Node.js: Using JavaScript to Build High-Performance Network Programs

Stefan Tilkov • *innoQ*
Steve Vinoski • *Verivue*

Node.js — also called Node — is a server-side JavaScript environment (see http://nodejs.org). It's based on Google's runtime implementation — the aptly named "V8" engine. V8 and Node are mostly implemented in C and C++, focusing on performance and low memory consumption. But, whereas V8 supports mainly JavaScript in the browser (most notably, Google Chrome), Node aims to support long-running server processes.

Unlike in most other modern environments, a Node process doesn't rely on multithreading to support concurrent execution of business logic; it's based on an asynchronous I/O eventing model. Think of the Node server process as a single-threaded daemon that embeds the JavaScript engine to support customization. This is different from most eventing systems for other programming languages, which come in the form of libraries: Node supports the eventing model at the language level.

JavaScript is an excellent fit for this approach because it supports event callbacks. For example, when a browser completely loads a document, a user clicks a button, or an Ajax request is fulfilled, an event triggers a callback. JavaScript's functional nature makes it extremely easy to create anonymous function objects that you can register as event handlers.

Multithreading versus Events

Application developers who deal with multiple I/O sources, such as networked servers handling multiple client connections, have long employed multithreaded programming techniques. Such techniques became popular because they let developers divide their applications into concurrent cooperating activities. This promised to not only make program logic easier to understand, implement, and maintain but also enable faster, more efficient execution.

For applications such as Web servers performing significant amounts of I/O, multiple threads enable applications to better use available processors. Running multiple concurrent threads on a modern multicore system is straightforward, with each core simultaneously executing a different thread with true parallelism. On single-core systems, the single processor executes one thread, switches to another and executes it, and so on. For example, the processor switches its execution context to another thread when the current thread performs an I/O operation, such as writing to a TCP socket. The switch occurs because completing that operation can take many processor cycles. Rather than wasting cycles waiting for the socket operation to finish, the processor sets the I/O operation in motion and executes another thread, thus keeping itself busy doing useful work. When the I/O operation ends, the processor again considers the original thread to be ready to execute because it's no longer blocked while waiting for I/O.

Even though many developers have successfully used multithreading in production applications, most agree that multithreaded programming is anything but easy. It's fraught with problems that can be difficult to isolate and correct, such as deadlock and failure to protect resources shared among threads. Developers also lose some degree of control when drawing on multithreading because the OS typically decides which thread executes and for how long.

Event-driven programming offers a more efficient, scalable alternative that provides developers much more control over switching between application activities. In this model, the application relies on event notification facilities such as

the `select()` and `poll()` Unix system calls, the Linux `epoll` service, and the `kqueue` and `kevent` calls available in BSD Unix variants such as OS X. Applications register interest in certain events, such as data being ready to read on a particular socket. When the event occurs, the notification system notifies the application so that it can handle the event.

Asynchronous I/O is important for event-driven programming because it prevents the application from getting blocked while waiting in an I/O operation. For example, if the application writes to a socket and fills the socket's underlying buffer, ordinarily, the socket blocks the application's writes until buffer space becomes available, thus preventing the application from doing any other useful work. But, if the socket is nonblocking, it instead returns an indication to the application that further writing isn't currently possible, thereby informing the application that it should try again later. Assuming the application has registered interest with the event notification system in that socket, it can go do something else, knowing that it will receive an event when the socket's write buffer has available space.

Like multithreaded programming, event-driven programming with asynchronous I/O can be problematic. One problem is that not all interprocess-communication approaches can be tied into the event notification facilities we mentioned earlier. For example, on most OSs, for two applications to communicate through shared memory, shared-memory segments provide no handles or file descriptors enabling the application to register for events. For such cases, developers must resort to alternatives such as writing to a pipe or some other event-capable mechanism together with writing to shared memory.

Another significant problem is the sheer complexity of writing applications in certain programming languages to deal with events and asynchronous I/O. This is because different events require different actions in different contexts. Programs typically employ callback functions to deal with events. In languages that lack anonymous functions and closures, such as C, developers must write individual functions specifically for each event and event context. Ensuring that these functions all have access to the data and context information they require when they're called to handle an event can be incredibly perplexing. Many such applications end up being little more than impenetrable, unmaintainable tangles of spaghetti code and global variables.

