NODE.JS DESIGN PATTERNS

The main concepts

INTRODUCTION

- Major characteristics of Node.js programming
- Asynchronous nature
- Its programming style that makes heavy use of callbacks
- Its module system which allows multiple versions of the same dependency to coexist in an application, and
- The observer pattern, implemented by the EventEmitter class
  - It perfectly complements callbacks when dealing with asynchronous code

PROPERTIES

- The Node.js core itself has its foundations built on a few principles; one of these is, having the smallest set of functionality, leaving the rest to the so-called **userland** (or *userspace*)
- Userland refers to the ecosystem of modules living outside the core
- This principle has an enormous impact on the Node.js culture, as it gives freedom to the community to experiment and iterate fast on a broader set of solutions within the scope of the userland modules
- Keeping the core set of functionality to the bare minimum bears certain advantages
- It becomes convenient in terms of maintainability
- Leaves a positive cultural impact that it brings on the evolution of the entire ecosystem.

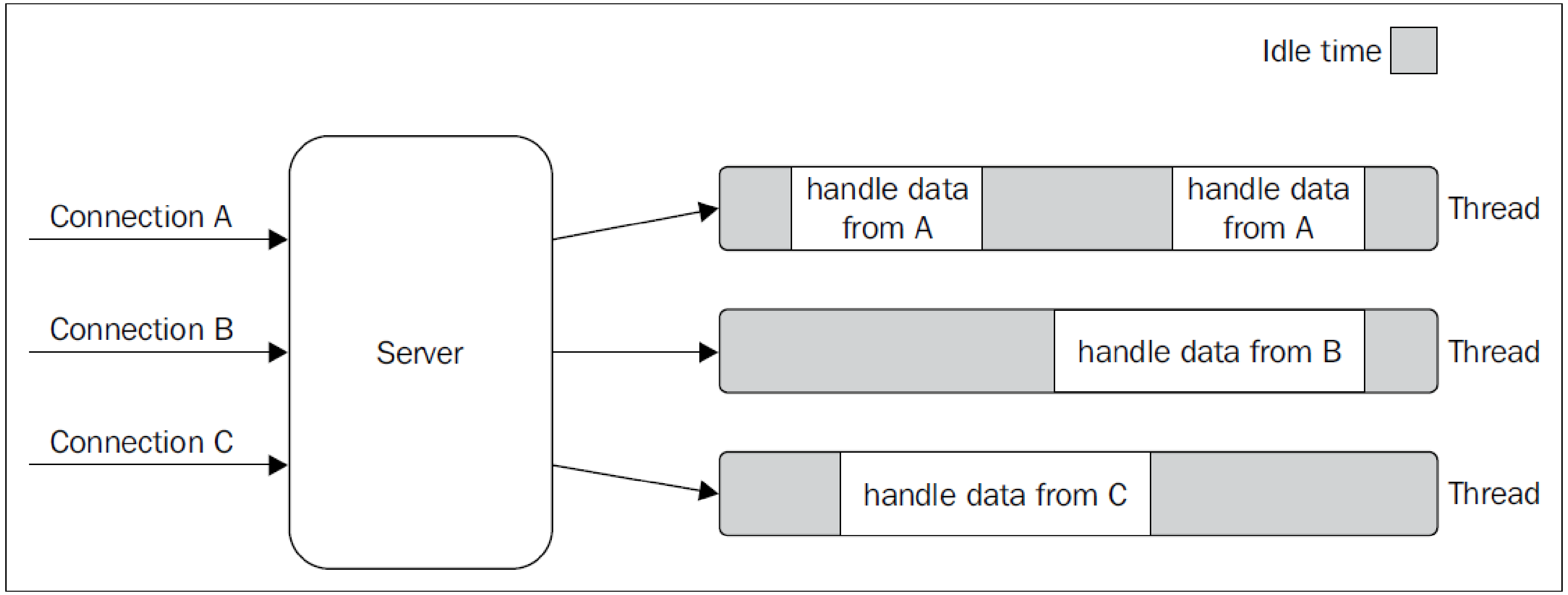
SLOW I/O

- I/O is definitely the slowest among the fundamental operations of a computer.
- Accessing the RAM is in the order of nanoseconds (10e-9 seconds), while accessing data on the disk or the network is in the order of milliseconds (10e-3 seconds).
- For the bandwidth, it is the same story; RAM has a transfer rate consistently in the order of GB/s, while disk and network varies from MB/s to, optimistically, GB/s.
- I/O is usually not expensive in terms of CPU, but it adds a delay between the moment the request is sent and the moment the operation completes.
- We also have to consider the *human factor*; often, the input of an application comes from a real person, for example, the click of a button or a message sent in a real-time chat application, so the speed and frequency of I/O don't depend only on technical aspects, and they can be many orders of magnitude slower than the disk or network.

SYNCHRONOUS I/O

```
//blocks the thread until the data is
available
data = socket.read();
