

# UDP Programming

RES, Lecture 5

---

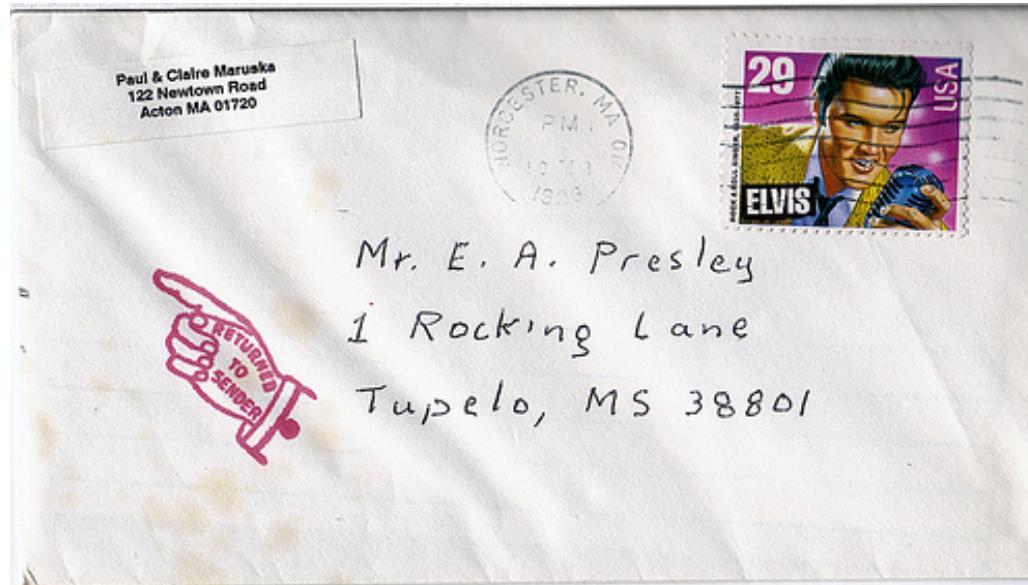
Olivier Liechti



HAUTE ÉCOLE  
D'INGÉNIERIE ET DE GESTION  
DU CANTON DE VAUD

[www.heig-vd.ch](http://www.heig-vd.ch)

# The UDP Protocol





TCP



UDP



# The UDP Protocol

- When using UDP, **you do not work with the abstraction of an IO stream.**
- Rather, **you work with the abstraction of individual datagrams**, which you can send and receive. Every datagram is independent from the others.
- **You do not have any guarantee:** datagrams can get lost, datagrams can arrive out-of-order, datagrams can be duplicated.
- What do you put “**inside**” the datagram (i.e. what is the payload)?
  - A notification.
  - A request, a query, a command. A reply, a response, a result.
  - A portion of a data stream (managed by the application).
- What do you put “**outside**” of the datagram (i.e. what is the header)?
  - A destination address (IP address in the IP packet header + UDP port)
  - A source address (IP + port)

# Unicast, Broadcast, Multicast

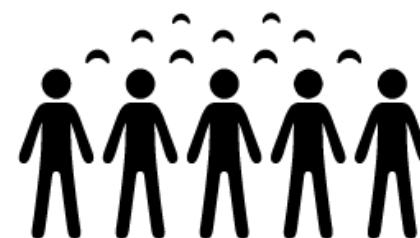
---

## Destination Address

192.168.10.2



255.255.255.255



239.255.22.5

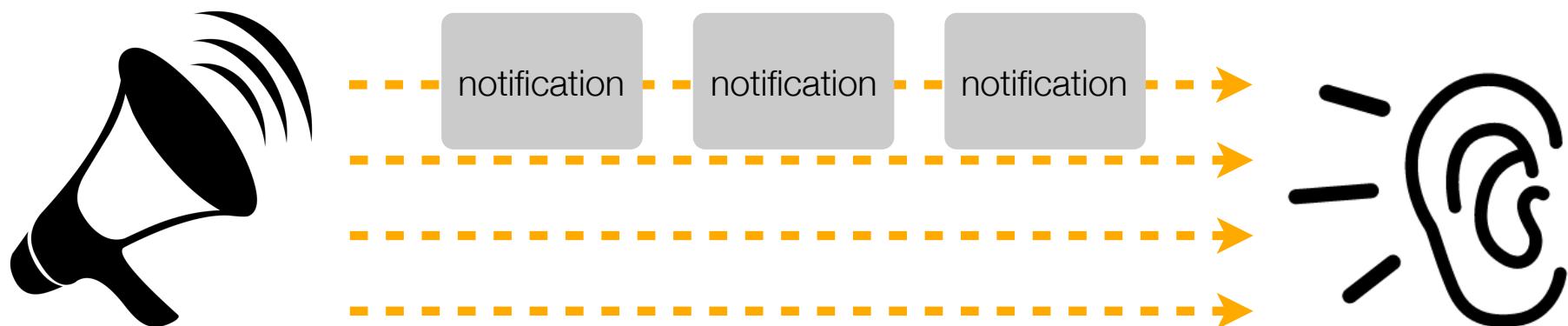


# Messaging Patterns



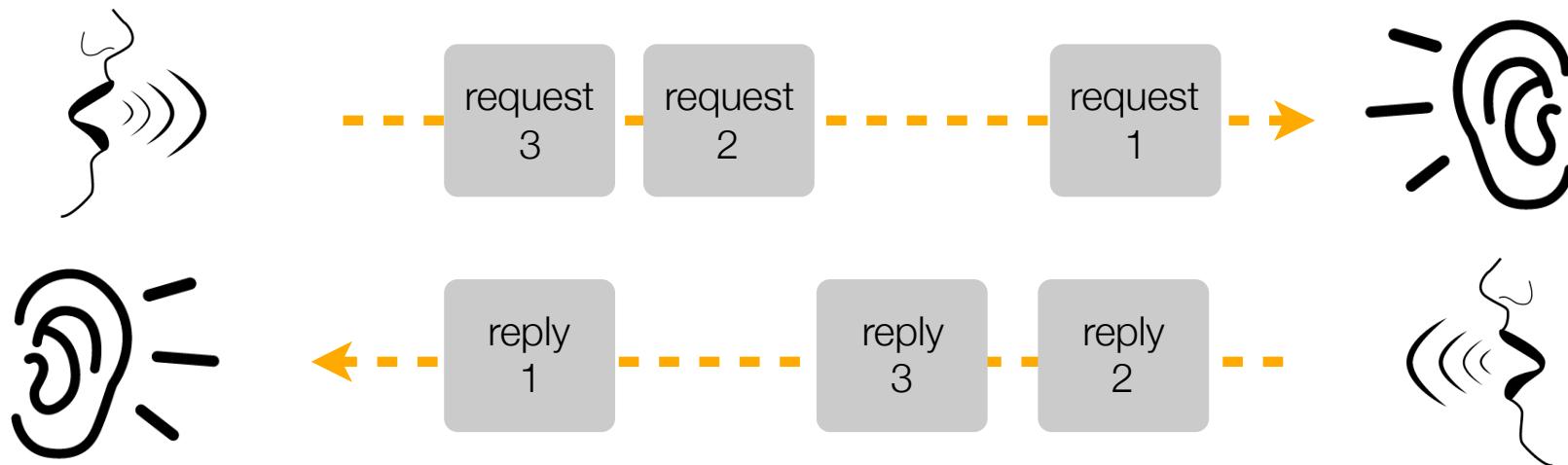
# Pattern: Fire-and-Forget

- The sender generates messages, sends them to a single receiver or to a group of receivers.
- The sender may do that on a **periodic** basis.
- The sender does not expect any answer. **He is telling, not asking.**



# Pattern: Request-Reply

- This pattern is used to implement the **typical client-server model**. Both the client and the server **produce and consume** datagrams.
- **The client produces requests** (aka queries, commands). The client also listens for incoming replies (aka responses, results). The server listens for requests. **The server also sends replies back to the client.**
- Can the client send a new request, even if he has not received the response to the previous request yet? If yes, and because UDP datagrams can be delivered out of order, how can the client **associate a reply** with the corresponding request?



