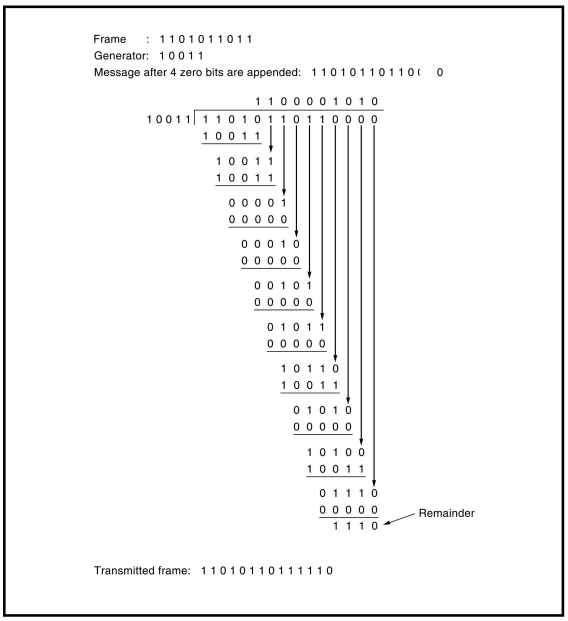
Digital communications - Error detection with CRC

Vasileios Papastergios (ID: 3651)

May 23, 2021



Aristotle University of Thessaloniki
School of Informatics

SCHOOL OF INFORMATICS

Course: Digital Communications
Semester: 4th
Instructors: Georgios Papadimitriou
Petros Nicopolitidis

Contents

1 Introduction and Overview

The present document serves as technical report and code documentation for the programming assignment in the course of *"Digital Communications"*. The author attended the course during their 4^{th} semester of undergraduate studies at the **School of Informatics AUTh**. The document is handed in complement with the source code files developed as programming solution to the assignment. The code files were genuinely developed by the author in *Java* programming language.

To start with, the assignment focuses on the **Cyclic Redundancy Check (CRC)**, an error-detecting code that is widely used in digital networks. Within the context of this assignment, we are going to adopt a code-driven approach. In specific, we are going to implement the CRC algorithm in Java programming language, in the first place. Afterwards, based on that implementation, we will attempt to execute the algorithm, simulating transmissions of randomly generated messages between an imaginary sending end and an imaginary receiving one.

Ultimate goal of such a procedure is to gather quantitative data related to the execution, as well as the efficiency of the CRC error-detection algorithm. Having gathered it, we can, then, analyze it in an experiment-oriented model and draw interesting conclusions. Before stepping into the CRC algorithm, however, it is deemed important to state that, throughout this logical and programming journey, special emphasis is given on clarifying the reasoning, as well as the developed source code. Mathematical relations and code snippets are introduced whenever necessary, in a try to clearly explain the author's approach.

2 Error Detection and CRC

Nowadays, an important part of how our communities are formed and operating is thanks to the rapid progress that has been made in the fields of *computer science* and *telecommunication*. It is common knowledge that, in almost every aspect of human life and activity, information is ceaselessly transmitted, received and/or displayed around us. In most of the cases, in fact, this procedure even slips our perception. Not getting perceived, however, does not mean that such a process is not important. On the contrary, it constitutes the fundamental building block of today's digital networks, the importance of whose in modern world is indisputable.

However, even after so many evolution steps in telecommunications, we still cannot boast of a way to perform a *perfect transmission*. The world we live in is not ideal, and so the communication channels are, as well. As a matter of fact, many communication channels are subject to *channel noise*. As a result, the information received at one end of the channel does not always have the intended form.

In a try to counteract such an ascertainment, techniques were developed, in order to detect errors that occur during data transmission. In simple words, these techniques get usually the form of special checks, that are executed on the delivered information at the receiving end of the transmission channel. Ultimate goal of these checks is to determine whether the information reaching the end has undergone transmission errors or not.

Within the context of this assignment, we are going to focus on a specific error-detection technique, known as **Cyclic Redundancy Check** or simply **CRC**, which is applied on digital data. Therefore, from now on, when we refer to *message*, we mean a sequence of bits. Every element in the sequence can get one out of two distinct values: digital $'0'$ or digital $'1'$.