//data is available
print(data);
```

It is trivial to notice that a web server that is implemented using blocking I/O will not be able to handle multiple connections in the same thread



This points to the problem of the amount of time each thread is idle, waiting for new data to be received from the associated connection

PROBLEMS WITH MULTI-THREADING

- a thread is not cheap in terms of system resources
- it consumes memory and causes context switches,
- Thus, having a long running thread for each connection and not using it for most of the time, is not the best compromise in terms of efficiency.

NON BLOCKING I/O

- The system call always returns immediately without waiting for the data to be read or written.
- If no results are available at the moment of the call, the function will simply return a predefined constant, indicating that there is no data available to return at that moment.
- The most basic pattern for accessing this kind of non-blocking I/O is to actively poll the resource within a loop until some actual data is returned; this is called **busy-waiting**.
- A busy-waiting loop will consume precious CPU only for iterating over resources that are unavailable most of the time.
- Polling algorithms usually result in a huge amount of wasted CPU time.

DEMULTIPLEXER

- Most modern operating systems provide a native mechanism to handle concurrent, non-blocking resources in an efficient way
- This mechanism is called **synchronous event demultiplexer** or **event notification interface**
- This component collects and queues I/O events that come from a set of watched resources, and block until new events are available to process

[1] The resources are added to a data structure, associating each one of them with a specific operation, in our example a read.

[2] The event notifier is set up with the group of resources to be watched.

This call is synchronous and blocks until any of the watched resources is ready for a read.

When this occurs, the event demultiplexer returns from the call and a new set of events is available to be processed.

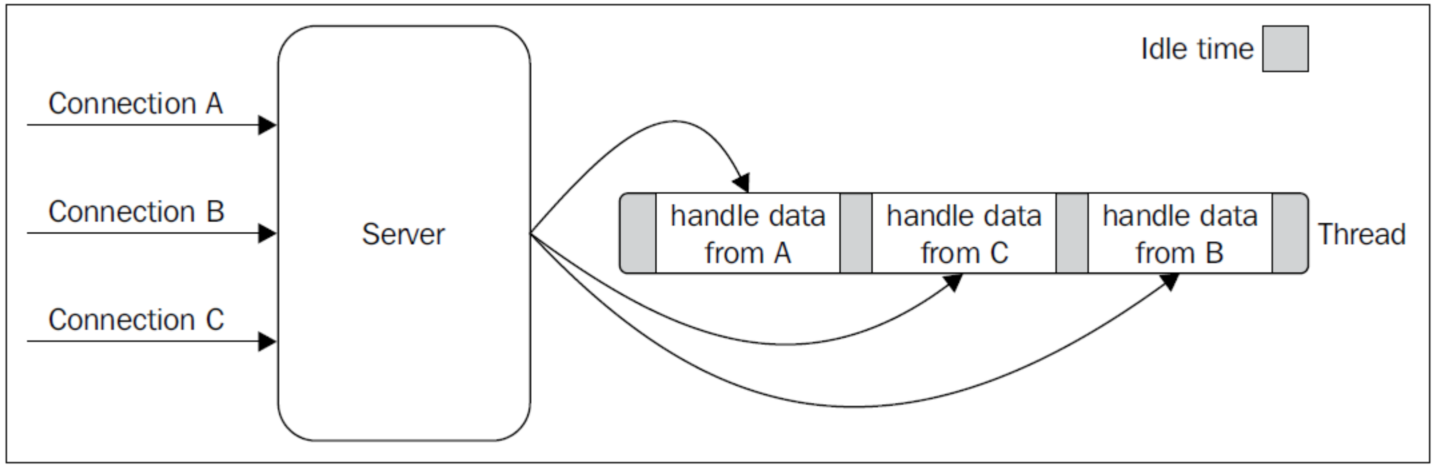
[3] Each event returned by the event demultiplexer is processed.

