Ministry of Science and Higher Education of the Russian Federation

ITMO UNIVERSITY

GRADUATION THESIS

PERFORMANCE ANALYSIS OF BROADCAST ALGORITHMS IN SYSTEMS WITH BYZANTINE FAILURES

Author: Hakimov Muhammadjon ______________

Subject area: 01.03.02 Applied mathematics and informatics

Degree level: Bachelor

Thesis supervisor: Kuznetsov P., PhD, professor ______________

Saint Petersburg, 2022

Student Hakimov Muhammadjon

Group M34361 Faculty of IT&P

Subject area, program/major

Mathematical models and algorithms in software engineering

Consultant(s):

a) Tonkikh A., PhD student ______________

Thesis received "____" ___________ 20__

Originality of thesis ____%

Thesis completed with grade ____

Date of defense "15" June 2022

Secretary of State Exam Commission Stumpf S. ______________

Number of pages ____

Number of supplementary materials/Blueprints _____________________________

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ АЛГОРИТМОВ РАССЫЛКИ В СИСТЕМАХ С ВИЗАНТИЙСКИМИ ОШИБКАМИ

Автор: Хакимов Мухаммаджон Рахматджонович ______________

Направление подготовки: 01.03.02 Прикладная математика и информатика

Квалификация: Бакалавр

Руководитель ВКР: Кузнецов П.В., PhD, профессор ______________

Санкт-Петербург, 2022 г.

Обучающийся Хакимов Мухаммаджон Рахматджонович

Группа M34361 Факультет ИТиП

Направленность (профиль), специализация

Математические модели и алгоритмы в разработке программного обеспечения

Консультанты:

a) Тонких А.А., студент PhD ______________

ВКР принята «____» ___________ 20__ г.

Оригинальность ВКР ____%

ВКР выполнена с оценкой ____

Дата защиты «15» June 2022 г.

Секретарь ГЭК Штумпф С.А. ______________

Листов хранения ____

Демонстрационных материалов/Чертежей хранения _____________________

CONTENTS

INTRODUCTION

Reliable broadcast is a very popular primitive in distributed systems, and, in general, broadcasts are used a lot — in notification systems, for content delivery (in content delivery networks, *CDN* [14] — geographically distributed group of servers), for replication (in replicated state machines, *RSM* [13] — a generalized approach for building fault-tolerant distributed systems, and *blockchains*, in particular), and in any other group communication of processes in distributed systems. Some striking examples where reliable broadcast is widely used are *pBFT* [2], *HyperLedger* [11], *HotStuff* [5], etc.

If we talk about blockchain, for example, then this is also a replicated state machine. It is important to mention that all such abstractions do not store data in one place and try to avoid the single point of malfunctioning, and the most basic mechanism that is used for data dissemination in these types of systems is broadcast.

In systems where there are no *Byzantine failures*, or in other words, there are no processes that behave arbitrarily, the broadcast algorithms look quite simple and are not very interesting to explore. So, in this work, we focus on systems with Byzantine failures.

Since broadcast is a basic primitive in this kind of system, it would be interesting for us to implement these primitives and implement them efficiently. Generally, there are two ways to evaluate efficiency of such algorithms.

The first method is theoretical complexity analysis, which typically boils down to asymptotic evaluation of ”worst-case” space and time complexity.

In our context, time complexity determines how long it takes to deliver the message to all receivers, and space complexity — how many bytes of data it took to transfer over the network to achieve delivery.

At the first sight, it may seem that communication (space) complexity should be $n \cdot m$, where n is the total number of receivers and m is the length of the message, but this is not the case in fault-prone system, especially when Byzantine failures may occur.

For instance, the sender can be Byzantine, which means that it can *equivocate* — send different messages to different processes.

To detect equivocation, some sort of validation is necessary.

And here *quorums* come. Usually, a quorum means a subset of the processes of the system. Quorums are organized in such a way that any two quorums intersect in at least one process. In the case where there are Byzantine failures, any two quorums must intersect in at least one *correct* process. Quorums are used as validating sets, meaning that they will guarantee that two correct processes will not deliver two different messages from the same source.

The second method is practical complexity analysis. For comparison, theoretical complexity bounds are usually defined on the ”worst-case” scenarios, which is good in the sense that it shows better than what the algorithm cannot physically do but does show nothing about effective complexity, and this is what exactly the second method does.

Broadcast is not used in the system by itself, it is used in some applications, the application calls broadcast depending on the needs of the application's customers and the load is difficult to predict. The broadcast algorithms under consideration themselves are very efficient, in the sense that they do not require anything from the model, the maximum that we guarantee is that we have a reliable channel, which means that if some correct process sends a message to another correct process, it will deliver it sooner or later.

On the one hand, it is good, the algorithm works reliably for all variants of execution, on the other hand, we mean a rather weak model, and the weaker the model, the more variants of execution, which also means more freedom for intentional or malicious attacks.

In the average case analysis, we take an algorithm and run it multiple times for a very long time, and only then we calculate the average latency value using simple math, and it is obvious that this approach has much more advantages when evaluating the values we need. So, from the point of view of the average case analysis, it is difficult to estimate the value of the average latency using the first method only.

Therefore, the standard approach when using the second method would be to conduct experiments. And this is exactly what is going to be done in this work.

CHAPTER 1. PROBLEM STATEMENT

1.1. Relevance

In this work, we will consider three broadcast algorithms and measure the average delivery time of a single message to all receivers for various parameters, for example, network latency, the throughput of communication channels, and various kinds of Byzantine failures.

We will consider the abstraction, which is called *Byzantine reliable broadcast*. Byzantine reliable broadcast is an interface and some algorithms implement it. The essence of the Byzantine reliable broadcast is that any algorithm that implements it guarantees the execution of the properties of this interface, namely *validity*, *no duplication*, *integrity*, *consistency*, and *totality*. The meaning and the importance of each of these properties will be revealed in section 1.2.

When it comes to Byzantine reliable broadcast, it is important to mention at least three parameters of the algorithms that implement this interface — *fault resilience*, or in other words, the number of allowed Byzantine processes, *communication steps* — the number of message delays required to complete one iteration of the algorithm, and the number of *messages* transmitted over the network. Another important parameter is the usage of digital signatures, which can negatively affect the delivery time.

Table 1 – Comparison of the parameters of the algorithms under consideration

Algorithm	**n**	**Delays**	**Signatures**
Bracha	$3f + 1$	3	no
Imbs & Raynal	$5f + 1$	2	no
Abraham & others	$3f + 1$	2	yes

All the algorithms considered in this work send about $O(n^2)$ messages during one iteration, but they differ in other parameters. As we can see in Table 1, *Bracha's algorithm* [1] uses 3 message delays, while the other two algorithms use only 2. The second algorithm described by Damien Imbs and Michel Raynal in the *article* [6] needs more processes in a system with when the value of the number of Byzantine processes is fixed, that is, if for the first and third algorithms, with the presence of one Byzantine process, a total of 4 processes is sufficient, then for the second algorithm we will need 6 processes in a system in total. *The third algorithm* [4] by

Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang uses digital signatures, which may affect performance in a negative sense, while the other two do not use them. We will call algorithm *signature-free* when it does not use digital signatures.

As we can see, each of the three selected algorithms has its own bottleneck, and our task is to find application scenarios for each of these algorithms by changing the different system parameters and selecting the most suitable parameters for each of the algorithms. And the parameters will set the environment, that is, in this way we will find the environments for each of the algorithms where their usage would be the most optimal.

1.2. Definitions

In this section, we will introduce the basic definitions that we will need to describe the subject area and the work done. We will begin the introduction of concepts with a distributed system and end with the definition of the interface for the Byzantine reliable broadcast.

1.2.1. Distributed system

Definition 1. As a *distributed system*, we will consider a system with n processes $p_1, \ldots, p_n$ isolated from each other, while being able to send messages to any other process in the system, including themselves.

1.2.2. Byzantine failure

Definition 2. The failure in the system when one or a set of processes begins to behave arbitrarily is called *Byzantine*. This is the most general definition of failure in distributed systems because it implies absolutely any incorrect behavior. When we use it, we do not make any assumptions about the behavior of incorrect processes that are allowed any kind of behavior and therefore sending any messages. The reason for such failure can be both errors in the code and malicious attacks. It is pretty easy to understand that this kind of errors are the most difficult to tolerate.

1.2.3. Byzantine process

Definition 3. *Byzantine* will be called a process in the system with Byzantine failure, which behaves as it likes to, that is, it is a process whose behavior is absolutely random. For example, it can be a process that does not respond to any request, or a process that behaves like a correct process or a process that constantly tries to break the integrity of the system, for example, by sending different messages to different processes, if the process itself is the source of the broadcast.

1.2.4. Byzantine quorum

The majority of processes, i.e., any set of more than $\frac{n}{2}$ processes, is called a *quorum* in a system with n systems. The main condition is that any two quorums must overlap in at least one process.

Definition 4. Everything is not always good in the system, and Byzantine failures often occur. In such cases, it may happen that the two quorums do not overlap in any *correct* process. And such quorums are pretty useless. Therefore, a *Byzantine quorum* is a quorum, which guarantees that any two Byzantine quorums overlap in at least one correct process. If we need to tolerate f faults in a system with n processes, we will need a set of more than $\frac{n+f}{2}$ processes.

At least $\frac{n+f}{2} - f = \frac{n-f}{2}$ correct processes make up every Byzantine quorum. And if we look at two disjoint Byzantine quorums, there are more than $\frac{n-f}{2} + \frac{n-f}{2} = n - f$ correct members in them, and since there are only $n - f$ correct processes at all, at least one correct process must occur in both Byzantine quorums.

After receiving a message from the Byzantine quorum, algorithms that rely on Byzantine quorums must frequently move forward. Because up to f faulty processes may fail to respond, the system must have at least a Byzantine quorum of correct processes that together will be able to come to some kind of decision on the subsequent progress. This criterion is only satisfied if

$$n - f > \frac{n+f}{2},$$

or equivalently when $n > 3f$. As a result, algorithms that tolerate Byzantine failures normally only allow $f < \frac{n}{3}$ processes to fail.

1.2.5. Cryptographic abstractions

In this subsection, we will define all the cryptographic abstractions that will be used in this work, starting from MAC.

