STACK4 and the D Programming Language (Part3)

I finally submitted this topic as a proposal for a talk at dconf2014[1] and found the time to continue this series.

This is Part3 in a series of articles.
Part1 roughly elaborated the different options of languages+frameworks that were evaluated before deciding to use D.
Part2 gave an overview of the libraries that I developed and open sourced that the server uses and scratched the surface of why I picked D over Node.

The basis for the STACK4 multiplayer feature is the Sockjs long polling connection model described in the previous article.
In the sockjs-unity3d-xhr-example[2] github project I implemented a simple chat in a unity3d project using 3 different server implementations to be able to compare them. There is a Java, a Node and a D implementation. The Node and D implementation are pretty much the same in length of code, but the Java version is 50% longer and much more verbose. Other than that, my Java version also performed worse than the Node and D version. But that may be due to me being rusty on the Java front.

The performance difference became obvious when running the server on my resource -constrained consumer server that also handles mail, website and more. But being able to start running on such a machine was exactly the intention.

Node vs. vibe.d

I love Node. It introduced me to a whole new mindset and enabled me to prototype a lot of stuff amazingly fast, but then I learned about Vibe.d[3]. For me it is just the next evolutionary step, combing all that is great about Node with an even better programming language and paradigm.

I coded some reasonable big server projects with Node already to know that JavaScript was too much of a deal breaker for me to use it for STACK4 again. In this installment of the series I am going to show some code that makes clear why D (Vibe.d) won over JavaScript (Node). There are three main problems for me concerning Node and JavaScript and that is the JavaScript programming language, readability of code and debugging. The problem with JavaScript maybe just personal taste, but I am used to write statically typed programs that pretty much work after I have them compiling. In JavaScript however after writing code the challenge begins, because a lot of coding errors are first popping up once a certain part of the code is being executed. The other two problems are discussed below.

Readable Code

What makes Node code so hard to read on my eyes is the fact, that it is often a hell of a lot nested callbacks.
For the purpose of this article I present a short example to demonstrate this. Imagine a simple http server that should respond after a second of delay with a "hello world" and also logs the request to a file:

```
1  var http = require("http");
2  var fs = require("fs");
3
4  http.createServer(function (req, res){
5          res.writeHead(200, {'content-type': 'text/plain'});
6
```

```
        setTimeout(function(){
                fs.appendFile('log.txt', 'data to append', function
(err) {
                        if (err) throw err;

                        res.end("World!");
                });
        }, 1000);
}).listen(80);
```

Now here is the D version using Vibe.d:

```
import vibe.d;

void handleRequest(HTTPServerRequest req, HTTPServerResponse res) {
        sleep(1.seconds);

        appendToFile("log.txt", "data to append");

        res.writeBody("Hello World!");
}

shared static this() {
        listenHTTP(new HTTPServerSettings, &amp;handleRequest);
}
```

Aside the fact that the D code is using just one indentation level it has a couple of other advantages over the Node version. Thanks to a D feature called Unified Function Call Syntax (UFCS)[4] line #4 is possible and makes it both easier to read and less error prone. It clearly states our intention to wait for a second before continuing the execution. What UFCS actually brings to the table is that you can rewrite every function foo taking T as a first argument as T.foo(). This way you can extend every type even those that you have no access to with standalone methods that act as if they are members of it. The seconds method is typesafe and is more expressive than a 1000 ms integer magic number.

Note, that the whole server process in not blocking. However the fiber[5] , that handles a single http request will be blocked eventually. This is the major difference of Vibe.d and Node. This way the intention of the code is instantly clear and behaves exactly as serial as it is written, even though the whole system is still highly asynchronous and event driven. Two parallel requests get handled in two different fibers and one can already be responding to the client while the other is still waiting in the sleep without any kind of multithreading risks. D calls them fibers, but they are also known as co-routines[6].

Note that there is the npm package co[7] to get co-routines in Node, in fact there is a couple of libraries to avoid the callback hell. The fact that there are so many libs to mitigate this issue shows that this is a problem that Node users crave to have solved. I also have to mention that I took this sample from a popular Node textbook, so it is safe to say that this callback nesting is common practice.

Debugging

There are two things that bothered me while writing this simple Node showcase:

1. Not a single typo got caught before actually running the related line of code.
2. Try reading the callstack when an exception in the appendFile occurs.

While 1. is simply due to the nature of JavaScript being interpreted just in time and maybe subject to taste, 2. is plain annoying when trying to debug even the simplest Node programs. And it does not stop there, since the callstack is totally messed up all error handling in the code based on exceptions is making knots in my head (just try to put try/catch blocks around some of this nested goodness and see…).

Let's imagine a simple throw "poo" exception in the above code inside the appendFile callback and have a look at the callstack printed:

```
e:\_docs\_bitbucket\test\js\main.js:10
throw "poo";
^
```

Following is the D version of this. It is pretty short too, but since the handleRequest runs in ordinary serial order the callstack always is correct and all try/catch and powerful scope guards[8] still work:

```
object.Exception@source\app.d(8): poo
----------------
0x00403972 in void app.handleRequest(vibe.http.server.HTTPServerRequest,
vibe.http.server.HTTPServerResponse) at
E:\_docs\_bitbucket\test\d\source\app.d(11)
[...vibe.d internal callstack snipped...]
```

So in Node there is no usable stacktrace. When you mix and reuse callbacks a lot you have no idea from where a certain exception got thrown. This also makes it hard to use try/catch correctly. In D using Vibe.d this just works safe and sound!

Huge systems have been successfully written using Node but it introduces a new execution and error-handling dimension that must be handled manually. With my C mind if just feels more natural to use the Vibe.d approach with it's own drawbacks.

Next time I will go into the design of the STACK4 server architecture.

Special thanks to Mason Browne (@neworb) for his constructive feedback on the first version of this post!

1. http://dconf.org/2014/index.html [↩]
2. https://github.com/Extrawurst/sockjs-unity3d-xhr-example/tree/master/server [↩]
3. http://vibed.org/ [↩]
4. http://dlang.org/function.html#interpretation [↩]
5. http://vibed.org/features#fibers [↩]
6. https://en.wikipedia.org/wiki/Coroutine [↩]
7. https://github.com/visionmedia/co [↩]
8. http://dlang.org/statement.html#ScopeGuardStatement [↩]