**Buffering vs Streaming**

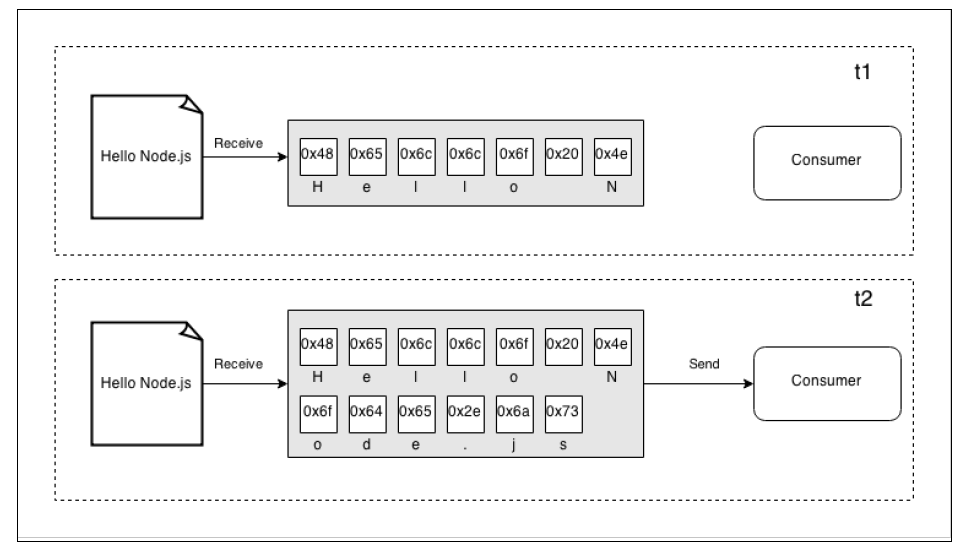
Almost all the asynchronous APIs that we've seen so far in the book work using

the *buffer mode*. For an input operation, the buffer mode causes all the data coming

from a resource to be collected into a buffer; it is then passed to a callback as soon

as the entire resource is read. The following diagram shows a visual example

of this paradigm:



In the preceding figure, we can see that, at the time t1, some data was received

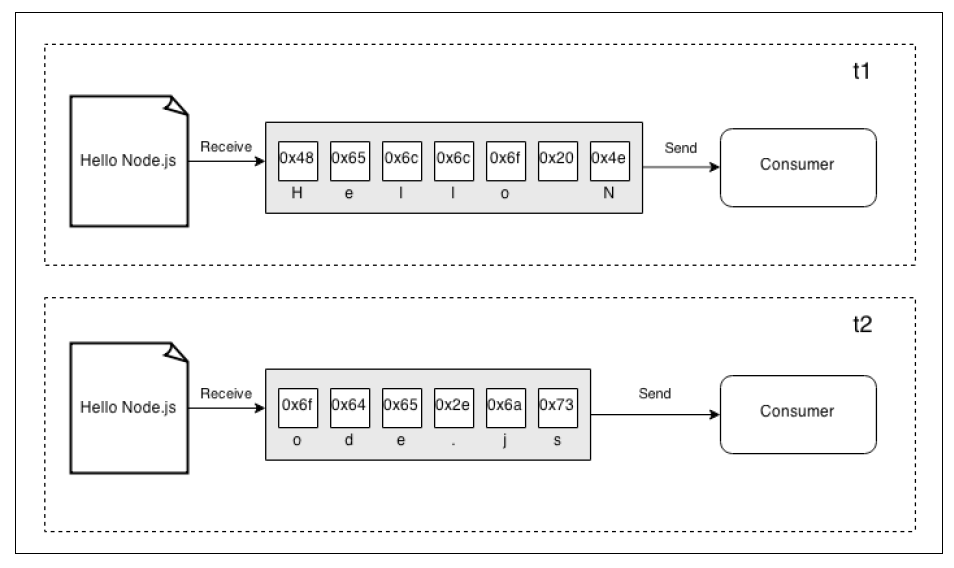
from the resource and saved into the buffer. At the time t2, another data chunk is

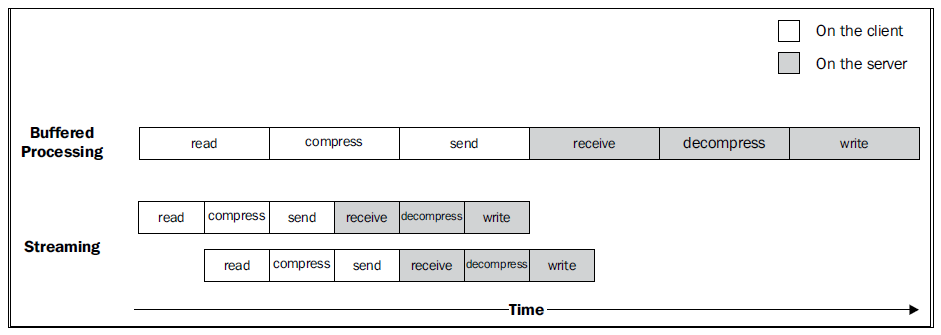
received—the final one—that completes the read operation and causes the entire

buffer to be sent to the consumer.

