NODE JS

Prof. Vinay Joshi and Dr. Sarasvathi V

Department of Computer Science and Engineering

Acknowledgement

The slides are created from various internet resources with valuable contributions from multiple professors.

NODE JS

Buffers and Streams

Department of Computer Science and Engineering

Buffers and Streams in Action – YouTube Example

- Pure JavaScript is great with Unicode encoded strings, but it does not handle binary data very well.
- It is not problematic when we perform an operation on data at browser level but at the time of dealing with TCP stream and performing a read-write operation on the file system is required to deal with pure binary data.
- To satisfy this need Node.js use Buffer, So, we are going to know about buffer in Node.js.

Buffers and Streams in Action – YouTube Example

- **Buffers in Node.js:** The Buffer class in Node.js is used to perform operations on raw binary data.
- Generally, Buffer refers to the particular memory location in memory. Buffer and array have some similarities, but the difference is array can be any type, and it can be resizable.
- Buffers only deal with binary data, and it can not be resizable.

- Each integer in a buffer represents a byte. console.log() function is used to print the Buffer instance.

NODE JS

Buffer Operations

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

Buffers and Streams in Action – YouTube Example

Methods to perform the operations on Buffer:

1 Buffer.alloc(size): It creates a buffer and allocates size to it.
2 Buffer.from(initialization) :It initializes the buffer with given data.
3 Buffer.write(data): It writes the data on the buffer.
4 toString() :It read data from the buffer and returned it.
5 Buffer.isBuffer(object): It checks whether the object is a buffer or not.
6 Buffer.length It returns the length of the buffer.

Buffers and Streams in Action – YouTube Example

How to create a buffer

- A buffer is created using the Buffer.from(), Buffer.alloc(), and Buffer.allocUnsafe() methods.
- While both alloc and allocUnsafe allocate a Buffer of the specified size in bytes, the Buffer created by alloc will be initialized with zeroes and the one created by allocUnsafe will be uninitialized.
- This means that while allocUnsafe would be quite fast in comparison to alloc, the allocated segment of memory may contain old data which could potentially be sensitive.

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.

Buffers and Streams in Action – YouTube Example

```
// Different Method to create Buffer
var buffer1 = Buffer.alloc(100);
var buffer2 = new Buffer('GFG');
var buffer3 = Buffer.from([1, 2, 3, 4]);
```

NODE JS

Buffers and Streams in Action – YouTube Example

Using a buffer
Access the content of a buffer

A buffer, being an array of bytes, can be accessed like an array:

JS

```
const buf = Buffer.from('Hey!')
console.log(buf[0]) //72
console.log(buf[1]) //101
console.log(buf[2]) //121
```

Buffers and Streams in Action – YouTube Example

Creating Buffers: Followings are the different ways to create buffers in Node.js:

- **Create an uninitiated buffer:** It creates the uninitiated buffer of given size.
  Syntax:

```
var ubuf = Buffer.alloc(5);
```

- The above syntax is used to create an uninitiated buffer of 5 octets.
- **Create a buffer from array:** It creates the buffer from given array.
  Syntax:

```
var abuf = new Buffer([16, 32, 48, 64]);
```

- The above syntax is used to create a buffer from given array.
- **Create a buffer from string:** It creates buffer from given string with optional encoding.
  Syntax:

```
var sbuf = new Buffer("Welcome", "ascii");
```

- The above syntax is used to create a Buffer from a string and encoding type can also be specified optionally.

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.

Buffers and Streams in Action – YouTube Example

```
// Writing data to Buffer
buffer1.write("Happy Learning");
```

Writing to Buffers in Node.js: The **buf.write() method** is used to write data into a node buffer.

Syntax:

```
buf.write( string, offset, length, encoding )
```

```
cbuf = new Buffer(256);
bufferlen = cbuf.write("Learn Programming!!!");
console.log("No. of Octets in which string is written : "+  bufferlen);
```

Buffers and Streams in Action – YouTube Example

Program to create a buffer,read data from buffer and write data into buffer

```
var buffer1 = Buffer.alloc(100);
var buffer2 = new Buffer('GFG');
var buffer3 = Buffer.from([1, 2, 3, 4]);

// Writing data to Buffer
buffer1.write("Happy Learning");

// Reading data from Buffer
var a = buffer1.toString('utf-8');
console.log(a);

// Check object is buffer or not
console.log(Buffer.isBuffer(buffer1));

// Check length of Buffer
console.log(buffer1.length);
```