1.2.5.1. MAC

Definition 5. *MAC* [12] (message authentication code) is a cryptographic abstraction that allows to quickly sign messages and to verify the sender of the messages, ensuring that any process taken cannot pretend to be another process, which greatly facilitates the life of developers of distributed algorithms, especially algorithms resistant to Byzantine failures.

1.2.6. Link abstractions

We will go over link abstractions in this section. Some are more powerful than others in terms of providing additional guarantees on reliability, which is the main property of interest. They are all *point-to-point* connection abstractions, which means they allow communication between any two processes. Moreover, we assume that the process can also send messages to itself.

We will begin with a description of the fair-loss link abstraction, which encapsulates the essential idea that data can be lost while transmitting over the network. This is a fairly powerful primitive, although it looks quite simple. All other known higher-level primitives, such as stubborn link or perfect link, use fair-loss links under the hood. They use repetitive transmission techniques to hide the implementation details from the developer, and making the network thus reliable. In the end, we will be interested in perfect link abstraction, foremost, that is why we will provide its implementation using fair-loss link abstraction in this subsection later.

1.2.6.1. Fair-Loss link

Definition 6. The fair-loss link abstraction is the weakest of the link abstractions considered in this work. Its interface consists of two events: a request event for sending messages and an indication event for receiving messages.

Three characteristics distinguish fair-loss links. First, the *no creation* property assures that the network does not even corrupt or produce any messages. Second, the *fair-loss* property ensures that we do not have such a situation that not a single sent message reaches the recipient. As a result, if both the sender and recipient processes are correct, and the sender keeps repetitively transmitting a message, the message will be delivered to the recipient sooner or later. The final, the *finite duplication* property, assures that we cannot receive a message infinitely many times if the number of messages sent is finite. A more formal description of all three properties:

— *Fair-loss:* If a correct process p sends a message m to a correct process q an unlimited number of times, then q delivers m an infinite number of times.

— *Finite duplication:* If a correct process p delivers a message m to process q a finite number of times, m cannot be sent infinitely many times by q.
— *No creation:* If a message m with sender p is delivered by some process q, then m was earlier sent to q by process p.

1.2.6.2. Stubborn link

Definition 7. In this subsection, we define the abstraction of *stubborn links*. When using genuine fair-loss networks, this abstraction hides the lower-layer retransmission mechanisms used by the sender process to ensure that its messages are finally delivered by the destination process.

Some properties are the same for many abstractions. For example, the *no creation* property prohibits the connection from generating messages just like in fair-loss link abstraction. Every message transmitted over the network using the stubborn link is delivered to the receiver an unlimited number of times due to the *stubborn delivery* property. A more formal description of both properties:
— *Stubborn delivery:* If a correct process p sends a message m to a correct process q once, q will deliver m indefinitely.
— *No creation:* If a message m with sender p is delivered by some process q, then m was earlier sent to q by process p.

Performance. The main purpose of this algorithm is educational, foremost. This is because the way that it works is clearly inefficient in practice at all. The more messages we send via this link, the larger the size of the messages stored and forwarded over time. Moreover, there is no point in resending already delivered messages. What can be done to make the algorithm more practical? First, we note that if the algorithm using the link has ended, then we can also stop resending messages. It looks pretty obvious, but not everyone sticks to this strategy in practice. Second, it is important to keep in mind that the concepts of infinity and infinitely often depend on the context, in our case primarily on the algorithm that uses the link. Therefore, it is important to take into account all the specificity in practice.

1.2.6.3. Perfect link

Definition 8. When we use the stubborn link abstraction, the receiver process is the one who is responsible for determining whether a sent message was delivered earlier or not. We can make this primitive even stronger by adding mechanisms that would allow checking received messages for duplication, and ignore them if they are already delivered. This is what exactly the implementation of *perfect link* abstraction does. Link properties allow calling the abstraction reliable. Therefore, perfect links are also known as reliable links.

There are three characteristics that define perfect links. *No creation* is the property, which is the same as in the other link abstractions described before, so we would not focus on it.

Let's assume that the sender is a correct process. So, if the receiver process is also correct, the *no duplication* property combined with the *reliable delivery* property assures that every message sent by a correct process is delivered exactly once by the receiver. Formal description of all properties of perfect links:

— *Reliable delivery:* If a correct process p sends a correct process q a message m, q will eventually deliver m.

— *No duplication:* A process never delivers the same message twice.

— *No creation:* If a message m with sender p is delivered by some process q, then m was earlier sent to q by process p.

Performance. In addition to the problems we discussed for stubborn link, which are relevant for perfect link, we have a new problem. We have an infinitely growing set of delivered messages from recipients. At first glance, it seems that recipients can send acknowledgment messages, and the sender can send acknowledgements to received acknowledgment messages and so on. But there is no guarantee that such acknowledgements will first be delivered to the recipient, and secondly, it is not a fact that some such messages are not already on the way. If messages that are on the way are delivered again, our perfect link properties will be violated. But there are solutions that are based on the idea that the messages expire, and they track either the storage time of each of the messages, or use the idea of an LRU cache with invalidation [8].

1.2.6.4. Authenticated perfect link

Messages transmitted in networks with Byzantine process abstractions are the subject of this subsection. We should keep in mind that in this model, communication channels can also act arbitrarily. In theory, it may be that link will close the connection and simply stop all the work of a distributed system in this way, but identifying such cases is a separate big task and is not a point of our interest.

Definition 9. With the presence of Byzantine process abstractions, a fair-loss link abstraction, or a stubborn link abstraction alone is not particularly beneficial, because in all the links discussed earlier, tolerance to such errors is simply not provided. In such cases, first of all, the idea of using cryptographic abstractions usually comes to mind. And this is the right direction, because cryptographic authentication, for example, can transform previously discussed links into a more useful *authenticated perfect link* primitive. This primitive avoids message forgery when the message is sent between two correct processes.

The latter property means that if a message from another correct process was delivered to some correct process, then that other process actually sent the message delivered by the first process.

Authenticated perfect links have the same *reliable delivery* and *no duplication* properties as perfect links. Only the *authenticity* property of Byzantine processes outperforms the *no creation* property for crash-stop processes. Here are formal descriptions of all the properties:

— *Reliable delivery:* If a correct process p sends a correct process q a message m, q will eventually deliver m.
— *No duplication:* A process never delivers the same message twice.
— *Authenticity:* If some correct process q delivers a message m with sender p and process p is correct, then m was previously sent to q by p.

Performance. It's worth noting right away that the use of cryptographic abstractions affects the execution time, no matter where we use them. This is the first problem of an authenticated perfect link abstraction. Second, because the processes for constructing an authenticated perfect link are identical to those for constructing a perfect link, we have the same issue of the delivered messages set extending infinitely over time. Possible practical solutions to the second problem were discussed in the previous subsection. Well, with the first problem, we have nothing left to do except to choose the most effective implementations of cryptographic abstractions we need.

1.2.7. Broadcast abstractions

Byzantine processes can deviate arbitrarily from the instructions that an algorithm assigns to them, making it look as if they are attempting to prevent the algorithm from achieving its objectives. Such conduct must be tolerated by an algorithm.

When a Byzantine process is involved, for example, a malfunctioning sender may interfere with the low-level broadcast primitive, causing other processes to provide different messages, violating the reliable broadcast's agreement or no duplication property.

Most algorithms in the fail-arbitrary model that implement primitives rely on cryptographic techniques, at least to implement the authenticated perfect links abstraction that all of them employ. However, cryptography is rarely the only method for tolerating Byzantine processes. In many cases, such algorithms are naturally more complicated.

The fail-arbitrary model, like the other system models, distinguishes between faulty and correct processes. The separation is static in the sense that a process is considered faulty even if it is correct at one point when it participates in a distributed algorithm and then fails later. This distinction is also warranted in the fail-arbitrary system paradigm because one cannot make any statements about the behavior of a Byzantine process. Byzantine process abstraction can operate in any way it wants, and no mechanism can ensure anything about its actions.

1.2.7.1. Byzantine Consistent Broadcast

Definition 10. In the fail-arbitrary model, a *Byzantine consistent broadcast* primitive solves one of the most basic agreement problems. A specified sender process s broadcasts a message m to every instance of consistent broadcast. If the sender s is correct, then every correct process should deliver m at some point in the future. If s is faulty, the primitive ensures that every correct process, if it delivers a message at all, delivers the same message. In other words, with a malfunctioning sender, some correct processes may convey a message while others may not, but a message delivered by two correct processes is unique. Consistency is the name for this property.

It has been demonstrated that the number of faulty processes must satisfy $f < \frac{n}{3}$ in order to implement the Byzantine consistent broadcast abstraction (as well as most other abstractions in the fail-arbitrary model, see section 1.2.4 for more details).

Formal description of Byzantine consistent broadcast's properties:

— *Validity:* If a correct process p sends out a message m, then every correct process will eventually deliver m.

— *No duplication:* A process never delivers the same message twice.

— *Integrity:* If a correct process delivers a message m with sender p, and process p is correct, then m has already been broadcast by p.

— *Consistency:* When one correct process delivers a message m and another correct process delivers a message m', then $m = m'$.

1.2.7.2. Byzantine Reliable Broadcast

Definition 11. The problem with Byzantine consistent broadcast is that it does not guarantee that every message that has been broadcast will either be delivered by every correct process, or not. That is, there may be situations in it when a message that was broadcast will be delivered only to some subset of the correct processes.

In other words, the previous section's Byzantine consistent broadcast primitive does not ensure *agreement* in the sense that a correct process only delivers a message if and only if every other correct process also delivers a message. By expanding Byzantine consistent broadcast with a *totality* condition, the *Byzantine reliable broadcast* abstraction provided here adds this guarantee.

Formal description of Byzantine reliable broadcast's properties:

— *Validity:* If a correct process p sends out a message m, then every correct process will eventually deliver m.

— *No duplication:* A process never delivers the same message twice.

— *Integrity:* If a correct process delivers a message m with sender p, and process p is correct, then m has already been broadcast by p.

— *Consistency:* When one correct process delivers a message m and another correct process delivers a message m', then $m = m'$.

— *Totality:* If any correct process delivers a message m, then every correct process eventually delivers a message m.

The agreement property for a Byzantine broadcast primitive is obtained by combining the consistency property and the totality property into one. It has the same requirements as a (regular) *reliable broadcast abstraction's* [1] agreement property.

Unsurprisingly, a Byzantine reliable broadcast method requires more steps than a Byzantine consistent broadcast algorithm, and we will see the details in the next section when exploring different implementations of Byzantine reliable broadcast.

