Node.js

Mendel Rosenblum

Threads versus Events

```
request = readRequest(socket);
reply = processRequest(request);
sendReply(socket, reply);
```

Implementation:

Thread switching (i.e. blocking) and a scheduler

```
startRequest(socket);
listen("requestAvail", processRequest);
listen("processDone", sendReplyToSock);
```

Implementation:

Event queue processing

Threads versus Events using Callbacks

```
request = readRequest(socket);
reply = processRequest(request);
sendReply(socket, reply);
```

Implementation:

Thread switching (i.e. blocking) and a scheduler

```
readRequest(socket, function(request) {
   processRequest(request,
       function (reply) {
         sendReply(socket, reply);
       });
});
```

Implementation:

Event queue processing

Event queue

- Inner loop

```
while (true) {
   if (!eventQueue.notEmpty()) {
      eventQueue.pop().call();
   }
}
```

- Never wait/block in event handler . Example `readRequest(socket);`

1. `launchReadRequest(socket);` // Returns immediately
2. When read finishes: `eventQueue.push(readDoneEventHandler);`

Node.js

- Take a JavaScript engine from a browser (Chrome's V8 JavaScript Engine)
  - Get same JavaScript on both browser and server
  - Don't need the DOM on the server
- Add events and an event queue
  - Everything runs as a call from the event loop (already had one for browser events)
- Make event interface to all OS operations
  - Wrap all the OS blocking calls (file and socket/network io)
  - Add some data handle support
- Add a proper module system (predated `import`/`export`)
  - Each module gets its own scope (not everything in `window`)

Example: Node.js reading a file

```
var fs = require("fs");  // require is a Node module call
                         // fs object wraps OS sync file system calls

 // OS read() is synchronous but Node's fs.readFile is asynchronous
fs.readFile("smallFile", readDoneCallback);  // Start read

function readDoneCallback(error, dataBuffer) {
    // Node callback convention: First argument is JavaScript Error object
    // dataBuffer is a special Node Buffer object
   if (!error)  {
       console.log("smallFile contents", dataBuffer.toString());
    }
}
```

Node Modules

- Import using `require() - Can use ES6 import if file name *.mjs`
  - System module: `require("fs"); // Looks in node_modules directories`
  - From a file: `require("./XXX.js"); // Reads specified file`
  - From a directory: `require("./myModule"); // Reads myModule/index.js`
- Module files have a private scope
  - Can declare variables that would be global in the browser
  - Require returns what is assigned to `module.exports`

```
var notGlobal;
function func1() {}
function func2() {}
module.exports = {func1: func1, func2: func2};
```

Node modules

- Many standard Node modules
  - File system, process access, networking, timers, devices, crypto, etc.
- Huge library of modules (npm)
  - Do pretty much anything you want
- We use:
  - Express - Fast, unopinionated, minimalist web framework (speak HTTP)
  - Mongoose - Elegant mongodb object modeling (speak to the database)

Node Buffer class

- Manipulating lots of binary data wasn't a strength of the JavaScript engine
  - Unfortunately that is what web servers do: DBMS ⇔ Web Server ⇔ Browser
- Node add a Buffer class - Optimized for storing and operating on binary data
  - Interface looks an array of bytes (like the OS system calls use)
  - Memory is allocated outside of the V8 heap
- Used by the wrapped OS I/O calls (fs, net, …)
- Optimized sharing with pointers rather than always copying
  - buffer.copy()
  - For example: `fs.readFile` to `socket.write`

Buffer operations

- Supports operations for picking values out or updating them
  - Can peek at some values and send the bulk of it on its way
- Can convert a buffer or parts of a buffer to a JavaScript string
  - `buf.toString("utf8");  // Convert to UTF8 - commonly used on the web`
  - `buf.toString("hex");  // Convert to hex encoding (2 digits per byte)`
  - `buf.toString("base64");  // Convert to base64 encoding`

