

PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC3585 Diseño Avanzado de Aplicaciones Web ~ Segundo Semestre 2016

Reactor Pattern

Node.js

Andrés Matte (aamatte@uc.cl)
Sebastián Soto (spsoto@uc.cl)

5 de Septiembre, 2016

Motivación

Motivación

Introducción

Ejemplo inicial

Conceptos previos

Reactor Pattern

libuv, el engine de Node.js

Conclusiones

Introducción

- **Node.js** es en su núcleo un framework de naturaleza asíncrona, logrando esto en una arquitectura single-threaded y con non-blocking I/O.
- Comprender el funcionamiento del núcleo es fundamental para poder diseñar software en él.
- Revisaremos conceptos básicos asociados a la naturaleza asíncrona de **Node.js** y luego desarrollaremos el concepto de **Reactor Pattern**.
- Este es un patrón de diseño de software ampliamente utilizado en teoría de redes, cuya comprensión no es fácil leyéndolo en dicho contexto.

Introducción

- **Node.js** es en su núcleo un framework de naturaleza asíncrona, logrando esto en una arquitectura single-threaded y con non-blocking I/O.
- Comprender el funcionamiento del núcleo es fundamental para poder diseñar software en él.
- Revisaremos conceptos básicos asociados a la naturaleza asíncrona de **Node.js** y luego desarrollaremos el concepto de Reactor Pattern.
- Este es un patrón de diseño de software ampliamente utilizado en teoría de redes, cuya comprensión no es fácil leyéndolo en dicho contexto.

Introducción

- **Node.js** es en su núcleo un framework de naturaleza asíncrona, logrando esto en una arquitectura single-threaded y con non-blocking I/O.
- Comprender el funcionamiento del núcleo es fundamental para poder diseñar software en él.
- Revisaremos conceptos básicos asociados a la naturaleza asíncrona de **Node.js** y luego desarrollaremos el concepto de **Reactor Pattern**.
- Este es un patrón de diseño de software ampliamente utilizado en teoría de redes, cuya comprensión no es fácil leyéndolo en dicho contexto.

Introducción

- **Node.js** es en su núcleo un framework de naturaleza asíncrona, logrando esto en una arquitectura single-threaded y con non-blocking I/O.
- Comprender el funcionamiento del núcleo es fundamental para poder diseñar software en él.
- Revisaremos conceptos básicos asociados a la naturaleza asíncrona de **Node.js** y luego desarrollaremos el concepto de **Reactor Pattern**.
- Este es un patrón de diseño de software ampliamente utilizado en teoría de redes, cuya comprensión no es fácil leyéndolo en dicho contexto.

Ejemplo inicial: Código

```
"use strict";

const fs = require("fs");

// Definimos funcion de lectura.
var onRead = function(err, data) {
  if (err)
    return console.log(err);

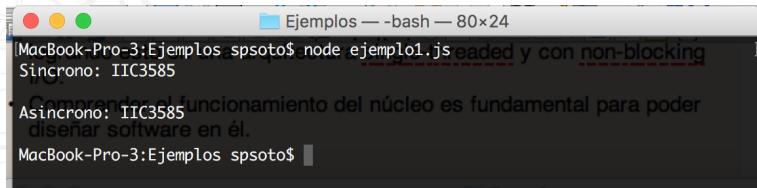
  console.log("Asincrono: " + data.toString());
};

// 1. Ejecutaremos asincrono
fs.readFile('input.txt', onRead);

// 2. Ejecutamos sincrono
var data = fs.readFileSync('input.txt');
console.log("Sincrono: " + data.toString());
```

Output

Output obtenido:

A terminal window titled 'Ejemplos — -bash — 80x24' is shown. The prompt is 'MacBook-Pro-3:Ejemplos spsoto\$'. The command 'node ejemplo1.js' has been executed, resulting in the output 'Sincrono: IIC3585'. A second prompt 'MacBook-Pro-3:Ejemplos spsoto\$' is visible at the bottom of the terminal. The background of the slide features a faint watermark of the University of Valencia seal.

```
MacBook-Pro-3:Ejemplos spsoto$ node ejemplo1.js
Sincrono: IIC3585
MacBook-Pro-3:Ejemplos spsoto$
```


¿Por qué la llamada I/O síncrona fue atendida primero que la asíncrona si la primera fue invocada antes?

Podemos responder esta pregunta entiendo el funcionamiento de la arquitectura de Node.js.

Conceptos previos

Motivación

Conceptos previos

Cuello de botella: operaciones I/O

Blocking I/O

Non-blocking I/O

Event demultiplexing

Reactor Pattern

libuv, el engine de Node.js

Conclusiones

Operaciones I/O

RAM

Tiempo acceso ~ 1 ns.

