NODEJS.EN 2.6 → FS, STREAMS AND MORE

- Purposes of "FS" module
- Working with binary data
- Why there is Buffer class in Node.js
- Idea of streaming data
- Types of stream instances
- Ability to start new processes from Node.js app

CLASS PROPERTIES

fs.__defineGetter__	fs.__defineSetter__	fs.__lookupGetter__	fs.__lookupSetter__	fs.__proto__	fs.constructor
fs.hasOwnProperty	fs.isPrototypeOf	fs.propertyIsEnumerable	fs.toLocaleString	fs.toString	fs.valueOf

FILE HANDLING -RELATED

fs.F_OK	fs.FileReadStream	fs.FileWriteStream	fs.R_OK	fs.ReadStream	fs.Stats
fs.SyncWriteStream	fs.W_OK	fs.WriteStream	fs.X_OK	fs._toUnixTimestamp	fs.access
fs.accessSync	fs.appendFile	fs.appendFileSync	fs.chmod	fs.chmodSync	fs.chown
fs.chownSync	fs.close	fs.closeSync	fs.constants	fs.copyFile	fs.copyFileSync
fs.createReadStream	fs.createWriteStream	fs.exists	fs.existsSync	fs.fchmod	fs.fchmodSync
fs.fchown	fs.fchownSync	fs.fdatasync	fs.fdatasyncSync	fs.fstat	fs.fstatSync
fs.fsync	fs.fsyncSync	fs.ftruncate	fs.ftruncateSync	fs.futimes	fs.futimesSync
fs.lchmod	fs.lchmodSync	fs.lchown	fs.lchownSync	fs.link	fs.linkSync
fs.lstat	fs.lstatSync	fs.mkdir	fs.mkdirSync	fs.mkdtemp	fs.mkdtempSync
fs.open	fs.openSync	fs.read	fs.readFile	fs.readFileSync	fs.readSync
fs.readdir	fs.readdirSync	fs.readlink	fs.readlinkSync	fs.realpath	fs.realpathSync
fs.rename	fs.renameSync	fs.rmdir	fs.rmdirSync	fs.stat	fs.statSync
fs.symlink	fs.symlinkSync	fs.truncate	fs.truncateSync	fs.unlink	fs.unlinkSync
fs.unwatchFile	fs.utimes	fs.utimesSync	fs.watch	fs.watchFile	fs.write
fs.writeFile	fs.writeFileSync	fs.writeSync			

METHOD GROUPS

- File content
- Placement
- Directories
- Properties and permissions
- File descriptor lifecycle
- Event handling
- Streams

CLASSES

- **Stats**

`.isBlockDevice(), .isCharacterDevice(), .isDirectory(), .isFIFO(), .isFile(), .isSocket(), .isSymbolicLink(), .dev, .ino, .mode, .nlink, .uid, .gid, .rdev, .size, .blksize, .blocks, .atimeMs, .mtimeMs, .ctimeMs, .birthtimeMs, .atime, .mtime, .ctime, .birthtime`

- **FSWatcher**

`'change', 'close', 'error', .close()`

- **Readstream**

`'close', 'open', 'ready', .bytesRead, .path`

- **WritestreamPromises API**

`‘close', 'open', 'ready', .bytesWritten, .path`

- **Promises API**

Stability: 1 - Experimental. This feature is still under active development and subject to non-backwards compatible changes, or even removal, in any future version. Use of the feature is not recommended in production environments. Experimental features are not subject to the Node.js Semantic Versioning model.

FILE CONTENT

fs.readFile()
fs.writeFile()
fs.appendFile()
fs.read()
fs.write()
fs.ftruncate/truncate()
fs.fdatasync/fsync()
fs.readlink()

PLACEMENT

fs.copyFile()
fs.realpath()
fs.rename()
fs.unlink()
fs.link()
fs.symlink()

DIRECTORIES

fs.write()
fs.ftruncate/truncate()
fs.fdatasync/fsync()
fs.readlink()

PROPERTIES AND PERMISSIONS

fs.access()
fs.exists()
fs.fstat/stat/lstat()
fs.futimes/utimes()
fs.chmod/fchmod/lchmod()
fs.chown/fchown/lchown()

FILE DESCRIPTOR

fs.open()
fs.close()

HANDLING

fs.watch()
fs.watchFile()
fs.unwatchFile()

STREAMS

fs.createReadStream()
fs.createWriteStream()

On POSIX systems, for every process, the kernel maintains a table of currently open files and resources. Each open file is assigned a simple numeric identifier called a file descriptor.

The fs.open() method is used to allocate a new file descriptor. Once allocated, the file descriptor may be used to read data from, write data to, or request information about the file.

Most operating systems limit the number of file descriptors that may be open at any given time so it is critical to close the descriptor when operations are completed. **Failure to do so will result in a memory leak that will eventually cause an application to crash.**

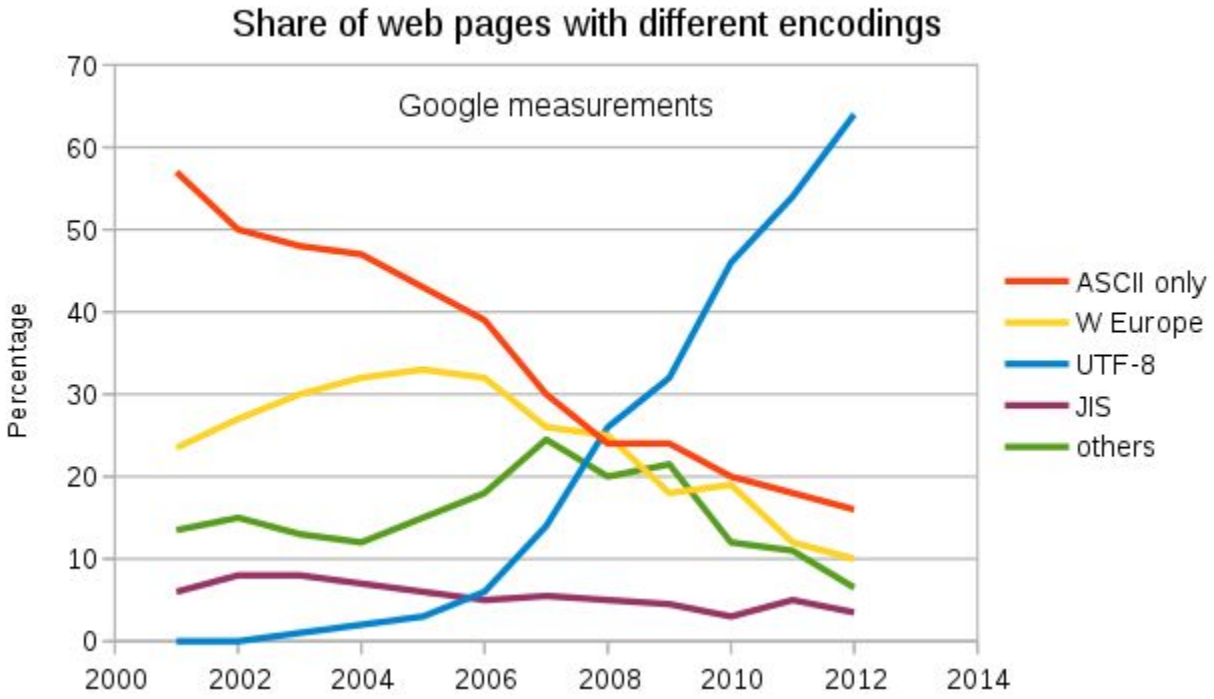
```
const fs = require('fs')

fs.readFile(__dirname + '/a.js', (err, data) => {
   if (err) throw err;
   console.log(data);
   // console.log(data.constructor.name);
   // console.log(data.toString());
 });
```

