    console.log(JSON.stringify(stability_to_names, null, 2));
}

var index_url2console = async.compose(mod_datas2stability_to_names,
                                      mod_urls2mod_datas,
                                      index_data2mod_urls,
                                      index_url2index_data);

var index_url = "http://nodejs.org/api/index.json";
index_url2console(index_url, stability_to_names2console);
```

Note that the code solves all four of the problems that we mentioned:

1. A way to force all the `requests` to complete before subsequent code was executed

   - We did this by using async.compose to force a given set of downloads to execute before another segment began (e.g. the function `mod_urls2mod_datas` ran before `mod_datas2stability_to_names` was invoked)

2. A way to determine when all `request` invocations been completed (e.g. by setting a flag)

   - Again, we did this with the async.compose structure. If you really wanted just a boolean flag, you could have combined async.map and async.every.

3. A way to accumulate the results of the requests into a datastructure for use by subsequent code

   - The async.compose function helped us out again, as the output of one stage was fed directly into the next. Alternatives here would be async.waterfall, async.series, or async.map.

4. A way to limit the parallelization of the number of requests

   - Here we used async.mapLimit to put an upper bound on the number of parallel downloads.

In general, async.compose is one of the cleanest ways to write server-side asynchronous JS code, especially for anything that can be reduced to a dataflow pipeline. One of the benefits of the compose approach is that you can write individual functions and debug them by themselves; another is that once you have the final composed function working, you can use `module.exports` to expose that alone to another module for import (e.g. in this case `index_url2console` would be the sole exported function).

As a useful exercise to test whether you understand this code, you might write a simple command line script that hits this Federal Register API JSON URL, builds up data structures for each listed agency, organizes them in a tree hierarchy as specified by the `parent_id` field, and then outputs the result to the command line.

Basic debugging with the Node Debugger

Now that we understand latency and have looked at a medium-size asynchronous script in detail, it's worth learning a bit more about debugging. To know how to debug code with confidence in a language is to understand that language. Let's do a quick example with the Node debugger. Suppose that we boot up an HTTP server as shown:

```
wget https://d396qusza40orc.cloudfront.net/startup/code/debugger-examples.js
node debugger-examples.js       # start the HTTP server
node debug debugger-examples.js # to invoke for debugging; see text
```

The source code is shown below:

```
#!/usr/bin/env node

var http = require('http');
var counter = 0;
var serv = http.createServer(function(req, res) {
    counter += 1;
    res.writeHead(200);
    debugger;
    res.end("Cumulative number of requests: " + counter);
});
var port = 8080;
serv.listen(port);
console.log("Listening at %s", port);
```

The goal of this server is simply to display the cumulative number of HTTP requests that have been made to the server over time. Our expectation is that as we refresh the page from any computer, that this number will increase. But if you download the script, run it with `node debugger-examples.js`, and view the results in Chrome, you will see something surprising. The counter increments by two rather than by one with each refresh. We can debug this with the built-in Node debugger, by doing `node debug debugger-examples.js`. See Figures 3-11 for a worked example.

Debugging Node code

This simple example illustrates how to debug the output produced by a Node HTTP server running on an EC2 instance.

① Download code (see text), determine the EC2 hostname, and run the script with: `node debugger-examples.js`

```
[ubuntu@ip-172-31-28-87:~]$wget https://d396qusza40orc.cloudfront.net/startup%2Fcode%2Fdebugger-examples.js -O debugger-examples.js
--2013-08-20 21:21:50--  https://d396qusza40orc.cloudfront.net/startup%2Fcode%2Fdebugger-examples.js
Resolving d396qusza40orc.cloudfront.net (d396qusza40orc.cloudfront.net)... 54.230.68.12, 54.230.68.147, 54.230.68.188, ...
Connecting to d396qusza40orc.cloudfront.net (d396qusza40orc.cloudfront.net)|54.230.68.12|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 318 [text/javascript]
Saving to: `debugger-examples.js'

100%[======================================================================================================>] 318