1.3. Problem formulation

In the previous section, we introduced the definition of the Byzantine reliable broadcast interface. Table 1 shows three algorithms, three implementations of Byzantine reliable broadcast, each of which has its own bottleneck, and it is really not very clear in which cases which of these algorithms is better to use in your distributed system. So, in this work, we will try to figure it out.

Also, because broadcast is one of the most important fundamental primitives in distributed systems, it would be fascinating to see how well, and how efficient, we might implement this primitive. In general, as we pointed out before, there are two methods for estimating the efficiency of such algorithms and their implementations. The first one is the theoretical complexity analysis, which gives ”worst-case” estimations for the number of transmitted messages over the network during the single iteration of an algorithm and the number of message delays, i.e., the number of steps required to a considering Byzantine reliable broadcast algorithm to commit and terminate. The second one is the average case analysis, which gives the real-world latency estimation, or at least estimations that are in fact very close to the real-world latency values.

So, the primary goal of this work would be to evaluate the performance of three Byzantine reliable broadcast algorithms, the properties of which are listed in Table 1. Experiments with different values of the system parameters will be used to collect data. Different volumes of transmitted messages, total number of processes in the system, number of Byzantine processes, and network delays, for example, will be considered.

After receiving the results of the experiments, we will analyze the data and draw conclusions on the cases where one of the algorithms performs better or worse than the others, and we will try to explain why and summarize the results. Because the conclusions of the experiments will be confirmed in practice, the collected results will help in the selection of the optimal algorithm for reliable broadcasting in systems with Byzantine failures.

Conclusions on Chapter 1

In this chapter, we have revealed in detail the relevance of the topic considered in this work, described the properties of the three algorithms under consideration and introduced all the basic concepts that we will refer to in the following chapters.

In the next chapter, we will take a closer look at each of the algorithms and prove their correctness, as well as review their ”worst-case” performance estimations from the corresponding papers.

CHAPTER 2. THEORETICAL RESULTS

In this chapter, we will have a look at the implementations of the algorithms from Table 1, prove their correctness and discuss their theoretical ”worst-case” performance. We will start with *Bracha's algorithm* [1], which was one of the earliest Byzantine reliable broadcast algorithms, then consider *algorithm proposed by Imbs and Raynal* [6] and *algorithm* with digital signatures *proposed by Abraham and others* [4].

2.1. Bracha's algorithm

This section introduces one of the first Byzantine reliable broadcast implementations — *Bracha's algorithm* [1], the broadcast primitive in the fail-arbitrary system paradigm. A single message is broadcast by an instance of the primitive. When the latter is limited to one message, it can be considered as the fail-arbitrary equivalent of the *reliable broadcast abstraction* [1] for crash-stop processes. $n > 3f$ is required to implement this primitive in a fail-arbitrary system model with n processes.

2.1.1. Overview

Let's now see how the algorithm actually works. The pseudocode of the algorithm is given in the Listing 1.

The sender sends a message to all processes first. After receiving the message from the sender, each process echoes it to all. To be more specific, the sender s that brb-broadcasts a message m disseminates m to all processes in the first round. In the second round, each process acts as a witness for the message m that it has received from the sender and resends it to all others in an *ECHO* message. When a process receives a Byzantine quorum of these echoes, it sends a *READY* message to all other processes, indicating its willingness to brb-deliver the message if enough other processes are willing as well. A process brb-delivers a message once it receives a total of $2f + 1$ such indications in *READY* messages.

Another mechanism used in the algorithm is that when a process receives just $f+1$ *READY* messages but has not yet sent a *READY* message to others, it sends one. This stage amplifies the *READY* messages and is essential for the totality property to work.

Listing 1 – Bracha's Algorithm, pseudocode

```
Implements:
    ByzantineReliableBroadcast, instance brb, with sender s.

Uses:
    AuthPerfectPointToPointLinks, instance al.

upon event { brb, Init } do
    sentEcho := false
    sentReady := false
    delivered := false
    echos := [EMPTY] ^ n
    readys := [EMPTY] ^ n

// only process s
upon event { brb, Broadcast | m } do
    forall q in Π do
        trigger { al, Send | q, [SEND, m] }

upon event { al, Deliver | p, [SEND, m] } such that p = s
  and sentEcho = false do
    sentEcho := true
    forall q in Π do
        trigger { al, Send | q, [ECHO, m] }

upon event { al, Deliver | p, [ECHO, m] } do
    if echos[p] = EMPTY then
        echos[p] := m

upon exists m != EMPTY such that
  #({p in Π |echos[p] = m}) > (n + f) / 2 and sentReady = false do
    sentReady := true
    forall q in Π do
        trigger { al, Send | q, [READY, m] }

upon event { al, Deliver | p, [READY, m] } do
    if readys[p] = EMPTY then
        readys[p] := m

upon exists m != EMPTY such that #({p in Π | readys[p] = m}) > f
  and sentReady = false do
    sentReady := true
    forall q in Π do
        trigger { al, Send | q, [READY, m] }

upon exists m = EMPTY such that #({p in Π | readys[p] = m}) > 2f
  and delivered = false do
    delivered := true
    trigger { brb, Deliver | s, m }
```

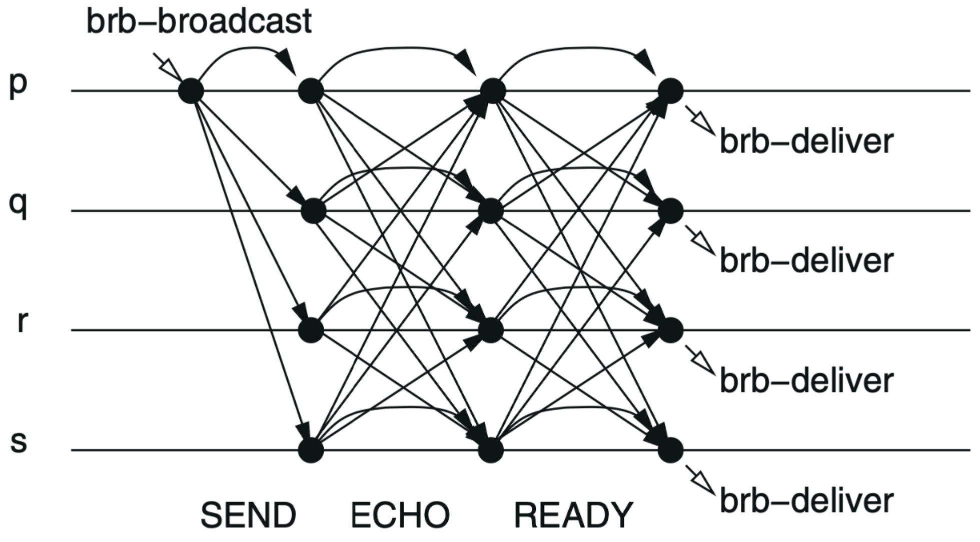

Figure 1 – Failure-free execution of Bracha’s algorithm, source: [1]

Figure 1 depicts a successful iteration of *Bracha’s algorithm* [1] with no failures with a correct sender p. The identical message is delivered by all processes.

Let’s consider the *Bracha’s algorithm* [1] one more time to see how the amplification step works. Assume that process p is the single correct process that al-sends a *READY* message containing m, but correct process r somehow brb-delivers m. Because r must have gotten three *READY* messages with m, at least, process p and one other correct process must have delivered a *READY* message with m as well. These two processes are correct, thus the third correct process al-delivers their *READY* messages. The third correct process then al-sends a *READY* message according to the amplification step. As a result, because three correct processes have delivered a *READY* message containing m, every correct process brb-delivers m.

2.1.2. Correctness

When $n > 3f$, *Bracha’s algorithm* [1] from Listing 1 implements a Byzantine reliable broadcast abstraction. If the sender is correct, every correct process al-sends an *ECHO* message and al-delivers at least $n - f$ of them, meaning that the *validity* property derives straight from the algorithm. Every correct process also brb-delivers the message m contained in the *ECHO* messages since $n - f > \frac{n+f}{2}$ under the condition that $n > 3f$.

The algorithm’s guarantees of *no duplication* and *integrity* are correct due to the properties and subsequent use of authenticated perfect links and the algorithm itself.

For the sake of argument, note that *consistency* property of Byzantine reliable broadcast means that if some of the correct processes al-send a *READY* message, they all do so with the same message containing m. It's impossible for the faulty processes to send out enough *READY* messages with content other than m.

Finally, as the example above has shown, the amplification step from $f + 1$ to $2f + 1$ *READY* messages ensures the *totality* property. If a correct process brb-delivers some m, at least $f + 1$ correct processes must have al-sent a *READY* message with m. Because these processes are correct, each one finally sends a *READY* message with m via the amplification step or after receiving enough *ECHO* signals. In either scenario, every correct process brb-delivers m at some point.

2.1.3. Performance

There are three communication steps in the algorithm, two of which are all-to-all message exchanges. For broadcasting one message, it needs $O(n^2)$ point-to-point messages in total. It is worth noting that the approach merely employs the authenticated connections concept, which may be implemented using a MAC (see section 1.2), rather than computationally expensive digital signatures.

So, summing up, the algorithm uses $O(n^2)$ messages for one iteration, the number of Byzantine processes in it cannot exceed $f < \frac{n}{3}$, the algorithm uses three message delays and does not use digital signatures.

2.2. Byzantine Reliable Broadcast Algorithm by D. Imbs and M. Raynal

This section introduces a simple and Byzantine reliable broadcast *algorithm* [6] for asynchronous message passing systems with n processes, among which up to $f < \frac{n}{5}$ can behave arbitrarily (Byzantine processes). Two communication steps and $n^2 - 1$ messages are required for the single iteration of this algorithm. The suggested algorithm exhibits an interesting tradeoff between communication efficiency and resilience when compared to *Bracha's algorithm* [1], which is resilience optimal ($f < \frac{n}{3}$) and takes three communication steps and $2n^2 - n - 1$ messages.

2.2.1. Overview

The algorithm from Listing 2 consists of a client side and a server side, which together implement the Byzantine reliable broadcast abstraction. When a (correct) process wants to broadcast an application message m, it simply broadcasts the algorithm message *INIT* on the client side.