Not Your Father's JavaScript

Whatever you might think about JavaScript as a programming language, there's little to no doubt it has become a central element of any modern HTML-based application. Server-side JavaScript is a logical next step, enabling the use of a single programming language for all aspects of a Web-based distributed application. This idea isn't new — for example, the Rhino JavaScript execution environment has been available for a long time. Still, server-side JavaScript isn't yet a mainstream approach and has only recently gained massive popularity.

We believe that a number of factors have led to this effect. The advent of the set of technologies collectively labeled "HTML 5" reduces the appeal of alternative client-side platforms, enforcing the need to get to know and exploit JavaScript to create rich user interfaces. NoSQL-type databases such as CouchDB and Riak use JavaScript to define data views and filter criteria. Other dynamic languages, such as Ruby and Python, have become acceptable choices for server-side development. Finally, both Mozilla and Google have released high-performance JavaScript runtime implementations that are extremely fast and scalable.

The Node Programming Model

Node's I/O approach is strict: asynchronous interactions aren't the exception; they're the rule. Every I/O operation is handled by means of higher-order functions — that is, functions taking functions as a parameter — that specify what to do when there's something to do. In only rare circumstances have Node's developers added a convenience function that works synchronously — for example, for removing or renaming files. But, generally, when operations that might require network or file I/O are invoked, control is immediately returned to the caller. When something interesting happens — for example, if data becomes available for reading from a network socket, an output stream is ready for writing, or an error occurs — the appropriate callback function is called.

Figure 1 is a simple example of implementing an HTTP Web server that serves static files from disk. Even to non-Web developers, JavaScript's syntax should be fairly obvious for those with prior exposure to any C-like language. One of the more specific topics is the `function(...)` syntax. This creates an unnamed function: JavaScript is a functional language and, as such, supports higher-order functions. A developer writing or looking at a Node program will see these everywhere.

The program's main flow is determined by the functions that are explicitly called. These functions never block on anything I/O-related, but rather register appropriate handler callbacks. If you've seen a similar concept in eventing libraries for other programming languages, you might wonder where the explicit blocking call to invoke the event loop hides. The event loop concept is so

```
var sys = require("sys"),
    http = require("http"),
    url = require("url"),
    path = require("path"),
    fs = require("fs");

http.createServer(function(request, response) {
    var uri = url.parse(request.url).pathname;
    var filename = path.join(process.cwd(), uri);
    path.exists(filename, function(exists) {
        if(exists) {
            fs.readFile(filename, function(err, data) {
                response.writeHead(200);
                response.end(data);
            });
        } else {
            response.writeHead(404);
            response.end();
        }
    });
}).listen(8080);
sys.log("Server running at http://localhost:8080/");
```

Figure 1. A simple HTTP file server. Events trigger anonymous functions that execute input or output operations. Incoming requests trigger the server to parse the target URI, look for a local file matching the URI path, and, if found, read the file contents and write them along with appropriate HTTP headers as a response to the client.

core to Node's behavior that it's hidden in the implementation; the main program's purpose is simply to set up appropriate handlers. The `http.createServer` function, which is a wrapper around a low-level efficient HTTP protocol implementation, is passed a function as the only argument. This function is invoked whenever data for a new request is ready to be read. In another environment, a naïve implementation might ruin the effect of eventing by synchronously reading a file and sending it back. Node offers no opportunity to read a file synchronously — the only option is to register another function via `readFile` that gets invoked whenever data can be read.

Concurrent Programming
A node server process, usually invoked from the command line using something like "`node <scriptname>`," runs single-threaded, yet can serve many clients concurrently. This seems a contradiction, but recall that there's an implicit main loop around the code, and what's actually happening in that loop is just a number of registration calls. No actual I/O, let alone business-logic processing, happens in the loop body. I/O-related events trigger the actual processing, such as a connection being made or bytes being sent or received from a socket, file, or external system.

