# Network programming & application protocols

Olivier Liechti

Published with GitBook

# Table of Contents


| | |
|---|---|
| Introduction | 0 |
| Intro to Java IOs | 1 |
| TCP Programming | 2 |
| UDP Programming | 3 |
| HTTP | 4 |
| Web Infrastructures | 5 |
| LDAP | 6 |

# Network programming, application-level protocols and web infrastructures

These are the lecture notes for the RES course at the University of Applied Sciences Western Switzerland (HEIG-VD), in the bachelor program.

The content of this book will be published in an iterative fashion during the semester.

# Introduction to Java IO


- Objectives
- Lecture
  - 1. A Universal API
  - 2. Sources, Sinks and Streams
    - 2.1. Reading Data From a Source
    - 2.2. Writing Data To a Sink
    - 2.3. Design Your Code to Be Universal
  - 3. A Simple Example: The File Duplicator
  - 4. Binary- vs Character-Oriented IOs
  - 5. The Mighty Filter Classes
  - 6. Performance and Buffering
  - 7. Shit Happens: Dealing with IO Exceptions
- Resources
  - MUST read
  - Additional resources
- What Should I Know For The Test and The Exam?


# Objectives

The goal of this lecture is to give an **overview of IO programming in Java** and to see how classes in the `java.io` package can be used to read and write files.

Why does it matter and why do we study that in this course?

- The first reason is simply that network programming ***is*** IO programming. A network program, whether it is a server or client, spends a lot of time reading and writing bytes. These bytes are encapsulated in IP packets and travel across the network. But as we will see later in the course, reading bytes from the network or reading bytes from a file is not really different. So, what we will learn in the first lecture will be very helpful when we start writing applications that use TCP or UDP via the Socket API.
- The second reason is that many network programs need to interact with the local file system. Have you ever thought about what happens when your web browser fetches a static html page? Well, the server on the other side opens a file, reads its content and sends it back to your browser. That shows that the web server is processing IOs in two ways (exchanging data with the file system and exchanging data with your browser). Again, what we will study in this first lecture will be useful later, for instance when we decide to implement our own HTTP server.

More specifically, here are the objectives of the lecture:

- Understand the concepts of **data sources**, **sinks** and **streams** and how they provide a foundation for standard IO classes.
- Understand that these concepts are applied in the same way, whether you are reading data from files, from the network or from other processes running on your machine (what we later refer to as the **universal API**).
- Understand the difference between **processing binary data** (bytes) and **processing text data** (characters). You should be able to decide when to use an `InputStream` or a `Reader`, and explain why. Be able to explain how **character encodings** work and how to deal with them in Java.
- Understand that it is possible to create **filter chains**, by wrapping `Writers` into `Writers` into `Writers` into `Writers` into `Writers`, with custom logic added at each level.
- Understand that the `BufferedReader`, `BufferedWriter`, `BufferedInputStream` and `BufferedOutputStream` classes use this mechanism. You should be able to explain how buffered IOs work and why they are important from a **performance** point of view.

**Note**: in this course, we only consider the **standard Java IO API**. For specific needs and in particular when writing scalable network servers, another API is available. This API has been named the New IO API, generally referred to as the NIO API.

# Lecture

## 1. A Universal API

In any programming language, dealing with IOs means **dealing with an exchange of data**. This can mean different things, for example:

- you want to read configuration data from a **file**
- you want to write the result of a computation into a **file**
- you implement some kind of **server** and want to read data sent by **clients** over the **network**
- you implement some kind of **server** and want to send data back to the **clients**
- you want to write data, byte by byte, in a temporary **memory** zone
- you want to read data from this **memory** zone
- you have code executing on two **threads**; you want the code executing on the first thread to produce messages that are consumed by the code executing on the second thread.

Instead of having a different API, in other words different abstractions, classes and methods, for each of these situations, you can use the Java IO as a generic, universal API to solve all your data transfer needs.

> At the end of the day, whether you are "talking" to a file, to a network endpoint or to a process does not matter. You are always doing the same thing: reading and/or writing bytes or characters. The Java IO API is the toolbox that you need for that purpose.

## 2. Sources, Sinks and Streams

If you think about it, every situation where you need to deal with IOs can be described with 3 abstractions:

- The first one is the concept of **source**, which is "something" that generates a sequence of bytes or characters. Again, it can be a file on the local file sytem, a client connected over the network or a process running on the local machine.
- The second one is the concept of **sink**, which "something" that can consume a sequence of bytes or characters. As before, it can be a file, a network endpoint or a process.
- The third one is the concept of **stream**, which you can think of as **a pipe that connects your program to either a source or a sink**. In the first case, it will be an input stream (that you can **read from**), in the second case, it will be an output stream (that you can **write to**). Note that the sources and the sinks are not connected directly.

> The beauty of the Java IO API, and what makes it universal, is that once you have a stream, you always use it in the same way. Whether it is connected to a file source, a network source or a memoy source, you are using the same class and have access to the same methods. Similarly, if your stream is connected to a file sink, to a network sink or to a memory sink, you always write bytes to the stream in the same way.

## 2.1. Reading Data From a Source

Concretely, when your program wants to read data from a source, it will:

1. **Get access to a source.** This will depend on the type of source you are using, but for instance you may use the `File` class to access a file in the local file system or the `Socket` class to access a network endpoint.
2. **Open an input stream**, that will connect your program to the source. Typically, you will do that by either creating a new instance of a class like `FileInputStream` or by getting an existing instance by calling a method like `socket.getInputStream()`.
3. **Use the various `read()` methods from the `InputStream` class**, typically until the end of the stream is reached. The end of the stream is reached when the source has no more data to send (e.g. the entire file has been read) or when connection with the source has been lost (e.g. the network client has disconnected).
4. **Close the stream**, by using the `close()` method defined in the `InputStream` class.

## 2.2. Writing Data To a Sink

Similarly, when your program wants to write data from a source, it will:

1. **Get access to a sink.** This will depend on the type of source you are using, but for instance you may use the `File` class to access a file in the local file system or the `Socket` class to access a network endpoint.
2. **Open an output stream**, that will connect your program to the sink. Typically, you will do that by either creating a new instance of a class like `FileOutputStream` or by getting an existing instance by calling a method like `socket.getOutputStream()`.
3. **Use the various `write()` methods from the `OutputStream` class**, until you don't have any more data to send.
4. **Close the stream**, by using the `close()` method defined in the `OutputStream` class.

## 2.3. Design Your Code to Be Universal

When you design your own classes and methods, make sure that you keep the spirit of the Universal API. As a rule of thumb, **pass streams and not sources as method parameters**. Compare the following two methods:

```
/**
 * This interface will work only for data sources on the file system. In the
 * method implementation, I would need to create a FileInputStream from f and
 * read bytes from it.
 */
 public interface IPoorlyDesignedService {
  public void readAndProcessBinaryDataFromFile(File f);
}
```

```
/**
 * This interface is much better. The client using the service has a bit more
 * responsibility (and work). It is up to the client to select a data source
 * (which can still be a file, but can be something else). The method implementation
 * will ignore where it is reading bytes from. Nice for reuse, nice for testing.
 */
public interface INicelyDesignedService {
  public void readAndProcessBinaryData(InputStream is);
}
```

## 3. A Simple Example: The File Duplicator

To illustrate these basic concepts, let us consider a very simple example. The code below shows that we have implemented a class named `FileDuplicator`. Its responsibility should be easy to guess from his name. It provides a method that, when invoked by a client, copies the content of a file into another file.

Let us look at some elements of the code:

- We see that we use 4 classes from the `java.io` package. That is where you will find all standard interfaces and classes for doing simple and sophisticated tasks around IOs. **Spend some time to browse through the javadoc reference** (http://docs.oracle.com/javase/7/docs/api/java/io/package-summary.html). Being aware that some classes exist will prevent you from reinventing the wheel!
- The `File` class is not used directly for reading and writing data. It is used to do file system operations, such as checking whether a file with a specific name exists or not. In this example, it would not be stricly required (as an alternative, we could pass the file name directly to the `FileInputStream` and `FileOutputStream` constructors).
- The program is using two different streams, one connected to a source and the other ocnnected to a sink. The `FileInputStream` and `FileOutputStream` classes are used to create these streams.

- Once the streams have been opened, the logic is very simple. We use a loop to consume all bytes, one by one, from the input stream. Each time that we read a byte, we write it immediately to the output stream. We can see that the `read()` method returns an int. This value is -1 if the end of the stream has been reached. Otherwise, it has a value between 0 and 255 (we are reading a single byte).
- Note that **this code is not very efficient and that copying large files would be painfully slow**. We will see later that is is much better to read/write blocks of bytes in a single read operation, or to use buffered streams.

```
package ch.heigvd.res.samples.io;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

/**
 *
 * @author Olivier Liechti
 */
public class FileDuplicator {

    public void duplicate(String inputFileName, String outputFileName) throws IOException

        // This is my data source
        File inputFile = new File(inputFileName);

        // This is my data sink
        File outputFile = new File(outputFileName);

        // These are 2 streams, 1 for reading and 1 for writing bytes
        FileInputStream fis = new FileInputStream(inputFile);
        FileOutputStream fos = new FileOutputStream(outputFile);

        int b;
        // I read all bytes from the input stream in a loop
        while ( (b = fis.read()) != -1 ) {
            // I write each of the read bytes to the output stream
            fos.write(b);
        }

        // Important: when dealing with IOs, always close and clean-up resources
        fis.close();
        fos.close();
    }

}
```

## 4. Binary- vs Character-Oriented IOs

If you browse through the java.io package, you will notice two parallel class hierarchies:

- On one hand, you will see a set of classes extending the `InputStream` and `OutputStream` abstract classes. These classes are used to read and write **binary data**. In other words, if you are writing an application that deals with images or sounds, then you will be happy to use these classes that will **process raw data**, **without doing any conversion**.
- On the other hand, you will see a set of classes extending the `Reader` and `Writer` abstract classes. These are used to read and write characters. In other words, if you are writing an application that deals with text data, then you will be happy to use these classes that will perform **conversions between raw data (bytes) and characters (in a particular encoding)**.

What does that mean? If you think about it, computers do not have a notion of character. They know about bytes, in other words about sequences of 8 bits. In order to manage text data, we have to agree on a particular **character encoding system**. The encoding system defines a correspondance between, on one hand, bit patterns and, on the other hand, characters. ASCII is a well-known character encoding system, which originally used 7 bits to represent characters. Here are a few examples for how bit patterns are mapped to characters in ASCII:

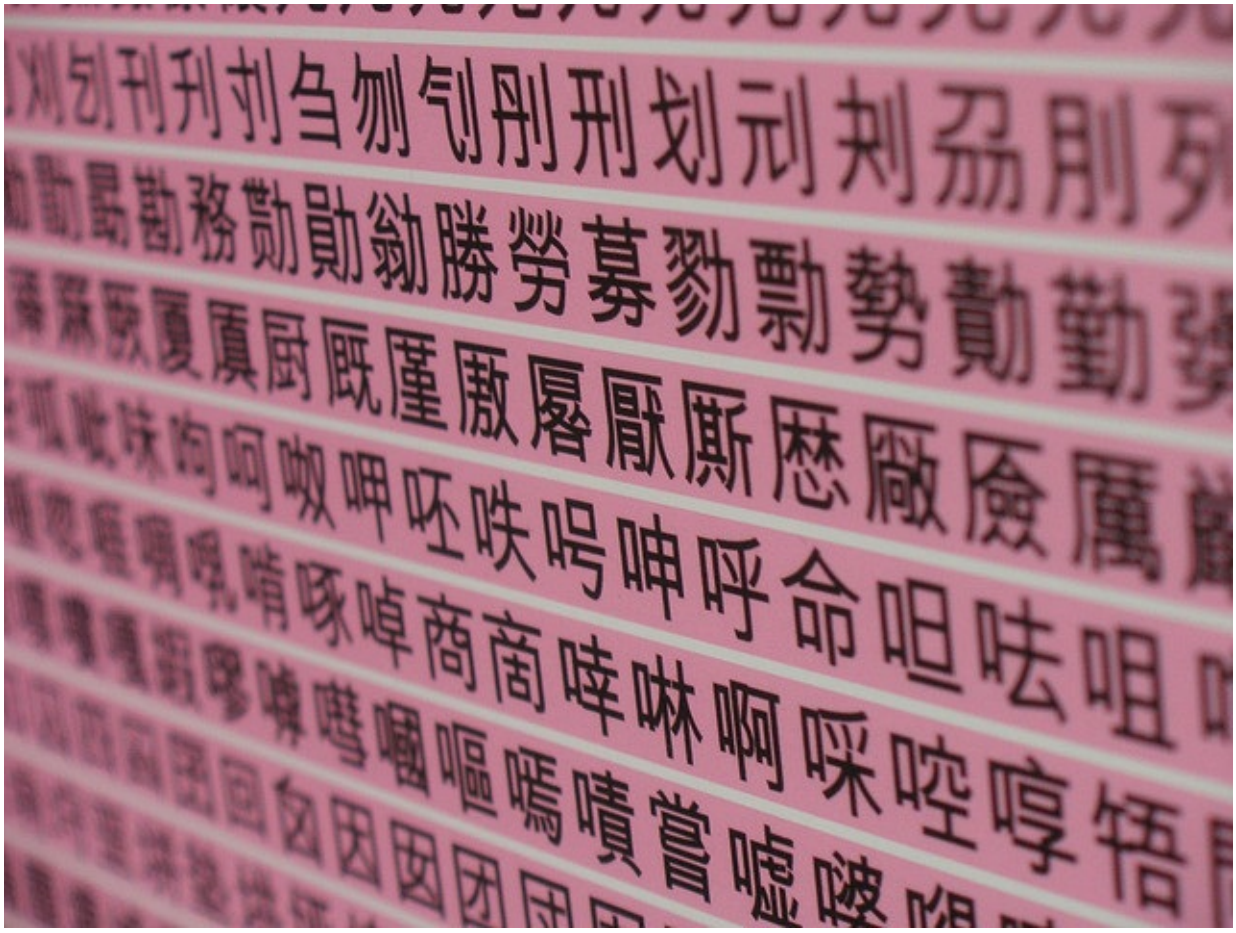
```
Binary value     Decimal value     Character
011 0110         54                '6'
100 0001         65                'A'
100 0010         66                'B'
…
110 0001         97                'a'
110 0010         98                'b'
```

**ASCII** worked well for many years, but there are many **languages with alphabets much larger than the latin alphabet**. For these languages, having only 7 bits (128 values) to represent characters is simply not enough. This is why several other character encoding systems have been developed over time. This has introduced quite a bit of complexity, especially when conversion from one encoding system to another is required.

In order to deal with internationalization, Java decided to use the **Unicode** standard to handle characters. When a Java program manipulates a character in memory, it uses **two bytes**. These two bytes are used to store what Unicode calls a **code point** (1'114'112 code points are defined in the range 0 to 10FFFF). A code point is a numeric value, which is often represented as `U+xxxxxx`, where `xxxxxx` is an hexadecimal value. What is useful (but also a bit confusing), is that the code points used to identify ASCII characters are the values defined in the ASCII encoding system. Huh? Take the character 'B' for instance. In ASCII, it is encoded with the decimal value 66. In Unicode, it has been decided that the code point `U+0042` (yes, 42 is the hexademical value of 66) would be used to identify the character 'B'.

Unicode is actually not a character encoding system. When you have a code point, you still need to decide how you are going to encode it as a series of bits. Sure, you could use 16 bits (4 bytes for each of the 4 hexadecimal values making up the code point) for each encoded character. But in general, that would be a waste. Think of a text written in english, with only latin characters. Since the code points of all characters are below 255,

Sounds complicated? Well, it is a bit. Have a look at this page. It will show you how the same character is represented in Unicode and in different encoding systems.



## 5. The Mighty Filter Classes

If you browse the `java.io` package, you will encounter 4 interesting classes: `FilterInputStream`, `FilterOutputStream`, `FilterReader` and `FilterWriter`. When you think about these classes, think about the **Decorator design pattern**. Think about **matriochkas** (poupées russes).

The role of these classes is to allow you to add behavior to a stream, in other words to do some processing on the bytes or characters being read or written. To illustrate this idea, let us consider an example:

- Let's imagine that **you hate the letter 'u'**. Each time that you work with a Writer, whether it is to write characters to a file or to send characters to a network server, you would like every occurrence of this letter to be **automatically removed from the stream**.
- While the logic is fairly simple (it is only a character comparison condition), **you don't want to repeat it over and over**. Also, you might one day realize that you love the letter 'u' but now hate the letter 'm'. That day, you don't want to have to go through all your codebase to reflect your change of heart and would like to **do the change in a single place**.
- The FilterWriter class is what you need to solve your problem. Here is what you would use it:
  - Firstly, you would create a class named `CensorshipWriter` that extends `FilterWriter`.
  - Secondly, you would override the various `write()` methods implemented by the `FilterWriter` class. This is where you would get rid of the the hated characters, before calling the `write()` method in the `super` class.

# 6. Performance and Buffering


In the Java IO, 4 classes use this filtering mechanism:

- For binary data, `BufferedInputStream` and `BufferedOutputStream` extend `FilterInputStream`, respectively `FilterOutputStream`.
- For character data, `BufferedReader` and `BufferedWriter` extend `FilterReader` and `FilterWriter`.

Why should you use these classes and what are they doing? Think about what is happening if you write a method that **consumes a binary stream connected to a file, byte by byte**. When you call the `read()` method, you are not really sure what is happening at a low level (because that is up to the JRE implementation, to the operating system, to the disk driver, etc.). To exagerate the situation, imagine that every single call the method would actually traverse all the layers and result in a hard disk operation. You realize that it is a lot of work, just to read a character. You also realize that you will be wasting a lot of time, because **you will have to pay the overhead for every read operation**.

To improve the situation, a common technique is to use **buffering** (and this can be done at each layer). Simply stated, when you use buffering when reading data, you apply the following principle:

> "If I am asked to read 1 character, then I will read 10. I will return the first one and keep the 9 remaining for later. If I am asked once more to read 1 character, then I will already have it and be able to respond quickly, without any effort."

**As an analogy, think about what you typically do when you drink beer**:

- If you are not organized, then every time that you feel thirsty, you will need to get dressed, walk to the shop, buy a can of beer, walk back home, sit down and enjoy your beer. Thirsty again? No problem, you will just have to spend another 40 minutes to go through the same process.
- On the other hand, if you are a bit more organized, when you go to the store the first time, you will directly buy a pack of 24 cans. Going back home, you will store 23 cans in your basement and enjoy the 24th one. Thirsty again? Easy, and now it will only take you 10 minutes to visit your basement and get a new can.
- If you are really very well organized, then you will do beer-buffering at different levels. Each time you need to visit your basement, you will not only get 1 can. You will directly get 5 and store 4 of them in your fridge. Thirsty again? Cool, you are now down from 10 minutes to 2 minutes before you can enjoy your drink.

That is pretty much what the `BufferedXXX` classes are doing. They manage an internal buffer for you (you don't really see or interact with this buffer), which means that even if you write a loop that reads data byte by byte, then each byte is read from the buffer and not from the original source. In order to use a BufferedInputStream, you wrap one around an existing source:

```
public void processInputStream(InputStream is) {

  // I don't know to which source "is" is connected. It is also possible that is is alrea
  // a chain of several filters wrapping each other. I don't really care, what I want is
  // make sure that I read bytes in an efficient way.

  // So, here I decorate "is" by wrapping a BufferedInputStream around it
  BufferedInputStream bis = new BufferedInputStream(is);

  // When I make this call, then "bis" will read a bunch of bytes, send me one and keep t
  // others in his buffer
  int b = bis.read();

  // When I make this call, I will get my byte faster because it will come straight from
  // buffer managed by "bis"
  b = bis.read();
}
```

Using a `BufferedOutputStream` or a `BufferedWriter` to send data towards a sink follows the same logic. **There is however one more thing to be aware of**. Since the bytes or characters that you produce transit via a buffer, there will be a delay until they are actually pushed towards the sink. Sometimes, you will want to influence this delay and to push content ***right now***. That is something that you can do with the `flush()` method defined in the classes.

## 7. Shit Happens: Dealing with IO Exceptions

When dealing with IOs, you will be interacting with external systems and components and you will quickly realize that **the environment is unreliable and that many things can go wrong**. Think about reading a corrupted file, think about a faulty hard drive. Think about loosing your network connection or seeing a remote client suddently cut the connection while you are reading the data it is sending you.

