Node.js

A deep dive

Index

Chapter I

Presentation

Pedro Teixeira

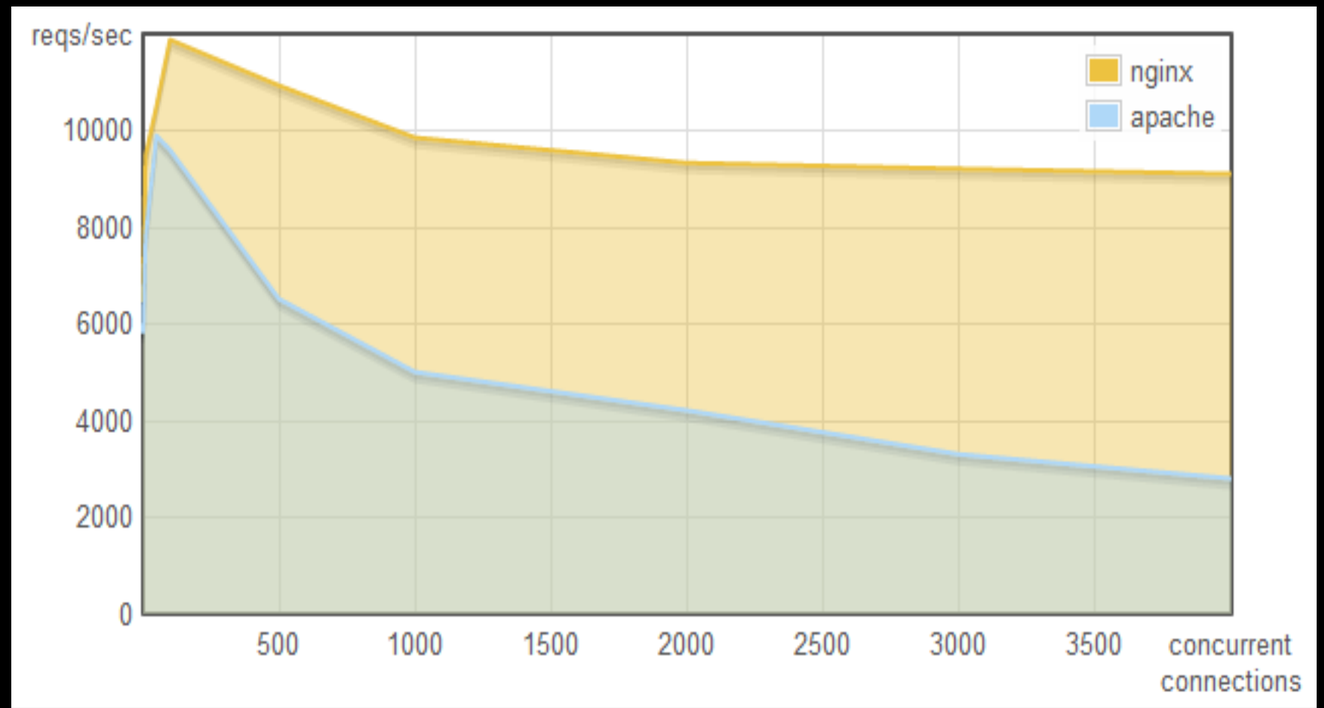
metaduck.com
http://github.com/pgte
@pedrogteixeira
pedro.teixeira@gmail.com