Listing 2 – Asynchronous signature-free Byzantine-tolerant reliable broadcast algorithm by D. Imbs and M. Raynal, pseudocode

```
Implements:
    ByzantineReliableBroadcast, instance brb, with sender s.

Uses:
    AuthPerfectPointToPointLinks, instance al.

// only process s
upon event { brb, Broadcast | m } do
    forall q in Π do
        trigger { al, Send | q, [INIT, m] }

upon event { al, Deliver | p, [INIT, m] } such that p = s
  and firstInit = true and sentWitness = false do
    firstInit := false
    sentWitness := true
    forall q in Π do
        trigger { al, Send | q, [WITNESS, m] }

upon event { al, Deliver | p, [WITNESS, m] } do
   if witnesses[p] = EMPTY then
        witnesses[p] := m

   if m != EMPTY such that #({p in Π | witnesses[p] = m}) > n − 2f
      and sentWitness = false do
        sentWitness := true
        forall q in Π do
            trigger { al, Send | q, [WITNESS, m] }

   if m != EMPTY such that #({p in Π | witnesses[p] = m}) > n − f
      and delivered = false do
        delivered := true
        trigger { brb, Deliver | s, m }
```

A process on the server can receive two types of messages.

When the process q receives a message *INIT* (necessarily from a process s because the processes are connected by bidirectional channels), it broadcasts the message *WITNESS*, if this is the first message *INIT* q receives from s, and q has not yet broadcast a message *WITNESS*.

When the process q receives the message *WITNESS* from any other process, it does the following actions.

— If q receives the same message from ”enough-1” processes (where ”enough-1” is $n-2f$, i.e., at least $n-3f \geqslant 2f+1$ correct processes sent this message and q

has not yet broadcast the same message *WITNESS*, it sends it to all processes. This concludes the "forwarding phase" of q in terms of a s message.

— If q receives the same message from "enough-2" processes (where "enough-2" indicates "at least $n - f$ processes", i.e., the message was received from at least $n - 2f \geqslant 3f + 1$ correct processes), q *brb*-delivers m locally if it hasn't already. As far as q is concerned, the "delivering phase" of a message from s is now complete.

2.2.2. Correctness

The algorithm from Listing 2 assumes that $f < \frac{n}{5}$, meaning that if $f < \frac{n}{5}$ then algorithm implements a Byzantine reliable broadcast abstraction.

Assume that *INIT* is a message that a correct process q never sends out. No correct process will forward the message *WITNESS* if it was broadcast by the Byzantine process.

Consider the worst-case scenario, in which exactly f processes are Byzantine and each broadcasts the same message *WITNESS*. The forwarding predicate (`m != EMPTY such that (p in Π | witnesses[p] = m) > n - 2f and sentWitness = false`) must be satisfied for s to forward this message correctly after. However, for this predicate to be true at a correct process s, the message *WITNESS* must be received from $n-2f$ distinct processes. This is impossible since $n - 2f > f$. This means that we have just proved that if *WITNESS* was broadcast by the Byzantine process, no correct process would forward it.

Let's prove the *validity* property. Let q be a correct process that calls broadcast of message m and sends the message *INIT* as a result. The following observation explains why no proper method *brb*-delivers a message that differs from m. A correct process must receive the message *WITNESS* from more than $n - f$ different processes in order to *brb*-deliver a message m. However, if (at most) f Byzantine processes forge a fake message *WITNESS* containing $m' \neq m$, this message will never be sent by correct processes because of the proved property before. Because $n - f > f$, the content of the message *brb*-delivered by any correct process cannot differ from the content of the message *brb*-delivered by any incorrect process.

The *integrity* property arises directly from the *brb*-delivery predicate (`m != EMPTY such that (p in Π | witnesses[p] = m) > n - f and` `delivered = false`), namely, any correct process q can only deliver one message m.

The algorithm's *no duplication* property is correct due to the properties and subsequent use of authenticated perfect links and the algorithm itself.

Let q be a correct process that calls brb-broadcast m and sends the message *INIT* as a result. So, any correct process s will receive this message. Remember that there is a channel linking s and q due to the network connectivity assumption, therefore the message *INIT* cannot be a fake message produced by a Byzantine process. Furthermore, no message *WITNESS* containing m' forged by Byzantine processes with $m' = m$, can be sent by a correct process. As a result, when s receives *INIT*, the message *WITNESS* is broadcast.

Allow q to be a correct process that brb-delivers the message m. Consequently, q received the message *WITNESS* from at least $n - f$ separate processes, i.e., from at least $n - 2f > f$ correct processes, as the brb-delivery predicate (`m != EMPTY such that (p in Π | witnesses[p] = m) > n - f and delivered = false`)) is true. Therefore, at least $n - 2f$ correct processes broadcast *WITNESS*, and as a result, predicate `m != EMPTY such that (p in Π | witnesses[p] = m) > n - 2f and sentWitness = false` is eventually true for every correct process. As a result, if it hasn't already, every correct process broadcasts the message *WITNESS*. Because there are at least $n - f$ correct processes, each of them receives *WITNESS* from $n - f$ different processes, and brb-delivers the message m at some point.

In this subsection, we have seen proofs of some of the properties of Byzantine reliable broadcast that must be satisfied in order to say that an algorithm implements a specified interface. A more detailed proof of all properties can be found in the *article* [6].

2.2.3. Performance

There are two communication steps in the algorithm. *INIT* and *WITNESS* are the only types of messages that are used in the algorithm. So, it is clear that a correct process for broadcasting a message necessitates two separate communication steps (broadcast of an *INIT* message whose receptions entail at most n broadcasts of *WITNESS* message). For broadcasting one message, it needs $n^2 - 1$ point-to-point messages in total, if we do not include the messages that a process sends to itself and assume that the sender is a correct process — it requires $n - 1$ *INIT* messages and at most $n \cdot (n-1)$ *WITNESS* messages. This algorithm uses an authenticated connection concept as well, which may be implemented using a MAC (see section 1.2), that is, it does not use computationally expensive digital signatures.

In this section we have explored a new *signature-free Byzantine-tolerant reliable broadcast algorithm proposed by Damien Imbs and Michel Raynal* [6] that requires just two message delays, i.e. two subsequent communication steps, and $n \cdot (n-1) + (n-1) = n^2 - n + n - 1 = n^2 - 1$ messages in total. With a weaker resilience to the number of allowed Byzantine processes, namely $f < \frac{n}{5}$ instead of $f < \frac{n}{3}$, the advantage in both time and number of messages is gained in comparison to *Bracha's algorithm* [1]. This illustrates a fascinating tradeoff between communication cost (number of communication steps and amount of messages) on the one hand, and fault-resilience on the other (Table 1).

So, summing up, the algorithm uses $O(n^2)$ messages for one iteration similarly to the *Bracha's algorithm* [1], the number of Byzantine processes in it cannot exceed $f < \frac{n}{5}$, the algorithm uses two message delays and does not use digital signatures. The algorithm that we reviewed in this section shows a fascinating tradeoff between communication efficiency and resilience to the number of allowed Byzantine processes f.

2.3. Byzantine Reliable Broadcast Algorithm by I. Abraham [et al.]

In this section we explore good-case latency asynchronous Byzantine fault-tolerant broadcast (Byzantine reliable broadcast), which for distributed systems with n processes allows $f < \frac{n}{3}$ Byzantine processes, requires two communication steps for broadcasting a single message, and uses digital signatures. When the designated broadcaster is not faulty, the good-case latency is the time it takes for all non-faulty processes to commit, or in other words, it is the time to all correct processes to deliver the same message m. The algorithm requires $O(n^2)$ messages to be sent over the network during a single broadcast to all correct processes to commit.

2.3.1. Overview

At this subsection, we will explain the algorithm described in Listing 3. This algorithm is communication efficient (requires two message delays) and resilience optimal ($f < \frac{n}{3}$). The bottleneck of the algorithm is that it uses digital signatures, Listing 3 – Asynchronous Byzantine-tolerant reliable broadcast algorithm by I. Abraham and others with the use of digital signatures, pseudocode

```
Implements:
    ByzantineReliableBroadcast, instance brb, with sender s.

Uses:
    AuthPerfectPointToPointLinks, instance al.

// only process s
upon event { brb, Broadcast | m } do
    forall q in Π do
        trigger { al, Send | q, [PROPOSE, m] }

upon event { al, Deliver | p, [PROPOSE, m] } such that p = s
  and receivedProposal = false do
    receivedProposal := false
    forall q in Π do
        // signing messages before sending
        trigger { al, Send | q, [VOTE, m] }

upon event { al, Deliver | p, [WITNESS, m] } do
    // verifying signature before putting
    if votes[p] = EMPTY then
        votes[p] := m

    if m != EMPTY such that #({p in Π | votes[p] = m}) > n − f
      and delivered = false do
        forall q in Π do
            // signing messages before sending
            trigger { al, Send | q, [VOTE, m] }

        delivered := true
        trigger { brb, Deliver | s, m }
```

which may affect the overall latency of the algorithm in a negative sense. For one broadcast, the algorithm needs to forward $O(n^2)$ messages over the network. That is, about as many messages as are needed by the other algorithms considered in the work. The algorithm is Byzantine-tolerant because satisfies the Byzantine reliable broadcast interface, and meant to be used in an asynchronous system. It was proposed by Ittai Abraham [et al.] in the *article* [4].

Let's consider a broadcast of a single message m with a dedicated source s. The dedicated source process s sends the *PROPOSE* message to all participants of a system, including itself.

When a broadcast participant receives the first proposal, that is, the first *PROPOSE* message from the source process s, it sends a signed *VOTE* message to all participants in the system.

When each of the participants receives a $n - f$ verified signed messages for a specific broadcast in which the message m was sent, it sends a *VOTE* message to all participants of the system, triggers deliver event for the message m (brb-delivers m) and terminates.

2.3.2. Correctness

In this subsection, we will have a look at the algorithm from Listing 3 and prove that it satisfies the properties of the Byzantine reliable broadcast, then consider its main parameters and evaluate the performance.

The algorithm satisfies the *no duplication* and *integrity* properties because of the algorithm itself, and the properties and subsequent use of authenticated perfect links.

If any two correct processes commit distinct values at the final step, the traditional quorum intersection argument requires two sets of $n - f$ *VOTE* messages to intersect at $\geqslant 2 \cdot (n - f) - n \geqslant f + 1$ processes, implying that some correct process transmits votes for different values, which is a contradiction. So, the *consistency* is satisfied.