**You have to detect and react to these error conditions**. The Java IO API gives you a mandate to do so: many methods in the API declare that they might throw exceptions that extend the `IOException` class ( `FileNotFoundException` , `SocketException` , `ZipException` , `FileSystemException` , `CharConversionException` are some of the many subclasses).

One important thing that you will have to do, is to **close the streams when you are done reading or writing data**. Since errors can happen while you are working with the stream, make sure that you close the streams in the `finally` clause. Check the **additional resources** list for two interesting links on that topic.

# Resources

## MUST read

- The Basic I/O Lesson on the Java Tutorial: http://docs.oracle.com/javase/tutorial/essential/io/index.html. Read and learn the section dedicated to IO streams, and browse through the section dedicated to file IO.
- A nice, brief introduction to Unicode: http://wiki.secondlife.com/wiki/Unicode_In_5_Minutes
- A interactive site for querying and browsing Unicode characters and getting their representations in various encodings (UTF-8, UTF-16, etc.) : http://www.fileformat.info/info/unicode
- A description of the Decorator design pattern: http://en.wikipedia.org/wiki/Decorator_pattern

## Additional resources

- Reference for the Java NIO API: http://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html
- A tutorial for the Java NIO API: http://tutorials.jenkov.com/java-nio/index.html
- Two articles about IO Exception handling and related design guidelines: http://tutorials.jenkov.com/java-io/io-exception-handling.html and http://tutorials.jenkov.com/java-exception-handling/exception-handling-templates.html.
- A list of supported character encodings for Java SE 7: http://docs.oracle.com/javase/7/docs/technotes/guides/intl/encoding.doc.html

# What Should I Know For The Test and The Exam?

Here is a **non-exhausive list of questions** that you can expect in the written tests and exams:

- Why can we say that the Java IO API is a universal API?
- Explain the notions of source, sink and stream in context of IO processing.
- What is the difference between an `InputStream` and a `Reader` ?
- What is the role of the `FilterInputStream` class?

- Write a program that reads the content of a file A, converts all characters to uppercase and writes the result into a file B. Design a solution that makes it possible to apply this logic to a stream connected to any kind of source and sink (e.g. reads input from a network socket and writes output to a file, or vice-versa)
- Write a class that allows you to read characters from a file, apply two different transformations (e.g. transform to uppercase, transform to lowercase) and write the resulting output into two files. You don't want to read the file twice, but only once. You don't know in advance what the two transformations will be. They will be implemented by two classes that extend the `FilterWriter` class.
- Why is it better to read blocks of bytes in a single operation, instead of reading a single byte multiple times?
- Write the code that is used to consume all bytes from a source connected to an `InputStream`.

# TCP Programming


- Objectives
- Lecture
  - 1. Client-Server Programming
  - 2. The TCP Protocol
  - 3. The Socket API
  - 4. So… What Do Servers and Clients Do With the Socket API?
  - 5. Handling Concurrency in TCP Servers
- Resources
  - MUST read
  - Additional resources
- What Should I Know For The Test and The Exam?


# Objectives

The goal of this lecture is to **introduce key concepts of network programming** and to explain how developers can write **client and server programs** that use the **TCP protocol** to communicate with each other. The goal is to describe the **Socket API**, which is standard way to do it, available **across operating systems and programming languages**.

After this lecture, you should be able to **write a multi-threaded TCP server in Java**, listening for connection requests. You should be able to **write a TCP client** that initiates a connection with this client. Finally, using your knowledge of IO processing, you should be able to **exchange a stream of bytes between the client and the server**. This will provide a foundation, on top of which you will be able to implement application-level protocols (such as HTTP) in your own programs.

# Lecture

## 1. Client-Server Programming

You are certainly already familiar with the notions of **network clients** and **servers**, at least intuitively. When you use your web browser to read the daily news, you use an HTTP client that talks to an HTTP server. When you send an e-mail, you use a SMTP client that talks to a SMTP server. When you use an online music streaming service, you use a client that implements a custom protocol to talk to a server in the cloud.

While all of these applications are different, they have some things in common:

- The **server is a program** that implements some functionality (we can say that it offers some kind of **service**). The server appears to be ***always on***, **waiting for clients** to contact him and to ask him to do something.
- The **client is a program** that ***you*** **use** to benefit from this functionality. Depending on what you do in the user interface, the client contacts the server and asks him to do something on your behalf (that is why clients are sometimes called **user agents**).
- The client and the server need a way to **exchange messages** (requests, replies, notifications, etc.). The **types of messages**, the **structure of messages**, **what needs to be done** when specific messages are received are all things that are specified in application-level protocols (in other words, HTTP, SMTP and the music stream service protocol have their own messages and rules)
- In order to read and write messages, the clients and servers **need a way to read and write sequences of bytes**. *Very often*, they need to be able to do that in a **reliable way** (with guarantees that all the bytes written on one side will be received on the other, in the right order). This reliability is ensured by TCP.
- The client and the server **ignore the implementation details** of the interlocutor on the other side. In particular, the client often does not know on which operating system the server runs. For many protocols, it does not know either in which programming language it as been written.

When we talk about **client-server programming**, we talk about **using a programming language** to implement some sort of **communication protocol**, whether it is a standard one (e.g. HTTP) or our own proprietary one. **Modern programming languages and libraries make this surprisingly easy**. Simple clients and servers can literally be written in a couple of lines. Learning how to write these lines is very easy if you have some experience working with IOs (which you should, having survived the previous lecture!)

## 2. The TCP Protocol

In this course, **we assume that you are familiar with the TCP/IP protocol stack**. We assume that you know how IP is used to exchange packets between hosts in interconnected networks and how TCP adds multiplexing/demultiplexing and reliability on top of IP. We also assume that you know how TCP defines what needs to be done in order to establish and terminate a connection between a **caller** and a **callee**.

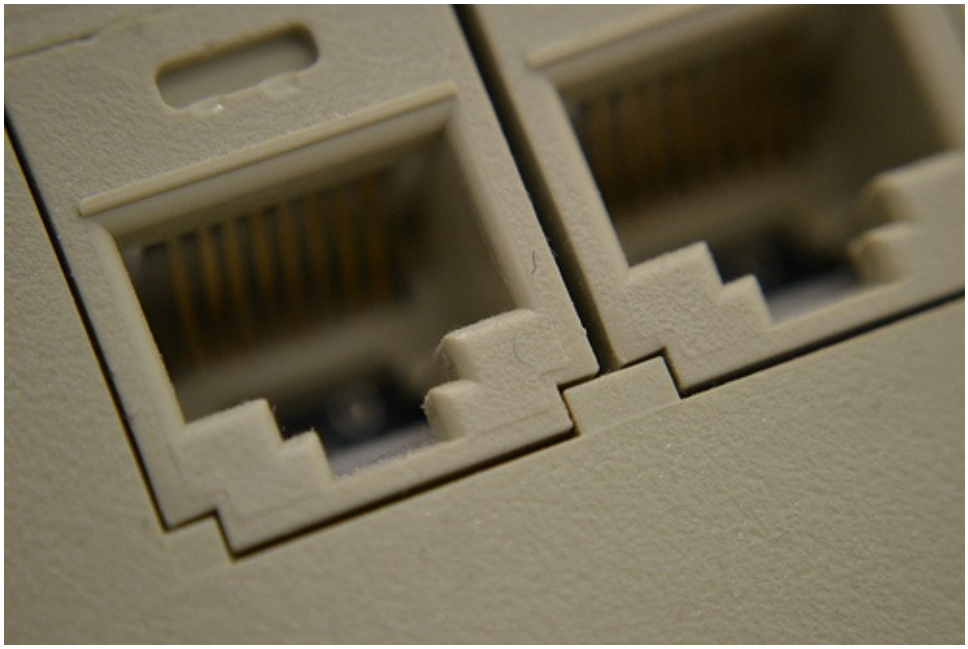

Here are some of the key points that you certainly remember:

- **By analogy, TCP can be described as a kind of telephone system**. Imagine that you are running a taxi company (you are offering a taxi booking service). You have published your company phone number, so customers know about it. The receptionist is patiently waiting for customers to call. When a client needs a ride, he makes a phone call. If the receptionist is available, he picks up the phone. A connection has now been established and the two persons can exchange messages, until someone hangs up. The TCP protocol handles all these steps: call setup, conversation, call termination.
- Whereas **IP is a host-to-host communication protocol**, **TCP is an application-to-application communication protocol** (in other words, TCP is used for a process running on a host to communicate with another process running on another host). Since multiple processes may run on the same host and may wish to use TCP, the notion of **port** is used for **multiplexing/demultiplexing** TCP segments. With TCP, it is an application (as opposed to the operating system) that decides to use the protocol, when and how.
- For a **server application**, that means **using a function** to **bind** (a *socket*, more on this later) to an available port and another function to **listen for connection requests**. For a **client application**, that means **using a function** to **initiate a connection** with a server, identified by an IP address and a port number. For both the client and the server, once the connection has been established, that also means **using functions** to send and

receive bytes to the interlocutor. That sounds good, but what does it actually mean to ***use these all of these functions***? In what interface, in what API are these functions defined?

Well, that is precisely what we want to explain in this lecture! As we are just about to see, it is the **Socket API** that provides all the enlisted functions and supporting data structures, in a **standardized way**.

## 3. The Socket API

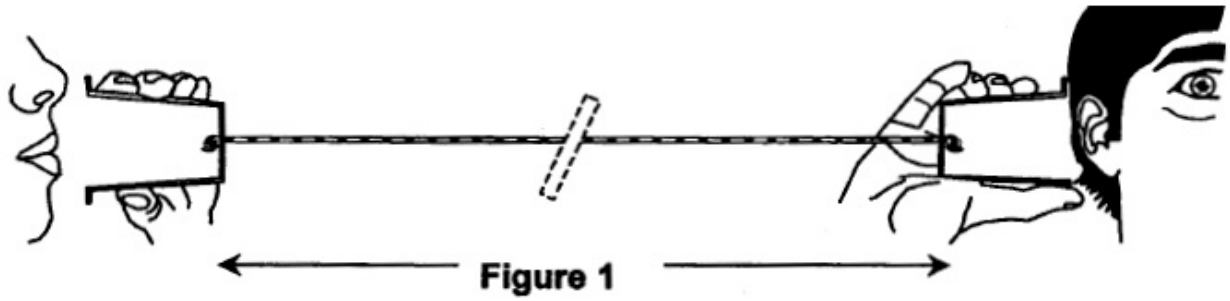


The **Socket API** is a standard programming interface which has its origins in the **Unix** operating system, but which is now available across operating systems and programming languages. To simplify things a bit, the API **provides a list of functions and data structures** to use TCP (and as we will see in the next lecture, to use UDP) from application-level code. In particular, the API allows us to:

- **once connection has been established, communicate with a remote application through a pair of *sockets***. Imagine a **virtual pipe** that would connect a client and a server. The client and the server each see **one extremity of the pipe**. Each extremity is called a socket and is used to read bytes from and send bytes to the other side of the pipe. In technical terms, **a socket is a data structure identified by 4 values**: a local IP address, a local port, a remote IP address and a remote port.
- **Bind a socket to an 'address'** (where an address is actually the combination of an IP address and of a port). This is something typically done when a server starts up and plans to listen for incoming connection requests, on a given network interface and on a given port (e.g. on *192.168.1.12:8080*). Binding a socket to an address means that the **local address** and **local port** fields of the socket are now set.

- Start **listening**, i.e. accepting connection requests, on a given port. This is done by setting a bound socket to **listening mode**. Once this is done, **the operating system will forward him the TCP segments** that have destination port matching the local port of the socket. Whenever a **connection request is made by a client**, a **new socket** is created (so, *we now have 2 sockets*: the one used to listen for connection requests and one for talking with the client). The values for the remote address and remote port of this socket are set to the client's IP address and local port (which most of the time has been randomly assigned by the remote operating system).
- **Initiate a connection request** with a remote process. This is done by a client that wants to contact and interact with a server.
- **Read and write bytes through a socket**. This can be done either by providing **socket-specific functions**, such as `send()` and `receive()` (e.g. in C), or by working with **more generic functions** that work on file descriptors (e.g. `read()` and `write()` in C) or IO streams (e.g. in Java).
- **Close a socket**, when the connection with the remote host (client or server) can be terminated.
- Finally, the Socket API also provides **utility methods** to work with **network interfaces**, **IP addresses** and to perform **data conversions**.

## 4. So… What Do Servers and Clients Do With the Socket API?




The following pseudo code shows the the typical sequence of operations performed by a server and a client in order to communicate. Let us start by looking at the server (note that steps 1 and 2 can be combined in a single function call in some languages):

```
What the server does…
1. Create a "receptionist" socket
2. Bind the socket to an IP address / port
3. Loop
   3.1. Accept an incoming connection (block until a client arrives)
   3.2. Receive a new socket when a client has arrived
   3.3. Read and write bytes through this socket, communicating with the client
   3.4. Close the client socket (and go back to listening)
4. Close the "receptionist" socket
```

Now, let us look at what the client is doing (again, steps 1 and 2 are often done in a single function call):

```
What the client does…
1. Create a socket
2. Make a connection request on an IP address / port
3. Read and write bytes through this socket, communicating with the client
4. Close the client socket
```

The details of the syntax depend on the programming language used to implement the client and/or the server. As mentioned before, the Socket API is available in pretty much every programming language. Here are some pointers to API documentations that you will get you started:

- Java
- C This is a great guide, which will provide you all the information you need to write client-server applications in C. Check it out!
- python
- Node.js
- Microsoft .Net
- PHP

The following code snippets show how the previous pseudo code can be implemented in Java.

```
// Server (exception handling removed for the sake of brevity)

boolean serverShutdownRequested = false;
ServerSocket receptionistSocket = new ServerSocket(80);

while (!serverShutdownRequested) {
  Socket clientSocket = receptionistSocket.accept(); // blocking call
  InputStream is = clientSocket.getInputStream();
  OutputStream os = clientSocket.getOutputStream();

  int b = is.read(); // read a byte sent by the client
  os.write(b);       // send a byte to the client

  clientSocket.close();
  is.close();
  os.close();
}

receptionistSocket.close();
```

```
// Client (again, exception handling has been removed)

Socket socket = new Socket("www.heig-vd.ch", 80);

InputStream is = socket.getInputStream();
OutputStream os = socket.getOutputStream();

int b = 22;
os.write(b);       // send a byte to the server
b = is.read();     // read a byte sent by the server

clientSocket.close();
is.close();
os.close();
```

In the code above, every byte that is written by the client on its `os` output stream will arrive on the server's `is` input stream. Similarly, every byte written by the server on its `os` stream will arrive on the client's `is` stream. We clearly see that the communication is bidirectional and happens through 2 sockets (remember: **one socket is one extremity of the virtual pipe**).

## 5. Handling Concurrency in TCP Servers

For most TCP servers, we want to be able to **talk to more than one client at the same time**. Furthermore, we want to do it in a *scalable* way, with the ability to talk to gazillions of clients at the same time (because we are living in 2014 and we are web scale). There are different ways to achieve this goal, by using **one or more processes** and **one or more threads**.

Let us compare 5 alternatives:

- single process, single-threaded, blocking servers
- multi process single-threaded, blocking servers
- mono process, multi-threaded, blocking servers
- mono process, single-threaded, non-blocking servers (multiplexing)
- mono process, single-threaded, non-blocking servers (asynchronous programming)

## 5.1. Single Process, Single-Threaded, Blocking Servers

This is **the simplest type of server that you can write**, but it as a limitation that makes it almost unviable. Indeed, this type of server is able to **talk to only one client at the time**. For some stateless protocols (where the client and the server communicate during a very short period) and when the traffic is very low, it might be an option, but frankly… Consider the following Java code:

```
try {
    serverSocket = new ServerSocket(PORT);
} catch (IOException ex) {
    Logger.getLogger(SingleThreadedServer.class.getName()).log(Level.SEVERE, null, ex);
    return;
}

while (true) {
    try {

        LOG.info("Waiting (blocking) for a new client…");
        clientSocket = serverSocket.accept();

        in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        out = new PrintWriter(clientSocket.getOutputStream());
        String line;
        boolean shouldRun = true;

        LOG.info("Reading until client sends BYE or closes the connection...");
        while ( (shouldRun) && (line = in.readLine()) != null ) {
            if (line.equalsIgnoreCase("bye")) {
                shouldRun = false;
            }
            out.println("> " + line.toUpperCase());
            out.flush();
        }

        LOG.info("Cleaning up resources...");
        clientSocket.close();
        in.close();
        out.close();

    } catch (IOException ex) {
    …
    }
}
```

The following line, which is used to set the socket in listening mode and to accept connection requests from clients, is very important:

```
        clientSocket = serverSocket.accept();
```

**You have to understand that `accept()` is a *blocking* call. This means that the execution of the current thread will suspend until a client arrives**. Nothing else will happen. So here is what happens when you execute the server. It will bind a socket to a

port, wait for a client to arrive, serve that client (which might take a while, depending on the application protocol…) and *only then* be ready to serv the next client. In other words, clients are served in pure sequence. *That sounds a bit like a Cronut queue…*

Have a look at the StreamingTimeServer example, which shows how a very (too) simple TCP server can be implemented in Java.

## 5.2. Multi Process, Single-Threaded, Blocking Servers

To improve the situation and to be able to handle several clients concurrently, a first idea is to use multiple processes. This was the standard approach for a long time, when servers where written in C and before multi-threading libraries became popular. The apache httpd server used (and still gives the option to use) this method.

The model works as follows: the server starts in a first process, which binds a socket and accepts incoming connection requests. **Each time a client makes a connection request, a child process is forked**. Since a child process inherits file descriptors from its parent, the new process has a socket that is ready to use for talking with the client. At this point, we thus have two processes that work in parallel: the parent does not care about the first client any more and can go back to listening for other clients, while the child can take care of the client and service its request.

The following code snippet, which is part of the great Beej's Guide to Network Programming, shows how this can be implemented in C (the complete code is available here):

```
printf("server: waiting for connections...\n");

while(1) {  // main accept() loop
    sin_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
    if (new_fd == -1) {
        perror("accept");
        continue;
    }

    inet_ntop(their_addr.ss_family,
        get_in_addr((struct sockaddr *)&their_addr),
        s, sizeof s);
    printf("server: got connection from %s\n", s);

    if (!fork()) { // this is the child process
        close(sockfd); // child doesn't need the listener
        if (send(new_fd, "Hello, world!", 13, 0) == -1)
            perror("send");
        close(new_fd);
        exit(0);
    }
    close(new_fd);  // parent doesn't need this
}
```

While this approach works, it is not perfect and **can show limitations under heavy load**. Forking a process is a relatively **expensive operation** (in terms of time and resource consumption). This is why we say that the approach does not scale very well.

## 5.3. Single Process, Multi-Threaded, Blocking Servers

Another approach, which has become very popular with the rise of the Java platform and of multi-threading libraries, is to **rely on multiple threads instead of multiple processes**. Remember that a thread is often described as a **lightweight process** (this indicates that there is **less overhead** in thread creation and switching, in comparison with process creation and switching).

This approach follows the same logic and structure as the previous one, but is implemented in a single process. A first thread is started and is responsible for accepting connection requests in a loop. **Whenever a client arrives, a new thread is created to handle the communication with the client**. This allows the "receptionist" thread to immediately go back to welcoming other clients. If you are not familiar with multi-threaded programming in Java, you should refer to this section of the Java tutorial.