Buffers and Streams in Action – YouTube Example

```
var bufferSrc = new Buffer('ABC');
var bufferDest = Buffer.alloc(3);
bufferSrc.copy(bufferDest);

var Data = bufferDest.toString('utf-8');
console.log(Data);

// Slicing data
var bufferOld = new Buffer('GeeksForGeeks');
var bufferNew = bufferOld.slice(0, 4);
console.log(bufferNew.toString());

// concatenate two buffer
var bufferOne = new Buffer('Happy Learning ');
var bufferTwo = new Buffer('With GFG');
var bufferThree = Buffer.concat([bufferOne, bufferTwo]);
console.log(bufferThree.toString());
```

Buffers and Streams in Action – YouTube Example

```
buf = new Buffer(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97;
}

console.log( buf.toString('ascii'));       // outputs: abcdefghijklmnopqrstuvwxyz
console.log( buf.toString('ascii',0,5));   // outputs: abcde
console.log( buf.toString('utf8',0,5));    // outputs: abcde
console.log( buf.toString(undefined,0,5)); // encoding defaults to 'utf8',
outputs abcde
```

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.

Buffers and Streams in Action – YouTube Example

You can print the full content of the buffer using the toString() method:

```
console.log(buf.toString())
```

Get the length of a buffer

Use the length property:

```
const buf = Buffer.from('Hey!')
console.log(buf.length)
```

Buffers and Streams in Action – YouTube Example

Iterate over the contents of a buffer

```
const buf = Buffer.from('Hey!')
for (const item of buf) {
  console.log(item) //72 101 121 33
}
```

Just like you can access a buffer with an array syntax, you can also set the contents of the buffer in the same way:

```
const buf = Buffer.from('Hey!')
buf[1] = 111 //o in UTF-8
console.log(buf.toString()) //Hoy!
```

NODE JS

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.

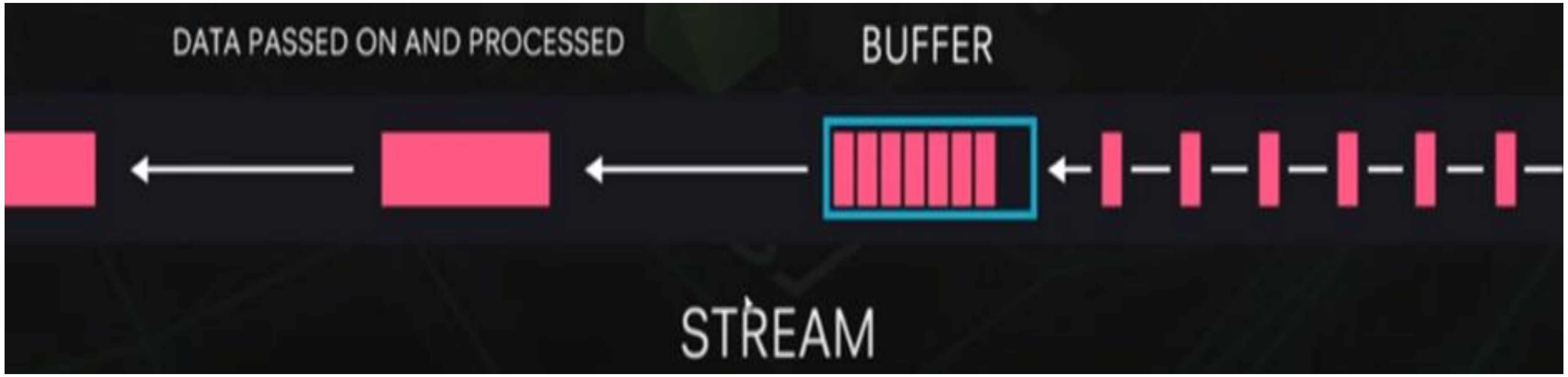
Copy Buffers

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.