If the broadcaster is a correct process, it will deliver the same message m to all system's participants. The *VOTE* message containing m will then be broadcast by all $n - f$ correct processes. Since $f < n - f$, the Byzantine processes cannot convince any correct process to commit on a different message. After receiving $n - f$ *VOTE* messages at the last step, all correct processes will finally commit m and terminate. The *validity* property is proved.

So, if the broadcaster is correct, two communication steps are required to commit. It means that the commit latency is two rounds.

If a correct process q commits m and terminates, all other correct processes will eventually commit and terminate as a result of the sent $n - f$ *VOTE* messages for m, which proves *totality*.

2.3.3. Performance

The *algorithm* [4] requires two communication steps for a single message broadcast, which is pretty clear from the number of types of messages used in the algorithm — *PROPOSE* and *VOTE*. First, a correct source process broadcasts *PROPOSE* messages to all the participants of the system. Second, participants after receiving proposals broadcast *VOTE* messages containing broadcast message m.

$O(n^2)$ messages need to be transmitted over the network during a single message broadcast. All the messages are point-to-point.

Also, this algorithm uses digital signatures, and in the next chapter we will reveal how this affects the real latency. The algorithm is resilience optimal, which means that $f < \frac{n}{3}$ but, at the same time, it uses only two communication steps, and comparing to *Bracha's algorithm* [1] it illustrates an interesting tradeoff between the number of communication steps and the usage of digital signatures (Table 1).

So, summing up, the algorithm uses $O(n^2)$ messages for one iteration, similarly to the previous two ones, the number of Byzantine processes in it cannot exceed $f < \frac{n}{3}$, which means that the algorithm is resilience optimal, and uses two communication steps for a single message broadcast. It is worth noting that the algorithm uses digital signatures, and the influence on the real-world latency of cryptographic computations will be revealed in the next chapter. It shows an interesting tradeoff between the number of communication steps and real-world latency because of the use of cryptographic signatures.

Conclusions on Chapter 2

All the algorithms that interest us in this work were considered in this chapter. For all the algorithms considered, theoretical estimates of their performance and proofs of the properties of Byzantine reliable broadcast were given. We have shown all the estimates given in Table 1, provided pseudocodes for each of the algorithms, and analyzed them in detail.

We have once again made sure that each of the algorithms has its own bottleneck, and without an assessment in practice in real or near-to-real conditions, it is difficult to say or even give an approximate estimate of the performance of each of them, and it is even more difficult to name specific cases when each of the algorithms is most suitable. This is exactly what we will do in the next chapter.

CHAPTER 3. BENCHMARKING AND ANALYSIS

In this chapter, we will show the setup for experiments — we will talk about the simulation of a distributed system on a single machine, conduct various experiments, and draw conclusions about the usage scenarios for each of the algorithms under consideration.

3.1. TCP vs UDP

In this section, we will look at two protocols for sending messages — TCP and UDP. We will look at the pros and cons of each of the protocols and try to understand which of them is most suitable for our theoretical model from chapter 2, or to be more precise, which of the protocols can be used as an authenticated perfect link from the subsection 1.2.6, possibly with some extensions or modifications.

When it comes to the interaction between two nodes in a distributed system, HTTP immediately comes to mind. Therefore, one of the ways to simulate a distributed system is to imagine that we have various nodes that interact with each other over HTTP. So, it would be interesting to look at the pros and cons of such an approach.

Listing 4 – Source node handlers

```
http.HandleFunc("/bracha/broadcast", bracha.Broadcast)
http.HandleFunc("/bracha/deliver?type=send", bracha.Send)
http.HandleFunc("/bracha/deliver?type=echo", bracha.Echo)
http.HandleFunc("/bracha/deliver?type=ready", bracha.Ready)

http.HandleFunc("/imbs/broadcast", imbs.Broadcast)
http.HandleFunc("/imbs/deliver?type=init", imbs.Init)
http.HandleFunc("/imbs/deliver?type=witness", imbs.Witness)

http.HandleFunc("/abraham/broadcast", abraham.Broadcast)
http.HandleFunc("/abraham/deliver?type=propose", abraham.Propose)
http.HandleFunc("/abraham/deliver?type=vote", abraham.Vote)

log.Fatal(http.ListenAndServe(":8080", nil))
```

Let’s first describe the model in more detail. Let there be one dedicated node that will be the broadcast source, which means that only this dedicated one can send broadcast messages. Let’s now consider the handlers of the source node. As can be seen in Listing 4, each of the algorithms under consideration has a handler for initiating broadcast, and there are handlers for processing received messages, depending on the type of the message. All other nodes of the system will have all the same handlers, except the handler that allows them to initiate the broadcast. This assumption was made in order to simplify the experiments. In fact, it does not matter at all which node initiates the broadcast.

For example, for Bracha's algorithm, we have three types of messages — *SEND*, *ECHO*, and *READY*, and therefore there are corresponding handlers for each type. And the same is true for the other two algorithms.

Since the Golang programming language was initially chosen to implement algorithms, and since there is no way to explicitly create threads in it, but only goroutines, it was decided to run each of the nodes in a separate goroutine, as can be seen in Listing 5.

Listing 5 – An example of starting a system with n nodes in separate goroutines

```
wg := new(sync.WaitGroup)

for port := 9000; port < 9000 + n; port++ {
    wg.Add(1)
    go func(port int) {
        server := createServer(
            fmt.Sprintf("http://localhost:%d", port), port
        )
        log.Fatal(server.ListenAndServe())
        wg.Done()
    }(port)
}

wg.Wait()
```

At the very beginning, a `WaitGroup` is created, which allows us to await the execution of all goroutines. Then n HTTP servers are created, each of which runs on a separate port, starting from 9000. At each iteration, 1 is added to the value of the `WaitGroup`. This allows us to correctly calculate the actual number of successfully launched servers and then await their completion.

The `createServer` function returns an instance of a server that has all the necessary handlers for all three algorithms, and imports the logic from the corresponding packages implementing these algorithms.

This architecture is quite flexible for at least two reasons. Firstly, with such an architecture, it is quite easy to control (increase or decrease) the total number of nodes in the system. Secondly, such an architecture is quite easily transferred to a real-life distributed system. It is clear that nodes in a real-life distributed system can be located in different datacenters, ports may not be sequential at all, and hosts may also differ, but it’s all quite easy to change and redo for the needs of a real-life distributed system.

It is important to clarify that here we are talking more about the idea of using HTTP servers rather than using goroutines because in a real-life distributed system, each of the nodes is usually run on a separate machine, and goroutines are used only to simulate at least some level of isolation for conducting experiments locally and for simulating a distributed system locally overall. Another obvious advantage of using communication via HTTP is that, due to the popularity of the protocol on the web, there are a huge number of frameworks that hide implementation details and allow creating HTTP servers fairly quickly.

It would seem that everything is fine and that we can safely use the above approach for experiments, but this is not quite true. The problem is that, firstly, HTTP is a synchronous protocol. This means that any client that sends an HTTP request must wait for a response and only then continue the subsequent work. That is, in our case, since our client is a node of a distributed system, and each node runs in a separate goroutine, this corresponding goroutine in which the node runs is blocked until it receives a response to the sent HTTP request, which does not correspond to the theoretical model, which requires sending messages to be asynchronous. Moreover, HTTP works over TCP, and TCP, for example, is relatively slower than UDP, and TCP also controls the order of message delivery between nodes, guaranteeing FIFO-delivery.

Of course, TCP, for example, can take care of lost packets by itself, providing opportunities for this, and also has quite impressive error checking and data confirmation, but requires a connection to send data, and has requirements for closing connections after use, and so on, while UDP is connectionless.

So TCP doesn’t suit us. What about UDP? If we recall the necessary properties for a link to implement an authenticated perfect link abstraction, then these are *reliable delivery*, *no duplication* and *authenticity*. UDP does not provide any of these properties by itself, but all three can be met by introducing fairly light additions above it. For example, to guarantee *reliable delivery*, we can resend packets ourselves since UDP itself does not guarantee the delivery of the sent packet, and it may simply get lost. To guarantee the *no duplication* property, we can create a hash table with already delivered messages and check every time after receiving a message whether we have received it before, and if so, just ignore it. To comply with the *authenticity* guarantee, we can either agree ourselves that one process in our system cannot pretend to be another and send messages on behalf of another process, or use a MAC.

Thus, we have shown that with small modifications, UDP is ideal for our purposes, which is why we will use it.

3.2. Node of a distributed system as a UDP server

In this section, we will implement a single instance of our distributed system as a UDP server that will be able to receive messages and thus implement each of the algorithms under consideration.

As we decided in the previous section, we will use UDP to communicate between nodes. Each message will usually consist of four parts — the type of message, from whom the message was sent, to whom it was sent and the message itself. The parts will be separated by underscores. For example, a typical message might look like this: ”`Echo_localhost:9000_localhost:9002_hello!`”. The meaning of this message is that the node that is running on port 9000 sent an *ECHO* message with the content ”hello!” to the node on port 9002.

It is also important to mention that since we agreed that the only one dedicated process broadcasts messages, then broadcast type messages will consist of two parts instead of four. Such messages will contain only the message type and the message itself. So, the typical broadcast message will look like ”`Broadcast_hello!`”.

It is clear that in this case we have to somehow escape the underscores, but the experiments did not use messages containing underscores, so this does not bother us. In addition, you can come up with some more complex ways of splitting and parsing parts quite easily.

Since it is impossible to explicitly create threads in Golang, it was decided to use the Kotlin programming language. Kotlin is a very convenient language and makes it easy to use libraries from Java, and the `java.util.concurrent` package provides a wide range of tools for multithreading. Now let’s move on to the details. Each node will be an instance corresponding to a class for a particular algorithm, but all classes will inherit from the `Thread` class, and will override the `run` method. For the *Bracha's algorithm* [1], such a method may look like this, as in Listing 6.

Listing 6 – The implementation of `run` method for *Bracha's algorithm* [1]

```
override fun run() {
  running = true

  while (running) {
    val packet = utils.receivePacket()
    val parts = packet.data.split("_")

    when (parts[0]) { // command
      "Broadcast" -> broadcast(parts[1]) // message
      "Send" -> send(parts[1], parts[3]) // from & message
      "Echo" -> echo(parts[1], parts[3])
      "Ready" -> ready(parts[1], parts[3])
      "End" -> running = false
      else -> {
        running = false
        log.println("unexpected command '${parts[0]}'")
      }
    }
  }

  utils.socket.close()
}
```

Listing 6 shows that each of the message types from *Bracha's algorithm* [1] is processed. Message processing occurs as follows: first, a packet is delivered by calling the `utils.receivePacket()` method, then the data from the packet is divided into parts and then parsed, and, depending on the type of message, the corresponding handler is called, or the process is terminated with the subsequent closure of the socket if the "End" message or some other unsupported message was received.