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

fs.readFile(__dirname + '/a.js', (err, data) => {
   if (err) throw err;
   console.log(data);
   // console.log(data.constructor.name);
   // console.log(data.toString());
 });
```

```
<Buffer 63 6f 6e 73 74 20 43 68 61 6e 63 65 20 3d 20 72 65 71 75 69 72 65 28 27 63 68 61 6e 63 65 27 29 3b 0... >
```

The Buffer class was introduced as part of the Node.js API to enable interaction with octet streams in TCP streams, file system operations, and other contexts.

```
Buffer.from('hello\n') →                  <Buffer 68 65 6c 6c 6f 0a>
Buffer.from('hello\n').toString('hex') →  '68656c6c6f0a'
Buffer.from('hello\n').toString('utf8') → 'hello\n'
```


Share of web pages with different encodings
Google measurements
Percentage
70
60
50
40
30
20
10
0
2000
2002
2004
2006
2008
2010
2012
2014
ASCII only
W Europe
UTF-8
JIS
others

With TypedArray now available, the Buffer class implements the Uint8Array API in a manner that is more optimized and suitable for Node.js.

- `new Buffer()` **is deprecated, use** `Buffer.alloc()`, `Buffer.concat()`, `Buffer.from()`
- Methods: `buf.toString([encoding[, start[, end]]])`
- Properties: `[index]`, `buf.length`

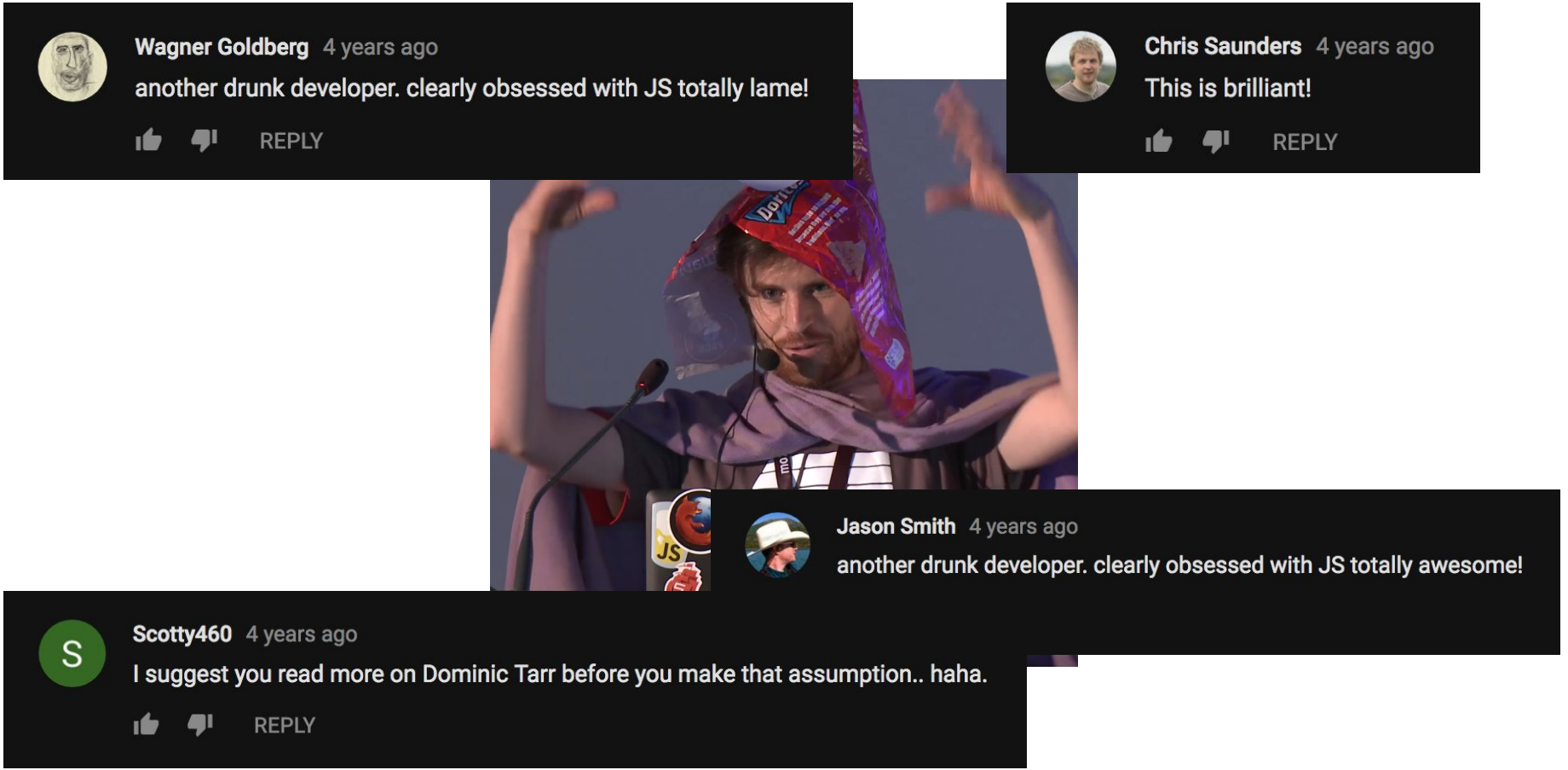
Instances of the Buffer class are similar to arrays of integers but correspond to fixed-sized, raw memory allocations outside the V8 heap. **The size of the Buffer is established when it is created and cannot be changed.**

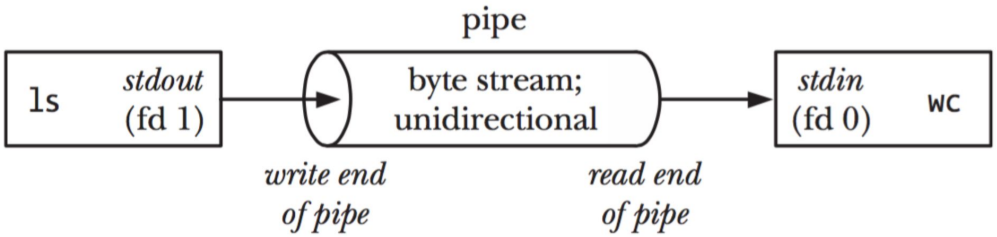
Transferring binary data instead of strings ensures **safe transportation, API universality, and speed.**

A stream is an abstract interface implemented by various objects in Node. For example a request to an HTTP server is a stream, as is stdout. Streams are readable, writable, or both. All streams are instances of EventEmitter.

“Streams are Node’s best and most misunderstood idea”

– Dominic Tarr


Wagner Goldberg 4 years ago
another drunk developer. clearly obsessed with JS totally lame!
REPLY
Chris Saunders 4 years ago
This is brilliant!
REPLY
Jason Smith 4 years ago
another drunk developer. clearly obsessed with JS totally awesome!
Scotty460 4 years ago
I suggest you read more on Dominic Tarr before you make that assumption.. haha.
REPLY



In Unix-like computer operating systems, a pipeline is a sequence of processes chained together by their standard streams, so that the output of each process (stdout) feeds directly as input (stdin) to the next one.

https://www.geeksforgeeks.org/piping-in-unix-or-linux/