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

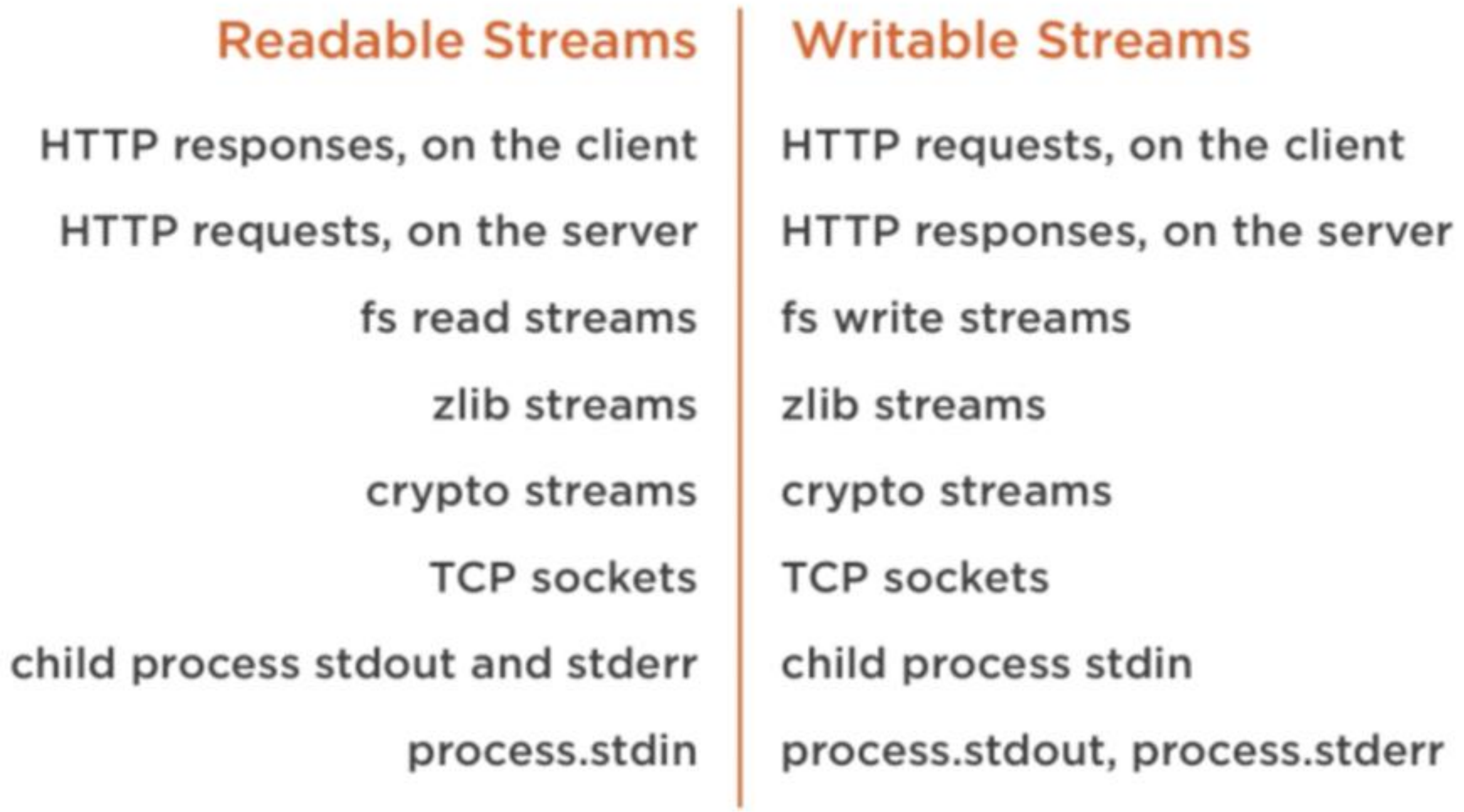
Each type of Stream is an **EventEmitter** instance and throws several events at different instance of times.

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

Buffers and Streams in Action – YouTube Example

```
const http = require('http')
const fs = require('fs')

const server = http.createServer(function(req, res) {
  fs.readFile(__dirname + '/data.txt', (err, data) => {
    res.end(data)
  })
})
server.listen(3000)
```

readFile() reads the full contents of the file, and invokes the callback function when it's done.

res.end(data) in the callback will return the file contents to the HTTP client.

Buffers and Streams in Action – YouTube Example

if the file is big, the operation will take quite a bit of time. Here is the same thing written using streams:

JS

```
const http = require('http')
const fs = require('fs')

const server = http.createServer((req, res) => {
  const stream = fs.createReadStream(__dirname + '/data.txt')
  stream.pipe(res)
})
server.listen(3000)
```

Instead of waiting until the file is fully read, we start streaming it to the HTTP client as soon as we have a chunk of data ready to be sent.

Buffers and Streams in Action – YouTube Example

pipe()

The above example uses the line stream.pipe(res): the pipe() method is called on the file stream.

What does this code do? It takes the source, and pipes it into a destination.

You call it on the source stream, so in this case, the file stream is piped to the HTTP response.

The return value of the pipe() method is the destination stream, which is a very convenient thing that lets us chain multiple pipe()

THANK YOU

Vinay Joshi and Dr.Sarasvathi V
Department of Computer Science and Engineering