**Event demultiplexing**

socketA, pipeB;

watchedList.add(socketA, FOR\_READ); //[1]

watchedList.add(pipeB, FOR\_READ);

while(events = demultiplexer.watch(watchedList)) { //[2]

//event loop

foreach(event in events) { //[3]

//This read will never block and will always return data

data = event.resource.read();

if(data === RESOURCE\_CLOSED)

//the resource was closed, remove it from the watched list

demultiplexer.unwatch(event.resource);

else

//some actual data was received, process it

consumeData(data);

}

**The non-blocking I/O engine of Node.js – libuv**

**The recipe for Node.js**

The reactor pattern and libuv are the basic building blocks of Node.js, but we need

the following three other components to build the full platform:

• A set of bindings responsible for wrapping and exposing libuv and other

low-level functionality to JavaScript.

• **V8**, the JavaScript engine originally developed by Google for the Chrome

browser. This is one of the reasons why Node.js is so fast and efficient.

V8 is acclaimed for its revolutionary design, its speed, and for its efficient

memory management.

• A core JavaScript library (called **node-core**) that implements the high-level

Node.js API.

Pattern: we guarantee that a callback is invoked asynchronously by

deferring its execution using process.nextTick().

**Callbacks come last**

fs.readFile(filename, [options], callback)

**Error comes first**

In CPS, errors are propagated as any other type of result, which means using the

callback. In Node.js, any error produced by a CPS function is always passed as

the first argument of the callback, and any actual result is passed starting from the

second argument. If the operation succeeds without errors, the first argument

will be null or undefined. The following code shows you how to define a callback

complying with this convention:

fs.readFile('foo.txt', 'utf8', function(err, data) {

if(err)

handleError(err);

else

processData(data);

});

**The revealing module pattern**

One of the major problems with JavaScript is the absence of namespacing.

Programs run in the global scope polluting it with data that comes from both

internal application code and dependencies. A popular technique to solve this

problem is called **revealing module pattern** and it looks like the following:

var module = (function() {

var privateFoo = function() {...};

var privateVar = [];

var export = {

publicFoo: function() {...},

publicBar: function() {...}

}

return export;

})();

**Module definition patterns**

**Named exports**

//file logger.js

exports.info = function(message) {

console.log('info: ' + message);

};

exports.verbose = function(message) {

console.log('verbose: ' + message);

};

The exported functions are then available as properties of the loaded module,

as shown in the following code:

//file main.js

var logger = require('./logger');

logger.info('This is an informational message');

logger.verbose('This is a verbose message');

**Exporting a function**

//file logger.js

module.exports = function(message) {

console.log('info: ' + message);

};

module.exports.verbose = function(message) {

console.log('verbose: ' + message);

};

//file main.js

var logger = require('./logger');

logger('This is an informational message');

logger.verbose('This is a verbose message');

**Modifying other modules or the global scope**

//file patcher.js

// ./logger is another module

require('./logger').customMessage = function() {

console.log('This is a new functionality');

};

Using our new patcher module would be as easy as writing the following code:

//file main.js

require('./patcher');

var logger = require('./logger');

logger.customMessage();

**The observer pattern**

Pattern (observer): defines an object (called subject), which can notify a

set of observers (or listeners), when a change in its state happens.

**The EventEmitter**

In traditional object-oriented programming, the observer pattern requires interfaces,

concrete classes, and a hierarchy; in Node.js, all becomes much simpler. The observer

pattern is already built into the core and is available through the **EventEmitter** class.

The EventEmitter class allows us to register one or more functions as listeners,

which will be invoked when a particular event type is fired. The following image

visually explains the concept:

The EventEmitter is a prototype, and it is exported from the events core module.

The following code shows how we can obtain a reference to it:

var EventEmitter = require('events').EventEmitter;

var eeInstance = new EventEmitter();

Pattern: create a function that accepts a callback and returns an

EventEmitter, thus providing a simple and clear entry point for

the main functionality, while emitting more fine-grained events

using the EventEmitter.

const EventEmitter = require('events');

function createEventEmitter(callback) {

const eventEmitter = new EventEmitter();

// Entry point event

eventEmitter.on('start', () => {

console.log('Main functionality started');

if (callback) {

callback();

}

});

// Fine-grained events

eventEmitter.on('processStep1', () => {

console.log('Processing step 1');

});

eventEmitter.on('processStep2', () => {

console.log('Processing step 2');

});

// Another fine-grained event

eventEmitter.on('finish', () => {

console.log('Main functionality finished');

});

return eventEmitter;

}

// Example usage

const myEventEmitter = createEventEmitter(() => {

console.log('Callback executed!');

});

myEventEmitter.emit('start');

myEventEmitter.emit('processStep1');

myEventEmitter.emit('processStep2');

myEventEmitter.emit('finish');

const EventEmitter = require('events');

function createEventEmitter(callback) {

const eventEmitter = new EventEmitter();

// Entry point event

eventEmitter.on('start', () => {

console.log('Main functionality started');

if (callback) {

callback();

}

});

// Fine-grained events

eventEmitter.on('processStep1', () => {

console.log('Processing step 1');

});

eventEmitter.on('processStep2', () => {

console.log('Processing step 2');

});

// Another fine-grained event

eventEmitter.on('finish', () => {

console.log('Main functionality finished');

});

return eventEmitter;

}

// Example usage

const myEventEmitter = createEventEmitter(() => {

console.log('Callback executed!');

});

myEventEmitter.emit('start');

myEventEmitter.emit('processStep1');

myEventEmitter.emit('processStep2');

myEventEmitter.emit('finish');