node.js

ry@tinyclouds.org

November 8, 2009

node.js in brief:

- Server-side Javascript
- Built on Google's V8
- Evented, non-blocking I/O. Similar to EventMachine or Twisted.
- CommonJS module system.
- 8000 lines of C/C++, 2000 lines of Javascript, 14 contributors.

I/O needs to be done differently.

Many web applications have code like this:

```
var result =
   db.query("select * from T");
// use result
```

What is the software doing while it queries the database?

In many cases, just waiting for the response.

I/O latency

L1: 3 cycles

L2: 14 cycles

RAM: 250 cycles

DISK: 41,000,000 cycles

NETWORK: 240,000,000 cycles

Better software can multitask.

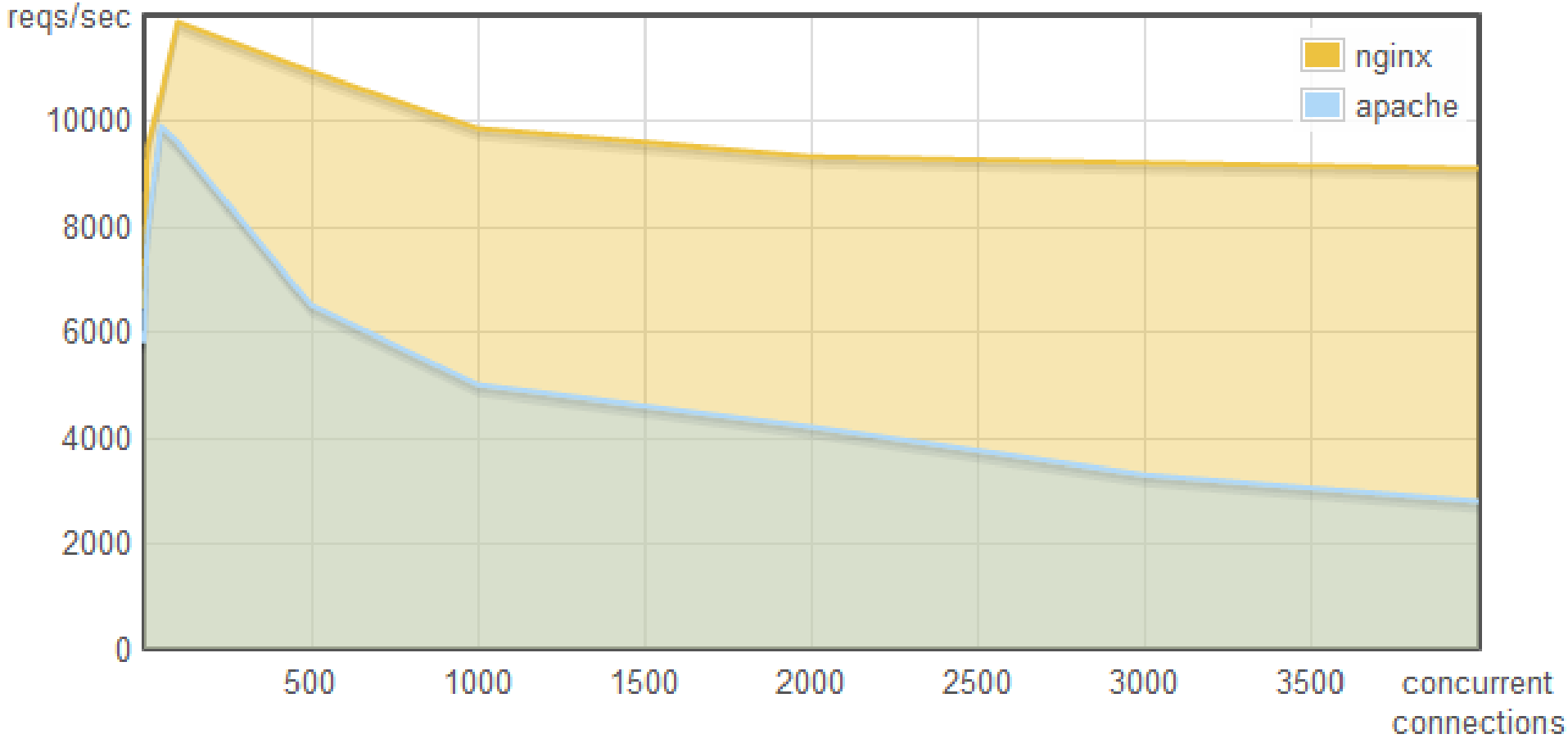
Other threads of execution can run while waiting.

Is that the best that can be done?

A look at Apache and NGINX.

