node.js

ry@tinyclouds.org

May 5, 2010

node.js is a set of bindings to the V8 javascript VM.

Allows one to script programs that do I/O in javascript.

Focused on performance.

First a silly benchmark:

```
100 concurrent clients
1 megabyte response

node     822 req/sec
nginx    708
thin      85
mongrel    4

(bigger is better)
```

The code:

```
1  http = require('http')
2  Buffer = require('buffer').Buffer;
3
4  n = 1024*1024;
5  b = new Buffer(n);
6  for (var i = 0; i<n; i++) b[i] = 100;
7
8  http.createServer(function (req, res) {
9    res.writeHead(200);
10   res.end(b);
11 }).listen(8000);
```

Please take with a grain of salt. Very special circumstances.

NGINX peaked at 4mb of memory

Node peaked at 60mb.

I/O needs to be done differently.

Many web applications have code like this:

```
result = query('select * from T');
// use result
```

What is the software doing while it queries the database?

In many cases, just waiting for the response.

Modern Computer Latency

L1: 3 cycles

L2: 14 cycles

RAM: 250 cycles

DISK: 41,000,000 cycles

NETWORK: 240,000,000 cycles

“Non-blocking”

L1, L2, RAM

“Blocking”

DISK, NETWORK

```
result = query('select * from T');
// use result
```

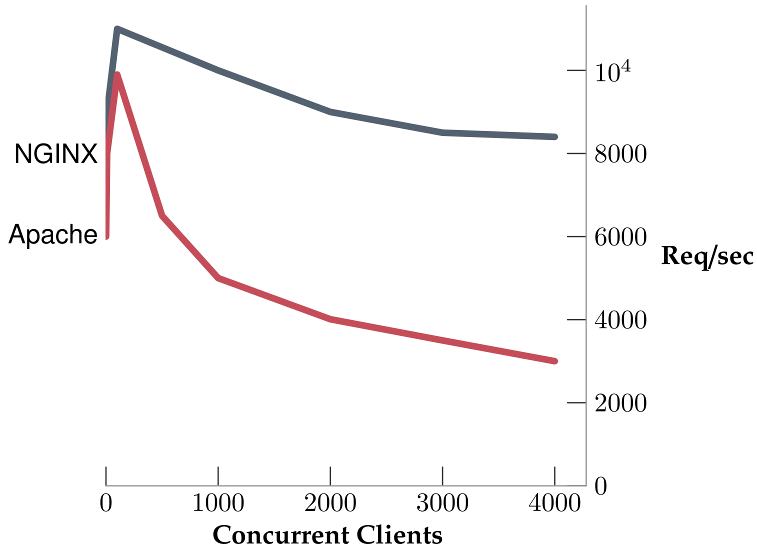
BLOCKS

Better software can multitask.

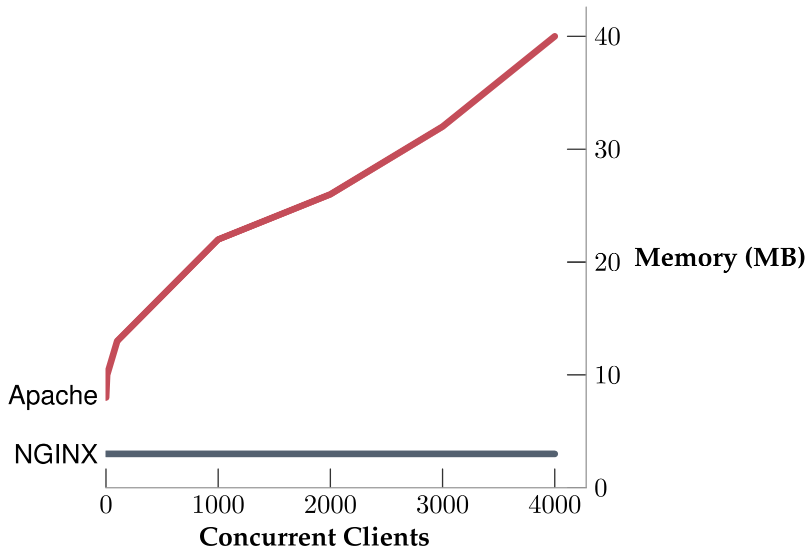
Other threads of execution can run while waiting.

Is that the best that can be done?