On the other side, streams allow you to process the data as soon as it arrives from the

resource. This is shown in the following diagram:





When a file is processed it goes through a set of sequential stages:

1. [Client] Read from the filesystem.

2. [Client] Compress the data.

3. [Client] Send it to the server.

4. [Server] Receive from the client.

5. [Server] Decompress the data.

6. [Server] Write the data to disk.

To complete the processing, we have to go through each stage like in an assembly

line, in sequence, until the end. In the preceding figure, we can see that, using a

buffered API, the process is entirely sequential. To compress the data, we first have

to wait for the entire file to be read, then, to send the data, we have to wait for the

entire file to be both read and compressed, and so on. When instead we are using

streams, the *assembly line* is kicked off as soon as we receive the first chunk of data,

without waiting for the entire file to be read. But more amazingly, when the next

chunk of data is available, there is no need to wait for the previous set of tasks to be

completed; instead, another *assembly line* is launched in parallel. This works perfectly

because each task that we execute is asynchronous, so it can be parallelized by Node.

js; the only constraint is that the order in which the chunks arrive in each stage must

be preserved (and Node.js streams take care of this for us).

As we can see from the previous figure, the result of using streams is that the entire

process takes less time, because we waste no time in waiting for all the data to be

read and processed all at once.

const fs = require('fs');

const path = require('path');

const generateSizeFile = (filename) => {

const targetSizeBytes = 1024 \* 1024 \* 1024; // 1GB

const chunkSize = 1024 \* 1024; // 1MB chunk size

const writeChunk = () => {

// Check if the file exists and its size is less than the target size

if (!fs.existsSync(filename) || fs.statSync(filename).size < targetSizeBytes) {

const buffer = Buffer.alloc(chunkSize);

fs.appendFileSync(filename, buffer);

process.nextTick(writeChunk); // Continue writing

} else {

console.log(`File ${filename} successfully generated at: ${path.resolve(filename)}`);

}

};

writeChunk();

};

const filename = './testFile1GB.dat';

generateSizeFile(filename);

1. **const fs = require('fs');** and **const path = require('path');**: Import the Node.js File System and Path modules for file operations and path resolution.
2. **const generateSizeFile = (filename) => {**: Define a function named **generateSizeFile** that takes a **filename** as an argument.
3. **const targetSizeBytes = 1024 \* 1024 \* 1024; // 1GB**: Set the target size for the file to be generated (1GB).
4. **const chunkSize = 1024 \* 1024; // 1MB chunk size**: Set the size of each chunk to be appended to the file (1MB).
5. **const writeChunk = () => {**: Define an inner function named **writeChunk**.
6. **if (!fs.existsSync(filename) || fs.statSync(filename).size < targetSizeBytes) {**: Check if the file doesn't exist or its size is less than the target size.
7. **const buffer = Buffer.alloc(chunkSize);**: Create a buffer with the specified chunk size.
8. **fs.appendFileSync(filename, buffer);**: Append the buffer to the file synchronously.
9. **process.nextTick(writeChunk); // Continue writing**: Schedule the next chunk to be written in the next tick of the event loop.
10. **} else { console.log(**File ${filename} successfully generated at: ${path.resolve(filename)}**); }**: If the file reaches the target size, print a success message with the resolved path of the generated file.
11. **writeChunk();**: Call the **writeChunk** function to start the file generation process.
12. **const filename = './testFile1GB.dat';**: Specify the filename for the generated file.
13. **generateSizeFile(filename);**: Call the **generateSizeFile** function with the specified filename to initiate the file generation.

This script generates a file by appending chunks of data until the file reaches the target size, and it prints a success message when the generation is complete. Adjustments can be made to the **targetSizeBytes** and **chunkSize** variables based on specific requirements.