3.3. Simulation of a distributed system

In this section, we will look at different ways to launch each of the instances and choose the most suitable one for us.

As already noted, each node will be a UDP server, and each node will be launched in a separate thread. Accordingly, we need a mechanism that allows us to create n threads and run n instances of implementations of each of the algorithms in the form of UDP servers in these threads.

There are at least 3 ways to create and manage work with n threads in the Java standard library — `ThreadPoolExecutor`, `ForkJoinPool`, and `ExecutorService`. We can also explicitly create threads ourselves, but this is often less convenient in the sense that it is not always easy to manage all resources and then close all these threads correctly.

We will notice right away that under the hood, `ExecutorService` creates either `ThreadPoolExecutor` or `ForkJoinPool`, depending on the method called. This is how, for example, the implementation of the newFixedThreadPool method looks like:

Listing 7 – The implementation of `newFixedThreadPool` from `ExecutorService`

```java
public static ExecutorService newFixedThreadPool(int nThreads) {
  return new ThreadPoolExecutor(
    nThreads, nThreads,
    0L, TimeUnit.MILLISECONDS,
    new LinkedBlockingQueue<Runnable>()
  );
}
```

`ForkJoinPool` is designed to work with a certain type of tasks that are suitable for the Fork & Join framework. In our case, we just want to run n servers on n threads. In addition, `ForkJoinPool` has a mechanism that allows to ”steal” tasks from other threads. This, of course, speeds up execution, when the number of tasks is greater than the number of threads, for example, but we simply don't need that either.

Thus, `ThreadPoolExecutor` is most suitable for us, and, therefore, we will use it. For convenience, we will use `Executors.newFixedThreadPool` to create a thread pool. Listing 8 shows how this might look like. At each iteration, one instance is created, which will represent one node of the system. All the necessary parameters are passed to it, and the instance itself is also stored in a local variable that will be needed for graceful shutdown.

We will also have a class for the client (see Listing 9) that will send messages over UDP, which we will mainly use to initiate broadcasts. Such a client will receive

Listing 8 – Starting nodes in separate threads example

```
val executor = Executors.newFixedThreadPool(n)
val instances = mutableListOf<BrachaAlgorithm>()

for (port in startPort until startPort + n) {
  try {
    instances.add(BrachaAlgorithm(startPort, port, hosts, n, f))
    executor.execute(instances.last())
  } catch (e: Exception) {
    println("$i: ${e.message}")
  }
}
```

message, as well as the port number to which this message should be sent, form a Datagram and send it.

Listing 9 – UDP client

```
class UDPClient {
  private val socket: DatagramSocket = DatagramSocket()
  private val address: InetAddress =
      InetAddress.getByName("localhost")

  fun send(msg: String, port: Int) {
    val buf = msg.toByteArray()

    val packet = DatagramPacket(buf, buf.size, address, port)
    socket.send(packet)
  }

  fun close() {
    socket.close()
  }
}
```

It is also important to mention one very important point. When it comes to a real-life distributed system, there are necessarily some network delays in it. We need to learn how to simulate them somehow on the same machine. In a real-life distributed system, everything would happen asynchronously, and our model requires the same, so it won't work to use just `Thread.sleep`, since sleep blocks the thread. We need to either make an asynchronous sleep, or, for example, await sleep explicitly in another thread and then send messages there.

After considering several options from the Java standard library, it was decided to use `ScheduledExecutorService`, which can be created, for example, by calling `Executors.newScheduledThreadPool(n)`, passing the number of threads instead of n. It just meets our requirements, allowing asynchronous message sending (without blocking the current thread) after the specified delay. Thus, before conducting experiments, you can create a pool of n threads and then reuse it. Thus, before conducting experiments, we can create a scheduled pool of n threads and then reuse it. This will be another pool, that is, not the same one that is used to run the instances themselves — nodes of a distributed system.

Note that in a real-life distributed systems there are no fixed message delays, so we will use a Gaussian normal distribution with different values of expectation and standard deviation. There are whole *articles* [10] on this topic, which are devoted to studying the behavior of message delays for different protocols. In Java, this can be done as follows. In `java.util.Random` there is a `nextGaussian` method that can return random values with a mathematical expectation equal to 0 and a standard deviation equal to 1. Therefore, in order to get a value in the range from `expectation ± deviation`, we need to do something like `expectation + (Random().nextGaussian() * (deviation / 2))`.

So, in this section we have modeled a distributed system that fully meets our requirements — it is easy to add new nodes to it and it is easy to modify and set various message delays in it.

3.4. Assumptions and justifications on different system and experiment parameters

In this section, we will discuss assumptions made on different system parameters such as message length, latency measurement, and classification of Byzantine processes, and try to justify them.

3.4.1. Message length

In this subsection, we will reflect on the length of the transmitted messages and show that there is not much point in doing different experiments with different message lengths.

We just need to send a message m that was broadcast to all participants exactly once, so that the participant, if it is decided to deliver the message m, knows what to do next, since his further behavior depends on the received message. All other messages that are transmitted over the network during the broadcast in all the algorithms under consideration are service messages and therefore it is possible to transmit, for example, initial message hashes or keep some kind of serial number and transmit only it. Therefore, there is no need for experiments where messages of different sizes are transmitted, since hashes for such messages are of a rather limited size and only they can be transmitted.

First of all, we have shown that the behavior of algorithms does not depend on the size of the message being transferred, and that there is no particular point in changing the size of the transmitted message while doing experiments. So, next, we will assume that all transmitted messages have the same size.

3.4.2. Latency measurement

In this subsection, we will talk about the approach of how to measure latency for asynchronous algorithms that do not have any trigger that allows to understand that the algorithm is terminated.

The problem is that we have no way to understand from the algorithm itself that it has completed, and, accordingly, there is no way to measure its execution time.

In order to solve this problem, we will use the following approach. Each of the processes will have its own file with logs, logs will be written with timestamps accurate to milliseconds. When the broadcast is initiated, we will also record the initiation time in a separate log file. Thus, for n processes, we will have $n + 1$ files (or $n - f + 1$, to be more precise, where f is the number of Byzantine processes). After conducting the experiments, we will parse all the files and be able to calculate the latency.

How to calculate average latency? To begin with, for each message (0000, 0001, . . .), we will find the longest time among all the log files to which it was delivered, that is, we find the most recent process that delivered this message and then subtract this time from the broadcast initiation time, so we will get the delivery time of one individual message. After doing the same for all other messages, and then summing up all the delivery times and dividing it by the total number of broadcasts, we get average latency.

Thus, we say that the broadcast delivery time is equal to the message delivery time until the last correct process in the system.

Listing 10 – Examples of result log files

```
2022.05.24 03:55:42.512 started broadcast: 0000
2022.05.24 03:55:46.695 started broadcast: 0001
2022.05.24 03:55:50.853 started broadcast: 0002
...

2022.05.24 03:55:44.603 localhost:9000 delivered message 0000
2022.05.24 03:55:48.787 localhost:9000 delivered message 0001
2022.05.24 03:55:52.944 localhost:9000 delivered message 0002
...

2022.05.24 03:55:44.603 localhost:9010 delivered message 0000
2022.05.24 03:55:48.787 localhost:9010 delivered message 0001
2022.05.24 03:55:52.945 localhost:9010 delivered message 0002
...
```

It is important to note that since the experiments are conducted on a single machine, there is no need to come up with globally synchronized time [3], which makes the latency calculation much easier.

3.4.3. Classification of Byzantine processes

In this subsection, we will consider possible Byzantine failures, classify them and highlight the most interesting to us.

Firstly, it is important to clearly separate two cases — when the sender is Byzantine, and when the sender is a correct process. In this work, we will mainly focus on the case when the broadcast initiator is a correct process, since at least an estimate of the number of message delays for the algorithm proposed by I. Abraham [et al.] [4] in Table 1 is given for the case when the broadcast initiator is correct. The latency of such a case is also called a *good-case*.

Secondly, let's now classify the Byzantine processes, i.e. other participants of the system. We are definitely interested in cases when Byzantine processes do nothing, send incorrect messages or do equivocation, i.e. send different messages to different processes, trying to confuse the algorithm. We are definitely not interested in cases when the Byzantine process is trying to make a DDoS attack, because identifying such attacks is a separate huge interesting task, which is now can be done even using neural networks [9], and which is definitely not the point of our interest. Also, we are not interested in cases when the Byzantine process tries to send large messages, respectively, by splitting them into several datagram packets, since we agreed that we would send only service messages, and the original message itself, which was broadcast, can sent by its hash.

Thus, we are interested in three types of Byzantine processes — when the process is disconnected and/or just does nothing, when the process sends different messages to different participants, trying to confuse everyone in the system, and when the process simply responds incorrectly.

3.5. Implementation details

In this section, we will talk about the implementation specificity of an implemented module for conducting experiments.

To conduct experiments, a Kotlin program was implemented, which can receive an algorithm in command-line arguments (one of the three algorithms considered in this work must be transmitted), the total number of processes in the system n, the number of allowed Byzantine processes f, desired message delay and the ID of the message being sent. All messages consist of 4 characters and are numbered starting from zero. Messages whose length is less than 4 are padded with zeros on the left. Thus, all messages transmitted in the system look like 0000, 0001, ..., 9999.

Each experiment is performed for a fixed pair of the number of allowed Byzantine processes f and desired message delay. The total number of processes n is chosen depending on f, and the algorithm, for example, for *Bracha's algorithm* [1] n is equal to $3f + 1$, and for the *Imbs and Raynal's algorithm* [6] n is equal to $5f + 1$.

In each experiment, all three algorithms are considered, and message ID varies from 0 to 9999. That is, 10,000 broadcasts are made for each of the algorithms and only then the average time is calculated.