2013-08-20 21:21:50 (69.3 MB/s) - `debugger-examples.js' saved [318/318]

[ubuntu@ip-172-31-28-87:~]$curl http://169.254.169.254/latest/meta-data/public-hostname; echo ""
ec2-54-213-98-94.us-west-2.compute.amazonaws.com
[ubuntu@ip-172-31-28-87:~]$node debugger-examples.js
Listening at 127.0.0.1:8080
```

Figure 3: *First, download debugger-examples.js and determine the current EC2 hostname. Then invoke* `node debugger-examples.js` *at the command line to start the HTTP server.*

② View in Chrome and refresh a few times. Note mysterious bug: why is the count incrementing by two?

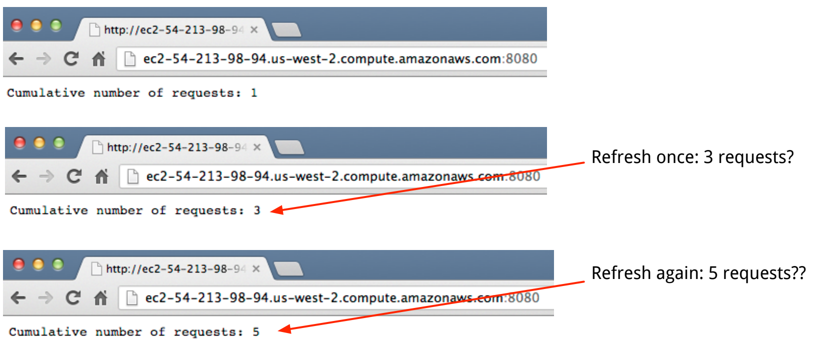

Figure 4: *Now open up Chrome and refresh a few times. You will see that the number of requests increments by two rather than one, which is surprising.*

③ Disable the rlwrap wrapper by using the bash alias command (as shown) or by commenting out the corresponding line in your ~/.bashrc file. This wrapper can be convenient, but will interfere with the ability to quit the repl in the debugger without exiting the whole thing.