# Service Discovery Protocols

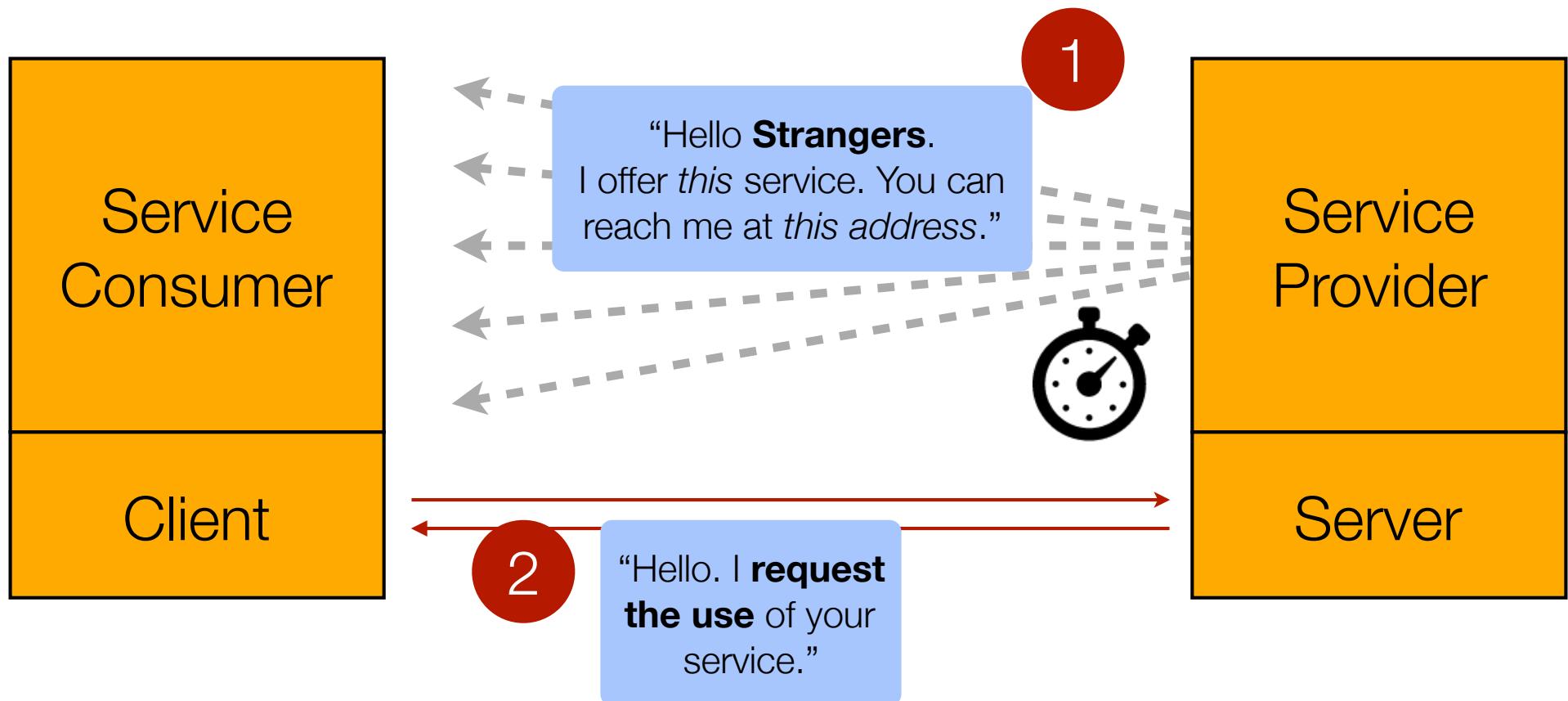


# Service Discovery Protocols

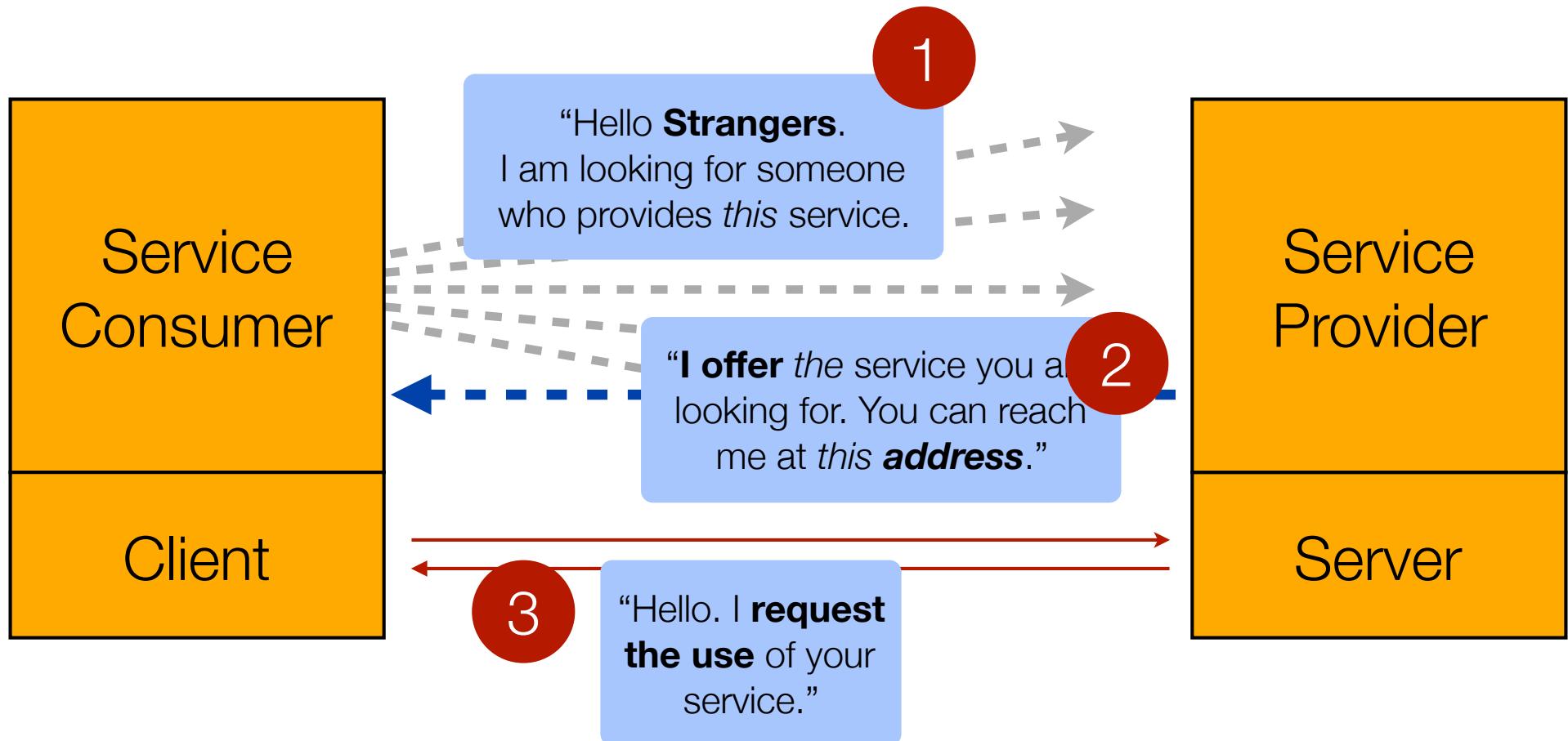
- With **unicast** message transmission, the sender must know who the receiver is and must know how to reach it (i.e. know its address).
- With **broadcast** and **multicast**, this is not required. The sender knows that nodes that are either *nearby*, or that have expressed *their interest*, will receive a copy of the message.
- This property can be used to **discover the availability of services or resources** in a *dynamic environment*.