```
$ ls -l | grep json > list.txt
$ process1 | process2 | process3
```

Node.js is built with the UNIX philosophy in mind. Should you be unfamiliar, one of the most important takeaways is this:

Do One Thing and Do It Well

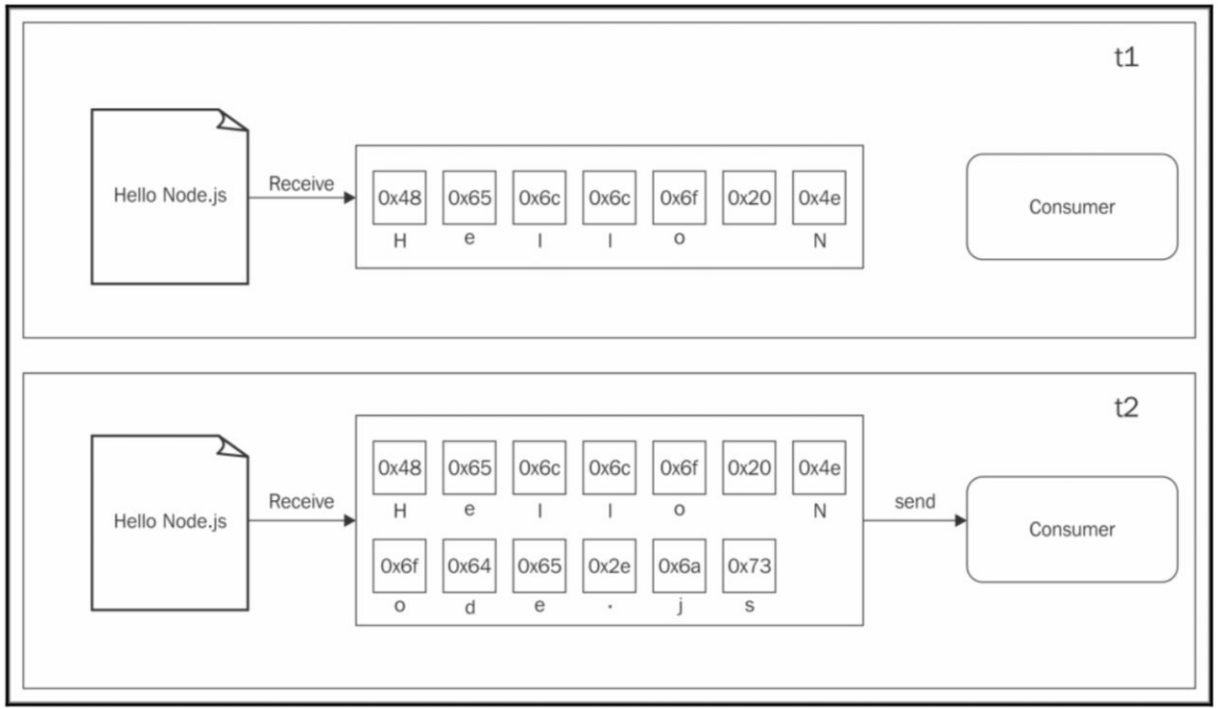
In following this principle, lightweight binaries and modules will be created to absolutely succeed in executing one simple task. With the connective properties of pipes (and analogically, streams) these several modules will be able to link up and create a complex system to execute complicated tasks.

NODEJS.EN 2.6 → “HELLO WORLD”

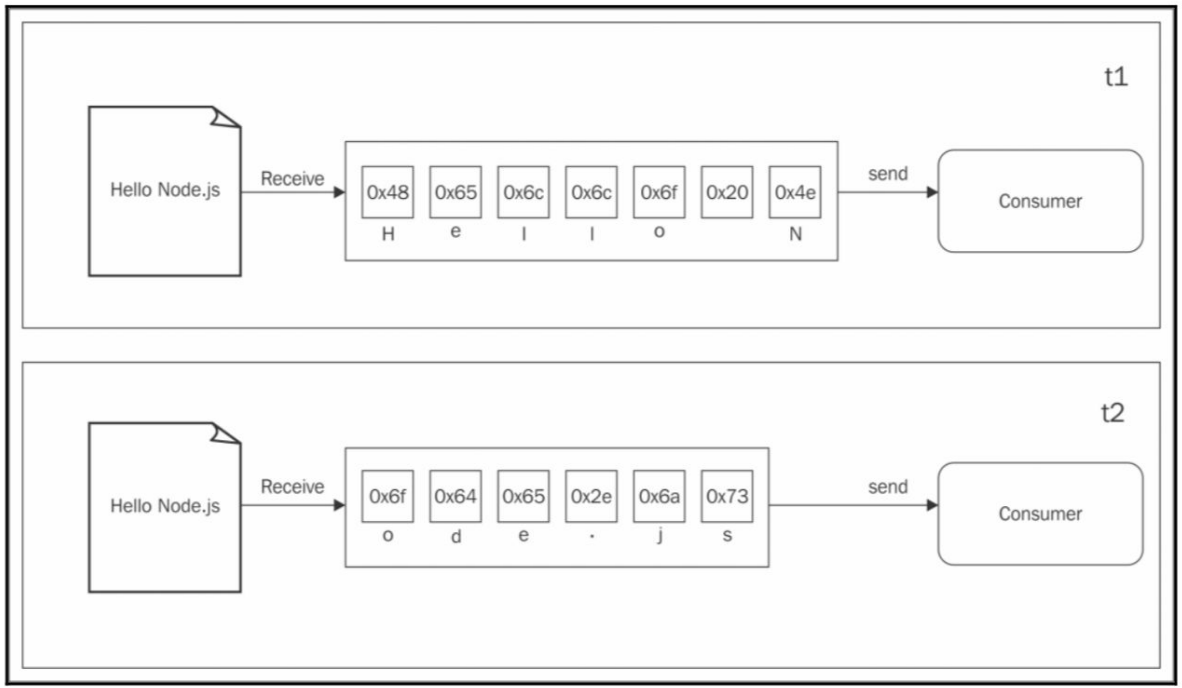
```
#!/usr/bin/env node
process.stdin.setEncoding('utf8');
process.stdin.on('readable', () => {
    let chunk = process.stdin.read();
    if (chunk !== null) {
        process.stdout.write(`${chunk}`);
    }
});
```

NODEJS.EN 2.6 → BETTER “HELLO WORLD”

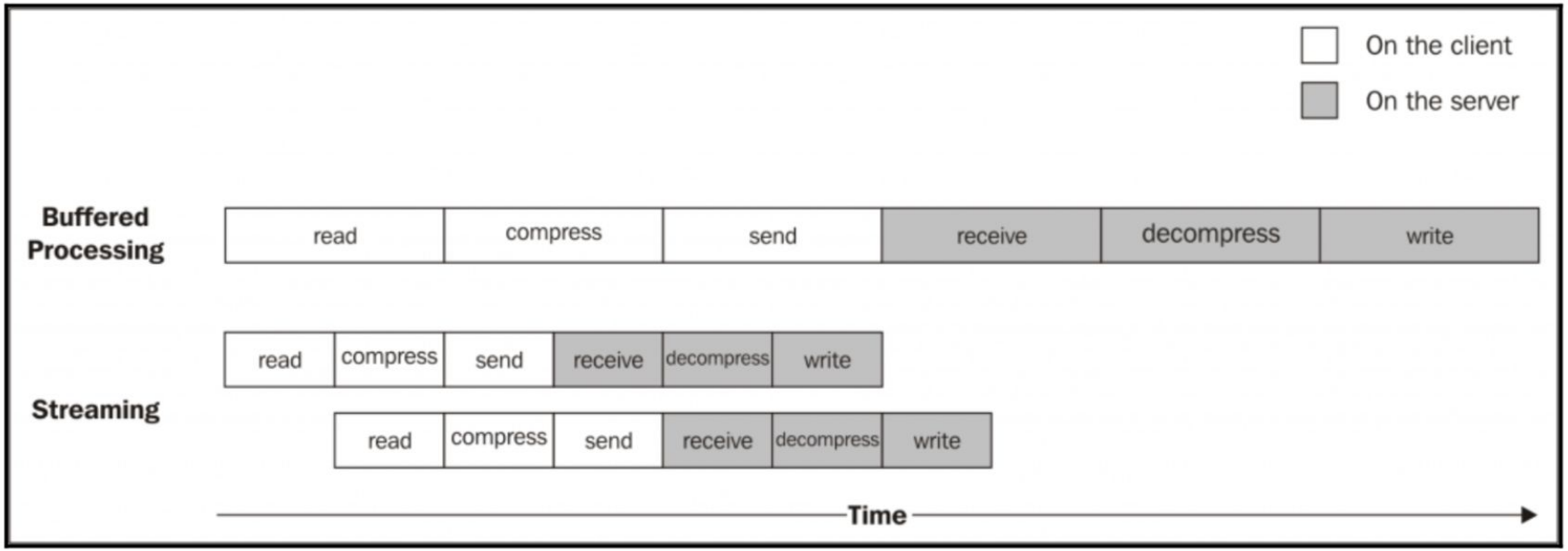