```
[ubuntu@ip-172-31-28-87:~]$alias | grep node
alias node='env NODE_NO_READLINE=1 rlwrap node'
alias node_repl='node -e "require('\''repl'\'').start({ignoreUndefined: true})"'
[ubuntu@ip-172-31-28-87:~]$alias node='node'
[ubuntu@ip-172-31-28-87:~]$alias | grep node
alias node='node'
alias node_repl='node -e "require('\''repl'\'').start({ignoreUndefined: true})"'
```

Figure 5: *To use the debugger we'll need to disable the* `rlwrap` *wrapper around* `node`*. This is recommended by the Node documentation as it provides a better REPL for most purposes, but will interfere with the debugger's internal REPL.*

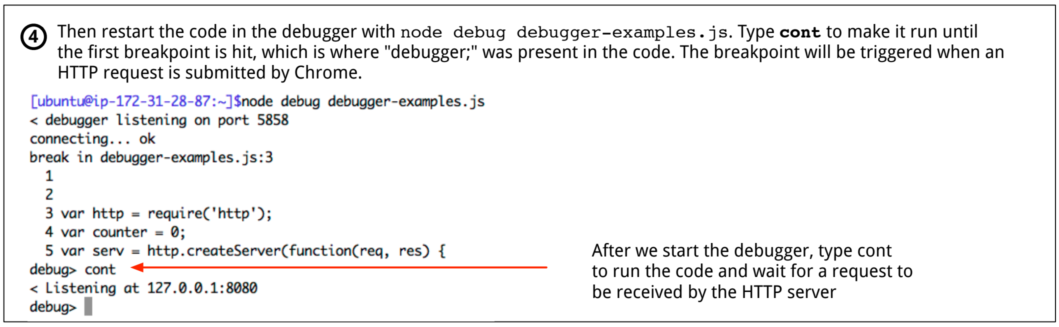

Figure 6: *Now we quit* `node debugger-examples.js` *and restart it as* `node debug debugger-examples.js`*, as shown. We then type* `cont` *to continue running the script until a breakpoint is reached, at which point we return control to the debugger.*

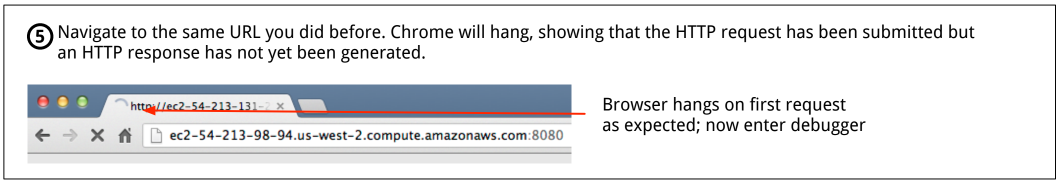

Figure 7: *Now we go to Chrome and artificially induce a breakpoint by connecting to the remote server. The browser will hang as an HTTP request has been set but no HTTP response has yet been received (because the code is halted at the breakpoint in the debugger, waiting for your input).*

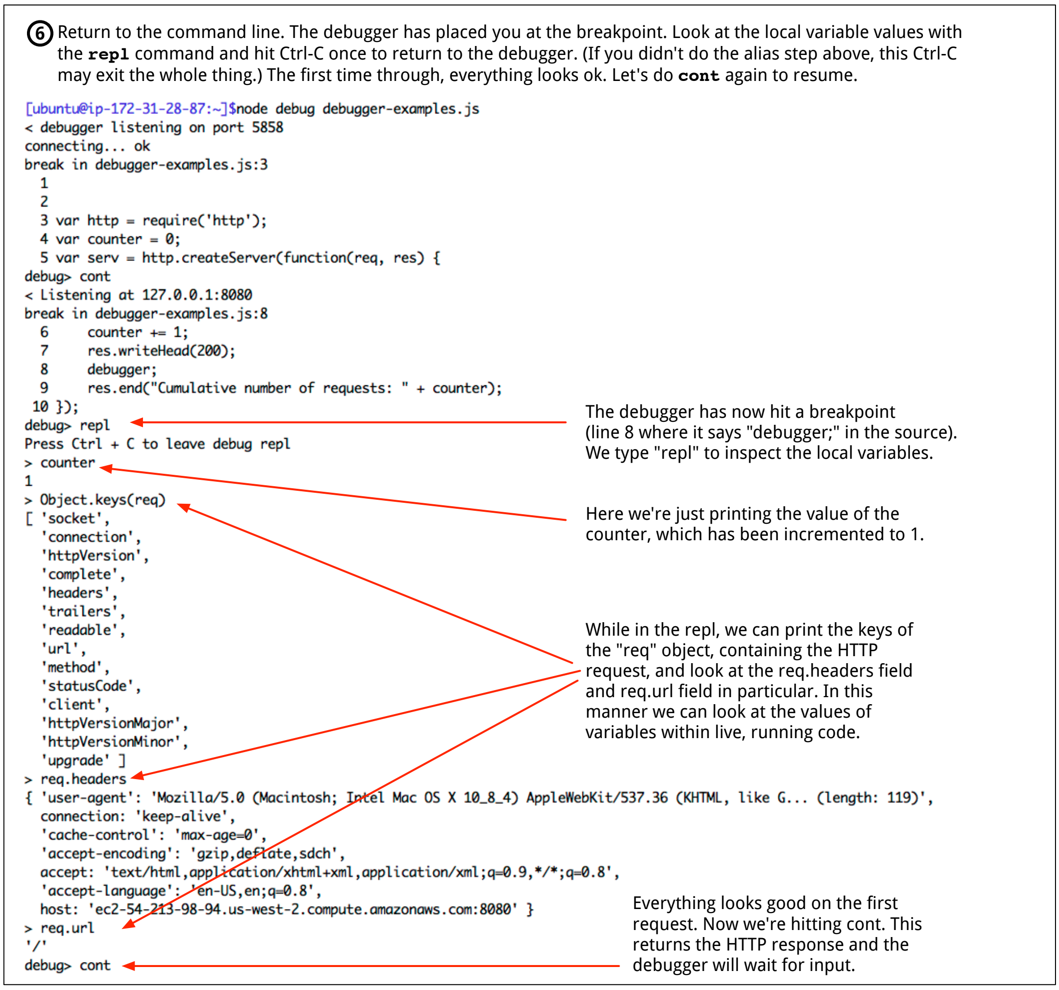

Figure 8: *We return to the debugger and run the* `repl` *command at the breakpoint. We poke around and print a few variables as shown. Everything looks in order so we exit the internal* `repl` *with Ctrl-C and resume execution with* `cont`*. This releases the HTTP response.*

