DEPARTAMENT OF AUTOMATION AND COMPUTER SCIENCE
TECHNICAL UNIVERSITY OF CLUJ-NAPOCA

Data Transmission
Laboratory - volume I

Dipl. Eng. Camelia Claudia AVRAM, PhD
Dipl. Eng. Dan RADU, PhD

Contents

LIST OF TABLES

LIST OF FIGURES

1 DESCRIPTION OF THE WORKING ENVIRONMENT

1.1 OBJECTIVES

- Introduction in HTML, CSS and JavaScript technologies
- Design and implementation of graphic interfaces
- Event management in software applications

1.2 INTRODUCTION

1.2.1 REASONING

Currently, the technology package ("technology stack") consisting of **HTML, CSS** and **JavaScript** is extremely popular in the IT industry, because it supports a wide range of software applications, among which the most notable are web applications. Lately, however, due to the apparition of various platforms such as Node.js [2] and Electron [3]), there is growing support for backend, desktop and mobile applications.

A famous quote from 2007 states that *"every application that can be written in JavaScript will eventually be written in JavaScript"* - Jeff Atwood, Co-founder of StackOverflow. This assertion was confirmed by the most recent surveys regarding the use of various programming languages. According to 2018 statistics, as shown in figures 1.1 and 1.2, JavaScript appears to be the most popular programming language, both on Github and on StackOverflow.

Also, many important players in the software industry (Facebook/React, Google/Angular, Microsoft, Mozilla, Yahoo, Alibaba) support and contribute to the development of tools and libraries required by development of applications using these three languages.

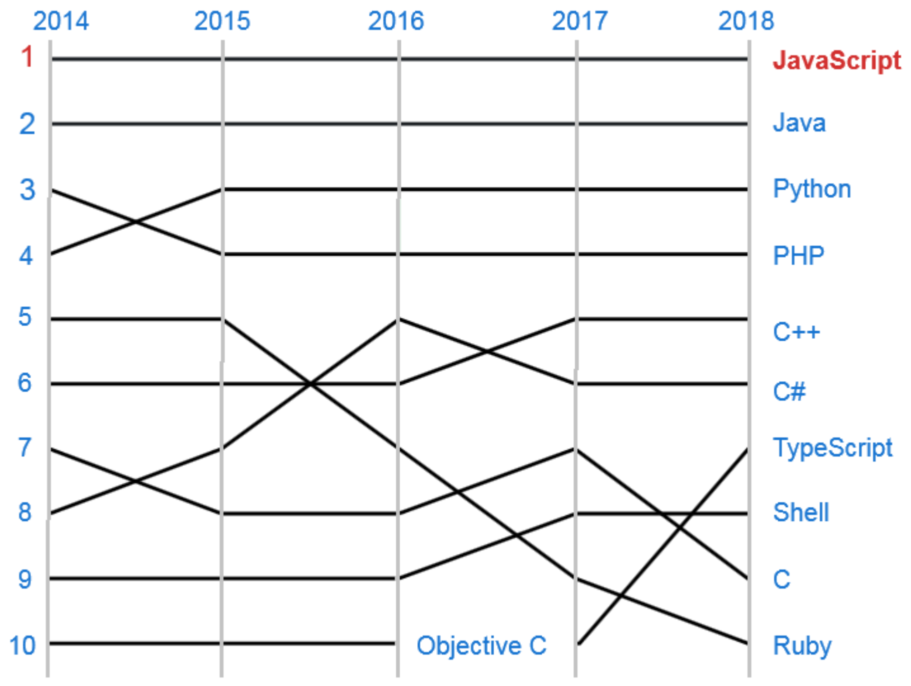


Figure 1.1: Top programming languages on Github in 2018 (Source: [4])

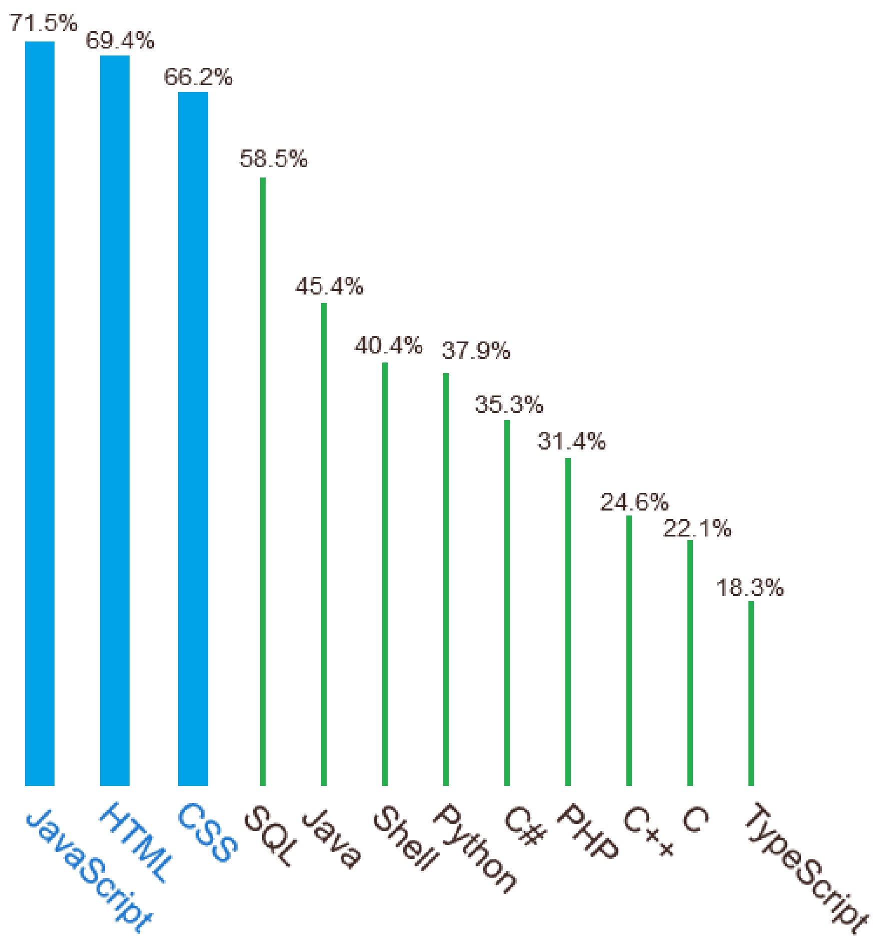


Figure 1.2: Top programming languages on StackOverflow in 2018 (Source: [5])

1.2.2 HTML

HTML, an acronym standing for "Hyper Text Markup Language", is a language that allows us to define graphic interfaces for browser-based software applications.

An HTML page has a tree-like structure of graphic elements, having as root node the <html> element, with two children, <head> and <body>. <head> contains the description of the web page (metadata, title, fonts used) and also possible external files such as those containing CSS styles and JavaScript scripts. <body> contains the description of the graphic interface, which in case of a web application includes: header, menu and contents.

Some of the most important HTML elements are: <div>, <input> and <button>.

1.2.3 CSS

CSS (Cascading Style Sheets) is the language that allows description of the graphical representation of HTML elements. The display of graphic elements observes certain styles, specific to each browser. However, this style may be overwritten and customized using the CSS syntax.

A few of the most frequently used CSS properties are: font size, border, background color, color, width and height.

CSS-defined styles allow us to control how applications will look like on various media (desktop screens, laptops, smartphones, or paper).

1.2.4 JavaScript

JavaScript is a programming language created by Brendan Eich in 1995, which was used initially with the Netscape browser.

JavaScript is the language that controls the behaviour of web graphic interfaces and more. Even if JavaScript was used mainly for web applications, over the last years it has become an increasingly popular choice for desktop and mobile applications.

There are several libraries that allow creation of complex client applications: jQuery, React, VueJS, Angular. They expand the capabilities of JavaScript by adding various usual functionalities such as: simplified syntax for referencing HTML elements, a bidirectional link between model (variable) and the graphic interface, component-based architecture.

All types of applications (frontend, backend, mobile) can make use of the NodeJS platform, which allows structuring and automation of many of the steps required in software development (library import, build of resource-serving web servers, configuration and execution of client build).

1.2.5 Development tools

Although a browser and Notepad are enough to write and test simple applications, once the programs become more complex, the use of specific code editors such as Visual Studio Code, Sublime sau Atom is recommended.

1.3 Lab work

1.3.1 First JavaScript application

The first application includes 3 files: **index.html**, **style.css** and **app.js**. The **index.html** file defines the structure of the graphic interface and will initially display a simple text field. Section <head> includes a CSS file and, in the end of the <body> section, the JavaScript file.

```html
<!DOCTYPE html>
<html>
    <head>
    </head>
    <body>
        <div id="message"></div>

        <script src="app.js"></script>
    </body>
</html>
```

It is important that the JavaScript instructions are executed only after all HTML graphic elements from <body> have been already uploaded into the browser. This can be done by placing the script at the end of the <body> element, as the resources (JS, CSS, HTML) are being uploaded sequentially, in the same order as they appear in the HTML file. Another way of doing this is to execute the code only when the event generated by the page loading occurs (ex: <body onload="run();">).

In the JavaScript file **app.js**, the browser console displays the following message:

```javascript
console.log("Welcome to data transmission");
```

☞ To execute the application and to read the JavaScript message, open **index.html** in any browser, press the F12 key and select the **Console** tab.

After that, the inner HTML string value will be modified in JavaScript, as follows:

```
document.getElementById("message").innerHTML = "Message
   from JavaScript";
```

The JavaScript native object named **document** [6] allows referencing of any HTML element, based on a selector that may be the element's type, ID or class. Once we have this reference, we can control the characteristics of the HTML element from JavaScript. We can also manage the events generated when a user interacts with the graphic interface.

☞ The identifier used in JavaScript as argument of the getElementById method ("**message**") is the same with the unique identifier specified in the HTML element "<div id="**messsage**"></div>".

1.3.1.1 Usual instructions in JavaScript

In JavaScript, variables are declared by means of the key word **var** placed in front of the variable's name and can assume various values. In the following script, four types of variables (Number, String, Boolean and Object) will be declared.

```
var sum = 10;
var name = "Alexandru";
var isActive = true;
var user = {id: 1, name:"Andrei", age: 21, };
```

The user object is given in JSON (JavaScript Object Notation) format. In JSON syntax, objects start with { and end with }. Inside the curly brackets, **key**: **value** pairs are specified, defining the object's charecteristics. JSON objects can also include element lists, which are declared within square brackets. The following script gives an example of a complex JSON object:

```
 var user = {
    "id": 1,
    "name": "Alexandru Cristea",
    "username": "acristea",
    "email": "acristea@test.com",
    "address": {
      "street": "Padin",
      "number": "Ap. 10",
      "city": "Cluj-Napoca",
      "zipcode": "123456",
      "geo": {
        "lat": "46.783364",
        "lng": "23.546472"
      }
    },
    "phone": "004-07xx-123456",
    "company": {
      "name": "XYZ",
      "domain": "Air Traffic Management",
      "cities": ["Cluj-Napoca", "Vienna", "Paris"]
    }
  }
```

The following script shows how various characteristics of the user object are displayed:

```

console.log(user.name);
console.log(user.address.geo.lat);
console.log(user.company.name);
console.dir(user.company.cities);
console.log(user.company.cities[0]);
...
```

This is how a function is defined and called:

```
function print(message){
    console.log(message);
}
print("hello");
```

The **ternar operator** is a simplification of the **if** instruction. Below is an example:

```
var password="123456";
console.log(password=="123456"?"ALLOW":"DENY");
```

The equivalent logic described by the **if** instruction is as follows:

```
var password="123456";
if(password == "123456"){
    console.log("permission accepted");
} else {
    console.log("permission accepted");
}
```

1.3.2 Implementing and testing a JavaScript function

The application consists of 3 files: **index.html**, **app.js** and **appTest.js**. The graphic interface allows us to put a value into an input field:

```html
<!DOCTYPE html>
<html>
    <head>
    </head>
    <body>
        <input type="text" id="n"  value="5"/>

        <script src="app.js"></script>
        <script src="appTest.js"></script>
    </body>
</html>
```

The **app.js** file is as follows:

```javascript
 1 document.getElementById("n").addEventListener('input',
      inputSum);
 2
 3 function inputSum(){
 4     var inputNumber = parseInt(document.getElementById("n")
          .value);
 5     sum(inputNumber);
 6 }
 7
 8 function sum(n){
 9     if (typeof n === 'undefined') return "n is undefined";
10     var sum = 0;
11     for(var i=1;i<=n;i++){
12         sum+=i;
13     }
14     return sum;
15 }
```

- Line 1: A handler event is specified, that will be called when the user changes the input value.
- Line 4: Variables in JavaScript are defined by means of the **var** keyword. The value in the input field is read and converted from String to integer.
- Line 5: The function is called that returns the sum of the elements from 1 to n.

The **appTest.js** file contains a set of 4 tests that validate the desired behaviour of the **sum** function:

```
function test(){
    console.log(sum(0)==0?"Passed":"Failed");
    console.log(sum(2)==3?"Passed":"Failed");
    console.log(sum(4)==10?"Passed":"Failed");
    console.log(sum()=="n is undefined"?"Passed":"Failed");
}
test();
```

JavaScript syntax allows us to write the code above also in form of an anonymous function executed immediately, as shown:

```
(function(){
    console.log(sum(0)==0?"Passed":"Failed");
    console.log(sum(2)==3?"Passed":"Failed");
    console.log(sum(4)==10?"Passed":"Failed");
    console.log(sum()=="invalid argument"?"Passed":"Failed"
       );
})();
```

1.3.3 Designing of a HTML graphic interface

The **index.html** file defines the structure of a graphic interface that contains a text label and a button.

```
<!DOCTYPE html>
<html>
    <head>
        <link rel="stylesheet" href="style.css">
    </head>
    <body>
        <div id="counter"></div>
        <button id="inc">+</button>

        <script src="app.js"></script>
    </body>
</html>
```

The CSS style will be modified next, in the **style.css** file:

```
body {
  padding: 20px;
  text-align: center;
  background-color:#f5f5f5;
}

#widget {
  width: 60px;
  background-color: #555;
  margin: 0 auto;
  padding: 10px;
}

#counter {
  height: 50px;
  width: 50px;
  margin: 0 auto;
  background-color: #fff;
  color: #555;
  text-align: center;
  line-height: 50px;
  font-size: 30px;
}

#inc {
  margin-top: 10px;
  height: 50px;
  width: 50px;
  background-color: #5b9aff;
  color: #fff;
  text-align: center;
  line-height: 50px;
  font-size: 20px;
  border: 0px;
}

#inc:hover {
  cursor: pointer;
  background-color: #8cb8ff;
  color:#fff;
}
```

The JavaScript **app.js** file describes the behaviour of the graphic interface, specifically the logic required for the value of the counter to increase every time a user presses the button.

```
1  var counter = 0;
2
3  function printValue(divId, value){
4    document.getElementById(divId).innerHTML = value;
5  }
6  printValue("counter", 0);
7
8  document.getElementById("inc").addEventListener("click",
      increment);
9
10 function increment(){
11   counter++;
12   printValue("counter", counter);
13 }
```

- Line 1: Application status.
- Line 3: The function that allows us to display a text value in a div with specified identifier.
- Line 6: Initial display of the counter on the graphic interface.
- Line 8: Specification of a handler event that will be called when the user presses the increment button. This handler event has 2 parameters: event type and the function that will be called when the specified event occurs.

1.3.4 jQuery

jQuery is a library written in JavaScript that simplifies and offers a more expressive programming syntax, at the same time expanding the functionalities of the JavaScript language. jQuery can be included in the head section of the HTML page, as follows:

```
<head>
    <script src="https://ajax.googleapis.com/ajax/libs/
       jquery/3.3.1/jquery.min.js"></script>
</head>
```

To reference a HTML element, the simplified syntax **$("#id")** is used. The "$" symbol specifies the beginning of a jQuery object/function. The JavaScript code in section 1.3.3 can be rewritten using jQuery as follows:

```
 1 var counter = 0;
 2
 3 function printValue(divId, value){
 4     $("#"+divId).html(value)
 5 }
 6 printValue("counter", 0);
 7
 8 $("#inc").on('click', increment);
 9
10 function increment(){
11   counter++;
12   printValue("counter", counter);
13 }
```

Line 4: a **div** element can be written using the function **html()** from the **jQuery** library.

If a text **input** field is desired, the reading (line 1) and setting (line 2) of the **input** element's value with jQuery is done as follows:

```
1 var textValue = $('#inputTextId').val();
2 $('#inputTextId').val('123');
```

1.4 Applications and problems

1.4.1 Problem 1 *(3 points)*

Implement/test all applications described in Section 1.3:

- Modify the value of a text field in JavaScript using the **getElementById** function, available in the interface of the **document** object *(0.25 p)*.
- Display console message in JavaScript *(0.25 p)*.
- Access a value from a JSON object *(0.25 p)*.
- Define and call a JavaScript function *(0.25 p)*.
- Use the **if** instruction as a ternar operator *(0.25 p)*.
- Design a HTML graphic interface *(0.5 p)*.
- Use the jQuery library in the JavaScript application *(0.25 p)*.

1.4.2 Problem 2 *(1.5 points)*

Expand the application in Section 1.3.3 by adding a decrement button and restrict the counter value to the [0-10] interval.

1.4.3 Problem 3 *(1.5 points)*

Modify the logic of the **sum** function in Section 1.3.2 so as to only accept numbers as input argument. For instance, if the input is a string, the return value "not a number" is expected.

Write two tests to verify the input of a string and a Boolean variable as argument for the sum function.

1.4.4 Problem 4 *(2 points)*

Write a JavaScript function with the header **getFibonacci(n)** that generates elements of the Fibonacci series up to a imposed limit **n** and write tests to validate the following behaviour:

- The function returns the sequence [1, 1] if the input argument is 2.
- The function returns the sequence [1, 1, 2, 3, 5] if the input argument is 5.

- The function called without argument or with any other data type other than a number returns "not allowed".
- The function called with n<1 or n>10 returns "not allowed".

1.4.5 PROBLEM 5 *(2 points)*

Based on the model in Section 1.3.2 and using **jQuery**, implement the graphic interface for a simple calculator that allows addition, subtraction, multiplication, division, and remainder [+, -, *, /].

Required fields:

- Two **input** text fields (having the ids **firstNumber** and **secondNumber**) for receiving the data put in by the user. jQuery uses the **val** function of the **input** HTML element (ex: var firstNumberText = $('#firstNumber').val();) to read the value of the text field. The function **parseInt** (ex: var firstNumber = parseInt(firstNumberText) can be used for the string-integer conversion.
- A **button** element for 'equals' (with event handler at the click).
- Result displayed in a **div**.

1.5 WHAT WILL THE NEXT LAB BE ABOUT?

- Wired transmission media
- Data transfer between software components distributed by means of JavaScript, both frontend (UI) and backend (API).
- Using of **http** and **Websocket** protocols for development of client-server applications.

We will also use **NodeJS**, **VueJS** and **Express.js** libraries and technologies.

Bibliography

[1] Adina Aştilean. *Data transmission, course notes.*

[2] `https://nodejs.org/en/`

[3] `https://electronjs.org`

[4] `https://octoverse.github.com/projects#languages`

[5] `https://insights.stackoverflow.com/survey/2018/#technology`

[6] `https://www.w3schools.com/js/js_htmldom.asp`

2 Wired communication systems

2.1 Objectives

- Becoming familiar with wired transmission media;
- Developing a client / server application using the http protocol;
- Developing a client / server application using the Websocket protocol.

2.2 Introduction

2.2.1 Reasoning

2.2.2 Wired transmission media

Guided media Information transmission media are being categorized according to the type of signal they carry. Advantages of wired transmission are:

- speed of transmission - the rapport between the amount of transmitted information and the time required to carry out the transmission;
- the band width - the amount of information that can be transmitted;
- the distance over which data can be transmitted - the maximal distance between emitter and receptor for which the transmission can be carried out without attenuation or error.

2.2.2.1 The coaxial cable

Signal band: 0 - 600 MHz

The coaxial cable (or coax) consists of a copper wire (core) through which the transmission takes place, surrounded by a flexible woven metallic tube, shielding against interferences. The two are separated by a insulating plastic sheath. The transmitted signal may be analogic or digital, fig. 2.1

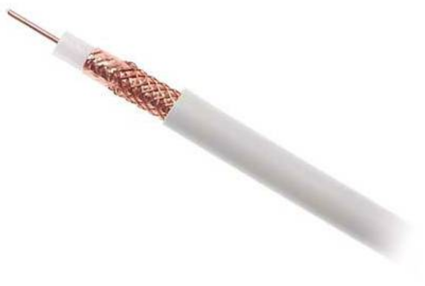


Figure 2.1: Coaxial cable

The most widely used coaxial cables, figure 2.2, figure [3] (RG - radio guide; /U - general use generală), are:

- RG-58 /U - used for low power signals and radio frequency (RF) connections; resistance 50 Ohm, impedance 25 pF/ft (82 pF/m);
- RG-59 /U - used for low power video signals and RF connections; resistance 75 Ohm, impedance 20pF/ft (60pF/m); most frequently used for CCTV (closed-circuit television) distribution.
- RG-62 /U - for industrial indoor use, for transmission of high frequency and power signals; resistance 93 Ohm;
- RG-6/U - for residential use; resistance 75 Ohm; most frequently used for TV distribution;
- RG-11/U - for industrial indoor use, for transmission of high frequency and power signals; resistance 75 Ohm; most frequently used for HDTV (High Definition TV).

Connectors, figure 2.3, figure 2.4.
Advantages and disadvantages:

+ cheap and durable;
+ easy to use;

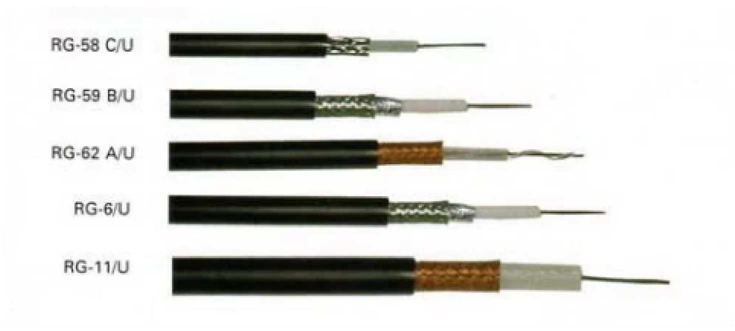


Figure 2.2: Coaxial cable

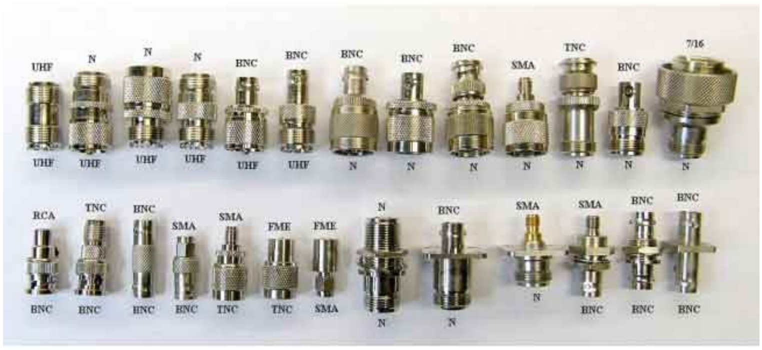

UHF N N N BNC BNC BNC BNC BNC SMA TNC BNC 7/16
UHF UHF UHF UHF UHF UHF N N N N N N N
RCA TNC BNC SMA SMA FME FME N BNC SMA SMA BNC BNC
BNC BNC BNC BNC TNC TNC SMA N N N BNC BNC BNC

Figure 2.3: Connectors (Source: [5], [6])

+ good EMI resistance;
+ transfer speed up to 10 Mbps;
- maintenance - a faulty cable renders the whole network inactive.

Figure 2.4: Connectors (Source: [5], [6])

2.2.2.2 TWISTED PAIR CABLES

A twisted pair cable consists of several insulated copper wire pairs and is used in communications. The twisting ensures electromagnetic protection, reducing or cancelling the parasitic signals (noise) from other neighbouring electric circuits and is accomplished by twisting together two cables, as in figure 2.5.

An electric current of variable intensity flowing through a conductor generates a variable magnetic field. Conversely, a variable magnetic field induces a variable electric tension (voltage) in a conductor. [7]

If a variable electric signal passes through two electric conductors that close a circuit, then the currents flowing through them will have opposite directions. The electromagnetic field that each of them generates will be in antiphase, thus cancelling each other out.

Among the most usual twisted pair types are:

- FTP - Foiled Twisted Pair;
- UTP - Unshielded Twisted Pair, figure 2.6;
- STP - Shielded Twisted Pair;
- S/UTP - Screened Unshielded Twisted Pair;
- S/FTP - Screened Foiled Twisted Pair;

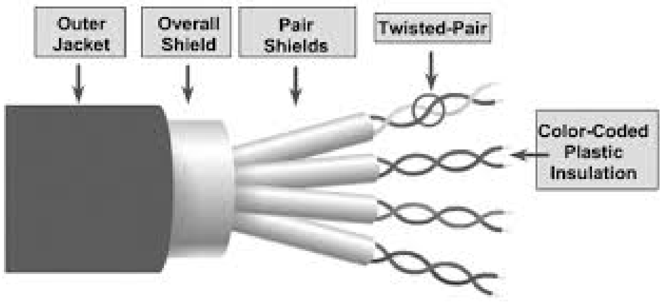


Figure 2.5: Twisted Pair, 4 pairs

- S/STP - Screened Shielded Twisted Pair, figure 2.7.

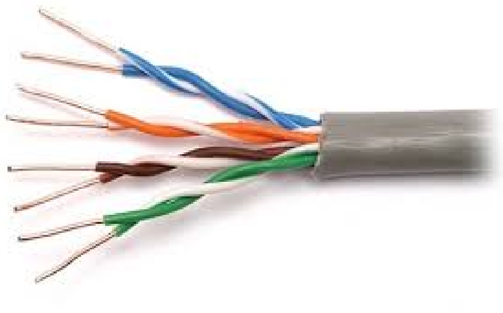


Figure 2.6: UTP cable, 4 pairs

Categories of twisted pairs
Electronic Industries Association (EIA) established UTP standards in 1995, [8]:

- CAT 1 - Category 1, for use in telephony networks. Only analog signal can be transmitted, with frequencies up to 0.4 MHz and transmission speed lower than 100 Kbps; has not been designed for data transmission.
- CAT 2 - Category 2, 4 twisted pair cable. Used for analog and digital signals with a transmission speed of up to 4 Mbps; used in old local area networks (LANs) (Token-passing ring LANs, IEEE 802.5). Bandwidth 1 MHz.
- CAT 3 - Category 3, 4 twisted pair cable, 7-8 twistings per metre. Used for Ethernet transmission speeds of up to 10 Mbps and for frequency signals up to 16 MHz. Used in old Ethernet networks 10base-T LANs (IEEE 802.3).

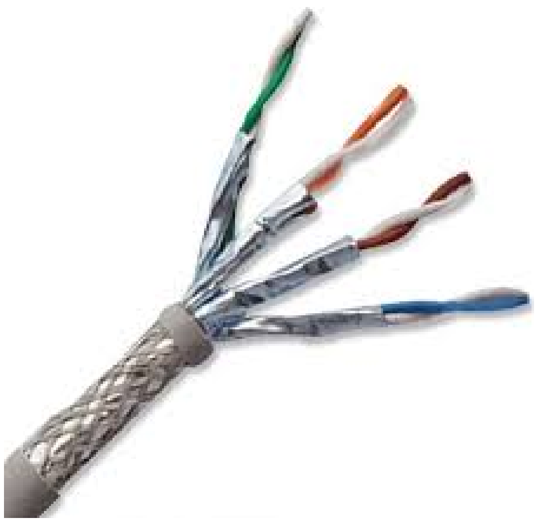


Figure 2.7: S/STP cable, 4 pairs

- CAT 4 - Category 4, 4 twisted pair cable. Used in networks with a transmission speed of up to 16 Mbps and frequencies up to 20 Mhz; used in token-based or 10Base-T networks.
- CAT 5 - Category 5, the cable type most frequently used in Ethernet networks, having 4 twisted pairs with 3-4 twistings per inch. Can be used for transmission speeds of up to 155 Mbps or frequencies up to 100 MHz over a distance up to 100 m. Ensures data transport with a speed up to 100 Mbps (Fast Ethernet, IEEE 802.3u) and is used for 100base-T and 10base-T networks.
- Category 5e – 4 twisted pair cable. May be used for a transmission speed of up to 1 Gbps or for frequencies up to 100 MHz, over a distance up to 100 m. For 100BASE-TX and 1000BASE-T Ethernet.
- CAT 6 - Category 6, 4 twisted pair cable. Can be used for transmission speeds of up to 10 Gbps or for frequencies up to 250 MHz, over distancies up to 100 m. For 10GBASE-T Ethernet.
- CAT 6a - Category 6a, 4 twisted pair cable. Can be used for a transmission speed up to 10 Gbps or for frequencies up to 500 MHz, over a distance up to 100 m. For 10GBASE-T Ethernet.
- CAT 7 - Category 7, supports up to 10 Gbps and frequencies up to 600 MHz. For telephony, CCTV, 1000BASE-TX on the same cable, 10GBASE-T Ethernet. Employs F/FTP and S/FTP.

- CAT 7a - Category 7a, supports up to 10 Gbps and frequencies up to 1000 MHz. For telephony, CATV (community antenna television), 1000BASE-TX on the same cable, 10GBASE-T Ethernet. Employs F/UTP.
- CAT 8.1 - Category 8.1, supports up to 40 Gbps and frequencies of up to 1600 - 2000 MHz. For telephony, CATV, 1000BASE-TX on the same cable, 40GBASE-T Ethernet. Employs F/UTP.
- CAT 8.2 - Category 8.2, supports up to 40 Gbps and frequencies of up to 1600 - 2000 MHz. For telephony, CATV, 1000BASE-TX on the same cable, 40GBASE-T Ethernet. Employs F/FTP and S/FTP.

UTP and FTP connectors (color codes for the A and B wiring standards), figure 2.8

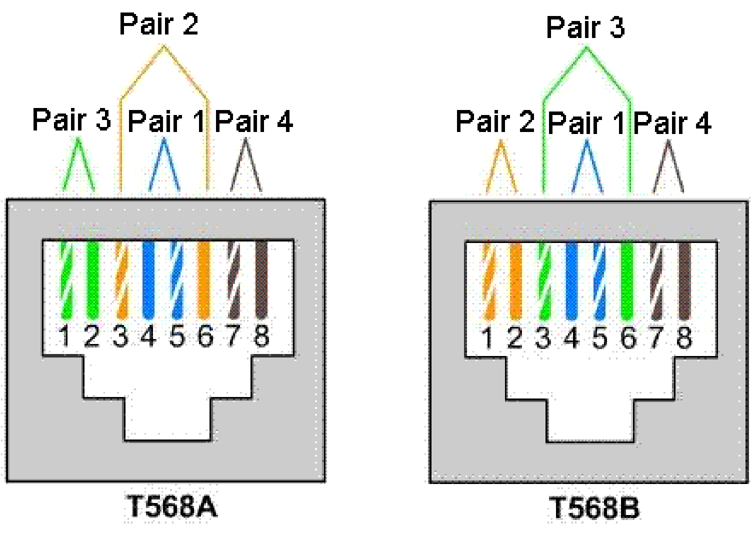


Figure 2.8: RJ-45 connectors, 568A and 568B wiring standards (Source: [9], [10])

Advantages and disadvantages:

+ cheap, easy to install;
+ less sensitive to interferences from nearby equipment; in turn, they do not cause interferences themselves;
+ STP support higher speed data transmission;
- STP is bulkier and more expensive than UTP;
- STP is harder to wire.

2.2.2.3 The optical fibre

An optical fibre cable consists of many glass or plastic threads (the optical fibre). A single cable can be made of several hundreds of fibres, figure 2.9. The core of the optical fibre is extremely thin, 2 - 125 µm. A single thread is able to transmit several thousand phone calls. The transmission capacity is in the gigabyte range, with very little attenuation (0.2 to la 1 dB/km) over several kilometres, as well as a very high immunity to noise.

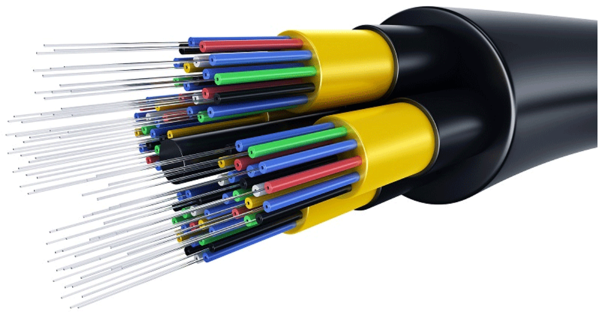


Figure 2.9: Optical fibre (Source: [11])

Types of optical fibre:

- Multimode fibre (LED): has a 62.5µm or 50pm core, allowing multiple light modes to be propagated.

  Multimode cables can carry information over relatively short distances only and are used for interconnecting two networks.

- Single mode fibre (laser): with a core of about 9 µm, it allows transmission of a single ray of light of specific wavelength.

  Cable television, Internet and telephone signals are usually based on single-mode fibres. The information can be carried over distances of over 100 km (60 miles).

Some premium optical fibres show a much lower degradation of the signal - less than 10%/km at 1.550 nm.

Digital signals are codified into analogic light impulses, NRZ – "Non-Return to Zero". Examples of optical fibre connectors are shown in figures 2.11 and 2.10.

Most fibres function as duplexes:

- one for transmission, and
- one for reception.

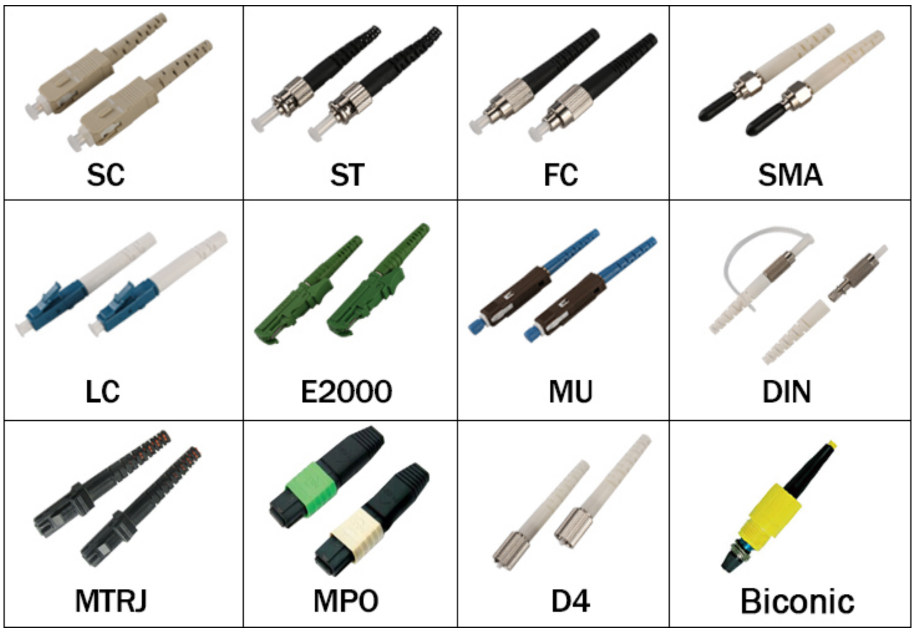


Figure 2.10: Optical fibre connectors (Source: [13])

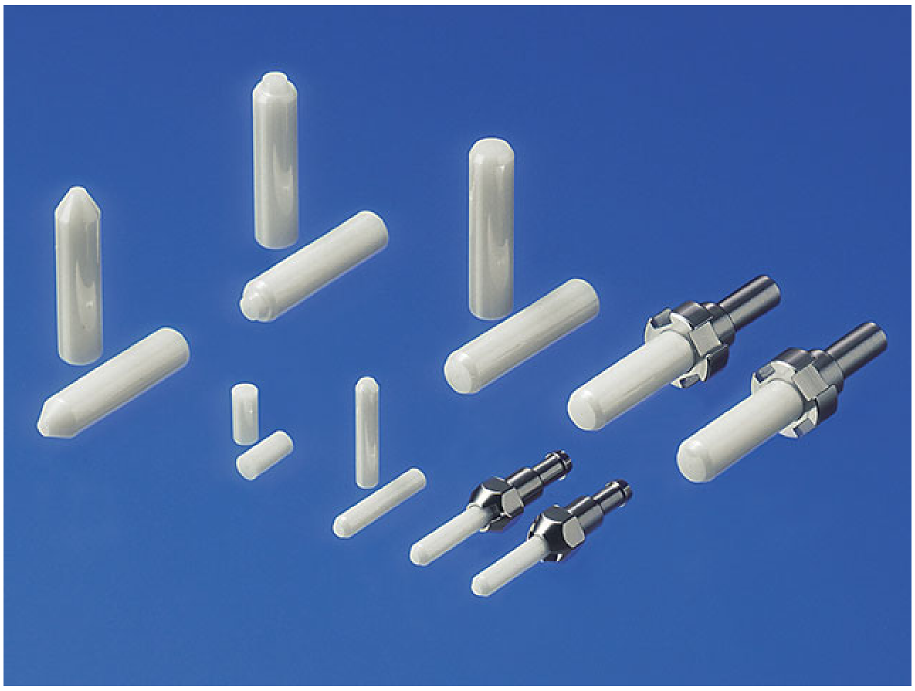


Figure 2.11: Optical fibre connectors - components (Source: [12])

Advantages and disadvantages:

+ slim compared to other cable types;
+ higher speed;
+ much higher bandwidth;
+ data security;
+ immune to EMI;
- expensive;
- wiring a new network is difficult;
- the optical fibre has to be exceedingly smooth in order to minimize losses of the transmitted light.

2.2.3 Network topologies

There are several network topologies, figure 2.12:

- Bus - In a bus type network, nodes are connected one after another.
- Star - Each node is connected individually to a central node.
- Ring - Each node is connected to two other nodes.
- Tree - Combines a bus and a star topology.
- Mesh (Fully connected) - Each node is connected to every other node.
- Point to point - Two same rank nodes are linked together.
- Client / Server - The server is a machine having a higher computation power, while the client nodes have a lower computational ability compared to the server.

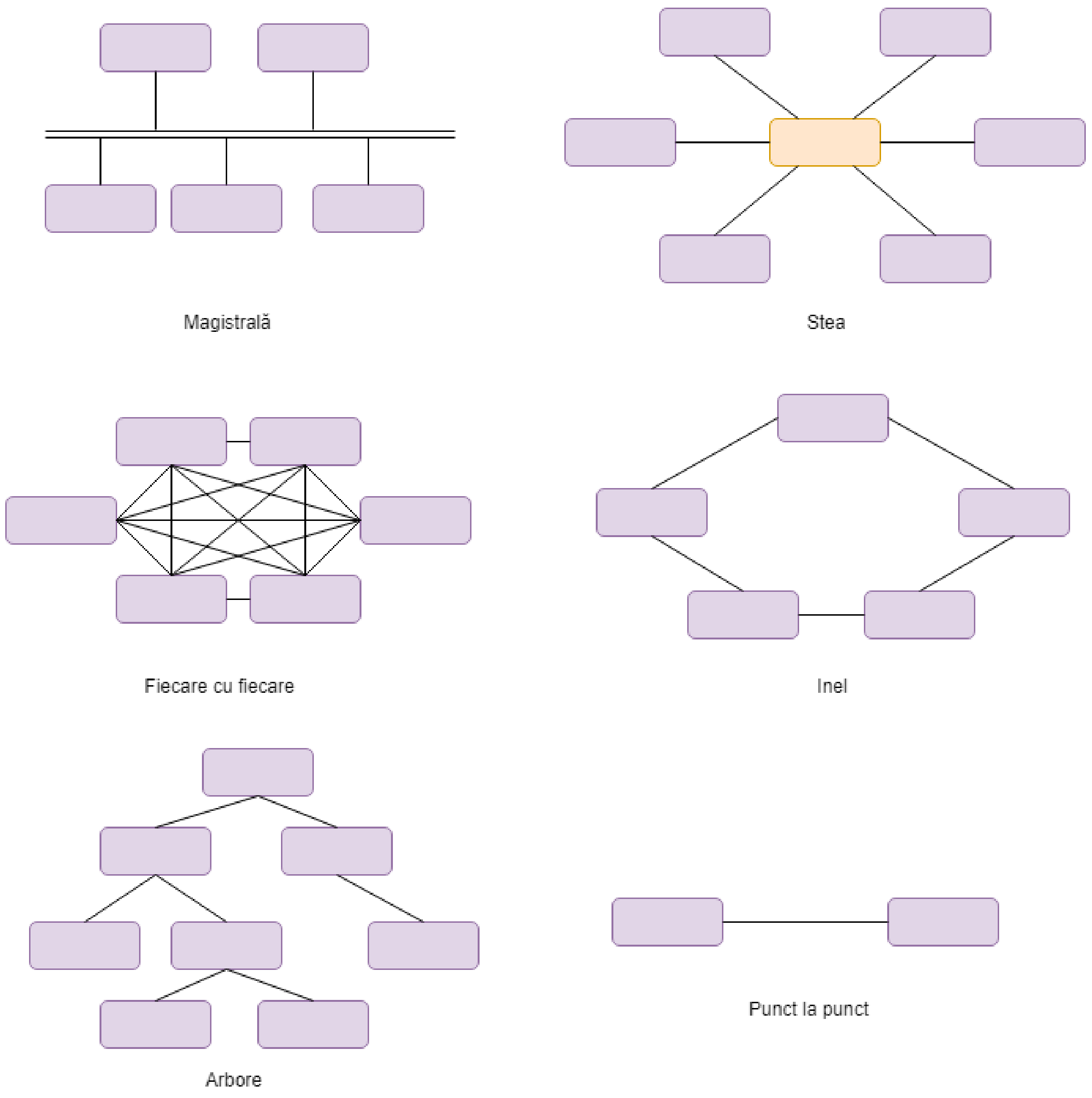


Figure 2.12: Network topologies (Source: [14])

2.2.4 Web applications

Several types of web applications exist nowadays, from classical web pages using back-end technology (php, java) to manage and generate web resources (html, css, js) and up to complex applications (single page applications) executed within a browser, fully isolated from the api module that manages client requests and displays a communication interface that enables the data transfer. Herein we will refer to the latter, for which several actual JavaScript technologies exist that bring along software architecture models, as well as a number of usual functionalities such as: MVC (Model View Controller) architecture, bidirectional link between the data module (JavaScript variables) and graphic interface, extended HTML syntax enabling data to be displayed on the interface.

2.2.5 Client-server architecture

Client-server architecture is a distributed arhitecture consisting of client modules (that manage resources) and servers (that offer the interface enabling access to the resources). Clients and servers are software applications most often executed on different devices. Also, servers operate in tight relationship with various data storage devices (ex: SQL, MongoDB, MySQL servers).

☞ A few examples of what resources may be are: application users, products in an online shop, Twitter/Facebook posts/tweets, Instagram images, emails.

Clients (also known as frontend applications) manage data at graphic interface level and initiate communication sessions with servers (known as backend applications), waiting for requests from them.

2.2.6 TCP/IP

Any device connected to a TCP (Transport Control Protocol)/IP network has a unique identification address known as IP (Internet Protocol).

In order to be able to run several applications and utilities over a network, besides the IP address, a machine (server, computer) needs a number of data transfer ports.

The IP identifies the server, while the port identifies the application or service that is executed on the server.

The following figure showns how two computers are connected over the Internet network by means of IPs and ports.

The ports assume 16-bit values from 0 to 65535 and are being grouped together as follows:

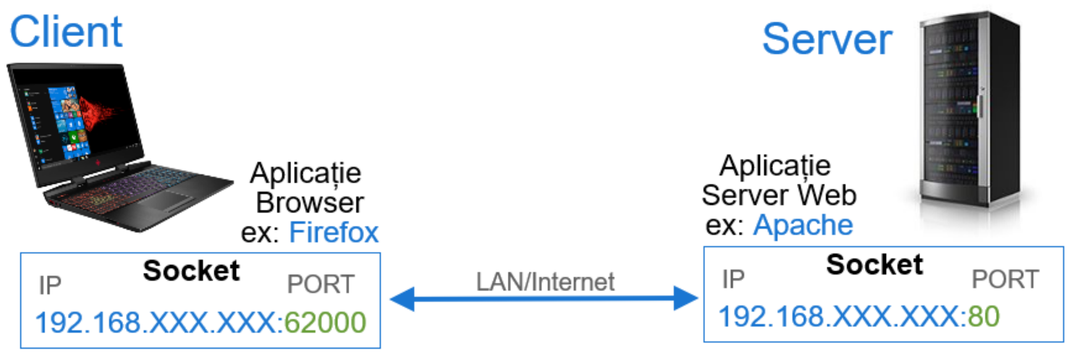


Figure 2.13: Connection between browser (client) application and web server

- 0-1023: ports allotted for system services. The best known are: 80 (web server), 22 (SSH), 20 (FTP), 23 (SMTP);
- 1024-49151: ports reserved for client applications;
- 49152-65535: dynamic/private ports.

A connection between to computers uses a communication socket that can be seen as a IP:port addressing pair. As can be taken from figure 2.13, a mutual connection between computers takes place by means of the sockets.
For instance, when we access the Google site, a source-destination connection takes place between the browser application of our local computer (which was attributed a socket something similar to 192.168.XXX.XXX:62000) and a server from the Google infrastructure supporting the web application (216.58.XXX.XX:80). The ports of the PC are dynamically attributed to the required applications for each communication session with an external socket and can be reused once the session is over.

TCP and UDP are protocols at data transport level, while the IP is implemented at network level. Ports are implemented at data transport level as part of the TCP or UDP message header, figure 2.14.

The TCP/IP protocol supports two types of ports, TCP and UDP. TCP is a connection-oriented transport protocol, having implemented a message-retransmission mechanism in case of error, making sure that all data packages are safely delivered. It is used for text messages, emails, etc.

UDP is used for multimedia data transmission (video, audio) and has no such mechanism implemented as to ensure successful delivery of the data packages.

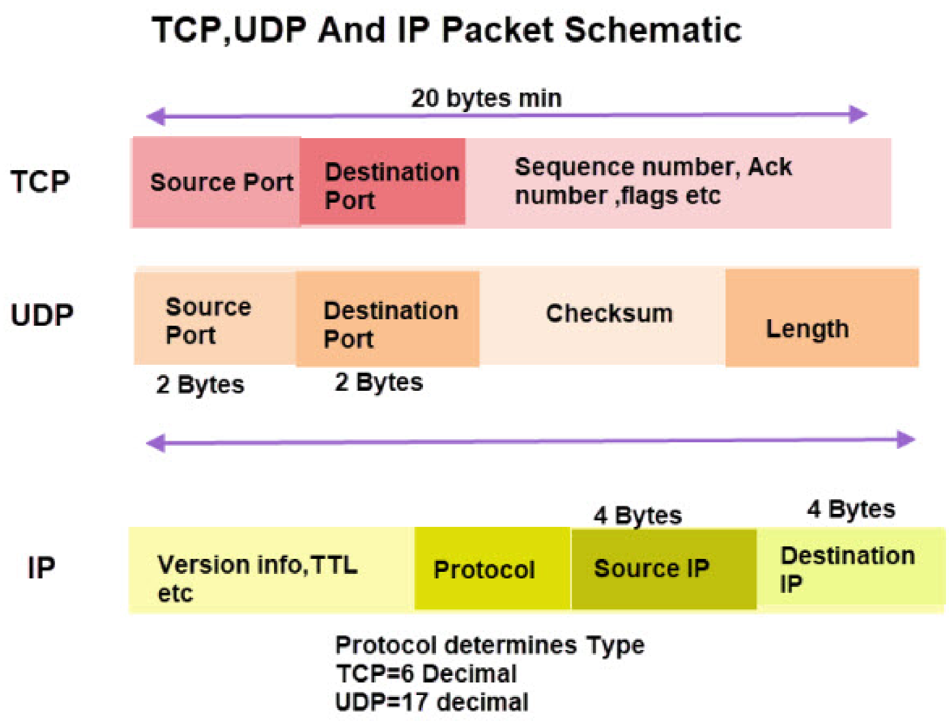


Figure 2.14: TCP, UDP, IP

2.2.7 HTTP vs. WebSocket

Data transfer between various network-connected software applications is usually accomplished by means of communication sockets. These sockets can use communication protocols from the TCP/IP stack, such as HTTP and WebSockets.

HTTP is a unidirectional protocol that allows clients to make so-called requests towards resource-managing servers (perhaps using a data base). The most usual HTTP requests are: GET, PUT, POST and DELETE. These four functions enable reading, saving, updating and deleting resources.

WebSockets is a protocol that enables bidirectional communication between distributed software systems. The most basic example are chat services, where clients send messages to the server, which saves them and in turn forwards them to all interested clients. Nevertheless, websockets enable transmission of audio and video data.

JavaScript offers native support for HTTP and WebSockets protocols for network applications.

2.3 Lab work

Today's lab work will be to execute an application consisting of two modules: a client module that gives the user access to the graphic interface, and a server module that gives the client module access to interface points (API). The two modules work together to accomplish transmission of messages between users and keeping track of the users that send messages by means of the client module. Communication between the two modules may be done in more than one way. Today's choice will be the HTTP and WebSocket protocols.
Resources used:

- vue.js library for the client module
- axios.js library for the client module
- Node.js runtime environment for the server module
- npm - package manager for Node.js

2.3.1 Client application using Vue.js

The **index.html** file contains:

```
<!DOCTYPE html>
<html>

<head>
  <meta content="text/html;charset=utf-8" http-equiv="
     Content-Type">
  <meta content="utf-8" http-equiv="encoding">
  <title>Vue first app</title>
  <link rel="stylesheet" href="style.css">
  <script type="text/javascript" src="libs/vue/vue.js"></
     script>
</head>

<body>
  <div id="app">

    <input class="message" v-model="message" placeholder="
       Name">

    <div class="output-empty" v-if="message == ''">The
       message is empty</div>
    <div class="output" v-if="message != ''"># {{ message
      }} #</div>

    <button class="process-button" v-on:click="process()"
      >Process</button>

  </div>
  <script src="app.js"></script>
</body>

</html>
```

The **app.js** file contains:

```
var app = new Vue({
    el: '#app',
    data: {
        message: ''
    },
    methods: {
        process: function(){
            console.log(this.message);
        }
    }
})
```

The **style.css** file contains:

```
.message {
  border: 1px solid #777;
  padding: 7px;
  margin: 5px;
}

.output-empty {
  padding: 7px;
  margin: 5px;
  color: rgb(255, 104, 34);
}

.output {
  padding: 7px;
  margin: 5px;
  color: rgb(27, 189, 27);
}

```

```
.process-button {
  border: 1px solid #777;
  background-color: #555;
  color: white;
  padding:15px;
}

.process-button:hover{
    background-color: #999;
    cursor:pointer;
}
```

2.3.2 HTTP-based client-server application

This application performs data transmission by means of the HTTP protocol and, as said before, it consists of two distinct modules: the client module and the server module.

2.3.2.1 Server module

The server module is made of 4 files: **api.js**, **run.js**, **users.json** and **package.json**.
The **api.js** file contains:

```
var express = require('express');
var cors = require('cors');
var app = express();
app.use(cors());


var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({
  extended: true
}));
app.use(bodyParser.json());

exports.app = app;
```

The **run.js** file contains:

```
var api = require('./api.js').app;
var users = require('./users.json');

api.get('/', function(request, response) {
  response.json("node.js backend");
});
```

```

api.get('/users', function(request, response) {
  response.json(users);
});

api.put('/users', function(request, response) {
  users[users.length] = request.body;
  response.json('User was saved succesfully');
});


api.delete('/users/:index', function(request, response) {
  users.splice(request.params.index, 1);
  response.json('User with index ' + request.params.index +
      ' was deleted');
});

api.listen(3000, function(){
  console.log('CORS-enabled web server is listening on port
      3000...');
});
```

The **users.json** file contains:

```
[
    {"name":"Cristina", "city":"Sebes"},
    {"name":"Ion", "city":"Turda"},
    {"name":"Sebastian", "city":"Bistrita-Nasaud"}
]
```

The **package.json** file contains:

```
{
  "name": "it-prototype",
  "description": "Node-Express Server",
  "private": true,
  "version": "0.0.1",
  "dependencies": {
    "body-parser": "^1.12.2",
    "cors": "^2.8.5",
    "express": "3.x"
  },
  "main": "Prototype",
  "devDependencies": {}
}
```

To install external dependencies the following command is executed: **npm install** in command line.

To execute the server module, the following command is executed: **node run.js**.

The server answers to requests declared in the run.js file (ex: `http://localhost:3000/users`)

2.3.2.2 Client module

The client module consists of four files: **index.html**, **app.js**, **users.js** and **style.css**. The module includes the vue.js and axios.js external libraries, added as separate files.
The **index.html** file contains:

```html
<!DOCTYPE html>
<html>

<head>
  <meta content="text/html;charset=utf-8" http-equiv="Content-Type">
  <meta content="utf-8" http-equiv="encoding">
  <title>Users manager</title>
  <link rel="stylesheet" href="style.css">
  <script type="text/javascript" src="libs/vue/vue.js"></script>
  <script type="text/javascript" src="libs/axios/axios.js"></script>
  <script type="text/javascript" src="users.js"></script>
</head>

<body>
  <div id="app">

    <ul class="user-list">
      <li v-for="(user, index) in users">
        <div class="user-row">
          <div class="user-name"><b>{{ user.name }}</b></div>
          <div class="user-city"><i>{{ user.city }}</i></div>
        </div>
      </li>
    </ul>

  </div>
  <script src="app.js"></script>
</body>

</html>
```

The **app.js** file contains:

```
var app = new Vue({
    el: '#app',
    data: {
        users: [],
        usersService: null
    },
    created: function () {
        usersService = users();
        usersService.get().then(response => (this.users =
           response.data));
    },
    methods: {
    }
})
```

The **users.js** file contains:

```
function users() {

    get = function() {
        return axios.get("http://localhost:3000/users");
    }

    return {
        get: get
    }
}
```

The **style.css** file contains:

```
ul {
  list-style-type: none;
}

.user-row {
  margin: 20px auto;
  background-color: rgb(255, 254, 234);
  padding: 10px;
  text-align: center;
  width: 400px;
}

.user-row:hover {
  background-color: rgb(255, 251, 175);
}

.user-name {
  width: 150px;
  display: inline-block;
}

.user-city {
  width: 150px;
  display: inline-block;
}
```

In order to execute the client module, there is need for a server to serve the web resources. A simple server can be installed using the node.js platform with the following command (from the client folder):

```
npm install http-server -g
```

After installing the server, the client application is executed with the command:

```
http-server
```

This will be available in any browser, at the following adresses: http://localhost:8080 and http://127.0.0.1:8080.

2.3.3 Client-server application using WebSockets

2.3.3.1 Server

The **package.json** file contains:

```
{
  "name": "chat-server",
  "version": "0.0.1",
  "description": "Data Transmission websocket api",
  "dependencies": {},
  "devDependencies": {
    "socket.io": "^2.2.0"
  }
}
```

The **server.js** file contains:

```javascript
const io = require("socket.io");
const server = io.listen(8000);
let connectedClients = new Map();


server.on("connection", (socket) => {
    console.info(`Client connected [id=${socket.id}]`);
    connectedClients.set(socket);


    socket.on("disconnect", () => {
        connectedClients.delete(socket);
        console.info(`Client [id=${socket.id}] disconnected
           ` );
    });

    socket.on("message-from-client", (payload) => {
        sendMessageToAllOtherClients(socket, payload);
    });

});

function sendMessageToAllOtherClients(sender, message) {
    for (const [client, sequenceNumber] of connectedClients
        .entries()) {
        if (sender.id != client.id) client.emit("message-
           from-server", message);
    }
}
```

Dependencies mentioned in package.json can be downloaded using the command:

```
npm install
```

The server is executed with the following command, in the folder containing the server:

```
node server.js
```

2.3.3.2 Client

The **package.json** file contains:

```json
{
  "name": "chat-client",
  "version": "0.0.1",
  "description": "Data Transmission socket application",
  "dependencies": {},
  "devDependencies": {
    "live-server": "^1.2.1",
    "socket.io-client": "^2.2.0"
  }
}
```

The **client.js** file contains:

```
var socket = io.connect('localhost:8000');

try {

    socket.on('connect', function (data) {
        socket.emit("message-from-client", "Hello to
           everyone from " + checkBrowser());
    });

    socket.on('message-from-server', function (message) {
        console.log(message);
    });

}
catch (err) {
    alert('ERROR: socket.io encountered a problem:\n\n' +
       err);
}

function checkBrowser() {
    var browser = 'Noname browser';
    if (navigator.userAgent.search("Chrome") > -1) {
        browser = "Chrome";
    }
    if (navigator.userAgent.search("Firefox") > -1) {
        browser = "Firefox";
    }
    return browser;
}

document.getElementById("send").addEventListener("click",
   sendMessage);
function sendMessage() {
    var message = document.getElementById("message").value;
    socket.emit("message-from-client", message);
}
```

The dependencies mentioned in package.json can be downloaded with the command:

```
npm install
```

The client is run executing the following command in the client folder:

```
live-server
```

This will be available in any browser, at the following addresses: `http://localhost:8080` and `http://127.0.0.1:8080`.

2.4 Applications and problems

2.4.1 Problem 1 *(1 point)*

- VueJS client application *(0,5 p)*.
- When pressing the 'Process' button, if the value in the message field is equal to '123', then the graphic interface should display 'Message is equal to 123' *(0,5 p)*.

2.4.2 Problem 2 *(4 points)*

- HTTP client-server application *(1 p)*.
- Implement deleting a user *(1 p)*.
- Implement adding a new user *(1 p)*.
- Implement modifying an existing user (based on its index in the element string) *(1 p)*.

2.4.3 Problem 3 *(3 points)*

- Test the WebSocket client-server application, using 2 clients open in different browsers *(1 p)*.
- Implement, using VueJS, a graphic inteface for a chat service (ex: facebook messenger). The server routes the message from a client towards all connected clients, which display it in the browser, not only in the console. The requirement is to keep received/sent messages on the graphic interface *(2 p)*.

BIBLIOGRAPHY

[1] Adina Aştilean. *Data transmission, course notes*

[2] Andrew S. Tanenbaum, David J. Wetherall. *COMPUTER NETWORKS*. Prentice Hall Pub.

[3] https://uk.rs-online.com

[4] https://tri-vcables.com/custom-assemblies/coax/

[5] http://www.l-com.com/content/Article.aspx?Type=L&ID=10057

[6] http://www.coax-connectors.com/

[7] https://cdn.kramerav.com

[8] http://www.berkteklevitontechnologies.com

[9] http://www.groundcontrol.com/galileo/ch5-ethernet.htm

[10] https://www.fiberoptics4sale.com

[11] http://www.netcom-activ.ro/fibra-optica.php

[12] https://global.kyocera.com/prdct/semicon/semi/fiber/

[13] https://www.fs.com/fiber-optic-connector-tutorial-aid-341.html

[14] https://www.orosk.com/

3 Error detection and correction - the Hamming code

3.1 Objectives

- Introducing error detection and correction algorithms;
- Developing an application for error detection and correction using the Hamming code.

3.2 Introduction

3.2.1 Theoretical background

3.2.1.1 The Hamming distance

The Hamming distance (named after Richard Hamming) is being calculated for two vectors of equal length. The Hamming distance is given by the number of positions for which the two vectors are different and indicates the number of errors that transform one vector into the other (ex.: 3.1).

Table 3.1: The Hamming distance - examples

Vector 1	Vector 2	Hamming distance
1101011	1001001	= 2 (different in positions 2 and 6)
1111011	1100111	= 3 (different in positions 3, 4, and 5)

The information is given in binary code (ex.: ASCII code). Regular operations in binary code are shown in Table 3.2.

Table 3.2: Binary code operations: Addition - "OR"; Multiplication - "AND"; Exclusive or - "XOR".

+	0	1	*	0	1	⊕	0	1
0	0	1	0	0	0	0	0	1
1	1	0	1	0	1	1	1	0

3.2.1.2 Hamming linear codes

With linear codes, the code word is usually written in form of a line vector:

$$V = [a_1 a_2 a_3 ... a_n]; \tag{3.1}$$

From the n symbols, k are information symbols and m are control symbols. Coding means to calculate control symbols depending on information symbols, such as to make [V] into a code word. In case of linear codes, control symbols are linear combinations of the information symbols. They can be obtained by means of solving a linear system of m equations with m unknowns:

$$\begin{cases} h_{11} * a_1 \oplus h_{12} * a_2 \oplus ... \oplus h_{1n} * a_n = 0 \\ h_{21} * a_1 \oplus h_{22} * a_2 \oplus ... \oplus h_{2n} * a_n = 0 \\ ... \\ h_{m1} * a_1 \oplus h_{m2} * a_2 \oplus ... \oplus h_{mn} * a_n = 0 \end{cases} \tag{3.2}$$

The equation system (3.2) can be written as follows:

$$[h][V]^T = [0]_{mx1} \tag{3.3}$$

Within the two relations (3.2), (3.3), the product is the usual one, while the sum is modulo two.

The following matrix:

$$H = \begin{bmatrix} h_{11} & h_{12} & \dots & h_{1n} \\ h_{21} & h_{22} & \dots & h_{2n} \\ \dots & & & \\ h_{m1} & h_{m2} & \dots & h_{mn} \end{bmatrix}; \quad h_{ij} \in [0,1]; \tag{3.4}$$

represents the control matrix of the code. Specifying a linear code implies indicating the control matrix [H]. The code's error detection or correction properties can be established by choosing the right [H] matrix. The relation (3.3) represents the coding condition: code words are considered only those binary sequences of n symbols that validate the relation (3.3).

The decoder receives the vector $[V^{'}] = [V] = [\epsilon]$,

where:

$$[\epsilon]^{'}[\epsilon_1\epsilon_2...\epsilon_n]; \tag{3.5}$$

is the received vector, defined by:

$$[\epsilon_i] = \begin{cases} 0, \text{ if there is no error in position "i"} \\ 1, \text{ if there is an error in position "i"} \end{cases} \tag{3.6}$$

The decoding operation consists in computing the corrector:

$$[Z] = [H] * [V^{'}]^T; \tag{3.7}$$

Given the relationship between $[V^{'}], [V]$ and $[\epsilon]$ and the coding relation (3.3), the following can be written:

$$[Z] = [H] * ([V]^{'} * [\epsilon])^T = [H] * [V]^T = [H] * [\epsilon]^T; \tag{3.8}$$

If no errors were introduced, $[\epsilon] = [0]$, so [Z]=[0]. If at least one error was introduced, $[\epsilon] \neq [0]$, thus $[Z] \neq [0]$. For error correction codes, the structure of the corrector allows identification of the number and position of the errors. For binary codes, the correction is made by negating the erroneous symbol (which is accomplished by adding modulo two of 1 to the erroneous symbol).

3.2.2 Applications

3.2.2.1 Hamming code for single error correction

The single error correction Hamming code is a linear code with the following [H] matrix:

$$H = \begin{bmatrix} 0 & 0 & \dots & 0 & 1 \\ 0 & 0 & \dots & 1 & 0 \\ & & & & \\ \dots & & & & \\ & & & & \\ 1 & 1 & \dots & 1 & 1 \end{bmatrix} = [h_1 h_2 \dots h_n]; \tag{3.9}$$

where each h_i column represents the transcription in binary code of the number of the column using n bits.

The code word [V] has the control symbols placed in positions 2^i, i = 0 ... N, thus in positions 1, 2, 4, 8, 16, ...:

$$[V] = [c_1 c_2 a_3 c_4 a_5 a_6 a_7]; \tag{3.10}$$

where c_1, c_2, c_4, ..., c_m denote control symbols and a_3, a_5, a_6, ..., a_n denote information symbols.

Due to this choice, each relation in the (3.2) system will contain a single control symbol, thus enabling easy solving of the equation system with n unknowns, as follows:

$$\begin{cases} c_1 \oplus a_3 \oplus a_5 \oplus ... \oplus = 0 \\ c_2 \oplus a_3 \oplus a_5 \oplus ... \oplus = 0 \\ c_4 \oplus a_3 \oplus a_5 \oplus ... \oplus = 0 \\ ... \end{cases} \tag{3.11}$$

Receiving $[V^{'}]$, the decoder computes $[Z] = [H][V^{'}]^T = [H][\epsilon]^T$. If no error is present, $[\epsilon] = [0]$, thus [Z]=[0]. If there is an error in position i, then:

$$[Z]^{'} = [H] * [\epsilon]^T \tag{3.12}$$

If an error is present, the corrector [Z] represents that particular column in the [H] matrix that corresponds to the error position, thus exactly the position of the error written in binary. Based on the corrector [Z] computed from the received word, the position of the error can be determined and the erroneous bit corrected.

3.2.2.2 Single error correction, double error detection Hamming code

In order to correct single errors and detect double errors, another control bit is added (parity), noted c_0 and calculated by XOR (modulo two) of all the other symbols:

$$c_0 c_1 c_2 a_3 c_4 a_5 a_6 \dots a_n; \tag{3.13}$$

If we denote by $[V_{H1}] = [c_1 c_2 a_3 c_4 a_5 \dots]$ the code word in the single error correction Hamming code and by $[V_{H2}] = [c_0 c_1 c_2 a_3 c_4 a_5 \dots]$ the code word in the single error correction, double error detection Hamming code, the relationship between them becomes obvious:

$$[V_{H2}] = [c_0 V_{H1}]; \tag{3.14}$$

The control matrix for the new code derives from the $[H_1]$ matrix of the single error correction code, by adding a first column of m zeroes (zero written in binary with m bits) and then a last row of n+1 units:

$$H_2 = \begin{bmatrix} 0 & \dots & & & \\ 0 & \dots & & & \\ \dots & & H_1 & & \\ 1 & 1 & \dots & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & \dots & 1 \\ 0 & 1 & \dots & 0 \\ \dots & & & \\ 1 & 1 & \dots 1 & 1 \end{bmatrix}; \tag{3.15}$$

Upon reception, the following is calculated:

$$Z_{H2} = [H_2] * [V_{H2}]^T = \begin{bmatrix} Z_{H1} \\ Z_0 \end{bmatrix}; \tag{3.16}$$

If the received word is $[v^{'}] = [c_0^{'} c_1^{'} c_2^{'} a_3^{'} c_4^{'} a_5^{'} ... a_n^{'}]$, relation (3.16) results into the following:

$$c_0 c_1 c_2 a_3 c_4 a_5 a_6 \dots a_n; \tag{3.17}$$

Comparing (3.13) to (3.17) results into the following:

- if the number of errors that occur over a channel is even, then $z_0 = 0$;
- if the number of errors that occur over the channel is odd, then $z_0 = 1$;
- if two errors at most occur over the transmission channel: if $z_0 = 0$ and $[Z_{H1}] = [0]$, it is decided that the received word contains two errors; if $z_0 = 1$, it is decided that the received word contains a single error in the position that results by transcribing the components of the corrector $[Z_{H1}]$ into decimal;
- if $z_0 = 0$ and $[Z_{H1}] = [0]$, it is decided that the received word is correct.

3.2.3 Examples

3.3 Lab work

3.3.1 Client

The **index.html** file contains:

```
<html>

<head>
    <meta content="text/html;charset=utf-8" http-equiv="
        Content-Type">
    <title>Hamming Encoder</title>
```

```html
    <link rel="stylesheet" type="text/css" href="css/style.css">
</head>

<body>

    <div id="hamming-encoder">

        <h1>Hamming code for
            <select v-on:change="initDataBits()" v-model="numberOfDataBits">
                <option value="4">4</option>
                <option value="8">8</option>
            </select>
        bits</h1>


        <br>
        <ul v-for="(bit, index) in dataBits">
            <li>
                <input maxlength="1" :placeholder="'D'+index" v-model="bit.data"
                    v-bind:class="[validateBit(bit.data) ? 'valid-input' : 'invalid-input']" />
            </li>
        </ul>
        <br>
        Data bits: <code>{{dataBits}}</code>
        <br>

        <button v-on:click="send()">Send</button>

        <p>{{ status }}</p>

    </div>

    <script type="text/javascript" src="libs/vue.js"></script>
    <script type="text/javascript" src="libs/axios.js"></script>
    <script type="text/javascript" src="js/app.js"></script>
</body>
```

```

</html>
```

The **app.js** file contains:

```

var app = new Vue({
    el: '#hamming-encoder',
    data: {
        dataBits: [],
        status: '',
        numberOfDataBits: 4
    },
    created: function () {
        this.initDataBits(4);
    },
    methods: {
        initDataBits: function(){
            this.dataBits=[];

            for(var i=0;i<this.numberOfDataBits;i++){
                var bit = { data: null };
                this.dataBits.push(bit);
            }
        },
        send: function () {
            if (this.validate(this.dataBits) === true){
                var encodedMessage = this.encode(this.
                   dataBits);


                return axios.put("http://localhost:3000/
                   message", {bits: encodedMessage}).then(
                     response => (this.status = response.
                        data)
                );
            } else {
                this.status = 'Input is not valid. Please
                   use 0 or 1 as data bit values';
            }
        },
        encode: function(bits){

```

```

      var c4=this.parity(parseInt(bits[1].data)+parseInt(bits[2].data)+parseInt(bits[3].data));
      var c2=this.parity(parseInt(bits[0].data)+parseInt(bits[2].data)+parseInt(bits[3].data));
      var c1=this.parity(parseInt(bits[0].data)+parseInt(bits[1].data)+parseInt(bits[3].data));
      console.log("Control bits: "+c1+","+c2+","+c4);
      return [c1,c2,parseInt(bits[0].data),c4,parseInt(bits[1].data),parseInt(bits[2].data),parseInt(bits[3].data)];
    },
    parity: function(number){
      return number % 2;
    },
    validate: function(bits){
      for(var i=0; i<bits.length;i++){
        if (this.validateBit(bits[i].data) === false)
        return false;
      }
      return true;
    },
    validateBit: function(character){
      if (character === null) return false;
      return (parseInt(character) === 0 ||
      parseInt(character) === 1);
    }
  }
})
```

The **style.css** file contains:

```
body {
 margin: 30px;
}

select {
 margin-top: -5px;
 padding: 5px;
```

```
}

ul {
  list-style: none;
  display: inline-block;
  margin: 0px;
  margin-bottom: 20px;
  margin-top: 0px;
  padding: 0px;
  padding-right: 10px;
}

input {
  font-size: 16px;
  padding: 10px;
  width: 50px;
  text-align: center;
}

.invalid-input {
  border: 3px solid orange;
  -webkit-transition: all 1000ms linear;
  -ms-transition: all 1000ms linear;
  transition: all 1000ms linear;
}

.valid-input {
  border: 3px solid green;
  background-color: green;
  color: white;
  -webkit-transition: all 1000ms linear;
  -ms-transition: all 1000ms linear;
  transition: all 1000ms linear;
}

button {
  margin-top: 10px;
  padding: 10px;
}

code {
  background-color: #efffec;
  margin: 0px;
```

```
}
```

3.3.2 Server

The **run.js** file contains:

```
var api = require('./api.js').app;
var hamming = require('./hamming.js');

api.put('/message', function(request, response) {
  var bits = request.body.bits;


  var decoded = hamming.decode(bits);
  if(decoded.errorCorrected){
    response.json('One error corrected on position: '+
       decoded.errorPosition);
  }
  response.json('Message received without errors');
});

api.listen(3000, function(){
  console.log('CORS-enabled web server is listening on port
      3000...');
});

function distortBit(bits, index){
  bits[index] = (bits[index]+1) % 2;
  return bits;
}
```

The **api.js** contains:

```
var express = require('express');
var cors = require('cors');
var app = express();
app.use(cors());


var bodyParser = require('body-parser');
app.use(bodyParser.urlencoded({
  extended: true
}));
app.use(bodyParser.json());
```

```

exports.app = app;
```

The **hamming.js** file contains:

```
function decode(bits) {
  var z4=parity(bits[3]+bits[4]+bits[5]+bits[6]);
  var z2=parity(bits[1]+bits[2]+bits[5]+bits[6]);
  var z1=parity(bits[0]+bits[2]+bits[4]+bits[6]);

    var errorPosition=z1*1+z2*2+z4*4;
  var errorDetected=false;
  if (errorPosition!=0) errorDetected=true;
  if (errorDetected) {
    bits[errorPosition-1]=parity(bits[errorPosition-1]+1);
  }
    return { errorCorrected: errorDetected, errorPosition:
       errorPosition-1, bits: bits };
}

parity = function(number){
  return number % 2;
}

exports.decode = decode;
```

The **package.json** file contains:

```
{
  "name": "it-prototype",
  "description": "Node-Express Server",
  "private": true,
  "version": "0.0.1",
  "dependencies": {
    "body-parser": "^1.12.2",
    "cors": "^2.8.5",
    "express": "3.x"
  },
  "main": "Prototype",
  "devDependencies": {}
}
```

3.4 PROBLEMS

3.4.1 PROBLEM *(5 points)*

- Solution for 4, respectively 8 data bits *(2 p)*.
- Solution for n data bits *(3 p)*.

BIBLIOGRAPHY

[1] Adina Aştilean. *Data transmission*, course notes

4 Data compression - Static compression algorithms. The Huffman Algorithm. The Shannon-Fano Algorithm.

4.1 Objectives

- Introducing data compression methods
- Introducing the Huffman data compression algorithm
- Introducing the Shannon-Fano data compression algorithm

4.2 Introduction

Data compression is a coding process that aims to reduce the redundancy of a source-generated message, so as to minimize the resources required to memorize or to transmit that message. Since memorizing or transmitting a message usually implies, perhaps at an intermediary stage, rewriting it into binary symbols, and because most compression methods use binary-binary coding methods, it can be said that compression essentially means to reduce the number of binary symbols required to represent the message.

Regarding the degree to which the reconstituted message resembles the original one, that is to say the degree to which the compression-decompression procedure alters the message, two categories of compression algorithms can be distinguished: with or without losses.

Lossless compression algorithms are reversible, meaning that decompression renders the exact original message. Lossy algorithms are irreversible, meaning that small differences will exist between the original message and the one reconstituted; such algorithms are used for compression of images and of video and audio messages.

In the following we will refer to lossless compression algorithms, used for text messages or for strictly binary data. If the full message is available before initiating the compression process, static compression algorithms can be used, such as static Shannon-Fano and Huffmann coding, which are based on the character frequency within a text. Characters with the highest occurrence probability are assigned shorter code words. With adaptive, dynamic compression algorithms, the compression process takes place in real time, the occurrence probability of characters being constantly approximated along the way, as the message is generated.

4.2.1 THEORETICAL BACKGROUND

Compression techniques are useful for text files, where some characters occur more often than others, for files that contain digitized images, sounds, or analogical signals - types of data that may contain a large number of iterative motifs. Even if the capacity of storage devices is much higher today than it used to be, file compression algorithms are still of crucial importance, given the constantly increasing volume of data that has to be stored. In addition to that, compression plays a vital role in communications, as data transmission is way more expensive than data processing.

One of the most popular compression techniques for text files is the so-called *Huffman encoding method* or *Huffmann code*. The method, invented by D. Huffman in 1952, allows a memory cut of 20% – 90%, depending on the characteristics of the file to be compressed. Instead of using a code in which all characters are assigned fixed length codes of 7 or 8 bits, the most frequent characters will be given shorter codes, while rarer characters will have longer ones.

Definitions:

- *Data compression* – methods and techniques to reduce the volume of a dataset written in binary, in such a way that the initial data can be reconstructed afterwards, either identically or with a certain degree of approximation.
- *Dataset, D* – a progression of symbols with numerical representation (original numerical code) consisting of a number of symbols (volume of the dataset) that is larger than the number of the distinct symbols that make up the dataset.
- *1 symbol* = 8 bytes (ASCII) → the dataset consists of a maximum of 28 bytes = 256 distinct symbols.
- *Alphabet* – the set of distinct symbols in a dataset.
- *Dataset length* – the total number of bits the dataset is represented over
- *Dataset volume* – the number of symbols that make up the dataset.

- *Compression*
  - *conservative* - compression methods that allow exact reconstruction of the dataset starting from its compressed version – text, source software.
  - *non-conservative* - compression methods that allow an approximative, inexact reconstruction of the original dataset – audio signal, image.
- *Compression rate* – parameter measuring the performance of the used compression method:
  $$\gamma = [1 - \frac{c}{o}]100\%; \tag{4.1}$$
  where:
  - c - length of compressed dataset
  - o - length of original dataset
  - γ:
    - $=$ 100%, ideal compression method (highest gain)
    - $=$ 0%, method does not accomplish compression (zero gain).
    - $<$ 0%, method does not accomplish compression (data expansion).
- γ - measures the relative gain in terms of length decrease accomplished with a certain compression method.
- *Compression speed* – $\frac{1}{\gamma}$ (depends on the complexity of the method used, will be a compromise between the two parameters).
- *Compression:*
  - *conservative* - high compression rate, low compression speed
  - *non-conservative* - low compression rate, high compression speed
- *Expansion* – length increase of the processed dataset compared to the original one (in case of inefficient methods).
- *Decompression* – (exact or approximative) reconstruction of the original dataset, starting from its compressed version.
- *Redundancy* – measured by means of a statistical model associated to the dataset, based on the occurrence frequency of the symbols that make up the data alphabet
- *Occurrence frequency* – the rapport between the occurrences of a given symbol and the total number of symbols in the dataset, denoted as v(s) (probability of occurrence of the "s" symbol, P(s)).

- *Entropy* – minimal number of bits required to re-encode symbol "s" so as to avoid confusion with other symbols (may be fractional, but is usually rounded off to the next larger integer). Cannot exceed the initial bit number.

$$\begin{aligned} H(s) = -log_2 v(s) \rightarrow v(s) = 2^{-H(s)} \in (0,1]. \\ 0 < v(s) \leqslant 1 \rightarrow H(s) > 0. \end{aligned} \tag{4.2}$$

4.2.1.1 Evaluation criteria for a code

The objective of any data compression method is to detect and to eliminate redundancies from a dataset. Since in data transmission the cost of exploring a transmission system increases linearly with time, a convenient measure of a code's efficiency is the average length of a word:

$$n = \sum_{i=1}^{n} n_i P_i. \tag{4.3}$$

where:

- P_i are the probabilities associated with the source's alphabet;
- n_i is the number of letters in the code word having the index i;
- n is a parameter related to the compactness of the code: the smaller n is, the greater the degree of compactness. Its value should be thus as small as possible.

The n parameter is limited downwards by the condition that ensures informational entropy per symbol of the code alphabet:

$$n \geq n_{min} = \frac{H}{\lim q} \tag{4.4}$$

where H represents the entropy of the source. Under these conditions, the efficiency of a code is defined by the following formula:

$$\eta = \frac{\overline{\mathrm{n}_{min}}}{\overline{\mathrm{n}}}. \tag{4.5}$$

4.2.1.2 The Huffman code

D. Huffman developed a greedy algorithm that constructs an optimal prefix code, called a Huffman code. The first step in constructing a Huffman code is to compute the occurrences of each character in the text. Sometimes standard character occurrence frequencies can be used, according to the language and the nature of the text.

Let us denote with $C = c_1, c_2, ..., c_n$ the character set in a given text and with $f_1, f_2, ..., f_n$ their respective occurrences. If l_i were the length of the string codifying symbol c_i, then the total length of the representation would be:

$$L = \sum_{i=1}^{n} l_i f_i. \qquad (4.6)$$

The aim is to construct a prefix code that can minimize the expression above. In order to do that, a bottom-up full binary tree will be constructed, as follows:

- We first consider a partition of the set

  $$C = \{\{c_1, f_1\}, \{c_2, f_2\}, ..., \{c_n, f_n\}\},$$

  represented by a forest of single node trees.

- To build the final free, n-1 merging operations will be performed.

Merging of two trees, A_1 and A_2 will result in creating a tree A having a left sub-tree, A_1, a right sub-tree, A_2, and the sum of the root frequencies of the two initial trees as its own root frequency. At every step, the two trees with the lowest root frequencies will be merged.

Construction of the tree progresses from leaves to roots → symbol codes are constructed from the less significant to the most significant bit.

Step 1: Going through the dataset D sequentially and building $A^0 \to N(s)$ for each $s \in A^0$

Step 2: Rearranging the alphabet symbols: $A^0 = \{s_1, s_2, ..., s_n\}$ with $N(s_1) \geq N(s2) \geq ... \geq N(s_n)$. The alphabet elements will be assigned new names and indices as follows:

$$A^0 = \{t_{N+k-1} = s_k\}_{k \in \overline{1,N}}$$

Symbols $t_n, t_{n+1}, ..., t_{2n-1}$ are used for labeling the leaves of the binary tree, while A^0 becomes the first current alphabet of the algorithm, being now named as $A^{0,N}$. Each current alphabet of the algorithm will be named as $A^{0,(N-k)}, k = 0...n-1$ → the tree's leaves are labeled with the symbols in the first current alphabet.

Step 3: Construction of the binary tree is done in parallel with the new current alphabet being established. The new current alphabet, $A^{0,(n-k-1)}$ with k=0 ... N-2, is constructed beginning from the old one $A^{0,(N-k)}$, according to the following rule: two symbols among those with the lowest counters will be removed from the old current alphabet and replaced with a virtual symbol named "γ", whose counter is equal to the sum of the two above.

If there are more than two weak symbols with the same counter, the last two symbols in the alphabet will be picked → the cardinal of the new alphabet will be smaller by one unit than the cardinal of the previous one.

$$\begin{aligned} A^{0,(n-k-1)} &= [A^{0,(n-k)} \left\{t_p, t_q\right\}] \cup \left\{t_{n-k-1}\right\} \\ t_{n-k-1} &= \text{virtual symbol} \\ N(t_{n-k-1}) &= N(t_p) + N(t_q) \\ \#A^{0,(n-k-1)} &= \#A^{0,(n-k)} - 1 = n - k - 1 \end{aligned}$$

The virtual symbol labels an intermediary node of the tree, whose descendants are the two nodes labeled by the removed symbols. Each intermediary node of the tree has two descendants, the left son's counter being at least equal to the right one's.

Step 4: Step 3 is repeated "n-1" times for each current alphabet $A^{0,(n-k)}$, progressing until the tree's root becomes labeled as well (t_1).

Step 5: New indices are assigned to the tree's nodes, so that the sons of every node are labeled with consecutive indices, from left to right, starting from the root.

4.2.1.3 THE SHANNON - FANO CODE

In the Shannon – Fano method, the binary tree associated to the dataset is constructed according to the following algorithm:

Step 1: Going through the dataset (D) sequentially, reading the symbols one by one:

- the 0 order alphabet (A^0) is constructed.
- the occurrences of each symbol
$$s \in A^0$$
are computed, using the counter counter $\rightarrow N(s) \in N$.

$$v(s) = \frac{N(s)}{D}, \forall s \in A^0 (\text{estimation of occurrence probability for symbol s})$$

$$D = \sum_{x \in A^0} N(s) (\text{dataset volume is constant})$$

Step 2: Symbols in A^0 are sorted in descending order of their counter values. Those with equal values will be sorted in lexicographic succession.

Step 3: The A^0 alphabet will be divided into two disjoint sections called sub-alphabets $(A^{0L}$ and $A^{0R})$. Two fundamental conditions have to be taken into consideration:

- order of symbols will be preserved;
- the two sub-alphabets have to be kept as close in weight (size) as possible.

$$A^0 = A^{0L} \cup A^{0R}.$$

$$A^{0L} \cap A^{0R} = 0.$$

$$N(A^{0L}) = \sum_{s \in A^{0L}} N(s) \approx N(A^{0R}) = \sum_{s \in A^{0R}} N(s)$$

Construction of the sub-alphabets starts from the two ends of the A^0 alphabet and progresses towards the middle, permanently weighing the partial sizes of the sub-alphabets against each other, until all symbols of the A^0 alphabet are allocated.

Step 4: Construction of the binary tree. The right branch is assigned the weight "1" and the left branch is assigned the weight "0".

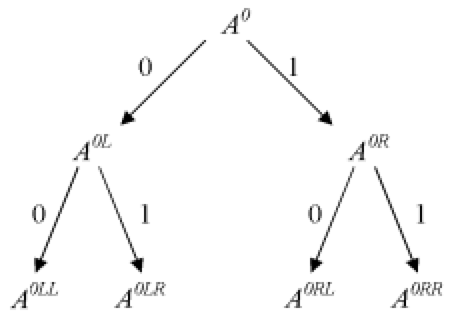


Figure 4.1: Shannon-Fano tree

Step 5: Repeat steps 3 and 4 for each terminal node of the tree (sub-alphabet) that contains more than one symbol.

Stop: The construction process ends when all sub-alphabets of the terminal nodes (leaves) end up contain one single symbol. The symbol code is obtained running through the tree from the root to the respective node and stringing the values of all arches encountered along the path.

Observation: In a dataset, the code of a symbol coming up with high frequency will be short, because its node will be close to the root. Rare symbols, farther down, will have longer codes.

4.2.2 APPLICATIONS

4.2.2.1 THE HUFFMAN ALGORITHM

Construction of the binary tree
Dataset D: "IT IS BETTER LATER THAN NEVER."
The alphabet associated to dataset D is:

A^0		E	T	R	A	I	N	.	B	H	L	S	V
N(s)	5	5	5	3	2	2	2	1	1	1	1	1	1

Construction of the tree is done from the leaves to the root.
Step 1: Each symbol, together with its respective occurrence frequency, receives a label 4.2.

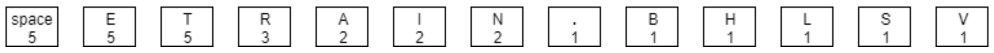


Figure 4.2: Huffman tree - step 1

Step 2: Symbols with the lowest frequency are being merged, 4.3.

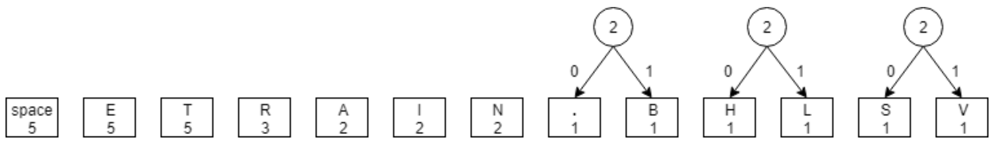


Figure 4.3: Huffman tree - step 2

Step 3: Symbols and/ or nodes with the lowest frequency are being merged, 4.4.

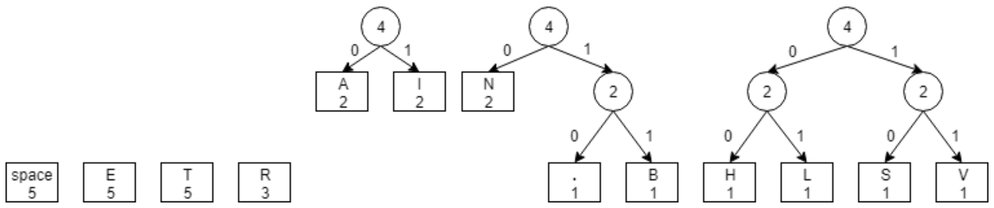


Figure 4.4: Huffman tree - step 3

Step 4: Symbols and/ or nodes with the lowest frequency are being merged, 4.5.

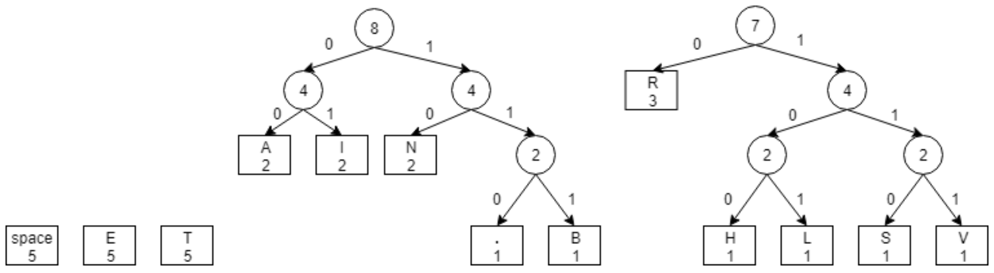


Figure 4.5: Huffman tree - step 4

Step 5: Symbols and/ or nodes with the lowest frequency are being merged, 4.6. If necessary, the tree is being sorted.

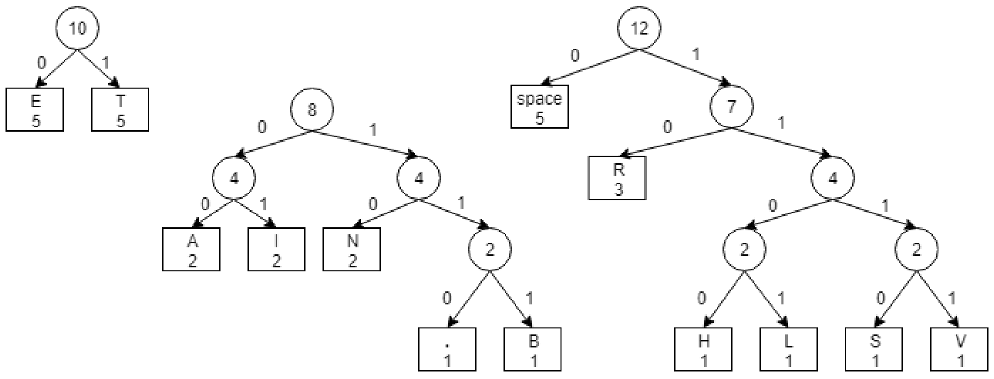


Figure 4.6: Huffman tree - step 5

Step 6: Symbols and/ or nodes with the lowest frequency are being merged, 4.6. If necessary, the tree is being sorted.

Step 7: The final tree, 4.8.

The new codes associated to the dataset D are as follows:

A^0		E	T	R	A	I	N	.	B
New code	00	110	111	010	1000	1001	1010	10110	10111
No. of bits	2	3	3	3	4	4	4	5	5

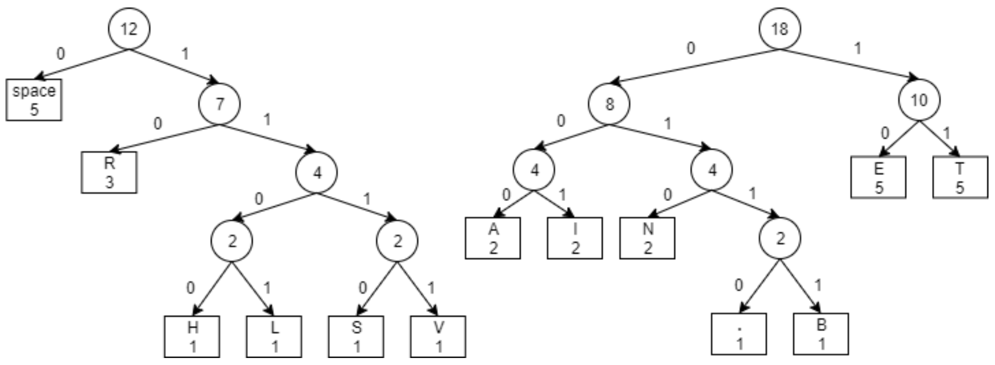


Figure 4.7: Huffman tree - step 6

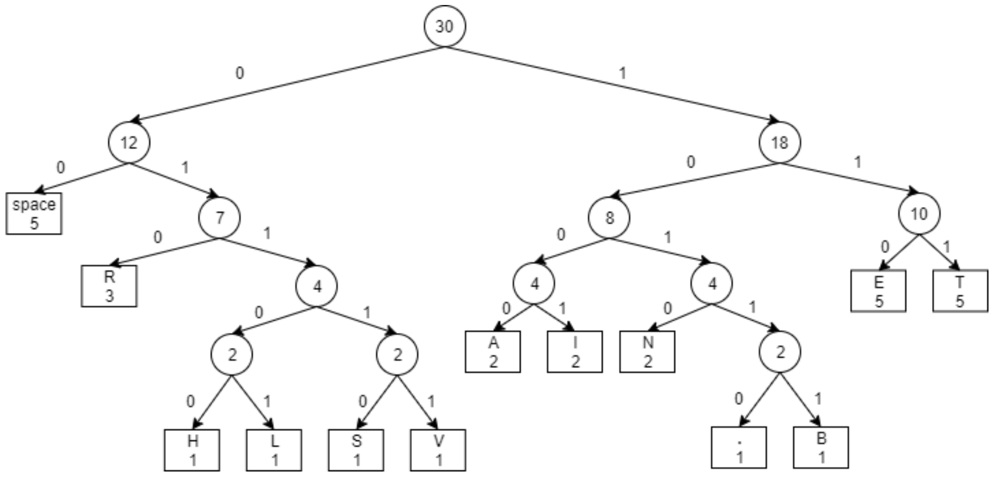


Figure 4.8: Huffman tree

A^0	H	L	S	V
New code	01100	01101	01110	01111
No. of bits	5	5	5	5

Numeric codes are written in decimal, but in reality these are binary codes with a length of 8 bits each. This is not the only way to arrange auxiliary information, but the arrangement manner has to be known in order to successfully decompress the data.

Code	1001	101	00	1001	01110	00	10111	110	111	111
Info	I	T		I	S		B	E	T	T
No. of bits	4	3	2	4	5	2	5	3	3	3

Code	110	010	00	01100	1000	111	110	010	00	111	01100
Info	E	R		L	A	T	E	R		T	H
No. of bits	3	3	2	5	4	3	3	3	2	3	5

Code	1000	1010	00	1010	110	01111	110	010	10110
Info	A	N		N	E	V	E	R	.
No. of bits	4	4	2	4	3	5	3	3	5

4.2.2.2 The Shannon - Fano algorithm

Dataset D: "IT IS BETTER LATER THAN NEVER."
Construction of the binary tree: the alphabet associated with the dataset D is:

A^0		E	T	R	A	I	N	.	B	H	L	S	V
N(s)	5	5	5	3	2	2	2	1	1	1	1	1	1

$N(A^0) = 30$, the binary tree is being constructed over five iterations in the sub-alphabets.

Step 1:

$N(A^{0L}) = 15$

A^{0L}		E	T
N(s)	5	5	5

$N(A^{0R}) = 15$

A^{0R}	R	A	I	N	.	B	H	L	S	V
N(s)	3	2	2	2	1	1	1	1	1	1

Equal weight symbols are being sorted in lexicographic order.

Step 2:

$N(A^{0LR}) = 10$

A^{0LR}	E	T
N(s)	5	5

$N(A^{0RL}) = 7$

A^{0RL}	R	A	I
N(s)	3	2	2

$N(A^{0RR}) = 8$

A^{0RR}	N	.	B	H	L	S	V
N(s)	2	1	1	1	1	1	1

The “space” symbol has the highest occurrence and reached a leaf, being re-codified on 2 bits (00), instead of the original 8-bit code, 0010 0000 or 32).

Step 3:

$N(A^{0RLR}) = 4$

A^{0RL}	A	I
N(s)	2	2

$N(A^{0RRL}) = 4$

A^{0RR}	N	.	B
N(s)	2	1	1

$N(A^{0RRR}) = 4$

A^{0RRR}	H	L	S	V
N(s)	1	1	1	1

The most frequent symbols are being re-codified during this iteration, their new codes having only 3 bits instead of 8.

Step 4:

$N(A^{0RRLR}) = 2$

A^{0RR}	.	B
N(s)	1	1

$N(A^{0RRRL}) = 2$

A^{0RRRL}	H	L
N(s)	1	1

$N(A^{0RRRR}) = 2$

A^{0RRRR}	S	V
N(s)	1	1

Only the most infrequent symbols in the dataset still have to be re-codified, however their respective codes will have 5 bits, not 8.

Step 5:
D = IT IS BETTER LATER THAN NEVER. None of the new codes exceeds 5 bits → redundancy was reduced to a minimum → the highest possible compression rate was achieved.

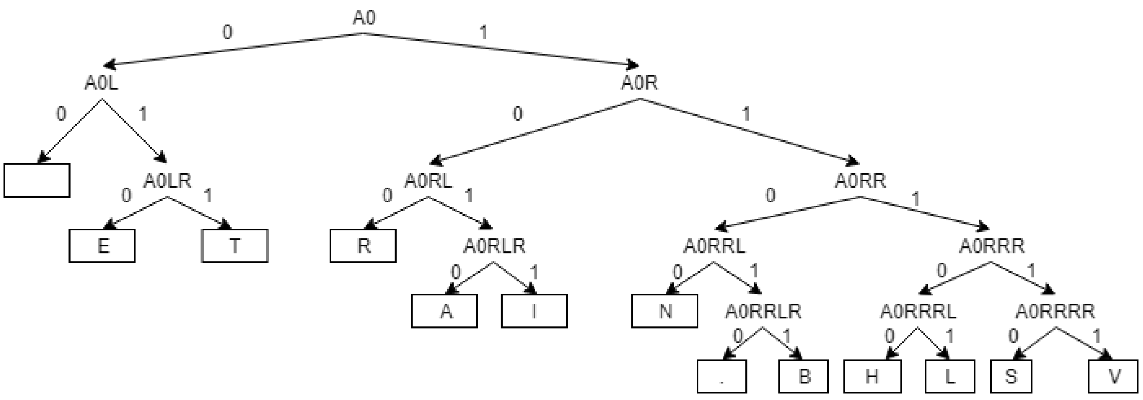


Figure 4.9: Shannon-Fano tree

Numeric codes are written in decimal, but in reality they are binary codes, having a length of 8 bits each. This is not the only way to arrange auxiliary information, but the arrangement manner has to be known in order to successfully decompress the data.

Code	1011	011	00	1011	11110	00	11011	010	011	011	010
Info.	I	T		I	S		B	E	T	T	E
No. bits	4	3	2	4	5	2	5	3	3	3	3

Code	100	00	11101	1010	011	010	100	00	011	11100	1010
Info.	R		L	A	T	E	R		T	H	A
No. bits	3	2	5	4	3	3	3	2	3	5	4

Code	1100	00	1100	010	11111	010	100	11010
Info.	N		N	E	V	E	R	.
No. bits	4	2	4	3	5	3	3	5

The codes for useful and for auxiliary information, respectively, succeed one another without any delimiters. A delimiter might occur in form of a virtual symbol, in the absence of the number of symbols in the alphabet, N. In this case, the code of this information delimiter has to be larger than 255 (256, for instance), but then its representation would take 16 bits, not 8, as N does.

Analysis of compression performance
In the following, we will analyse entropy, that is the number of bits effectively allocated after compression. There are two types of entropy: one regarding auxiliary information $H_a(D)$ and one regarding useful information $H_u(D)$.

Data decompression:

- auxiliary information is being read (5 consecutive bytes indicate a symbol-counter pair);
- the associated binary tree is being built;
- useful information is being read sequentially, bit by bit;
- each symbol in the dataset is being decrypted and a binary tree is used.

Example:

- First bit is 1 → move from A^0 to A^{0R}.
- Second bit is 0 → move to A^{0RL}.
- Third bit is 1 → move to A^{0RLR}.
- Fourth bit is 1 → symbol I will be reached. This will be written down on the output flow.

Reaching a leaf re-initializes the search starting at the tree's root.

Observation: The advantage of this method is that is forgoes the necessity to know the dimensions of the new compression codes, which usually vary from one symbol to another.

4.3 Lab work

The task consists in performing a text compression using the two algorithms described before.
Resources:

- the d3js library for displaying data in form of a binary tree
- npm - Node.js package manager

4.3.1 D3JS APPLICATION

The following application can be used for representing binary trees which is based on one of the "collapsible tree" samples from the D3 library. The root node is A0 and each node has two branches at most.

To install external dependencies the following command is executed: **npm install d3** in the command line.

To install the http server: **npm install -g http-server** in the command line.

To start the server: **http-server &** in the command line.

The **index.html** file contains:

```

<!DOCTYPE html>
<html>
<head>
  <meta content=" text/html;charset=utf -8" http-equiv="
     Content-Type">
  <meta content ="utf-8"http-equiv="encoding">
  <!-- load the style.css -->
  <link rel="stylesheet" href="style.css">
  <!-- load the d3.js library https://d3js.org/d3.v5.min.js
      -->
  <script src="https://d3js.org/d3.v5.min.js"></script>
</head >
<body>
    <!-- arbore.js  - a function to display a tree
                - is based on "collapsible tree" sample from d3
                    library
    -->
    <script src="arbore.js"></script>
</body>
</html>
```

The **style.css** file contains:

```
.node circle {
  fill: #fff;
  stroke: steelblue;
  stroke-width: 3px;
}

.node text {
  font: 12px sans-serif;
}

.link {
  fill: none;
  stroke: #ccc;
  stroke-width: 2px;
}
```

The **arbore.js** file contains:

```
 %
 An example of displaying a compression tree:
 A0 is the tree's root.
 Every level adds a bit at the end of the code.
 Each node will have a maximum of two children (L - left (
    introduces a "0") and R - right (introduces a "1"))
 The node's name contains the name of the node visited
    before.

 ex.: A0LR - level 2, cod: 01
%
var treeData = {
  "name": "A0",
    "children": [
      {
        "name": "Level 1: _",
        "children": [
          { "name": "Level 2: _",
      "children": [
        { "name": "Level 3: _",
        "children": [
            { "name": "Level 4: _ - space (frequency" },
            { "name": "Level 4: _ - 3 (frequency)" }
          ]
        },
        { "name": "Level 3: _ - q (frequency)" }
      ]
      },
          { "name": "Level 2: _ - M (frequency)" }
        ]
      },

      { "name": "Level 1: _",
        "children": [
          { "name": "Level 2: _ - A (frequency)" },
          { "name": "Level 2: _ - * (frequency)" }
        ]
    }
    ]
};


 Defining the dimensions of the tree's display:
var margin = {top: 20, right: 90, bottom: 30, left: 90},
    width = 1960 - margin.left - margin.right,
    height = 800 - margin.top - margin.bottom;

// append the svg object to the body of the page
// appends a 'group' element to 'svg'
// moves the 'group' element to the top left margin

var svg = d3.select("body").append("svg")
    .attr("width", width + margin.right + margin.left)
    .attr("height", height + margin.top + margin.bottom)
  .append("g")
    .attr("transform", "translate("
          + margin.left + "," + margin.top + ")");

var i = 0,
    duration = 750,
    root;

//
 Define tree layout, assign dimensions

var treemap = d3.tree().size([height, width]);

//
 Assign parent, child, height, depth

root = d3.hierarchy (treeData, function(d) { return d.
   children; });
root.x0 = height / 2;
root.y0 = 0;

//
```

```
 Close after second level

root.children.forEach(collapse);

update(root);

//
 Close node and all of its children

function collapse(d) {
  if(d.children) {
    d._children = d.children
    d._children.forEach(collapse)
    d.children = null
  }
}

function update(source) {
  //
  Assign node x and y positions

  var treeData = treemap(root);

  //
  Calculate new layout

  var nodes = treeData.descendants(),
      links = treeData.descendants().slice(1);

  //
  Normalize for fixed depth.

  nodes.forEach(function(d){ d.y = d.depth * 180});

  //***************** Nodes section
     ***************************
  //
  Update nodes ...

  var node = svg.selectAll('g.node')
      .data(nodes, function(d) {return d.id || (d.id = ++i)
         ; });

```

```
// Enter any new modes at the parent's previous position.
 var nodeEnter = node.enter().append('g')
     .attr('class', 'node')
     .attr("transform", function(d) {
       return "translate(" + source.y0 + "," + source.x0 +
           ")";
   })
   .on('click', click);


 Add circle for each node.

 nodeEnter.append('circle')
     .attr('class', 'node')
     .attr('r', 1e-6)
     .style("fill", function(d) {
         return d._children ? "lightsteelblue" : "#fff";
     });


 Add label for each node.

 nodeEnter.append('text')
     .attr("dy", ".35em")
     .attr("x", function(d) {
         return d.children || d._children ? -13 : 13;
     })
     .attr("text-anchor", function(d) {
         return d.children || d._children ? "end" : "start
            ";
     })
     .text(function(d) { return d.data.name; });


 var nodeUpdate = nodeEnter.merge(node);


 nodeUpdate.transition()
   .duration(duration)
   .attr("transform", function(d) {
       return "translate(" + d.y + "," + d.x + ")";
    });

```

```

Update node attributes and styles.

nodeUpdate.select('circle.node')
  .attr('r', 10)
  .style("fill", function(d) {
      return d._children ? "lightsteelblue" : "#fff";
  })
  .attr('cursor', 'pointer');


Remove every exiting node.

var nodeExit = node.exit().transition()
    .duration(duration)
    .attr("transform", function(d) {
        return "translate(" + source.y + "," + source.x +
            ")";
    })
    .remove();


Upon exit reduce circle dimension to 0.

nodeExit.select('circle')
  .attr('r', 1e-6);


nodeExit.select('text')
  .style('fill-opacity', 1e-6);


***************** link section **********************

var link = svg.selectAll('path.link')
    .data(links, function(d) { return d.id; });


var linkEnter = link.enter().insert('path', "g")
    .attr("class", "link")
    .attr('d', function(d){
      var o = {x: source.x0, y: source.y0}
      return diagonal(o, o)
```

```
        });


    var linkUpdate = linkEnter.merge(link);


    linkUpdate.transition()
        .duration(duration)
        .attr('d', function(d){ return diagonal(d, d.parent)
           });


    var linkExit = link.exit().transition()
        .duration(duration)
        .attr('d', function(d) {
          var o = {x: source.x, y: source.y}
          return diagonal(o, o)
        })
        .remove();


    nodes.forEach(function(d){
      d.x0 = d.x;
      d.y0 = d.y;
    });



    function diagonal(s, d) {
      path = `M ${s.y} ${s.x}
              C ${(s.y + d.y) / 2} ${s.x},
                ${(s.y + d.y) / 2} ${d.x},
                ${d.y} ${d.x}`
      return path
    }


    function click(d) {
      if (d.children) {
          d._children = d.children;
          d.children = null;
        } else {
          d.children = d._children;
```

```
            d._children = null;
        }
      update(d);
    }
}
```

4.3.1.1 The Huffman algorithm

The Huffman algorithm - data sequence for the text: "IT IS BETTER LATER THAN NEVER". The **arbore.js** file contains:

```

binary tree "IT IS BETTER LATER THAN NEVER"

var treeData =
  {
    "name": "A0 = 30",
    "children": [
      {
        "name": "Level 1: 18",
        "children": [
          { "name": "Level 2: 8",
              "children": [
                { "name": "Level 3: 4",
                    "children": [
                      { "name": "Level 4: 2",
                          "children": [
                            { "name": "Level 5: B" },
                            { "name": "Level 5: ." }
                          ]
                      },
                      { "name": "Level 4: N"}
                    ]
                },
                { "name": "Level 3: 4",
                    "children": [
                      { "name": "Level 4: I"},
                      { "name": "Level 4: A" }
                    ]
                }
              ]
          },
          { "name": "Level 2: 10",
              "children": [
                { "name": "Level 3: T"},
                { "name": "Level 3: E" }
              ]
          }
        ]
      },
      { "name": "Level 1: 12",
        "children": [
          { "name": "Level 2: 7",
              "children": [
                { "name": "Level 3: 4",
                    "children": [
                      { "name": "Level 4: 2",
                          "children": [
                            { "name": "Level 5: V" },
                            { "name": "Level 5: S" }
                          ]
                      },
                      { "name": "Level 4: 2",
                          "children": [
                            { "name": "Level 5: L" },
                            { "name": "Level 5: H" }
                          ]
                      }
                    ]
                },
                { "name": "Level 3: R" }
              ]
          },
          { "name": "Level 2: space" }
        ]
    }
    ]
  };
```

4.3.1.2 The Shannon-Fano algorithm

The Shannon-Fano algorithm - data sequence for the text: "IT IS BETTER LATER THAN NEVER". The **arbore.js** file contains:

```

binary tree "IT IS BETTER LATER THAN NEVER"
```

```
var treeData =
  {
    "name": "A0",
    "children": [
      {
        "name": "Level 1: A0R",
        "children": [
          { "name": "Level 2: A0RR",
              "children": [
                { "name": "Level 3: A0RRR",
                    "children": [
                      { "name": "Level 4: A0RRRR",
                          "children": [
                            { "name": "Level 5: A0RRRRR = V" },
                            { "name": "Level 5: A0RRRRL = S" }
                          ]
                      },
                      { "name": "Level 4: A0RRRL",
                          "children": [
                            { "name": "Level 5: A0RRRLR = L" },
                            { "name": "Level 5: A0RRRLL = H" }
                          ]
                      }
                    ]
                },
                { "name": "Level 3: A0RRL",
                    "children": [
                      { "name": "Level 4: A0RRLR",
                          "children": [
                            { "name": "Level 5: A0RRLRR = B" },
                            { "name": "Level 5: A0RRLRL = ." }
                          ]
                      },
                      { "name": "Level 4: A0RRLL = N" }
                    ]
                }
              ]
          },
          { "name": "Level 2: A0RL",
              "children": [
                { "name": "Level 3: A0RLR",
                    "children": [
                      { "name": "Level 4: A0RLRR = I" },
                      { "name": "Level 4: A0RLRL = A" }
                    ]
                },
                { "name": "Level 3: A0RLL = R" }
              ]
          }
        ]
      },
      { "name": "Level 1: A0L",
        "children": [
          { "name": "Level 2: A0LR",
              "children": [
                { "name": "Level 3: A0LRR = T" },
                { "name": "Level 3: A0LRL = E" }
              ]
         },
          { "name": "Level 2: A0LL = spatiu" }
        ]
    }
    ]
  };
```

4.4 Suggested problems

1. Implement the Huffman compression algorithm. Display the obtained compression tree.
2. Implement the Shannon-Fano compression algorithm. Display the obtained compression tree.
3. For a keyboard-generated character string:
    - Apply the Huffman compression algorithm;
    - Calculate the compression rate;
    - Display the new codes for each symbol.

4. For a keyboard-generated character string:
    - Apply the Shannon-Fano compression algorithm;
    - Calculate the compression rate;
    - Display the new codes for each symbol.

BIBLIOGRAPHY

[1] Adina Aştilean. *Data transmission,* course notes.

5 DATA ENCODING TECHNIQUES

5.1 OBJECTIVES

- Accomplishing receiver-transmitter synchronization in case of asynchronous and synchronous transmission. General presentation.
- General presentation of non-return-to-zero, return-to-zero, and biphase line codes.

5.2 INTRODUCTION

5.2.1 THEORETICAL BACKGROUND

Transmission of a signal can be done directly in its baseband, if the unaltered signal with its original spectrum, as delivered by the sensing device that picks up the information from the real environment and changes it into a signal (such as gauges for various physical quantities, microphones for audio signals, cameras for video signals) is used as input to the transmission channel. Baseband transmission does not use carrier wave modulation of the signal, which would result in a shift of the initial frequency spectrum.

This type of transmission has the advantage of being simple, since the signals go through no further processing before being transmitted, but there are a number of disadvantages as well. The relative frequency band is wide (relative band = signal band to average frequency ratio) and is easily influenced by perturbations in a wide frequency range. Baseband transmission is not possible over long distances, because of the low immunity to perturbations that occur especially in the lower part of the spectrum.

Digital baseband transmission generally means sending the rectangular signal directly through the communication channel, with two tension (current) values representing logical ”0” and logical ”1”, respectively. Since data transmission is a serial event, the receiver’s input will be nothing more than a succession of two distinct tension values, from which the device has to discriminate not only the beginning and the end of each bit cell (bit synchronization), but also the global packaging of the string into characters and data blocks (character and frame synchronization). Emitter-receiver synchronization (self-synchronization) can be accomplished in two ways, depending on the relationship between the transmitter's and the receiver's clocks.

In case of asynchronous transmission, the two clocks are independent signals. Each characters is treated independently and the receiver has to be resynchronized with every beginning character.

In case of synchronous transmission, the receiver's clock is in a tight phase relationship with the transmitter's clock. Each frame will be transmitted continuously, without breaks between characters, all that has to be done being to ensure the clocks' dependency at the beginning of each character. Asynchronous transmission is usually employed in situations when the transferred data are generated at random intervals, the line being inactive for long and unpredictable periods.

Each character is framed between a ”START” bit and 1, ½, or 2 “STOP” bits. “START” and “STOP” bits have different polarity, so as to ensure at least one 0→1→0 transition between two successive characters arriving continuously, with no break. The first 1→0 transition after an inactive spell is used by the receiver in order to determine the moments when bit samples are received. Character reception begins on the negative front of the “START” bit.

Bit synchronization is accomplished by determination of the “START” transition with the highest possible precision. The receiver clock's period must be at least 16 times smaller than the time it takes for one bit to be transmitted, in order to minimize the start of character detection error, which occurs due to the lack of synchronism between the transitions on the reception line and the transitions of the receiver's tact. To illustrate this, figure 5.1 shows a situation where the clock's period is only 4 times smaller than the time required to transmit one bit.

Character synchronization, meaning establishing the start and the end of a character, is accomplished simply by counting the received bits and checking the validity of the final ”STOP” bits, which should have the logical value of 1. This is possible because the number of bits per character is a transmission characteristic, part of the configuration of both the transmitter and the receiver.

Frame (character block) synchronization is accomplished in a distinct way, depending on the nature of the transmitted information: string of ASCII codes (text) or binary data.

Binary codes may be classified into four categories:

1. NRZ ('Non Return to Zero') codes;

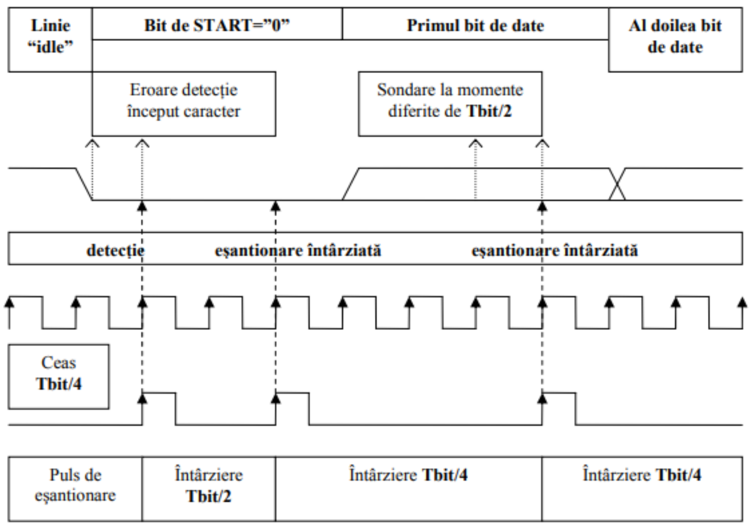


Figure 5.1: Occurrence and propagation of character start detection error (Source:[2])

2. RZ ('Return to Zero') codes;
3. PE ('Phase Encoded' or 'Split Phase') codes;
4. MLB ('Multi Level Binary') codes.

Another classification of codes can be done according to the sign of the associated tensions or currents: if the signs are different for "0" and for "1", respectively, the code is called bipolar; a code with the same sign for both values is called unipolar. Bipolar codes allow using ground level as reference.

5.2.1.1 NRZ ('Non Return to Zero') codes

This type of encoding uses two different tension levels. A logical "1" is represented by a positive tension (+V), while a logical "0" corresponds either to a zero tension (0V) in unipolar NRZ codes or to a negative tension (-V) in the bipolar version.

NRZ encoding allows several variants:

1. NRZ-L (Level): equivalent to NRZ ("1" - high level, "0" – low level);
2. NRZ-M (Mark): "1" - transition, "0" – no transition;
3. NRZ-S (Space): "1" – no transition, "0" – transition.

The highest theoretical flow that can be reached during a NRZ transition is twice the signal's frequency band (2 bits/Hertz can be transmitted). The main disadvantage of

NRZ coding is the lack of transitions during long sequences of identical bits, which can lead to the loss of synchronization with the receiver.

Utes: Fast Ethernet (100 Base Fx), FDDI.

5.2.1.2 RZ CODES

This code family includes an additional information regarding the transmission clock, by the return to zero of the tension after a half-period of the clock (at the middle of the bit). The RZ-unipolar line code includes positive clock pulses only for logical ”1” bits, while logical ”0” bits lack such pulses completely. In RZ-bipolar line codes, signals for ”0” bits include negative clock pulses. The RZ-bipolar signal has the disadvantage of requiring two decision thresholds in order to discriminate for 3 distinct tension levels, but recovery of the bit clock at reception is almost direct, as long sequences of both ”0” and ”1” have inbuilt clock pulses.

5.2.1.3 BIPHASE ENCODING

Three types of biphase encoding methods are used:

1. Biphase - level - $BI\Phi - L$ - also known as Manchester encoding.
2. Biphase - mark - $BI\Phi - M$. In this encoding method, a transition occurs at the beginning of every bit interval. If the bit is a ”1”, then a second transition occurs in the middle of the bit interval. No transition occurs during transmission of a ”0” bit.
3. Biphase - space - $BI\Phi - S$ is the exact reverse of $BI\Phi - M$ (a transition at the beginning and another one after the first half of ”0” bits, and no transitions for ”1” bits).

Manchester encoding

Manchester encoding is based upon determining a transition for the emitted signal that should occur at the middle of the bit interval. Thus, a ”1” is represented by a transition from +V to –V, while a –V to +V transition corresponds to a ”0”. Obviously, this method enables self-synchronization even for long ”0” or ”1” sequences. Even more so, given that binary symbols are represented by transitions and not by static levels, as is the case in NRZ encoding, the chance of transmission medium induced errors becomes incomparably lower. A noise affecting the signal may alter the levels of transmission, but the likelihood of its reversing or canceling a transition and thus inducing reception errors is indeed very low.

The disadvantage of Manchester encoding consists in the fact that sending a signal with a given binary flow requires twice the bandwidth that would be required with other encoding methods. For instance, sending with a 10 Mbps flow requires a 10 MHz bandwidth. This is an inconvenience that makes Manchester encoding difficult to use when high flows are required.

Uses: Ehernet 10Base5, 10Base2, 10BaseT, 10 BaseFL

Differential Manchester encoding

Differential Manchester encoding is based on the presence or absence of a transition at the start of a tact interval. Thus, a ”1” bit is represented by the lack of a transition, while a transition signifies a ”0” bit. Advantages and disadvantages of this encoding method are much the same as for the undifferentiated Manchester method.

Uses: Token-Ring-type networks.

5.2.1.4 MLB ('Multi Level Binary') codes

MLB code types:

1. Bipolar AMI (AMI-Alternate Mark Inversion);
2. HDB-3 (High Density Bipolar Order 3);
3. B8ZS (Bipolar with 8 Zeros Substitution);
4. 4B/5B NRZI;
5. MLT-3 (MultiLevel Transmission-3).

AMI (AMI-Alternate Mark Inversion) bipolar encoding

Principle: zeroes are represented by a zero potential (no electric signal over the line), while ”1” bits are represented alternately by positive (+V) and negative (-V) tensions. With this encoding method, long intervals with no signal may occur (for long ”0” sequences), which may lead to a loss of synchronization.

There is also the reverse version of the method above, namely the pseudoternary encoding, where a lack of signal indicates a ”1” bit and ”0” is represented by alternating positive and negative potentials. Departing from the AMI encoding method, new encoding techniques were developed that tend to replace the former in modern data transmission systems.

Uses: ADSL (Additional Digital Subscriber Loop) transmission

HDB-3 (High Density Bipolar Order 3) encoding

The aim here is to avoid the desynchronizations that might occur with long “0” sequences. The following workaround is used to fix this inconvenience: if a four zero sequence occurs, the last bit is replaced with a tension having the same polarity as the last “1” bit. However, this approach may lead to the occurrence of a significant continuous component. For instance, the 100000000 sequence could be encoded as +000+000+. To avoid such situations, each modified bit has to be chosen as to be of the opposite sign than the one before.

For the same reason, to avoid dealing with a signal with long continuous components, a few rules have to be followed:

- if the number of ”1”s after the last modified sequence is even, then a group of 4 consecutive zeroes will be replaced with a ”+00+” sequence (if the last non-zero level before this sequence was negative. If it was positive, the replacement sequence will be ”-00-”);
- if the number of ”1” s following the last modified sequence is odd, then a group of four consecutive zeroes will be replaced with a ”000+” sequence (if the last non-zero level before this sequence was negative. If it was positive, the replacement sequence will be „000-”).

Uses: Standards E1, E3

B8ZS (Bipolar with 8 Zeros Substitution) encoding

Based on the AMI code, 8 consecutive zero sequences are replaced with sequences containing transitions, enabling thus self-synchronization of the transmission.

So:

- if the impulse prior to the ”0” sequence is positive, then the corresponding code is 000+-0-+;
- if the impulse prior to the ”0” sequence is negative, then the corresponding code is 000-+0+-.

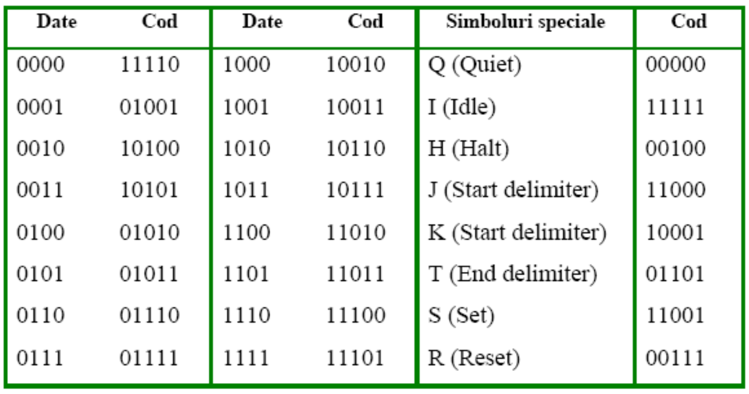
This encoding techniques introduces two changes of the “+-“ alternance, a situation that is quite improbable to be caused by a noise.

Uses: Standard T1 (fast voice and data transmission over twisted pair or coaxial cable).

4B/5B NRZI encoding

This method is a combination between two different encoding algorithms. To understand the relevance of this choice, let us first consider the alternative of the simple NRZ scheme. With NRZ, one state of the system represents a binary ”1” and the other one a binary ”0”. The disadvantage here is the impossibility of self-synchronization. With the 4B/5B scheme, encoding is done 4 bits at a time. Every group of 4 signal bits are encoded into a word made of 5 code bits. This encoding method’s efficiency is of 80%: an 125 MHzs bandwidth has to be available for sending data with a 100 Mbps flow. It should be noted that, with this method, the exact manner of how the code word's bits are transmitted remains unspecified. The NRZI or the MLT-3 encoding methods are usually employed, with a required bandwidth of 62.5 MHz or of 31.25 MHz, respectively.

Since for the encoding of the 16 possible 4-bit combinations only 16 of the 32 possible 5-bit combinations are used, 16 other 5-bit groups remain available. The code words are chosen so as to have no more than two successive zeroes and no less than two transitions (two ”1”s in the NRZI method) in a 5-bit code word. The next table is used for the encoding process:



Date	Cod	Date	Cod	Simboluri speciale	Cod
0000	11110	1000	10010	Q (Quiet)	00000
0001	01001	1001	10011	I (Idle)	11111
0010	10100	1010	10110	H (Halt)	00100
0011	10101	1011	10111	J (Start delimiter)	11000
0100	01010	1100	11010	K (Start delimiter)	10001
0101	01011	1101	11011	T (End delimiter)	01101
0110	01110	1110	11100	S (Set)	11001
0111	01111	1111	11101	R (Reset)	00111

Figure 5.2: 4B/5B encoding

Let's consider for example the binary entry sequence 10000101111. This will be split in 4-bit groups that will be encoded according to the table:

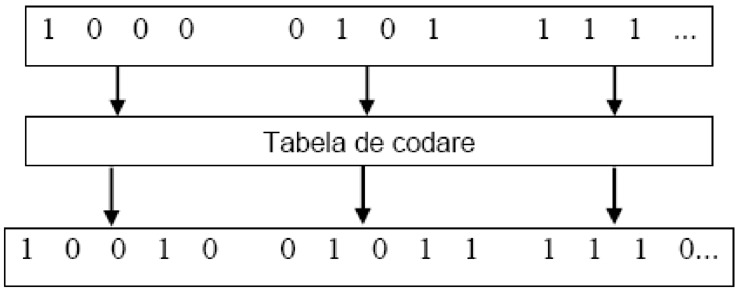


Figure 5.3: 4B/5B encoding example

Code groups that remain unused are either declared invalid or become transmission control symbols, as is shown in the coding table.

Uses: FDDI, 100Base-X

MLT-3 (MultiLevel Transmission-3) encoding

Albeit 4B/5B-NRZI is efficient over optical fibre, it cannot be used for twisted pair cable transmission. The UTP cable acts upon the signal much like a low pass filter, attenuating the high frequency components and strongly distorting the data sequence.

The principle of MLT encoding is the following:

- ”1” bits induce a state change of the signal transmitted over the line, while the ”0” bits leave the state corresponding to the previous bit unaltered
- ”1” bits are encoded successively into 3 states: +V, 0, -V.
- The great advantage of this method is the significant reduction of the bandwidth required for a given flow by using the three states: only 25MHz for a 100 Mbps flow.

Uses: Fast Ethernet 100BaseTx, 100BaseT4

5.2.2 Applications

The task is to represent a sequence of bits using js libraries and packages.
Resources used:

- axios library
- vue library
- npm - Node.js package manager

To install external dependencies the following command is executed: **npm install ...** in the command line.

To install the server http: **npm install -g http-server** in the command line.

To start the server: **http-server &** in the command line.

The **index.html** file contains:

```
<html>
<head>
    <meta content="text/html;charset=utf-8" http-equiv="
       Content-Type">
    <title>Baseband Encoder</title>
    <link rel="stylesheet" type="text/css" href="css/style.
       css">
```

```html
</head>
<body>
    <div id="baseband-encoder">
        <ul v-for="(bit, index) in bits">
            <li>
                <input
                    v-on:change="encode()"
                    maxlength="1" :placeholder="'B'+index" v-model="bit.value"
                    v-bind:class="[validateBit(bit.value) ? 'valid-input' : 'invalid-input']" />
            </li>
        </ul>
        <p>Bitstream data model: <code>{{bits}}</code></p>
        <p>Baseband code representation: <code class="encoded">{{encodedBits.join('')}}</code></p>
    </div>

    <script type="text/javascript" src="libs/vue.js"></script>
    <script type="text/javascript" src="libs/axios.js"></script>
    <script type="text/javascript" src="js/helpers.js"></script>
    <script type="text/javascript" src="js/baseband-codes.js"></script>
    <script type="text/javascript" src="js/app.js"></script>
</body>
</html>
```

The **style.css** file contains:

```css
body {
  margin: 30px;
}

select {
  margin-top: -5px;
  padding: 5px;
}

ul {
  list-style: none;
  display: inline-block;
  margin: 0px;
  margin-bottom: 20px;
  margin-top: 0px;
  padding: 0px;
  padding-right: 0px;
}

input {
  font-size: 16px;
  padding: 10px;
  width: 50px;
  text-align: center;
}

.invalid-input {
  border: 2px solid green;
  -webkit-transition: all 1000ms linear;
  -ms-transition: all 1000ms linear;
  transition: all 1000ms linear;
}

.valid-input {
  border: 2px solid green;
  background-color: green;
  color: white;
  -webkit-transition: all 1000ms linear;
  -ms-transition: all 1000ms linear;
  transition: all 1000ms linear;
}

button {
  margin-top: 10px;
  padding: 10px;
}

code {
  background-color: #efffec;
  margin: 0px;
}

.encoded{
  letter-spacing: -6px;
  font-size: 30px;
  font-weight:bold;
}
```

The **app.js** file contains:

```
var app = new Vue({
    el: '#baseband-encoder',
    data: {
        bits: [],
        encodedBits: [],
        status: '',
        numberOfBits: 8,
        validateBit: validateBit
    },
    created: function () {
        this.bits = getBitstream(this.numberOfBits);
    },
    methods: {
        encode: function(){
            this.encodedBits = getManchesterLevelEncoding(
               this.bits);
        }
    }
})
```

The **helpers.js** file contains:

```
var validate = function (bits) {
        for (var i = 0; i < bits.length; i++) {
                if (this.validateBit(bits[i].value) ===
                   false)
                        return false;
        }
        return true;
}

var validateBit = function (character) {
        if (character === null) return false;
        return (parseInt(character) === 0 ||
                parseInt(character) === 1);
}

function getBitstream(n) {
        result = [];
        for (var i = 0; i < n; i++) {
                let bit = { value: null };
                result.push(bit);
        }
        return result;
}
```

The Manchester code is represented in **baseband-codes.js**:

```
function getManchesterLevelEncoding(bits) {
    var result = [];
    for (var i = 0; i < bits.length; i++) {
        let symbol = '|--|--|';
        if (parseInt(bits[i].value) == 1) symbol = '|__|--|';
        if (parseInt(bits[i].value) == 1 && i > 0 && parseInt(bits[i - 1].value) == 1) symbol = '|__|--|';
        if (parseInt(bits[i].value) == 0) symbol = '|--|__|';
        if (parseInt(bits[i].value) == 0 && i > 0 && parseInt(bits[i - 1].value) == 0) symbol = '|--|__|';
        result.push(symbol);
    }
    return result;
}
```

5.3 Lab work

1. Considering the following sequence: 1100100001000, encode it using all the previously described techniques.
2. For the following sequence: 1000010000110000, represent the bit sequence resulting after AMI encoding.
3. For the following bit sequence: 1001010110100100, represent the bit sequence resulting after MLT encoding.
4. For the following bit sequence: 0000110000100001, represent the bit sequence resulting after 4B/5B NRZI encoding.

BIBLIOGRAPHY

[1] Adina Aştilean. *Data transmission*, course notes

[2] Ioan, Aleodor Daniel. *Transmisia datelor: consideraţii teoretice şi experimente de laborator* [Data transmission: theoretical considerations and laboratory experiments].