Listing 11 – A minimalistic example for collecting logs for further latency calculation

```
for delay in [0.25, 0.5, 0.75, 1, 2, 5, 10, ...]:
  for algo in ['Bracha', 'Imbs', 'Abraham']:
    for f in range(1, maxF(algo)):
      for msgId in range(10000):
        os.system(f"java -jar broadcasts.jar"
                  f" {algo} {minN(f)} {f} {delay} {msgId}")
```

In Listing 11, the maximum value of f is chosen depending on the algorithm, this is done because n depends on f, and we cannot do infinitely large n, therefore, for example, for the *Imbs and Raynal's algorithm* [6], the maximum f will be 12, since with such f, the value of n will be 61, which is equivalent to running 61 threads, one for each distributed system's node and another 61 threads for asynchronous message sending in the worst case.

The algorithm implementations themselves are not very interesting, as they are quite simple. At least implementations of *Bracha's algorithm* [1] and *Imbs and Raynal's algorithm* [6]. Therefore, in the next subsection, let's talk about the choice of digital signatures for the *algorithm of Abraham [et al.]* [4].

3.5.1. RSA vs Ed25519

Between public-key encryption algorithms, there was initially a choice between the very popular and widely used RSA and the slightly less well-known Ed25519. It was decided to use Ed25519, since this algorithm uses a much smaller number of bytes, since the key sizes of Ed25519 are much smaller than RSA keys. Moreover, the Ed25519 is better in performance and more secure compared to RSA [7]. Naturally, if our priority was compatibility, then RSA has no competitors with this and we would undoubtedly take advantage of it.

Thus, the Ed25519 algorithm was chosen to generate and verify digital signatures for the *Abraham's [et al.] algorithm* [4].

3.6. Experiments results

In this section we will talk about the conducted experiments, and their results.

In order to make sure that the distributed system is modeled correctly, it is important to make sure that latency does not grow with the number of processes and/or the number Byzantine processes. Initially, incorrect attempts were made to simulate message delay, since `sleep` was done in the same thread and, moreover, synchronously. It obviously blocked the thread and execution was interrupted. Because of this, with the increase in the number of processes, the latency also grew. Then `sleep` in the same thread was replaced with asynchronous sleep using `ScheduledExecutorService` and there was no linear growth anymore. Moreover, initially the experiments were conducted using Golang programming language and the work with process isolation was incorrectly organized there, which also affected latency, since in that case it also grew linearly. With the transition to Kotlin and the explicit creation of threads for each node of a distributed system, linear growth has disappeared again.

Figure 2 shows that *Bracha's algorithm* [1] works in about 3 message delays, and the *Imbs and Raynals'* [6] and *Abraham's [et al.]* [4] algorithms work in 2, as expected and as shown in Table 1. Moreover, there is no linear dependence from the number of Byzantine processes, and hence from the total number of processes, too, which is also good news.

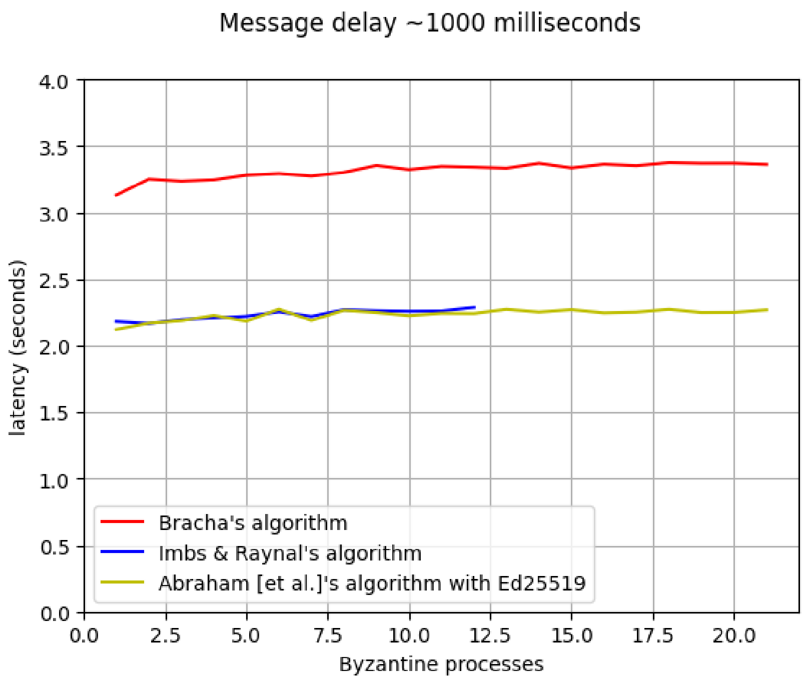

Figure 2 – Benchmarking results for significantly big message delays

This shows that the system was modeled correctly and that the right tools and primitives were chosen for its implementation. The message delay of 1000 milliseconds was chosen in order for local calculations of each node to be as insignificant as possible in relation to message delay, in order to make sure that there is no linear dependency of latency from the number of processes.

Various integration tests were also written that check the correctness and fulfillment of the properties of the Byzantine reliable broadcast for each of the implementations.

Further experiments were carried out for various classes of Byzantine processes, which we identified in the subsection 3.4.3. For the selected classes of Byzantine processes, the average latency time did not differ much, therefore, as the most difficult for handling and tolerating, it was decided in further experiments to use the type of Byzantine failure when the faulty process does nothing, i.e., it does not respond to none of the messages sent.

If it is not very important for us to be able to tolerate a certain number of Byzantine processes, and we believe that the number of such processes is negligible, but at the same time we want to have as many nodes in the system as we can, then *Imbs and Raynal's algorithm* [6] will suit us.

The algorithm does not use digital signatures and, moreover, needs only two message delays, which is very good for us.

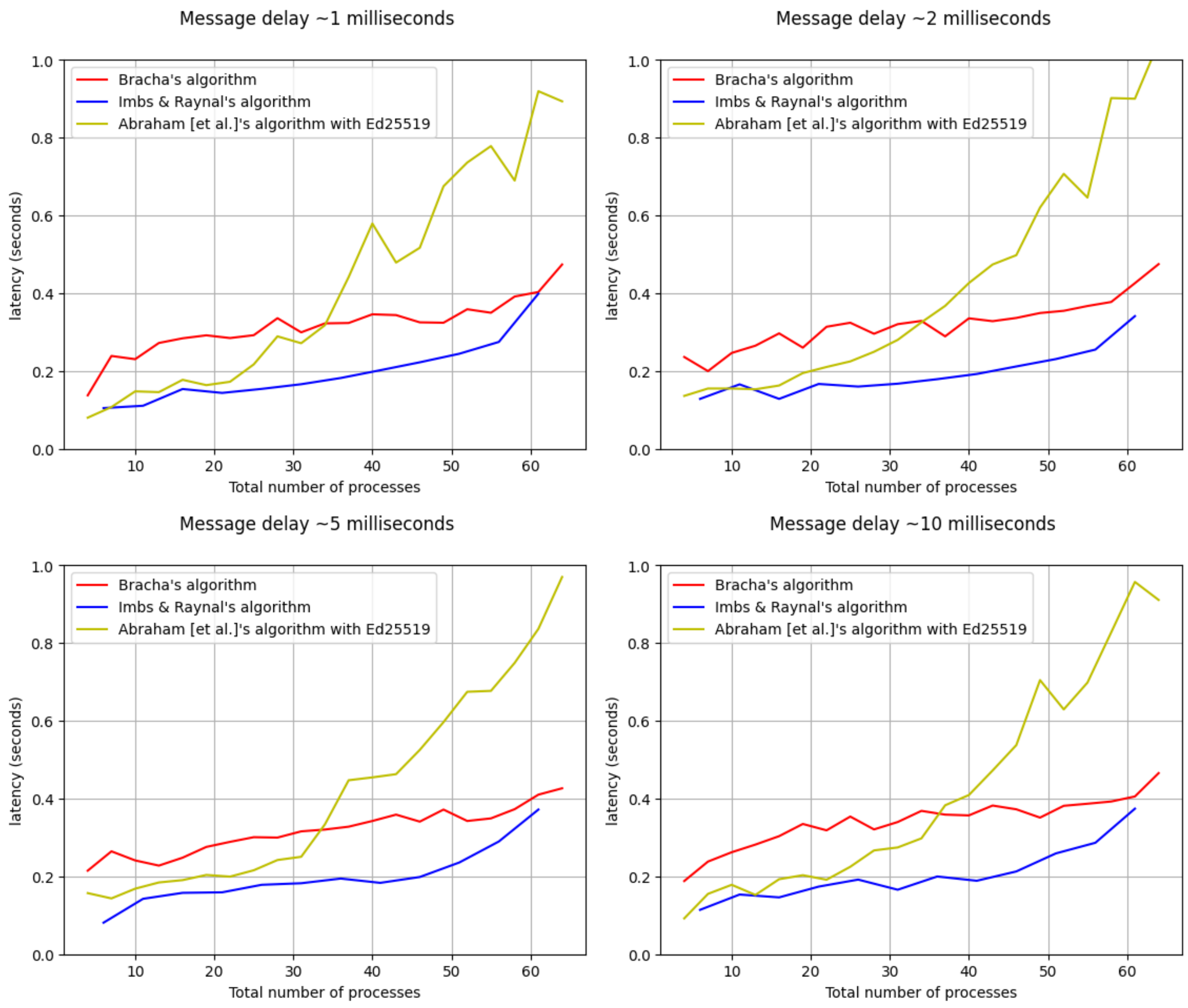

Figure 3 – Benchmarking results for small message delays

That is, specifically in our case, the algorithm is in some sense the most optimal even in the theoretical aspect, since reliable delivery cannot be guaranteed for less than 2 message delays [4]. And we also don't have an overhead from using cryptographic abstractions.

Figure 3 shows the dependence of latency on the total number of processes in the system. As we can see, with all the message delays set, the *Imbs and Raynal's* *algorithm* [6] outperforms the other two. Moreover, it is important to note that at first the number of Byzantine processes was fixed and, depending on this value, the total number of processes was selected, which could tolerate a given number of Byzantine processes. The total number of processes n is calculated using the simple formulas $n = 3f + 1$ and $n = 5f + 1$, depending on the algorithm, where f is the maximum number of Byzantine processes allowed.