Technically speaking, in the CRC way of detecting errors, some redundancy is added (appended) to the delivered message *before transmission*. The redundancy is officially called **Frame Check Sequence** or simply **FCS**. Based on this redundancy, the receiving end can check the consistency of the delivered message, defining whether there are errors in it or not. In order to calculate the appended redundancy, CRC utilizes a polynomial code, the so-called **generator polynomial**. This polynomial becomes the divisor in a *polynomial long division*, while the original message takes place as the dividend. The remainder of this special division is the FCS to be appended, whilst the quotient is ignored.

Having followed the above algorithmic steps, the message has now a greater size, since the FCS has been appended at the end of the original one. This new, reformed bit sequence is the one that is going to be actually transmitted via the communication channel. During transmission, however, errors might occur, leading some bits in the sequence to toggle value (from $'0'$ to $'1'$ or vice versa). How is CRC supposed to detect them?

The answer is quite simple: the delivered message is divided again with the same polynomial generator, this time at the receiving end of the communication channel. In case the delivered message has not undergone transmission errors, the polynomial long division leaves no remainder. In such a case, no errors are detected by the CRC, so the message is considered to be consistent.

Otherwise, in case the remainder of the division is not equal to zero, we can obviously conclude that some error(s) have occurred during transmission. This time the CRC detects at least one error; enough information in order to discard the delivered message as inconsistent. In such a case, the receiving end has to ask for a re-transmission and the process repeats, until the message is transmitted correctly (remainder equal to zero).

3 Implementing a CRC-based system in Java

Having covered the basic theoretical background of the CRC algorithm, transferring it into code comes naturally as the next step of our reasoning. In this section, we will try to present the key parts of a system that implements the CRC algorithm and utilizes it for a realistic application. The handed source code files that are related to this section are `BitSequence.java`, `Message.java` and `TransmissionSimulator.java`. The latter contains the `main` method, as well, in case running the system is desired. Last but not least, the whole project, as well as the statistic reports we are going to analyze in the next section can be found online, on **GitHub**.

3.1 Basic Analysis

According to the wording of the assignment, we aim for the construction of a system that simulates message transmissions between two imaginary points, a sending and a receiving one. In particular, a sequence of bits is generated in a random manner by the sender of the system. That bit sequence plays the role of the initial message that needs to be transmitted. Afterwards, the message undergoes CRC and the appropriate redundancy (FCS) is appended to it. The reformed message is then transmitted over an imaginary communication channel, subject to a specific bit error rate, that is known from beforehand.

Via this channel, the - possibly distorted - message reaches the receiving end, where CRC takes over to detect possible errors that have occurred during transmission. The system decides whether a re-transmission is needed or the message has been successfully delivered. At the receiving end, the system keeps records related to the delivered message and its consistency, in order to statistically analyze them later. The following steps in natural language sum up the workflow of a system's typical execution:

```
1. Generation of the random message (at the sending end)
2. Frame Check Sequence appendix (at the sending end)
3. Message transmission (via a noisy channel)
4. Error checking with CRC (at the receiving end)
5. Execution data records (at the receiving end)
```

Paying tribute to the elegant, object-oriented attributes of the Java programming language, the whole system was approached, designed and implemented using object-oriented reasoning. In particular, several classes were developed, one for every logical entity of the system. In the sub-sections that follow we are going to get a deeper insight into the structure and implementation of these entities, following a *bottom-up* presentation.

3.2 The `BitSequence` class

The fundamental logical and programming entity of our system is the `BitSequence` class. Exactly as its name suggests, the class represents a sequence of bits. When placed the one next to the other, these bits form a data representation that can be delivered over a communication channel. The class features no complexity, concerning the programming reasoning behind its implementation. The sequence is represented as

a `StringBuilder` instance and offers a variety of methods. The following UML diagram depicts the way the class is structured.

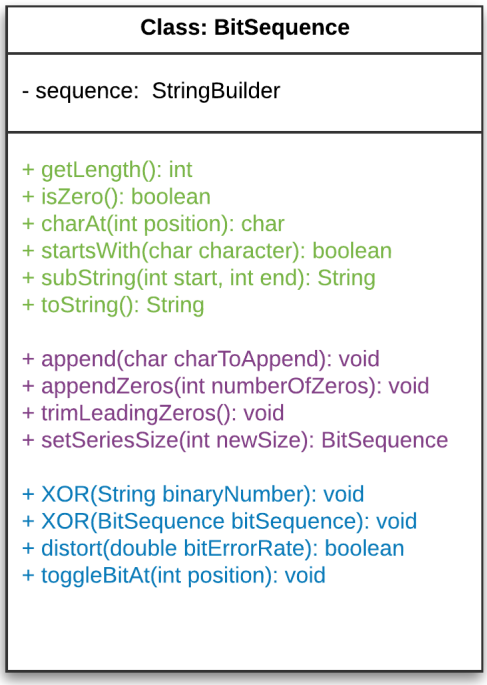


Figure 1: UML Diagram for the `BitSequence` class

The methods in the diagram are separated into three different color groups, based on their functionality. In specific, the class features the following three method groups:

- **Utility methods that do not alter the instance:** these methods execute auxiliary operations on a `BitSequence` instance. Their functionality is pretty simple and is fully described by their name and return type. They are not closely related to the CRC algorithm, but they do contribute to encapsulation and enhanced code readability. They do **not** alter the instance they are applied on.
- **Utility methods that do alter the instance:** these methods offer auxiliary functionality, as well. They are closely related with the size and the content of the bit sequence and **do alter** the current instance. Nevertheless, their functionality is commonplace, in full accordance with their name.
- **CRC-related methods:** these methods are directly utilized by the CRC algorithm or the simulating system in total. That is the reason why they are worth-analyzing. The subsection that follows is devoted to them.

3.2.1 Supporting `XOR` operation

The CRC-related methods inside the `BitSequence` class contain two overloaded versions of the `XOR` operation. In specific, both methods receive an external bit series as argument, calculate the `XOR` operation between the given argument and the current sequence, and store the result in the current sequence. The only difference between the two methods is *the type* of the externally passed argument. The following snippet hosts the source code implementation of the two overloaded `XOR` versions.

```java
public void XOR(String binaryNumber) {
    int thisIndex = sequence.length() - 1;
    int otherIndex = binaryNumber.length() - 1;

    /*
     * '1' XOR '1' => '0'
     * '1' XOR '0' => '1'
     * '0' XOR '1' => '1'
     * '0' XOR '0' => '0'
     * So, if left == right => '0', else '1'
     */
    while (thisIndex >= 0 && otherIndex >= 0) {
        if (this.sequence.charAt(thisIndex) == binaryNumber.charAt(otherIndex)) {
            this.sequence.setCharAt(thisIndex, '0');
        } else {
            this.sequence.setCharAt(thisIndex, '1');
        }
        thisIndex--;
        otherIndex--;
    }
}

public void XOR(BitSequence bitSequence) {
    XOR(bitSequence.toString());
}
```

Listing 1: Java code for the `XOR` operation between two bit sequences

The `XOR` operation between two bit sequences (binary numbers) is directly used in the CRC algorithm. In specific, the *polynomial long division* that the algorithm executes requires one `XOR` operation between the current remainder and the generator polynomial at every division step.

3.2.2 Simulating transmission over a noisy channel

Apart from the `XOR` operation, the `BitSequence` class provides two really interesting methods that play a crucial role in simulating a communication channel that is subject to noise. In fact, these methods alter some part of the bit sequence, under a specified probability. After all, that is exactly the way a noisy channel can affect the delivered message.

Getting a deeper insight into the code, we can easily see that the bits in sequence are serially accessed. Each one of the bits is potentially toggled, under a specific probability. This probability is nothing else than the bit error rate of the communication channel, which is passed as an external argument.

```java
public boolean distort(double bitErrorRate) {
    boolean isSeriesDistorted = false;

    for (int bitPosition = 0; bitPosition < sequence.length(); bitPosition++) {
        if (RandomEngine.getInstance().nextFloat() < bitErrorRate) {
            toggleBitAt(bitPosition);
            isSeriesDistorted = true;
        }
    }
    return isSeriesDistorted;
}

public void toggleBitAt(int position) {
    if (sequence.charAt(position) == '1') {
        sequence.setCharAt(position, '0');
        return;
    }
    sequence.setCharAt(position, '1');
}
```

Listing 2: Java code for simulating transmission over a noisy channel

Both methods feature no complexity, concerning their implementation. An interesting note, however, that has to be made is the fact that the `distort` method returns a `boolean` value, instead of `void`. The reason behind such an approach lies in the desired functionality of our CRC-based system. More specifically, we would like the system to keep track of the form and consistency that the delivered message reaches the receiver in. In simple words, we would like to know whether the delivered message has been distorted or not, irrelevant of the error-detecting CRC process that is, anyway, executed by the receiver.

Keeping track of this kind of information, lets the system have *global knowledge* about the delivered message. It is obvious, though, that such knowledge cannot be available in realistic systems, where actual messages are transmitted over actual communication channels in real time. Nonetheless, within the context of the current assignment, we make use of such global information only for purposes of statistical analysis on the CRC algorithm execution. The global knowledge is not interfering in any way in the CRC algorithm execution. Only after the CRC check is executed by the receiver, is the information revealed, in order to compare between the presence of an actual and a detected error, respectively.

3.3 The `Message` class

Having analyzed the bit sequence representation, the next entity we encounter in this bottom-up presentation is the `Message` class. As its name indicates, the class represents a digital message that can be transmitted from a sender to a receiver, via a noisy communication channel. The class is developed using the *Composite design pattern*. In particular, every `Message` object composes of a `BitSequence` one, forging, this way, a **"has-a"** relationship between the two classes.

Except for this, a `Message` instance has an auxiliary boolean attribute that stores whether the message has been distorted or not. Such an approach is in full accordance with what we have already covered in detail, about the *global knowledge* the system is provided with, back in section 3.2.2. The following UML diagram depicts the way the class is structured.

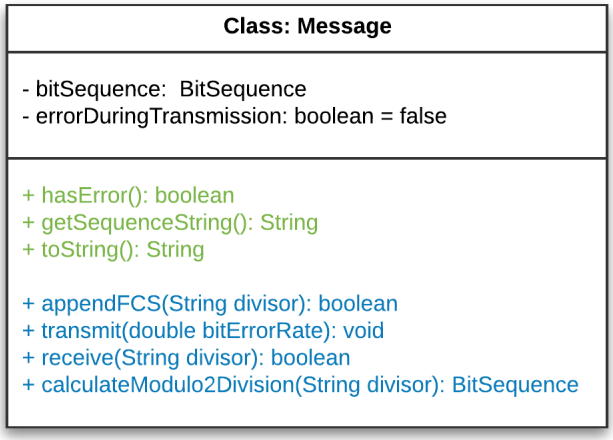


Figure 2: UML Diagram for the `Message` class

The class' methods are again divided into groups, according to their functionality and contribution to the CRC algorithm. For uniformity purposes, we are going to keep the same color conventions as we did back in the `BitSequence` class. More specific, the color groups are the following:

- **Utility methods that do not alter the instance:** these methods are quite simple and offer just an auxiliary functionality. They are either accessors of the class' attributes or `String` representations of them.
- **CRC-related methods:** these methods are directly utilized by the CRC algorithm, as well as the simulator, in total. They play a decisive role in the system's execution. The subsections that follow are devoted to them.

3.3.1 Preparing a `Message` for transmission

Preparing a message for transmission is one of the the core actions the system is in charge of performing. In fact, this preparation gets the form of a CRC algorithm execution, in order to calculate the Frame Check Sequence (FCS) that needs to be appended to the original message. Taking it a step further, the system, actually, calculates the *polynomial long division* between the original message (dividend) and the polynomial generator (divisor). The remainder of the division is the wanted FCS, while the quotient is discarded.

In order to successfully complete this task, the system utilizes several methods that are provided inside the `Message` class. The first one of them is the `calculateModulo2Division` method, which implements the polynomial long (modulo-2) division. Its implementation is presented in the following code listing.

```
public BitSequence calculateModulo2Division(String divisor) {

    /* represents the index to the sequence bit that is next to be used during division */
    int seriesIndex = divisor.length();

    /* represents the remainder of the division, its final value is formed dynamically (
    inside loop) */
    BitSequence remainder = new BitSequence(bitSequence.subString(0, seriesIndex));

    while (seriesIndex < bitSequence.getLength()) {

        /* if the currently checked word has leading zeros, they are trimmed
         * and the missing bits are appended at the end, taken from the actual sequence  */
        if (remainder.startsWith('0')) {
            remainder.trimLeadingZeros();
            int numberOfMissingBits = divisor.length() - remainder.getLength();
            for (int missingBit = 0; missingBit < numberOfMissingBits; missingBit++) {
                remainder.append(bitSequence.charAt(seriesIndex++));
                if (seriesIndex >= bitSequence.getLength()) return remainder;
            }
        }

        /* the remainder is updated as: remainder = remainder XOR P */
        remainder.XOR(divisor);
    }

    /* at the end of the while-loop, we have found the actual remainder of the division */
    return remainder;
}
```

Listing 3: Java code for the polynomial long division operation (modulo-2 division)

As we have already analyzed, the above method determines the Frame Check Sequence that needs to be appended to the original message, before it gets transmitted. The action of FCS appendix is performed in `appendFCS` method, which is shown below:

```
public boolean appendFCS(String divisor) {
    if (!checkCRCDivisorValidity(divisor)) return false;

    int numberOfFcsBits = divisor.length() - 1;

    /* simulates n shifts to the left, by appending n 0 at the end of the sequence */
    bitSequence.appendZeros(numberOfFcsBits);

    /* initializes the FCS as the remainder of the modulo-2 division between M and P */
    BitSequence frameCheckSequence = calculateModulo2Division(divisor);

    /* appends the FCS at the end of the original message */
    bitSequence.XOR(frameCheckSequence.setSeriesSize(numberOfFcsBits));
    return true;
}
```

Listing 4: Java code to append FCS to an original message

The method takes a polynomial generator (divisor) as argument and calls the `modulo2Division` with the appropriate parameters: the current sequence takes place as the dividend, while the given polynomial generator plays the role of the divisor. The result (remainder) of the division is appended at the end of the original message. After the execution of the method, the message is ready to get transmitted over the noisy channel.

3.3.2 Transmitting a `Message`

We already know a way to simulate the noisy channel: we have provided a `distort` method inside the `BitSequence` instance of the `Message` class. In addition, we have already analyzed the special design of that method: it returns a boolean value that indicates whether the bit sequence has been **actually** distorted or not. Taking these into account, this is the ideal point to introduce the complete process of transmitting a `Message`. Not surprisingly, this process utilizes all previously presented "tools", in a concise, yet really powerful, method:

```
1 public void transmit(double bitErrorRate) {
2     errorDuringTransmission = bitSequence.distort(bitErrorRate);
3 }
```

Listing 5: Java code for transmitting a `Message` via a noisy channel

Only a couple lines of code, but fair enough, in order to simulate a message transmission. The `distort` method is called to alter the current bit sequence under a specified probability; the channel's bit error rate. The returned value is assigned to the boolean attribute of the `Message` class, waiting to be used later, at statistical analysis time. After the execution of the method, the message is ready to get delivered. From now on, it is receiver's time to act.

3.3.3 Receiving a `Message`

Up to that moment, the final message (original plus the appended FCS) has been delivered to the receiver. The latter is in charge of checking the delivered message, concerning the potential presence of errors in it, due to the noisy channel. We could describe the process that takes place at the receiver as a "reverse" CRC algorithm execution.

In particular, the delivered message undergoes one more polynomial long division. Exactly like the previous time, the delivered message gets place as the dividend, while the generator polynomial as the divisor. However, this time the remainder is used to detect possible errors, rather than append a redundancy to the message. The logic is pretty simple and straightforward: if the remainder is zero, then the message has been delivered without errors. In any other case (remainder not equal to zero), the receiver detects at least one error. In such a case, the message has not been delivered successfully and a re-transmission is needed. The following listing presents the way this consistency check is implemented in our system.

```
1 public boolean receive(String divisor) {
2     return calculateModulo2Division(divisor).isZero();
3 }
```

Listing 6: Java code for receiving a `Message`

An obvious note that can be made on this method is the fact that the returned `boolean` value indicates whether the transmission was successful or not. In detail, if the returned value is:

- `true`, then transmission is considered *successful* and the message can be used by the receiver for whatever reason it was meant to
- `false`, then transmission is considered *unsuccessful*, due to the presence of error(s) in the delivered message, and re-transmission is needed.

3.4 The `TransmissionSimulator` class

Up to that point we have covered in detail all the distinct entities of our CRC-based system. Observing these entities under an abstract perspective, we could claim that the `BitSequence` class provides primitive functionality for representing a sequence of bits. The latter is wrapped by the `Message` class, extending its functionality and, at the same time, offering concise methods for preparing, transmitting and receiving messages. Combining them all together, in a completely functioning system, comes easily as the next step of our reasoning. The `TransmissionSimulator` class serves exactly that purpose.

More specifically, the class stands for the system's top-level representation. In actual fact, it contains the driving method, to execute random transmissions between two imaginary points, a sending and a receiving one. The method that is in charge of doing so, is called `executeASingleRandomTransmission` and features a high level of abstraction. The following listing hosts its implementation.

```
public class TransmissionSimulator {

    private static final int numberOfMessageBits = 10;
    private static final String CRCDivisor = "110101";
    private static final double bitErrorRate = 1e-3;

    private static void runASingleRandomTransmission() {

        /* constructs a random original message */
        Message message = new Message(numberOfMessageBits);
        System.out.println("Message to transmit: " + message.getSequenceString());

        /* simulates the CRC execution and FCS appendix */
        message.appendFCS(CRCDivisor);
        System.out.println("After FCS appendix : " + message.getSequenceString());

        /* simulates the transmission via a noisy channel */
        message.transmit(bitErrorRate);
        System.out.println("After transmission : " + message.toString());

        /* simulates the CRC check at the receiver end */
        System.out.println((message.receive(CRCDivisor)) ?
                "No error detected. Transmission was successful." :
                "Error(s) Detected. Re-transmission is needed.");
    }
}
```

Listing 7: Java code for the polynomial long division operation (modulo-2 division)

In order to run transmissions, we have to define several necessary values from beforehand. In particular, these values are:

- the number of bits in the original message
- the polynomial generator (divisor) to use in CRC
- the bit error rate of the noisy communication channel

In our case, these values are represented as `static final` attributes of the class and are initialized to indicative values, based on the wording of the assignment. It is obvious, though, that these attributes could have been initialized to any valid value, without the need of any alteration to the actual system code.

Focusing back to the driving method again, a random original message is created through the `Message` constructor. The appropriate Frame Check Sequence is calculated and appended to it, and then transmitted over a noisy channel. The receiver checks the consistency of the delivered message and informs on whether a re-transmission is needed or not. Throughout this procedure, the method prints informative messages about the method's workflow to the console. Our system is, from now on, up and running, offering, at the same time, easy access to possibly desired customization, in terms of the three communication properties.

4 Bulk execution

In the previous sections we have thoroughly presented not only the distinct entities of our system, but also the CRC-based simulator as a whole. In the current section, we are going to make use of the system, executing a specific number of random transmissions and keeping records of the results, in a try to draw conclusion about the execution of the system.

4.1 The context of execution

Before letting the system simulate random transmissions, we have to summary the conventions and default values that were adopted throughout this process. In particular, we have to define the following three execution parameters:

- **The number of bits in the original message**: It defines the multitude of bits that are going to be randomly generated at every execution of the system. The number refers to the *initial* message, before the action of FCS appendix. In our approach, this number was set to **20 bits/initial message**.
- **The CRC polynomial generator**: It represents the divisor of the *polynomial long division* that is executed by the CRC algorithm. In terms of validity, the divisor has to start and end with $'1'$, else it is considered invalid. In addition, it defines the length of the appended redundancy (FCS), since $len(FCS) = len(divisor) - 1$. Within the context of our approach, the polynomial generator was set to **"110101"**.
- **The bit error rate of the communication channel**: It represents the probability that a bit in the delivered sequence is toggled, due to channel noise. Within the context of our approach, this rate was set to **1e-3**; thus 1‰ of the bits that travel over the the communication channel are finally affected by noise.

4.2 Execution statistics

Having defined the context of execution, we are now ready to enforce our experiment-oriented approach. More specifically, the system was executed a specified number of times and the results were measured and statistically analyzed. In a try to gather a reliable sample of data to analyze, the system was set to simulated **100 million random transmissions**. All transmissions were ruled by the previously defined execution parameters. The gathered information can be summarized in the following three metrics:

- the number of messages that reach the receiving end being **actually distorted**
- the number of messages that are **detected** by the receiver as distorted
- the number of messages that are actually distorted but **not detected** by the receiver

The following table presents the execution results. For each one of the three metrics, both the actual number and the percentage rate are noted.

	absolute frequency	percentage rate
actual errors	2,469,383	$\approx$ 2.469%
detected errors	2,468,504	$\approx$ 2.468%
undetected errors	879	879e-8%
Total random transmissions: 100,000,000		

Table 1: Absolute frequency and percentage rate of the three execution metrics

According to the findings, about 2.5% of the transmitted messages are **actually** affected bi the communication channel noise. The respective number of **detected** errors is $\approx$ 2.4%; slightly lower than the previous

one. A quick conclusion could be that the CRC algorithm successfully detects the vast majority of the occurring errors. However, it seems that it misses a tiny percentage of them.

In a try to shed light on the execution findings, we are going to utilize several evaluation metrics. For this purpose, we introduce two data classes: a *positive* and a *negative* one. The reasoning behind them is quite simple. A delivered message is classified as **positive**, in case there are communication error(s) in it. Otherwise (no errors present), the delivered message is classified as **negative**. Combining this classification with the frequency noted in Table 1, we can construct the following *contingency matrix*:

		Detected class	
		Positive	**Negative**
Actual class	**Positive**	2,468,504 (True Positive)	879 (False Negative)
	Negative	0 (False Positive)	97,530,617 (True Negative)

Table 2: Contingency matrix for delivered messages' classification

Utilizing this values, we can calculate several **evaluation metrics**. Before presenting the metrics, we have to note that for each one of the values, an *abbreviation* is adopted, deriving from the first letters of the words in their description. For example, "**T**rue **P**ositive" will be noted from now on as "**TP**", "**F**alse **N**egative" as "**FN**" etch. The calculated evaluation metrics are the following:

- **True Positive Rate – TPR**: It is also known as **Hit Rate** or **Sensitivity** and, within our context, represents the rate of actually distorted messages that are detected as inconsistent by the system.

$$TPR = \frac{TP}{TP + FN} = \frac{2,468,504}{2,468,504 + 879} \approx 0.999644$$

  This metric verifies the intuitive conclusion that CRC successfully deals with the vast majority of the inconsistent messages. Transforming it to a percentage rate indicates that more than 99.9% of the erroneous messages are detected as such.

- **False Positive Rate – (FPR)**: It is also known as **False Alarm Rate** and, in our case, represents the rate of messages that are detected with errors, whilst they do not have any of them.

$$FPR = \frac{FP}{FP + TN} = 0$$

  This metric discloses an important property of the CRC algorithm: no messages can be (falsely) detected as inconsistent, when they have no distortion errors in them. Alternatively, we can claim that the CRC "alarm" is always fair (when a message is detected with errors, it surely has at least one of them).

- **Accuracy – (ACC)**: It represents the rate of messages that are correctly detected (including both positive and negative delivered messages).

$$ACC = \frac{TP + TN}{sample_size} = \frac{2,468,504 + 97,530,617}{100,000,000} = \frac{99,999,121}{100,000,000} = 0.99999121$$

  The figures speak for themselves. As a matter of fact, less than 0.000001% of the delivered messages were unsuccessfully dealt by the CRC algorithm, while all the rest was successfully detected.

5 Conclusion

Up to that point, the current report comes to its semantic completion. We started with covering the theoretical background of the error-detection need in modern telecommunication and digital networks. We especially focused on a specific technique, called *Cyclic Redundancy Check* or *CRC*.

Next, we tried to utilize the theoretical information as the base for a system that simulates random transmissions between two imaginary points. Throughout this process, we placed special emphasis on the system's *implementation in Java* programming language. In particular, we thoroughly presented the key parts and algorithms of the system, such as the way a polynomial long division is executed and the object-oriented approach of delivering messages between a sender and a receiver.

Last but not least, we *tested* our system, by letting it simulate an importantly large number of random transmissions, under a strictly specified execution context. The latter formed an ideal environment, in order for us to statistically analyze the results. This analysis indicated in an experiment-driven matter that CRC features high rates in *Sensitivity* and *Accuracy* and, at the same time, zero *"false alarm"* percentage.

References

[1] William Stallings. *Data and Computer Communications, Eighth Edition*.

[2] Taub H., Schilling D. *Principles of Communication Systems, Second Edition*. McGraw-Hill Book Company, 1986.

[3] Georgios Papadimitriou. *Ψηφιακές Επικοινωνίες - Συμπληρωματικές σημειώσεις*. School of Informatics, AUTh, 2011.

[4] Georgios Papadimitriou, Petros Nicopolitidis. *Lecture Notes on the course of Digital Communications*. School of Informatics AUTh, 2021.