At this point, the resource associated with each event is guaranteed to be ready to read and to not block during the operation.

When all the events are processed, the flow will block again on the event demultiplexer until new events are again available to be processed. This is called the **event loop**.

```
 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);
}
}
```

HANDLING CONCURRENCY



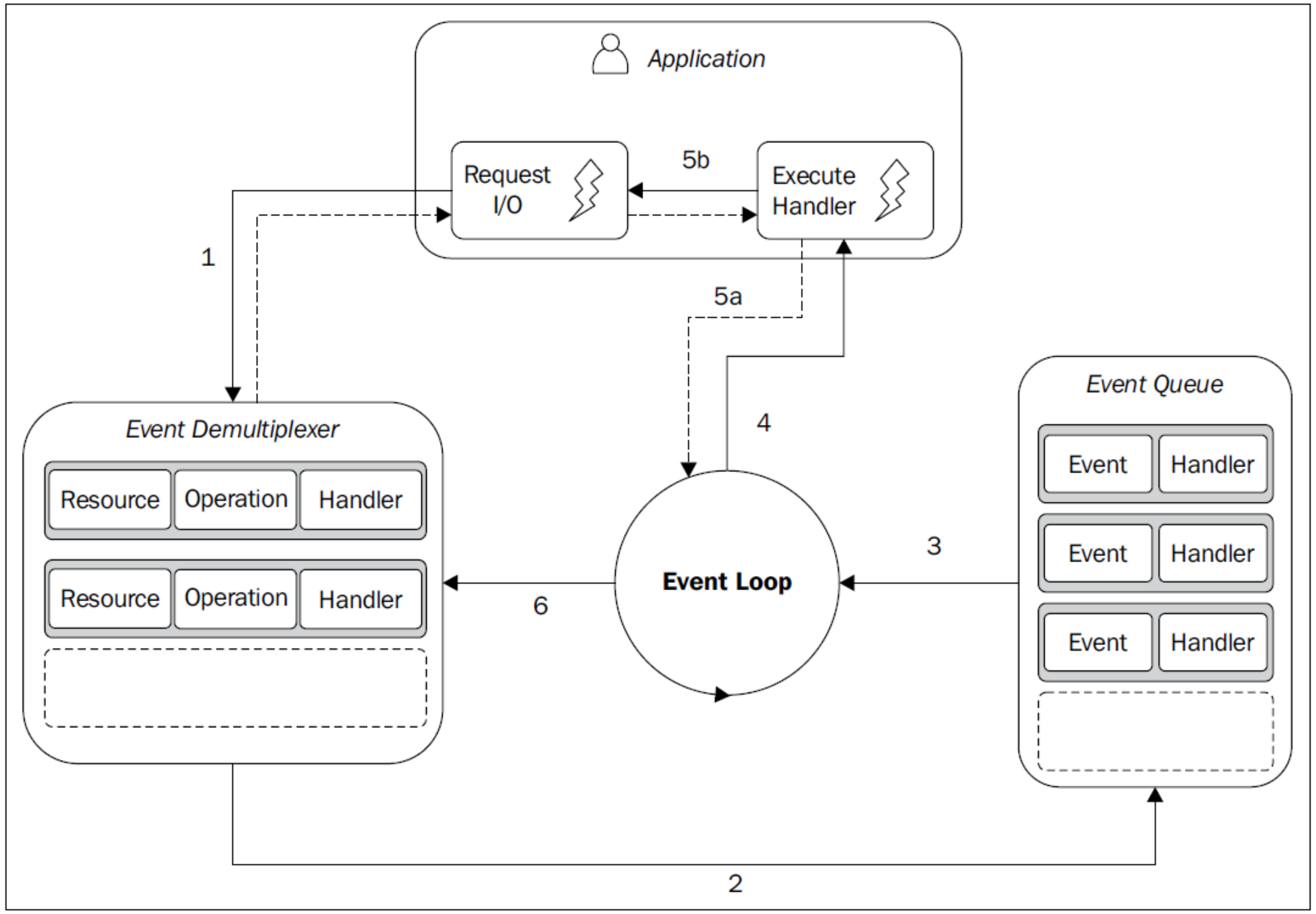
- ❑how concurrency works in a single-threaded application using a synchronous event demultiplexer and non-blocking I/O
- ❑We can see that using only one thread does not impair our ability to run multiple I/O bound tasks *concurrently*.
- ❑The tasks are spread over time, instead of being spread across multiple threads
- ❑the absence of in-process race conditions and multiple threads to synchronize, allows us to use much simpler concurrency strategies in Node.js

THE REACTOR PATTERN

The main idea behind it is to have a **handler** (which in Node.js is represented by a **callback** function) associated with each I/O operation, which will be invoked as soon as an event is produced and processed by the event loop

REACTOR PATTERN

- The application generates a new I/O operation by submitting a request to the **Event Demultiplexer**.
- The application also specifies a handler, which will be invoked when the operation completes.
- Submitting a new request to the Event Demultiplexer is a non-blocking call and it immediately returns the control back to the application.
- 2. When a set of I/O operations completes, the Event Demultiplexer pushes the new events into the **Event Queue**.



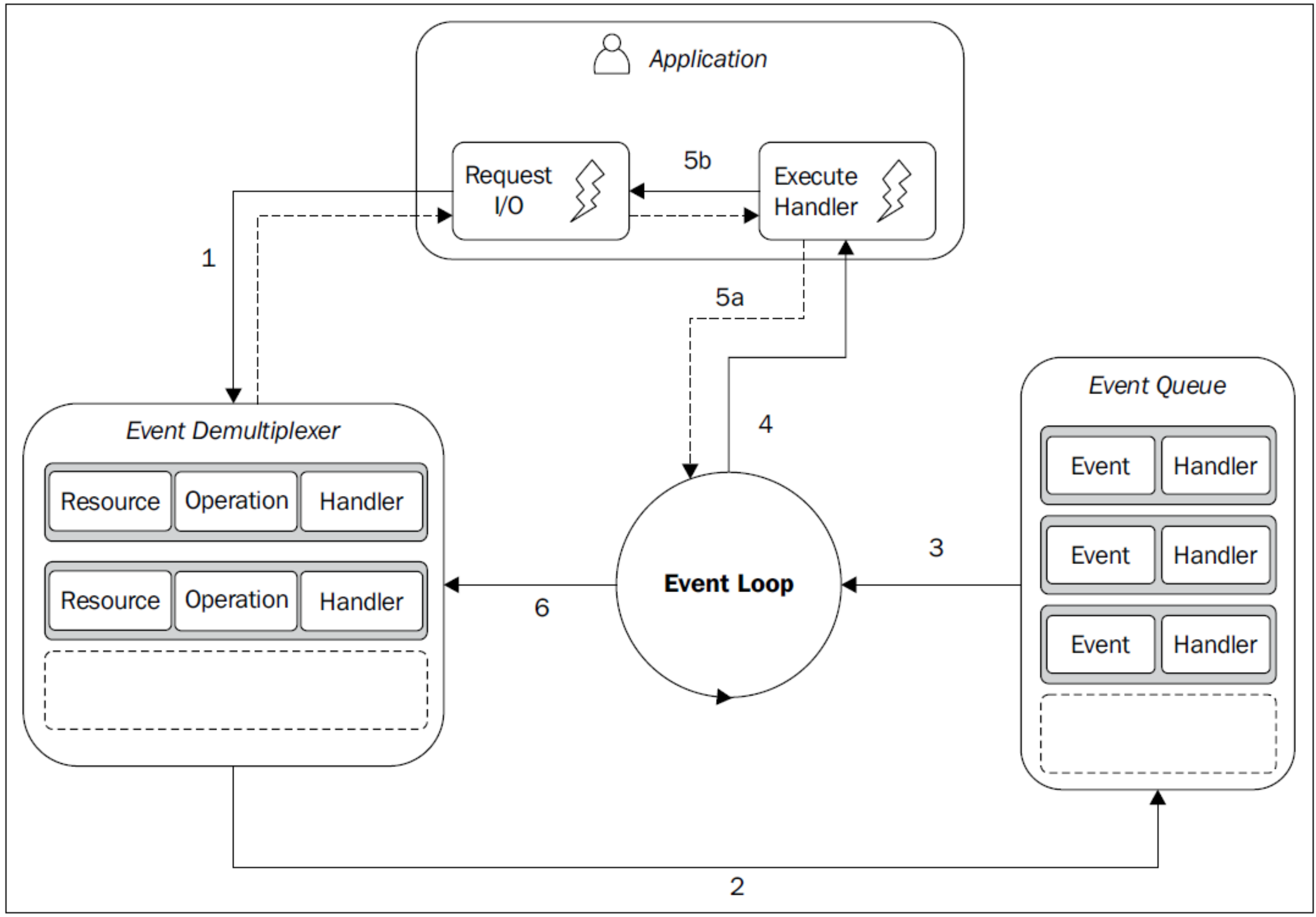
REACTOR PATTERN