# Model 1: Advertise Service Periodically



# Model 2: Query Service Availability



# Reliability



# Reliability

- UDP is **not a reliable transport protocol**:
  - Datagrams can arrive in a different order from which they were sent.
  - Datagrams can get lost.
- For some application-level protocols, this is not an issue because they are **tolerant to data loss**. For example, think of a **media streaming** protocol.
- But **if no data loss can be tolerated at the application level** (which is typically the case for file transfer protocols), does it mean that it is impossible to use UDP?

# Who Is Responsible for Reliability?



- **It is actually possible** to implement reliable application-level protocols on top of UDP, but...
  - It is the **responsibility of the application-level protocol specification** to include appropriate mechanisms to overcome the limitations of UDP.
  - It is the **responsibility of the application developer** to implement these mechanisms.
- In other words, it is up to the application developer to “**do the kind of stuff that TCP usually does**”.
- Do you remember about **acknowledgments, timers, retransmissions, stop-and-wait, sliding windows**, etc?

# A First Example: TFTP

- **Trivial File Transfer Protocol (TFTP)**
  - <http://tools.ietf.org/html/rfc1350>
  - “TFTP is a very simple protocol used to transfer files. It is from this that its name comes, Trivial File Transfer Protocol or TFTP. **Each nonterminal packet is acknowledged separately.** This document describes the protocol and its types of packets. The document also explains the reasons behind some of the design decisions.”
  - “Any transfer begins with a request to read or write a file, which also serves to request a connection. If the server grants the request, the connection is opened and the file is sent in fixed length blocks of 512 bytes. Each data packet contains one block of data, and must be acknowledged by an acknowledgment packet **before** the next packet can be sent.”

# A Second Example: CoAP

- **Constrained Application Protocol (CoAP)**

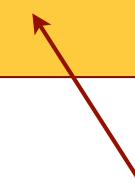
- <http://tools.ietf.org/html/draft-ietf-core-coap-18>
- “The Constrained Application Protocol (CoAP) is a **specialized web transfer protocol** for use with constrained nodes and constrained (e.g., low-power, lossy) networks. [...] The protocol is designed for **machine-to-machine (M2M) applications** such as smart energy and building automation.”
- “CoAP provides a **request/response interaction model** between application endpoints, supports built-in discovery of services and resources, and includes key concepts of the Web such as URLs and Internet media types.”
- “**2.1. Messaging Model.** The CoAP messaging model is based on the exchange of messages over UDP between endpoints. CoAP uses a short fixed-length binary header (4 bytes) that may be followed by compact binary options and a payload. This message format is shared by requests and responses. The CoAP message format is specified in Section 3. **Each message contains a Message ID used to detect duplicates and for optional reliability.** (The Message ID is compact; its 16-bit size enables up to about 250 messages per second from one endpoint to another with default protocol parameters.)
- “**4.2. Messages Transmitted Reliably.** The reliable transmission of a message is initiated by marking the message as Confirmable in the CoAP header. [...] A recipient MUST **acknowledge** a Confirmable message with an Acknowledgement message [...].”

# Using the Socket API for a UDP **Sender**

1. Create a datagram socket (without giving any address / port)
2. Create a datagram and put some data (bytes) in it
3. Send the datagram via the socket, specifying the destination address and port

**If a reply is expected:**

4. Accept incoming datagrams on the socket



If we do not specify the port when creating the socket, the operating system will **automatically assign a free one** for us.

This port will be in the “source port” field of the UDP header. The receiver of our datagram will extract this value and will use it to send us the reply (it is a part of the **return address**).

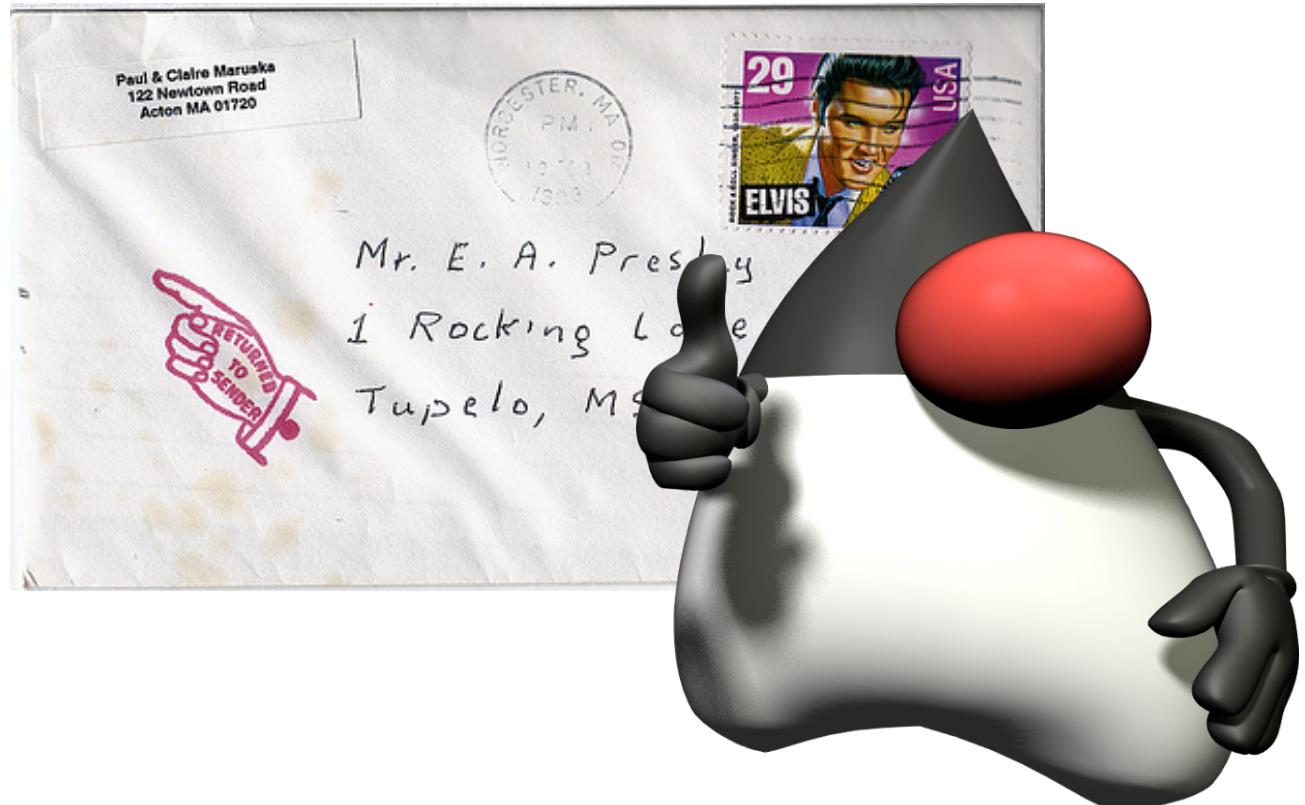
# Using the Socket API for a UDP Receiver

Application-level protocol specifications often define **a standard UDP port**, where the clients can send their requests. When this is the case, we have to specify the port number when creating the socket.