Apache vs NGINX

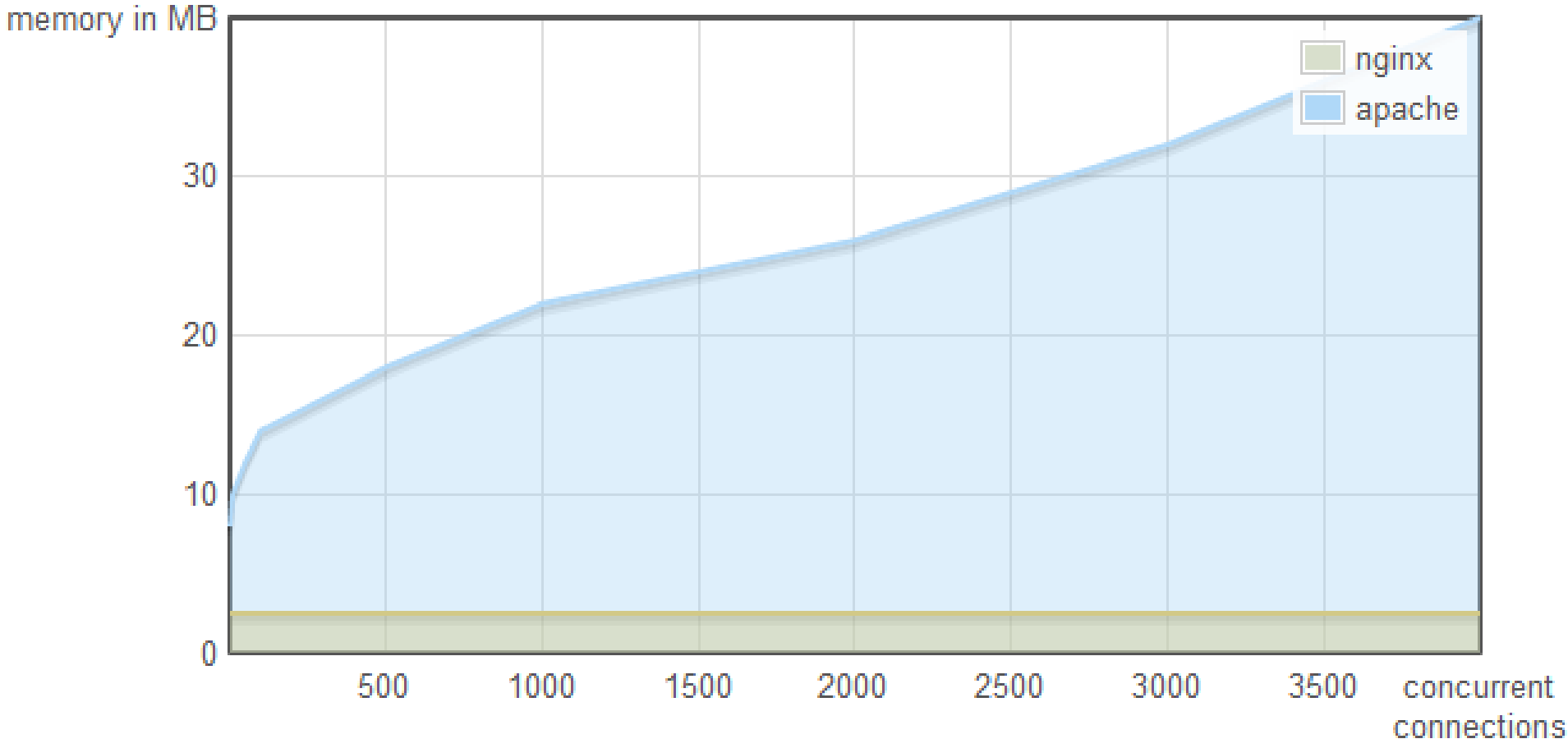
concurrency × reqs/sec



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

Apache vs NGINX

concurrency × memory



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.

Threaded concurrency is a leaky abstraction.

Code like this

```
var result = db.query("select..");
// use result
```

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

But a line of code like this

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

allows the program to return to the event loop immediately.

No machinery required.

```
db.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

No function should direct 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:

TCP, DNS, HTTP

Design Goals

Support many HTTP features.

- Chunked requests and responses.
- Keep-alive.
- Hang 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.

Usage and Examples

(using node 0.1.16)

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
%
```

Change the “hello world” program to loop forever, but print an exit message when the user kills it.

We will use the special object `process` and the `"SIGINT"` signal.

```
1  puts = require("sys").puts;
2
3  setInterval(function () {
4    puts("hello");
5  }, 500);
6
7  process.addListener("SIGINT",
8      function () {
9    puts("good bye");
10   process.exit(0)
11 });
```

```
process.addListener("SIGINT", ...)
```

The `process` object emits an event when it receives a signal. Like in the DOM, you need only add a listener to catch them.

Also:

```
process.pid

process.ARGV

process.ENV

process.cwd()

process.memoryUsage()
```

Like `process`, many other objects in Node emit events.

A TCP server emits a `"connection"` event each time someone connects.

An HTTP upload emits a `"body"` event on each packet.

All objects which emit events are are instances of `process.EventEmitter`.

Write a program which:

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

```
1  var tcp = require("tcp");
2
3  var s = tcp.createServer();
4  s.addListener("connection",
5      function (c) {
6    c.send("hello!");
7    c.close();
8  });
9
10 s.listen(8000);
```

```
% node server.js &
[1] 9120

% telnet localhost 8000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
hello!
Connection closed by foreign host.

%
```

The `"connection"` listener can be provided as the first argument to **`tcp.createServer()`**, so the program can be simplified:

```
1 var tcp = require("tcp");
2 tcp.createServer(function (c) {
3   c.send("hello!\n");
4   c.close();
5 }).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:

```
1 var stat = require("posix").stat,
2     puts = require("sys").puts;
3
4 var promise = stat("/etc/passwd");
5
6 promise.addCallback(function (s) {
7   puts("modified: " + s.mtime);
8 });
```

A promise is a kind of **`EventEmitter`** which emits either **`"success"`** or **`"error"`**. (But not both.)

All file operations return a promise.

```
promise.addCallback(cb)
```

is just API sugar for

```
promise.addListener("success", cb)
```

Simple HTTP Server:

```
1  var http = require("http");
2
3  http.createServer(function (req,res) {
4     res.sendHeader(200,
5        {"Content-Type": "text/plain"});
6     res.sendBody("Hello\r\n");
7     res.sendBody("World\r\n");
8     res.finish();
9  }).listen(8000);
```

```
% node http_server.js &
[4] 27355

% curl -i http://localhost:8000/
HTTP/1.1 200 OK
Content-Type: text/plain
Connection: keep-alive
Transfer-Encoding: chunked

Hello
World

%
```

Streaming HTTP Server:

```
1  var http = require("http");
2  http.createServer(function (req,res) {
3     res.sendHeader(200,
4        {"Content-Type": "text/plain"});
5
6     res.sendBody("Hel");
7     res.sendBody("lo\r\n");
8
9     setTimeout(function () {
10       res.sendBody("World\r\n");
11       res.finish();
12    }, 2000);
13 }).listen(8000);
```

```
% node http_server2.js &
[4] 27355
% curl http://localhost:8000/
Hello
```

Two seconds later...

```
% node http_server2.js &
[4] 27355
% curl http://localhost:8000/
Hello
World
%
```

```
1  var sys = require("sys");
2  sys.exec("ls -l /")
3    .addCallback(function (output) {
4      sys.puts(output);
5    });
```

Programs can be run with

`sys.exec()`

But Node never forces buffering

$\exists$ a lower-level facility to stream data through the STDIO of the child procresses.

Simple IPC.

```
1  var puts = require("sys").puts;
2
3  var cat =
4    process.createChildProcess("cat");
5
6  cat.addListener("output",
7      function (data) {
8    if (data) sys.puts(data);
9  });
10
11 cat.write("hello ");
12 cat.write("world\n");
13 cat.close();
```

Demo / Experiment

An IRC Daemon written in javascript.

irc.nodejs.org #node.js

Source code:

`http://tinyurl.com/ircd-js`

`http://gist.github.com/a3d0bbbff196af633995`

Internal Design

- **V8** (Google)
- **libev** event loop library (Marc Lehmann)
- **libeio** thread pool library (Marc Lehmann)
- **http-parser** a ragel HTTP parser (Me)
- **evcom** stream socket library on top of libev (Me)
- **udns** non-blocking DNS resolver (Michael Tokarev)

Blocking (or possibly blocking) system calls are executed in the thread pool.

Signal handlers and thread pool callbacks are marshaled back into the main thread via a pipe.

```
% node myscript.js < hugefile.txt
```

`STDIN_FILENO` will refer to a file.

Cannot **`select()`** on files;
`read()` will block.

Solution: Start a pipe, and a "pumping thread".

Pump data from blocking fd into pipe.

Main thread can poll for data on the pipe.

(See **`deps/coupling`** if you're interested)

Future

- Fix API warts.
- More modularity; break Node into shared objects.
- Include libraries for common databases in distribution.
- Improve performance.
- TLS support
- Web Worker-like API. (Probably using **`ChildProcess`**)

Future

Version 0.2 in late December or January.

Core API will be frozen.

Questions...?

http://nodejs.org/

ry@tinyclouds.org