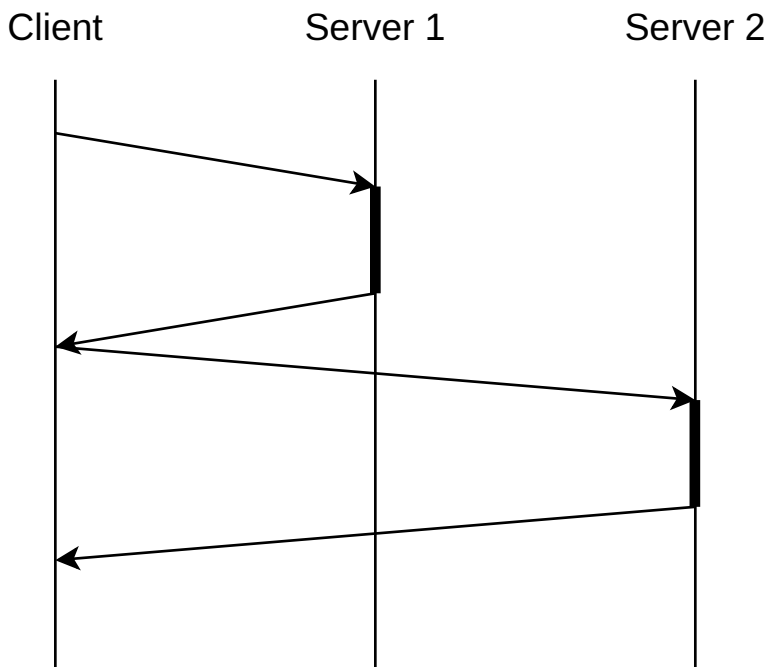
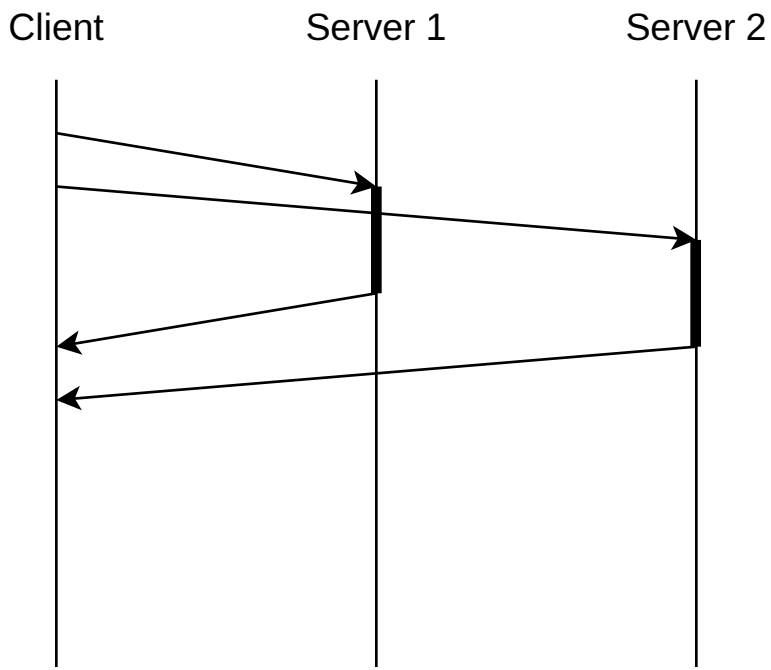
Asynchronous Communication

Avoid Blocking



Synchronous



Asynchronous

When to use it

Parallelism across different machines (drives, servers, ...)

Input and output (files, network, ...)

Only when there is more than one device!

JavaScript with Node.js

https://developer.mozilla.org/en-US/docs/Web/JavaScript

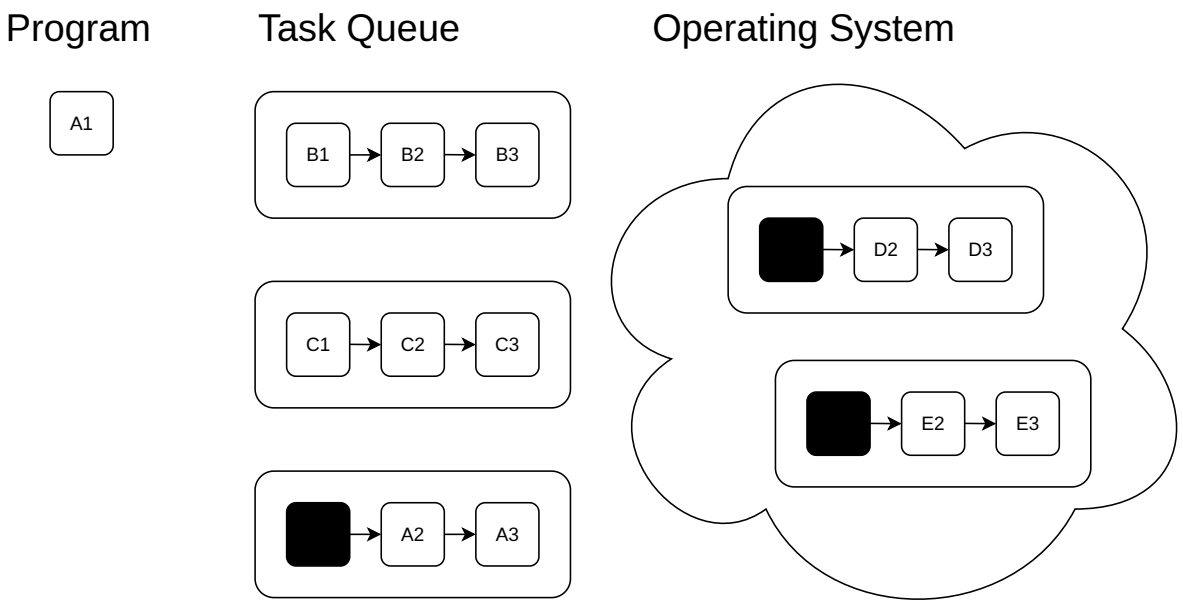
```
import { createServer } from 'node:http';

let server = createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World!\n');
});

server.listen(3000, '127.0.0.1', () => {
  console.log('Listening on 127.0.0.1:3000');
});
```

JavaScript originally designed by Brendan Eich, around 1995

Node.js originally created by Ryan Dahl, 2009

demo/server1.js

Eventloop

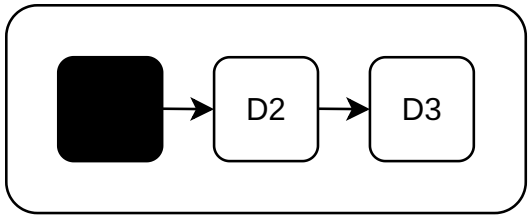


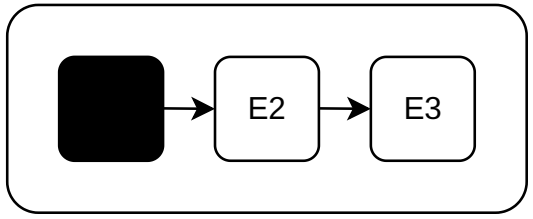
Runs on a single processor.

Push-based, not pull-based.

Continuations (Callbacks)

Instructions on how to continue.





Inversion of control. "Don't call us, we call you."

Continuation-Passing Style (CPS)

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

function multiply(a, b, k) {
  k(a * b);
}

function multiply_add(x, k) {
  // 3 * x + 5
  multiply(3, x, y => add(y, 5, k));
}

multiply_add(4, r => console.log(r));
```

demo/continuations.js

CPS for Asynchronous Communication

```
setTimeout(() => console.log('A done'), 1000);
setTimeout(() => console.log('B done'), 2000);
```

```
setTimeout(() => {
  console.log('A done');
  setTimeout(() => {
    console.log('B done');
  }, 2000);
}, 1000);
```

demo/timing.js

demo/callback_client.js

Error Handling with Continuations

Exceptions in callbacks cannot be caught.

A common pattern is to pass an error handling callback.

demo/filesystem.js

Promises in Node.js

`Promise.resolve` enqueues a new task, `.then` defines task steps.

```
console.log('START');

Promise.resolve()
  .then(() => console.log('A1'))
  .then(() => console.log('A2'));

Promise.resolve()
  .then(() => console.log('B1'))
  .then(() => console.log('B2'));

console.log('END');
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

demo/eventloop.js

demo/promise_client.js

Fulfilling Promises

A Promise represents a future value: it may be pending, fulfilled, or rejected.

A new promise starts as pending.

Fulfilling a promise means giving it a value.

Rejecting a promise means giving it an error value.

Fulfilling Promises in JavaScript

```
let A = new Promise((resolve) => {
  fs.readFile('input1.txt', 'utf8', (err, data) => {
    resolve(data);
  });
});

let B = new Promise((resolve) => {
  fs.readFile('input2.txt', 'utf8', (err, data) => {
    resolve(data);
  });
});

A.then((a) => {
  B.then((b) => {
    console.log(a, b);
  });
});
```

demo/fulfill.js

Error Handling with Promises

Promises propagate the error to the conceptual caller.

```
let A = new Promise((resolve, reject) => {
  fs.readFile('input1.txt', 'utf8', (err, data) => {
    if (err) reject(err);
    else resolve(data);
  });
});

A.then((a) => {
  B.then((b) => {
    console.log(a, b);
  });
}).catch((err) => {
  console.log('Error:', err);
});
```

demo/errors.js

Async Await Syntax

Transform this:

```
async function f() {
  let a = fetch("a.net");
  let b = fetch("b.net");
  return (await a) + (await b);
}
```

Into:

```
function f() {
  let a = fetch("a.net");
  let b = fetch("b.net");
  return a.then( (x) =>
    b.then( (y) =>
      Promise.resolve(x + y)
    )
  );
}
```

Async Syntax in JavaScript

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

> The async function declaration creates a binding of a new async function to a given name. The `await` keyword is permitted within the function body, enabling asynchronous, promise-based behavior to be written in a cleaner style and avoiding the need to explicitly configure promise chains.

```
import * as fs from 'node:fs/promises';

let a = fs.readFile('input1.txt', 'utf8');
let b = fs.readFile('input2.txt', 'utf8');

console.log(await a, await b);
```

demo/await.js

Error Handling with Async Syntax

Error is rethrown at await point.

```
import * as fs from 'node:fs/promises';

try {
  let a = fs.readFile('input1.txt', 'utf8');
  let b = fs.readFile('input2.txt', 'utf8');
  console.log(await a, await b);
} catch (err) {
  console.log('Error:', err);
}
```

demo/await_errors.js

demo/await_client.js

Async Generators in JavaScript

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function*

```
async function* asyncCounter() {
  for (let i = 0; i < 4; i++) {
    await sleep(1000);
    yield i;
  }
}
```

Asynchronous User Interfaces in JavaScript

Waiting for communication should not freeze the user interface.

```
for (let i = 0; i < 1e8; i++) {
  counter.textContent = i;
}
```

After updating the document we should yield control to the browser.

```
let i = 0;
function step() {
    counter.textContent = i;
    i++;
    if (i < 1e8) setTimeout(step, 0);
}
```

demo/freeze.html

Races

Racing two promises for completion gives a non-deterministic result.

```
let timeout = new Promise((_, reject) =>
  setTimeout(() => reject(new Error('timeout')), 1000)
);

let request = fetch('http://localhost:8001')
  .then(response => response.text());

Promise.race([request, timeout]).then(result => {
  console.log(result);
}).catch(err => {
  console.log(err.message);
});
```

Now we are starting to talk about concurrency.

Summary and Outlook

Asynchronous communication enables parallelism across devices.

Next week: Channels.