node.js

ryan@joyent.com

October 14, 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 (aging) benchmark:

```
100 concurrent clients
1 megabyte response

node    822 req/sec
nginx   708
thin     85
mongrel   4

 (bigger is better)
```

The code:

```
http = require('http')

b = new Buffer(1024*1024);

http.createServer(function (req, res) {
  res.writeHead(200);
  res.end(b);
}).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

`http://duartes.org/gustavo/blog/post/what-your-computer-does-while-you-wait`

“Non-blocking”

L1, L2, RAM

“Blocking”

DISK, NETWORK

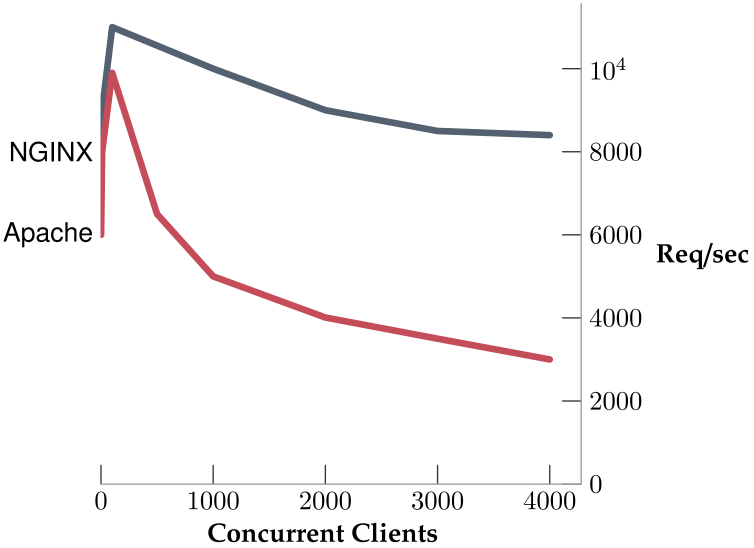
```
result = query('select * from T');
// use result
```

BLOCKS

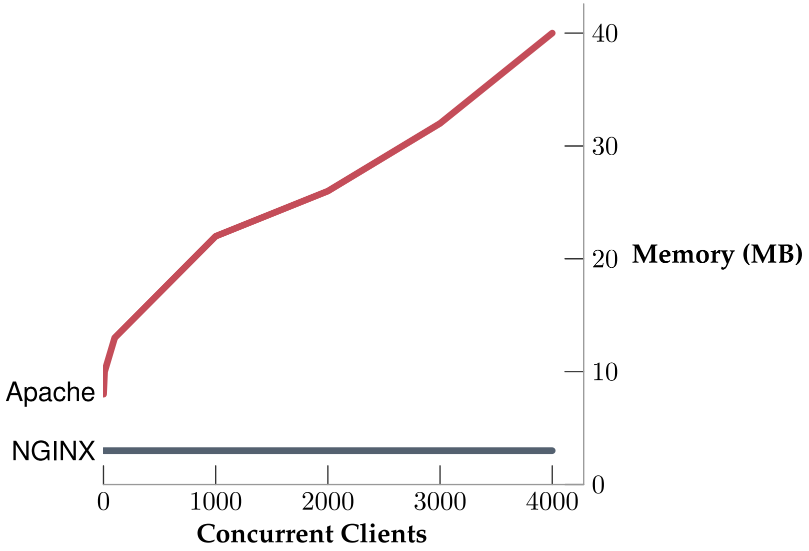
Better software can multitask.

Other threads of execution can run while waiting.

Is that the best that can be done?



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 OS thread** or implies **multiple execution stacks**.

But a line of code like this

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

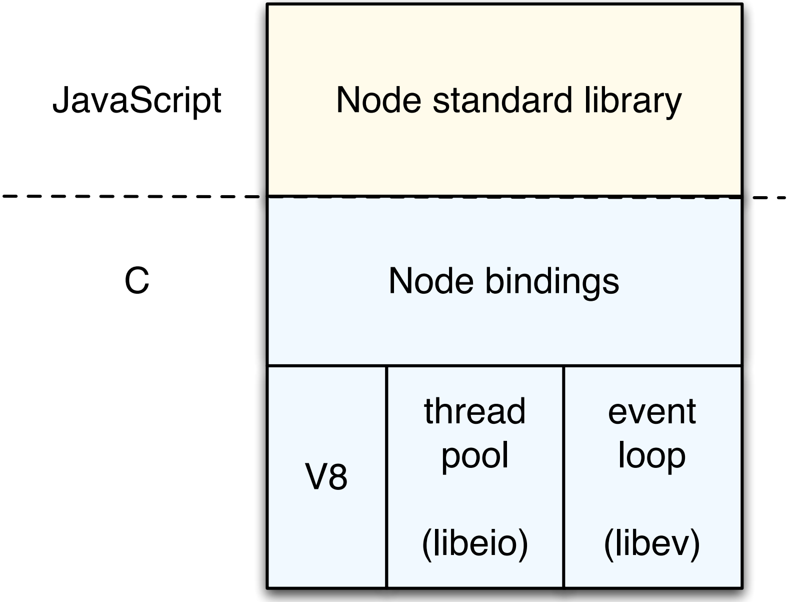
allows the program to return to the event loop immediately.

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

This is how I/O should be done.

- Good **servers** are written with **non-blocking I/O**.
- **Web browsers** are scripted with **non-blocking I/O**.
- Lots of interest in building good JavaScript VMs (Google, Mozilla, Apple, Microsoft)

Build the foundations of a server–allow the rest to be customized.


JavaScript
Node standard library
C
Node bindings
V8
thread pool
(libeio)
event loop
(libev)

Examples

using node 0.2.3

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.