3. At this point, the Event Loop iterates over the items of the Event Queue.

4. For each event, the associated handler is invoked.

5. The handler, which is part of the application code, will give back the control to the Event Loop when its execution completes (**5a**).

6. When all the items in the Event Queue are processed, the loop will block again on the Event Demultiplexer which will then trigger another cycle.



DEFINING THE REACTOR PATTERN

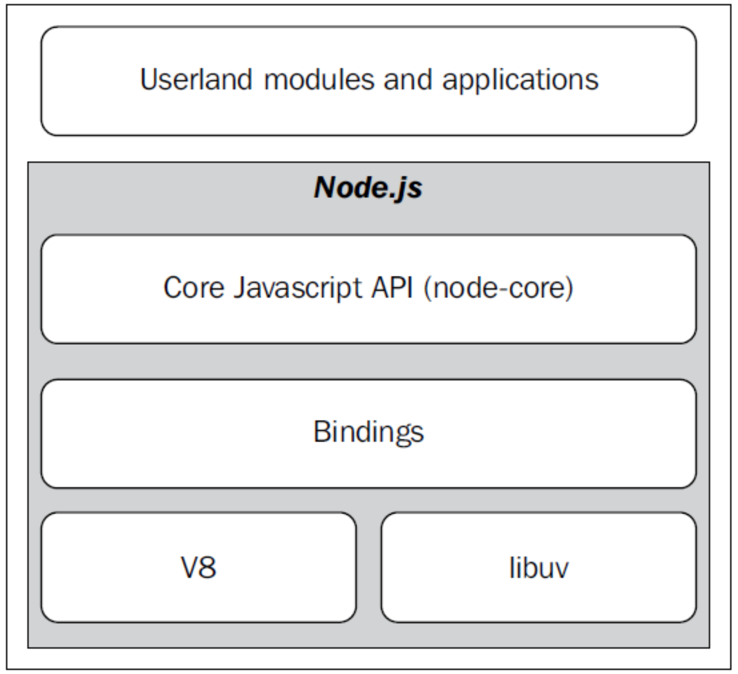
- A Node.js application will exit automatically when there are no more pending operations in the Event Demultiplexer, and no more events to be processed inside the Event Queue.
- Pattern (reactor): handles I/O by blocking until new events are available from a set of observed resources, and then reacting by dispatching each event to an associated handler.

IMPLEMENTING THE REACTOR PATTERN

- ❑Each operating system has its own interface for the Event Demultiplexer
- ❑`epoll` on Linux, `kqueue` on Mac OS X, and **I/O Completion Port API** (**IOCP**) on Windows.
- ❑Besides that, each I/O operation can behave quite differently depending on the type of the resource, even within the same OS.
- ❑All these inconsistencies across and within the different operating systems required a higher-level abstraction to be built for the Event Demultiplexer.
- ❑This is why the Node.js core team created a C library called `libuv`, with the objective to make Node.js compatible with all the major platforms and normalize the non-blocking behavior of the different types of resource; `libuv` today represents the low-level I/O engine of Node.js.
- ❑Besides abstracting the underlying system calls, `libuv` also implements the reactor pattern, thus providing an API for creating event loops, managing the event queue, running asynchronous I/O operations, and queuing other types of tasks.

THE NODE.JS RECIPE

- ❑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.


Userland modules and applications
Node.js
Core Javascript API (node-core)
Bindings
V8
libuv

CONTINUATION PASSING STYLE

- A **synchronous** function blocks until it completes its operations.
- An **asynchronous** function returns immediately and the result is passed to a handler (in our case, a callback) at a later cycle of the event loop.

- In JavaScript, a callback is a function that is passed as an argument to another function and is invoked with the result when the operation completes.
- In functional programming, this way of propagating the result is called **continuation-passing style**, for brevity, **CPS.**
- it simply indicates that a result is propagated by passing it to another function (the callback), instead of directly returning it to the caller
- Direct Style-using a return statement

```
function add(a, b) {
return a + b;
}
```

- CPS Style

```
function add(a, b, callback)
{
callback(a + b);
}
```

- The `add()` function is a synchronous CPS function, which means that it will return a value only when the callback completes its execution

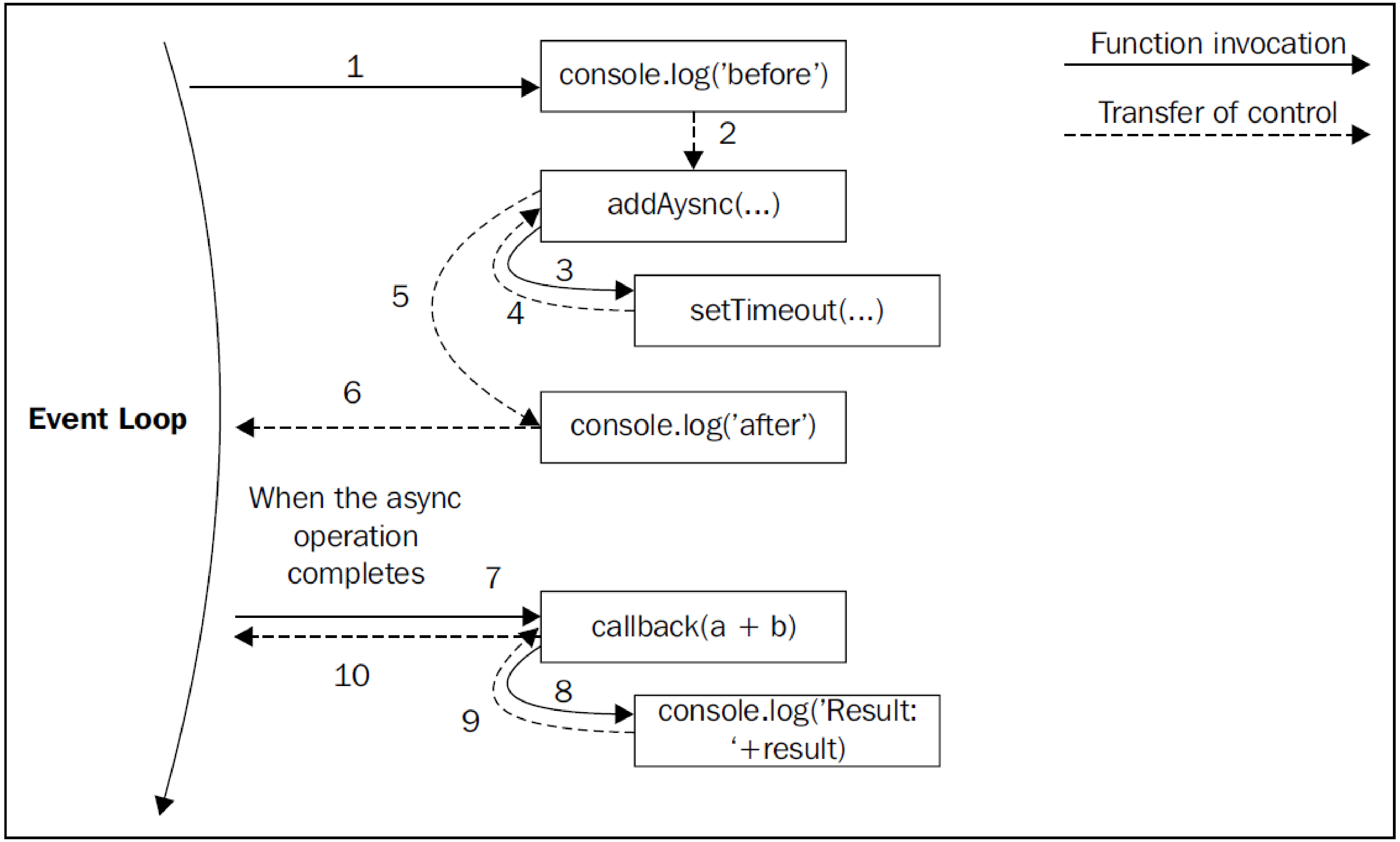
ASYNCHRONOUS CONTINUATION PASSING STYLE

```
console.log('before');
add(1, 2, function(result) {
console.log('Result: ' + result);
});
console.log('after');
```

Since add() is synchronous, the outcome will be sequential here

```
function addAsync(a, b, callback) {
setTimeout(function() {
callback(a + b);
}, 100);
}
```

- Since `setTimeout()` triggers an asynchronous operation, it will not wait anymore for the callback to be executed, but instead, it returns immediately giving the control back to `addAsync()`, and then back to its caller.
- This property in Node.js is crucial, as it allows the stack to unwind, and the control to be given back to the event loop as soon as an asynchronous request is sent, thus allowing a new event from the queue to be processed.



- ❑ When the asynchronous operation completes, the execution is then resumed starting from the callback provided to the asynchronous function that caused the unwinding.
- ❑ The execution will start from the Event Loop, so it will have a fresh stack.

CPS

```
if(cache[filename]) {
process.nextTick(function() {
callback(cache[filename]);
});
}
```

- Pattern: prefer the direct style for purely synchronous functions.
- A synchronous API will block the event loop and put the concurrent requests on hold.
- Use blocking API only when they don't affect the ability of the application to serve concurrent requests.
- Pattern: we guarantee that a callback is invoked asynchronously by deferring its execution using `process.nextTick()`.
- The trick here is to schedule the synchronous callback invocation to be executed "in the future" instead of being run immediately in the same event loop cycle
- Two ways to defer the execution
- Using `process.nextTick()`, which defers the execution of a function until the next pass of the event loop.
- It is run before any other I/O event is fired
- With `setImmediate()`, the execution is queued behind any I/O event that is already in the queue