A look at Apache and NGINX.



http://blog.webfaction.com/a-little-holiday-present



http://blog.webfaction.com/a-little-holiday-present

Apache vs NGINX

The difference?

Apache uses one thread per connection.

NGINX doesn't use threads. It uses an **event loop**.

- Context switching is not free
- Execution stacks take up memory

For massive concurrency, **cannot** use an OS thread for each connection.

Green threads or coroutines can improve the situation dramatically

BUT there is still machinery involved to create the **illusion** of holding execution on I/O.

Code like this

```
result = query('select..');
// use result
```

either **blocks the entire process** or implies **multiple execution stacks**.

But a line of code like this

```
query('select..', function (result) {
  // use result
});
```

allows the program to return to the event loop immediately.

No machinery required.

```
query('select..', function (result) {
  // use result
});
```

This is how I/O should be done.

So why isn't everyone using event loops, callbacks, and non-blocking I/O?

For reasons both **cultural** and **infrastructural**.

Cultural Bias

We're taught I/O with this:

```
1 puts("Enter your name: ");
2 var name = gets();
3 puts("Name: " + name);
```

We're taught to demand input and do nothing until we have it.

Cultural Bias

Code like

```
1 puts("Enter your name: ");
2 gets(function (name) {
3   puts("Name: " + name);
4 });
```

is rejected as too complicated.

Missing Infrastructure

So why isn't everyone using event loops?

Single threaded event loops require I/O to be non-blocking

Most libraries are not.

Missing Infrastructure

- POSIX async file I/O not available.
- Man pages don't state if a function will access the disk. (e.g `getpwuid()`)
- No closures or anonymous functions in C; makes callbacks difficult.
- Database libraries (e.g. `libmysql_client`) do not provide support for asynchronous queries
- Asynchronous DNS resolution not standard on most systems.

Too Much Infrastructure

EventMachine, Twisted, AnyEvent provide very good event loop platforms.

Easy to create efficent servers.

But users are confused how to combine with other available libraries.

Too Much Infrastructure

Users still require expert knowledge of event loops, non-blocking I/O.

Javascript designed specifically to be used with an event loop:

- Anonymous functions, closures.
- Only one callback at a time.
- I/O through DOM event callbacks.

The culture of Javascript is already geared towards evented programming.

This is the **node.js** project:

To provide a **purely evented, non-blocking infrastructure** to script **highly concurrent** programs.

Design Goals

Design Goals

No function should directly perform I/O.

To receive info from disk, network, or another process **there must be a callback.**

Design Goals

Low-level.

Stream everything; never force the buffering of data.

Do not remove functionality present at the POSIX layer. For example, support half-closed TCP connections.

Design Goals

Have built-in support for the most important protocols:

DNS, HTTP, TLS

(Especially because these are so difficult to do yourself)

Design Goals

Support many HTTP features.

- Chunked encoding
- Pipelined messages
- Hanging requests for comet applications.

Design Goals

The API should be both familiar to **client-side JS programmers** and **old school UNIX hackers**.

Be platform independent.

Design Goals

Simply licensed (Almost 100% MIT/BSD)

Few dependencies

Static linking

Design Goals

Make it enjoyable.

Architecture

JavaScript

Node standard library

C

Node bindings

V8

thread pool

(libeio)

event loop

(libev)

The JavaScript layer can access the main thread.

The C layer can use multiple threads.

Deficiency? No. Feature.

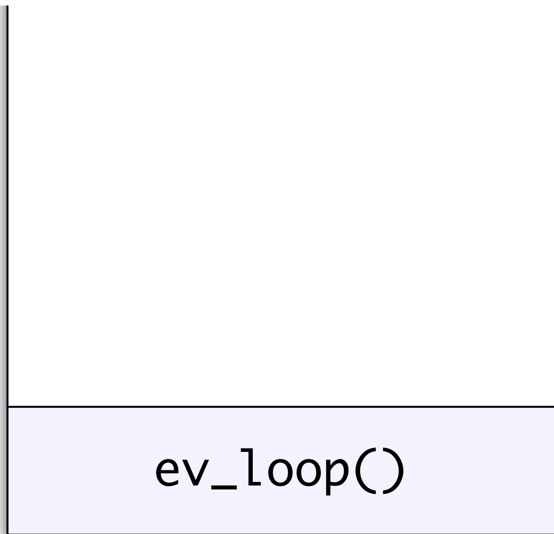
Threads should be used by experts only.

Jail users in non-blocking environment.

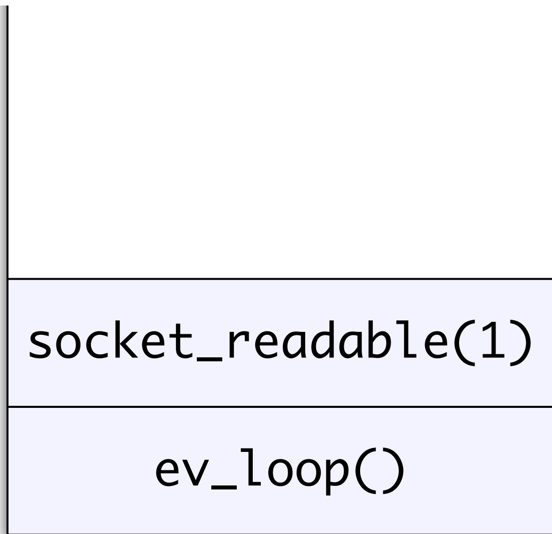
Don't allow I/O trickery.

There is exactly one execution stack in Node.

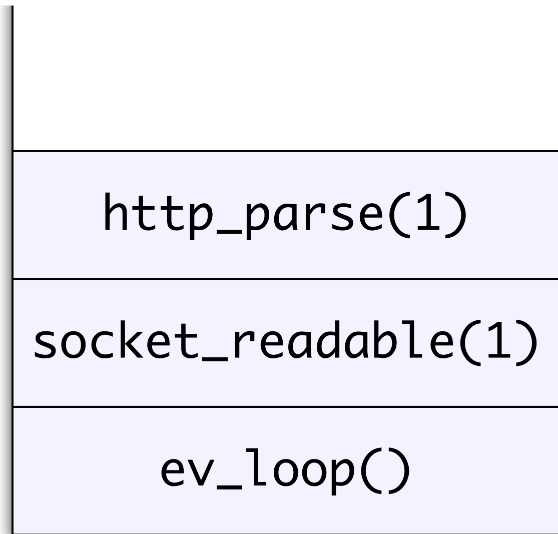
Node execution stack



Node execution stack



Node execution stack



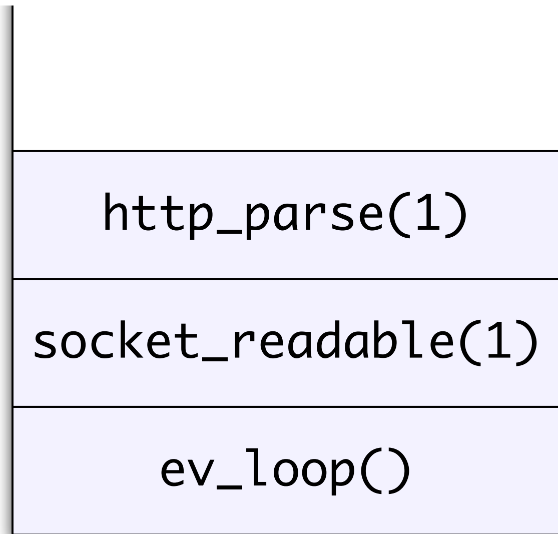
Node execution stack

```
load("index.html")
http_parse(1)
socket_readable(1)
ev_loop()
```

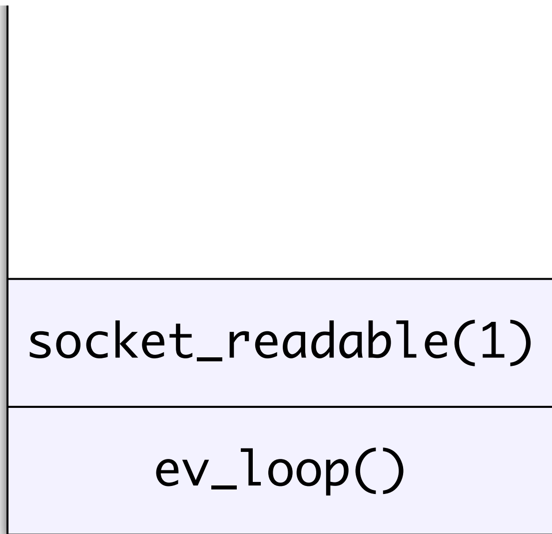
Node sends a request to the thread pool to load **`"index.html"`**.

The stack unwinds...

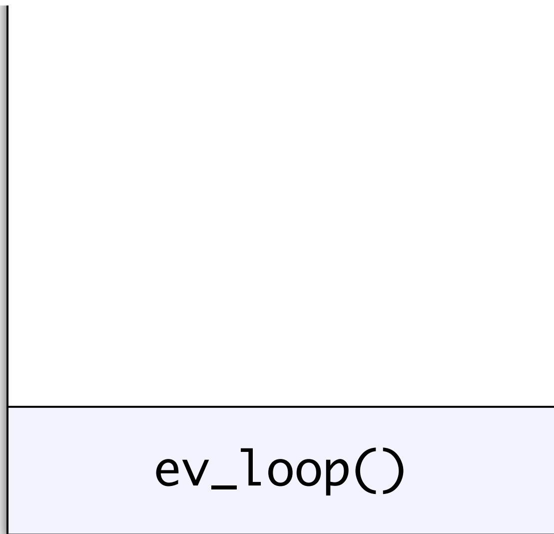
Node execution stack



Node execution stack



Node execution stack



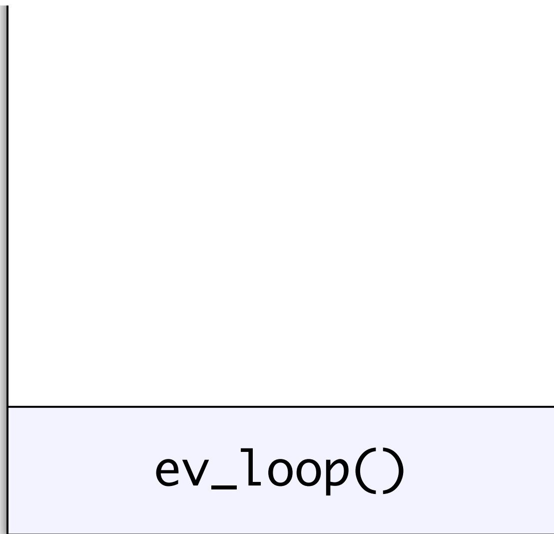
The request is sent to the disk.

Millions of clock cycles will pass before the process hears back from it.

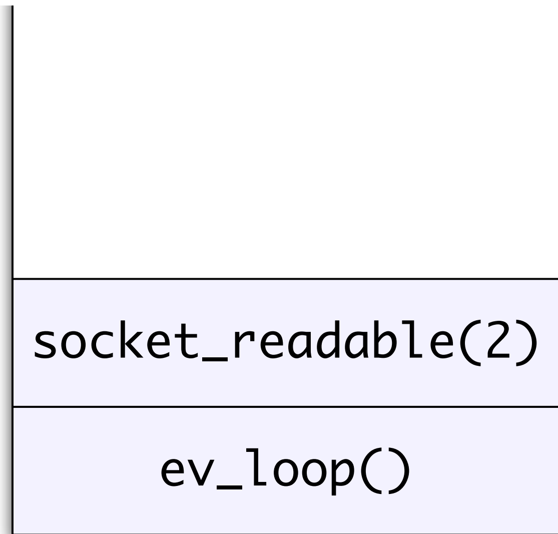
In the meantime someone else connects to the server.

This time requesting an in-memory resource.

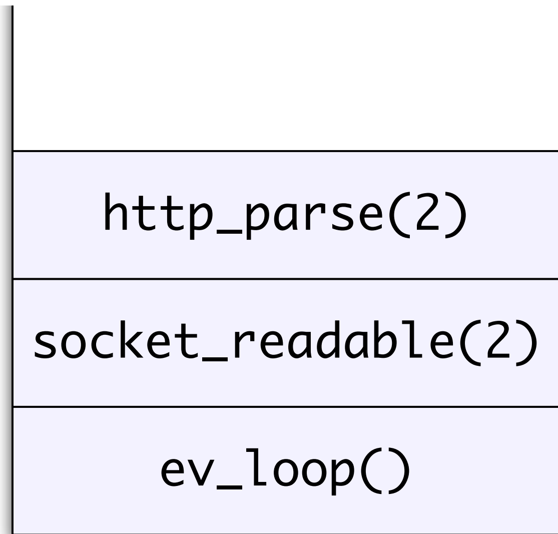
Node execution stack



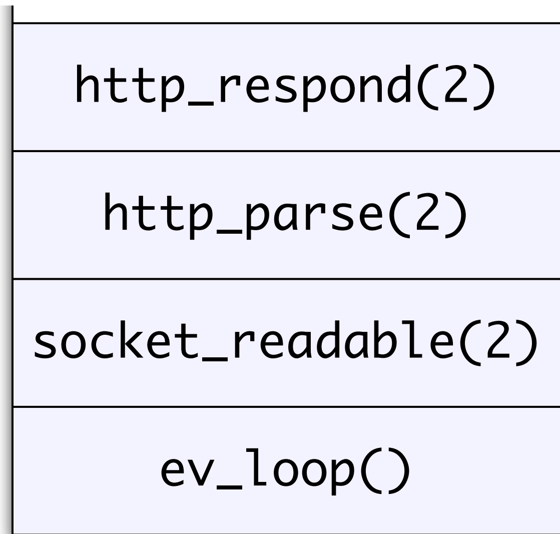
Node execution stack



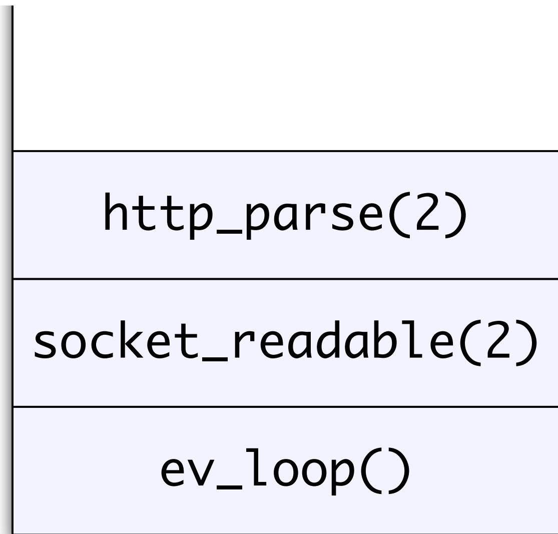
Node execution stack



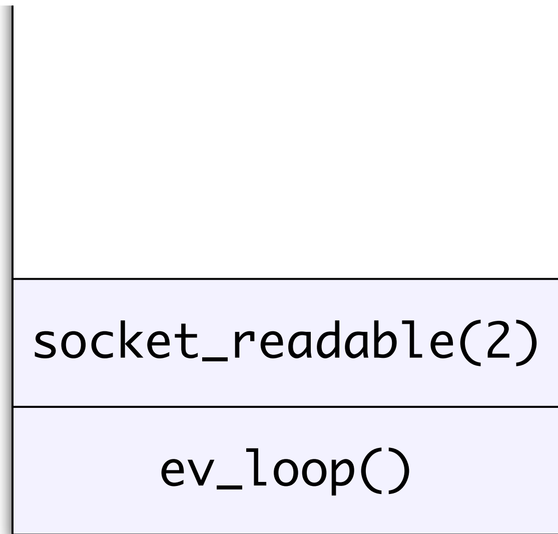
Node execution stack