BW \sim GB/s

HDD

Tiempo acceso ~ 1 ms.

BW \sim MB/s

- Operaciones I/O no son costosas computacionalmente pero existen retrasos desde el envío de la operación hasta que es completada.
- Factor humano: I/O de operaciones humanas es impredecible.
- Operaciones de red pueden ser aún más lentas.

Conclusión: ¡I/O es el cuello de botella!

Operaciones I/O

RAM

Tiempo acceso ~ 1 ns.

BW \sim GB/s

HDD

Tiempo acceso ~ 1 ms.

BW \sim MB/s

- Operaciones I/O no son costosas computacionalmente pero existen **retrasos** desde el envío de la operación hasta que es completada.
- Factor humano: I/O de operaciones humanas es impredecible.
- Operaciones de red pueden ser aún más lentas.

Conclusión: ¡I/O es el cuello de botella!

Operaciones I/O

RAM

Tiempo acceso ~ 1 ns.

BW \sim GB/s

HDD

Tiempo acceso ~ 1 ms.

BW \sim MB/s

- Operaciones I/O no son costosas computacionalmente pero existen **retrasos** desde el envío de la operación hasta que es completada.
- Factor humano: I/O de operaciones humanas es impredecible.
- Operaciones de red pueden ser aún más lentas.

Conclusión: ¡I/O es el cuello de botella!

Operaciones I/O

RAM

Tiempo acceso ~ 1 ns.

BW \sim GB/s

HDD

Tiempo acceso ~ 1 ms.

BW \sim MB/s

- Operaciones I/O no son costosas computacionalmente pero existen **retrasos** desde el envío de la operación hasta que es completada.
- Factor humano: I/O de operaciones humanas es impredecible.
- Operaciones de red pueden ser aún más lentas.

Conclusión: ¡I/O es el cuello de botella!

Operaciones I/O

RAM

Tiempo acceso ~ 1 ns.

BW \sim GB/s

HDD

Tiempo acceso ~ 1 ms.

BW \sim MB/s

- Operaciones I/O no son costosas computacionalmente pero existen **retrasos** desde el envío de la operación hasta que es completada.
- Factor humano: I/O de operaciones humanas es impredecible.
- Operaciones de red pueden ser aún más lentas.

Conclusión: ¡I/O es el cuello de botella!

Blocking I/O

- En programación blocking, la función asociada a la operación I/O bloqueará la ejecución del thread hasta que la operación finalice.

Pseudocódigo de ejemplo:

```
data = socket.read();  
// datos están disponibles  
print(data);
```

- Un servidor implementado de esta forma no puede manejar múltiples conexiones en cada thread.
- Por esta razón tradicionalmente se inicia un proceso o hilo para cada conexión concurrente.

Blocking I/O

- En programación blocking, la función asociada a la operación I/O bloqueará la ejecución del thread hasta que la operación finalice.

Pseudocódigo de ejemplo:

```
data = socket.read();  
// datos están disponibles  
print(data);
```

- Un servidor implementado de esta forma no puede manejar múltiples conexiones en cada thread.
- Por esta razón tradicionalmente se inicia un proceso o hilo para cada conexión concurrente.

Blocking I/O

- En programación blocking, la función asociada a la operación I/O bloqueará la ejecución del thread hasta que la operación finalice.

Pseudocódigo de ejemplo:

```
data = socket.read();  
// datos están disponibles  
print(data);
```

- Un servidor implementado de esta forma no puede manejar múltiples conexiones en cada thread.
- Por esta razón tradicionalmente se inicia un proceso o hilo para cada conexión concurrente.

Blocking I/O

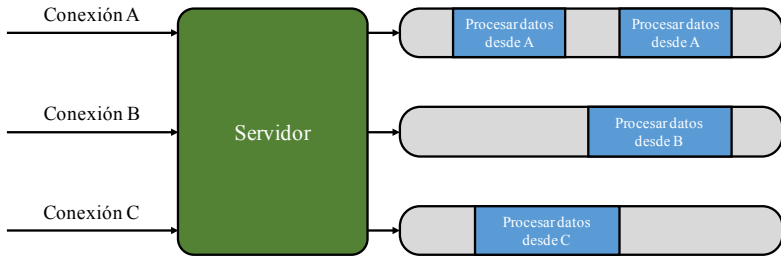
- En programación blocking, la función asociada a la operación I/O bloqueará la ejecución del thread hasta que la operación finalice.