Example: Node.js reading a file (redux)

```
var fs = require("fs");
// fs has 81 properties readFile,  writeFile, most OS calls, etc.
fs.readFile("smallFile", readDoneCallback);  // Start read
// Read has been launched - JavaScript execution continues
// Node.js exits when no callbacks are outstanding
function readDoneCallback(error, dataBuffer) {
      // console.log(dataBuffer) prints  <Buffer 66 73 20 3d 20 72 65 71 ...
      if (!error)  {
        console.log("smallFile contents", dataBuffer.toString());
       }
}
```

Programming with Events/Callbacks

- Key difference
  - Threads: Blocking/waiting is transparent
  - Events: Blocking/waiting requires callback

- Mental model
  If code doesn't block: Same as thread programming

  If code does block (or needs to block): Need to setup callback

  Often what was a return statement becomes a function call

Example: Three step process

Threads

```
r1 = step1();
console.log('step1 done', r1);
r2 = step2(r1);
console.log('step2 done', r2);
r3 = step3(r2);
console.log('step3 done', r3);
console.log('All Done!');
```

Works for **non-blocking** calls in both styles

Callbacks

```
step1(function(r1) {
   console.log('step1 done', r1);
   step2(r1, function (r2) {
      console.log('step2 done', r2);
      step3(r2, function (r3) {
         console.log('step3 done',r3);
      });
   });
}):
console.log('All Done!'); // Wrong!
```

Example: Three step process

Threads

```
r1 = step1();
console.log('step1 done', r1);
r2 = step2(r1);
console.log('step2 done', r2);
r3 = step3(r2);
console.log('step3 done', r3);
console.log('All Done!');
```

Callbacks

```
step1(function(r1) {
   console.log('step1 done', r1);
   step2(r1, function (r2) {
      console.log('step2 done', r2);
      step3(r2, function (r3) {
         console.log('step3 done', r3);
         console.log('All Done!');
      });
   });
});
```

Listener/emitter pattern

- When programing with events (rather than threads) a listener/emitter pattern is often used.
- Listener - Function to be called when the event is signaled
  - Should be familiar from DOM programming (`addEventListerner`)
- Emitter - Signal that an event has occurred
  - Emit an event cause all the listener functions to be called

Node: `EventEmitter = require('events');`

- Listen with `on()` and signal with `emit()`

```
myEmitter.on('myEvent', function(param1, param2) {
    console.log('myEvent occurred with ' + param1 + ' and ' + param2 + '!');
});
myEmitter.emit('myEvent', 'arg1', 'arg2');
```

- On emit call listeners are called synchronously and in the order the listeners were registered
- If no listener then `emit()` is a nop

Typical `EventEmitter` patterns

- Have multiple different events for different state or actions

```
myEmitter.on('conditionA', doConditionA);
myEmitter.on('conditionB', doConditionB);
myEmitter.on('conditionC', doConditionC);
myEmitter.on('error', handleErrorCondition);
```

- Handling `'error'` is important - Node exits if not caught!

```
myEmitter.emit('error', new Error('Ouch!'));
```

Streams

- Build modules that produce and/or consume streams of data
- A popular way of structuring servers
  - Network interface ⇔ TCP/IP protocol processing ⇔ HTTP protocol processing ⇔ your code
- Can build connected streams dynamically
  - Add modules on stream: E.g. `stream.push(Encryption)`
    Network interface ⇔ TCP/IP protocol processing ⇔ **Encryption** ⇔ HTTP processing
- Node's APIs heavily uses streams
  - Readable streams (e.g. `fs.createReadStream`)
  - Writable stream (e.g. `fs.createWriteStream`)
  - Duplex stream (e.g. `net.createConnection`)
  - Transform stream (e.g. `zlib, crypto`)

Readable streams - File reading using streams

```
var readableStreamEvent = fs.createReadStream("bigFile");

readableStreamEvent.on('data', function (chunkBuffer) {   // Could be called multiple times
  console.log('got chunk of', chunkBuffer.length, 'bytes');
});

readableStreamEvent.on('end', function() {
  // Called after all chunks read
  console.log('got all the data');
});

readableStreamEvent.on('error', function (err) {
  console.error('got error', err);
});
```

Writable streams - File writing using streams

```
var writableStreamEvent = fs.createWriteStream('outputFile');


writableStreamEvent.on('finish', function () {
  console.log('file has been written!');
});


writableStreamEvent.write('Hello world!\n');


writableStreamEvent.end();
```