```
#!/usr/bin/env node
process.stdin.pipe(process.stdout);
```



The user experience is poor too because users will need to wait for the whole file to be buffered into memory on your server before they can start receiving any contents.



Chunk of data is received from the resource and is immediately provided to the consumer, who now has the chance to process it straight away without waiting for all the data to be collected in the buffer.


On the client
On the server
Buffered Processing
read
compress
send
receive
decompress
write
Streaming
read
compress
send
receive
decompress
write
read
compress
send
receive
decompress
write
Time

```
const fs = require('fs')
const zlib = require('zlib')
const fileName = process.argv[2]

fs.readFile(fileName, (err, buffer) => {
      zlib.gzip(buffer, (err, buffer) => {
            fs.writeFile(fileName + '.gz', buffer, (err) => {
                  console.log('File successfully compressed')
            })
      })
})
```

```
RangeError: File size is greater than possible Buffer: 0x3FFFFFFF bytes
```

```
const fs = require('fs')
const zlib = require('zlib')
const file = process.argv[2]

fs.createReadStream(file)
.pipe(zlib.createGzip())
    .pipe(fs.createWriteStream(file + '.gz'))
    .on('finish', () => console.log('File successfully compressed'))
```

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

const server = http.createServer((req, res) => {
   fs.readFile(__dirname + '/data.txt', (err, data) => {
       res.end(data)
   })
})
server.listen(8000)
```

```
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(8000)
```

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

const server = http.createServer((req, res) => {
   const stream = fs.createReadStream(__dirname + '/data.txt')
   stream.pipe(oppressor(req)).pipe(res)
})
server.listen(8000)
```

Streams allow us to do things that would not be possible, by buffering data and processing it all at once.

Consider the case in which we have to read a very big file, let's say, in the order of hundreds of megabytes or even gigabytes.

Reading a few of these big files concurrently; our application will easily run out of memory.

Buffers in V8 cannot be bigger than 0x3FFFFFFF bytes (~1GB). So, we might hit a wall way before running out of physical memory.

Spatial efficiency

Time efficiency

Composability

In information systems, the essential features that make a component composable are that it be:

self-contained (modular): it can be deployed independently

Stateless: it treats each request as an independent transaction, unrelated to any previous request

It is widely believed that composable systems are more trustworthy than non-composable systems because it is easier to evaluate their individual parts.

Every stream in Node.js is an implementation of one of the four base abstract classes available in the stream core module:

- Readable - streams from which data can be read (ex.: fs.createReadStream, process.stdin)
- Writable - streams to which data can be written (ex.: fs.createWriteStream, process.stdout)
- Duplex - streams that are both Readable and Writable (ex.: net.Socket)
- Transform - streams that can modify or transform the data (ex.: zlib.createDeflate)

READABLE STREAMS

HTTP responses, on the client

HTTP requests, on the server

Fs read streams

Zlib streams

Crypto streams

TCP sockets

Child process stdout and stderr

Process.stdin

WRITABLE STREAMS

HTTP requests, on the client

HTTP responses, on the server

Fs write streams

Zlib streams

Crypto streams

TCP sockets

Child process stdin

Process.stdout, process.stderr

...and many more

All the different types of streams use .pipe() to pair inputs with outputs.

.pipe() is just a function that takes a readable source stream src and hooks the output to a destination writable stream dst:

```
src.pipe(dst)
```

.pipe(dst) returns dst so that you can chain together multiple .pipe() calls together:

```
a.pipe(b).pipe(c).pipe(d)
```

which is the same as:

```
a.pipe(b);
b.pipe(c);
c.pipe(d);
```

This is very much like what you might do on the command-line to pipe programs together:

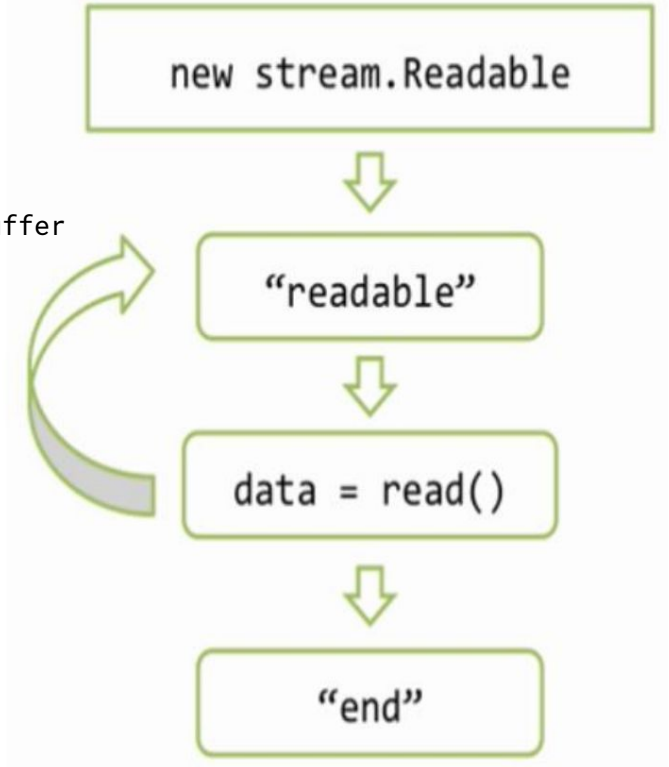
```
a | b | c | d
```

Events:

- ‘readable’ – when stream ready for reading from his internal buffer
- ‘error’ – emits when an error occurs
- ‘end’ – when achieved end of source data

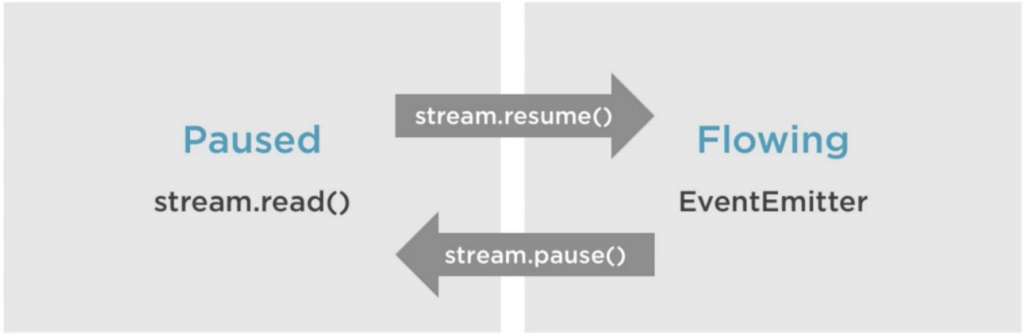
Methods:

- read() – read chunk of data from internal buffer of stream
- read(N) – read chunk of data with size N bytes



```
const stream = require('stream')
class RandomStream extends stream.Readable {
   constructor(options) {
       super(options)
       this.index = 0
   }
   _read() {
       let count = 10, limit = 100
       while (count--) {
           let chunk = `${this.index++} `
           this.push(chunk, 'utf8')
           if (this.index === limit) {
               this.push(null)
               break
           }
       }
   }
}
module.exports = RandomStream
```

```
const RandomStream = require('./randomStream')
const randomStream = new RandomStream()

randomStream.on('readable', () => {
   let chunk;
   while((chunk = randomStream.read(50)) !== null) {
       console.log(`Chunk received: ${chunk}`)
   }
})
```



When a readable stream is in the paused mode, we can use the read() method to read from the stream on demand.

For a readable stream in the flowing mode, the data is continuously flowing and we have to listen to events to consume it.

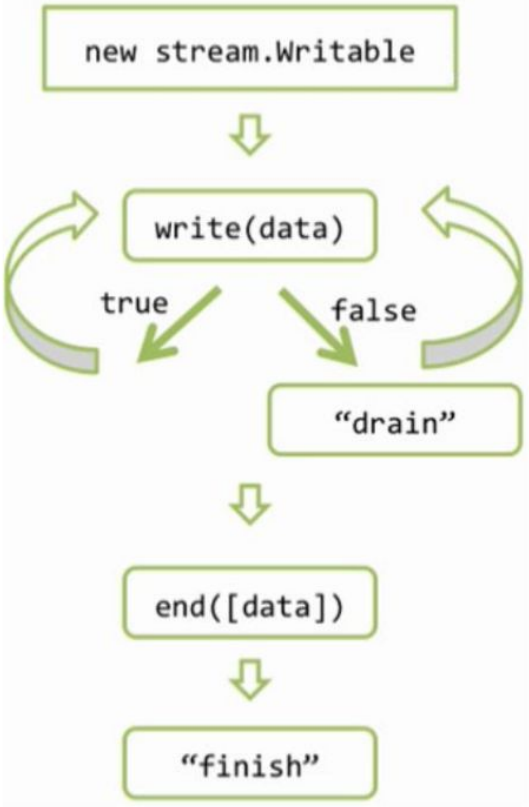
In the flowing mode, data can actually be lost if no consumers are available to handle it. This is why, when we have a readable stream in flowing mode, we need a **data** event handler.

Events:

- ‘drain’ – when internal buffer is ready to get new data by write()
- ‘error’ – emits when an error occurs
- ‘finish’ – when end() called + after all data from internal buffer were written
- ‘pipe’/’unpipe‘ – when stream is piped/unpiped to readable stream

Methods:

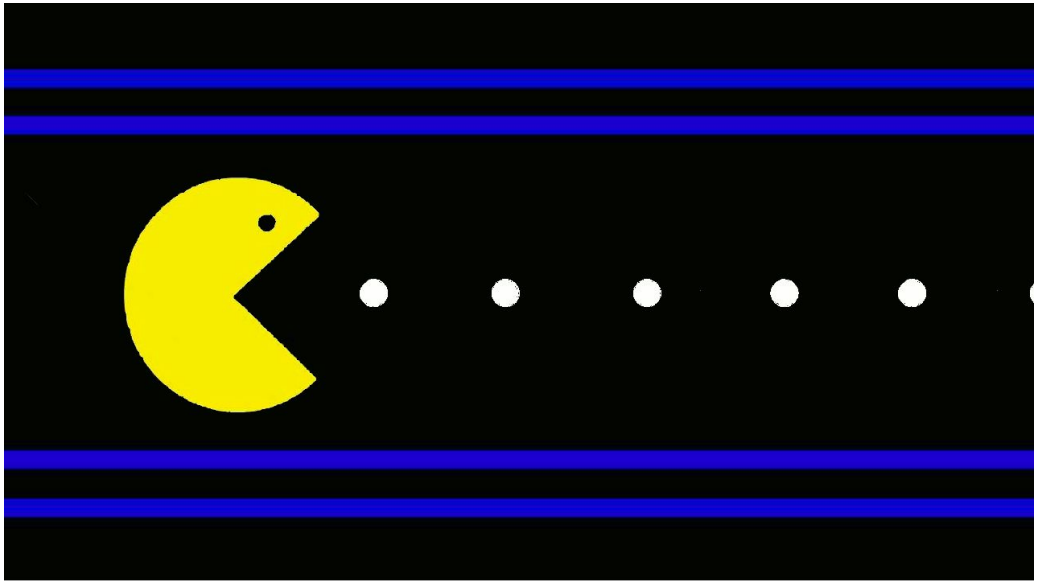
- write() – write data to internal buffer of stream, return flag
- end() – finish working with stream, can take last chunk of data to write



NODEJS.EN 2.6 → WRITABLE STREAM


```
require('http').createServer((req, res) => {
   res.writeHead(200, {'Content-Type': 'text/plain'})

   let count = 0, limit = 10000
   while(count < limit) {
       res.write(`${count++} `)
   }
   res.end('\nThe end...\n')
   res.on('finish', () => console.log('All data was sent'))

}).listen(8080, () => console.log('Listening on http://localhost:8080'))
```

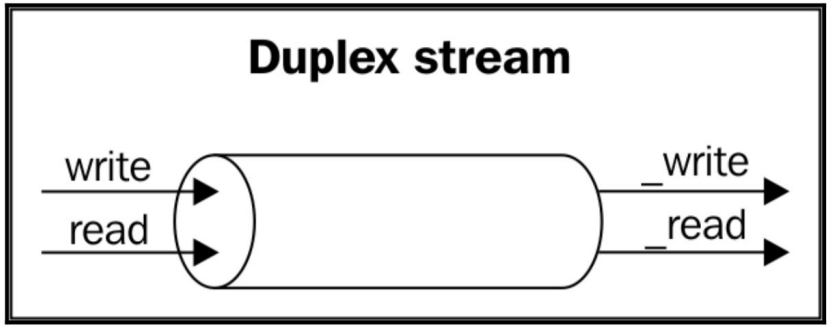
Similar to a liquid flowing in a real piping system, Node.js streams can also suffer from bottlenecks, where data is written faster than the stream can consume it. The mechanism to cope with this problem consists of buffering the incoming data; however, if the stream doesn't give any feedback to the writer, we might incur a situation where more and more data is accumulated into the internal buffer, leading to undesired levels of memory usage.



The term is also used analogously in the field of information technology to describe the build-up of data behind an I/O switch if the buffers are full and incapable of receiving any more data; the transmitting device halts the sending of data packets until the buffers have been emptied and are once more capable of storing information.

```
require('http').createServer((req, res) => {
   res.writeHead(200, { 'Content-Type': 'text/plain' })

   function generateData() {
       let count = 0, limit = 10000
       while(count < limit) {
           let shouldContinue = res.write(`${count++} `)

           if (!shouldContinue) {
               console.log('Backpressure')
               return res.once('drain', generateData)
           }
       }
       res.end('\nThe end...\n', () => console.log('All data was sent'))
   }
   generateData()

}).listen(8080, () => console.log('Listening on http://localhost:8080'))
```

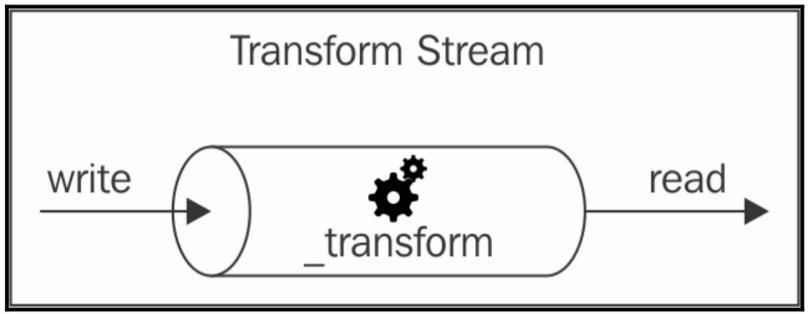
```
const { Writable } = require('stream')
const outStream = new Writable({
   write(chunk, encoding, callback) {
       console.log(chunk.toString())
       callback()
   }
})

process.stdin.pipe(outStream)
```



Duplex streams are readable/writable and both ends of the stream engage in a two-way interaction, sending back and forth messages like a telephone. An rpc exchange is a good example of a duplex stream. Any time you see something like:

```
a.pipe(b).pipe(a)
```

you're probably dealing with a duplex stream.



Transform streams are a certain type of duplex stream (both readable and writable). The distinction is that in Transform streams, the output is in some way calculated from the input.

Through streams are simple readable/writable filters that transform input and produce output.

```
const stream = require('stream')

class ReplaceStream extends stream.Transform {
   constructor(searchString, replaceString) {
       super()
       this.searchString = searchString
       this.replaceString = replaceString
       this.tailPiece = ''
   }
   _transform(chunk, encoding, callback) {
       ...
   }
   _flush(callback) {
       this.push(this.tailPiece)
       callback()
   }
}
module.exports = ReplaceStream
```

```
class ReplaceStream extends stream.Transform {
...
   _transform(chunk, encoding, callback) {
       const pieces = (this.tailPiece + chunk).split(this.searchString)

       const lastPiece = pieces[pieces.length - 1]
       const tailPieceLen = this.searchString.length - 1

       this.tailPiece = lastPiece.slice(-tailPieceLen)
       pieces[pieces.length - 1] = lastPiece.slice(0, -tailPieceLen)

       this.push(pieces.join(this.replaceString))
       callback()
   }
...
}
```

```
const ReplaceStream = require('./replaceStream');
const rs = new ReplaceStream('World', 'Me');
rs.on('data', chunk => console.log(chunk.toString()));
rs.write('Hello W');
rs.write('orld!!!');
rs.end();
```

NODEJS.EN 2.6 → THROUGH2 + MISSISSIPPI

through2

A tiny wrapper around Node streams.Transform (Streams2) to avoid explicit subclassing noise

https://www.npmjs.com/package/through2

mississippi

a collection of useful stream utility modules. learn how the modules work using this and then pick the ones you want and use them individually

the goal of the modules included in mississippi is to make working with streams easy without sacrificing speed, error handling or composability.

https://www.npmjs.com/package/mississippi

Asynchronous control flow with streams

Sequential execution

Unordered parallel execution

Ordered parallel execution

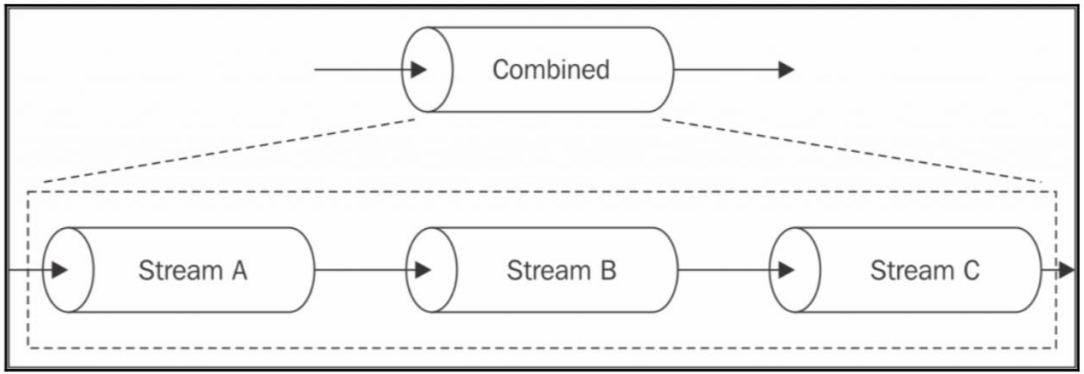
Piping patterns

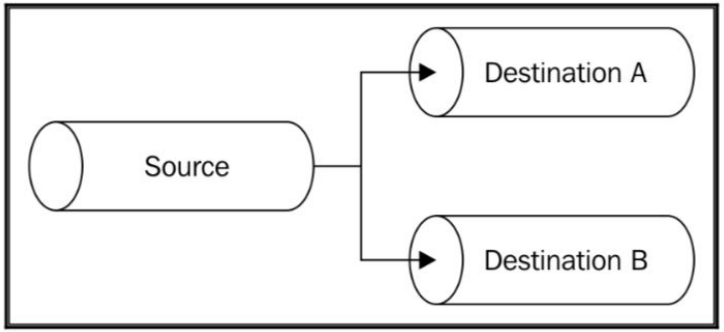
Combining streams

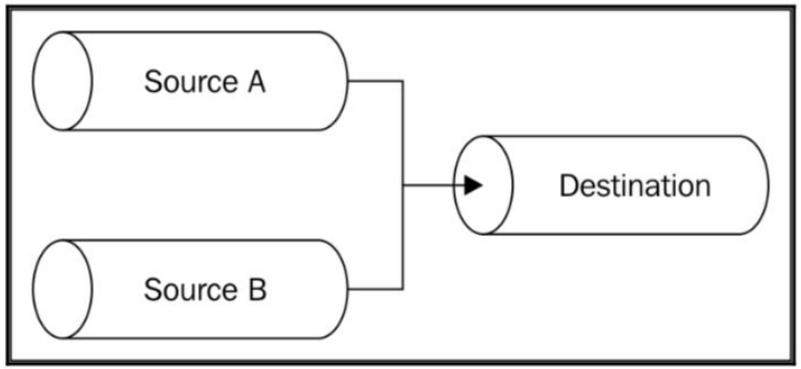
Forking streams

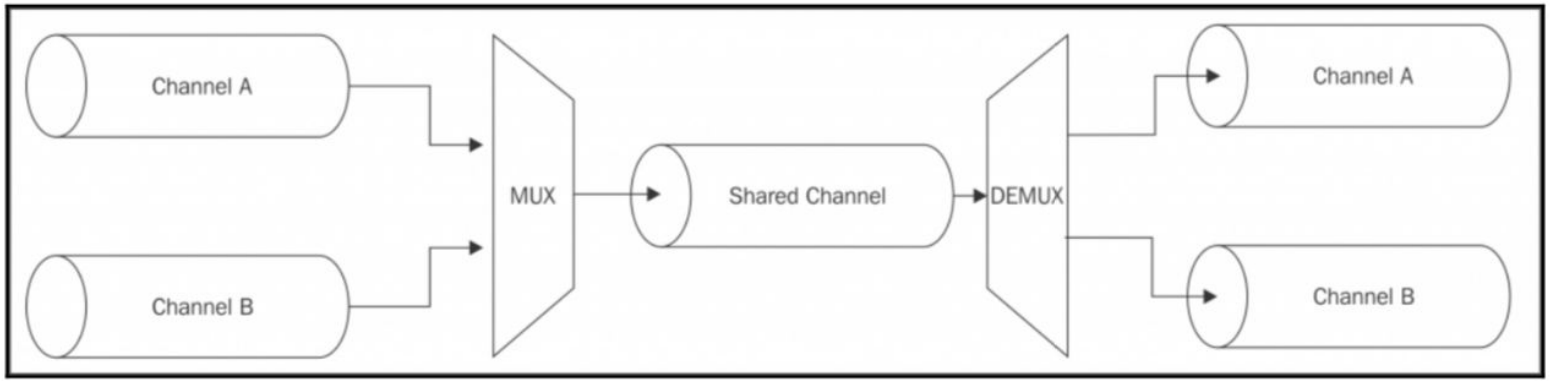
Merging streams

Multiplexing and demultiplexing


Combined
Stream A
Stream B
Stream C


Source
Destination A
Destination B


Source A
Source B
Destination


Channel A
Channel B
MUX
Shared Channel
DEMUX
Channel A
Channel B

«child_process» - module for creating new processes in OS and managing them

Methods (all returns ChildProcess instance):

- exec(command[, options][, callback]) – spawn a subshell and execute the command in that shell
- execFile(file[, args][, options][, callback]) – executes an external application
- fork(modulePath[, args][, options]) – spawn new Node.js instance with running module in it
- spawn(command[, args][, options]) – spawns an external application in a new process and returns a streaming interface for I/O

Class ChildProcess:

- Emits child process events (close, disconnect, error, exit, message)
- Lets send signals to child process (send, disconnect, kill)
- Contains readable and writable streams for transferring data

```
const child_process = require("child_process")

child_process.exec("ls -la", {cwd:"/"}, (err, stdout, stderr) => {
   if (err) {
       console.log(err.toString())
   } else if (stdout !== "") {
       console.log(stdout)
   } else {
       console.log(stderr)
   }
})
```

NodeJS processes runs on a single process, which means it does not take advantage from multi-core systems by default. If you have an 8 core CPU and run a NodeJS program it will run in a single process, wasting the rest of CPUs.

NodeJS offers the cluster module that help us to create programs that uses all the CPUs. Not a surprise the mechanism the cluster module uses to maximize the CPU usage was via forking processes, similar to the old fork() system call Unix systems.

«cluster» - module for horizontal scaling Node.js application

Method:

- fork() – spawn a new worker process, returns Worker instance

Properties:

- isMaster/isWorker – is/isn't current process master-process
- worker - reference to the current worker object (not available in the master process)
- workers – object with IDs as keys and workers as values (only available in master process)

Class Worker:

- wraps ChildProcess instance, which was originally created by child_process.fork()
- property "id" – unique id for worker, key in cluster.workers
- property "process" – ChildProcess instance
- method "send()" - send a message to master (from worker) or to worker (from master)

NODEJS.EN 2.6 → CLUSTER MODULE

```
const cluster = require('cluster')
const numCPUs = require('os').cpus().length

if (cluster.isMaster) {
 masterProcess()
} else {
 childProcess()
}

function masterProcess() {
 console.log(`Master ${process.pid} is running`)

 for (let i = 0; i < numCPUs; i++) {
   console.log(`Forking process number ${i}`)
   cluster.fork()
 }
 process.exit()
}

function childProcess() {
 console.log(`Worker ${process.pid} started and finished`)
 process.exit()
}
```

NODEJS.EN 2.6 → SCALABLE EXPRESS SERVER WITH CLUSTER


```
const cluster = require('cluster')

if(cluster.isMaster) {
   const numWorkers = require('os').cpus().length
   console.log('Master cluster setting up ' + numWorkers + ' workers...')
   for(let i = 0; i < numWorkers; i++) {
       cluster.fork();
   }
   cluster.on('online', (worker) => {
       console.log('Worker ' + worker.process.pid + ' is online')
   })
   cluster.on('exit', (worker, code, signal) => {
       cluster.fork()
   })

} else {
   const app = require('express')()
   app.all('/*', (req, res) => {
       res.send('process ' + process.pid + ' says hello!').end()
   })
   const server = app.listen(8080, () => {
       console.log('Process ' + process.pid + ' is listening to all incoming requests')
   })
}
```

- Node.js v10.1.0 Documentation https://nodejs.org/api/
- Quynh J., A Brief History of Node Streams
https://medium.com/the-node-js-collection/a-brief-history-of-node-streams-pt-1-3401db451f21
https://medium.com/the-node-js-collection/a-brief-history-of-node-streams-pt-2-bcb6b1fd7468
- Casciaro M., Node.js Design Patterns, 2nd edition, 2016 (you can easily find a pdf version in internet)
- Stream-handbook https://github.com/substack/stream-handbook
- Node.js Streams: Everything you need to know
https://medium.freecodecamp.org/node-js-streams-everything-you-need-to-know-c9141306be93
- Baumgartner S., The Definitive Guide to Object Streams in Node.js
https://community.risingstack.com/the-definitive-guide-to-object-streams-in-node-js/
- Santiago S., Understanding the NodeJS cluster module
http://www.acuriousanimal.com/2017/08/12/understanding-the-nodejs-cluster-module.html
- Kamali B., How to Create a Node.js Cluster for Speeding Up Your Apps
https://www.sitepoint.com/how-to-create-a-node-js-cluster-for-speeding-up-your-apps/
- Bansal V., Node Js Child Process Module http://www.tothenew.com/blog/node-js-child-process-module/