```
setTimeout(function () {
  console.log('world');
}, 2000);
console.log('hello');
```

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

```
setTimeout(function () {
  console.log('world');
}, 2000);
console.log('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
%
```

```
dns = require('dns');

console.log('resolving yahoo.com...');

dns.resolve('yahoo.com', function (err, addresses) {
  if (err) throw err;
  console.log('found: %j', addresses);
});
```

A program which resolves `yahoo.com`

```
% node resolve-yahoo.js
resolving yahoo.com...
found: ['67.195.160.76','69.147.125.65','72.30.2.43',
'98.137.149.56','209.191.122.70']
%
```

Node includes good support for DNS and HTTP out of the box.

A simple HTTP server:

```
http = require('http');

s = http.Server(function (req, res) {
  res.writeHead(200);
  res.end('hello world\n');
});

// Listen on port 8000
s.listen(8000);
```

```
% node http-server.js &
[1] 9120
% curl http://127.0.0.1:8000/
hello world
%
```

Goal:

```
% curl http://127.0.0.1:8000/yahoo.com
query: yahoo.com
['67.195.160.76','69.147.125.65','72.30.2.43',
'98.137.149.56','209.191.122.70']
%
% curl http://127.0.0.1:8000/google.com
query: google.com
['74.125.95.103','74.125.95.104','74.125.95.105',
'74.125.95.106','74.125.95.147','74.125.95.99']
%
```

```javascript
dns = require('dns');
http = require('http');

var s = http.Server(function (req, res) {
  var query = req.url.replace('/','');

  res.writeHead(200);
  res.write('query: ' + query + '\n');

  dns.resolve(query, function (error, addresses) {
    res.end(JSON.stringify(addresses) + '\n');
  });
});

s.listen(8000);
```

- the DNS resolution is async
- the HTTP server “streams” the response.

The overhead for each connection is low so the server is able to achieve good concurrency: it juggles many connections at a time.

Parallel Programming

Node is inherently single threaded. (Arguably JavaScript too.)

Single threaded means that a Node process can only use one core.

```
// Don't do this!
while (true);
```

While Node is good at **concurrency** it's unclear how to run code in **parallel** with it.

How to run code in parallel?

Use processes. Pass messages. Allow the OS scheduler to run.

Node's focus on asynchronous networking makes IPC easy, and thus parallel programming easy.

Web Workers: An HTML5 spec adapted for Node.js

Peter Griess: `http://github.com/pgriess/node-webworker`

Malte Ubl: `http://github.com/cramforce/node-worker`

- Each worker is a real OS process
- Communicate with each other through sockets
- Each process is able to send and receive messages (actor-style)
- Useful to take heavy calculations out of a server

Web Worker Example: Start a new process to calculate Fibonacci numbers:

```
var w = new Worker('fib.js');

w.postMessage({ calculate: 10 });

w.onmessage = function(m) {
  console.log('result: %j', m);
  w.terminate();
};
```

Different problems require different solutions.

Problem: accepting connections on multiple cores.

Solution: Spawn a copy of yourself *N* times, send the server file descriptor to your clones, accept connections in each process.

- UNIX-domain socket
- **`sendmsg()`**
- **`SCM_RIGHTS`**

Effectively allows the kernel to load balance connections across processes.

Here is **parent.js**:

```
http = require('http');
net = require('net');

// Create a web server
web = http.Server(function (req, res) {
  res.writeHead(200);
  res.end('hello world\n');
});
web.listen(8000);

// File Descriptor server
s = net.Server(function (c) {
  c.write('blah', 'ascii', web.fd); // HERE
  c.end();
});
s.listen('/tmp/node_server.sock');
```

Here is **child.js**

```
net = require('net');
http = require('http');

web = http.Server(function (req, res) {
  res.writeHead(200);
  res.end("hello from " + process.pid);
});

c = net.createConnection('/tmp/node_server.sock');

c.on('fd', function (fd) {
  web.listenFD(fd);
});
```

```
% node parent.js &
[1] 37242
% node child.js &
[2] 37243
%

% curl http://127.0.0.1:8000/
hello from 37243
%

% curl http://127.0.0.1:8000/
hello world
%

% curl http://127.0.0.1:8000/
hello from 37243
%
```

Modules for doing this more easily:

Kris Zyp: `http://github.com/kriszyp/multi-node`

Tim Caswell: `http://github.com/senchalabs/spark`

A 'hello world' server should scale linearly:

processes	req/sec
1	6985.85
2	13420.92
3	19375.66
4	23868.80

(benchmark with Spark from Mathias Pettersson)

Questions...?

`http://nodejs.org/`

`ryan@joyent.com`

Other links

http://howtonode.com/

http://developer.yahoo.com/yui/theater/video.php?v=dahl-node

http://www.youtube.com/watch?v=F6k8lTrAE2g