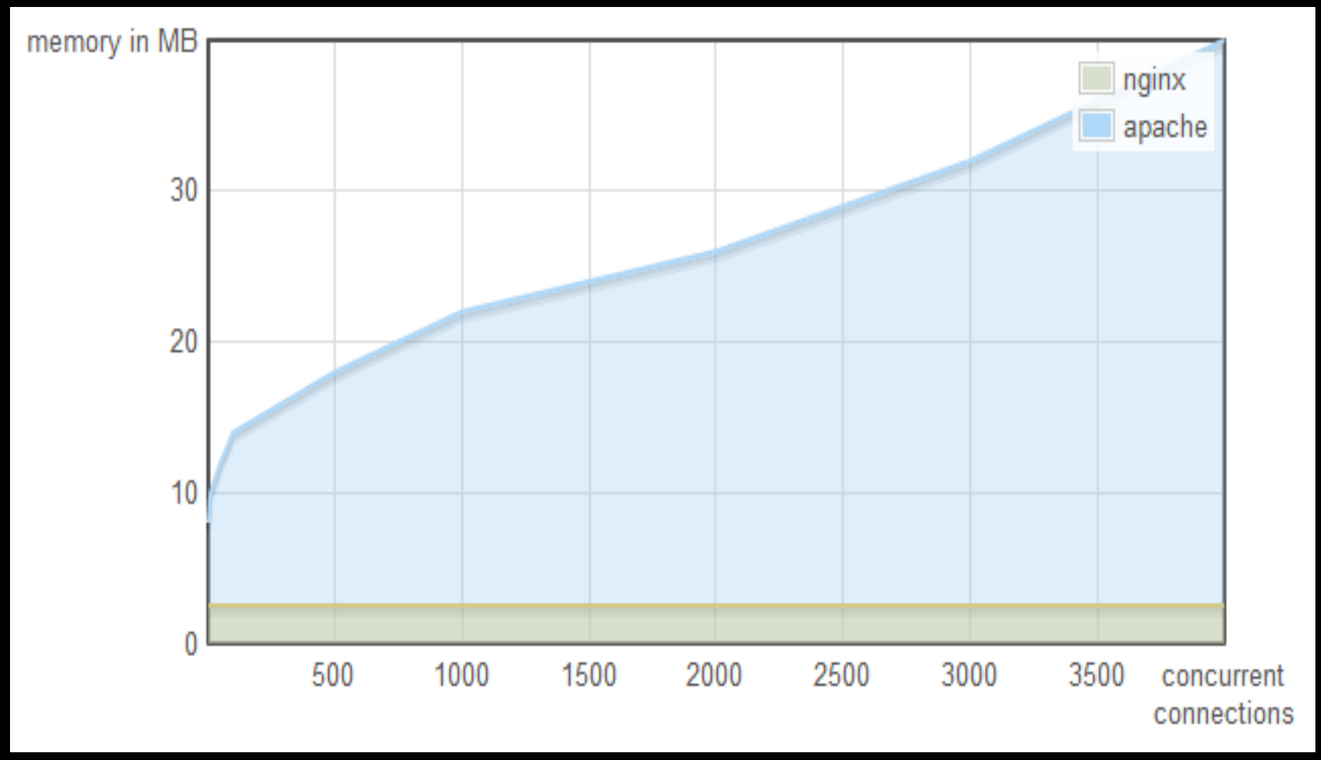
Chapter II

Motivation

Node.js enables you to write high-performance servers **easily.**

Node.js is an evented I/O framework for the V8 Javascript engine


reqs/sec
10000
8000
6000
4000
2000
0
nginx
apache
500
1000
1500
2000
2500
3000
3500
concurrent
connections


memory in MB
30
20
10
0
nginx
apache
500
1000
1500
2000
2500
3000
3500
concurrent
connections

I'm a Ruby / Rails developer

Managing ruby processes

... requires some expertise and kung fu

Ok, no more funny images...

There is passenger (mod_rails), but

There must be a better way

But we lack the programming infra-structure...

```
post = Post.find(1);
render(post)...
```

```
Post.find(1, function(post) {
  render(post)...
})
```

So, what is blocking?

- L1: 3 cycles
- L2: 14 cycles
- RAM: 250 cycles
- Disk: 41_000_000 cycles
- Network: 240_000_000 cycles

Ways around blocking:

- Multi-threading
- Multiplexing

Multi-threading

Threads are bad

- Very complex for the average programmer
- Shared state
- Deadlocks
- Race conditions
- Context switching costs

Threads are good

- When you need CPU-heavy work
- When you need to support multiple core CPUs
- No/almost none shared state

Example: gzipping

Threads are a part of Node.js, but they are hidden from the standard lib!

Chapter III

Back to school

Basic Server

```
1  /* create socket */
2  serverSocket = socket(...);
3
4  /* bind the socket to an address and port */
5  bind(serverSocket, ...);
6
7  /* socket will listen for incoming connections */
8  listen(serverSocket, ...);
9
10 for(;;) {
11
12   /* Wait for client connections */
13   clientSocket = accept(serverSocket, ...);
14
15   /* do something with it */
16
17 }
```

Problem: serializes client requests

One solution: use threads

```
/* create socket */
serverSocket = socket(...);
/* bind the socket to an address and port */
bind(serverSocket, ...);
/* socket will listen for incoming connections */
listen(serverSocket, ...);

listen_fds = [];

listen_fds << serverSocket;

for(;;) {

  /* Listen to all file descriptors in listen_fds */
  active_fds = listen_on_all(listen_fds, ...);
  for_each active_fd in active_fds {

    if (serverSocket == active_fd) {

      new_fd = accept(serverSocket, ....);
      listen_fds << new_fd;

    } else {

      // handle data from a client
      (...)

      // disconnect if needed
      close(active_fd);
      listen_fds.delete(active_fd);

    }
  }
}
```

Back To school

How to make other things while idle?

```
/* create socket */
serverSocket = socket(...);
/* bind the socket to an address and port */
bind(serverSocket, ...);
/* socket will listen for incoming connections */
listen(serverSocket, ...);

listen_fds = [];

listen_fds << serverSocket;

timeout = 10ms;

for(;;) {

  /* Listen to all file descriptors in listen_fds */
  active_fds = listen_on_all(listen_fds, timeout);
  if(active_fds.count > 0) {
    for_each active_fd in active_fds {

      if (serverSocket == active_fd) {

        new_fd = accept(serverSocket, (...));
        listen_fds << new_fd;

      } else {

        // handle data from a client
        (...)

        // disconnect if needed
        close(active_fd);
        listen_fds.delete(active_fd);

      }
    }
  } else { // if(active_fds.count > 0)
    // Timeout
    // do some more stuff
    do_some_other_stuff(...)
  }
}
```

Reactor pattern

- Generalization and improvement of the last example:
  - I/O Resources
  - Synchronous Event Demultiplexer (select, poll, etc. as back-ends)
  - Dispatcher (figuring out which handler to call)
  - Request Handler - Watcher => app

Why not use Event Machine (Ruby), Twisted (Python) or AnyEvent (Perl)?

On other frameworks

User is aware of event loop

On other frameworks

It's hard to know how to write a non-blocking server (is this lib call not blocking?)

On Node.js

User is inside a Jail.

Node isolates user from multi-threading programming or blocking functions.

Javascript is geared torwards event-driven programming

Viva la web!

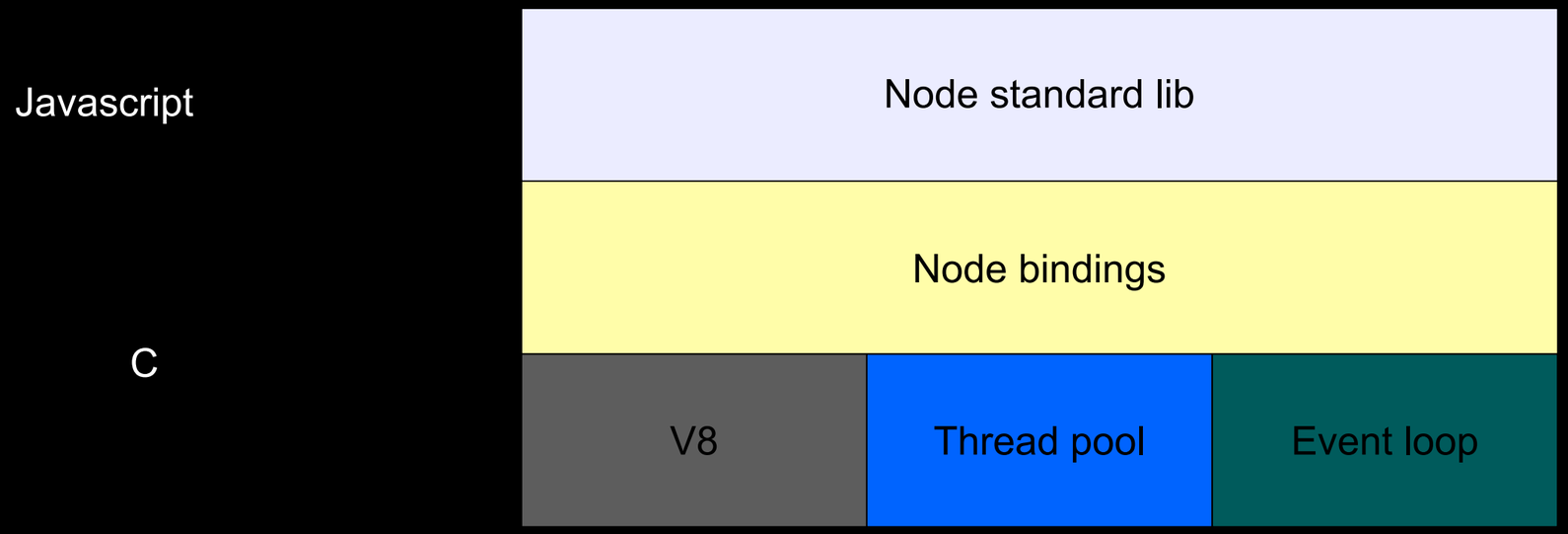
Javascript helps because Javascript has:

- Anonymous functions
- Closures

Chapter IV

Under the hood

Architecture



V8

Extremely performant Javascript VM developed by Google

libev

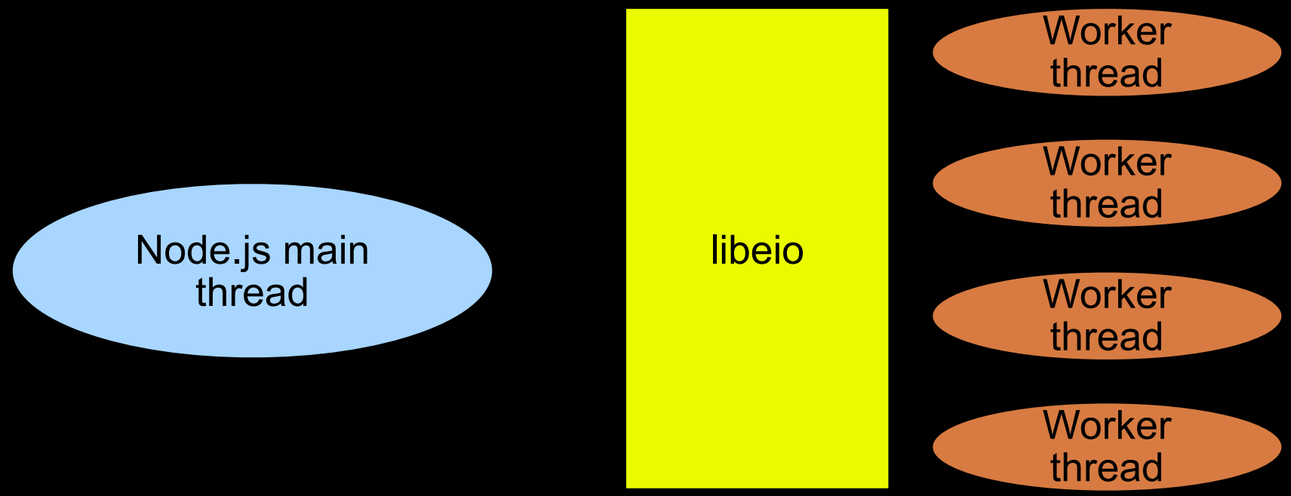
"a high performance full-featured event loop written in C".

Listening on multiple file descriptors using epoll, kqueue, /dev/poll, or select.

libeio

Solves the problem of how to transform a blocking function into a non-blocking function by using worker threads.

libeio


Node.js main thread
libeio
Worker thread
Worker thread
Worker thread
Worker thread
Worker thread

libeio

Writes are deferred to worker threads

- libeio - “truly asynchronous POSIX I/O”
- Node wraps and “saves” V8 javascript callbacks and sends requests to libeio
- Libeio executes each request asynchronously (using a worker thread pool) and at the end notifies main thread to execute callback

Node standard lib

It's a Jail

“Everyone thinks they are expert programmers, but they're not”

- Ryan Dahl

Node standard lib

You're in control!

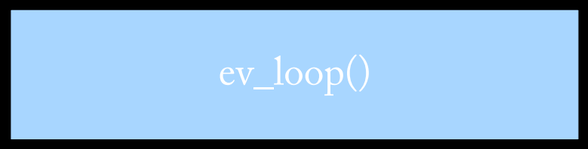
You are given a low-level (POSIX) API

Chapter V

The execution stack

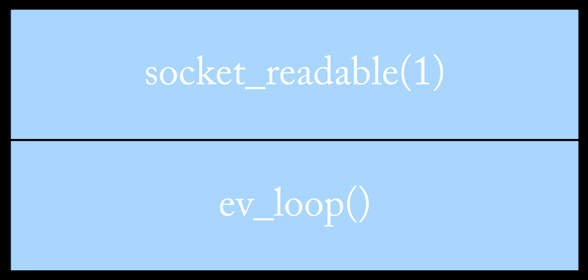
The execution stack

Idle



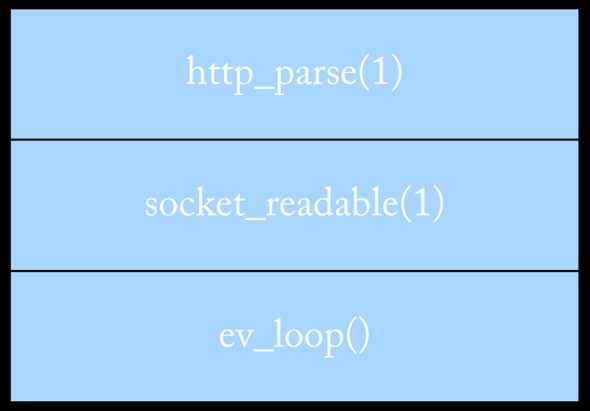
The execution stack

Client connects



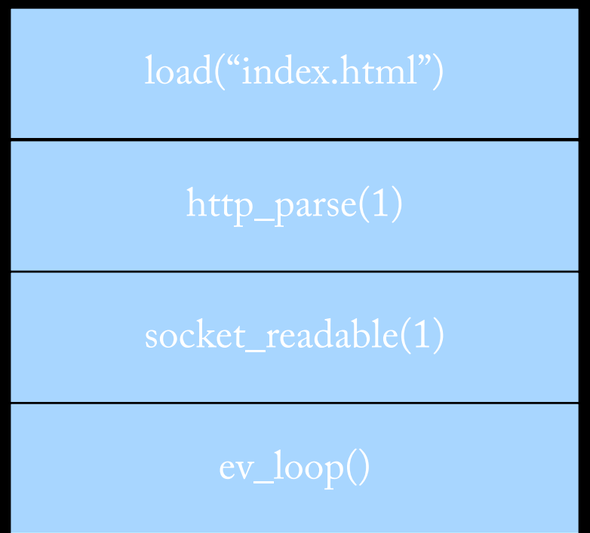
The execution stack

Parse client request



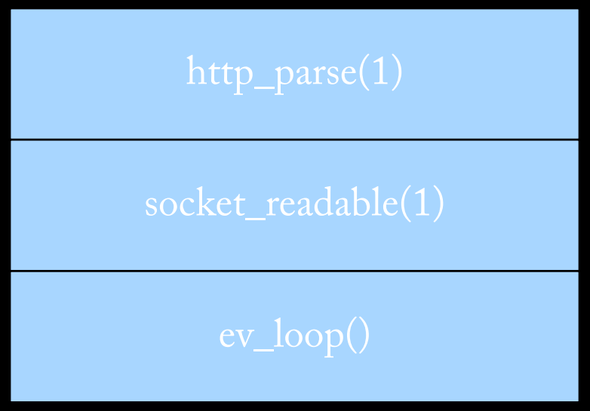
The execution stack

Request file - ASYNC



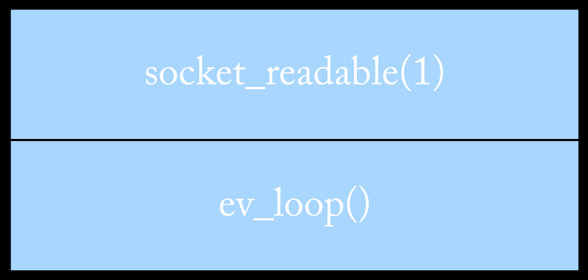
The execution stack

Stack unwinds



The execution stack

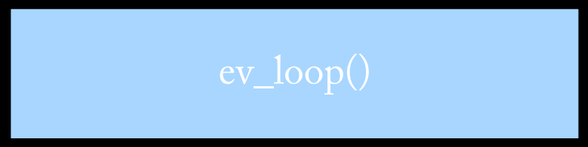
Stack unwinds



The execution stack

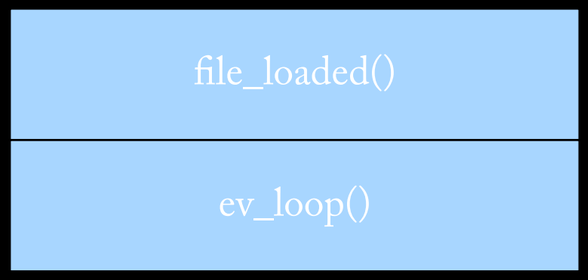
Back to the event loop

Meanwhile, there is a thread
doing the read work for us
during which **we serve other clients**



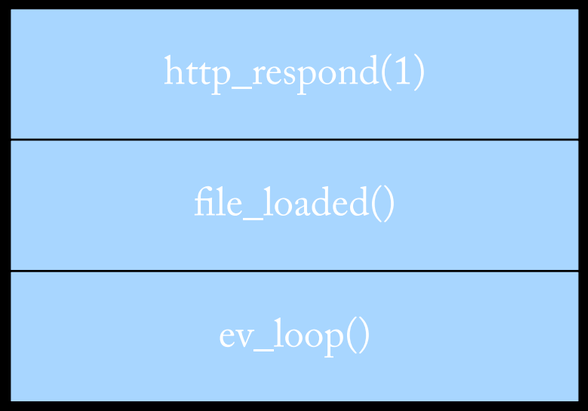
The execution stack

Much much much later,
the file finished to load



The execution stack

We respond



Evented I/O is not magic scalability pixie dust, and like anything, there is a tradeoff.