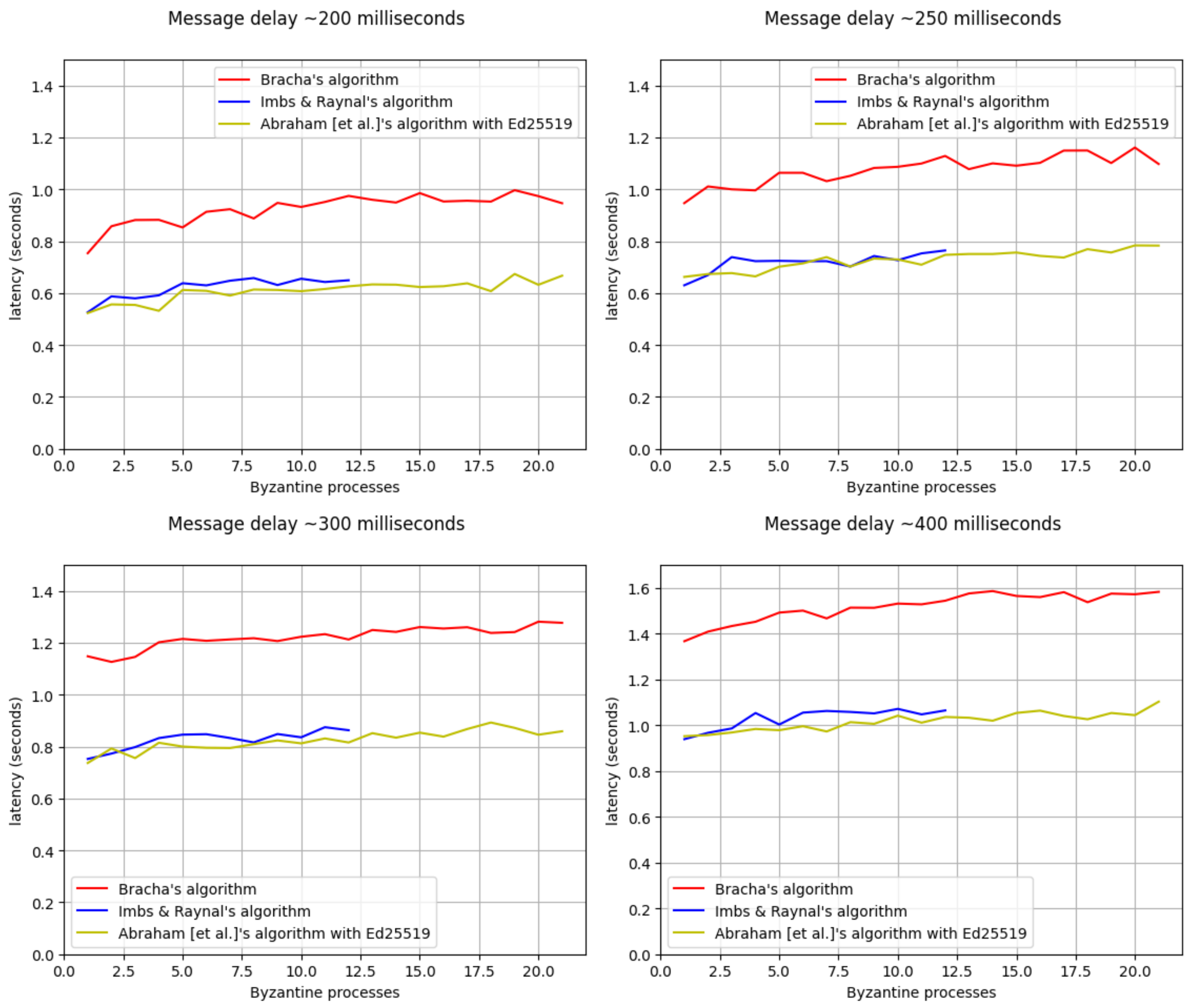

Figure 4 – Benchmarking results for big message delays

Figure 4 shows the dependence of latency on the number of Byzantine processes. Then, as the message delay increases, we observe that the *Abraham's [et al.] algorithm* [4] performs better than the other two algorithms. Why it performs better than *Bracha's algorithm* [1] is generally understandable, because *Bracha's algorithm* [1] requires 3 message delays, and this algorithm requires only 2. But why does it performs better than *Imbs and Raynal's algorithm* [6]? All because the *Imbs* *and Raynal's algorithm* [6] requires more processes to tolerate the fixed number of Byzantine processes f. Moreover, the message delay is getting quite large and the percentage of calculations that fall on cryptographic calculations is getting smaller and smaller.

This gives us a hint that perhaps with a decrease in message delay, we will be able to pick it up so that it makes up a significant part of the cryptographic calculations and, perhaps, we will see an interesting case.

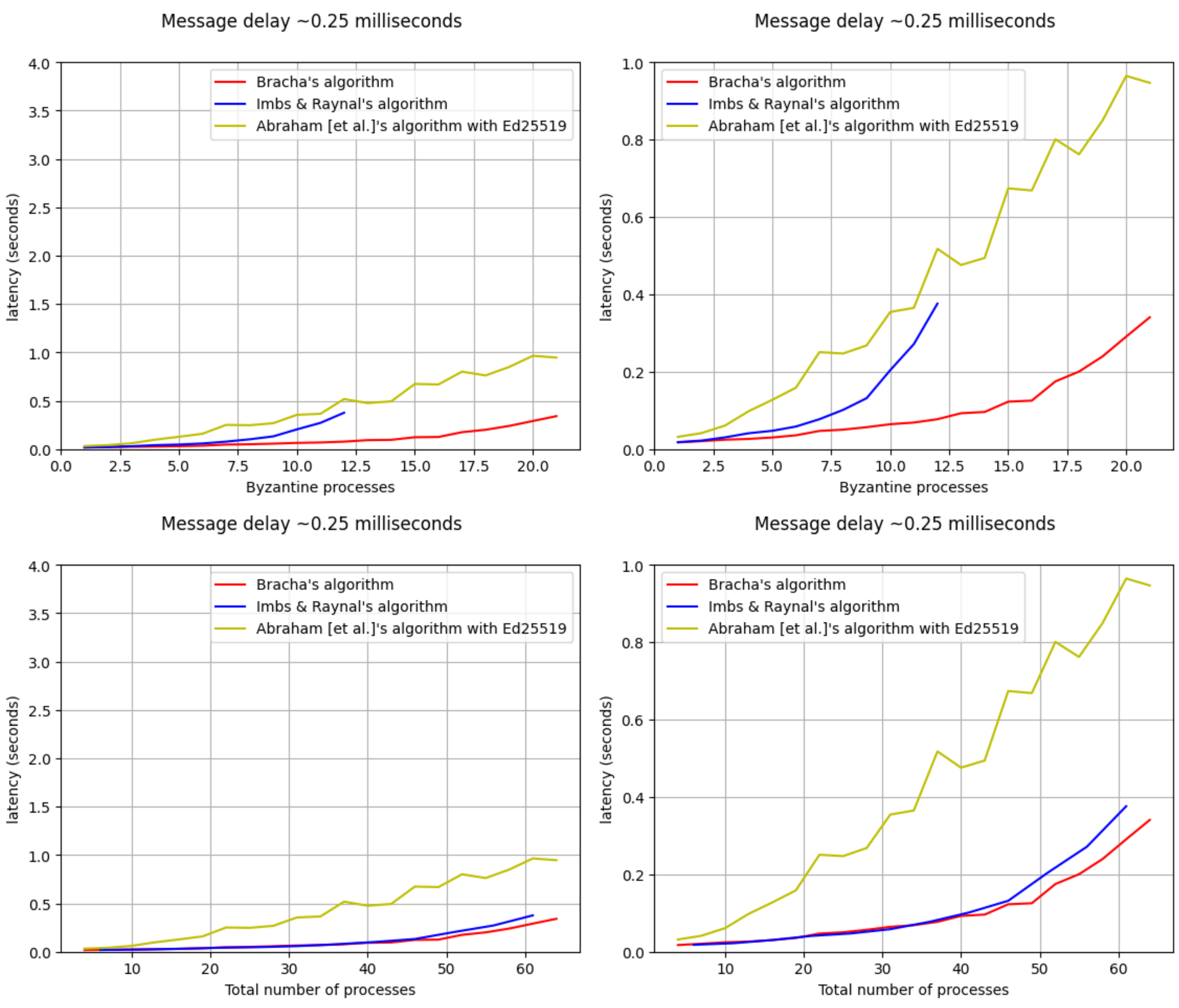

Figure 5 – Benchmarking results for significantly small message delays

This is exactly what we see in Figure 5. For a significant small message delay, the part of cryptographic computations starts to grow and the other two algorithms start to perform better. And moreover, due to the fact that the *Imbs and Raynal's algorithm* [6] requires a large number of processes, the *Bracha's algorithm* [1], which is resilience optimal, starts to perform better than the other two algorithms.

Thus, we found cases for each of the three algorithms when each of them shows results better than the other two.

Conclusions on Chapter 3

In this chapter, we talked in detail about the implementation of all three algorithms, about the method of simulating a distributed system on a local machine, and talked about the parameters that were considered during the experiments. We also told in detail how one experiment looks like, conducted experiments, and got the results. We have drawn conclusions from the results obtained, which allow us to say that we have found cases for each of the algorithms under consideration, when each of them can be applied and its application will be the most optimal.

CONCLUSION

In this work, we have revealed in detail the relevance of the topic under consideration, introduced all the necessary definitions and concepts, which later came in handy for us to substantiate and set theoretical results.

In Table 1, we showed three algorithms that implement the Byzantine reliable broadcast interface, for each of them we showed their bottlenecks.

In the last chapter, we conducted a huge number of experiments after we learned how to simulate a distributed system on a single machine, the experiments allowed us to see different tradeoffs between using cryptographic abstractions, for example, and better fault resilience. The experiments carried out allowed us to discover scenarios where the use of each of the algorithms is most optimal. All the goals and objectives of the work have been fulfilled.

Thus, with average message delays, when, for example, all nodes of a distributed system are located within the same region, but maybe in different data centers, it is necessary to use the *Imbs and Raynal's algorithm* [6].

If the nodes of a distributed system are located on different continents and the message delays are quite large, despite the use of cryptographic abstractions, *Abraham's [et al.] algorithm* [4] performs better than all others and it is recommended to use it.

If the message delay is slightly small, for example, all nodes are located inside one datacenter and close to each other, then classic *Bracha's algorithm* [1] should be used. This algorithm does not have an overhead due to the use of cryptographic abstractions, besides it is quite simple to implement.

REFERENCES

1 *Cachin C.*, *Guerraoui R.*, *Rodrigues L.* Introduction to Reliable and Secure Distributed Programming. — Berlin : Springer, 2011. — 388 p.

2 *Castro M.*, *Liskov B.* Practical Byzantine Fault Tolerance // