PES UNIVERSITY ONLINE

NODE JS

Aruna S
Department of
Computer Science and Engineering

NODE JS

Buffers and Streams

S. Aruna

Department of Computer Science and Engineering

Buffers Introduction

- Pure JavaScript is Unicode friendly, but it is not so for binary data.
- Working with TCP streams or the file system, it's necessary to handle octet streams.
- Node provides Buffer class which provides instances to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.
- Buffer class is a global class that can be accessed in an application without importing the buffer module.

Buffer Operations

- Creating Buffers
- Writing to Buffers
- Reading from Buffers
- Concatenate Buffers
- Copy Buffers
- Compare Buffers

Creating Buffers

- The **Buffer.alloc() method** is used to create a new buffer object of the specified size.
- This method is slower than **Buffer.allocUnsafe() method** but it assures that the newly created Buffer instances will never contain old information or data that is potentially sensitive.

Syntax

```
Buffer.alloc(size, fill, encoding)
```

size: It specifies the size of the buffer.
fill: It is an optional parameter and specifies the value to fill the buffer. Its default value is 0.
encoding: It is an optional parameter that specifies the value if the buffer value is a string. Its default value is **'utf8'**.
Return Value: This method returns a new initialized Buffer of the specified size.
A TypeError will be thrown if the given size is not a number.

Writing to Buffers

Syntax

buf.write(string[, offset][, length][, encoding]) Parameters

string − This is the string data to be written to buffer.
offset − This is the index of the buffer to start writing at. Default value is 0.
length − This is the number of bytes to write. Defaults to buffer.length.
encoding − Encoding to use. 'utf8' is the default encoding.

Return Value

This method returns the number of octets written. If there is not enough space in the buffer to fit the entire string, it will write a part of the string.

NODE JS

Reading from Buffers

Syntax

buf.toString([encoding][, start][, end]) Parameters

encoding − Encoding to use. 'utf8' is the default encoding.
start − Beginning index to start reading, defaults to 0.
end − End index to end reading, defaults is complete buffer.

Return Value

This method decodes and returns a string from buffer data encoded using the specified character set encoding.

Compare Buffers

buf.compare(target[, targetStart[, targetEnd[, sourceStart[, sourceEnd]]]])

- target <Buffer> | <Uint8Array> A Buffer or Uint8 Array with which to compare buf.
- targetStart <integer> The offset within target at which to begin comparison. Default: 0.
- targetEnd <integer> The offset within target at which to end comparison (not inclusive). Default: target.length.
- sourceStart <integer> The offset within buf at which to begin comparison. Default: 0.
- sourceEnd <integer> The offset within buf at which to end comparison (not inclusive). Default: buf.length.
- Returns: <integer>

Compares buf with target and returns a number indicating whether buf comes before, after, or is the same as target in sort order.

0 is returned if target is the same as buf

1 is returned if target should come before buf when sorted.

-1 is returned if target should come after buf when sorted.

Copy Buffers

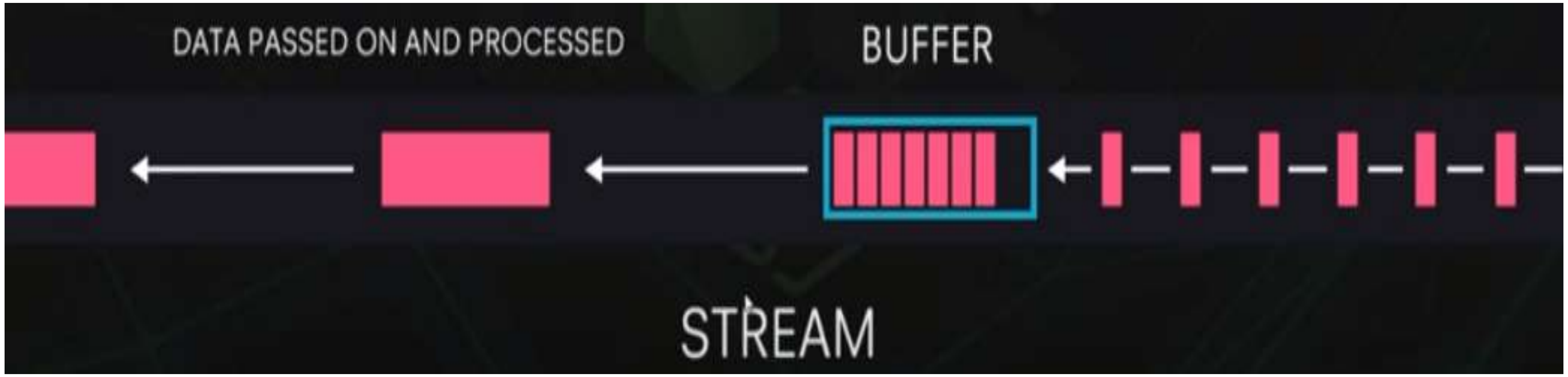
buf.copy(target[, targetStart[, sourceStart[, sourceEnd]]])#

- target <Buffer> | <Uint8Array> A Buffer or Uint8Array to copy into.
- targetStart <integer> The offset within target at which to begin writing. Default: 0.
- sourceStart <integer> The offset within buf from which to begin copying. Default: 0.
- sourceEnd <integer> The offset within buf at which to stop copying (not inclusive). Default: buf.length.
- Returns: <integer> The number of bytes copied.

Copies data from a region of buf to a region in target, even if the target memory region overlaps with buf.

NODE JS

Buffers and Streams in Action – YouTube Example



Streams

- Streams are one of the fundamental concepts that power Node.js applications.
- They are data-handling method and are used to read or write input into output sequentially.
- Streams are a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way.
- A program reads a file into memory **all at once** like in the traditional way, whereas streams read chunks of data piece by piece, processing its content without keeping it all in memory.
- This makes streams really powerful when working with **large amounts of data**, for example, a file size can be larger than your free memory space, making it impossible to read the whole file into the memory in order to process it.
- Streams also give us the power of ‘composability’ in our code

For Example “streaming” services such as **YouTube or Netflix**

Why Streams

Streams basically provide two major advantages compared to other data handling methods:

Memory efficiency: you don’t need to load large amounts of data in memory before you are able to process it

Time efficiency: it takes significantly less time to start processing data as soon as you have it, rather than having to wait with processing until the entire payload has been transmitted

Types of streams in Node Js

There are 4 types of streams in Node.js:

Writable: streams to which we can write data. For example, fs.createWriteStream() lets us write data to a file using streams.

Readable: streams from which data can be read. For example: fs.createReadStream() lets us read the contents of a file.

Duplex: streams that are both Readable and Writable. For example, net.Socket

Transform: streams that can modify or transform the data as it is written and read. For example, in the instance of file-compression, you can write compressed data and read decompressed data to and from a file.

In HTTP server, **request is a readable stream and response is a writable stream.**

Streams as Events

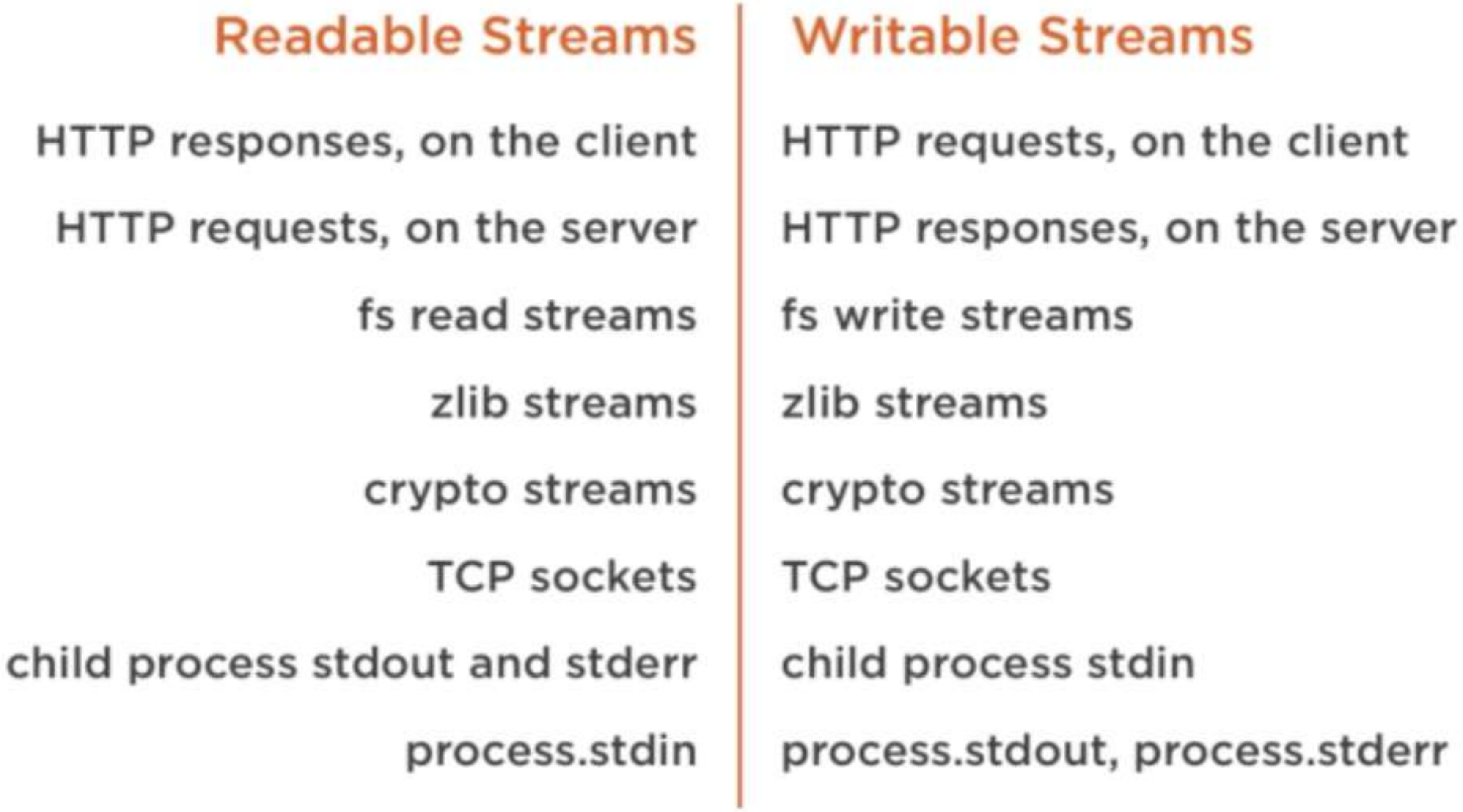
Each type of Stream is an **EventEmitter** instance and throws several events at different instance of times.

For example, some of the commonly used events are

- **data** − This event is fired when there is data is available to read.
- **end** − This event is fired when there is no more data to read.
- **error** − This event is fired when there is any error receiving or writing data.
- **finish** − This event is fired when all the data has been flushed to underlying system.

NODE JS

Native Nodejs Objects - streams



Readable Streams	Writable Streams
HTTP responses, on the client	HTTP requests, on the client
HTTP requests, on the server	HTTP responses, on the server
fs read streams	fs write streams
zlib streams	zlib streams
crypto streams	crypto streams
TCP sockets	TCP sockets
child process stdout and stderr	child process stdin
process.stdin	process.stdout, process.stderr

THANK YOU

Aruna S
Department of
Computer Science and Engineering
arunas@pes.edu