Figure 2 is a slightly more complex variant of the simplistic HTTP server, but it does a lot more. Again, it parses the URI from an HTTP request and maps the URI's path component to a filename on the server. But this time, the file is read in smaller chunks rather than all at once. In certain situations, the function provided for the scenario as a callback is invoked. Example situations include when the file system layer is ready to hand a number of bytes to the application, when the file has been read completely, or when some kind of error occurs. If data is available, it's written to the HTTP output stream. Node's sophisticated HTTP library supports HTTP 1.1's chunked transfer encoding. Again, both reading from the file and writing to the HTTP stream happen asynchronously.

The example in Figure 2 shows how easily developers can build a high-performance, asynchronous, event-driven network server with modest resource requirements. The main reason is that JavaScript, owing to its functional nature, supports event callbacks. In fact, this pattern is well known to any client-side JavaScript developer. In addition, making asynchronous I/O the default forces developers to adopt the asynchronous model from the start. This is one of the main differences between Node and using asynchronous I/O in other programming environments, in which it's only one of many options and is often considered too advanced.

Running Multiple Processes
In hardware environments in which more than one physical CPU or core is available, parallel execution isn't an illusion but a reality. Although the OS can efficiently schedule a Node process with its asynchronous I/O interactions in parallel with other processes running on the system, Node still runs in a single process and thus never executes its core business logic in parallel. The common solution to this problem in the Node world is to run multiple process instances.

To support this, the multi-node library (see http://github.com/kris zyp/multi-node) leverages the OS's capability of sharing sockets between processes (and is implemented in fewer than 200 lines of Node JavaScript). For example, you can run HTTP servers such as

those in Figures 1 and 2 in parallel by invoking multi-node's `listen()` function. This starts multiple processes that all listen on the same port, effectively using the OS as an efficient load balancer.

A Server-Side JavaScript Ecosystem

Node is one of the better-known frameworks and environments that support server-side JavaScript development. The community has created a whole ecosystem of libraries for, or compatible with, Node. Among these, tools such as node-mysql or node-couchdb play an important role by supporting asynchronous interaction with relational and NoSQL data stores, respectively. Many frameworks provide a full-featured Web stack, such as Connect and Express, which are comparable to Rack and Rails in the Ruby world in scope, if not (yet?) in popularity. The Node package manager, npm, enables installation of libraries and their dependencies. Finally, many libraries available for client-side JavaScript that were written to comply with the CommonJS module system also work with Node. An impressive list of modules available for Node is at http://github.com/ry/node/wiki/modules.

G iven that, in most Web development projects, JavaScript knowledge is a prerequisite for advanced UI interactions, the option of using one programming language for everything becomes quite tempting. Node.js's architecture makes it easy to use a highly expressive, functional language for server programming, without sacrificing performance and stepping out of the programming mainstream.

```
var sys = require("sys"),
    http = require("http"),
    url = require("url"),
    path = require("path"),
    fs = require("fs");

http.createServer(function(request, response) {
    var uri = url.parse(request.url).pathname;
    var filename = path.join(process.cwd(), uri);
    path.exists(filename, function(exists) {
        if(exists) {
            f = fs.createReadStream(filename);
            f.addListener('open', function() {
                response.writeHead(200);
            });
            f.addListener('data', function(chunk) {
                response.write(chunk);
                setTimeout(function() {
                    f.resume()
                }, 100);
            });
            f.addListener('error', function(err) {
                response.writeHead(500, {"Content-Type":
                                        "text/plain"});
                response.write(err + "\n");
                response.end();
            });
            f.addListener('close', function() {
                response.end();
            });
        } else {
            response.writeHead(404);
            response.end();
        }
    });
}).listen(8080);
sys.log("Server running at http://localhost:8080/");
```

Figure 2. A simple streaming HTTP file server. Chunks of the file are read from disk and sent to the client using HTTP's "chunked" transfer encoding.

Stefan Tilkov is cofounder of innoQ, a technology consultancy with offices in Germany and Switzerland. He blogs at www.innoq.com/blog/st and is addicted to a certain social network where he's identified as @stilkov.

Steve Vinoski is a member of the technical staff at Verivue. He's a senior member of the IEEE and a member of the ACM. You can read Vinoski's blog at http://steve.vinoski.net/blog and contact him at vinoski@ieee.org.

cn *Selected CS articles and columns are also available for free at http:// ComputingNow.computer.org.*