1. Create a datagram socket, specifying a particular port
2. Loop
  - 2.1. Accept an incoming datagram via the socket
  - 2.2. Process the datagram
  - 2.3. If a reply is expected by the sender
    - 2.3.1. Extract the return address from the request datagram
    - 2.3.2. Prepare a reply datagram
    - 2.3.3. Send the reply datagram via the socket

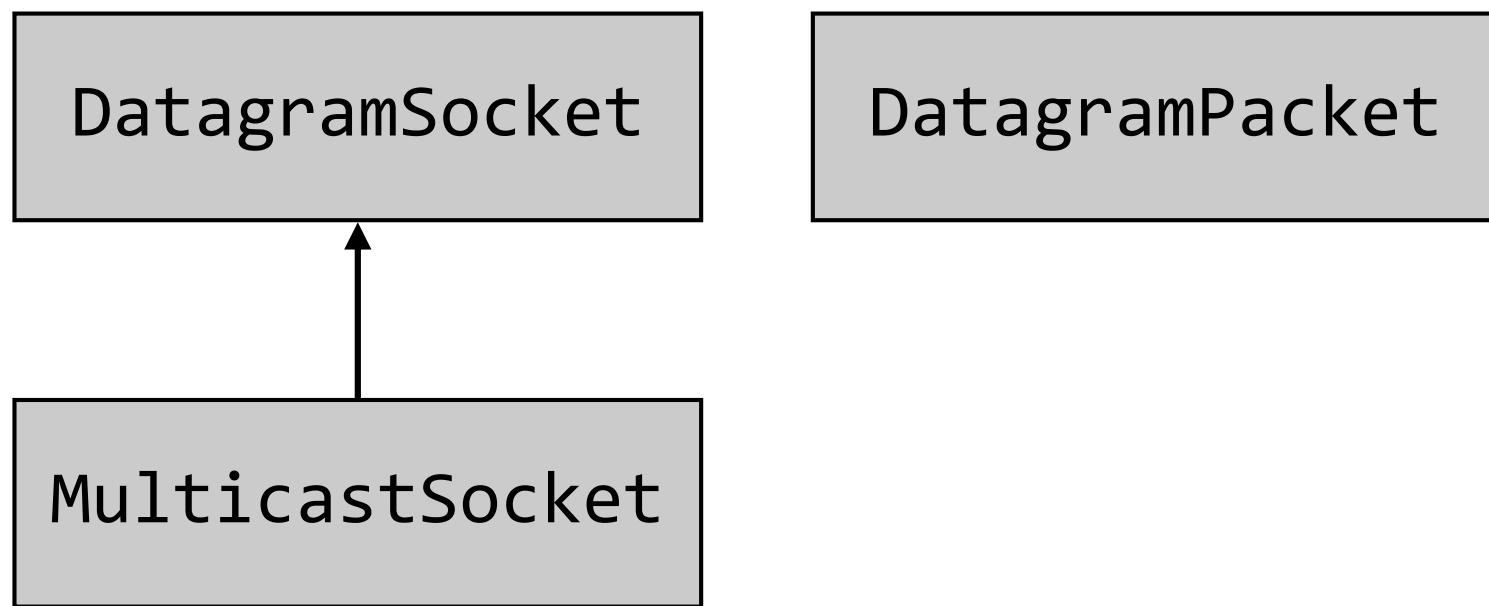
It is at this stage that we extract the source port and source IP address from the incoming datagram. We have found the **return address**, so we know where to send the response.

# Using UDP in Java



# Using UDP in Java

`java.net`



```
// Sending a message to a multicast group

socket = new DatagramSocket();
byte[] payload = "Java is cool, everybody should know!!".getBytes();

DatagramPacket datagram = new DatagramPacket(payload, payload.length,
InetAddress.getByName("239.255.3.5"), 4411);

socket.send(datagram);
```

## Publisher (multicast) Subscriber (multicast)

```
// Listening for broadcasted messages on the local network

MulticastSocket socket = new MulticastSocket(port);
InetAddress multicastGroup = InetAddress.getByName("239.255.3.5");
socket.joinGroup(multicastGroup);

while (true) {
    byte[] buffer = new byte[2048];
    DatagramPacket datagram = new DatagramPacket(buffer, buffer.length);
    try {
        socket.receive(datagram);
        String msg = new String(datagram.getData(), datagram.getOffset(), datagram.getLength());
        LOG.log(Level.INFO, "Received a datagram with this message: " + msg);
    } catch (IOException ex) {
        LOG.log(Level.SEVERE, ex.getMessage(), ex);
    }
}
socket.leaveGroup(multicastGroup);
```

```
// Broadcasting a message to all nodes on the local network

socket = new DatagramSocket();
socket.setBroadcast(true);

byte[] payload = "Java is cool, everybody should know!!".getBytes();

DatagramPacket datagram = new DatagramPacket(payload, payload.length,
InetAddress.getByName("255.255.255.255"), 4411);

socket.send(datagram);
```

## Subscriber

## Publisher

```
// Listening for broadcasted messages on the local network

DatagramSocket socket = new DatagramSocket(port);

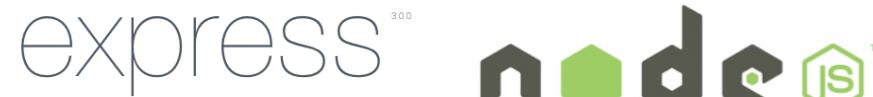
while (true) {
    byte[] buffer = new byte[2048];
    DatagramPacket datagram = new DatagramPacket(buffer, buffer.length);
    try {
        socket.receive(datagram);
        String msg = new String(datagram.getData(), datagram.getOffset(), datagram.getLength());
        LOG.log(Level.INFO, "Received a datagram with this message: " + msg);
    } catch (IOException ex) {
        LOG.log(Level.SEVERE, ex.getMessage(), ex);
    }
}
```

# JavaScript 101

# JavaScript - End to End



Client



Server

# Experimenting with JavaScript



HAUTE ÉCOLE  
D'INGÉNIERIE ET DE GESTION  
DU CANTON DE VAUD  
[www.heig-vd.ch](http://www.heig-vd.ch)

*What do I need to write, execute and debug JavaScript code?*

- Should I work on the **server side** or on the **client side**?
- Should I use a **simple text editor** or a **complete IDE**?
- Should I rather use an **online programming environment**?
- What kinds of **developer tools**, such as debuggers, are available?

# Browsers and Developers Tools

# Chrome DevTools Overview

The Chrome Developer Tools (DevTools for short), are a set web authoring and debugging tools built into Google Chrome. The DevTools provide web developers deep access into the internals of the browser and their web application. Use the DevTools to efficiently track down layout issues, set JavaScript breakpoints, and get insights for code optimization.

**Note:** If you are a web developer and want to get the latest version of DevTools, you should use [Google Chrome Canary](#).

The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. The left pane displays the code in `run.html`. A blue box highlights the line `var status = document.getElementById("status");`. To the right of the code, the status bar indicates 'Continue to Here'. The right pane contains the 'Breakpoints' section, which lists two checked breakpoints: one at line 25 and another at line 32. The bottom status bar shows 'Line 25, Column 1'.

```
24 var status = document.getElementById("status");
25 Continue to Here
26     +completed) / benchmarks) * 100;
27     running: " + Math.round(percentage) + "%
28 Remove Breakpoint
29 Edit Breakpoint...
30 Disable Breakpoint
31 console.timestamp('Adding result');
32 var text = name + ': ' + result;
33 var results = document.getElementById("results");
34
```



The screenshot shows the Chrome DevTools Network tab with several network requests listed. A context menu is open over the third request, which is a GET request for 'data:image/gif;base...'. The menu options are: Open Link in New Tab, Copy Link Address, Copy Response Headers, Copy as cURL, and Copy All as HAR. The 'Copy All as HAR' option is highlighted with a blue selection bar. The Network tab also displays columns for Name, Path, Method, Status, Type, Initiator, Size, Time, and Timeline.

Name	Path	Method	Status	Type	Initiator	Size	Time	Timeline	
log01lw.png	/images/srp	GET	200	Image	img	123 KB	3 ms	3 ms	400 ms
v2_e5392fe6.png	ssl.gstatic.com/gb/images	GET	200	Image	img	123 KB	89 ms	89 ms	
data:image/gif;base...		GET	200	Image	img	0 B	0 ms	0 ms	
rs=AltRSTM...6xeJclSicN1c...	/xjs/_/js/k=xjss.en_US.iiOPis...	GET	200	Script	script	126 kB	58 ms	58 ms	
c99460e63ee217ee.js?bav=...	/extern_chrome	GET	200	Script	script	22.5 kB	52 ms	52 ms	

<https://developer.chrome.com/devtools>



## How to access the DevTools

## The DevTools window

## Inspecting the DOM and styles

## Working with the console

# Debugging JavaScript

## Improving network performance

## Audits

## Improving rendering performance

## JavaScript CSS performance

## Inspecting storage

## **Further reading**



“Node.js is a **platform** built on Chrome's JavaScript runtime for easily building **fast, scalable network applications.** **not only!**

Node.js uses an **event-driven, non-blocking I/O model** that makes it lightweight and efficient, perfect for **data-intensive real-time applications that run across distributed devices.**”

# Node.js v0.10.32 Manual & Documentation



HAUTE ÉCOLE  
D'INGÉNIERIE ET DE GESTION  
DU CANTON DE VAUD  
[www.heig-vd.ch](http://www.heig-vd.ch)

- [Assertion Testing](#)
- [Buffer](#)
- [C/C++ Addons](#)
- [Child Processes](#)
- [Cluster](#)
- [Console](#)
- [Crypto](#)
- [Debugger](#)
- [DNS](#)
- [Domain](#)
- [Events](#)
- [File System](#)

- [Globals](#)
- [HTTP](#)
- [HTTPS](#)
- [Modules](#)
- [Net](#)
- [OS](#)
- [Path](#)
- [Process](#)
- [Punycode](#)
- [Query Strings](#)
- [Readline](#)
- [REPL](#)
- [Stream](#)
- [String Decoder](#)
- [Timers](#)
- [TLS/SSL](#)
- [TTY](#)
- [UDP/Datagram](#)
- [URL](#)
- [Utilities](#)
- [VM](#)
- [ZLIB](#)

```
console.log("hello");
```

index.js

```
$ node index.js
hello
```

# JavaScript 101



HAUTE ÉCOLE  
D'INGÉNIERIE ET DE GESTION  
DU CANTON DE VAUD  
[www.heig-vd.ch](http://www.heig-vd.ch)

- Types
- Scopes
- Objects
- Prototypal inheritance
- Functions
- Constructors
- Arrays



*Things are changing... we do not cover ES 6 in these slides*

# JavaScript defines 6 types



```
const aNumber = 3.12;
const aBoolean = true;
const aString = "HEIG-VD";
const anObject = {
    aProperty: null
};

// t is true for all of these:
var t;
t = typeof aNumber === "number";
t = typeof aBoolean === "boolean";
t = typeof aString === "string";
t = typeof anObject === "object";
t = typeof anObject.aProperty ===
    "object";
t = typeof anObject.foobar ===
    "undefined";
```

- The 6 types are:
  - number
  - boolean
  - string
  - object
  - undefined
  - null
- null is a type, but `typeof null === object`.
- JavaScript is a dynamic language: when you declare a variable, you don't specify a type (and the type can change over time).

# Objects are dynamic bags of properties



HAUTE ÉCOLE  
D'INGÉNIERIE ET DE GESTION  
DU CANTON DE VAUD  
[www.heig-vd.ch](http://www.heig-vd.ch)

```
// let's create an object
const person = {
  firstName: 'olivier',
  lastName: 'liechti'
};

// we can dynamically add properties
person.gender = 'male';
person['zip'] = 1446;

// and delete them
delete person.zip;

for (let key in person) {
  console.log(key + " : " +
person[key]);
};
```

- There are different ways to **access properties** of an object.
- JavaScript is **dynamic**: it is possible to **add** and **remove** properties to an object at any time.
- Every object has a different list of properties (**no class**).

# Before ES6, the language did not have support for classes.

## There are 3 ways to create objects

```
// create an object with a literal
const person = {
  firstName: 'olivier',
  lastName: 'liechti'
};

// create an object with a prototype
const child = Object.create(person);

// create an object with a constructor
const child = new Person('olivier',
  'liechti');
```



- A **constructor** is function like any other (uppercase is a coding convention).
- It is the use of the **new** keyword that triggers the object creation process.

# Arrays are objects



HAUTE ÉCOLE  
D'INGÉNIERIE ET DE GESTION  
DU CANTON DE VAUD  
[www.heig-vd.ch](http://www.heig-vd.ch)

```
var fruits = ["apple", "pear"];

fruits.push("banana");
console.log(Object.getPrototypeOf(fruits));

for (var i=0; i<fruits.length; i++) {
  console.log("fruits[" + i + "] = " + fruits[i]);
}

var transformedFruits = fruits.map( function(fruit) {
  return fruit.toUpperCase();
});
transformedFruits.forEach( function(fruit) {
  console.log(fruit);
});

var count = fruits.reduce( function(val, fruit) {
  console.log("reducer invoked with " + val);
  return val+1;
}, 0);
console.log("There are " + count + " fruits in the array");
```

# Functions are objects

```
function aFunction() {  
}  
  
var f = function() {  
}  
  
var g = function g() {  
    g(); // recursive call  
}  
  
var h = function(functionParam) {  
    functionParam();  
}  
  
h(f);  
h(g);
```

## #1 Functions are objects

```
var aFunction = function() {  
    console.log("I am doing my job");  
};  
aFunction.aProperty = "aValue";  
aFunction();
```

## #2 Passing functions as arguments

```
var aFunction = function() {...};  
var anotherFunction = function( f1 ) {... f1(); ...};  
anotherFunction( aFunction );
```

## #3 Return functions as values

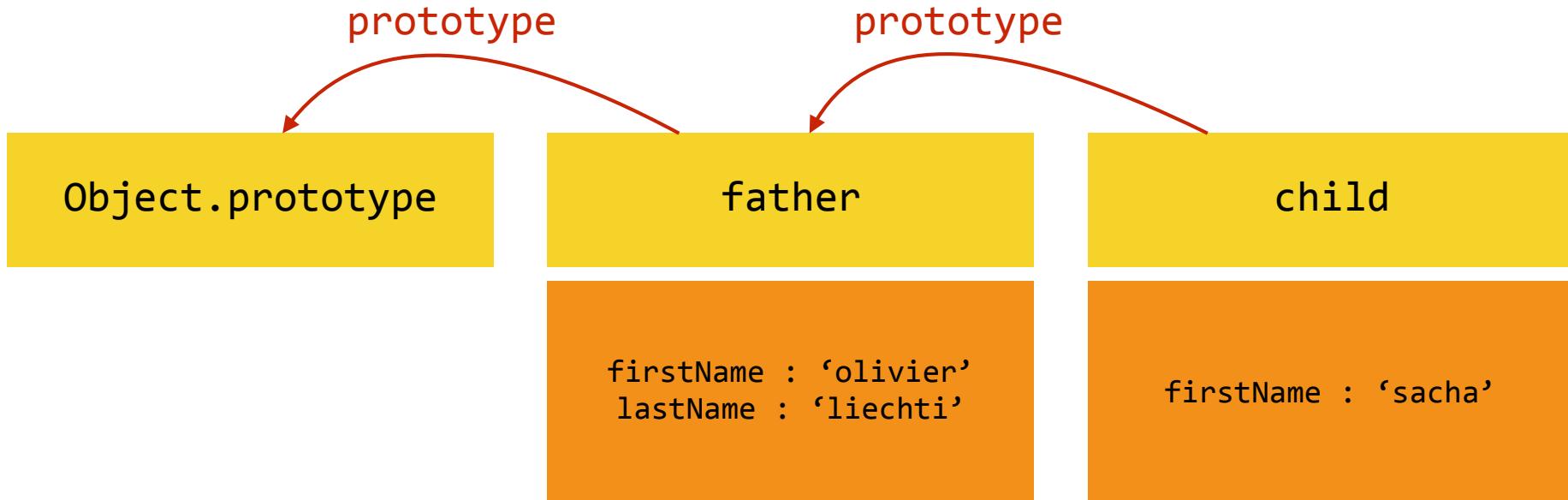
```
var aFunction = function() {  
    return function() {...}  
};  
aFunction()();
```

# Every object inherits from a prototype object



```
var person = {  
    firstName: "olivier",  
    lastName: "liechti"  
};  
// person's prototype is Object.prototype  
  
var father = {};  
var child = Object.create(father);  
// child's prototype is father  
  
function Person(fn, ln) {  
    this.firstName = fn;  
    this.lastName = ln;  
}  
var john = new Person("John", "Doe");  
// john's prototype is Person.prototype
```

# Every object inherits from a prototype object



```
console.log(child.lastName);
// prints 'liechti' on the
console
```

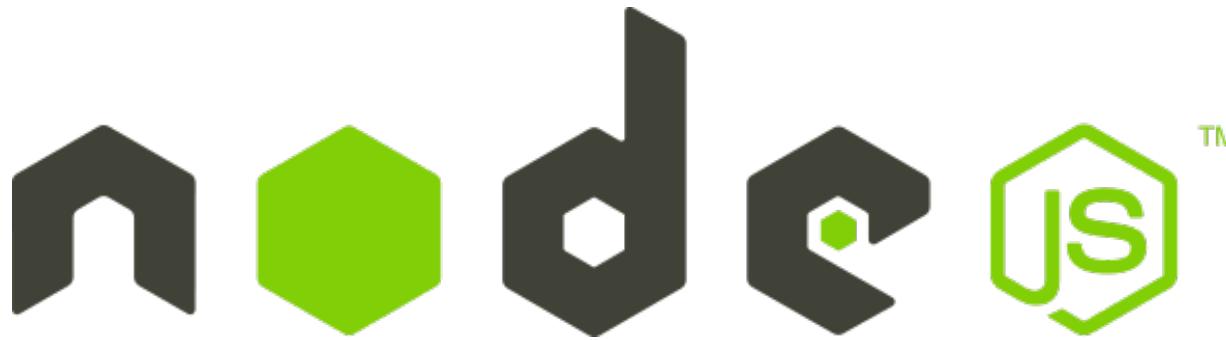
- Every object inherits from a prototype object. **It inherits and can override its properties**, including its methods.
- Objects created with object literals inherit from **`Object.prototype`**.
- When you access the property of an object, JavaScript **looks up the prototype chain** until it finds an ancestor that has a value for this property.

# With patterns, it is possible to implement class-like data structures



```
function Person(fn, ln) {  
    var privateVar;  
    this.firstName = fn;  
    this.lastName = ln;  
    this.badGreet = function() {  
        console.log("Hi " + this.firstName);  
    };  
};  
  
Person.prototype.greet = function() {  
    console.log("Hey " + this.firstName);  
};  
  
var p1 = new Person("olivier", "liechti");  
  
p1.badGreet();  
p1.greet();
```

- **badGreet** is a property that will be replicated for every object created with the Person constructor:
  - poor memory management
  - not possible to alter behavior of all instances at once
- **greet** is a property that will be shared by all instances (because it will be looked up along the object inheritance chain).
- **privateVar** is not accessible outside of the constructor.
- **firstName** is publicly accessible (no encapsulation).



# Node.js 101



## Let's look at a **first example (NOT** good)

```
/*global require */ ←  
  
var fs = require("fs"); ←  
  
/**  
 * Simple function to test the synchronous readFileSync function provided by  
 * Node.js  
 * @param {string} filename - the name of the file we want to read in the test  
 */  
  
function testSyncRead(filename) {  
    console.log("I am about to read a file: " + filename);  
    var data = fs.readFileSync(filename); ←  
    console.log("I have read " + data.length + " bytes (synchronously).");  
    console.log("I am done.");  
}  
  
// We get the file name from the argument passed on the command line  
var filename = process.argv[2]; ←  
  
console.log("\nTesting the synchronous call");  
testSyncRead(filename);
```

*This is for the Brackets editor*

We use a standard **Node module** for accessing the file system

**fs.readFileSync** is synchronous: it blocks the main thread until the data is available.

**process** is a global object provided by Node.js



```
$ node sample2.js medium.txt
```

Testing the synchronous call  
I am about to read a file: medium.txt  
I have read 1024 bytes (synchronously).  
I am done.

*Synchronous functions are easier to use, but they have **severe** performance implications!!*



## Let's look at a **second example (BETTER)**

```
/*global require */  
  
var fs = require("fs");  
  
/**  
 * Simple function to test the asynchronous readFile function provided by Node.js  
 * @param {string} filename - the name of the file we want to read in the test  
 */  
function testAsyncRead(filename) {  
    console.log("I am about to read a file: " + filename);  
  
    fs.readFile(filename, function (err, data) {  
        console.log("Nodes just told me that I have read the file.");  
    });  
  
    console.log("I am done. Am I really?????");  
}  
// We get the file name from the argument passed on the command line  
var filename = process.argv[2];  
  
console.log("\nTesting the asynchronous call");  
testAsyncRead(filename);
```

```
$ node sample2.js medium.txt  
  
Testing the asynchronous call  
I am about to read a file: medium.txt  
I am done. Am I?  
Nodes just told me that I have read the file.
```

**fs.readFile** is asynchronous:  
it does not block the main  
thread until the data is  
available.

We must provide a  
**callback function**, which  
Node.js will invoke when the  
data is available.

**Problems** can happen  
when an (asynchronous)  
function is called.



Node.js developers **have to** learn  
the asynchronous programming  
style.



## Let's look at a **third example**

```
/*global require */  
  
var http = require("http"); ←  
  
/**  
 * This function starts a http daemon on port 9000. It also  
 * registers a callback handler, to handle incoming HTTP  
 * requests (a simple message is sent back to clients).  
 */  
function runHttpServer() {  
    var daemon = http.createServer(); ←  
  
    daemon.on("request", function (req, res) { ←  
        console.log("A request has arrived: URL=" + req.url);  
        res.writeHead(200, {  
            'Content-Type': 'text/plain' ←  
        });  
        res.end('Hello World\n'); ←  
    });  
  
    console.log("Starting http daemon...");  
    daemon.listen(9000); ←  
}  
  
runHttpServer();
```

We use a standard **Node module** that takes care of the HTTP protocol.

Node can provide us with a **ready-to-use** server.

We can attach **event handlers** to the server. Node will notify us asynchronously, and give us access to the request and response.

We can **send back** data to the client.

We have wired everything, let's **welcome** clients!

HTTP Node.js v0.10.32 Manual & Documentation

nodejs.org/api/http.html#http\_event\_request

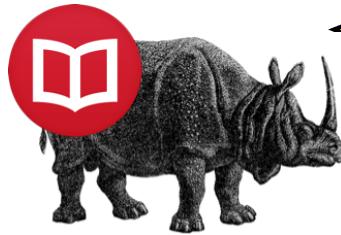
NPM Registry Docs Blog Community Logos Jobs

Index | View on single page | View as JSON

## Table of Contents

- HTTP
  - http.STATUS\_CODES
  - http.createServer([requestListener])
  - http.createClient([port], [host])
  - Class: http.Server
    - Event: 'request'
    - Event: 'connection'
    - Event: 'close'
    - Event: 'checkContinue'
    - Event: 'connect'
    - Event: 'upgrade'
    - Event: 'clientError'
    - server.listen(port, [hostname], [backlog], [callback])
    - server.listen(path, [callback])
    - server.listen(handle, [callback])
    - server.close([callback])
    - server.maxHeadersCount
    - server.setTimeout(msecs, callback)
    - server.timeout
  - Class: http.ServerResponse
    - Event: 'close'
    - Event: 'finish'
    - response.writeContinue()

These are the events that are **emitted** by the class. You can write callbacks and **react** to these events.



How does Node.js use an **event loop** to offer an asynchronous programming model?

```
on('request', function(req, res) { // my code});
```

```
on('data', function(data) { // my code});
```

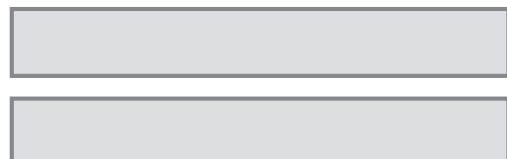
**Callback functions** that **you** have written and registered

'request' **event**

'request' **event**

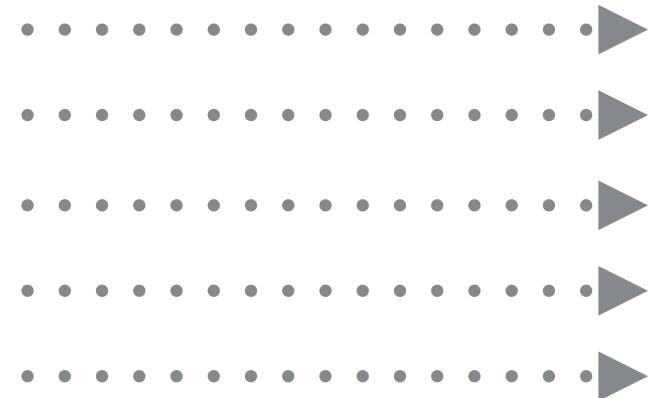
'data' **event**

'request' **event**



### Event Loop

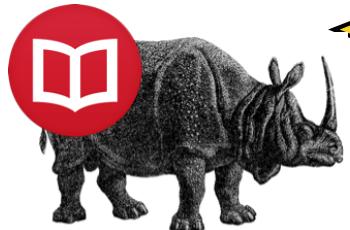
get the next event in the queue; invoke the registered callbacks in sequence; delegate I/O operations to the Node platform



All the code that **you** write runs on a **single thread**

The **long-running tasks** (I/Os) are executed by Node in parallel; Node emits events to report progress (which triggers your callbacks).

Another pattern is to provide a callback to node when invoking an asynchronous function.



## What is **npm**?

*"**npm** is the **package manager** for the Node JavaScript platform. It puts **modules** in place so that node can find them, and manages **dependency** conflicts intelligently."*

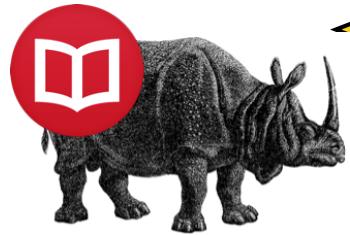
*It is extremely configurable to support a wide variety of use cases. Most commonly, it is used to **publish**, **discover**, **install**, and **develop** node programs."*

<https://www.npmjs.org/doc/cli/npm.html>



You **have to** read this:

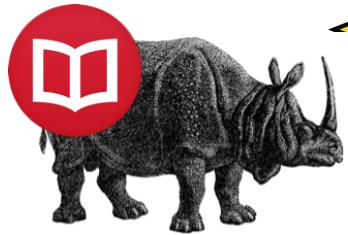
<https://www.npmjs.org/doc/misc/npm-faq.html>



npm is a set of command line tools that work together with the **node registry**

The screenshot shows a web browser window displaying the npm website (<https://www.npmjs.com>). At the top, there is a yellow promotional banner with a black rhinoceros icon and the text "npm is a set of command line tools that work together with the **node registry**". Below the banner, the main page features a large illustration of a nutty penguin holding a laptop and a mug. The text "private npm is here." is displayed, along with the offer "publish unlimited private modules for just \$7/month". There are "sign up" and "no thanks" buttons. The navigation bar includes links for "nutty penguin music", "npm private modules", "npm for the Enterprise", "documentation", "blog", "npm weekly", "jobs", and "support". The main content area has a red header with the "npm" logo and a search bar. Below this, the text "npm is the package manager for" is followed by four statistics: "187,190 total packages", "94,194,137 downloads in the last day", "568,188,116 downloads in the last week", and "2,515,481,679 downloads in the last month". A red link at the bottom reads "packages people 'npm install' a lot".

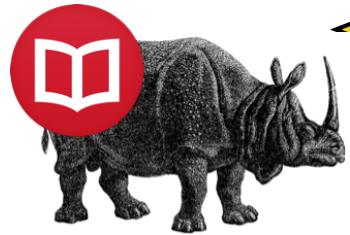
<https://www.npmjs.org>



npm is a set of command line tools that work together with the **node registry**

The screenshot shows a web browser window for the URL <https://www.npmjs.com/package/chance>. The page features a large banner at the top with a cartoon character holding a laptop and a mug, with the text "private npm is here. publish unlimited private modules for just \$7/month". Below the banner are two buttons: "sign up" and "no thanks". The main content area has a red header bar with links for "nerd play mate", "npm private modules", "npm for the Enterprise", "documentation", "blog", "npm weekly", "jobs", and "support". On the left, there's a sidebar for the "chance" package, which is described as a "Chance - Utility library" and includes an illustration of a die. The main search results for "expr" are listed, including "express", "express-session", "express-handlebars", "express-validator", and "express-jwt". To the right of the search results, there's a sidebar with links for "npm install chance", "victorquinn published 3 weeks ago", "0.7.7 is the latest of 33 releases", "github.com/victorquinn/chancejs", and "MIT license". At the bottom, there's a "Collaborators" section.

<https://www.npmjs.org>



npm is a set of command line tools that work together with the **node registry**

The screenshot shows the npmjs.com package page for 'express'. The page has a header with the npm logo and a search bar. Below the header, the package name 'express' is displayed with a 'public' badge. A brief description follows: 'Fast, unopinionated, minimalist web framework'. The main title 'express' is prominently displayed in large, lowercase letters. Below the title, there's a snippet of Node.js code demonstrating its usage. The page also includes sections for 'Installation' (with a command line example), 'Features' (listing 'Robust routing'), 'Keywords', 'Collaborators' (showing profile icons for several contributors), and 'Stats' (displaying download counts for the last day, week, and month, along with GitHub issue and pull request counts).

express public

Fast, unopinionated, minimalist web framework

# express

npm v4.13.3 | downloads 4M/month | linux | invalid | windows | passing | coverage 100%

```
var express = require('express')
var app = express()

app.get('/', function (req, res) {
  res.send('Hello World')
})

app.listen(3000)
```

## Installation

```
$ npm install express
```

## Features

- Robust routing

## Keywords

npm install express

dougwilson published 2 months ago

4.13.3 is the latest of 272 releases

github.com/strongloop/express

MIT license

## Collaborators

StrongLoop

## Stats

135,585 downloads in the last day

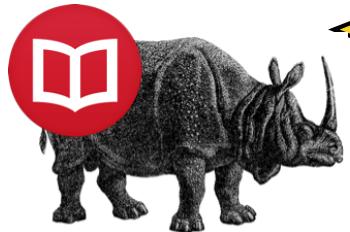
818,847 downloads in the last week

3,533,048 downloads in the last month

67 open issues on GitHub

28 open pull requests on GitHub

<https://www.npmjs.org>



npm is a set of command line tools that work together with the **node registry**

```
Last login: Wed Sep 23 19:59:09 on console
$ npm help

Usage: npm <command>

where <command> is one of:
  access, add-user, adduser, apihelp, author, bin, bugs, c,
  cache, completion, config, ddp, dedupe, deprecate, dist-tag,
  dist-tags, docs, edit, explore, faq, find, find-dupes, get,
  help, help-search, home, i, info, init, install, issues, la,
  link, list, ll, ln, login, ls, outdated, owner, pack,
  prefix, prune, publish, r, rb, rebuild, remove, repo,
  restart, rm, root, run script, s, se, search, set, show,
  shrinkwrap, star, stars, start, stop, t, tag, test, tst, un,
  uninstall, unlink, unpublish, unstar, up, update, v,
  verison, version, view, whoami

  npm <cmd> -h      quick help on <cmd>
  npm -l          display full usage info
  npm faq        commonly asked questions
  npm help <term>  search for help on <term>
  npm help npm    involved overview

Specify configs in the ini-formatted file:
  /Users/admin/.npmrc
or on the command line via: npm <command> --key value
Config info can be viewed via: npm help config

npm@2.5.1 /usr/local/lib/node_modules/npm
$
```

init  
install  
publish  
update

# Example: 09-thermometers

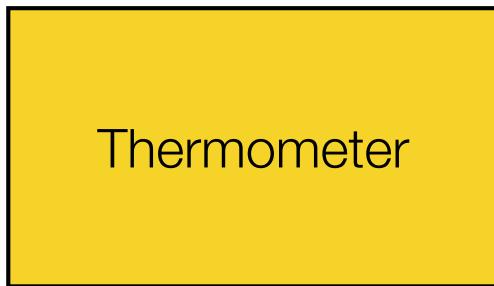


node thermometer.js bedroom 17 2

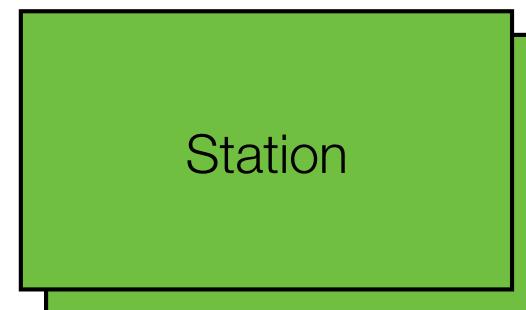
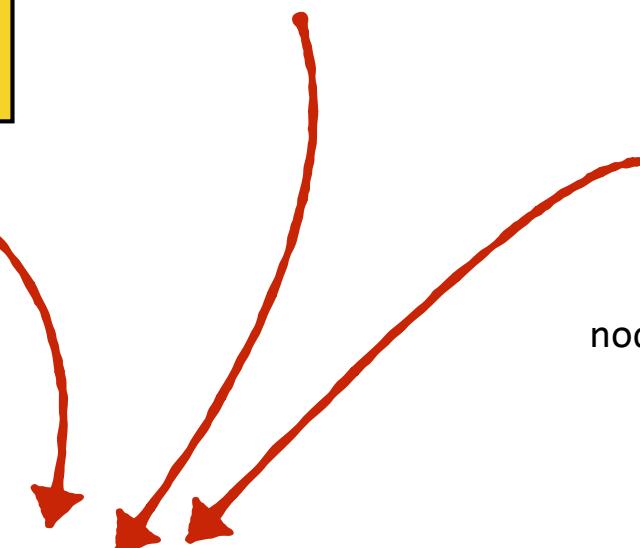


HAUTE ÉCOLE  
D'INGÉNIERIE ET DE GESTION  
DU CANTON DE VAUD  
[www.heig-vd.ch](http://www.heig-vd.ch)

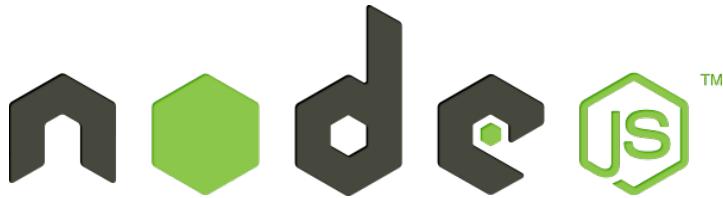
node thermometer.js office 20 3



**multicast**  
(239.255.22.5:9907)



node station.js



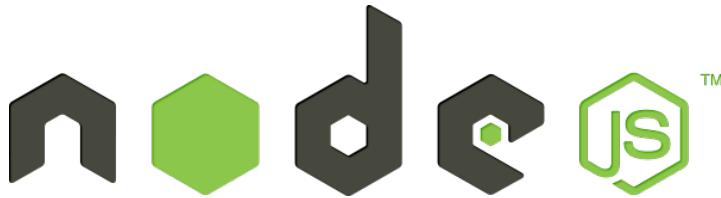
## thermometer.js (fragment)

```
// We use a standard Node.js module to work with UDP
const dgram = require('dgram');

// Let's create a datagram socket. We will use it to send our UDP datagrams
const s = dgram.createSocket('udp4');

// Create a measure object and serialize it to JSON
const measure = new Object();
measure.timestamp = Date.now();
measure.location = that.location;
measure.temperature = that.temperature;
const payload = JSON.stringify(measure);

// Send the payload via UDP (multicast)
message = new Buffer(payload);
s.send(message, 0, message.length, protocol.PROTOCOL_PORT, protocol.PROTOCOL_MULTICAST_ADDRESS,
function(err, bytes) {
  console.log("Sending payload: " + payload + " via port " + s.address().port);
});
```



## station.js (fragment)

```
// We use a standard Node.js module to work with UDP
const dgram = require('dgram');

// Let's create a datagram socket. We will use it to listen for datagrams published in the
// multicast group by thermometers and containing measures
const s = dgram.createSocket('udp4');
s.bind(protocol.PROTOCOL_PORT, function() {
  console.log("Joining multicast group");
  s.addMembership(protocol.PROTOCOL_MULTICAST_ADDRESS);
});

// This call back is invoked when a new datagram has arrived.
s.on('message', function(msg, source) {
  console.log("Data has arrived: " + msg + ". Source IP: " + source.address + ". Source
port: " + source.port);
});
```