Pseudocódigo de ejemplo:

```
data = socket.read();  
// datos están disponibles  
print(data);
```

- Un servidor implementado de esta forma no puede manejar múltiples conexiones en cada thread.
- Por esta razón tradicionalmente se inicia un proceso o hilo para cada conexión concurrente.

Modelo tradicional



Tiempo marcado en gris es tiempo desperciado (**¡ineficiencia!**)

Cada thread consume recursos y no todas las librerías son thread-safe.

Non-blocking I/O

- En este tipo de operaciones el llamado retorna inmediatamente sin esperar la lectura o escritura de datos.
- Si no hay resultados al momento de la llamada, se retorna una constante indicando que no hay datos disponibles.
- El método tradicional para leer información de esta forma se conoce como **busy-wait**, en el que se lee la función de forma iterativa en un loop.

Non-blocking I/O

- En este tipo de operaciones el llamado retorna inmediatamente sin esperar la lectura o escritura de datos.
- Si no hay resultados al momento de la llamada, se retorna una constante indicando que no hay datos disponibles.
- El método tradicional para leer información de esta forma se conoce como **busy-wait**, en el que se lee la función de forma iterativa en un loop.

Non-blocking I/O

- En este tipo de operaciones el llamado retorna inmediatamente sin esperar la lectura o escritura de datos.
- Si no hay resultados al momento de la llamada, se retorna una constante indicando que no hay datos disponibles.
- El método tradicional para leer información de esta forma se conoce como **busy-wait**, en el que se lee la función de forma iterativa en un loop.

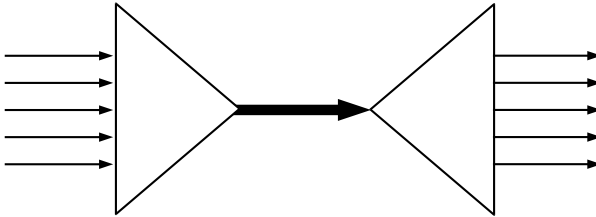
Busy-wait

Pseudo-código de ejemplo:

```
do:
  let data = resource.read()
  if data === NO_DATA_AVAILABLE:
    continue
  else if data === RESOURCE_CLOSED:
    break
  else:
    consume(data)
while true
```


Demultiplexing

Concepto utilizado en teoría de comunicaciones



Event demultiplexing

- Los sistemas operativos usan este sistema, llamado **synchronous event demultiplexer** ó **event notification interface**.
- Se recolectan eventos I/O que vienen desde un conjunto de recursos observados, y bloquea hasta que haya nuevos eventos disponibles para ser procesados.

Event demultiplexing

- Los sistemas operativos usan este sistema, llamado **synchronous event demultiplexer** ó **event notification interface**.
- Se recolectan eventos I/O que vienen desde un conjunto de recursos observados, y bloquea hasta que haya nuevos eventos disponibles para ser procesados.

Event demultiplexing

Pseudo-código de ejemplo:

```
list.push(socketA, READ) // [1]
list.push(pipeB, READ)

while events = demultiplexer.watch(list): // [2]
    for event of events:
        data = event.resource.read() // [3]

        if data === RESOURCE_CLOSED:
            demultiplexer.unwatch(event.resource)
        else:
            consume(data)
```

Event demultiplexing

Pseudo-código de ejemplo:

```
list.push(socketA, READ) // [1]
list.push(pipeB, READ)

while events = demultiplexer.watch(list): // [2]
    for event of events:
        data = event.resource.read() // [3]

        if data == RESOURCE_CLOSED:
            demultiplexer.unwatch(event.resource)
        else:
            consume(data)
```

Event demultiplexing

Pseudo-código de ejemplo:

```
list.push(socketA, READ) // [1]
list.push(pipeB, READ)

while events = demultiplexer.watch(list): // [2]
    for event of events:
        data = event.resource.read() // [3]

        if data == RESOURCE_CLOSED:
            demultiplexer.unwatch(event.resource)
        else:
            consume(data)
```

Event demultiplexing

Pseudo-código de ejemplo:

```
list.push(socketA, READ) // [1]
list.push(pipeB, READ)

while events = demultiplexer.watch(list): // [2]
    for event of events:
        data = event.resource.read() // [3]

        if data === RESOURCE_CLOSED:
            demultiplexer.unwatch(event.resource)
        else:
            consume(data)
```

Event demultiplexing

Analicemos el código anterior:

- [1] Recursos agregados a la estructura de datos.
- [2] Se le indica al notificador de eventos la lista que debe observar. **Esta función es síncrona y se bloquea hasta que haya recursos listos para ser leídos.** Cuando esto sucede el método retorna y se define la lista de eventos a procesar.
- [3] **Loop de eventos:** Cada evento del demultiplexor es procesado. Se garantiza que cada evento tendrá lectura de datos. Una vez hecho, se vuelve a esperar hasta que hayan nuevos eventos listos.