Digression: Socket setup for TCP connections

Client (Browser)

```
sock = socket(AF_INET, SOCK_STREAM, 0);

connect(sock, &serverAddr,
                  sizeof(serverAddr));

write(sock, "Hi!", 3);

read(sock, buf, 3)
```

Server (Web Server)

```
lfd = socket(AF_INET, SOCK_STREAM, 0);

bind(lfd, &serverSddr, sizeof(serveraddr));

listen(lfd, 5);

sock = accept(lfd, &clientaddr, &clientlen);

read(sock, buf, 3);

write(sock, buf, 3)
```

- TCP/IP socket connection is a reliable, in-order byte stream
  - Note: reads can return data in different chunks that sent

TCP Networking on Node.js

- Node `net` module wraps OS's network routines
- Includes higher level functionality like:

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

net.createServer(processTCPconnection).listen(4000);
```

Creates a socket, binds port 4000, and listens for connections

Calls function `processTCPconnection` on each TCP connection

Example: A chat server

```
var clients = []; // List of connected clients
function processTCPconnection(socket) {
  clients.push(socket); // Add this client to our connected list

  socket.on('data', function (data) {
    broadcast( "> " + data, socket);  // Send received data to all
  });
  socket.on('end', function () {
    clients.splice(clients.indexOf(socket), 1); // remove socket
  });
}
```

Chat Server: broadcast

```
// Send message to all clients
function broadcast(message, sender) {
  clients.forEach(function (client) {
     if (client === sender) return; // Don't send it to sender
    client.write(message);
  });
}
```

- Our chat server implementation is done!

Putting it together: A real simple file server

```
net.createServer(function (socket) {
   socket.on('data', function (fileName) {
      fs.readFile(fileName.toString(), function (error, fileData) {
        if (!error) {
           socket.write(fileData);  // Writing a Buffer
        } else {
           socket.write(error.message); // Writing a String
        }
        socket.end();
     });
  });
}).listen(4000);
```

- Think about concurrency going on here

Example: Read three files

- Linux
  read(open("f1"), buf1, f1Size);
  read(open("f2"), buf2, f2Size);
  read(open("f3"), buf3, f3Size);

- Node.js

```
fs.readFile("f1", function (error, data1) {
    fs.readFile("f2", function (error, data2) {
        fs.readFile("f3", function (error, data3) {
            // Call Pyramid of Doom or Callback Hell
}); }); });
```

Example: Read N files

```
var fileContents = {};
['f1','f2','f3'].forEach(function (fileName) {
    fs.readFile(fileName, function (error, dataBuffer) {
        assert(!error);
        fileContents[fileName] = dataBuffer;
   });
});
```

If we want to use `fileContents` how do we know when all reads are finished?

Recall: Can't wait in NodeJS

Example: Read N files

```
var fileContents = {};
['f1','f2','f3'].forEach(function (fileName) {
    fs.readFile(fileName, function (error, dataBuffer) {
        assert(!error);
        fileContents[fileName] = dataBuffer;
        if (gotLastCallBack()) allDoneCallback(fileContents);
    });
});
```

- Yuck!

Async module: `var async = require('async');`

- Solution: Write a function that turns waiting into a callback

```
var fileContents = {};

async.each(['f1','f2','f3'], readIt, function (err) {

   if (!err) console.log('Done'); // fileContents filled in
   if (err) console.error('Got an error:', err.message);
});

function readIt(fileName, callback) {
   fs.readFile(fileName, function (error, dataBuffer) {
       fileContents[fileName] = dataBuffer;
       callback(error);
   });
}
```

Node.JS - many useful built-in modules

- Buffer
- C/C++ Addons
- Child Processes
- Cluster
- Console
- Crypto
- Debugger
- DNS
- Errors
- Events
- File System
- Globals
- HTTP
- HTTPS
- Modules
- Net
- OS
- Path
- Process
- Punycode
- Query Strings
- Readline
- REPL
- Stream
- String Decoder
- Timers
- TLS/SSL
- TTY
- UDP/Datagram
- URL
- Utilities
- V8
- VM
- ZLIB