Be careful. Never block.

Callbacks have to return fast.

(Iterating over a 100000 result set IS BLOCKING)

If you are planning on doing some CPU-intensive work like parsing large amounts of data, consider alternatives or use something like workers.

Chapter VI

The API

Modules are CommonJS goodness

The API

- Buffers – handles binary data (JS sucks at it)
- Process
  - Exit event, chdir, environment, exit, pid, kill, etc.
  - Nexttick – execute function next time around the loop
- Sys – print, log, debug, etc.
- Timers – typical JS timers
- Fs – File system
  - Open, close, Read, write, etc.

fs (potentially bad) example

```
1  fs.readFile('/etc/passwd', function (err, data) {
2    if (err) throw err;
3    console.log(data);
4  });
```

The API

- HTTP
  - http.Server – events 'connection', 'request'
  - http.ServerRequest – events 'data' and 'end'
  - http.ServerResponse – writeHead, write, end

HTTP - example

```
1  var http = require('http');
2  http.createServer(function (req, res) {
3    res.writeHead(200, {'Content-Type': 'text/plain'});
4    res.end('Hello World\n');
5  }).listen(8124, "127.0.0.1");
6  console.log('Server running at http://127.0.0.1:8124/')
```

The API

- net.Server
- net.Stream – a duplex stream interface
  - Used for clients and servers
  - Connect, read, write, end, 'drain' event

net - examples

```
1  var net = require('net');
2  net.createServer(function (socket) {
3    socket.setEncoding("utf8");
4    socket.write("Echo server\r\n");
5    socket.on("data", function (data) {
6      socket.write(data);
7    });
8    socket.on("end", function () {
9      socket.end();
10   });
11 }).listen(8124, "127.0.0.1");
```

The API

- dgram – Datagram socket

What about writes?

Don't they block?

- Node.js queues writes for you and flushes them on idle time
- But...watch out for memory consumption
- It's better if you use the Streams API
- Listen to the “drain” event

The API

- Streams – abstract API
  - WriteableStream
    - Write and it tries to flush to kernel
    - If not (write returns false), it eventually emits 'drain' events, and then you can write again
  - ReadableStream – emmits 'data', 'error', 'end' and 'close' events

The API

sys.pump – Experimental

sys.pump(readableStream, writeableStream, [callback])

Read the data from readableStream and send it to the writableStream. When writeableStream.write(data) returns false readableStream will be paused until the drain event occurs on the writableStream. callback is called when writableStream is closed.

Example – a streaming server using sys.pump

```
1  var http = require('http');
2  var fs = require('fs');
3  var sys = require('sys');
4
5  http.createServer(function (request, response) {
6    var filename = 'test.mov';
7    fs.stat(filename, function(err, stats) {
8      response.writeHead(200, {
9        'Content-Type': 'video/quicktime',
10       'Content-Length': stats.size
11     });
12     var read_stream = fs.createReadStream(filename);
13     sys.pump(read_stream, response, function() {
14       response.end();
15     });
16
17   })
18
19 }).listen(8124);
```

a better streaming server using sys.pump

```
1  var http = require('http');
2  var fs = require('fs');
3  var sys = require('sys');
4
5  var filename = 'test.mov';
6  var content_type = 'video/quicktime';
7
8  fs.stat(filename, function(err, stats) {
9    http.createServer(function (request, response) {
10       response.writeHead(200, {
11         'Content-Type': content_type,
12         'Content-Length': stats.size
13       });
14       sys.pump(fs.createReadStream(filename), response, function() {
15         response.end();
16       });
17
18   }).listen(8124);
19 })
20
```

An even better streaming server using sys.pump

```
1  var http = require('http');
2  var fs = require('fs');
3  var sys = require('sys');
4
5  var filename = 'test.mov';
6  var content_type = 'video/quicktime';
7
8  http.createServer(function (request, response) {
9      response.writeHead(200, {
10       'Content-Type': content_type
11     });
12     sys.pump(fs.createReadStream(filename), response);
13
14 }).listen(8124);
```

Chapter VII

Cool stuff

Cool trick in “UNIX” streams

- You can pass in a file descriptor
- Good way of passing connections around – load balancing, fail-over, etc.
- Read it using a “fd” event

Process spawn example

```
1  var sys   = require('sys'),
2      spawn  = require('child_process').spawn,
3      http  = require('http');
4
5  http.createServer(function (request, response) {
6    response.writeHead(200, {'Content-Type': 'text/plain'});
7    var cat = spawn('tail', ['-f', '/var/log/system.log']);
8    cat.stdout.on('data', function(data) {
9      response.write(data);
10   });
11 }).listen(8124);
```

Better process spawn example

```
1  var sys   = require('sys'),
2      spawn  = require('child_process').spawn,
3      http  = require('http');
4
5  http.createServer(function (request, response) {
6    response.writeHead(200, {'Content-Type': 'text/plain'});
7    var tail = spawn('tail', ['-f', '/var/log/system.log']);
8
9    var kill_tail = function() {
10     tail.kill();
11   }
12   process.on('exit', kill_tail);
13
14   request.connection.on('end', kill_tail);
15
16   tail.stdout.on('data', function(data) {
17     response.write(data);
18   });
19 }).listen(8124);
```

multiple parallel external calls

```
var db1 = initialize_connection_1();
var db2 = initialize_connection_2();
http.createServer(function (request, response) {
  all_results = [];
  var try_reply = function(result) {
    all_results.push(result);
    if (all_results.length == 2) {
      response.end(render(all_results));
    }
  }
  db1.query('select * from transactions where ...', function(results) {
     try_reply(results);
  });
  db2.query('select * from profiles where ...', function(results) {
    try_reply(results);
  });

}).listen(8124);
```

... with timeout

```
1  var db1 = initialize_connection_1();
2  var db2 = initialize_connection_2();
3  http.createServer(function (request, response) {
4    all_results = [];
5
6    var try_reply = function(result, force) {
7      all_results.push(result);
8      if (all_results.length == 2 || force) {
9        response.end(render(all_results));
10       clearTimeout(timeout);
11     }
12   }
13
14   var timeout = setTimeout(function() {
15     try_reply(null, true);
16   }, 500);
17
18
19   db1.query('select * from transactions where ...', function(results) {
20      try_reply(results);
21   });
22   db2.query('select * from profiles where ...', function(results) {
23     try_reply(results);
24   });
25
26 }).listen(8124);
```

... the right way

```
var db1 = initialize_connection_1();
var db2 = initialize_connection_2();
http.createServer(function (request, response) {
  var all_results = [];
  var replied = false;

  var try_reply = function(result, force) {
    all_results.push(result);
    if (!replied && (all_results.length == 2 || force)) {
      replied = true;
      response.end(render(all_results));
      clearTimeout(timeout);
    }
  }

  var timeout = setTimeout(function() {
    try_reply(null, true);
  }, 500);


  db1.query('select * from transactions where ...', function(results) {
     try_reply(results);
  });
  db2.query('select * from profiles where ...', function(results) {
    try_reply(results);
  });

}).listen(8124);
```

Step

```
1  var Step = require('step');
2
3  Step(
4    function readSelf() {
5      fs.readFile(__filename, this);
6    },
7    function capitalize(err, text) {
8      if (err) {
9        throw err;
10     }
11     return text.toUpperCase();
12   },
13   function showIt(err, newText) {
14     sys.puts(newText);
15   }
16 );
```

Chapter VII

Cool stuff

multiple parallel external calls

```
var db1 = initialize_connection_1();
var db2 = initialize_connection_2();
http.createServer(function (request, response) {
  all_results = [];
  var try_reply = function(result) {
    all_results.push(result);
    if (all_results.length == 2) {
      response.end(render(all_results));
    }
  }
  db1.query('select * from transactions where ...', function(results) {
     try_reply(results);
  });
  db2.query('select * from profiles where ...', function(results) {
    try_reply(results);
  });

}).listen(8124);
```

... with timeout

```
1  var db1 = initialize_connection_1();
2  var db2 = initialize_connection_2();
3  http.createServer(function (request, response) {
4    all_results = [];
5
6    var try_reply = function(result, force) {
7      all_results.push(result);
8      if (all_results.length == 2 || force) {
9        response.end(render(all_results));
10       clearTimeout(timeout);
11     }
12   }
13
14   var timeout = setTimeout(function() {
15     try_reply(null, true);
16   }, 500);
17
18
19   db1.query('select * from transactions where ...', function(results) {
20      try_reply(results);
21   });
22   db2.query('select * from profiles where ...', function(results) {
23     try_reply(results);
24   });
25
26 }).listen(8124);
```

... the right way

```
var db1 = initialize_connection_1();
var db2 = initialize_connection_2();
http.createServer(function (request, response) {
  var all_results = [];
  var replied = false;

  var try_reply = function(result, force) {
    all_results.push(result);
    if (!replied && (all_results.length == 2 || force)) {
      replied = true;
      response.end(render(all_results));
      clearTimeout(timeout);
    }
  }

  var timeout = setTimeout(function() {
    try_reply(null, true);
  }, 500);


  db1.query('select * from transactions where ...', function(results) {
     try_reply(results);
  });
  db2.query('select * from profiles where ...', function(results) {
    try_reply(results);
  });

}).listen(8124);
```

Cool trick in “UNIX” streams

- You can pass in a file descriptor
- Good way of passing connections around – load balancing, fail-over, etc.
- Read it using a “fd” event

Step

```
1  var Step = require('step');
2
3  Step(
4    function readSelf() {
5      fs.readFile(__filename, this);
6    },
7    function capitalize(err, text) {
8      if (err) {
9        throw err;
10     }
11     return text.toUpperCase();
12   },
13   function showIt(err, newText) {
14     sys.puts(newText);
15   }
16 );
```

Chapter VII

*Do*s and *Don't*s

Memory consumption on writes

Callbacks are called some time in the far and distant future.

```
1  var filename = "log.txt";
2  fs.open(filename, 'r', 0666, function(err, fd) {
3    sys.log('opened file '+filename);
4  });
5  filename = "/etc/passwd";
```

```
1  var filename = "log.txt";
2  (function(filename) {
3    fs.open(filename, 'r', 0666, function(err, fd) {
4      sys.log('opened file '+filename);
5    });
6  })(filename);
7  filename = "/etc/passwd";
```

Don't insert a require inside callbacks

CommonJS *require* function is blocking,
don't do them inside callbacks!

```
1  fs.open('/etc/passwd', 'r', 0666, function(err, fd) {
2    http = require('http');
3    http.createClient(80, 'www.myserver.net');
4    // ...
5  });
6
```

Error handling

Catching exceptions

Event-driven programming exception handling is hard.

Exceptions happen during callback.

Exception are caught by event loop.

Try-catch won't work across callbacks

Some APIs provide an error callback when the API can produce an error.

Others provide the error on the callback (like fs.read)

Example:

Error we get trying to connect with Redis

```
ErrorECONNREFUSED, Connection refusedError:
  ECONNREFUSED, Connection refused
at IOWatcher.callback (net:854:22)
at node.js:768:9
```

```
  function f() {
  throw new Error('foo');
}
setTimeout(f, 10000*Math.random());
setTimeout(f, 10000*Math.random());

$ node x.js
x.js:2
throw new Error('foo');
^
Error: foo
at Timer.f [as callback] (x.js:2:9)
at node.js:266:9
```

Not all is sunshine in the world of callbacks:
From which line does the error arise?

Catch all

```
process.on('uncaughtException', function (err) {
  console.log('Caught exception: ' + err);
});
```

Stream error

Example: net.Stream 'error' event

debugging

- sys.inspect() - useful
- node_debug- HTTP based console and object explorer for node.js

Chapter VIII

A case study

fugue - Unicorn for node.js

The feature set is similar, but:

Lines of code:

Unicorn: 1537

Fugue: 342 (~4.5 times smaller)

fugue - Unicorn for node.js

Node.js really helps you with:

- Managing and passing around multiple file descriptors
- Nice idiom for reacting to events
- Easy to setup a server or a client

(proper) javascript really helps you with:

- easily keeping state inside callbacks.
- no more observers, listeners, watchers. just function closures.

Chapter IX

Final word

Infra-structure

Think about wasted resources.

Thoeoretically, if the main thing you are doing is I/O and you haven't saturated your box's bandwidth, you can live with one Node.js process.

Some Modules and Frameworks

- Connect – Rack-like framework for node.js HTTP servers
- Express – A sinatra-inspired web development framework
- Socket.IO - ['websocket', 'server-events', 'flashsocket', 'htmlfile', 'xhr-multipart', 'xhr-polling']
- Node-xmpp - an idiomatic XMPP library for Node.js