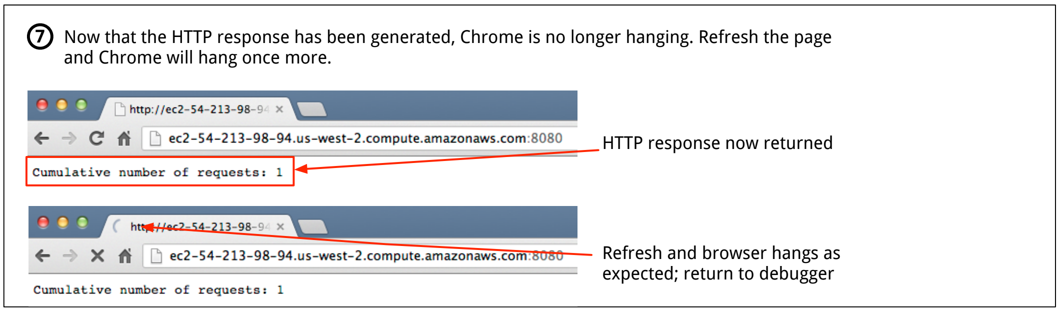

Figure 9: *Now when we return to Chrome, the HTTP response has been received and displayed. Let's try refreshing again to see if we can reproduce the oddity where the counter is incrementing by two each time.*

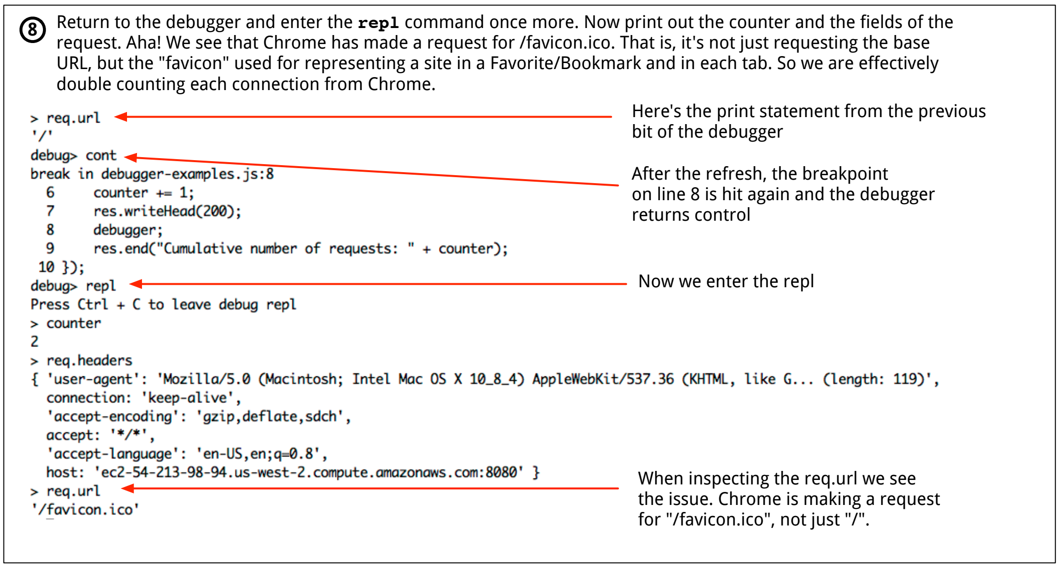

Figure 10: *Within the debugger again, we again do the* `repl` *command and look at the counter. It's only incrementing by 1, to the value 2. So what's going on? Aha, if we look at the* `req` *data structure in more detail, the* `req.url` *field is* `/favicon.ico` *rather than /. So two HTTP requests are being initiated rather than one by Chrome. And that is happening very fast and imperceptibly, so it seems like the counter is increasing by two each time.*

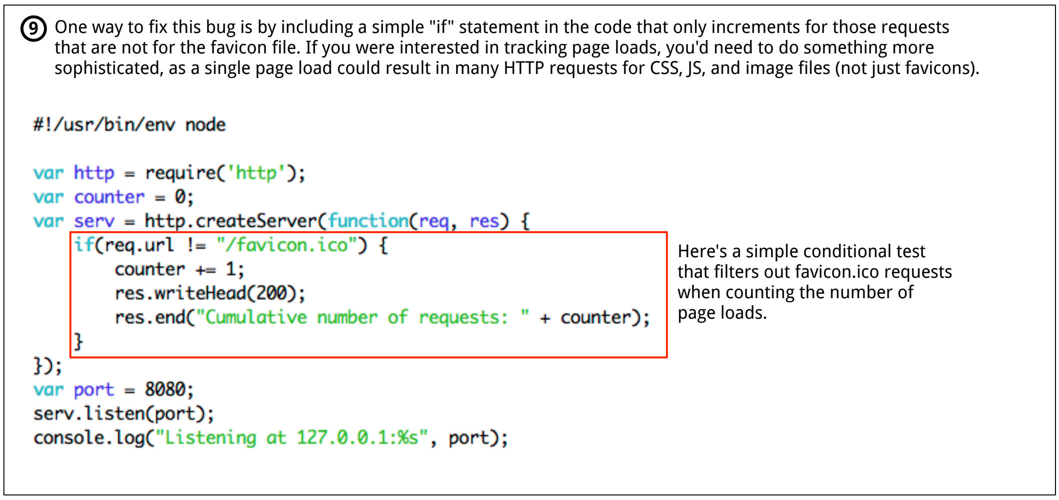

Figure 11: *Now that we understand what is causing the bug, we can address it by including a simple* `if` *statement in our code.*

This simple example illustrates several things:

- How to create breakpoints by including `debugger` statements
- How to debug a network application by initiating a request and jumping into the function that is processing that request to generate a response on the other side
- The kinds of issues that arise in web applications (often they can be reduced to an "unexpected response for this request")

Now, you can get much more sophisticated in terms of debugging.

1. First, you don't actually need to include `debugger` statements in the source code, as you can create and delete breakpoints in the debugger via `setBreakpoint/sb` and `clearBreakpoint/cb`.
2. Second, you can combine the use of the `debugger` with logging statements, e.g. via console.log.
3. Third, you can use something like the node-inspector module for accomplishing many of the same tasks in a web interface (you may find the command line debugger we just reviewed to be faster for most purposes).
4. Fourth, you can use Eclipse or WebStorm if you want a fancier UI than the command line debugger.
5. Finally, and perhaps most importantly, you can define a set of custom debugging functions like the `save` and `log` functions as we did in the list-stable-modules.js script. Any reasonably sophisticated application should have quite a few of these custom debugging tools, which you want to co-develop along with your data structures and functions.

Overall, the `debugger` command and `console.log` along with some custom functions should be sufficient for most backend debugging. The advantage of command line debugging combined with logging is its extreme speed and portability, but do feel free to experiment with these other options.

Summary and Recap

To summarize, in this lecture we went through the concept of latency, looked at sample code to understand the importance of async vs. sync coding, looked at a medium-size script to get a sense of how a command line script involving asynchrony could be developed, and finally got to learn about several techniques to debug Node code.