**Anatomy of streams**

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

available in the stream core module:

• stream.Readable

• stream.Writable

• stream.Duplex

• stream.Transform

import {Transform} from 'stream';

export class LimitedParallelStream extends Transform {

    constructor(concurrency, userTransform, opts) {

        super({...opts, objectMode: true})

        this.concurrency = concurrency;

        this.userTransform = userTransform;

        this.running = 0;

        this.continueCB = null;

        this.terminateCB = null;

    }

    \_transform(chunk, enc, done) {

        this.running++;

        this.userTransform(

            chunk,

            enc,

            this.push.bind(this)

        )

        if (this.running < this.concurrency) {

            done();

        } else {

            this.continueCB = done;

        }

    }

    \_flush(done) {

        if (this.running > 0) {

            this.terminateCB = done;

        } else {

            done();

        }

    }

    \_onComplete(err) {

        this.running--

        if (err) {

            return this.emit('error', err)

        }

        const tmpCp = this.continueCB

        this.continueCB = null

        tmpCp && tmpCp()

        if (this.running === 0) {

            this.terminateCB && this.terminateCB()

        }

    }

}

**Constructor:**

* The constructor sets up the **LimitedParallelStream** class.
* It takes three parameters:
  + **concurrency**: The maximum number of concurrent transformations.
  + **userTransform**: The user-defined transformation function.
  + **opts**: Additional options for the Transform stream.
* It calls the parent class constructor with **objectMode** set to **true**.

**\_transform Method:**

* This method is called for each incoming chunk.
* It increments the **running** count, indicating an active transformation.
* It invokes the user-defined transformation function (**userTransform**) with the chunk and a callback (**this.push.bind(this)**).
* If the number of active transformations is less than the specified concurrency (**this.concurrency**), it calls the provided **done** callback to continue processing.
* Otherwise, it saves the **done** callback (**this.continueCB**) for later execution.

**\_flush Method:**

* This method is called when there are no more incoming chunks.
* If there are still active transformations (**this.running > 0**), it sets the **terminateCB** for later execution.
* If there are no active transformations, it calls the provided **done** callback.

**\_onComplete Method:**

* This method is called to signal the completion of a transformation.
* It decrements the **running** count.
* If an error occurred during the transformation, it emits an 'error' event.
* It executes the saved **continueCB** (if any) to continue processing if there is room for more transformations.
* If there are no more active transformations (**this.running === 0**), it executes the saved **terminateCB** to complete the stream.

**Client side – Multiplexing**

1. **import {fork} from 'child\_process';**: Import the **fork** function from the 'child\_process' module, which is used to spawn child processes.
2. **import {connect} from 'net';**: Import the **connect** function from the 'net' module, which is used to establish a TCP connection.
3. **function multiplexChannels (sources, destination) {**: Define a function named **multiplexChannels** that takes an array of **sources** (readable streams) and a **destination** (writable stream).
4. **let totalChannels = sources.length;**: Initialize a variable **totalChannels** with the number of sources, representing the total number of channels.
5. **for (let i = 0; i < sources.length; i++) {**: Iterate over each source stream.
6. **sources[i].on('readable', function() {**: Attach a 'readable' event listener to each source stream.
7. **let chunk**: Declare a variable to store chunks of data read from the source stream.
8. **while ((chunk = this.read()) !== null) {**: Use a while loop to read chunks from the source stream until no more data is available.
9. **const outBuff = Buffer.alloc(1 + 4 + chunk.length)**: Create a new buffer (**outBuff**) with a specific size to hold channel index, data length, and the chunk of data.
10. **outBuff.writeUInt8(i, 0)**: Write the channel index to the buffer at position 0.
11. **outBuff.writeUInt32BE(chunk.length, 1)**: Write the length of the data chunk to the buffer at position 1 using Big Endian encoding.
12. **chunk.copy(outBuff, 5)**: Copy the data chunk to the buffer starting from position 5.
13. **console.log(**Sending packet to channel: ${i}**)**: Log a message indicating that a packet is being sent to a specific channel.
14. **destination.write(outBuff)**: Write the constructed buffer to the destination (writable stream).
15. **}).on('end', () => {**: Attach an 'end' event listener to each source stream.
16. **if (--openChannels === 0) {**: Check if the number of open channels has reached zero.
17. **destination.end()**: End the destination stream if there are no more open channels.
18. **const socket = connect(3000, () => {**: Establish a TCP connection to the specified port (3000) using the **connect** function.
19. **const child = fork(**: Spawn a child process using the **fork** function.
20. **process.argv[2], process.argv.slice(3), {silent:true}**: Pass the script name, command-line arguments, and options to the child process. The **silent:true** option suppresses the output to the parent's stdio streams.
21. **multiplexChannels([child.stdout, child.stderr], socket)**: Call the **multiplexChannels** function with the child process's stdout and stderr streams as sources and the established socket as the destination. This multiplexes the output streams of the child process through the TCP socket.

Note: There is a reference to **openChannels** in the 'end' event listener, but the variable is not defined in the provided code snippet. It seems to be a mistake, and you might want to replace it with **totalChannels**.