The following Java code illustrates the idea. Here are the key points about the code:

- The `MultiThreaderServer` class contains two inner classes. These two inner classes implement the standard `Runnable` interface, which means that their `run()` method is meant to execute on its own thread.
- The responsibility of the `ReceptionistWorker` class is to accept incoming connection requests, in a loop. In other words, it should create a server socket and accept connection requests. Whenever a client arrives, it should create a new instance of the `ServantWorker` class, pass it to a new instance of the `Thread` class, which it should then start. When this is done (which is fast), the class should go back to accepting connection requests.
- The responsibility of the `ServantWorker` class is to take care of a particular client. It provides a `run()` method that executes on its own thread. This method has access to the input and output streams connected to the socket and can use them to receive bytes from, respectively send bytes to the client.

```
public class MultiThreadedServer {

    final static Logger LOG = Logger.getLogger(MultiThreadedServer.class.getName());

    public void serveClients() {
        new Thread(new ReceptionistWorker()).start();
    }

    private class ReceptionistWorker implements Runnable {

        @Override
        public void run() {
            ServerSocket serverSocket;

            try {
                serverSocket = new ServerSocket(PORT);
            } catch (IOException ex) {
                LOG.log(Level.SEVERE, null, ex);
                return;
            }

            while (true) {
                LOG.info("Waiting (blocking) for a new client...");
                try {
                    Socket clientSocket = serverSocket.accept();
                    new Thread(new ServantWorker(clientSocket)).start();
                } catch (IOException ex) {
                    LOG.log(Level.SEVERE, ex.getMessage, ex);
                }
            }

        }
```

```
        private class ServantWorker implements Runnable {

            Socket clientSocket;
            BufferedReader in = null;
            PrintWriter out = null;

            public ServantWorker(Socket clientSocket) {
                try {
                    this.clientSocket = clientSocket;
                    in = new BufferedReader(new InputStreamReader(clientSocket.getInputSt
                    out = new PrintWriter(clientSocket.getOutputStream());
                } catch (IOException ex) {
                    LOG.log(Level.SEVERE, ex.getMessage(), ex);
                }
            }

            @Override
            public void run() {
                String line;
                boolean shouldRun = true;

                try {
                    LOG.info("Reading until client sends BYE or closes the connection..."
                    while ((shouldRun) && (line = in.readLine()) != null) {
                        if (line.equalsIgnoreCase("bye")) {
                            shouldRun = false;
                        }
                        out.println("> " + line.toUpperCase());
                        out.flush();
                    }

                    LOG.info("Cleaning up resources...");
                    in.close();
                    out.close();
                    clientSocket.close();

                } catch (IOException ex) {
                    if (in != null) {
                        try {
                            in.close();
                        } catch (IOException ex1) {
                            LOG.log(Level.SEVERE, ex.getMessage(), ex1);
                        }
                    }
                    if (out != null) {
                        out.close();
                    }
                    if (clientSocket != null) {
                        try {
                            clientSocket.close();
                        } catch (IOException ex1) {
                            LOG.log(Level.SEVERE, ex1.getMessage(), ex1);
                        }
```

```
                    }
                    LOG.log(Level.SEVERE, ex.getMessage(), ex);
                }
            }
        }
    }
}
```

Even if threads are more lightweight than processes, this approach has also shown limitations under heavy load. For servers that need to handle a large number of concurrent connections, it has shown not be viable. Also note that this approach is sometimes combined with the previous one. In other words, some servers fork several process and use several threads in each of the processes. Have a look at the apache httpd MPM worker module for instance.

If you want to compare the behavior of single-threaded and multi-threaded servers in Java, have a loot at the TcpServers example.

## 5.4. Single Process, Single-Threaded, Non-Blocking Servers (multiplexing)

Even if we are implementing a server with a single process and a single thread, there are ways to handle multiple clients concurrently. What this means is that there are ways to **monitor more than one socket at the time** and to react to incoming data on them. The first one is to set the sockets in a particular state (non-blocking). to use specific system calls. The way to implement this model depends on the operating system (i.e. it depends on the system calls supported by the operating system). While solutions have existed for a long time, they have evolved over the years.

To understand how multiplexing works, **let us take the analogy of a restaurant**. Waiters are taking care of customers, sitting at their tables:

- **Without multiplexing**, we could think of two approaches for serving a room full of customers. The **first approach** would be to have a single waiter (single process, single thread) in the restaurant. We would ask him to take care of each table, in sequence. In other words, the waiter would start to serve a new table only when the previous one has be fully served (greetings, drinks, food, desert, bill, greetings). That does not seem very realistic, does it? The **second approach** would be to hire more waiters (one waiter per table). The job of a waiter would still be the same, in other words to take care of a single table at the time. This approach (multiple processes and/or threads) is better, but it proves to be quite expensive for the restaurant who has extra salaries to pay.

- **With multiplexing**, the idea is to assign a number of tables to each waiter (maybe there is one, maybe there are several). The job of the waiter is now a bit different. His job is to have an eye on what is happening at each of *his* tables. Is anyone out of bread at any of my tables? Is anyone calling me at any of my tables? Is anyone asking for the bill at one of my tables? It is an interesting idea, but we have to be careful about the time it takes to check the status of all assigned tables. Imagine a waiter who would have to check what is happening at 100 tables. He would not have the time to do anything else (performance is an issue with some of the related system calls).

How does it work in practice? We have mentioned special system calls before. The `select()` and `poll()` functions are two of them. They work with sockets that have been put in a special, *non-blocking* state. As a result, the usual calls ( `accept()` , `read()` , `write()` ) do not block the execution of the current thread. The `select()` and `poll()` functions are blocking, but they allow the programmer to give a list of sockets. The execution of the thread will block until *something* happens to *at least one* of the sockets in the list. Have a look at Beej's Guide to Network Programming, which has a section dedicated to this topic with code examples.

## 5.5. Single Process, Single-Threaded, Non-Blocking Servers (asynchronous programming)

There is another way to handle multiple connections at the same time, with a single process and a single thread. It also consists of using non-blocking IOs, but implies a different programming style and control flow structure. It is associated with an event-based approach and with the use of callback functions.

Before we get to the technical details, let us consider a real world analogy again. This time, we will think about what is happening at a coffee shop, where customers have to order and collect their drinks at the counter (self-service):

- **As a customer, when you enter the coffee shop**, you have to **stand in a first queue**. You have to wait for someone to take your order and collect your money. You then have to **move into a second queue**, waiting for someone to prepare your drinks. **This is the world of synchronous customer service**. It works, but it does not allow you to make the most efficient use of your time. When you are waiting in a queue, you cannot do anything valuable at the same time. If there is a single employee doing everything (single process, single thread), the queue will grow quickly. Hiring additional employees to have parallel queues (multiple threads) is of course possible, but it has a cost.
- **Compare this scenario** with what happens in some coffee shops. Firstly, instead of directly standing in a queue, you pick up a ticket with a number (e.g. '67'). You then go to a table, sit down, take your laptop out of your bag and start working on your RES lab.

> At some point, you hear "Customer number 67, please!". You know that you can now stand up and go directly to the person who takes the order and collects the money. No waiting time. You place your order, pay for it, and receive a small device. You go back to your table and can resume your work on the damn lab… A couple of minutes later, the device starts to flash, letting you know that your order is ready. You get up, walk to the distribution area and collect your drink (again, without having to wait). **This is the world of asynchronous customer service**. It works because **you are notified when certain events happen**: *'Someone is available to take your order'*, *'Your order is ready'*, etc. It also works because you know how to behave when you receive these notifications (as we will see, you implement *callback* functions).

In technical terms, doing asynchronous IO programming consists of using non-blocking IOs (hence of setting sockets in non-blocking state) in combination with an event-based approach. This is possible in various programming environments and has become very popular. It is one of the features of the *Node.js platform*, which applies the asynchronous pattern across the board. It is also one aspect of the *reactive programming* approach. To illustrate what that means, consider the following code:

```
// This is an example for a simple echo server implemented in Node.js. It
// demonstrates how to write single-threaded, asynchronous code that allows
// one server to talk to several clients concurrently.
//
// readline is a module that gives us the ability to consume a stream line
// by line

var net = require('net');
var readline = require('readline');

// let's create a TCP server
var server = net.createServer();


// it can react to events: 'listening', 'connection', 'close' and 'error'
// let's register our callback methods; they will be invoked when the events
// occur (everything happens on the same thread)
server.on('listening', callbackFunctionToCallWhenSocketIsBound);
server.on('connection', callbackFunctionToCallWhenNewClientHasArrived);

// we are ready, so let's ask the server to start listening on port 9907
server.listen(9907);

// This callback method is invoked after the socket has been bound and is in
// listening mode. We don't need to do anything special.
function callbackFunctionToCallWhenSocketIsBound() {
    console.log("The socket is bound and the server is listening for connection requests.
    console.log("Socket value: %j", server.address());
}
```

```
// This callback method is invoked after a client connection has been accepted.
// We receive the socket as a parameter. We have to attach a callback function to the
// 'data' event that can be raised on the socket.
function callbackFunctionToCallWhenNewClientHasArrived(socket) {

    // We wrap a readline interface around the socket IO streams; every byte arriving
    // on the socket will be forwarded to the readline interface, which will take care
    // of buffering the data and will look for end-of-line separators. We will not regist
    // a callback handler on the socket (it is a possibility), but rather on the readline
    // interface. The 'line' events are raised whenever a new line is available.
    var rl = readline.createInterface({
      input: socket,
      output: socket
    });

    rl.on('line', callbackFunctionToCallWhenNewDataIsAvailable);

    // This callback method is invoked when new data is available on the socket
    // We can process it, which in our case simply means dumping it to the console
    function callbackFunctionToCallWhenNewDataIsAvailable(data) {
        console.log("Client has sent: " + data);
        if (data.toString().toUpperCase() == 'BYE') {
            console.log("Client has sent 'bye', closing connection...");
            socket.end();
        } else {
            socket.write(data.toString().toUpperCase() + '\n');
        }
    }

  console.log('A client has arrived: ' + socket.remoteAddress + ":" + socket.remotePort);

}
```

This code implements a TCP server capable of servicing multiple clients at the same time, on a single thread. There are 2 important elements to look at in the code. Firstly, we see that the programmer has defined several callback functions. This is where he has expressed what to do when certain events happen. Secondly, he has subscribed to several types of events and registered the callback functions with them. In other words, he has expressed the fact that **when** some events happen, **then** some functions need to be called. Note that Node.js developers usually prefer to use **anonymous callback functions** (defined inline), but this example was written to emphasize the nature of the callback functions.

The code is available in the TcpServerNode example, which you can run with the following command:

```
node server.js
```

# Resources

## MUST read

- This section of the Java tutorial, which explains how to use the Socket API in Java.

## Additional resources

- A section of the TCP/IP Guide dedicated to TCP, if you need a refresh on that topic.
- Beej's Guide To Network Programming, which a great resource if you want to use the Socket API in C.
- A nice course about network programming with the Socket API in C.
- An example for a simple TCP server that uses non-blocking IOs and multiplexing with the `select()` system call.
- A presentation, with embedded videos, which introduces Nodes.js, talks about asynchronous IOs and about the C libraries used by Nodes.js. The presentation also has two interesting slides that compare apache (using threads) and nginx (using an event loop). Another presentation on the same topic.
- An article that explains how Java NIO adds non-blocking IO to Java IO. Another article that explains how Java NIO.2 adds support for asynchronous IO to Java NIO.
- The home page for the lib event C library, which provides a unified interface for non-blocking IO programming, isolating the developer from operating system specific function calls. This section of the documentation gives a good introduction to asynchronous IO.

# What Should I Know For The Test and The Exam?

Here is a **non-exhausive list of questions** that you can expect in the written tests and exams:

- When writing a network client in Java, you have created a socket and established a connection with a TCP server. The variable `mySocket` holds a reference to this socket. How can you send bytes to the server?
- When writing a network server in Java, you want to bind a socket to port 2367 and accept at most 4 connection requests. How do you do that?

- What is the difference between a process and a thread?
- Describe 4 different ways to handle concurrency in TCP servers.
- Explain the difference between a synchronous and an asynchronous function call.
- **Write the Java code of a single threaded "echo" server**, which accepts connection on TCP port 3232. When a client connects, the server should send back the following bytes `'HELLO\n'`, read one line and send back this line converted to uppercase.
- **Write the Java code an "echo" client**, which connects to this server, sends the following bytes `'ThIS is a TeSt'`, reads the response and validates that the response is what was expected.
- **Write the Java code to make the single threaded "echo" server multi-threaded**. Be able to explain how the solution uses the `Runnable` interface and the `Thread` class.

# UDP Programming


- Objectives
- Lecture
  - 1. The UDP Protocol
  - 2. UDP in the Socket API
  - 3. So… How Do Clients and Servers Use UDP?
    - 3.1. Messaging Patterns
    - 3.2. Making API Calls
  - 4. Unicast, Broadcast and Multicast
    - 4.1. Implementing Broadcast in Your Programs
    - 4.2. Implementing Multicast in Your Programs
  - 5. Service Discovery Protocols
- Resources
  - MUST read
  - Additional resources
- What Should I Know For The Test and The Exam?


# Objectives

The goal of this lecture is to **continue our study of network programming** and to explain how developers can write **client and server programs** that use the **UDP protocol** to communicate with each other.

After this lecture, you should be able to **write Java programs that send and receive UDP datagrams**. You should also be able to describe the notions of **unicast**, **broadcast** and **multicast** message distribution modes. Furthermore, you should be able to explain how **service discovery protocols** can be implemented thanks to multicast messaging.

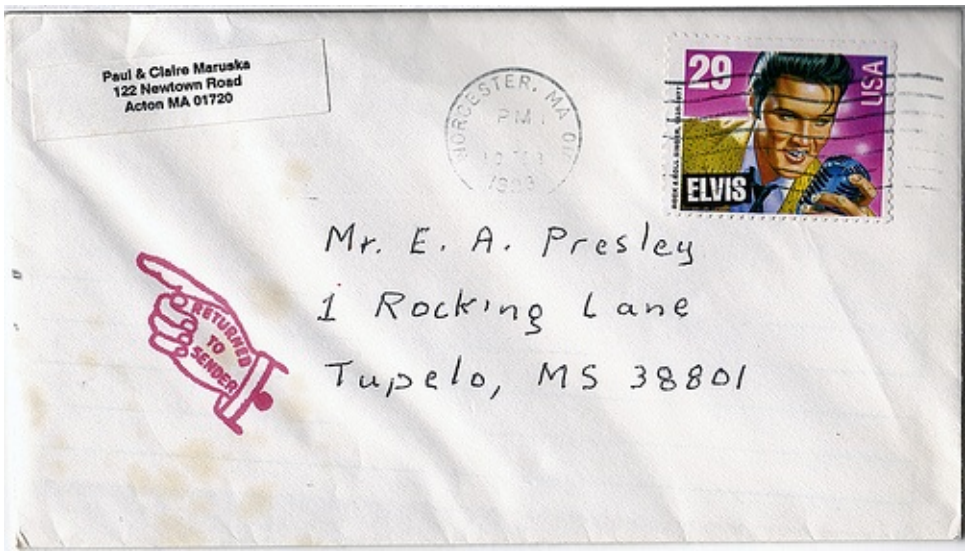
Last but not least, you should be **aware of what it means to use an unreliable transport protocol** as a basis for your application-level protocol. You should be able to describe what clients and servers have to do in order to deal with this unreliability.

# Lecture

## 1. The UDP Protocol

In the previous lecture, we have looked at the TCP protocol and seen as it offers **one way to transport application-level messages** across the network (in a reliable fashion). We have seen how the telephone system is a good analogy for how TCP works. In this lecture, we will look at **another transport protocol**, namely **UDP**. Here, it is the **postal system** that provides a useful analogy.

When you write and send letters, **you do not need to establish a synchronous connection** with your peers. You drop your letters in a mailbox, they are taken care of by the system and at some point they *should* be delivered at destination.




In the postal system, every letter is handled independantly and has all the **information required for the routing** (i.e. the elements of the destination address). Furthermore, the letter should also mention the **sender address**. When you receive a letter, this is very useful because it gives you the information you need if you want to **send back a response**. UDP applies the same logic: datagrams contain both the source and destination coordinates, which can be used by the recipients to prepare their replies.

Also, remember that in some situations, letters *may* get lost. Does that mean that it is impossible to specify and implement **reliable application-level protocols on top UDP** (which is a requirement for file transfer protocols for instance)? No, it is not, but the burden is on the application developer's shoulders. What is offered by TCP has to be implemented on top of UDP.

What does that mean concretely? What does the application developer need to do when implementing a UDP-based protocol? Here are a couple of important points:

- Firstly, **there are application-level protocols where this is not really an issue** (in other words, there are application protocols that are **tolerant to data loss**). Think of a video streaming application protocol: it does not matter if some of the data is lost. The

worse that can happen is a lower quality. Think of a protocol used by sensors to report measurements to a data collection infrastructure (see this example). Again, loosing some measurements may be totally acceptable.

- For applications where data loss cannot be accepted, the client needs a way to **know if a datagram that it has sent has been received by the server**. If it thinks that the datagram has been lost, then it should try again and **resend the datagram**. That should remind you of the way TCP works: a combination of **acknowledgments** and **timers** are certainly one way to address this issue.
- However, you should also remember that sometimes, the client will *think* that one of its datagrams has been lost, although it as actually been received and processed by the server. This would be the case if **it is the server's acknowledgment that has been lost or has arrived too late**. The consequence is that the server will then probably **receive multiple copies of the same datagram**. It needs to be aware of that possibility and act appropriately.
- For illustration purposes, **think of a rudimentary application-level protocol**, where clients can send commands to increase the value of a counter (there is a single counter managed by the server). Clients send `INC-COUNTER` commands encapsulated in UDP datagrams. As you can guess, when the server receives an `INC-COUNTER` command, it increments the value of the counter by one. Let us consider the following scenarios:
  - Let's imagine that the client sends 10 commands. If no datagram is lost, then at the end of the process, **the counter will have a value of 10**. Fine, but we *know* that we are not living in an ideal world and that we *have to assume* that something bad will happen at some point.
  - **If some datagrams are lost** and not resent by the client, then **the counter will have a value smaller than 10**. As we have seen, the client and the server have to agree on an acknowledgment protocol. If the client does not receive an ack for a command *within a specified period*, it should resend the command.
  - **If the same command is sent multiple times** (because the client wrongly thinks that it has been lost) **and is processed multiple times** by the server, then **the counter will have a value greater than 10**. To avoid this problem, the server has to keep track of the commands that it has already processed and acknowledged. To be precise, it has to keep track of non-idempotent commands. That is where *stop-and-go* and *sliding window* algorithms are providing you with a proven solution. Have a look at the Trivial File Transfer Protocol (TFTP) specification for an example of a reliable application-level protocol built on top of UDP. Also have a look at a flaw in the first design of this protocol.

## 2. UDP in the Socket API

The **Socket API** provides functions for using UDP in application-level code. When using the C API, the first thing to be aware of is that it is when creating the socket that one specifies whether we want to use TCP or UDP. Compare the following two instructions:

```
// If we want to use TCP
int sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

// If we want to use UDP
int sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

Once we have a socket, we can use it both to **receive** and to **send** datagrams. In C, the developer does not use a special data structure that would represent a datagram. He uses the `sendto()` and `rcvfrom()` functions and passes byte arrays as parameters. Unlike with TCP, no connection has been established before making the calls to send and receive data. For that reason, when calling the `sendto()` function, the developer **needs to specify a destination address and a destination port**. Similarly, when using the `rcvfrom()` function, the developer has to pass a data structure, where he will **find the source address and port** after the call.

```
// Let's send a datagram. The payload is the content of buffer and the destination
// address is specified in sa
bytes_sent = sendto(sock, buffer, strlen(buffer), 0,(struct sockaddr*)&sa, sizeof sa);

// Let's wait for a datagram to arrive. We will find the payload in the buffer and
// the return address in sa
recsize = recvfrom(sock, (void *)buffer, sizeof(buffer), 0, (struct sockaddr *)&sa, &from
```

## 3. So… How Do Clients and Servers Use UDP?

## 3.1. Messaging Patterns

Different messaging patterns can be implemented with UDP:

- A common and simple pattern is the **fire-and-forget notification** pattern. The client prepares a datagram and sends it to one or more recipients. If data loss can be tolerated, nothing else needs to be done. In the thermoter example, we use this pattern. Smart thermometers publish measurements on the network on a regular basis. No response is expected and if some measurements are lost, it is not an issue.
- Another common pattern is the **request-reply** pattern. The client prepares a datagram, in which it encapsulates an application-level message (in other words, the application level message is the payload of the UDP datagram). The client then **creates a datagram socket**, which will be **used both for sending the request and for receiving the response**. Note that it is possible, but **not required**, to specify a port. If no port is specified, then the operating system will automatically assign a free port to the socket. Now, remember what we said about the **return address** on physical letters. Every datagram (together with the encapsulating IP packet) sent via the socket will contain 4 values. Firstly, the **destination address and port** (**explicitly** set by the developer). Secondly, the **source address and port** (**automatically** set by operating system, retrieved from the socket).

  After sending the datagram, the client will **wait for datagrams to come back** on the socket. When the **server** receives the datagram with the request (it has previously **created a socket on a known port** and is using it to accept datagrams), it extracts the application-level request, processes it, generates and application-level response and

prepares a reply datagram. **It extracts the source IP and port from the request datagram, and uses these values as destination parameters for its reply**. Finally, it sends the datagram via its socket.

In some cases, **one more question** needs to be addressed by the application developer (and by the application-level protocol specification). Imagine that a client sends 10 different requests to a server (*the protocol allows the client to send a new request before it has received a response to the previous one*). Let us assume that the server receives the 10 requests and sends back 10 responses. Because of the properties of UDP, **we have no guarantee that the requests and the responses will be received in sequential order**. Hence, it is quite possible that the client will receive the reply number 6 before the reply number 2. How can that work? **How can the client make sense of that unordered flow of responses**? The answer is that some information needs to be added to the application-level messages, in order to associate a reply with a given request. This information is often referred to as a **correlation identifier**. The client has to generate a different correlation identifier for each request and the server has to include it in the corresponding response.

Have a look at the COAP RFC to see what this means in practice. In the RFC, check out *Paragraph 5.3. Request/Response Matching*. What the COAP RFC authors call a *token* is nothing else than a correlation identifier.

- We will look at a third common pattern a bit later, which relies on multicast or broadcast, and which we will present in the context of **service discovery protocols**.

## 3.2. Making API Calls

In **Java**, the API provides an **explicit abstraction** for dealing with datagrams: the `DatagramPacket` class. Developers use it both when sending and receiving UDP datagrams from their programs. Here are the important methods defined in the class, as defined in the Javadoc documentation:

```
// Returns the IP address of the machine to which this datagram
// is being sent or from which the datagram was received.
public InetAddress getAddress();

// Returns the port number on the remote host to which this
// datagram is being sent or from which the datagram was received.
public int getPort();

// Returns the data buffer. The data received or the data to be sent
// starts from the offset in the buffer, and runs for length long.
public byte[] getData();
```

Furthermore, datagrams are sent and received via sockets that are instances of the `DatagramSocket` class. Notice that the class provides several constructors. Let us compare two of them:

```
public DatagramSocket() throws SocketException;
public DatagramSocket(int port) throws SocketException;
```

In the first constructor, we do not specify any port number. This is fine if we use the socket for **sending** datagrams. What will happen is that the system will assign us a **random UDP port** and use it as the **source port value** in the UDP header. If the application on the other side of the network wants to send us a response, it will be able retrieve and use this value as the **destination port value** in the response datagrams.

In the second constructor, we **specify a port number**. This means that we will be waiting for UDP datagams that have this value in the destination port of the datagram header. This approach is typically used to write a server according to an application-level protocol specification. **The specification defines the standard port** to be used by servers and clients. It is how client and servers can find each other to initiate an interaction.

## 4. Unicast, Broadcast and Multicast

As you remember, **applications that use TCP as a transport-layer protocol always see a pair of processes communicate with each other**. The server listens on a known port, the client makes a connection request on that port and when they are connected, they exchange data until the connection is closed. Standard one-to-one communication, which relies on **unicast** data transmission. Another way to look at it is that the TCP segments that carry application-level messages are **always going to a single destination** (defined by the destination IP address and the destination port).

From your study of the TCP/IP stack and of protocols such as ARP (used to *discover* the MAC address corresponding to an IP address), you should remember that there are **other data transmission models** within a network of nodes:

- **Broadcast** is one of them, where **a message is distributed to all nodes in the network**. As an analogy, think of what happens when **your mailbox is flooded with commercials** (just like your neighbour's). You didn't ask to receive the commercials and the advertising business did not need to know you in order to send you the glossy brochure with weekly deals. Everybody living in the city got them.
- **Multicast** is another one, where **a message is distributed to all *interested* nodes in the network**. What does it mean for a node to *be interested* and how does it work in practice? Well, multicast protocols rely on the notion of *group*, which you can think of as

**a kind of radio channel**. Just like music amateurs can decide to listen to particular radio channels, nodes in a network can decide to listen (i.e. *subscribe to*, *join*) to specific multicast groups. When other nodes (playing the role of the radio station) send messages to the group, then **all subscribers to that group will receive a copy of the message**.

Broadcast and multicast transmission models are very useful to implement certain interactions between clients and servers. They also offer ways to improve the efficiency of the communication within a network (e.g. by avoiding the need to resend the same information to multiple receipients and thus by reducing traffic). **UDP gives you a way to implement broadcast and multicast transmissions very easily**. However, be aware that there can be restrictions on the generated traffic. Sometimes, these restrictions are explicitly defined in standard specifications (for instance, broadcast in IPv4 networks is limited to the local network). Other times, they are enforced by system administration policies (for instance, **multicast is filtered on the HEIG-VD network**).

## 4.1. Implementing Broadcast in Your Programs

In order to broadcast a UDP datagram to all nodes on the local network, simply use the `255.255.255.255` broadcast address in the destination address of your datagrams. Note that the socket has to be set in a mode where it agrees to send broadcast datagrams (to avoid accidental broadcast storms). Here is how you would do it in Java:

```
// Sending 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.getByNa

socket.send(datagram);
```

And here is how you would do it in Node.js:

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

var dgram = require('dgram');
var s = dgram.createSocket('udp4');

s.bind(0, '', function() {
    s.setBroadcast(true);
});

var payload = "Node.js rocks, everybody should know!";
message = new Buffer(payload);

s.send(message, 0, message.length, 4411, "255.255.255.255", function(err, bytes) {
  console.log("Sending ad: " + payload + " via port " + s.address().port);
});
```

On the other side, if you implement an application that should listen for and process broadcasted datagrams, you do not need to do anything special. Just bind a datagram socket on the application-specific port. Here is what you do in Java:

```
// 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.getLen
        System.out.println("Received a datagram with this message: " + msg);
    } catch (IOException ex) {
        Logger.getLogger(BroadcastListener.class.getName()).log(Level.SEVERE, ex.getMessa
    }
}
```

And here is what you do in Node.js

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

var s = dgram.createSocket('udp4');
s.bind(4411, function() {
  console.log("Listening for broadcasted ads");
});

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

## 4.2. Implementing Multicast in Your Programs

In order to use multicast instead of broadcast in your programs (which is strongly recommended), what you have to do is pretty similar. Here are the additional tasks that you have to do:

1. You need to specify on which multicast group your program is goint to publish and subscribe. A multicast group is defined by an IP address in a specific range.
2. **In Java**, you need to use a special class, namely `MulticastSocket`, in order to listen and receive datagrams published on a multicast group.
3. This class defines two methods, namely `joinGroup(InetAddress group)` and `leaveGroup(InetAddress group)`. You use these methods to subscribe to and unsubscribe from the specified group.

Here is Java code for subscribing to a multicast group:

```
private InetAddress multicastGroup;
int port;
MulticastSocket socket;

public MulticastSubscriber(int port, InetAddress multicastGroup) {
    this.port = port;
    this.multicastGroup = multicastGroup;
    try {
        socket = new MulticastSocket(port);
        socket.joinGroup(multicastGroup);
    } catch (IOException ex) {
        Logger.getLogger(BroadcastListener.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

And here is the Node.js equivalent:

```
var dgram = require('dgram');
var s = dgram.createSocket('udp4');

s.bind(protocol.PROTOCOL_PORT, function() {
  console.log("Joining multicast group");
  s.addMembership(protocol.PROTOCOL_MULTICAST_ADDRESS);
});
```

## 5. Service Discovery Protocols

As we have seen before, one particularity of broadcast and multicast data transmission is that **the sender of a message does not need to know who the consumer(s) of that message will be**. In the case of broadcast, the sender knows that *all nearby nodes* will receive it. In the case of multicast, the sender knows that *all nodes that have expressed their interest* will receive it.

That is a property that is very useful in situations, where **you want to find out whether some sort of service provider is *around* and *available*** and where you are requesting for *contact information*. As an analogy, think about what is happening when a pregnant woman breaks water in an airplane. Most likely, someone will shout "Is there a doctore on board!?" (i.e. the request for a *delivery* service will be addressed to all passengers). If one or more doctors are present, they will manifest themselves and an interaction between the pregnant woman and one doctor will then be able to start.


There are **at least two methods for supporting service discovery** in a dynamic environment:

1. **The first one consists for the service providers to periodically advertise their presence**. Imagine a printer that would publish a UDP datagram on a multicast group every 5 seconds. A laptop that would be looking for available printers would create a datagram socket, join the multicast group (effectively expressing its interest in printers) and would thus receive the presence datagrams. The presence datagrams should contain the contact details (i.e the data that is required for the laptop to initiate an interaction with the printer and use its printing service).
2. **The other one works in other direction.** It consists for the printers to join a multicast group and for the printers to send datagrams containing a message with the semantic *"I am looking for a printing service"*. When receiving these requests, the printers should send back a message informing the laptop that they are available and giving the required contact details.

# Resources

## MUST read

- This section of the Java tutorial, which explains how to use UDP in Java.
- This section and this section of Beej's Guide to Network Programming.

## Additional resources

- The COAP RFC, which is an example for a protocol built on top UDP and which has reliability constraints. COAP is a protocol that was developed for Internet-of-Things (IoT) applications.
- The TFTP RVD, which is an example for a simple file transfer protocol that uses a stop-and-go algorithm for reliability and datagram ordering.

- Node.js documentation on how to use datagram sockets.

# What Should I Know For The Test and The Exam?

Here is a **non-exhausive list of questions** that you can expect in the written tests and exams:

- Explain the difference between unicast, broadcast and multicast transmission models.
- Describe a situation where it is a good idea to broadcast datagrams on the local network.
- Describe an application that makes use of multicast datagram distribution.
- Write the Java code for a client-server pair, which communicate with each other with messages encapsulated in UDP datagrams.
- Explain the notion of service discovery. Explain how you would specify a protocol allowing chat clients to discover available chat servers. Explain how you would implement it with multicast.
- Is it possible to implement reliable application-level protocols on top of UDP? If no, explain why. If yes, explain how.
- Give one reason why you would use UDP instead of TCP as a transport protocol for your application-level protocol.

# Lecture 4: The HyperText Transfer Protocol


- Lecture
  - 1. Introduction
  - 2. A First Look at an HTTP Conversation
  - 3. HTTP is a Stateless Request-Reply Transfer Protocol
    - 3.1. The HTTP Protocol is Stateless, But Many Web Applications are NOT
    - 3.2. HTTP is Stateless, But Several Requests Can Be Handled in One TCP Connection
  - 4. HTTP is Used to Transfer Representations of Resources
    - 4.1. Resources vs Representation of Resources
    - 4.2. Content Negotiation
  - 5. HTTP Messages: Requests and Responses
    - 5.1. Syntax
    - 5.2. Some Interesting Methods
    - 5.3. Some Interesting Headers
    - 5.4. Some Interesting Status Codes
  - 6. Parsing HTTP Messages
- Resources
  - MUST read
  - Additional resources
- What Should I Know For The Test and The Exam?


# Lecture

## 1. Introduction

Let's face it: there is not a day, probably not an hour, maybe not even a minute where you are not using the HTTP protocol in one way or another:

- You are certainly spending a lot of time **browsing and reading information** published on the WWW.
- You are also using a wide variety of **applications** that have been built on top the WWW technical infrastructure. Think about social networks, e-commerce sites, online games, e-banking services: they all rely on HTTP to transfer information between the "brain" of

the application (i.e. the server) and its "face" (i.e. the client). It does not matter if you are using the applications via your browser or via a **mobile** application: without HTTP you would not be able to poke your friends or to pay your bills.

- Even if you are not seeing any LCD display nearby, you will increasingly be exposed to HTTP without noticing it. The protocol is already used to connect all sorts of **machines** and **objects** to *The Cloud*. Just think about the coffee vending machine down the hall or about the baby scale that you are likely to get some day.

This shows that HTTP has become a generic protocol, on top of which many distributed systems are being built. Whether you are a **systems engineer**, a **software engineer**, an **embedded systems engineer** or a **security engineer**, you have to speak this protocol natively and must master its concepts and syntax. You have to be able to open a terminal, open a TCP connection with a remote server, type an HTTP request, hit return and interpret the response. *Resistance is futile*.

In this lecture, we will present the **key protocol concepts** and look at what a **typical HTTP conversation** looks like. We will also include **fragments of the 176-pages [RFC 2616]** (http://tools.ietf.org/html/rfc2616), which specify how these concepts are put in practice.

## 2. A First Look at an HTTP Conversation

Before looking at what is defined in the HTTP specification, let us observe what a typical HTTP exchange between a client and a server looks like. To capture the traffic, different options are available. Of course, it is possible to use a packet analyzer like **Wireshark**. But a lot of information is already available directly in your favorite browser. For instance, you may install the **Firebug** browser extension (and use the Network tab), or use the various developer-oriented features that are commonly available (every browser has its own menu, but the kind of information you get is always very similar).

In our test scenario, the user has typed `http://www.nodejs.org` in the browser navigation bar and hit return. At this point, you can observe that the following bytes are being **sent from the client to the server**:

```
GET / HTTP/1.1 CRLF
Host: www.nodejs.org CRLF
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:28.0) Gecko/20100101 Firefox/
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 CRLF
Accept-Language: en-us,en;q=0.8,fr;q=0.5,fr-fr;q=0.3 CRLF
Accept-Encoding: gzip, deflate CRLF
Cookie: __utma=212211339.431073283.1392993818.1395308748.1395311696.27;
__utmz=212211339.1395311696.27.19.utmcsr=stackoverflow.com|utmccn=(referral)|utmcmd=refer
|utmcct=/questions/7776452/retrieving-a-list-of-network-interfaces-in-node-js-ioctl-siocg
Connection: keep-alive CRLF
CRLF
```

There are a couple of interesting things that we can observe. We can quite **easily read HTTP messages**. We see that we are facing a text-based protocol. We see that the structure of the **first line** is a bit different from the following ones, which all start with a name, followed by a colon (they look like **headers** or some kind of metadata)

Right after the previous bytes have been sent to the server, we see that **a response comes back from the server**:

```
HTTP/1.1 200 OK CRLF
Server: nginx CRLF
Date: Sat, 05 Apr 2014 11:45:48 GMT CRLF
Content-Type: text/html CRLF
Content-Length: 6368 CRLF
Last-Modified: Tue, 18 Mar 2014 02:18:40 GMT CRLF
Connection: keep-alive CRLF
Accept-Ranges: bytes CRLF
CRLF
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <link type="image/x-icon" rel="icon" href="favicon.ico">
    <link type="image/x-icon" rel="shortcut icon" href="favicon.ico">
    <link rel="stylesheet" href="pipe.css">
...
```

Again, the response is easy to interpret. Once again, we see that there is a **first line** with some information (we see a `OK`, so it looks like everything went well). Once again, we see a number of lines that look like **headers**. And this time, after an **empty line,** we see **HTML content** (note: only the beginning has been included in the previous transcript). So that must be the markup that should be rendered by the browser!

Now that we have a first sense of what HTTP messages look like, let us dig into the RFC and examine what are the most important concepts defined in the specification.

# 3. HTTP is a Stateless Request-Reply Transfer Protocol

**HTTP is a client-server protocol, which is used to transfer representations of resources (more on this later) over TCP/IP networks.** The server accepts connection requests on a TCP port (the default is port 80). Once connected, the clients send requests to either *ask for* (this is what happens when you click on a hyperlink) or *send* data (this is what happens when you press on the submit button in a HTML form). HTTP is used to transfer all sorts of **media types** (text, images, binary data, etc.), in all sorts of **formats** (plain text, HTML markup, XML markup) and using all sorts of **character encoding systems** (UTF-8, etc.).

**HTTP is stateless** because every request issued by a client is considered as independent from the others. Hence, **HTTP servers do not need to manage a session** and do not need to maintain a state for each client. Once they have sent back a response to a client, they can just forget about it. This design choice has a number of benefits:

- it makes the **implementation of clients and servers easier**;
- it **reduces the consumption of resources** (maintaining state for each client can require a lot of memory on the server side);
- it makes it easier to **deal with load and failure**, by using a cluster of equivalent HTTP servers (the client does not care which server will be fulfill its requests and each request might be fulfilled by a different server in the pool).

Having said that, there are two important points that you have to be aware and that we will discuss now. Firstly, while the HTTP protocol is stateless, many Web applications are not stateless. Secondly, the fact that HTTP is stateless does not mean that we have a different TCP connection for each request-reply exchange.

## 3.1. The HTTP Protocol is Stateless, But Many Web Applications are NOT

If you think about what you do with your web browser, this point about statelessness should be puzzling:

- When you are buying groceries on your favorite e-commerce site, you have a shopping cart and can add one product after the other. *So the server does maintain a state about the shopping session*.
- When you are checking your grades on the school information system, you first have to login, then can check your progress in several courses and finally can logout. *So the server does maintain a state about the conversation, knows who you are and remembers that you have provided valid credentials earlier in the conversation*.

Does that mean that the HTTP RFC is lying to us? Of course not. But that means that **it is the responsibility of the application built on top of HTTP to manage the conversation state**. There are a number of ways to do that:

- One way is to transport the entire conversation state in every request and every response exchanged in the context of an application session. One technique for doing that is to use hidden HTML form fields. The conversation between a client and a server would look like this (the client needs to **repeat** what he has said before each time he talks to the server):

```
C: Hello, I am new here. My name is Bob.
S: Welcome, let's have a chat [You told me that "My name is Bob"].
C: [My name is Bob]. What's the time?
S: Hi again Bob. It's 10:45 AM. [You told me that "My name is Bob". You asked me what is
```

- Another way is to store the state on the server side, to assign a unique id to every application session and to pass this session id back and forth in every request and response. The conversation between a client and a server would look like:

```
C: Hello, I am new here. My name is Bob.
S: Welcome Bob, let's have a chat. Your session id is 42.
C: My session id is 42. What's the time?
S: -- checking my notes… hum… ok, I found what I remember about session 42…
S: Hi again Bob. It's 10:45 AM.
C: My session id is 42. How do you do?
S: -- checking my notes… hum… ok, I found what I remember about session 42…
S: I am fine, Bob, thank you.
C: My session id is 42. How do you do?
S: -- checking my notes… hum… ok, I found what I remember about session 42…
S: I told you I am fine… are you stupid or what?
C: My session id is 42. If you take it like that, I am gone. Forever.
S: -- checking my notes… hum… ok, I found what I remember about session 42…
S: -- putting 42 file into trash…
S: Bye Bob.
```

There are different ways to pass the session id back and forth. One way is to use a parameter in the query string (which has security implications). Another way is use **cookies**, as defined in the RFC 6265. We will talk about HTTP *headers* in a short while and this is how cookies are passed back and forth (there is a `Set-Cookie` header sent by the server and a `Cookie` header sent by the client). For now, let us focus on the principle of cookie-based state management:

- When the client sends the initial request to a server (starting a session), the server generates a session id, puts it into a cookie and sends the cookie back to the client with its response.
- Whenever the client sends a subsequent request to the server, it **has to** send the cookie(s) that it has previously received from this server.

Various features and subtleties are specified in RFC 6265 and implemented by browsers. We will come back to them at a later stage, when we look at HTTP from an infrastructure point of view (dealing with reverse proxies and load balancers).

## 3.2. HTTP is Stateless, But Several Requests Can Be Handled in One TCP Connection

In the previous version of the protocol, i.e. in HTTP 1.0, the interaction between a client and a server was based on the use of a different TCP connection for every request-reply exchange. The client would establish a connection, send its request. The server would send its response and close the connection. Very often, the client would immediately establish a new connection with the same server, send a new request and receive a response. Why immediately? Think about typical web pages: the main content may be the HTML markup, but various resources are referenced in HTML tags: inline images, CSS stylesheets, javascript scripts, etc. Often, the main markup and the referenced resources are hosted on the same server, so a typical interaction between the client and the server looks like:

```
C: GET /beautifulPage.html HTTP/1.0
S: HTTP/1.0 200 OK
C: -- parsing the html so that I can render it…
C: -- looking for <img>, <link> and similar tags…
C: GET /img/banana.jpg HTTP/1.0
S: HTTP/1.0 200 OK
C: GET /img/apple.jpg HTTP/1.0
S: HTTP/1.0 200 OK
C: GET /img/pear.jpg HTTP/1.0
S: HTTP/1.0 200 OK
C: GET /img/unicorn.jpg HTTP/1.0
S: HTTP/1.0 200 OK
C: GET /img/grape.jpg HTTP/1.0
S: HTTP/1.0 200 OK
C: GET /img/ananas.jpg HTTP/1.0
S: HTTP/1.0 200 OK
C: GET /css/happy.css HTTP/1.0
S: HTTP/1.0 200 OK
C: GET /js/app.js HTTP/1.0
S: HTTP/1.0 200 OK
C: -- ok… now I am able to fully render the page…
```

**Is that a problem? Yes, it is and the HTTP specification tells us why:**

```
Prior to persistent connections, a separate TCP connection was
established to fetch each URL, increasing the load on HTTP servers
and causing congestion on the Internet. The use of inline images and
other associated data often require a client to make multiple
requests of the same server in a short amount of time. Analysis of
these performance problems and results from a prototype
implementation are available [26] [30]. Implementation experience and
measurements of actual HTTP/1.1 (RFC 2068) implementations show good
results [39].

[…]

A significant difference between HTTP/1.1 and earlier versions of
HTTP is that persistent connections are the default behavior of any
HTTP connection. That is, unless otherwise indicated, the client
SHOULD assume that the server will maintain a persistent connection,
even after error responses from the server.

Persistent connections provide a mechanism by which a client and a
server can signal the close of a TCP connection. This signaling takes
place using the Connection header field (section 14.10). Once a close
has been signaled, the client MUST NOT send any more requests on that
connection.
```

**Source**: Section 8

One of the sources mentioned in the RFC is available here and explains why the slow start algorithm used by TCP for congestion control has a negative impact on HTTP 1.0 performance.

To address this issue, HTTP 1.1 has introduced **persistent connections**. This means that a client can now establish a TCP connection, send a first request (e.g. to retrieve one HTML page), receive the response, send a second request (e.g. to retrieve and embedded image), receive the response, and continue like this until the TCP connection is finally closed. Furthermore, HTTP 1.1 has also introduced the notion of **pipelining**, allowing the client to send a first request and to immediately send a second request before having received the first response (under certain conditions).

## 4. HTTP is Used to Transfer Representations of Resources

In the introduction, we have already mentioned that HTTP is not only used to transport static (HTML) documents, but also to implement distributed systems and applications. For this reason, you should not say that "HTTP is used to transport Web pages", because it is not precise enough and misleading.

> While this topic is out of scope for this course, you should also be aware that REST APIs, based on the REST architectural style have become a very popular, if not the de facto standard, way to expose and consume web services. You should also be aware that the concepts that we are about to discuss are at the core of the REST approach.

## 4.1. Resources vs Representation of Resources

The HTTP protocol specification defines three key concepts that you really have to understand:

- Firstly, the RFC defines the notion of **resource** as "a network data object or service that can be identified by a URI". The point is that **the notion of resource is very generic and can apply to very different things**. An article published in an online newspaper is a resource. A list of articles in a given category is a resource. A stock quote updated in realtime is a resource. A webcam is a resource.
- Secondly, the RFC defines the notion of **representation** and states that what is transported in the protocol messages is not the resources themselves (*would it make sense to transport an actual webcam?*), but **representations of the resources**. You have to remember that for one resource, there might be several ways to represent it. Think of the stock quote example: you might want a graphical representation (PNG data showing the evolution of the price over time), a human-readable representation (HMTL data) or a machine-understandable representation (XML or JSON data).
- Thirdly, the RFC explains that HTTP supports **content negotiation** between clients and server (and describes how special headers provide the actual mechanism). We will get back to this notion in the next paragraph.

**Here is an excerpt from the RFC, in the Terminology Section:**

```
message
   The basic unit of HTTP communication, consisting of a structured
   sequence of octets matching the syntax defined in section 4 and
   transmitted via the connection.

request
   An HTTP request message, as defined in section 5.

response
   An HTTP response message, as defined in section 6.

resource
   A network data object or service that can be identified by a URI,
   as defined in section 3.2. Resources may be available in multiple
   representations (e.g. multiple languages, data formats, size, and
   resolutions) or vary in other ways.

entity
   The information transferred as the payload of a request or
   response. An entity consists of metainformation in the form of
   entity-header fields and content in the form of an entity-body, as
   described in section 7.

representation
   An entity included with a response that is subject to content
   negotiation, as described in section 12. There may exist multiple
   representations associated with a particular response status.

content negotiation
   The mechanism for selecting the appropriate representation when
   servicing a request, as described in section 12. The
   representation of entities in any response can be negotiated
   (including error responses).
```

**Source**: Section 1.3

## 4.2. Content Negotiation

Content negotiation applies to resource representation and works on the following principle:

- When sending a request to get the representation of a given resource, **the client expresses its capabilities and preferences** (which partly depend on the capabilities and preferences of the end-users behind the client). For instance, it may say: my preference would be to receive a JSON representation of the resource, but I could also do something with an XML or plain text representation. Or it could say: my preference would be to receive a representation of this resource in french, but I could also deal with a representation in english or in german.

- When the server processes the request, it will **try to do its best to satisfy the client**. However, not it will not be able to fulfill the first choice of the user. As the outcome is uncertain, the server will indicate what he has been able to do when sending back the response.

In both cases, **content negotiation is achieved thanks to special headers**, which we will look at shortly when describing the syntax of HTTP requests and responses.

## 5. HTTP Messages: Requests and Responses

Now that we have seen what a typical HTTP exchange looks like and that we have presented important concepts underlying the protocol design, let us look at some details. Namely, let us look at the syntax of requests and responses, as well at some of the methods, headers and status codes defined in the RFC.

## 5.1. Syntax

The syntax of HTTP messages is specified in BNF form. We will only describe some of its elements here, you will find all details in the RFC.

The high-level constructs are defined by the following rules:

```
HTTP-message   = Request | Response     ; HTTP/1.1 messages

Request        = Request-Line              ; Section 5.1
                 *(( general-header        ; Section 4.5
                  | request-header         ; Section 5.3
                  | entity-header ) CRLF)  ; Section 7.1
                 CRLF
                 [ message-body ]          ; Section 4.3


Request-Line   = Method SP Request-URI SP HTTP-Version CRLF

Response       = Status-Line               ; Section 6.1
                 *(( general-header        ; Section 4.5
                  | response-header        ; Section 6.2
                  | entity-header ) CRLF)  ; Section 7.1
                 CRLF
                 [ message-body ]          ; Section 7.2

Status-Line    = HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

**Source**: Section 4.1, Section 5 and Section 6.

**Here are some comments about these definitions:**

- The requests and responses have the same basic structure: there is a **first line**, followed by a variable number of **header lines**, followed by a **blank line**, followed by a **message body** (*note: the message body is sometimes empty*).
- The first line of the **request** includes a **method** (which indicates what the client wants to do with the target resource) and a **URI** (which identifies the target resource).
- The first line of the **response** includes a **status code** and a **reason phrase** that inform the client of the outcome (i.e. whether the server was able to fulfill the request).
- The **header lines** start with a **header name,** followed by a **colon**, followed by a **header value**.
- The sequence of `CR LF` (ASCII(13) ASCII(10) or `\r \n`) is used to **separate lines**.
- The message body may contain **binary data** (e.g. image), so it MUST NOT be read line by line.
- The **protocol version** is indicated both in the request and in the response, on the first line.

## 5.2. Some Interesting Methods

The first line in an HTTP request contains what is called a **method**. We have said that the method is used to indicate what the client intends to do with the target resource. The HTTP specification defines a number of methods, but also suggests that other protocols may extend this list with their own methods (which is an approach that has been taken by the WebDAV protocol for instance).

The default HTTP methods are listed in Section 5.1.1 and fully specified in Section 9:

```
Method         = "OPTIONS"                ; Section 9.2
               | "GET"                    ; Section 9.3
               | "HEAD"                   ; Section 9.4
               | "POST"                   ; Section 9.5
               | "PUT"                    ; Section 9.6
               | "DELETE"                 ; Section 9.7
               | "TRACE"                  ; Section 9.8
               | "CONNECT"                ; Section 9.9
               | extension-method
extension-method = token
```

**Source**: Section 5.1.1.

From this list, the most important ones are the following:

- `GET`, which is used by the client to request (i.e. read) the representation of the target resource. When you type a URL in the browser navigation bar or when you click on a hyperlink, this is what is sent to the server. The request body is empty.
- `POST`, which is used by the client to send data to the target resource. When you fill out an HTML form and hit the submit button, a POST request is issued and the content of the form is sent as the request body. (Note: we have mentioned REST APIs before; without digging into the details, POST requests are used to create data).
- `PUT`, which is used by the client to send the representation of a resource in order to replace the target resource. This is not something that you can typically do with a web browser, but which is used by other types of HTTP clients (in REST APIs, PUT requests are used to update data).
- `DELETE`, which is used by the client to request the deletion of the target resource (again, this is not something that you can usually do in a web browser but that that other types of clients can do).

To be complete, we should also mention that an additional method, PATCH, has been defined in RFC 5789. It is used to do partial updates of resources and its usage is growing (because of the popularity of REST APIs and of related needs).

## 5.3. Some Interesting Headers

The standard HTTP headers are described in Section 14. We will only describe some of them, in relation to the key protocol concepts that we have presented before.

- the `Accept` request header and the `Content-type` header are used for **one aspect of the content negotiation**. In the `Accept` header, the client indicates the media type that it is able to handle (e.g. `text/html`, `text/plain`, `application/json`, `image/png`). In the `Content-type` header, the server indicates what it has been able to do and how the message body should be interpreted.
- the `Accept-Language` and `Content-Language` headers are used for **another aspect of the content negotiation**, allowing the client to indicate a preference for a representation in French and allowing the server to indicate that it is sending a representation in Schwyzerdütsch.
- the `Accept-Charset` and the `Content-type` headers are used for **yet another aspect of the content negotiation**, namely the preferred character encodings.
- the `Content-Length` header is used to indicate the **length of the message body**. This allows the client or the server receiving the message to know how many bytes must be read before reaching the next message. Note that this header may not always be

provided, we will get back to this when looking at the ways to parse HTTP messages (see Section 4.4 in the RFC).

- the `Host` header, which is mandatory since HTTP 1.1 and which is extremely important to enable **virtual hosting**. We will get back to the notion of virtual hosting in an upcoming lecture.
- the `Authorization` header, which is used when accessing **protected resources**. We will get back to security aspects in an upcoming lecture.
- the `Transfer-Encoding` header, which is used to indicate that the message body has been transformed in order to transfer it in a particular way. For instance, it is possible for the message body to be compressed before being transferred. Also, in the case of dynamic content, the message body may be structured in multiple *chunks* (more on this later).
- the `If-Modified-Since`, `Last-modified`, `ETag`, `Expires` headers that are related to caching, a notion that we will look at in an upcoming lecture.

Note: a nice summary of HTTP headers, with references to the HTTP RFC is available here.

## 5.4. Some Interesting Status Codes

The status codes that are used by the server to indicate whether the request could be fulfilled or not are listed in Section 6.1.1 and documented in Section 10.

```
The first digit of the Status-Code defines the class of response. The
last two digits do not have any categorization role. There are 5
values for the first digit:

   - 1xx: Informational - Request received, continuing process

   - 2xx: Success - The action was successfully received,
     understood, and accepted

   - 3xx: Redirection - Further action must be taken in order to
     complete the request

   - 4xx: Client Error - The request contains bad syntax or cannot
     be fulfilled

   - 5xx: Server Error - The server failed to fulfill an apparently
     valid request

Status-Code    =
      "100"  ; Section 10.1.1: Continue
    | "101"  ; Section 10.1.2: Switching Protocols
    | "200"  ; Section 10.2.1: OK
```

```
     | "201"  ; Section 10.2.2: Created
     | "202"  ; Section 10.2.3: Accepted
     | "203"  ; Section 10.2.4: Non-Authoritative Information
     | "204"  ; Section 10.2.5: No Content
     | "205"  ; Section 10.2.6: Reset Content
     | "206"  ; Section 10.2.7: Partial Content
     | "300"  ; Section 10.3.1: Multiple Choices
     | "301"  ; Section 10.3.2: Moved Permanently
     | "302"  ; Section 10.3.3: Found
     | "303"  ; Section 10.3.4: See Other
     | "304"  ; Section 10.3.5: Not Modified
     | "305"  ; Section 10.3.6: Use Proxy
     | "307"  ; Section 10.3.8: Temporary Redirect
     | "400"  ; Section 10.4.1: Bad Request
     | "401"  ; Section 10.4.2: Unauthorized
     | "402"  ; Section 10.4.3: Payment Required
     | "403"  ; Section 10.4.4: Forbidden
     | "404"  ; Section 10.4.5: Not Found
     | "405"  ; Section 10.4.6: Method Not Allowed
     | "406"  ; Section 10.4.7: Not Acceptable
     | "407"  ; Section 10.4.8: Proxy Authentication Required
     | "408"  ; Section 10.4.9: Request Time-out
     | "409"  ; Section 10.4.10: Conflict
     | "410"  ; Section 10.4.11: Gone
     | "411"  ; Section 10.4.12: Length Required
     | "412"  ; Section 10.4.13: Precondition Failed
     | "413"  ; Section 10.4.14: Request Entity Too Large
     | "414"  ; Section 10.4.15: Request-URI Too Large
     | "415"  ; Section 10.4.16: Unsupported Media Type
     | "416"  ; Section 10.4.17: Requested range not satisfiable
     | "417"  ; Section 10.4.18: Expectation Failed
     | "500"  ; Section 10.5.1: Internal Server Error
     | "501"  ; Section 10.5.2: Not Implemented
     | "502"  ; Section 10.5.3: Bad Gateway
     | "503"  ; Section 10.5.4: Service Unavailable
     | "504"  ; Section 10.5.5: Gateway Time-out
     | "505"  ; Section 10.5.6: HTTP Version not supported
     | extension-code
```

Some comments can be made:

- remember that a status code starting with **4** indicates that the client responsible for the error, while a status code starting with **5** indicates that the server is responsible.
- If everything went well, you will get a **200**.
- **401** and **403** are related to security. **401** is used when the server does not know the identity of the client or user and asks for **authentication**. **403** is used when the server knows the identify of the client or user but decides that the access should not be **authorized**.

- **301** and **302** are used to indicate that the target resource has moved. The client should send a new request, using the URL provided in the `Location` header.
- **304** is used to indicate that the resource has not been modified since last request. Therefore, the server is not sending any representation and the client can reuse what it has previously received.

## 6. Parsing HTTP Messages

Whether you are implementing a HTTP server or a HTTP client, you will need to parse HTTP messages. In other words, you will receive a series of bytes from a socket and will need to convert it into either an HTTP request or an an HTTP response. The procedure is pretty straightforward, but there are a couple of possible variations:

- You should **read bytes, not characters** (because the payload may be binary data and you would not want this data to be interpreted as characters and modified).
- When **reading line by line** (at the beginning of the process), you should read bytes until you find the sequence of CR (ASCII 13, `\r`) and LF (ASCII 10, `\n`) bytes.
- You should start by **reading the first line** (request line or status line). You should extract the tokens (method, URI, protocol version, status code or reason phrase).
- You should then **read all of the headers**, line by line, until you find an empty line (in other words, until you find a sequence of CR LF CR LF bytes). You should keep these headers in a data structure (e.g. a map).
- Depending on the situation, the next step will vary (the details are specified in Section 4.4 of the RFC 2616):
  - If you have found a `Content-length: n` header, then you have to read `n` bytes from the input stream.
  - If you have found a `Transfer-Encoding: chunked` header, then you have to start by reading the size of the first chunk or response. You will find it on the first line, in hexadecimal value. You will then read this amount of bytes, before finding a new line with the size of the next chunk. You will go through this process until you find a chunk size of 0. The RFC provides pseudo code for this algorithm in Section 19.4.6 (note that headers might appear after the last chunk, which we will not discuss here to keep things simple).
  - If you don't have any indication about the length of the message, then you have to assume that the process on the other side will close the TCP connection when it has sent all of the content (this is how HTTP 1.0 servers used to work). Therefore, you should read bytes until you detect the end of the stream.

    - The RFC also specifies a special case when the `multipart/byteranges` media type is used, but we will not discuss it here. The details are available in Section 19.2.

# Resources

## MUST read

- These Sections of the RFC 2616:
    - Section 1
    - Section 4 and in particular Section 4.4
    - Section 19.3
    - Section 5
    - Section 6
    - Section 8.1.1
    - Section 8.1.2
    - Section 19.3
    - Section 19.4.6

## Additional resources

- The rest of the HTTP RFC
- History of the HTTP protocol specification
- History of the Web, going back to its roots

# What Should I Know For The Test and The Exam?

Here is a **non-exhausive list of questions** that you can expect in the written tests and exams:

- Write a complete HTTP request and a complete HTTP response. Explain the structure and the role of the different lines and of the different elements on each line.
- Explain **how** and **why** TCP connections are handled differently in HTTP/1.0 and in HTTP/1.1.
- Explain the notion of **content negotiation** in HTTP and illustrate it with an example (include an HTTP request and an HTTP response in your explanation).

- What does it mean when we say that HTTP is a **stateless** protocol? Why is that sometimes a problem (give an example of an application where this is a problem) and what can be done about it?
- Why is the **chunked transfer encoding** useful (what would be the problem if we did not have it?)? Explain what a client needs to do in order to parse a response that uses the chunked transfer encoding.
- What are **cookies** in the context of the HTTP protocol? Explain why they are useful. Explain how they work at the protocol level (explain what the client and the server have to do, give examples of requests and responses).

# Lecture 5: Web Infrastructures


- Objectives
- Lecture
  - 1. Introduction
  - 2. Non-Functional Requirements, aka Systemic Qualities
    - 2.1. Performance
    - 2.2. Scalability
    - 2.3. Availability
    - 2.3. Security
  - 3. HTTP Reverse Proxies and Load Balancers
  - 4. Virtualization and Provisioning Technologies
  - 4.1. Virtual Box: *OS Virtualization*
  - 4.2. Vagrant: *Development environments made easy*
  - 4.3. Docker: *Lightweight containers for your apps*


## Objectives

Until now, we have focused on two aspects of the HTTP protocol (and of application-level protocols in general). Firstly, have looked at what **developers** need to do in order to implement protocols in client and server programs (how to use classes and methods for managing IOs, how to use the socket API, etc.). Secondly, we have looked at the protocol **specification** and at the rules that it defines (syntax and semantics of messages, connection and state management, etc.).

In this lecture, we will look at HTTP from another perspective, namely the **infrastructure** perspective. As you can imagine, a company running a service (whether this service is a content publication web site or a web application) on top of HTTP is unlikely to *just* install a single HTTP server and let it handle all the traffic. Think about the CNN web site or the local e-commerce service. **Because of various constraints (security, performance, availability, scalability), you can imagine that the infrastructure enabling these services is a bit more sophisticated than a single HTTP server running on a single machine directly accessible from the Internet.**

The objective of this lecture is to **look at the building blocks of a typical web infrastructure** and to show how they can be put together and configured. In particular, we will look at the role of HTTP reverse proxies and show that they are useful for various reasons. Another objective of this lecture is to **introduce some modern virtualization and**

**infrastructure management technologies**, which we will use to create a multi-nodes setup on our machines. We will explain how different technologies complement each other and use them in the lab.

# Lecture

## 1. Introduction

In this lecture, we will cover three topics:

- We will start by introducing the notions of *non-functional requirements* and of *systemic qualities*. Note that these notions are not specific to HTTP, nor to web infrastructures, but that they apply to any kind of service or system. We will take a closer look at some specific systemic qualities, namely **scalability**, **availability**, **performance** and **security**.
- We will then describe a typical web infrastructure and see how it consists of several nodes. In particular, we will describe the role of **reverse proxies** and of **load balancers**. We will explain that introducing these components in the infrastructure is a way to address several non-functional requirements (again, we will talk about scalability, availability, performance and security).
- Finally, we will introduce three **virtualization** and **infrastructure management** technologies: **Virtual Box** (which you certainly already know), **Vagrant** (which makes it easier to work with Virtual Box in a repeatable way) and **Dockers** (which allows you to run lightweight containers in your Virtual Box VM). In the lab, we will use Dockers to create a web infrastructure with multiple nodes (reverse proxy, load balancer, web servers, application servers), all running on your machine.

## 2. Non-Functional Requirements, aka Systemic Qualities

In every IT project, there are **two types of requirements**. Firstly, there are **functional requirements**, which specify *what* the system should do, i.e. what *features* it should offer to users. Secondly, there are **non-functional requirements**, which specify *how* the system should behave and what qualities it should exhibit.

**To illustrate this notion with an example, think about an e-commerce web site:**

- The **functional requirements** specify the **features** that should be offered to **customers** ("I should be able to browse products", "I should be able put products in my shopping cart", "I should be able to checkout and pay for my purchases"). The functional requirements also specify the features that should be offered to other types of users, including **marketing experts** ("I should be able to get statistics about sales"), **IT**

**staff** ("I should be able to see how many users have accessed the service in the last 2 hours"). You may be familiar with the notions of **use cases** or **user stories**, commonly used in software development methodologies to document functional requirements.

- The **non-functional requirements** specify various constraints or characteristics for the system behavior. For instance, a requirement might specify that "at least 1'000 users should be able to use the service concurrently". A requirement might specify that "the maximum down-time (i.e. unavailabilty) for the service is 5 minutes per year". A requirement might specify that "the average response time should be less than 600 ms and the maximum response time should be less than 2000 ms". A requirement might specify that "two factors should be used to authenticate users accessing the service back-office interface". As you can see, non-functional requirements cover different areas, including scalability, availability, performance and security.

In english, **many systemic qualities have names ending with the *-ilty* suffix**: availab-*ilty*, scalab-*ilty*, exensib-*ilty*, usab-*ilty*, testab-*ilty*, maintainab-*ilty*, etc. For this reason, people often use the word "**ilities**" as a synonym for non-functional requirements. **Systemic quality** is another synonym for the same concept.

We will not discuss all systemic qualities here and will only consider a few of them, namely performance, scalability, availability and security. We will start by defining each of these qualities and later show how they can be addressed in a typical web infrastructure.

## 2.1. Performance

Performance is a rather generic term and can mean different things. Very often, two dimensions of performance are considered when specifying non-functional requirements:

- The first one is the **response time** experienced by users. For web applications, it is generally measured in milliseconds. Specifying target **response times** for a web application can be a bit tricky, especially if the application is somewhat complex. The same requirements may not apply across all features (for an e-commerce service, think of a feature used by customers while they shop, vs a feature used by company employees once in a month). Hence, **non-functional requirements are often specified in the context of a particular use case**. In other words, if you have a use case or a user story of the form "As a customer, I want to add a product in my shopping cart", then it is a good practice to add a note about the performance expectations directly in the use case documentation (e.g. "The average response time for this use case should be less than 400 ms"). When dealing with response time, you also need to consider network latency. If your service is used globally, with users in different countries, you may have to distribute your infrastructure if you have aggressive requirements.

- The second one is **throughput**, which measures the number of requests that can be processed within a given time frame. For instance, you may specify that "The service should sustain a load of at least 1'200 requests per second". In some cases, you may specify additional constraints, by specifying resource consumption requirements ("The service should sustain a peak load of 2'000 requests per second with a CPU consumption of max 80% per node") or response time requirements ("The service should sustain a peak load of 2'000 requests per second and offer an average response time below 1000 ms").

## 2.2. Scalability

Intuitively, scalability is often associated with the idea of "a large number of users" or "a lot of requests to handle". **But scalability is not defined as the ability to handle a heavy load!** Rather, scalability is defined as the ability to **evolve** in order to handle a growing load (in an efficient manner).

To illustrate this important difference, let us consider the example of a photo sharing service developed by a start-up:

- During the development and pilot phases, the start-up will certainly want to save money and to deploy an infrastructure that is as cheap as possible. This will translate into a small number of servers, CPUs, memory, etc. Maybe it will consist of a single virtual machine.
- That infrastructure may perhaps only handle 10 concurrent users. This is not a problem and this does **not** mean that the service is not scalable!
- At some point, the service becomes popular and the current infrastructure cannot handle the load anymore. It is at this point that we can evaluate to what extent the system is scalable:
    - **if it is not scalable at all**, then we have a major problem. Even with a lot of money, it is not possible to add resources to the system. It is not possible to accept new users. The start-up may have a great service, but is unable to make a business out of it. Quite a few companies have experienced this situation, often with monolithic applications than cannot be distributed.
    - **If it is poorly scalable**, then it means that it is possible to evolve the system in order to adapt to the growing load, but not in a very efficient manner. Two elements should be considered here: **time** and **money**. In other words, when you evaluate the scalability of a system, you should ask yourself two questions: "If I need to handle a load that is twice as big as today, how much will I have to pay?" and "If I need to handle a load that is twice as big as today, how much time will I need to

update my system?". If a system is poorly scalable, then it will become more and more expensive to sustain a growing load. If a system is poorly scalable, then it will take a lot of time (hours, if not days or weeks) to add resources (think about the difference between having to order physical servers and being able to provision virtual machines in a cloud environment).

    - As you can guess, **if it is highly scalable**, then it means that it will always be possible to extend the capacity of the system by adding extra resources (CPU, memory, etc.), ideally in a linear fashion. Furthermore, adding extra capacity should be possible in short time frames (minutes, if not seconds). In an ideal solution, the system should be able to scale in both directions and automatically: depending on the variations of the traffic (hourly and weekly peaks, special events, etc), the amount of resources should be adapted (sometimes by adding resources, sometimes by removing services). This is what some people call *elasticity*.

There are two general approaches to scalability: **vertical** and **horizontal** scalability:

- **Vertical scalability** means that the capacity of the system is increased by **adding resources to a single node**. Imagine that you have installed a database management system on a single server. If you decide to add more CPUs, more memory, more disks, more network interfaces to this server in order to adapt to a growing load, then you are scaling up your system vertically.
- **Horizontal scalability** means that the capacity of the system is increased by **adding more nodes** to the system. Imagine that you have installed a web application on a single server. If you decide to install the same web application on multiple servers and find a way to distribute the load between all of these similar nodes, then you are scaling up your system horizontally.

*Note that these two approaches are not mutually exclusive and in many situations, they are combined for different parts of the infrastructure.*

## 2.3. Availability

*Allo…? allo…? Anyone on the line???*. The availability of a service is **expressed as a percentage, for a given period**. It is computed by dividing the time when the service was operational (i.e. when its features could be used as expected) by the total time.

**Let us illustrate this idea with an example:**

- Let's consider a period of one week (or 7 days or 168 hours or 10'080 minutes).
- Let's imagine that during this week, the service could not be used by its users because:
    - the operating system of the server was patched (15 minutes)

    - a new release of the service was deployed (25 minutes)
    - a network switch had a hardware problem and had to be changed (4 hours) and
    - a bug in the code lead to an infinite loop and a fix had to be developed, tested and deployed (12 hours).

**As a result, the service was unavailable for 15 + 25 + 240 + 720 = 1'000 minutes. Hence, the service had an availability of (10'080 - 1'000) / 10'080 = 90.08%.**

The first thing to be aware of, when specifying availability requirements, is that there are two types of service outages (a service outage occurs when the service is not available to its users):

- Firstly, there are **unplanned outages**, which are incidents that happen by surprise. Think of an earthquake destroying your data center, a denial-of-service attack bringing down your web site, a bug crashing your application or a hard-disk failing.
- Secondly, there are **planned outages**, which happen in situations that you can control. When you decide to bring down your service because you want to patch your operating system or because you want to deploy a new release of your application, you know it in advance. Not only is that useful because you can warn your users (and in some cases, select a period that will impact as few users as possible), but also because you may exclude these outages from your **service level agreements**.

The second thing to be aware of, when defining your strategy to meet availability requirements, is that **it is not only a matter of technical design, but also of organization and processes**. On one hand, the availability will depend on the reliability of all elements in your system (i.e. how often do these elements fail and cause an outage). On the other hand, the availability will also depend on the time you need to restore the service (i.e. how quickly can you recover from the outage). This can be expressed in the following formula:

```
Availability = (MTBF) / (MTBF + MTTR)

where MTBF = Mean Time Between Failures and
      MTTR = Mean Time To Repair

Exemple: the service fails every 24 hours and take 1 hour to be restored
   A = 24 / (24 + 1) = 96%
```

To illustrate this idea, let us consider the situation where your service is a web application, developed in PHP. Let us compare three situations:

- In the first situation, you are running your application on a single physical server, installed in your office. One day, the power supply fails and your server goes down - the unavailability clock kicks off… You have to order a new power supply. After placing the

order, you will have to wait at least for one day to get it. Still need some time to install it in your server, reboot, restart the services and check that everything works well. Not great.

- In the second situation, you are still running your application on a single physical server. But learning from your past mistakes, you have decided to invest a bit of money in higher quality hardware. Your server now has two power supplies. Like in the previous scenario, one of them dies. Not a problem! The other is still alive and your service is still available. You are carefully watching system notifications, so you know that you need to order a new power supply to replace the faulty one. Let us imagine that you are very unlucky and that the two power supplies die at the same time. Again, because you have learned about your past mistakes, you keep a stock of spare parts (extra disks, extra RAM, extra power supplies). Your service may go down of some time, but you will not need to wait 2 days before being able to replace the faulty component.
- In the third situation, you are now running your application on several servers. A load balancer, in front of them, distributes the requests between the servers. If one of them fails (again, perhaps because of a hardware problem), the incident is transparent to users. The other servers are there and remain operational. The load balancer itself should not be a single point of failure, so it should be implemented with some redundancy.

As these scenarios suggest, the processes that you put in place are very important if you want to be able to recover quickly from a service outage. Typical questions that you should ask yourself incluce:

- *How am I going to notice (be alerted) that the service is down?*
- *When the service is down, how much time is needed until someone can look at the incident?(i.e. are there people on call?)*
- *When the service is down, is there a checklist and documented procedure to follow in order to do a diagnostic for the outage?*
- *If we run on our own hardware, do we have a stock of spare parts?*
- *Do we have a disaster recovery plan?*

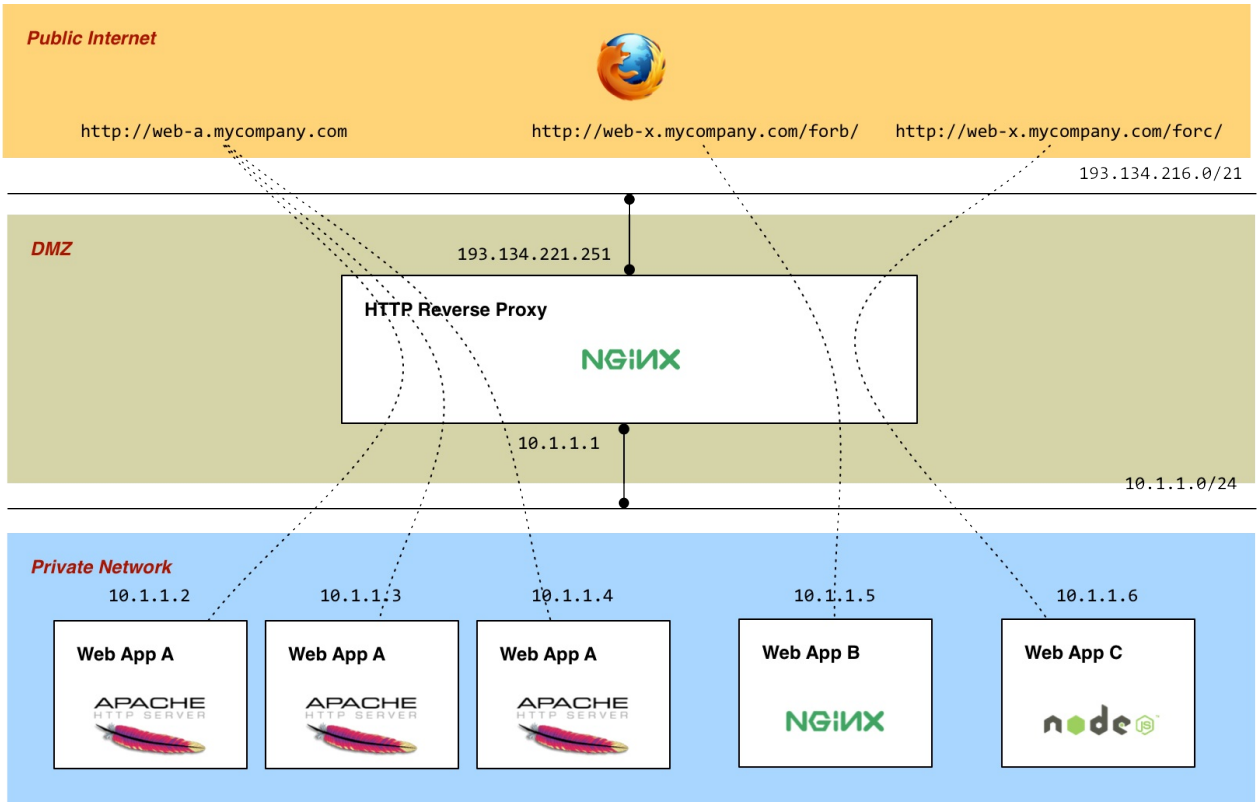
## 2.3. Security

Security is another systemic quality and one that has many different aspects. Just think about the needs that you may express around confidentiality, authentication, authorization, traceability, protection against denial-of-service attacks. A coverage of these questions goes beyond the scope of this lecture, but we will later show that introducing a reverse proxy in a web infrastructure contributes to making it more secure.

# 3. HTTP Reverse Proxies and Load Balancers

Now that we have introduced the notion of **systemic quality**, let us see how they can be addressed in the design of typical web hosting infrastructure. By web hosting infrastructure, we refer to a set of machines (or nodes) which make it possible to deploy a web site or a web application. Usually, this means that both static and dynamic content is made available to end-users.

The simplest form of web infrastructure consists of a single machine, directly accessible from the Internet, on top of which an HTTP server has been installed and is running. Once again, this is not realistic for real applications. A typical infrastructure looks more like this:




Here are some points about the infrastructure:

- The key point is that the HTTP clients do not talk directly to the HTTP server(s). As you can see, they go through an intermediary, which in this case is the element labeled "**HTTP Reverse Proxy**". A reverse proxy is a software component that implements the HTTP specification and that plays at the same time the role of client and of server. On one hand, it receives requests from the *real* clients (i.e. the browsers). On the other hand it forwards the requests to the appropriate server (it decides which is the appropriate server by applying a set of rules).
- Notice that the reverse proxy is deployed in the **DMZ**, i.e. between two firewalls. This means that external clients cannot send requests directly to the servers located in the private network. They **have to** go through the reverse proxy, which is obviously a good

thing from a security point of view.

- In this example, the reverse proxy performs two different, complementary functions:
  - Firstly, **it knows that the infrastructure hosts three different applications** (Web App A, Web App B and Web App C). When it receives a request from an external browser, it inspects the target URL and **decides to which server (or group of servers) it should forward the request**. In some cases, it makes the decision based on the sub-domain in the hostname (e.g. `web-a` in `web-a.mycompany.com`). Sometimes, it makes the decision based on a prefix in the path of the URL (e.g. `/forb/` vs `/forc/` in `http://web-x.mycompany.com/forb/`).
  - Secondly, it knows that one of the applications, namely Web App A, has higher requirements in terms of availability and/or performance. It knows that, for this reason, the application has been deployed on three equivalent servers. The role of the reverse proxy is to distribute the client requests between these three servers (i.e. it performs the function of HTTP **load balancing**).
- In some infrastructures, the functions of **forwarding** and of **load balancing** may be implemented by different components. Furthermore, they may be implemented by hardware and/or software components (it is quite common to encounter hardware load balancers). Also, as mentioned before, the reverse proxy should not be a single point of failure. For this reason, it is quite common to deploy two reverse proxies with the same configuration and to distribute the requests between them. Should one reverse proxy suffer from an outage, the other is able to keep the infrastructure operational.
- In the case of **load balancing**, special care as to be taken for stateful web applications. Remember that while HTTP is a stateless protocol, many applications built on top of HTTP are not. Think about e-commerce web sites, where the content of the shopping cart is maintained between successive requests. Think about web applications, where users have a login/logout feature. There are many different places where the state of a user session can be stored and we will not describe all of them. Very often, it is stored in the application server (i.e. in the PHP engine, in the Java server, etc.) and is **not** distributed. This means that in our example, every server hosting Web App A is responsible to manage its own session store (with the associated state). Let us consider the following elements:
  - When the client makes its first request, it is perfectly ok for the reverse proxy to choose **any** of the servers in the group (this is done in a round-robin fashion, or possibly by applying a more sophisticated set of rules).
  - The server will initiate the session, generate a session id, possibly store some initial data in the session store and send back a cookie with the session id.

    - When the client makes the second request, the reverse proxy **cannot select an arbitrary server** anymore. Indeed, if it does not select the one that served the first request, then the selected server will simply not know about the user session: it will be the first time that it sees that particular client. In this scenario, the user would experience something like *"Giving credentials and be logged in. Having to provide credentials again"* or *"Putting products in a shopping cart. Seeing an empty cart again"*.
    - To avoid this situation, the **reverse proxy should therefore send all requests issued by one client to the same server** (again, because in this scenario we assume that the session store is not distributed nor shared between servers). This is something that is called **session stickiness**. It is often implemented with cookies, either with the cookies that are generated by the servers (i.e. the reverse proxy uses the application session id to route the requests), or with specific cookies that are added by the reverse proxy itself. Session stickiness can also be implemented by other means, for example based on the IP address of the client (we will use this in the nginx configuration).

- As mentioned before, the reverse proxy can be implemented in different ways, either in software or in hardware. Two popular software implementations are the **apache httpd server** and the **nginx server**. The former has been used for a very long time and is encountered very often, the latter exhibits very good performance and resource usage metrics.
- The configuration is fairly easy to do and is documented here for apache httpd and here for nginx. It mostly consists of defining the rules for doing the routing.
- If, in addition to the routing function, you need the load balancing function, you will find documentation for apache http here and for nginx here.

## 4. Virtualization and Provisioning Technologies

The previous diagram, which shows a typical web hosting infrastructure, focuses on the logical system architecture. In other words, it does not mean that every rectangle is implemented on a dedicated hardware server. That might be the case in some cases, but in others some rectangles might be implemented as virtual machines, lightweight OS containers or event simple virtual hosts.

Nowadays, many infrastructures are built on top of a public or private cloud environment. Hence, to the IT engineers, the boxes are most likely to appear either as virtual machines, as lightweight containers running in these virtual machines or as virtual hosts.

In this lecture, and in the associated lab, we will work with some associated technologies. Note that some of them are very new (and not yet recommended for a usage in production). We will use these technologies not only because they are becoming part of the standard *toolbox* that any IT engineer should master, but also because it makes it easier to observe consistent behavior across heterogenous host machines and operating systems.

## 4.1. Virtual Box: *OS Virtualization*

> VirtualBox is a powerful x86 and AMD64/Intel64 virtualization product for enterprise as well as home use. Not only is VirtualBox an extremely feature rich, high performance product for enterprise customers, it is also the only professional solution that is freely available as Open Source Software under the terms of the GNU General Public License (GPL) version 2.

You are probably already familiar with the first tool that we will use, namely Virtual Box. Virtual Box is one of the virtualization technologies (VMWare is another one), which allows you to run a virtual machine with a *guest operating system* on top your *host operating system*. In other words, Virtual Box allows you to run a Linux virtual machine on top of your Windows laptop. Or to run a Windows virtual machine on top of your Mac OS laptop.

Virtual Box has the advantage to be open source and free. It also supports a wide range of guest host and guest operating systems.

## 4.2. Vagrant: *Development environments made easy*

> Vagrant provides easy to configure, reproducible, and portable work environments built on top of industry-standard technology and controlled by a single consistent workflow to help maximize the productivity and flexibility of you and your team.
>
> To achieve its magic, Vagrant stands on the shoulders of giants. Machines are provisioned on top of VirtualBox, VMware, AWS, or any other provider. Then, industry-standard provisioning tools such as shell scripts, Chef, or Puppet, can be used to automatically install and configure software on the machine.

The way you have worked with Virtual Box in the past is probably as follows. You have started from a base virtual machine image, have manually installed a number of software components, modified some of the operating system and application configuration files, written some scripts, etc. You may have used the *snapshot* feature to save the state of your virtual machine at various stages.

This has probably worked fine, but you may have been through the same lengthy process over and over again. Essentially, for each new project, you may have created a new virtual machine, reinstalled the same software, done the configuration again, etc. Not only is that a

waste of time, but it is also error prone and can lead to hard-to-detect differences.

Furthermore, if at some point you had deploy what you had running in a virtual machine running on your laptop in a cloud environment (such as Amazon EC2), then you had to redo the setup once again.

Vagrant is an attempt to address this problem. It gives you a way to declare what are the steps required to build a virtual machine with a given setup. You start by doing a `vagrant init`, editing a special file named `Vagrantfile` and doing a `vagrant up`. This will download what Vagrant calls a "box", ask Virtual Box to start the resulting virtual machine, go through basic configuration steps, initiate the provisioning process (which can be done in various ways, including with Docker). At this point, you will be able to do a `vagrant ssh` to connect to you virtual machine from the command line (you should not expect to work with a GUI, event if it is possible).

## 4.3. Docker: *Lightweight containers for your apps*

> Docker is an open-source project to easily create lightweight, portable, self-sufficient containers from any application. The same container that a developer builds and tests on a laptop can run at scale, in production, on VMs, bare metal, OpenStack clusters, public clouds and more.

The last technology that we will use is named Docker and works in Linux environments. We use Vagrant to configure and start our Linux virtual machine. We use Docker within this virtual machine to run several logical, isolated nodes (corresponding to the rectangles in the infrastructure diagram).

Docker is quite different from other virtualization technologies and is very lightweight. Usually, **a Docker container** hosts a single service (an apache httpd service, a nginx service, a mysql service, etc.). A **container is a running instance of a Docker image**, which are templates that cover both operating system and application elements. Docker images can be shared in **docker registries** (which can be public or private).

When using Docker, you have a command line utility that you use to manage your images and your containers (you can list them, delete them, start and stop them). One thing to be aware is that every time you launch a container, you run a single command (maybe it is the apache httpd daemon, maybe it is a shell). The state of the container is the same each time to create a new container. The documentation tells us important facts about state management (and about the relationship between containers and images):

> Containers can change, and so they have state. A container may be running or exited.
>
> When a container is running, the idea of a "container" also includes a tree of processes running on the CPU, isolated from the other processes running on the host.
>
> When the container is exited, the state of the file system and its exit value is preserved. You can start, stop, and restart a container. The processes restart from scratch (their memory state is not preserved in a container), but the file system is just as it was when the container was stopped.
>
> You can promote a container to an Image with docker commit. Once a container is an image, you can use it as a parent for new containers.

# Lecture 6: LDAP


- Objectives
- Lecture
  - The LDAP Data Model
  - The LDAP Protocol
  - Software Components
  - Searching for Information in the Directory
  - Importing and Exporting Directory Information


## Objectives

Almost every company or organization is managing information about users (employees, members, customers, etc.). At the very least, security credentials and contact information are stored in some kind of *identity store*, so that a user can be authenticated when accessing an network service.

The standard approach to manage this type of information is to deploy a server that implements the LDAP protocol (such as Microsoft Active Directory or OpenLDAP). Many applications (messaging applications, business applications, web servers, etc.) can be configured to delegate the authentication of users to a server that *talks LDAP*. Each time that you are prompted your HEIG-VD user id and password, these credentials are verified by the directory service that has been deployed at the School.

As a matter of fact, servers that implement the LDAP protocol can be used to store other types of information as well. For instance, they are often used to manage information about devices (printers, file servers, etc.) or about network services (web service endpoints, etc.).

For all of these reasons, it is important for you to understand the core concepts underlying the LDAP data model and protocol. It is also important to get some practical experience with some LDAP tools, both on the client and on the server side. This is what we will aim to achieve in this part of the course.

Here are the objectives for this lecture:

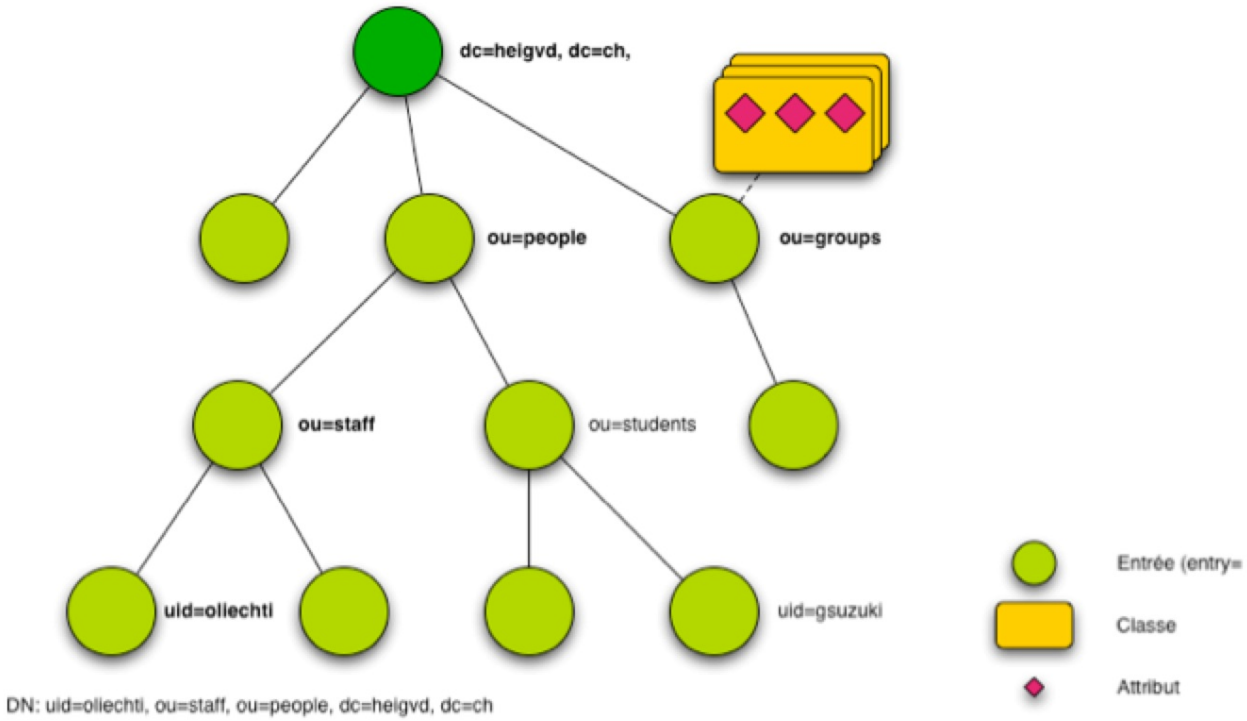
- Learn how information is **hierarchically organized** in a LDAP server, i.e. become familiar with the notion of **Directory Information Tree** (DIT). Be able to explain what *object classes* and *attributes* are in the context of the LDAP specifications. Understand the role of the *LDAP schema*.

- Be able to describe the key **characteristics** and **operations** defined in the **LDAP protocol**. Be able to explain how LDAP clients can connect with LDAP servers and how they can interact with each other. Learn how to use **LDAP filters to query information** stored in a directory. Learn about **dynamic groups** and how to use them.
- Understand the role of the **LDIF data format**, become familiar with its syntax and learn how to use it to perform **import and export** operations.
- Learn how to use various **software components** that implement the LDAP protocol: a **directory server** (OpenDJ), a **directory browser** (Apache Directory Studio) and **command line tools** (e.g. `ldapsearch`).

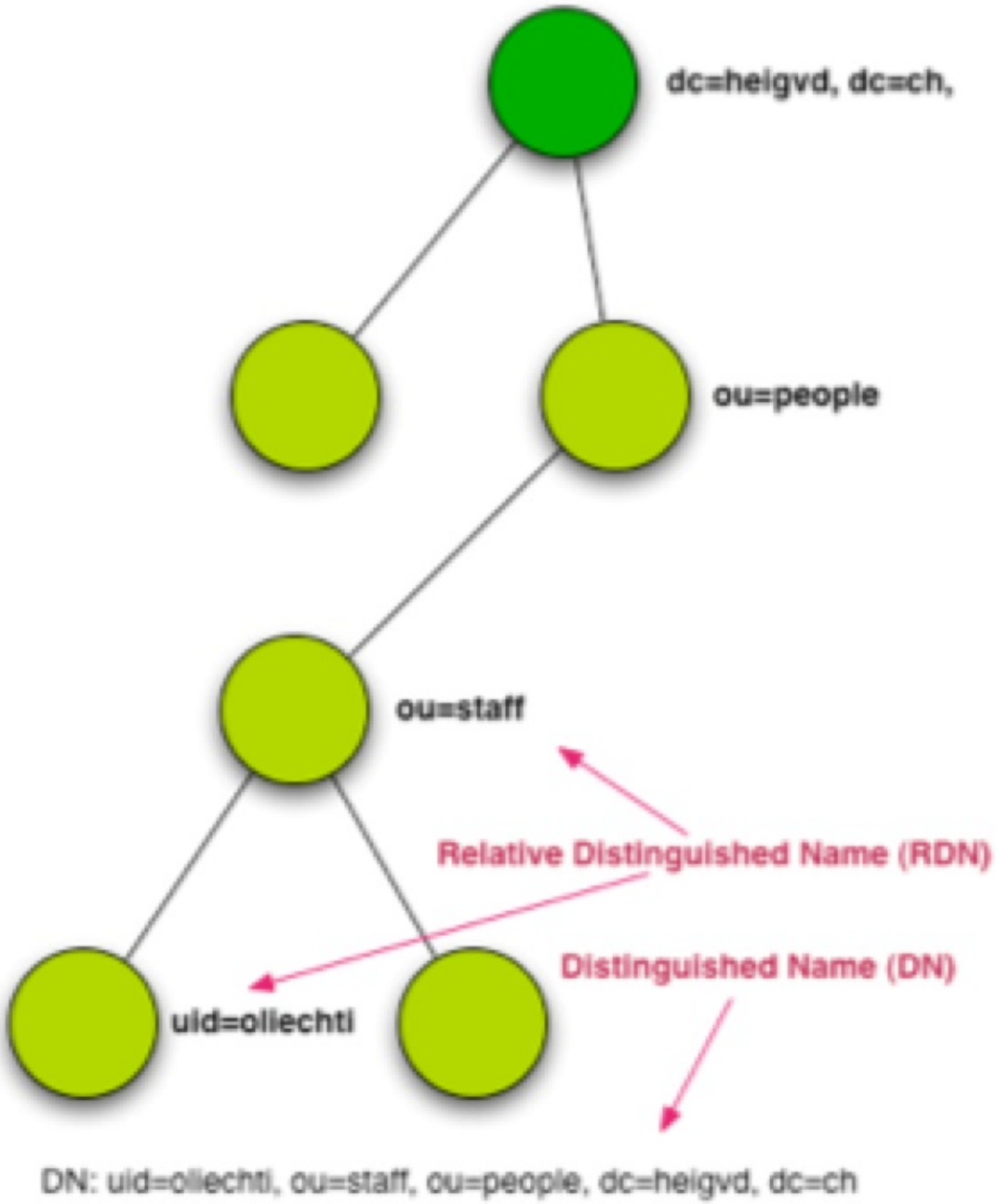
# Lecture

## The LDAP Data Model

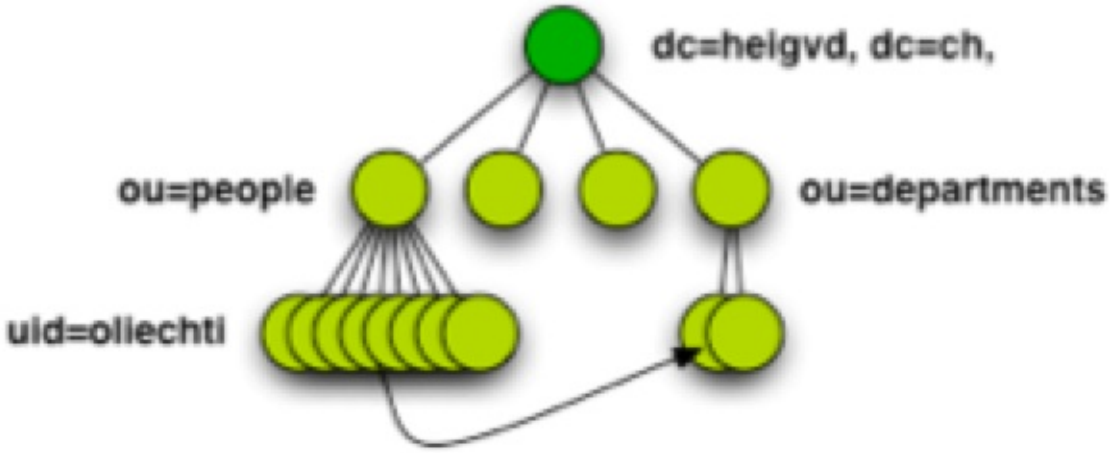
Unlike in a Relational Database Management System, information is not stored in a tabular format in a directory server. Rather, **it is stored in a hierarchical data structure called the Directory Information Tree (DIT)**.

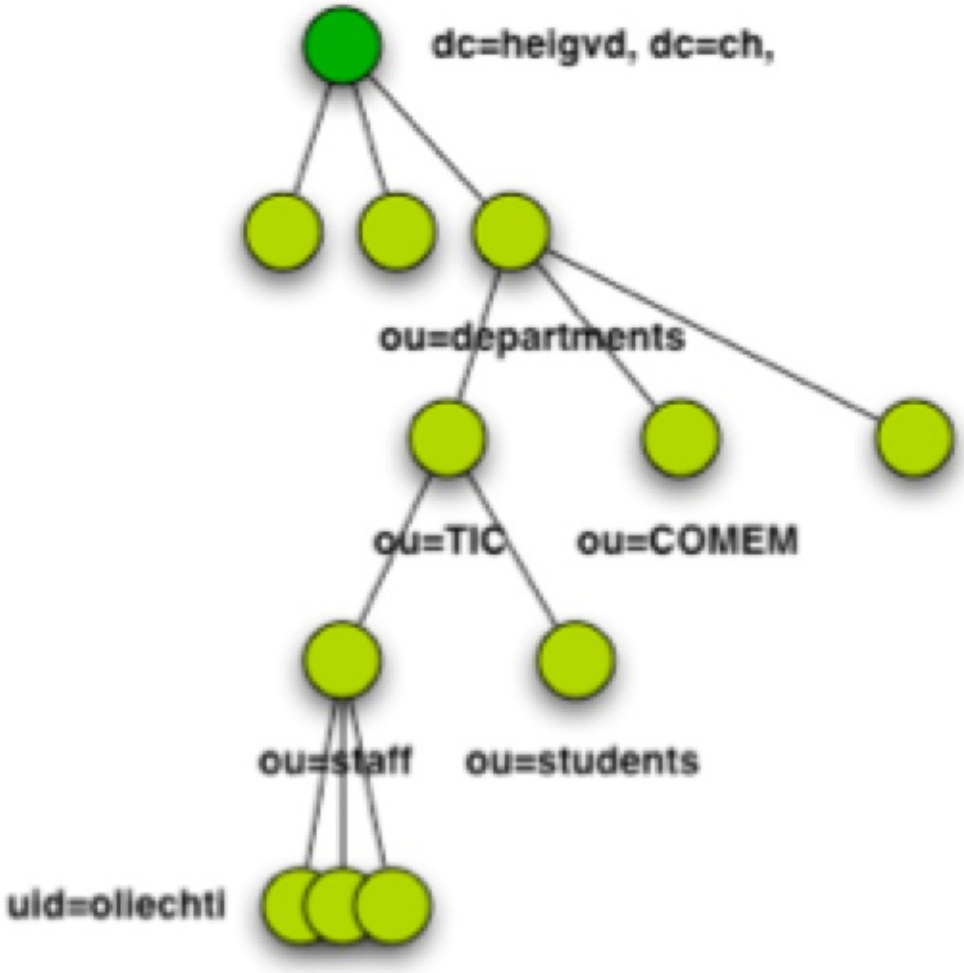



The **root** of the directory typically represents the organization (such as the HEIG-VD). Every node in the tree has a **type** (i.e. it represents a person, an organizational unit, a device, etc.) and a number of **attributes** (e.g. a name, an phone number, a department number). Every node is uniquely identified by its **Distinguished Name** (DN), which is a string value that also makes it possible to locate it in the tree (see below).




When you decide to use an LDAP server to manage information, **it is up to you to decide how you want to structure your tree**. In other words, it is up to you to decide how many levels you want to use, what kind of information you want to store in the different branches, etc. In the past, there was often a tendency to create deep hierarchies that were reproducing org charts. In these hierarchies, the nodes representing people were scattered under different nodes representing different departments (e.g. there was a *marketing* node, under which people working for the marketing department were placed, there was an *engineering* node, under which engineers were placed, etc). These structures have turned out to lack flexibility and to cause issues (for instance, where should a person be placed if working part-time for the marketing department and part-time for the engineering department?). For this reason, **it is recommended to create LDAP structures that are more flat and to capture relationships that exist between some nodes in relevant attributes**.




As we introduce LDAP concepts, we will provide references to the relevant RFCs. You should be aware tha**t there is not a single RFC that defines all aspects of the LDAP service. In the contrary, there are several distinct RFCs that cover respective topics**. As a starting point, you should have a look at RFC 4510, which provides the list of all LDAP-related specifications.

## RFC 4512

```
2.1.  The Directory Information Tree

   As noted above, the DIB is composed of a set of entries organized
   hierarchically in a tree structure known as the Directory Information
   Tree (DIT); specifically, a tree where vertices are the entries.

   The arcs between vertices define relations between entries.  If an
   arc exists from X to Y, then the entry at X is the immediate superior
   of Y, and Y is the immediate subordinate of X.  An entry's superiors
   are the entry's immediate superior and its superiors.  An entry's
   subordinates are all of its immediate subordinates and their
   subordinates.

   Similarly, the superior/subordinate relationship between object
   entries can be used to derive a relation between the objects they
   represent.  DIT structure rules can be used to govern relationships
   between objects.

   Note: An entry's immediate superior is also known as the entry's
         parent, and an entry's immediate subordinate is also known as
         the entry's child.  Entries that have the same parent are known
         as siblings.
```

```
2.2.  Structure of an Entry

   An entry consists of a set of attributes that hold information about
   the object that the entry represents.  Some attributes represent user
   information and are called user attributes.  Other attributes
   represent operational and/or administrative information and are
   called operational attributes.

   An attribute is an attribute description (a type and zero or more
   options) with one or more associated values.  An attribute is often
   referred to by its attribute description.  For example, the
   'givenName' attribute is the attribute that consists of the attribute
   description 'givenName' (the 'givenName' attribute type [RFC4519] and
   zero options) and one or more associated values.

   The attribute type governs whether the attribute can have multiple
   values, the syntax and matching rules used to construct and compare
   values of that attribute, and other functions.  Options indicate
   subtypes and other functions.

   Attribute values conform to the defined syntax of the attribute type.
```

```
2.3.1.  Relative Distinguished Names

   Each entry is named relative to its immediate superior.  This
   relative name, known as its Relative Distinguished Name (RDN)
   [X.501], is composed of an unordered set of one or more attribute
   value assertions (AVA) consisting of an attribute description with
   zero options and an attribute value.  These AVAs are chosen to match
   attribute values (each a distinguished value) of the entry.

   An entry's relative distinguished name must be unique among all
   immediate subordinates of the entry's immediate superior (i.e., all
   siblings).

2.3.2.  Distinguished Names

   An entry's fully qualified name, known as its Distinguished Name (DN)
   [X.501], is the concatenation of its RDN and its immediate superior's
   DN.  A Distinguished Name unambiguously refers to an entry in the
   tree.  The following are examples of string representations of DNs
   [RFC4514]:

      UID=nobody@example.com,DC=example,DC=com
      CN=John Smith,OU=Sales,O=ACME Limited,L=Moab,ST=Utah,C=US
```

Like in a Relational Database Management System, it is often desirable to **specify rules and constraints to enforce the consistency of the information stored in the directory**. In particular, it is desirable to specify what kind of entries can be created (i.e. what kind of object classes can be used), what kind of attributes can be added to these entries. These aspects are specified in what is called the **Directory Schema**.

**One thing to be aware of is that when you install an LDAP server, you most often do not start from scratch**. Indeed, most implementations provide you with a **standard schema**. In other words, they provide you with standard classes and attributes that you can use to create your entries. These standard classes and attributes have been specified in RFCs, such as the RFC 4519 or the RFC 2798.

This is a major difference with relational databases, where there is no notion of standardized *Employee* or *Organization* tables that would always have the same columns. Every DBA needs to define his own relational schema and specify the structure of all its tables. With LDAP, there are RFCs that specify standard LDAP classes and attributes. For this reason, whether you install Microsoft Active Directory or OpenDJ, you will be able to create entries that are instances of the `inetOrgPerson` or of the `organizationalUnit` classes. You will always have the same attributes available for these entries. **This is valuable not only because it allows you to get started quickly, but more importantly because it greatly**

**facilitates the exchange of information between directories**. Indeed, what is exported from the Microsoft Active Directory server deployed in a company A can be easily imported in an OpenDJ server deployed in a company B.

```
4.  Directory Schema

   As defined in [X.501]:

      The Directory Schema is a set of definitions and constraints
      concerning the structure of the DIT, the possible ways entries are
      named, the information that can be held in an entry, the
      attributes used to represent that information and their
      organization into hierarchies to facilitate search and retrieval
      of the information and the ways in which values of attributes may
      be matched in attribute value and matching rule assertions.

      NOTE 1 - The schema enables the Directory system to, for example:

      - prevent the creation of subordinate entries of the wrong
        object-class (e.g., a country as a subordinate of a person);

      - prevent the addition of attribute-types to an entry
        inappropriate to the object-class (e.g., a serial number to a
        person's entry);

      - prevent the addition of an attribute value of a syntax not
        matching that defined for the attribute-type (e.g., a printable
        string to a bit string).
```

```
2.4.  Object Classes

   An object class is "an identified family of objects (or conceivable
   objects) that share certain characteristics" [X.501].

   [...]

   Each object class identifies the set of attributes required to be
   present in entries belonging to the class and the set of attributes
   allowed to be present in entries belonging to the class.  As an entry
   of a class must meet the requirements of each class it belongs to, it
   can be said that an object class inherits the sets of allowed and
   required attributes from its superclasses.  A subclass can identify
   an attribute allowed by its superclass as being required.  If an
   attribute is a member of both sets, it is required to be present.

   [...]

   Each object class is identified by an object identifier (OID) and,
   optionally, one or more short names (descriptors).

   [...]

3.3.  The 'objectClass' attribute

   Each entry in the DIT has an 'objectClass' attribute.

      ( 2.5.4.0 NAME 'objectClass'
        EQUALITY objectIdentifierMatch
        SYNTAX 1.3.6.1.4.1.1466.115.121.1.38 )

   The 'objectIdentifierMatch' matching rule and the OBJECT IDENTIFIER
   (1.3.6.1.4.1.1466.115.121.1.38) syntax are defined in [RFC4517].

   The 'objectClass' attribute specifies the object classes of an entry,
   which (among other things) are used in conjunction with the
   controlling schema to determine the permitted attributes of an entry.
   Values of this attribute can be modified by clients, but the
   'objectClass' attribute cannot be removed.
```

# The LDAP Protocol

The LDAP protocol is used when a client wants to connect to a directory server and to **perform operations such as authenticating a user, searching for information or updating information**. A client can be a business application, which often uses the information managed in the directory to apply a security policy (e.g. it queries the directory to check if the current user as a given role). A client can also be an administration tool, which is used by an IT staff member to browse through the directory data.

RFC 4511

```
3.  Protocol Model

   The general model adopted by this protocol is one of clients
   performing protocol operations against servers.  In this model, a
   client transmits a protocol request describing the operation to be
   performed to a server.  The server is then responsible for performing
   the necessary operation(s) in the Directory.  Upon completion of an
   operation, the server typically returns a response containing
   appropriate data to the requesting client.

   Protocol operations are generally independent of one another.  Each
   operation is processed as an atomic action, leaving the directory in
   a consistent state.
```

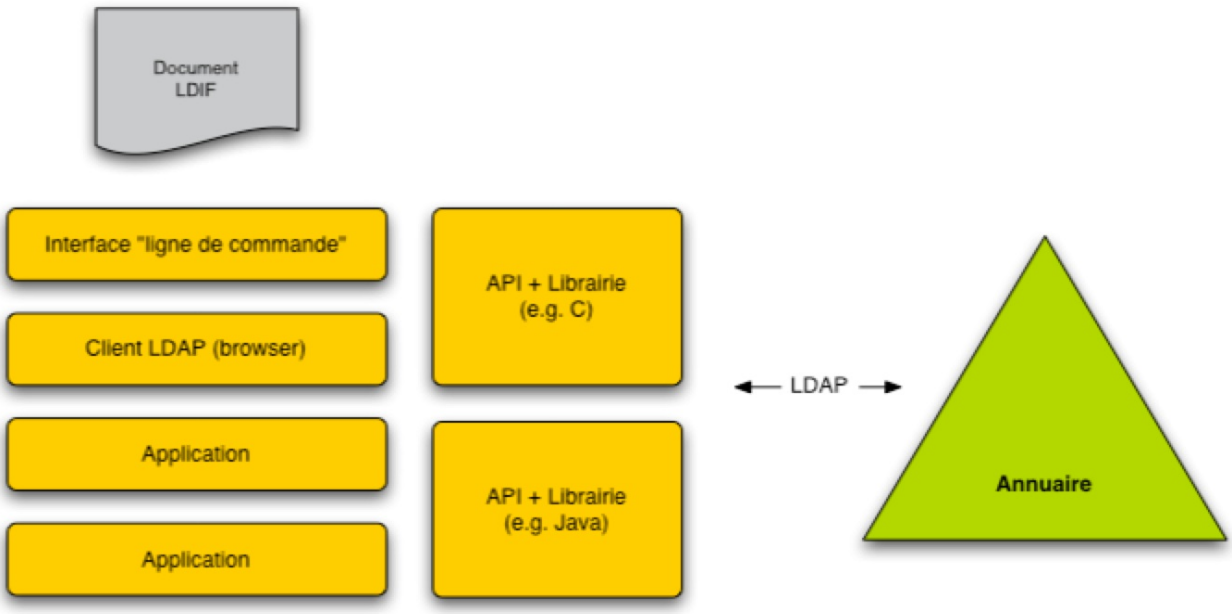
The LDAP protocol is implemented on top of **TCP**, with a standard port defined as **389**. Essentially, the LDAP server listens for connections on this port. Once the client has established a connection, it can send a series of operations. The operations that are understood by the LDAP server are defined in the RFC and are pretty straightforward. Again, they are used to i) authenticate users, ii) search for information and iii) modify information.

```
4.2.  Bind Operation

   The function of the Bind operation is to allow authentication
   information to be exchanged between the client and server.  The Bind
   operation should be thought of as the "authenticate" operation.

4.3.  Unbind Operation

   The function of the Unbind operation is to terminate an LDAP session.
   The Unbind operation is not the antithesis of the Bind operation as
   the name implies.  The naming of these operations are historical.
   The Unbind operation should be thought of as the "quit" operation.

4.5.  Search Operation

   The Search operation is used to request a server to return, subject
   to access controls and other restrictions, a set of entries matching
   a complex search criterion.  This can be used to read attributes from
   a single entry, from entries immediately subordinate to a particular
   entry, or from a whole subtree of entries.

4.6.  Modify Operation

   The Modify operation allows a client to request that a modification
   of an entry be performed on its behalf by a server.

4.7.  Add Operation

   The Add operation allows a client to request the addition of an entry
   into the Directory.

4.8.  Delete Operation

   The Delete operation allows a client to request the removal of an
   entry from the Directory.

4.9.  Modify DN Operation

   The Modify DN operation allows a client to change the Relative
   Distinguished Name (RDN) of an entry in the Directory and/or to move
   a subtree of entries to a new location in the Directory.

4.10.  Compare Operation

   The Compare operation allows a client to compare an assertion value
   with the values of a particular attribute in a particular entry in
   the Directory.
```

## Software Components

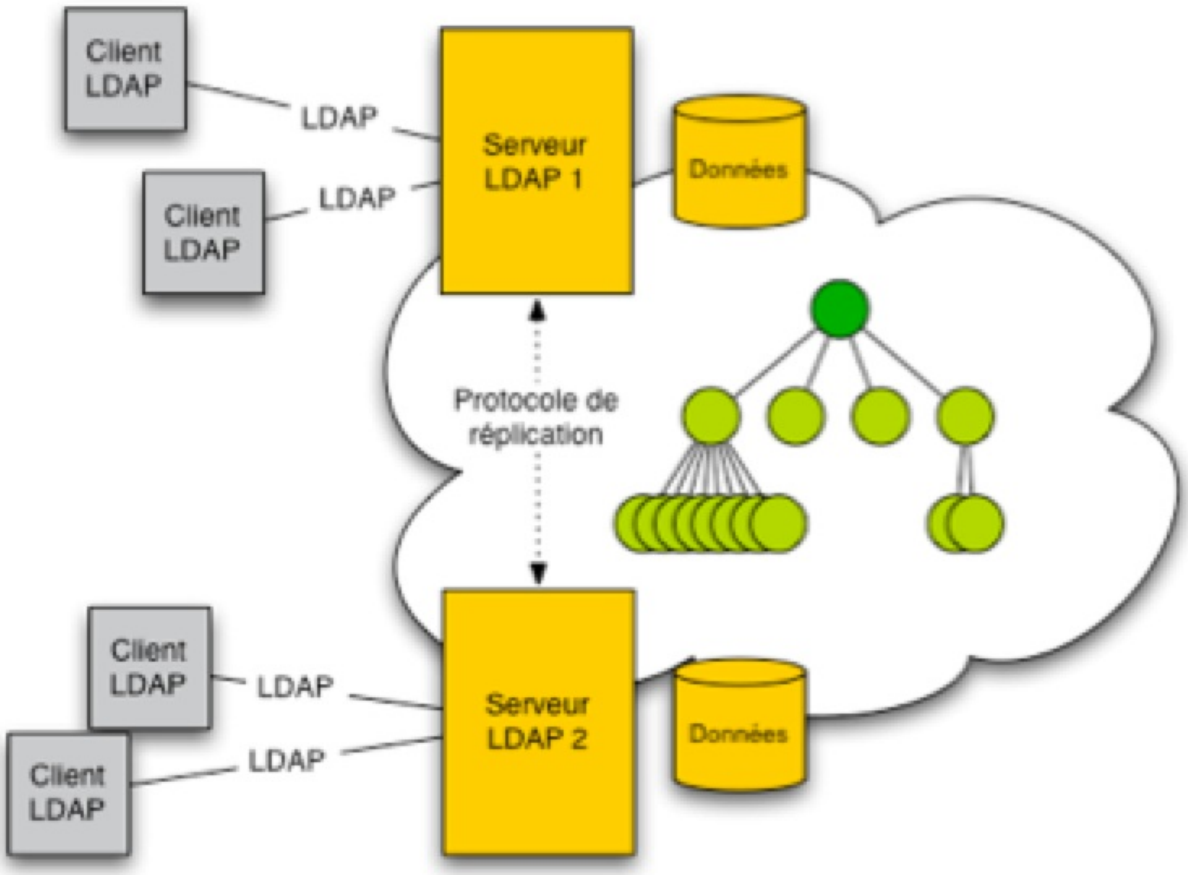
LDAP is a client-server protocol and as such, you can expect to interact with software components that play these respective roles. But what are LDAP clients exactly? We have already given some examples before. The first thing that you have to be aware of is that there are two main types of clients:

- Firstly, there are clients that an IT administrator is likely to use to browse and update information in the server. In this category, we include both tools with a graphical user interface (such as Apache Directory Studio) and command line tools (such as `ldapsearch` or `ldapmodify`).
- Secondly, there are clients that a *regular* user is likely to use, often without realizing that he is interacting with a LDAP server. In this case, the user is actually using some kind of business application (e.g. an ERP, a collaboration platform, a web service client, etc.). This business application embeds an LDAP module (a library) that connects to a server and issues LDAP commands. The typical example is a web application, where the user can enter credentials in a login form. When the submit button is pushed, the application issues a bind operation to validate the credentials.




Beyond browsers, libraries and servers, **there are other types of components** that implement the LDAP specification. All what we said about **systemic qualities** (performance, scalability, availability, etc.) in the context of the HTTP service also applies to the LDAP service. For this reason, there are also **LDAP proxies and gateways** that can be used to create sophisticated infrastructures to support the LDAP service (which is often critical: just think about what happens when nobody in the company is able to login!). If you want to learn more on that topic, have a look at this link.

As a matter of fact, the ability to create **distributed architectures for a directory service**, leveraging **replication** capabilities, was a key differentiator with Relational Database Management Systems in the past. Think about a **multinational company** where every employee has his credentials stored in a directory service. If this service was running on top of a centralized infrastructure, every single login would generate a request that traverses the Internet. **This would not be ideal from a response time and cost point of view**, especially in countries and locations with poor network connectivity. With LDAP servers, it is fairly easy to create an infrastructure where all write operations are made on a single server (called the **master**), but where read operations can be made on other servers (called the **slaves**) that can be *nearby*. The servers take care of the data replication, which is thus not a concern for application developers.




## Searching for Information in the Directory

When you use a Relational Database Management System, you use SQL to submit **queries** to the database server in order to **fetch records that match certain criteria**. With LDAP, you can do the same, but with a different syntax. The queries that you submit to an LDAP server are called **LDAP filters**.

The OpenDJ documentation provides many examples for filters. You should check this paragraph and test the filters and your local installation to become familiar with the syntax.

RFC 4515

```
1.  Introduction

   The Lightweight Directory Access Protocol (LDAP) [RFC4510] defines a
   network representation of a search filter transmitted to an LDAP
   server.  Some applications may find it useful to have a common way of
   representing these search filters in a human-readable form; LDAP URLs
   [RFC4516] are an example of one such application.  This document
   defines a human-readable string format for representing the full
   range of possible LDAP version 3 search filters, including extended
   match filters.
```

```
4. Examples

   This section gives a few examples of search filters written using
   this notation.

        (cn=Babs Jensen)
        (!(cn=Tim Howes))
        (&(objectClass=Person)(|(sn=Jensen)(cn=Babs J*)))
        (o=univ*of*mich*)
        (seeAlso=)
```

A feature that is related to searching is the ability to group entries in the directory. OpenDJ supports both **static groups** and **dynamics groups** (as explained here). The difference is that:

- with static groups, you have to explicitly specify who is a member of a given group. For instance, let us imagine that you have defined a **VIP** group. You would indicate that Bob, John and Barbara are members of the group by explicitly adding a value to the `member` attribute in the VIP group entry.
- with dynamic groups, you specify a condition that is evaluated against candidate entries, by means of an LDAP filter. For instance, you could define a **A-Level Students** dynamic group and specify that all students with a grade above a given threshold should belong to the group (assuming that grades would be stored in the LDAP directory). When requesting the list of members for the group, the server would run a query based on the condition and return the list of qualifying students. Again, make sure to read the OpenDJ documentation and to experiment with dynamic groups on your local installation.

## Importing and Exporting Directory Information

One task that is very common in real-word settings is the exchange of directory information with other systems. There are numerous examples for this, just think about the following scenarios:

- a legacy business application was using its own user repository (i.e. its user accounts were stored in its own database). At some point, you want to implement a single user repository across the information system and thus need to transfer application accounts toward the central directory server.
- Two companies have merged and you need to integrate their employees contact details in a single directory.
- Your HR application is able to generate an XML or a CSV file for new hires and you job is to create an LDAP entry for every new employee.
- You are asked to provide a contact list (with first name, last name, email and phone) for all employees in the directory.

When facing this type of requirements, you can think of two approaches:

- Firstly, you can use an LDAP library (possibly implementing a standard API such as the Java Naming and Directory, or JDNI, in Java) and make calls from an application that you write. In this case, you application becomes an LDAP client and issues requests across the network. For large batch processes, this is not an ideal solution (from a reliability, performance and efficiency point of view).
- The alternative is to use a data format to serialize the information that should go into or out of the directory. The Ldap Data Interchange Format (LDIF) was created exactly for that purpose. LDAP servers are able to import and export LDIF files. Hence, if you are able to produce an LDIF file, you are able to do an import procedure. Similarly, if you are able to parse an LDIF file, you are able to handle an export procedure.

You will see concretely what the second approach means in the lab. You will receive a CSV file containing user data and will have to convert it into a well-formed LDIF file. Once done, you will be able to import it in your local OpenDJ installation, before experimenting wit LDAP filters.

RFC 2849

```
Abstract

   This document describes a file format suitable for describing
   directory information or modifications made to directory information.
   The file format, known as LDIF, for LDAP Data Interchange Format, is
   typically used to import and export directory information between
   LDAP-based directory servers, or to describe a set of changes which
   are to be applied to a directory.

Background and Intended Usage

   There are a number of situations where a common interchange format is
   desirable.  For example, one might wish to export a copy of the
   contents of a directory server to a file, move that file to a
   different machine, and import the contents into a second directory
   server.

   Additionally, by using a well-defined interchange format, development
   of data import tools from legacy systems is facilitated.  A fairly
   simple set of tools written in awk or perl can, for example, convert
   a database of personnel information into an LDIF file. This file can
   then be imported into a directory server, regardless of the internal
   database representation the target directory server uses.

   The LDIF format was originally developed and used in the University
   of Michigan LDAP implementation.  The first use of LDIF was in
   describing directory entries.  Later, the format was expanded to
   allow representation of changes to directory entries.

   [...]

   Definition of the LDAP Data Interchange Format

   The LDIF format is used to convey directory information, or a
   description of a set of changes made to directory entries.  An LDIF
   file consists of a series of records separated by line separators.  A
   record consists of a sequence of lines describing a directory entry,
   or a sequence of lines describing a set of changes to a directory
   entry.  An LDIF file specifies a set of directory entries, or a set
   of changes to be applied to directory entries, but not both.

   There is a one-to-one correlation between LDAP operations that modify
   the directory (add, delete, modify, and modrdn), and the types of
   changerecords described below ("add", "delete", "modify", and
   "modrdn" or "moddn").  This correspondence is intentional, and
   permits a straightforward translation from LDIF changerecords to
   protocol operations.
```

```
Example 1: An simple LDAP file with two entries

version: 1
dn: cn=Barbara Jensen, ou=Product Development, dc=airius, dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
cn: Barbara Jensen
cn: Barbara J Jensen
cn: Babs Jensen
sn: Jensen
uid: bjensen
telephonenumber: +1 408 555 1212
description: A big sailing fan.

dn: cn=Bjorn Jensen, ou=Accounting, dc=airius, dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
cn: Bjorn Jensen
sn: Jensen
telephonenumber: +1 408 555 1212
```

```
Example 6: A file containing a series of change records and comments

version: 1
# Add a new entry
dn: cn=Fiona Jensen, ou=Marketing, dc=airius, dc=com
changetype: add
objectclass: top
objectclass: person
objectclass: organizationalPerson
cn: Fiona Jensen
sn: Jensen
uid: fiona
telephonenumber: +1 408 555 1212
jpegphoto:< file:///usr/local/directory/photos/fiona.jpg

# Delete an existing entry
dn: cn=Robert Jensen, ou=Marketing, dc=airius, dc=com
changetype: delete

# Modify an entry's relative distinguished name
dn: cn=Paul Jensen, ou=Product Development, dc=airius, dc=com
changetype: modrdn
newrdn: cn=Paula Jensen
deleteoldrdn: 1

# Rename an entry and move all of its children to a new location in
# the directory tree (only implemented by LDAPv3 servers).
dn: ou=PD Accountants, ou=Product Development, dc=airius, dc=com
```

```
changetype: modrdn
newrdn: ou=Product Development Accountants
deleteoldrdn: 0
newsuperior: ou=Accounting, dc=airius, dc=com
# Modify an entry: add an additional value to the postaladdress
# attribute, completely delete the description attribute, replace
# the telephonenumber attribute with two values, and delete a specific
# value from the facsimiletelephonenumber attribute
dn: cn=Paula Jensen, ou=Product Development, dc=airius, dc=com
changetype: modify
add: postaladdress
postaladdress: 123 Anystreet $ Sunnyvale, CA $ 94086
-

delete: description
-
replace: telephonenumber
telephonenumber: +1 408 555 1234
telephonenumber: +1 408 555 5678
-
delete: facsimiletelephonenumber
facsimiletelephonenumber: +1 408 555 9876
-

# Modify an entry: replace the postaladdress attribute with an empty
# set of values (which will cause the attribute to be removed), and
# delete the entire description attribute. Note that the first will
# always succeed, while the second will only succeed if at least
# one value for the description attribute is present.
dn: cn=Ingrid Jensen, ou=Product Support, dc=airius, dc=com
changetype: modify
replace: postaladdress
-
delete: description
-
```