Web Applications

Jose L. Muñoz-Tapia

Universitat Politècnica de Catalunya (UPC)

Introduction

Outline

Layered model

- Packet networks are complex.
- Different abstraction levels are defined.
- “Divide and conquer”.
- Each layer performs some functions and hides the implementation details to the other layers.
- Interesting features of layered models:
  - Divide problems into easy to use parts.
  - Provide flexibility (modular design).

A Toy Model i

Two philosophers, Alice at USA and Bob at England want to talk about philosophy using the telegraph service.

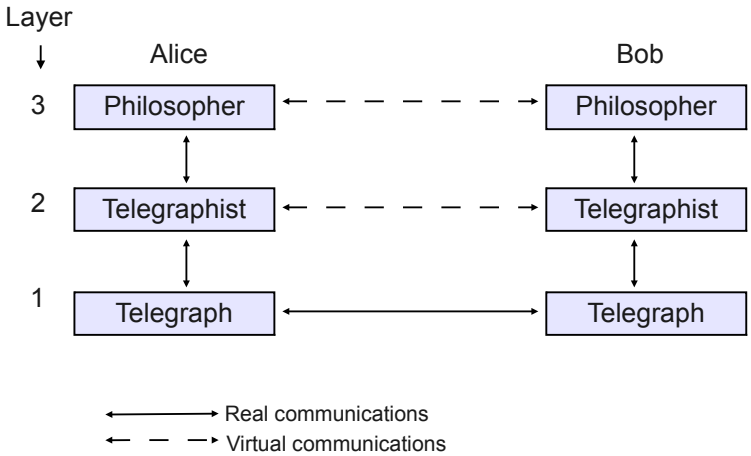
Monolithic design:

- Alice and Bob must know about philosophy and about how build, connect and use telegraphs.

Layered design with three ”layers“:

- **Philosophers**. Alice and Bob only know about philosophy.
- **Telegraphists**. There are telegraphists that how to use telegraphs.
- **Engineers**. There are engineers that build and connect telegraphs.

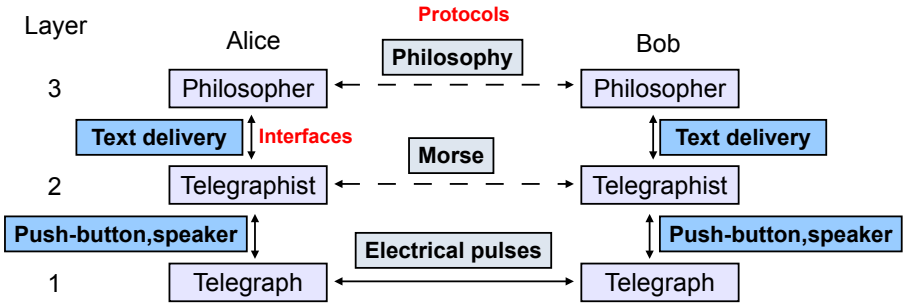
A Toy Model ii



A Toy Model iii

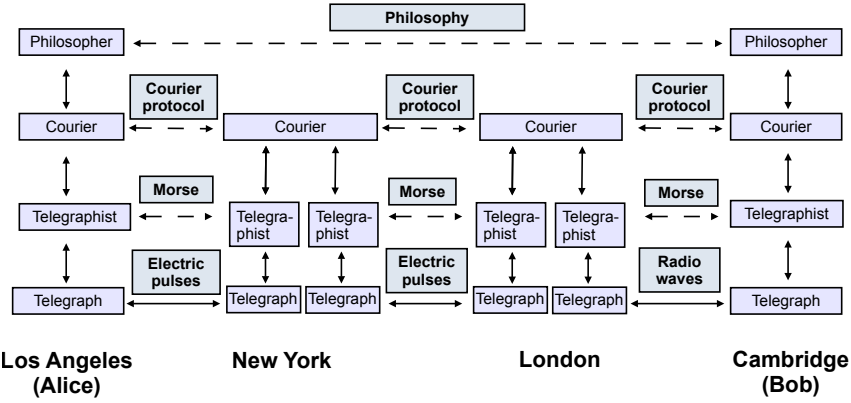
- Layered models are based on the following principles:
  - Layer n provides services **only** to layer n+1.
  - Layer n+1 is the **user** of the services of layer n.
- Interfaces are used to communicate layers up and down in the same stack.
- Protocols are used to communicate layers of the same level in different systems (n-layer protocol).
- The set of protocols is normally known as “protocol stack“.

A Toy Model iv

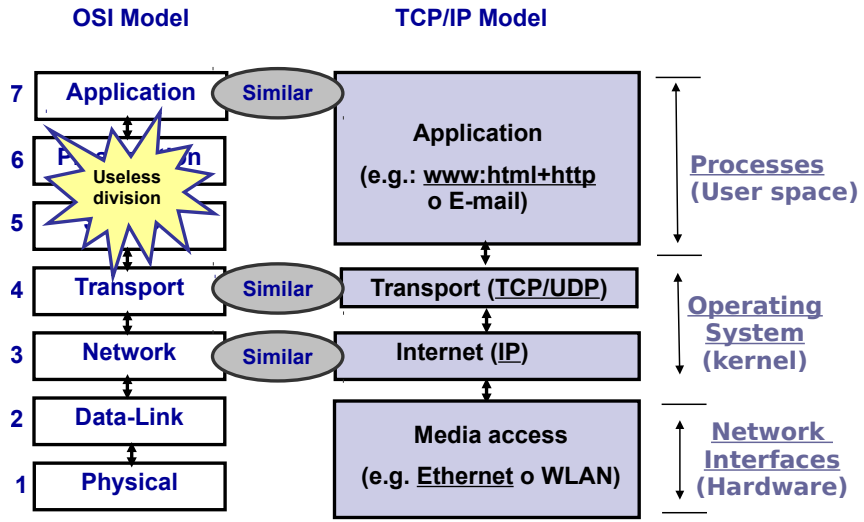


- Alice is at Los Angeles (USA) and Bob at Cambridge (UK) and they have not a direct telegraphic communication.
- We introduce a new layer, "courier companies", to manage some aspects of the communication.

A Toy Model v


Philosophy
Philosopher
Philosopher
Courier protocol
Courier protocol
Courier protocol
Courier
Courier
Courier
Courier
Morse
Morse
Morse
Telegraphist
Telegra-phist
Telegra-phist
Telegra-phist
Telegra-phist
Telegraphist
Electric pulses
Electric pulses
Radio waves
Telegraph
Telegraph
Telegraph
Telegraph
Telegraph
Telegraph
Los Angeles (Alice)
New York
London
Cambridge (Bob)

OSI vs. TCP/IP


OSI Model
TCP/IP Model
7
Application
Similar
6
5
Useless division
4
Transport
Similar
3
Network
Similar
2
Data-Link
1
Physical
Application
(e.g.: www:html+http o E-mail)
Transport (TCP/UDP)
Internet (IP)
Media access
(e.g. Ethernet o WLAN)
Processes (User space)
Operating System (kernel)
Network Interfaces (Hardware)

Interconnection Devices

On the three lower levels:

- L1. Repeaters and Hubs interconnect physical medium to transmit bit streams (physical layer signals). These devices also amplify or restore the signal.
- L2. Switches and bridges manage frames (data link layer PDUs[1]).
- L3. Routers manage network packets (network layer PDUs).

[1] PDU: Protocol Data Unit

Outline

LXC Concept

- LXD is a next generation system container manager.
- It offers a user experience similar to virtual machines but using Linux containers instead.
- Can take advantage of copy on write filesystems.
- LXD uses ZFS by default (but there are other storage options).
- In particular, LXD uses ZFS filesystems for:
  - Images.
  - Snapshots.
  - Containers.

Create a LXC Container

- Creating your first container is as simple as[2]:

```
$ lxc launch ubuntu:22.04 myubuntu
```

- Check that the container is created with:

```
$ lxc list
```

- Your container here is called ”myubuntu” but you also could let LXD give it a random name (by just executing without a name):

```
$ lxc launch ubuntu:22.04
```

[2] You can also try LXD containers online in `https://linuxcontainers.org/lxd/try-it`

LXC Basic Usage i

- Shell and commands:
  - We can get an interactive shell in a running container with:

```
$ lxc exec myubuntu bash
```

  - Or with the equivalent command line:

```
$ lxc exec myubuntu -- /bin/bash
```

  - The previous can be also used to execute any command:

```
$ lxc exec myubuntu -- apt-get update
```

  - To exec a command with another user:

```
$ lxc exec webserver -- sudo --login --user myuser ls -la
```

LXC Basic Usage ii

- Pull/push files:
  - To pull a file from the container:

```
$ lxc file pull myubuntu/etc/hosts .
```

  - To push a file to the container:

```
$ lxc file push hosts myubuntu/tmp/
```

- Stop/remove containers:
  - To stop the container:

```
$ lxc stop myubuntu
$ lxc info mycontainer --show-log
```

  - And to remove it entirely:

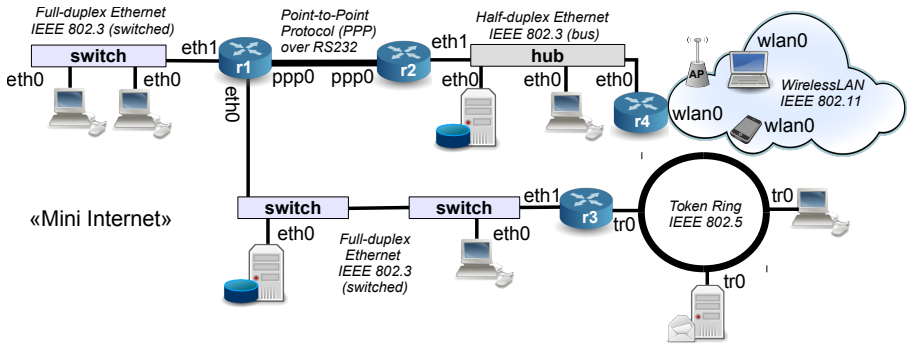
```
$ lxc delete myubuntu
```

Internet

Outline

Goal of Network Layer

Allow scalable communications between devices in different link technologies.



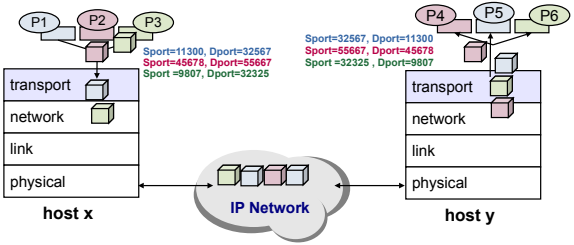
In the Internet we have only one network layer protocol: the Internet Protocol (IP) with two versions IPv4 (32-bit addresses) and IPv6 (128-bit addresses).

Goal of Transport Layer

Allow communications between processes (running applications) that are in general in different systems.

Transport communications are also called **end-to-end communications**.

Introduces the concept of **PORT** for multiplexing and demultiplexing.



Client/Server Toy Example i

. . .

Client/Server Toy Example ii

- Clients initiate the communication with servers.
- Clients must know the address of the server.

...

Internet Client/Server Model

- In the Internet:
  - Server processes or network daemons run continuously in background.
  - The address of a process is composed of three parameters: **IP address, transport protocol and port**.
  - The transport protocols (L4) provide multiplexing identifiers for their users, which are processes (running applications) on the system.
  - The transport multiplexing ID has a size of 16 bits and it is called **port**.
  - There are two transport protocols: TCP and UDP.

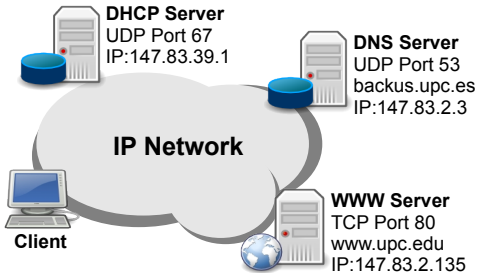
Internet Transport Protocols

- **User Datagram Protocol (UDP):**
  - UDP is the simplest transport protocol.
  - UDP is a message-oriented protocol or datagram protocol.
  - Each UDP datagram (message) is encapsulated in a IP datagram.
  - For its users, UDP only offers multiplexing and a checksum for discarding wrong data.
  - For multiplexing, UDP uses the source and destination ports.

- **Transport Control Protocol (TCP):**
  - TCP provides applications with a full-duplex communication, encapsulating its data over IP datagrams.
  - TCP communication is connection-oriented because there is a handshake of three messages before data can be sent.
  - The TCP communication is managed as a data flow (TCP is not message-oriented).
  - Apart from multiplexing capabilities, TCP is a reliable protocol.
  - TCP adds support to detect errors or lost data and to trigger retransmission until the data is correctly and completely received.

Well-known Ports

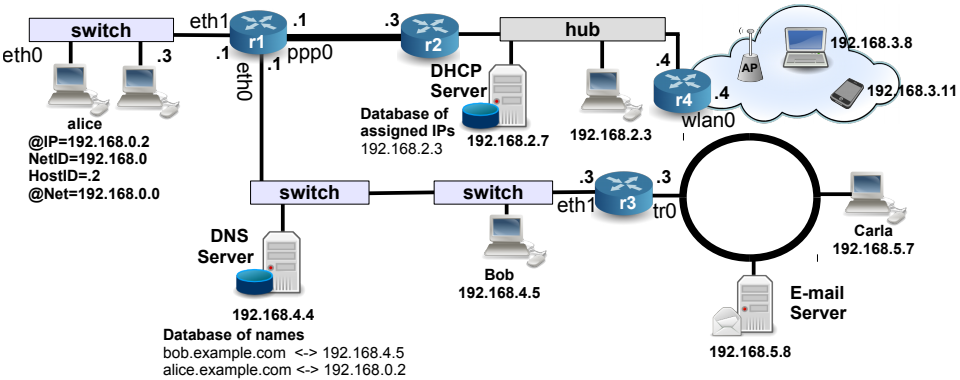
- The client needs to know IP, protocol and port to create a ”socket“ (communication) with a server.
- The client usually knows the IP (or name) of the server and there is a well-known transport protocol and port per service.



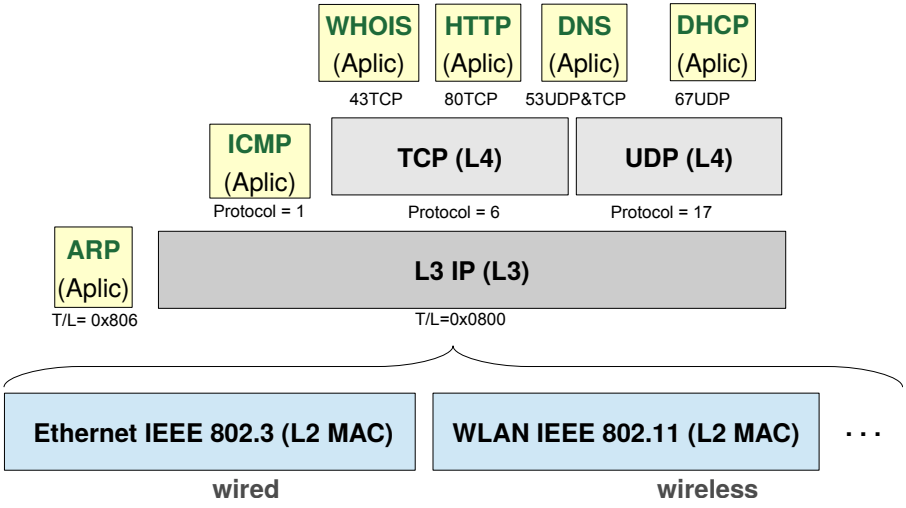
- For example, HTTP servers (for the Web) use TCP/80, DNS servers use UDP/53 for name queries and the DHCP servers use UDP/67.

- Example clients: `host` command (DNS), `firefox` (DNS and HTTP), etc.

- The DNS and DHCP services (protocols), and other services such as e-mail, Web, whois, etc. use the IP network with a client/server model.



Basic Protocol Stack of Internet



Outline

Basic **netcat** i

- The **netcat** application can be used to create a process that opens a raw TCP or UDP socket as client or server.
- It is very useful tool for testing networks (known as “Swiss Army Knife of networking“).
- Note. Make sure that your **netcat** command is the “traditional” one.
- **netcat** as client:

```
$ nc hostname port
```

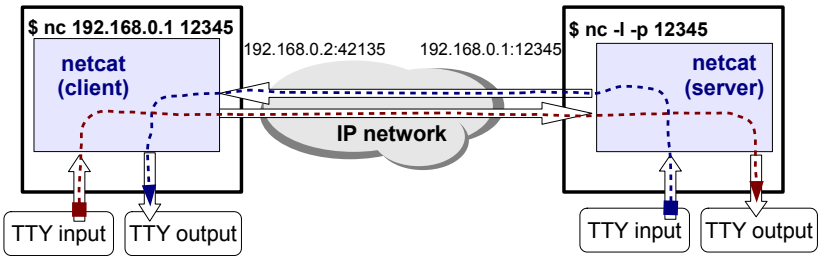
- We can use the -l (listening) option to make **netcat** work as a server:

```
$ nc -l -p port
```

- A server with **netcat is not multi-client**.

Basic **netcat** ii

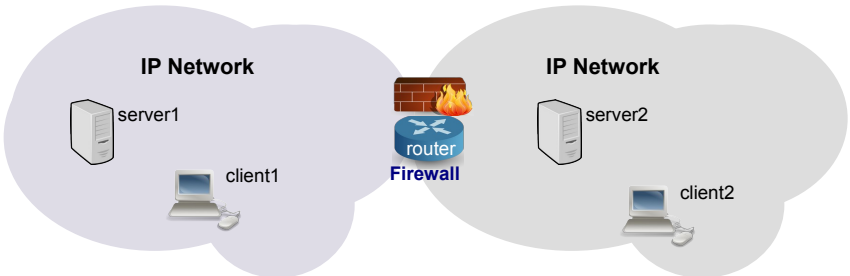
- Once a client is connected, the behavior of **netcat** until it dies (e.g. CRL+c) is as follows.



Outline

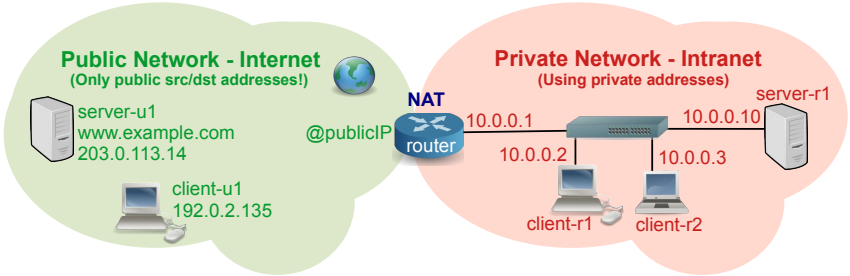
Firewalls

- Routers route packets (recall that they do not forward frames).
- But routers can do more things.
- A router with “firewall“ capabilities:
  - Can filter (do not forward) other traffic.
  - Firewalls take filtering decisions based on protocols and protocol parameters.
  - E.g. filter UDP port 53 (DNS service).



Network Address Translation (NAT)

- NAT (Network Address Translation) is typically performed by the router that connects a private network to the Internet.

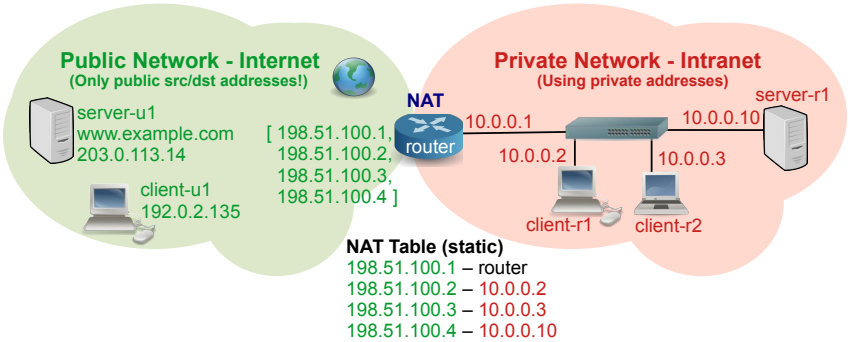


- The router that implements NAT must process the packets and replace private addresses by public addresses.
- There are several ways of implementing NAT.

Static NAT Concept

Static NAT

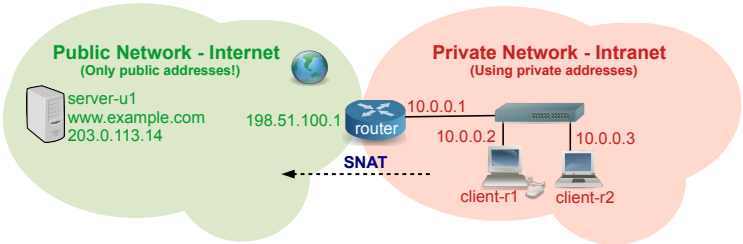
Static NAT is the simplest way of implementing NAT in which the router uses a public address per each private address that translates.



Dynamic SNAT Concept

Dynamic SNAT

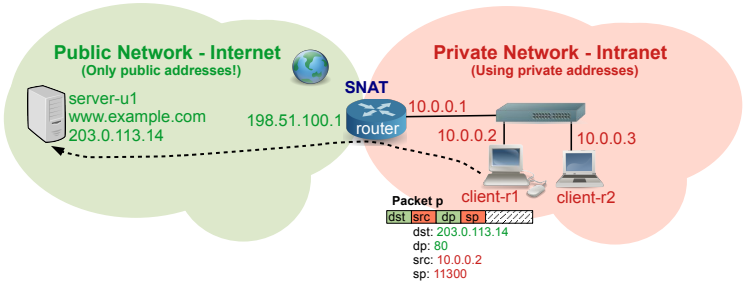
Dynamic SNAT (Source NAT) is a process in a router that allows communications initiated by clients in a private network towards servers in the public network.



SNAT allows communications `from` a **private network to** the **public network** using just **a single public IP address** in the NAT router for **all the communications** of the complete private network.

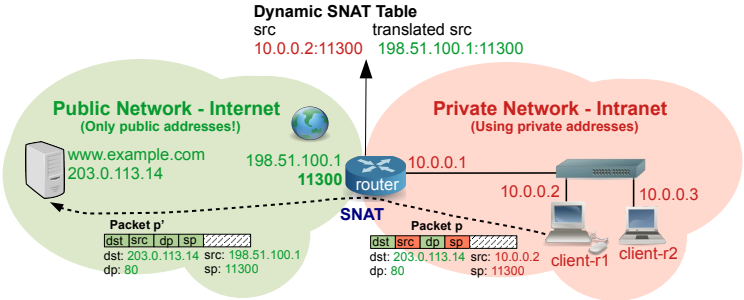
A client in the private network sends a packet to a server in the Internet.

In this example to the web server wwww.example.com.

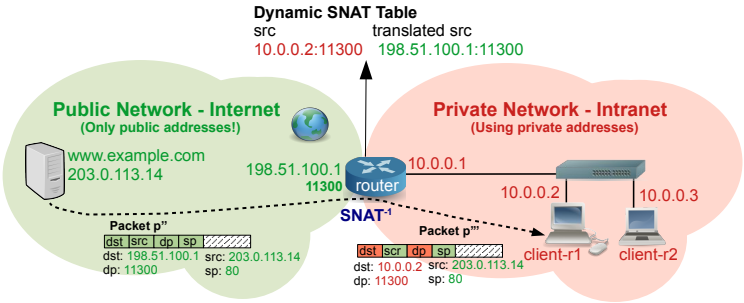


The router translates the source parameters before sending the packet to the Internet.

The combination IP/Port is used in the dynamic SNAT table.

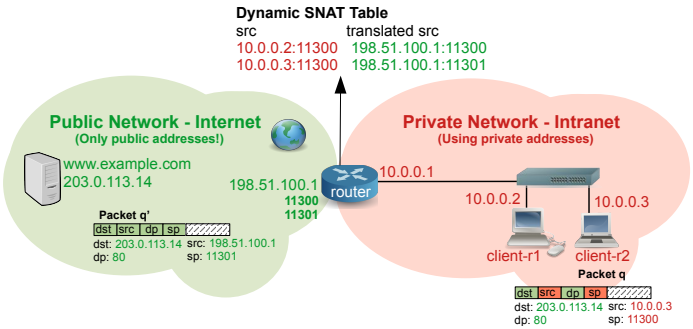


The NAT router inverts the SNAT manipulation (SNAT^{-1}) done when forwarding the packet to the internet.



Several SNAT Connections

A worse case is to have two simultaneous connections to the same server with the same source port on the clients:



Dynamic SNAT solves this problem taking another port in the router to differentiate the communications.

SNAT Features, Configuration and Problems

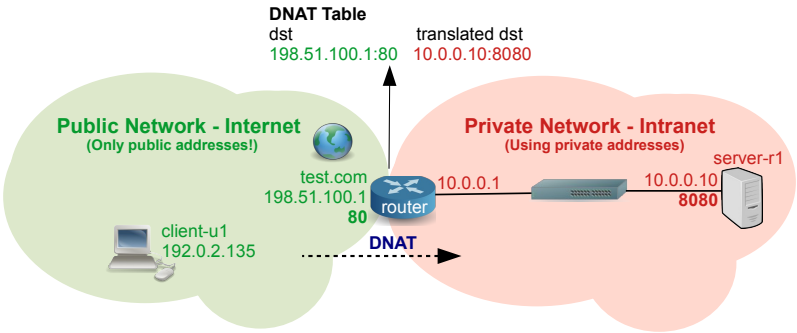
- In a Linux router, we can configure dynamic SNAT with:

```
# iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

- `MASQUERADE` means take the IP address of the output interface (`eth0` in the example).
- SNAT features:
  - Has enlarged the live of IPV4.
  - It can be thought like a ”redistribution” of bits from the port portion (16 bits) to the address (32 bits).
  - Many telecommunication operators event do several NATs before sending the packet to the Internet.
- SNAT problems:
  - NAT limits direct communications, we need support from a router.
  - Communications have to go through the same NAT router in the forward and backward path.

DNAT

DNAT is for communications from clients in the public network to servers in a private network.



DNAT Features, Configuration and Problems

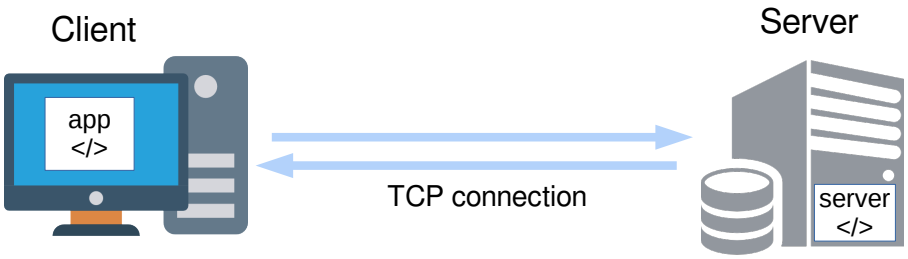
- In a Linux we can configure a DNAT rule as follows:

```
# iptables -t nat -A PREROUTING -p tcp --dport 5000 -j DNAT --to 192.168.0.2:12345
# iptables -t nat -nL
Chain PREROUTING (policy ACCEPT)
target prot opt source     destination
DNAT   tcp  --  0.0.0.0/0  0.0.0.0/0    tcp dpt:12345 to:192.168.0.1:23456
```

- DNAT issues:
  - DNAT is manual, you have to manually configure each translation.
  - You need to know "a priori" the ports you need for your service.
  - Notice that if you are using only one public IP address in the router, you can only have one private server per service.
  - In other words, there will be only one default port per service.
  - For example, if only using `198.51.100.1`, you can only have **one** HTTP server publicly available in the default port **80**.

Outline

Client/Server TCP Apps



The client and server are communicating using a **TCP connection**.

Once connected, we have a raw **bi-directional** stream of data.

Implementing **TCP** Services with JS

- A TCP service is the simplest form of a networking service we can start off.
- We will use the net builtin package:
  `https://nodejs.org/api/net.html`
- With this package we can build servers and clients.

TCP server with JS i

Creating a server:

```
const net = require("net");

const server = net.createServer();

server.on("connection", handleConnection);

server.listen(12345, "0.0.0.0", () => {
  const serverAddress = JSON.stringify(server.address());
  console.log(`server listening to ${serverAddress}`);
});

function handleConnection() {
}
```

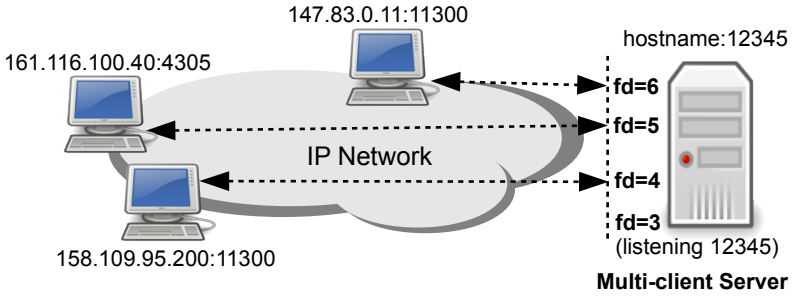
- `createServer()` give us a server object.
- We make it listen to port ”12345” and ”any” (0.0.0.0) IPv4 address.

TCP server with JS ii

Let's code handleConnection():

```
function handleConnection(socket) {
  const remoteAddress = `${socket.remoteAddress}:${socket.remotePort}`;
  console.log(`New client from ${remoteAddress}`);
  socket.on("data", onConnData);
  socket.on("close", onConnClose);
  socket.on("error", onConnError);

  function onConnData(data) {}
  function onConnClose() {}
  function onConnError(err) {}
}
```

TCP server with JS iii



The socket object can emit a **data event**, a **close event** and an **error event**

Let's add some logic to the handlers of these events:

```
function onConnData(data) {
  console.log(`Data from:${remoteAddress} Data:${data}`);
  socket.write(data);
}

function onConnClose() {
  console.log(`Closed connection from ${remoteAddress}`);
}

function onConnError(err) {
  console.error(`Error: ${err.message} in connection from ${remoteAddress}`);
}
```

Example: Capitalized **echo** server

```
function handleConnection(socket) {
  const remoteAddress = `${socket.remoteAddress}:${socket.remotePort}`;
  console.log(`New client from ${remoteAddress}`);

  socket.setEncoding("utf8");
  socket.on("data", onConnData);
  socket.on("close", onConnClose);
  socket.on("error", onConnError);

  function onConnData(data) {
    console.log(`Data from:${remoteAddress} Data:${data}`);
    socket.write(data.toUpperCase());
  }

  function onConnClose() {
    console.log(`Closed connection from ${remoteAddress}`);
  }

  function onConnError(err) {
    console.log(`Error: ${err.message} in connection from ${remoteAddress}`);
  }
}
```

- This service will expect data from the TCP connection coded with UTF-8.
- UTF-8 is a universal way of encoding characters.
- Then, the server will uppercase all the characters.

TCP JS Client

Example of a TCP client written for node.js.

```
1  const net = require('net');
2
3  const clientSocket = new net.Socket();
4  clientSocket.connect(12345, '127.0.0.1', function() {
5      console.log('Connected');
6      clientSocket.write('Hello, how are you?');
7  });
8
9  clientSocket.on('data', function(data) {
10     console.log(`Received: ${data}`);
11     clientSocket.destroy(); // kill client after server's response
12 });
13
14 clientSocket.on('close', function() {
15     console.log('Connection closed');
16 });
```

A Chat Server i

Creating a **chat Server**:

```
const net = require("net");
let clients = [];

const server = net.createServer();
server.on("connection", handleConnection);

server.listen(12345, "0.0.0.0", () => {
  const serverAddress = JSON.stringify(server.address());
  console.log(`server listening to ${serverAddress}`);
});
```

The clients array will store an array of sockets.

Each socket is a connection (i.e. a client)

`slice()` returns a **new array**

`splice()` **mutates** the array

A Chat Server ii

```
function handleConnection(socket) {

  // we get a new property of socket to identify the client
  socket.name = `${socket.remoteAddress}:${socket.remotePort}`;
  clients.push(socket);

  // Send a nice welcome message and announce
  socket.write(`Welcome ${socket.name}`+"\n");
  broadcast(`${socket.name} joined the chat`+"\n", socket);

  // Handle incoming messages from clients
  socket.on("data", data => {
    broadcast(`${socket.name}> ${data}`, socket);

  });

  // Remove the client from the list when it leaves
  socket.on("end", () => {
    clients.splice(clients.indexOf(socket), 1);
    broadcast(`${socket.name} left the chat`+"\n");
  });
```

```
  // Send a message to all clients
  function broadcast(message, sender) {

    // Log the message to the server output
    //process.stdout.write(message);
    console.log(message);

    clients.forEach(client => {
      // Don't want to send it to sender
      if (client === sender) return;
      client.write(message);
    });
  }
}
```

WWW

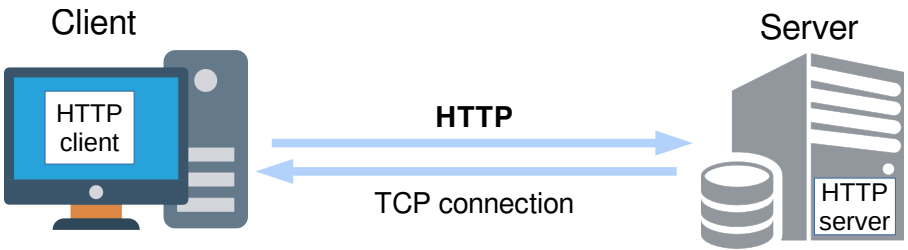
Outline

World Wide Web

WWW is a system to share resources over a network. **Web resources** may be any type of downloaded media such as documents, images, videos, ...

- Tim Berners-Lee is credited with having created the initial World Wide Web (WWW) during 1985-1991.
- Four basic technologies were part of his proposal:
  1. **HTML** (HyperText Markup Language): a language to describe graphical documents.
  2. **HTTP** (HyperText Transfer Protocol): a protocol to transmit resources.

3. **An HTTP (WEB) server**: a software that serves resources using HTTP.
4. **A WEB browser**: a software that acts as client to send requests and process responses for resources available on a WEB server.



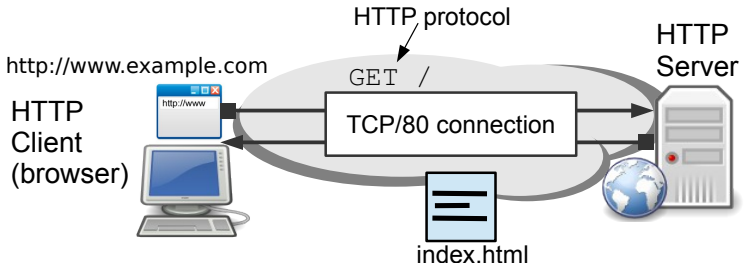
HTTP Motivation & Main Features

Motivation

HTTP was designed to cover the necessity of downloading resources and their linked resources from servers in the Internet.

1. HTTP is based on the Internet **client/server model**:
   - WEB browsers send HTTP (Hypertext Transfer Protocol) requests to HTTP servers asking for resources.
   - The server responds to each client with the requested resource.
2. It is a **request/response protocol** in which requests are **initiated by clients**.
3. It is a **stateless protocol**:
   - There is no state or context between different requests/responses.
   - The client always provides the necessary information to the server so the server can create the response.
4. It is a **textual** protocol (we will see it later).

HTTP is Client/Server



HTTP clients:

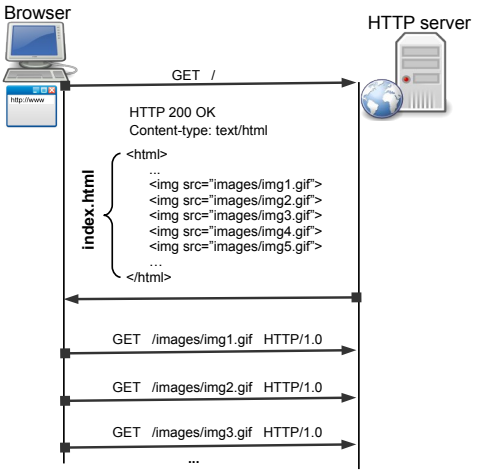
- Typically are WEB Browsers (e.g. Firefox, Explorer, Chrome, Lynx, etc.).
- However, there are other HTTP clients (e.g. curl, netcat!, ...).

HTTP servers:

- By default listen to the well-known TCP port 80.

Standard HTTP is Initiated by Clients

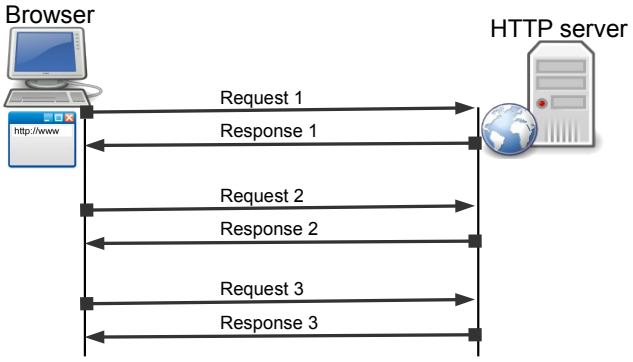
- In the standard HTTP flow, **the protocol is always initiated by clients**.
- This means that by default, **the server cannot send any message if the client has not requested this previously**.
- This design was initially proposed to avoid security issues.
- Now we have some alternatives to this behaviour (discussed later).



HTTP is Stateless

Stateless Protocol

There is no state or context between different requests/responses. The client always provides the necessary information to the server so that the server can create the response.



E.g. cannot use anything from Resquest/Response 1 in Resquest/Response 2.

Stateless Pros and Cons

- Disadvantages:
  - Requests are bigger that it we would use context.
  - We cannot relate the current request with a previous one of the same client (Cookies).
- Main advantage:
  - Request/responses can be **cached!**
  - Caching is very important for a massive service like the WWW.
  - Caching can be done at the browser or at intermediary network nodes.

Universal Resource Locator (URL)

- An URL is **an identifier that can be used to locate a representation of a resource on the Web**.
- The simplest syntax is: **protocol://host/directory/resource**
  - Example: **http://www.example.com/doc.html**
    The host (HTTP server) is www.example.com, the directory is / (DocumentRoot) and the resource is doc.html.
- The syntax of an URL can be more complex[3]:
  protocol://user:pwd@host:port/directory/resource
  - http://www.example.com
  - http://192.168.0.5
  - http://www.example.com/pictures/upc.jpg
  - http://www.example:8080/cgi-bin/time.sh
  - https://www.example.com
  - sftp://user@someserver.com/

[3] The detailed specification is in RFC 1738

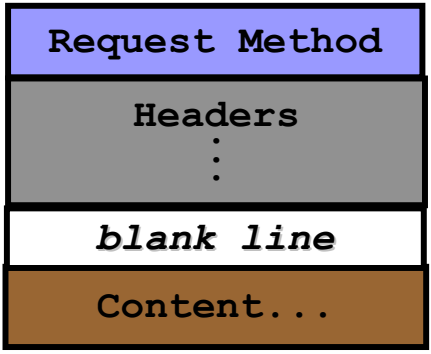
- In HTML documents we can include hyperlinks with absolute and relative paths on the same server and to other servers:

```
<html>
  <head>
  <title> Hello World</title>
  </head>
  <body>
  <p>Hello <b>World</b>!!!!!!!</p>
  <p>Go to <a href=docs/otherdoc.html> another document </a></p>
  <p>Visit UPC site at <a href="http://www.upc.edu">UPC</a></p>
  <img src="/images/upc1.gif">
  <img src="/images/upc2.gif">
  <img src="http://www.example.com/images/upc1.gif">
  </body>
</html>
```

- When using the absolute path, HTTP servers use the concept of **Document Root**.
- Hyperlinks provide the basic underlying mechanism to build the World Wide Web (WWW).
- WWW can be defined as **a big set of linked resources**.

HTTP Requests

- Requests are text.
- They contain the request method, headers, a blank line (CR+LF) and an optional message body.
- Now, we will describe HTTP v1.0 (v1.1 is described later).



HTTP 1.0 Requests

- The minimal HTTP request for version 1.0 is:

```
GET / HTTP/1.0
```

- GET means “give me this resource”.
- After the GET method we find a “/”.
- This means that the resource that we are requesting is the index file of the WEB server (file index.html).
- Another example is:

```
GET /images/upc1.gif HTTP/1.0
```

- This is requesting a file called upc1.gif in the directory images regarding the DocumentRoot of the server.

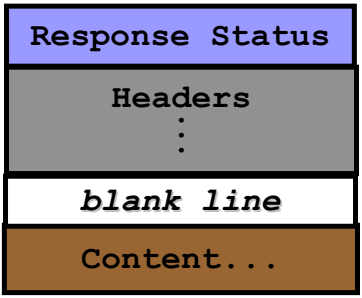
Headers

- Requests (and also responses) can have header lines.
- Headers are text lines that provide additional information or functionality.
- The format is ”Header-Name: value1, value2”, ending with CR+LF.
- HTTP 1.0 defines 16 headers, though none is required.
- Typical headers included in the requests are:
  - From: gives the email address of the user who makes the request.
  - User-Agent: name of the browser and OS.
- Example:

```
GET /path/file.html HTTP/1.0
From: user@example.net
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:24.0) Gecko/20100101 Firefox/24.0
```

HTTP Responses

- Responses are also text.
- They contain the status, headers, a blank line (CR+LF) and an optional message body.
- Typical status lines are:
  - `HTTP/1.0 200 OK`
  - `HTTP/1.0 404 Not Found`
- For the status, the first digit identifies the category:
  - `1xx` indicates an informational message only.
  - `2xx` indicates success of some kind.
  - `3xx` redirects the client to another URL.
  - `4xx` indicates an error in the client side.
  - `5xx` indicates an error in the server.



HTTP 1.0 Response Example

```
HTTP/1.0 200 OK
Date: Mon, 12 Oct 2020 22:29:59 GMT
Content-Type: text/html
Content-Length: 50

<html>
<body>
<h1>It works!</h1>
</body>
</html>
```

- The client closes the TCP socket after receiving the response.
- There are usually two headers to describe the body:
  - **Content-Type**: MIME-type of data, example **text/html** or **image/gif**.
  - **Content-Length**: number of bytes in the body.

Testing HTTP with **netcat**

- For example, we can use a netcat as server and make a request from a browser:

```
$ nc -l -C -p 80
```

- We can also use an HTTP server (like nginx, apache2 or live-server) and use netcat as client:

```
$ nc -C 10.1.1.1 80
GET /index.html HTTP/1.0
Accept: */*
```

- Notes:
  - The `-C` is to send CR+LF as line-ending.
  - There is a new version of HTTP (v2) that is binary.

HTTP Methods

The main HTTP methods are:

- **POST**: it is used to **Create** resources.
- **GET**: it is used to get or **Read** a resource.
- **PUT**: it is used to **Update**/replace a resource.
- **PATCH**: it is used to **Update**/modify a resource.
- **DELETE**: it is used to **Delete** a resource.

The previous verbs allow to do **CRUD** (Create, Read, Update and Delete) operations over resources.

There are few additional methods (less important):

- An example is **HEAD** that is used when we want a response without body (only status and headers which is useful to make tests).

HTTP (Until v2.0) is Textual

- Up to v1.1, HTTP is textual (HTTP 2.0 is binary):

```
GET:    3 bytes
POST:   4 bytes
PUT:    3 bytes
DELETE: 6 bytes
```

- With a binary protocol we would just need 2 bits to code the verbs:

```
00 GET
01 POST
10 PUT
11 DELETE
```

- There are quite a lot of textual application protocols:
  `FTP` (file transfer), `TELNET` (remote shell), `POP` (receive e-mails), `SMTP` (send e-mails), `IMAP` (visualize e-mails), ...
- The nice thing about text protocols is that you can test and debug them with textual tools (like `netcat`).

HTTP 1.1

- HTTP 1.1 was defined to face up new needs and overcome the shortcomings of HTTP 1.0.
- The most important improvements are:
  - **Caching.** The protocol provides headers to implement caching.
  - **Host header.** For multiple domains.
  - **Chunked encoding.** Pages are divided and sent in chunks (fragments).
  - **Persistent connections.** The connection is not opened/closed for each request.
- HTTP 1.1 requires changes in both client and server.

Caching Headers

- Responses can include a header date:

  `Date: Fri, 3 Apr 2020 09:59:59 GMT`

- In requests, two headers can be included in HTTP 1.1:
  - **If-Modified-Since** means “send the response if it has changed since that date“.
  - **If-Unmodified-Since** means ”send the response if it has not changed since that date“.
- In addition to a date, an **ETag** can be used for caching:
  - It is an opaque identifier.
  - Assigned by a server as a version (e.g. resource hash).
- Caching is performed by:
  1. Browsers.
  2. Intermediate network devices: HTTP cache servers.

Host Header

- HTTP 1.1 allows a server to be enabled as multi-site.
- For example, ”www.example.com“ and ”www.example.net” may be on the same server.
- The server might have only one IP address.
- **How we differentiate requests for each site?**
- Each HTTP 1.1 request must have a **host** header that specifies a name and optionally a port.
- Example:

```
GET / HTTP/1.1
Host: www.example.com:80
```

Local DNS Resolution

- Next, we create a local resolution for different domain names:
  - Local DNS resolutions can be configured in Unix systems in **/etc/hosts**.
  - Edit **/etc/hosts** (need to be root) to add the following name resolutions:

```
127.0.0.1    localhost
127.0.0.1    www.test.com
127.0.0.1    www.example.com
...
```

  - Notice that we translate www.test.com and www.example.com to the localhost address 127.0.0.1.
- Most web servers:
  - Allow to set different configurations for the different domain names.
  - Some names for these domain names are ”virtual hosts”, ”server blocks”, ...

Nginx Configuration Files

- Nginx logically divides the configurations meant to serve different content into **server blocks**.
- Server blocks can be saved in different files.
- The default configuration file is `/etc/nginx/conf.d`
- However, many installations use a two-folder approach[4]:
  - The files of available server blocks are in: `/etc/nginx/sites-available`.
  - The activated sites are in: `/etc/nginx/sites-available`.
- Sites are activated creating a symbolic link:

```
/etc/nginx/sites-enabled# ln -s ../sites-available/site-to-enable
```

- Conf files are read by `nginx` sorted in alphabetical order.

[4] This approach is original from apache.

Different Sites (Server Blocks)

- /etc/nginx/sites-available/001-www.example.com

```
server {
   server_name www.example.com;
   root /var/www/www.example.com;
}
```

- /etc/nginx/sites-available/002-www.test.net

```
server {
   server_name www.test.net;
   root /var/www/www.test.net;
}
```

- /etc/nginx/sites-available/999-default

```
server {
  server_name _;
  root  /var/www/default;
}
```

- The last block is a default catch all server block (optional).
- Now we can create content sites, active them and reload **nginx**!

Outline

Static Servers (Python/Flask) i

- Apart from apache and nginx, we can use servers based on many libraries of your favourite programming languages.
- Next, we use Python and the flask library to create a static HTTP server.
- First, we install the venv Python module to generate an environment, in which, we create our project:

```
$ sudo apt update && sudo apt install python3-venv
```

- Now, we create our project and activate the environment:

```
$ mkdir myproj
$ cd myproj
myproj$ python3 -m venv myenv
myproj$ source myenv/bin/activate
```

- In the terminal, you should see (env) indicating that you are working in an environment.

Static Servers (Python/Flask) ii

- Now, we can install flask inside the environment for our project:

```
myproj$ python -m pip install Flask
```

- Finally, we create the file web.py:

```python
from flask import Flask

app = Flask(__name__, static_url_path='', static_folder='public')

if __name__ == "__main__":
    app.run(host="localhost", port=3000)
```

- Create a file `test.html` in public and run the server:

```
myproj$ python3 web.py
```

- To exit the environment:

```
myproj$ exit
```

But today, the World Wide Web **is not just a database of static resources accessible via HTTP**.

The WWW is a platform to build applications, **web applications** (Web 2.0).

Server-based Web Applications

A natural way of creating dynamic web applications is to relay on the server:

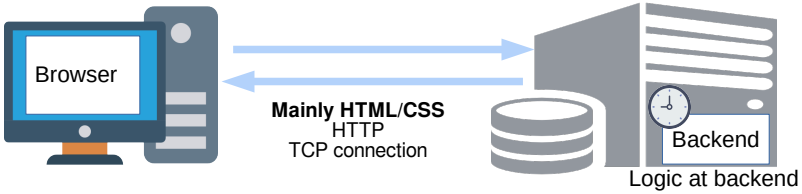
- In this case, the server executes code to dynamically generate the resources that form the application.
- The server and the logic it executes is called **the backend**.
- In fact, other systems to which the server connects like databases are also considered part of the backend.

Server-based Web Technologies

We have several technologies to create server-based web applications:

1. Common Gateway Interface (CGIs).
2. Scripting virtual machines like PHP.
3. Java servlets with an application server like Tomcat (or spring).
4. We can also run our self-made server in almost any other language:
   - Python (**flask**, django).
   - Javascript (**express**, koa).
   - Golang, Rust, C, C++, ...

Backend Logic




```python
from flask import Flask
from datetime import datetime
app = Flask(__name__)

@app.route('/bdate.html')
def backend_date():
    current_date = datetime.now()
    return f"""<html> <body>
      <h1>Backend Logic</h1>
      <h2>Current date in the backend server is {current_date}</h2>
      </body> </html>"""

if __name__ == "__main__":
    app.run(host="localhost", port=3000)
```

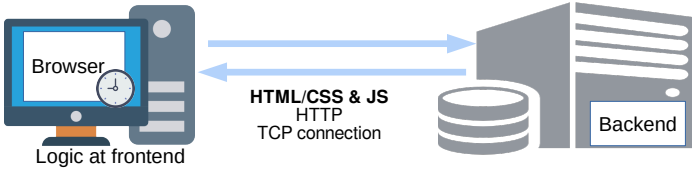
Client-based Dynamic Web

- Another option to build web applications is to **execute logic at the client**:
  - **Browsers have been able to execute natively only Javascript**[5].
  - Now, for heavy computations, we have another option called Webassembly (WASM)[6].
- The client (browser) and the logic/assets used by the client is what we call **the frontend**.
- Assets include images, browser storage, cookies, etc.

[5] With plugins, browser could also run other code like Java bytecode (applets) but this has not been very popular.

[6] WASM is an efficient virtual machine and we can compile code from languages like Rust, Golang, C++ or even Javascript to WASM.

Frontend Logic




```python
from flask import Flask
app = Flask(__name__)

@app.route('/fdate.html')
def frontend_date():
    return """<html> <body>
        <h1> Frontend Logic</h1>
        <h2> Current date in the frontend is <span id="currentDate">  </span></h2>
        <script>
            var date = new Date();
            var currentDateDomObj = document.getElementById("currentDate");
            currentDateDomObj.innerHTML = date;
        </script>
      </body> </html>"""

if __name__ == "__main__":
    app.run(host="localhost", port=3000)
```

Notice that resources with client logic can be **statically served**, e.g. in a Content Delivery Network (CDN).

Express Static Web Server

- Express (`http://expressjs.com`) is a small Javascript library that contains tooling for building HTTP servers.
- We are going ot use `Express.js` to serve static resources with a Javascript server (like with `nginx` or `apache`):

```
mywebapp$ npm install express
```

```
// index.js
const express = require('express');
const app = express();

app.use(express.static(__dirname + '/public'));
app.listen(3000);
```

- You can create a static resource (e.g. `test.html`) and access it:

  `http://localhost:3000/test.html`

Backend and Frontend (Express)

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

app.get("/bdate.html", (request, response) => {
  const date = new Date();
  response.send(
    `<html>
      <body>
        <h1>Backend Logic</h1>
        <h2>Current date in the backend server is ${date}</h2>
      </body>
    </html>` );
});

app.get("/fdate.html", (request, response) => {
  response.send(
    `<html> <body>
        <h1> Frontend Logic</h1>
        <h2> Current date in the frontend is <span id="currentDate">  </span></h2>
        <script>
            var date = new Date();
            var currentDateDomObj = document.getElementById("currentDate");
            currentDateDomObj.innerHTML = date;
        </script>
      </body> </html>`);
});

app.listen(3000, () => console.log("Express server running on port 3000"));
```

Backend-based Tasks App with Express i

The endpoint / is used to send the app to the frontend:

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

const tasks = [];

app.get("/", (request, response) => {
  response.send(
    `<html>
        <body>
          <h1> Current Task List </h1>
          <form action ="/process-task" method="get">
            <label for="task">Write a task:</label> <input type="text" name="task" />
            <label for="submit">Submit your task:</label> <input type="submit" name="submit" />
          </form>
        </body>
    </html>` );
});
```

Backend-based Tasks App with Express ii

The endpoint /process-task is used to receive new tasks from the form:

```
app.get("/process-task", (request, response) => {
  tasks.push(request.query.task);
  let tasksHTML = "";
  for (let task of tasks) {
    tasksHTML = tasksHTML + "<li>" + task + "</li>";
  }
 response.send(
  `<html>
    <body>
      <h1> Current Task List </h1>
      <ul> ${tasksHTML} </ul>
      <form action ="/process-task" method="get">
        <label for="task">Write a task:</label> <input type="text" name="task" />
        <label for="submit">Submit your task:</label> <input type="submit" name="submit" />
      </form>
    </body>
  </html>` );
});

app.listen(3000, () => console.log("Express server running on port 3000"));
```

Parsing URL-encoded Bodies in POST

- We can also use POST messages in HTML forms.
- Then, when we send a URL encoded POST body to the `/` route with the body `name=Mary&age=10`, we obtain a JSON object like the following:

```
{"name":"Mary","age":"10"}
```

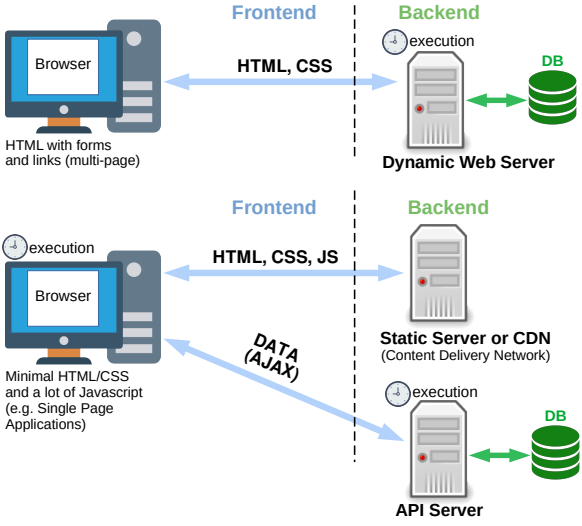
- We can use the `urlencoded` middleware to manage this JSON object in the body:

```
const express = require('express');
const app = express();

app.use(express.urlencoded({extended: true}));

app.post('/', (req, res) => {
  res.send(req.body);
});

app.listen(3000);
```

Frontend-based Tasks App with Express

```
const express = require('express');
const app = express();

app.get("/", (request, response) => {
  response.send(
    `<html> <body>
       <h1> Current Task List </h1>
       <ul id="list">  </ul>
       <form>
         <label for="task">Write a task:</label> <input type="text" name="task" id="task" />
         <label for="submit">Submit your task:</label> <input type="submit" name="submit" onclick="addTask(event)" />
       </form>
       <script>
         function addTask(event){
           event.preventDefault();
           const taskString = document.getElementById("task").value;
           const elem = document.createElement('li');
           elem.textContent= taskString;
           document.getElementById("list").appendChild(elem);
         }
       </script>
     </body> </html>`
  );
});
app.listen(3000, () => console.log("Express server running on port 3000"));
```

Frontend-based vs. Backend-based Web Apps



AJAX Requests

AJAX

Asynchronous JavaScript And XML (AJAX) allows a client script make HTTP requests to servers without the need for a page reload.

- AJAX is **initiated by the client logic** and allows getting data (not complete pages).
- In this case, the server acts as a data server or **API server**.
- Nowadays XML is not widely used and other formats like JSON are more common.

API Server: Express

```
const express = require('express');
const app = express();

app.use(express.json());

const tasks = [];

app.get("/tasks", (request, response) => {
  response.send({tasks:tasks});
});

app.post("/task", (request, response) => {
  tasks.push(request.body.task);
  console.log(tasks);
  console.log(request.headers);
  response.send({ result: "task added" });
});

app.listen(12345, () => console.log("api server started on port 12345"));
```

```
$ curl -X POST --header "Content-Type: application/json" --data '{"task":"study fullstack"}' http://localhost:12345/task
$ curl -X GET http://localhost:12345/tasks
```

API Server: Adding CORS

- If the API server and the App server are in different hosts, we need to configure CORS (we will see CORS in detail later):

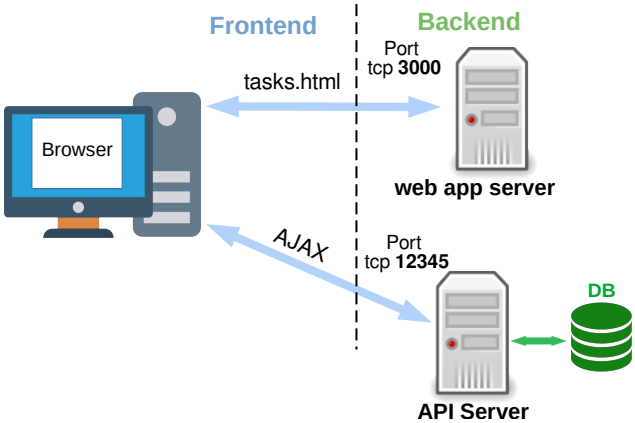
```
$ npm i cors
```

```
const express = require('express');
const app = express();
var cors = require('cors');
app.use(express.json());
app.use(cors()); // Accept everything

const tasks = [];

app.get("/tasks", (request, response) => {
  response.send({tasks:tasks});
});

app.post("/task", (request, response) => {
  tasks.push(request.body.task);
  console.log(tasks);
  console.log(request.headers);
  response.send({ result: "task added" });
});

app.listen(12345, () => console.log("api server started on port 12345"));
```

Our Example with the Static Server

Our static server for `tasks.html`:

```
1  const express = require('express');
2  const app = express();
3
4  app.use(express.static(__dirname + '/public'));
5
6  app.listen(3000, () => {
7    console.log("webapp server started on port 3000");
8    }
9  );
```



Frontend: **tasks.html**

```
<html>
  <body>
      <form>
        <h1> Current Task List </h1>
        <ul id="list"> </ul>
        <label for="task">Write a task:</label>
        <input type="text" name="task" id="task" />
        <label for="submit">Submit your task:</label>
        <input type="submit" name="submit" onclick="addTask(event)" />
      </form>

      <script>
        function addTask(event){
          event.preventDefault();
          const taskString = document.getElementById("task").value;

         // Send the task to the API server using an ajaX Http Request (xhr)
          const xhr = new XMLHttpRequest();
          xhr.open('post', 'http://localhost:12345/task');
          xhr.setRequestHeader('Content-Type', 'application/json');
          xhr.send(JSON.stringify({task:taskString}));
        }
      </script>
  </body>
</html>
```

tasks.html with **axios**

```
<html> <body>
  <form>
    <h1> Current Task List </h1>
    <ul id="list"> </ul>
    <label for="task">Write a task:</label>
    <input type="text" name="task" id="task" />
    <label for="submit">Submit your task:</label>
    <input type="submit" name="submit" onclick="addTask(event)" />
  </form>

  <script src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
  <script>
    function addTask(event) {
        event.preventDefault();
        const taskString = document.getElementById("task").value;
        axios.post('http://localhost:12345/task', {task:taskString}); // Send the task to the API server with axios
    }
  </script>
</body> </html>
```

https://github.com/axios/axios

Bundlers: **Webpack**

Bundlers (webpack, vite, rollup, gulp, ...):

- Are node.js applications that are use to create a bundle.
- A bundle is a single file (`e.g. main.js`) with all the Javascript code.
- Uglify the code.
- Transcompile.
- With ESM (ECMA script modules with import) they can do tree shaking (include only used functions from libs).
- Configure source maps (for debug, show source code not the bundle).
- Some (like Webpack with **Webpack-dev-server**) provide nice development servers that follows changes (hot reload).

Our Source File

```
myapp$ npm init -y
myapp$ npm i axios
```

Edit myapp/src/index.js:

```
1  import axios from 'axios';
2
3  function addTask(event){
4    event.preventDefault();
5    const taskString = document.getElementById("task").value;
6    axios.post('http://localhost:12345/task', { task : taskString });
7  }
8
9  window.addTask = addTask;
```

Setting the App with Webpack

- Install webpack in the project:

```
myapp$ npm i webpack webpack-cli --save-dev
```

- Modify the scripts section of **package.json** to create the bundle:

```
"scripts": {
  "build": "webpack --entry ./src/index.js --output-path ./public --mode production"
}
```

- Create the bundle:

```
myapp$ npm run build
```

Using our Bundle

- Use the bundle in `myapp/public/tasks.html`:

```
<html>
  <body>
    <form>
      <h1> Current Task List </h1>
      <ul id="list"> </ul>
      <label for="task">Write a task:</label>
      <input type="text" name="task" id="task" />
      <label for="submit">Submit your task:</label>
      <input type="submit" name="submit" onclick="addTask(event)" />
    </form>
    <script src="main.js"></script>
  </body>
</html>
```

- Notice that now we **are not depending on external CDNs** and our code is optimal.

Polling the API Server

Now, we want to periodically get the updated tasks list:

```
import axios from 'axios';

function addTask(event) {
    event.preventDefault();
    const taskString = document.getElementById("task").value;
    axios.post('http://localhost:12345/task', {task:taskString});
}

window.addTask = addTask;

async function polling() {
    const response = await axios.get('http://localhost:12345/tasks');
    let tasksHTML = "";
    for (let task of response.data.tasks) {
        tasksHTML = tasksHTML + "<li>" + task + "</li>";
    }
    document.getElementById("list").innerHTML = tasksHTML;
}

setInterval(polling, 2000); // poll the server for tasks every 2 seconds
```

Note. The **async/await** keywords will be explained later.