Event demultiplexing

Analicemos el código anterior:

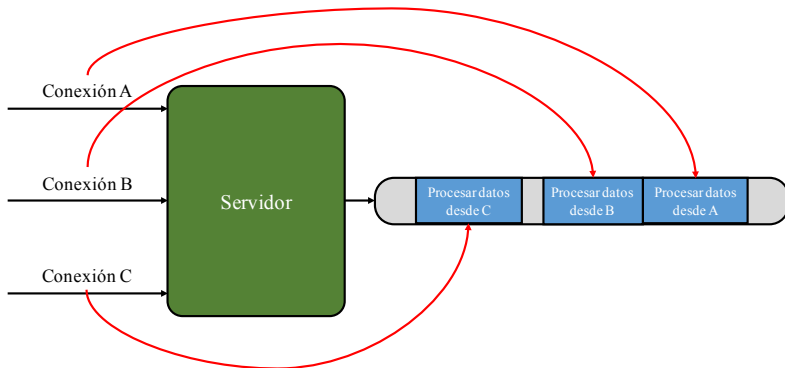
- [1] Recursos agregados a la estructura de datos.
- [2] Se le indica al notificador de eventos la lista que debe observar. **Esta función es síncrona y se bloquea hasta que haya recursos listos para ser leídos.** Cuando esto sucede el método retorna y se define la lista de eventos a procesar.
- [3] **Loop de eventos:** Cada evento del demultiplexor es procesado. Se garantiza que cada evento tendrá lectura de datos. Una vez hecho, se vuelve a esperar hasta que hayan nuevos eventos listos.

Event demultiplexing

Analicemos el código anterior:

- [1] Recursos agregados a la estructura de datos.
- [2] Se le indica al notificador de eventos la lista que debe observar. **Esta función es síncrona y se bloquea hasta que haya recursos listos para ser leídos.** Cuando esto sucede el método retorna y se define la lista de eventos a procesar.
- [3] **Loop de eventos:** Cada evento del demultiplexor es procesado. Se garantiza que cada evento tendrá lectura de datos. Una vez hecho, se vuelve a esperar hasta que hayan nuevos eventos listos.

Diagrama resultante



- Se puede trabajar en un solo thread concurrentemente.
- Se minimiza el tiempo en reposo del thread.
- Desaparecen condiciones de sincronización y race conditions.

Reactor Pattern

Motivación

Conceptos previos

Reactor Pattern

Definiciones

Diagrama de funcionamiento

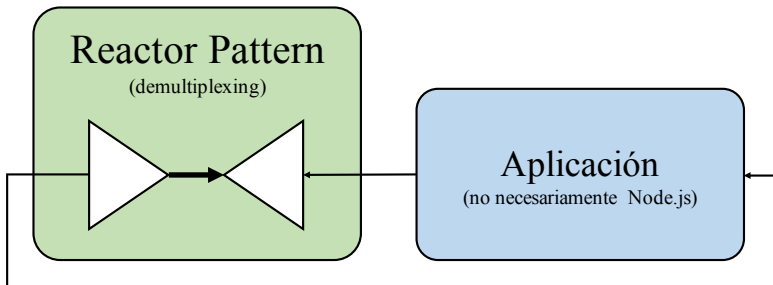
Consecuencias

libuv, el engine de Node.js

Conclusiones

Reactor pattern

El **reactor pattern** es un patrón de diseño de software en que se especializa el algoritmo de demultiplexación de eventos descrito anteriormente que se puede resumir en el siguiente diagrama de bloques:



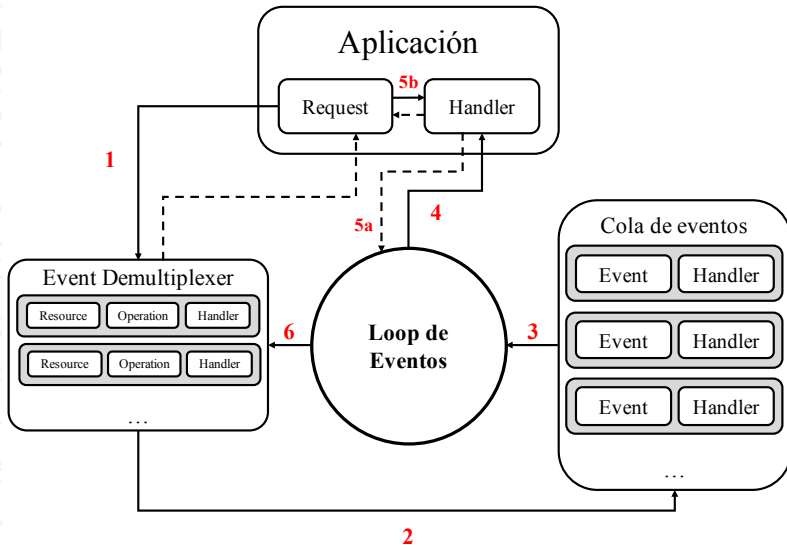
Requests y handlers

- **Request:** cada operación I/O que se desea realizar en la aplicación. En **Node.js**, puede ser una respuesta HTTP, una conexión a un socket, una query en una base de datos, etc.
- **Handler:** bloque de código que maneja los datos obtenidos mediante un request. En el contexto de **Node.js** son funciones de **callback** asociadas a la operación I/O.

Requests y handlers

- **Request:** cada operación I/O que se desea realizar en la aplicación. En **Node.js**, puede ser una respuesta HTTP, una conexión a un socket, una query en una base de datos, etc.
- **Handler:** bloque de código que maneja los datos obtenidos mediante un request. En el contexto de **Node.js** son funciones de **callback** asociadas a la operación I/O.

Diagrama funcional



Funcionamiento

1. Aplicación genera nueva operación I/O (**request**) y la envía al **Event Demultiplexer**, especificando un callback (**handler**). Esta operación es non-blocking.
2. Cuando un conjunto de operaciones I/O finaliza se envían a la cola de eventos (**Event Queue**).
3. El loop de eventos itera sobre la cola de eventos.

Funcionamiento

1. Aplicación genera nueva operación I/O (**request**) y la envía al **Event Demultiplexer**, especificando un callback (**handler**). Esta operación es non-blocking.
2. Cuando un conjunto de operaciones I/O finaliza se envían a la cola de eventos (**Event Queue**).
3. El loop de eventos itera sobre la cola de eventos.

Funcionamiento

1. Aplicación genera nueva operación I/O (**request**) y la envía al **Event Demultiplexer**, especificando un callback (**handler**). Esta operación es non-blocking.
2. Cuando un conjunto de operaciones I/O finaliza se envían a la cola de eventos (**Event Queue**).
3. El loop de eventos itera sobre la cola de eventos.

Funcionamiento (cont.)

4. Se invoca el handler asociado a cada evento.
5. Pueden suceder dos eventos (no excluyentes):
 - (a) Handler devuelve control al loop de eventos.
 - (b) Pueden aparecer nuevas operaciones I/O.
6. Cuando todos los elementos son procesados el loop se bloquea nuevamente. Se inicia un nuevo ciclo cuando el **Event Demultiplexer** lo inicia.

Funcionamiento (cont.)

4. Se invoca el handler asociado a cada evento.
5. Pueden suceder dos eventos (no excluyentes):
 - (a) Handler devuelve control al loop de eventos.
 - (b) Pueden aparecer nuevas operaciones I/O.
6. Cuando todos los elementos son procesados el loop se bloquea nuevamente. Se inicia un nuevo ciclo cuando el **Event Demultiplexer** lo inicia.

Funcionamiento (cont.)

4. Se invoca el handler asociado a cada evento.
5. Pueden suceder dos eventos (no excluyentes):
 - (a) Handler devuelve control al loop de eventos.
 - (b) Pueden aparecer nuevas operaciones I/O.
6. Cuando todos los elementos son procesados el loop se bloquea nuevamente. Se inicia un nuevo ciclo cuando el **Event Demultiplexer** lo inicia.

Observaciones

- Funcionamiento queda claro: ninguna acceso a un recurso bloquea el código y entrega un callback a ejecutar cuando la operación finaliza.
- Por lo general el loop de eventos se ejecuta en otro thread.
- Una aplicación de **Node.js** finaliza automáticamente cuando no hay más operaciones pendientes en el Event Demultiplexer y no hay más eventos que procesar en la cola de eventos.

Observaciones

- Funcionamiento queda claro: ninguna acceso a un recurso bloquea el código y entrega un callback a ejecutar cuando la operación finaliza.
- Por lo general el loop de eventos se ejecuta en otro thread.
- Una aplicación de **Node.js** finaliza automáticamente cuando no hay más operaciones pendientes en el Event Demultiplexer y no hay más eventos que procesar en la cola de eventos.

Observaciones

- Funcionamiento queda claro: ninguna acceso a un recurso bloquea el código y entrega un callback a ejecutar cuando la operación finaliza.
- Por lo general el loop de eventos se ejecuta en otro thread.
- Una aplicación de **Node.js** finaliza automáticamente cuando no hay más operaciones pendientes en el Event Demultiplexer y no hay más eventos que procesar en la cola de eventos.