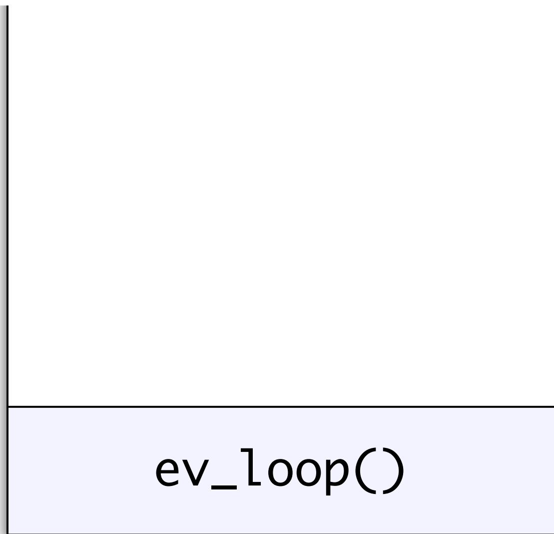
Node execution stack



Node execution stack



Node execution stack

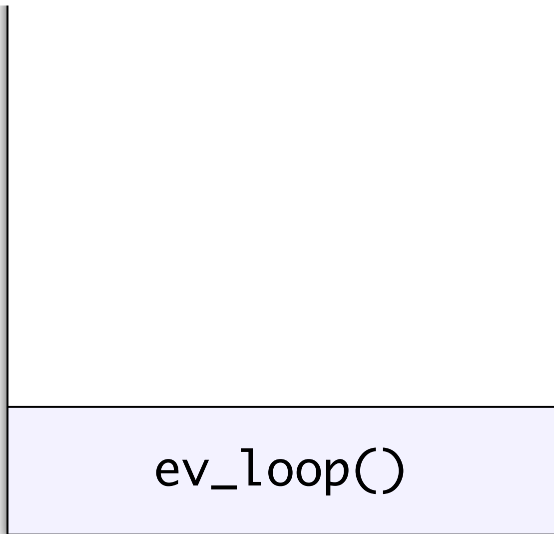


The process sits idle.

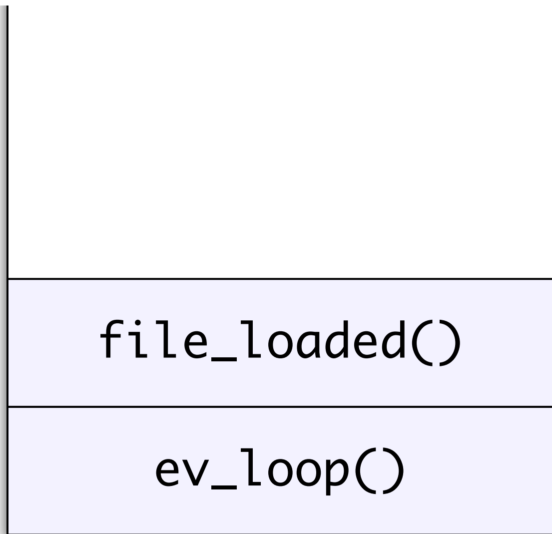
The first request is still hanging.

Eventually the disk responds:

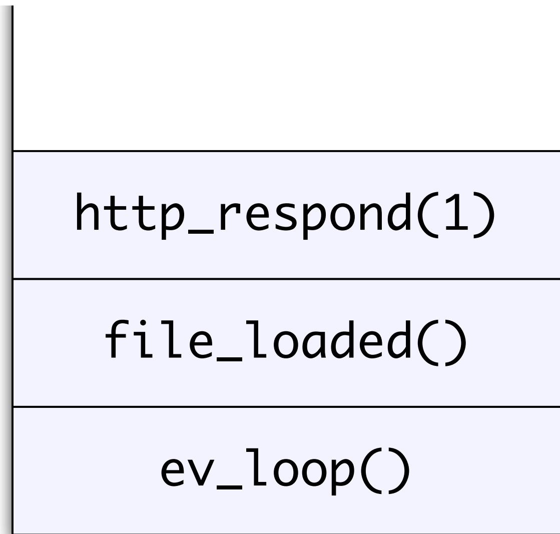
Node execution stack



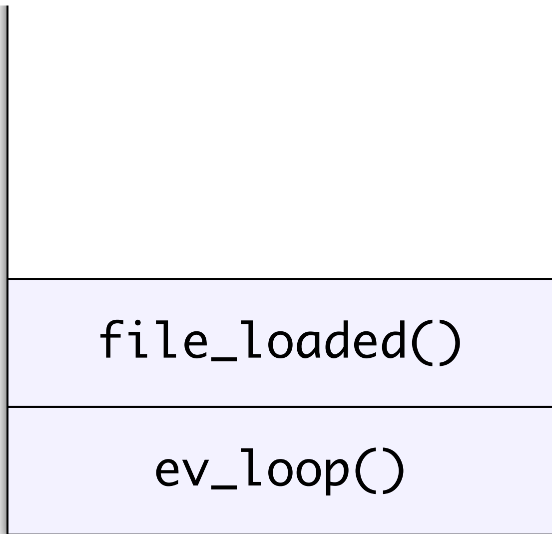
Node execution stack



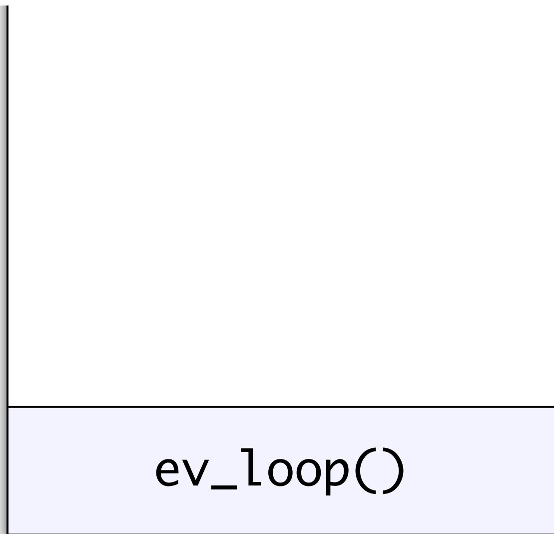
Node execution stack



Node execution stack



Node execution stack



Usage and Examples

(using node 0.1.93)

Download, configure, compile, and **`make install`** it.

`http://nodejs.org/`

No dependencies other than Python for the build system. V8 is included in the distribution.

```
1 var sys = require('sys');
2
3 setTimeout(function () {
4   sys.puts('world');
5 }, 2000);
6 sys.puts('hello');
```

A program which prints “hello”, waits 2 seconds, outputs “world”, and then exits.

```
1 var sys = require('sys');
2
3 setTimeout(function () {
4   sys.puts('world');
5 }, 2000);
6 sys.puts('hello');
```

Node exits automatically when there is nothing else to do.

```
% node hello_world.js
hello
```

2 seconds later...

```
% node hello_world.js
hello
world
%
```

A program which:

- starts a TCP server on port 8000
- send the peer a message
- close the connection

```
1 net = require('net');
2
3 var s = net.createServer();
4 s.addListener('connection',
5       function (c) {
6   c.end('hello!\n');
7 });
8
9 s.listen(8000);
```

File I/O is non-blocking too.

(Something typically hard to do.)

As an example, a program that outputs the last time `/etc/passwd` was modified:

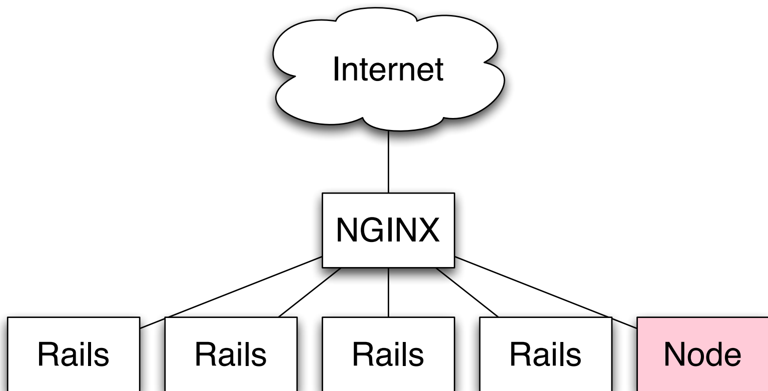
```
var stat = require('fs').stat,
    puts = require('sys').puts;

stat('/etc/passwd', function (e, s) {
  if (e) throw e;
  puts('modified: ' + s.mtime);
});
```

Road Map

Now:

Solution for real-time problems along side traditional stack.

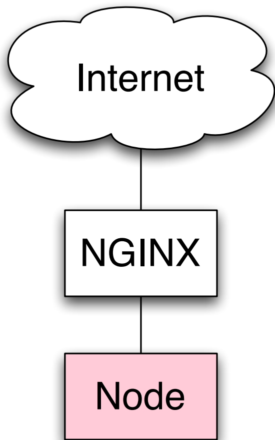
(long poll, WebSocket)


Internet
NGINX
Rails
Rails
Rails
Rails
Node

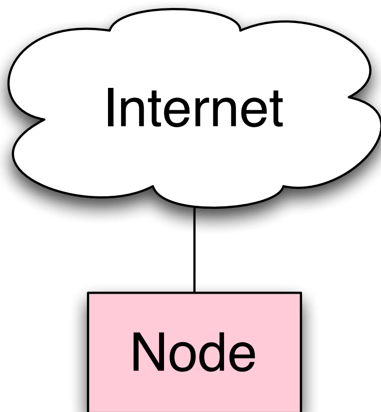
As frameworks built on Node mature, use it for entire websites.

For security set it behind a stable web server.

Load balancing unnecessary.


Internet
NGINX
Node

When Node is stable, remove the front-end web server entirely.



Questions...?

http://nodejs.org/

ry@tinyclouds.org