Ventajas

- ✓ **Modularidad:** composición sencilla de cada uno de los módulos a ser llamados por el reactor.
- ✓ **Stateless:** los handlers o callbacks no deben preocuparse de cómo los eventos son despachados. Se desacoplan mecanismos independientes de la aplicación de sus políticas específicas.

Ventajas

- ✓ **Modularidad:** composición sencilla de cada uno de los módulos a ser llamados por el reactor.
- ✓ **Stateless:** los handlers o callbacks no deben preocuparse de cómo los eventos son despachados. Se desacoplan mecanismos independientes de la aplicación de sus políticas específicas.

Desventajas

- ✗ **Curva de aprendizaje:** Aprendizaje no es sencillo ni intuitivo al momento de diseñar interacciones complejas entre diversos handlers.
- ✗ **Difícil de depurar:** causa de llamado a callbacks no es fácil de esclarecer. Bugs no son fácilmente replicables.

Desventajas

- ✗ **Curva de aprendizaje:** Aprendizaje no es sencillo ni intuitivo al momento de diseñar interacciones complejas entre diversos handlers.
- ✗ **Difícil de depurar:** causa de llamado a callbacks no es fácil de esclarecer. Bugs no son fácilmente replicables.

libuv, el engine de Node.js

Motivación

Conceptos previos

Reactor Pattern

libuv, el engine de Node.js

¿Qué es libuv?

Entendiendo el núcleo de Node

Conclusiones

¿Qué es libuv?

- Cada OS tiene su Event Demultiplexer: **epoll** (Linux), **kqueue** (macOS), **IOCP** (Windows).
- Sin embargo, las operaciones I/O se comportan distinto:
 - **Ejemplo:** En Unix no se permiten operaciones non-blocking en el sistema de archivos.
- El equipo de **Node.js** crea una librería en C llamada **libuv**.

¿Qué es libuv?

- ▶ Cada OS tiene su Event Demultiplexer: **epoll** (Linux), **kqueue** (macOS), **IOCP** (Windows).
- ▶ Sin embargo, las operaciones I/O se comportan distinto:
 - ▶ **Ejemplo:** En Unix no se permiten operaciones non-blocking en el sistema de archivos.
- ▶ El equipo de **Node.js** crea una librería en C llamada **libuv**.

¿Qué es libuv?

- ▶ Cada OS tiene su Event Demultiplexer: **epoll** (Linux), **kqueue** (macOS), **IOCP** (Windows).
- ▶ Sin embargo, las operaciones I/O se comportan distinto:
 - ▶ **Ejemplo:** En Unix no se permiten operaciones non-blocking en el sistema de archivos.
- ▶ El equipo de **Node.js** crea una librería en C llamada **libuv**.

¿Qué es libuv? (cont.)

libuv es una abstracción de alto nivel para hacer Node.js compatible con todas las plataformas y cuyas tareas son:

1. Normalizar el comportamiento non-blocking de los distintos recursos.
2. Implementar **Reactor** sobre el **Event Demultiplexer** de cada OS.
3. Convertirse en el motor I/O de bajo nivel de Node.js.
4. Permitir la creación de loops de eventos y gestionar la cola de eventos.

¿Qué es libuv? (cont.)

libuv es una abstracción de alto nivel para hacer Node.js compatible con todas las plataformas y cuyas tareas son:

1. Normalizar el comportamiento non-blocking de los distintos recursos.
2. Implementar **Reactor** sobre el **Event Demultiplexer** de cada OS.
3. Convertirse en el motor I/O de bajo nivel de Node.js.
4. Permitir la creación de loops de eventos y gestionar la cola de eventos.

¿Qué es libuv? (cont.)

libuv es una abstracción de alto nivel para hacer Node.js compatible con todas las plataformas y cuyas tareas son:

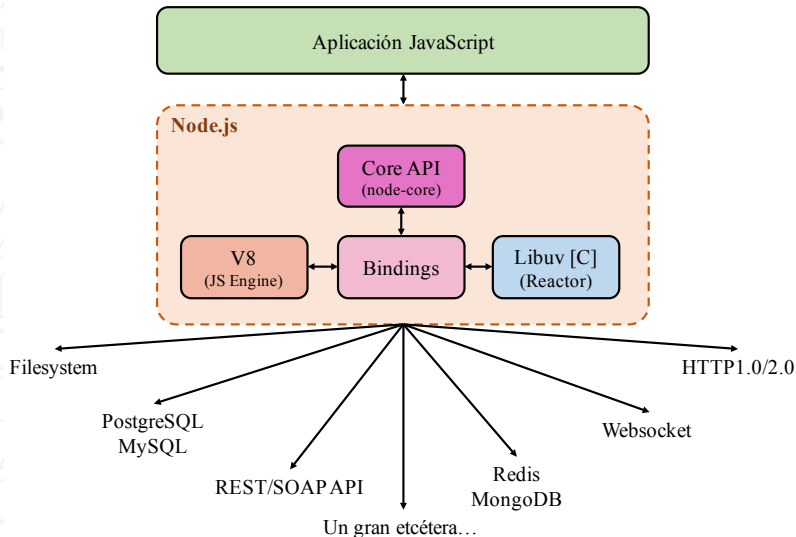
1. Normalizar el comportamiento non-blocking de los distintos recursos.
2. Implementar **Reactor** sobre el **Event Demultiplexer** de cada OS.
3. Convertirse en el motor I/O de bajo nivel de Node.js.
4. Permitir la creación de loops de eventos y gestionar la cola de eventos.

¿Qué es libuv? (cont.)

libuv es una abstracción de alto nivel para hacer Node.js compatible con todas las plataformas y cuyas tareas son:

1. Normalizar el comportamiento non-blocking de los distintos recursos.
2. Implementar **Reactor** sobre el **Event Demultiplexer** de cada OS.
3. Convertirse en el motor I/O de bajo nivel de Node.js.
4. Permitir la creación de loops de eventos y gestionar la cola de eventos.

Diagrama funcional de Node.js



Conclusiones

Motivación

Conceptos previos

Reactor Pattern

libuv, el engine de Node.js

Conclusiones

Volviendo al ejemplo

Conclusiones

Aplicación interesante: Nodecopter

Código de ejemplo

- **Node.js** presume que todas las operaciones I/O son non-blocking. Por lo tanto, el script principal tiene prioridad por sobre el loop de eventos.
- En consecuencia, primero debe terminar de ejecutarse lo síncrono y luego puede ejecutarse el callback `onRead()`.
- Es decir, puede resultar complicado tener claro cuáles callbacks se ejecutarán primero en el loop de eventos.

Conclusión: ¡mucho cuidado con funciones que no son puras!

Código de ejemplo

- **Node.js** presume que todas las operaciones I/O son non-blocking. Por lo tanto, el script principal tiene prioridad por sobre el loop de eventos.
- En consecuencia, primero debe terminar de ejecutarse lo síncrono y luego puede ejecutarse el callback `onRead()`.
- Es decir, puede resultar complicado tener claro cuáles callbacks se ejecutarán primero en el loop de eventos.

Conclusión: ¡mucho cuidado con funciones que no son puras!

Código de ejemplo

- **Node.js** presume que todas las operaciones I/O son non-blocking. Por lo tanto, el script principal tiene prioridad por sobre el loop de eventos.
- En consecuencia, primero debe terminar de ejecutarse lo síncrono y luego puede ejecutarse el callback `onRead()`.
- Es decir, puede resultar complicado tener claro cuáles callbacks se ejecutarán primero en el loop de eventos.

Conclusión: ¡mucho cuidado con funciones que no son puras!

Código de ejemplo

- **Node.js** presume que todas las operaciones I/O son non-blocking. Por lo tanto, el script principal tiene prioridad por sobre el loop de eventos.
- En consecuencia, primero debe terminar de ejecutarse lo síncrono y luego puede ejecutarse el callback `onRead()`.
- Es decir, puede resultar complicado tener claro cuáles callbacks se ejecutarán primero en el loop de eventos.

Conclusión: ¡mucho cuidado con funciones que no son puras!

Conclusiones

- Las operaciones I/O son el cuello de botella de cualquier aplicación. Operaciones blocking generan tiempo muerto en los hilos de un programa.
- Operaciones non-blocking solucionan este problema, pero deben ser gestionadas eficientemente.
- Cada OS posee su propio **Event Demultiplexer**, una implementación eficiente para gestionar operaciones non-blocking.

Conclusiones

- Las operaciones I/O son el cuello de botella de cualquier aplicación. Operaciones blocking generan tiempo muerto en los hilos de un programa.
- Operaciones non-blocking solucionan este problema, pero deben ser gestionadas eficientemente.
- Cada OS posee su propio **Event Demultiplexer**, una implementación eficiente para gestionar operaciones non-blocking.

Conclusiones

- ▶ Las operaciones I/O son el cuello de botella de cualquier aplicación. Operaciones blocking generan tiempo muerto en los hilos de un programa.
- ▶ Operaciones non-blocking solucionan este problema, pero deben ser gestionadas eficientemente.
- ▶ Cada OS posee su propio **Event Demultiplexer**, una implementación eficiente para gestionar operaciones non-blocking.

Conclusiones (cont.)

- **libuv** genera una abstracción en C para cada plataforma e implementa el Reactor Pattern sobre el Event Demultiplexer de cada OS.
- **Node.js** es un framework que une libuv con el V8 Engine de Javascript para generar un punto de confluencia de diversas operaciones I/O en alto nivel, con alta velocidad de respuesta y gran eficiencia.
- El Reactor Pattern es el núcleo de **Node.js**, por lo cual es fundamental comprenderlo para comprender su funcionamiento.

Conclusiones (cont.)

- **libuv** genera una abstracción en C para cada plataforma e implementa el Reactor Pattern sobre el Event Demultiplexer de cada OS.
- **Node.js** es un framework que une libuv con el V8 Engine de Javascript para generar un punto de confluencia de diversas operaciones I/O en alto nivel, con alta velocidad de respuesta y gran eficiencia.
- El Reactor Pattern es el núcleo de **Node.js**, por lo cual es fundamental comprenderlo para comprender su funcionamiento.

Conclusiones (cont.)

- ▶ **libuv** genera una abstracción en C para cada plataforma e implementa el Reactor Pattern sobre el Event Demultiplexer de cada OS.
- ▶ **Node.js** es un framework que une libuv con el V8 Engine de Javascript para generar un punto de confluencia de diversas operaciones I/O en alto nivel, con alta velocidad de respuesta y gran eficiencia.
- ▶ El **Reactor Pattern** es el núcleo de **Node.js**, por lo cual es fundamental comprenderlo para comprender su funcionamiento.

Nodecopter



www.nodecopter.com

¡Node.js en sistemas embebidos!

- Sistemas embebidos son de naturaleza asíncrona por definición, pues responden a eventos no determinísticos del medio.
- En el caso del Drone: control externo, movimiento de motores, giroscopios, I/O cámara, etc...

¿Por qué no usar **Node.js** para estos propósitos?

- Parrot AR Drone 2.0 + Linux (BusyBox) + **Node.js**.
- `npm install ar-drone` y a entretenernos...

¡Node.js en sistemas embebidos!

- Sistemas embebidos son de naturaleza asíncrona por definición, pues responden a eventos no determinísticos del medio.
- En el caso del Drone: control externo, movimiento de motores, giroscopios, I/O cámara, etc...

¿Por qué no usar **Node.js** para estos propósitos?

- Parrot AR Drone 2.0 + Linux (BusyBox) + **Node.js**.
- `npm install ar-drone` y a entretenernos...

¡Node.js en sistemas embebidos!

- Sistemas embebidos son de naturaleza asíncrona por definición, pues responden a eventos no determinísticos del medio.
- En el caso del Drone: control externo, movimiento de motores, giroscopios, I/O cámara, etc...

¿Por qué no usar **Node.js** para estos propósitos?

- Parrot AR Drone 2.0 + Linux (BusyBox) + **Node.js**.
- `npm install ar-drone` y a entretenernos...

¡Node.js en sistemas embebidos!

- Sistemas embebidos son de naturaleza asíncrona por definición, pues responden a eventos no determinísticos del medio.
- En el caso del Drone: control externo, movimiento de motores, giroscopios, I/O cámara, etc...

¿Por qué no usar **Node.js** para estos propósitos?

- Parrot AR Drone 2.0 + Linux (BusyBox) + **Node.js**.
- `npm install ar-drone` y a entretenernos...

Código de ejemplo

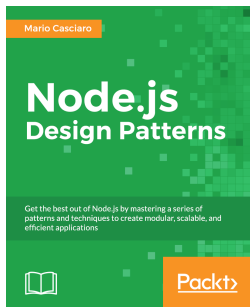
```
var arDrone = require('ar-drone');
var client = arDrone.createClient();

client.takeoff();

client
  .after(5000, function() {
    this.clockwise(0.5);
  })
  .after(3000, function() {
    this.animate('flipLeft', 15);
  })
  .after(1000, function() {
    this.stop();
    this.land();
  });
```

Texto base

Trabajo basado en **Node.js Design Patterns**, de Mario Casciaro, Packt Publishing (2014).



Explicaciones fueron simplificadas, complementadas, y condimentadas.

Otras referencias

- [Reactor Pattern a nivel de patrones de diseño de software.](#)
- [Reactor. An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events.](#) Paper del autor que lo diseñó.
- [Blog con implementación en Java del Reactor Pattern.](#)
- [Más información sobre Nodecopter.](#)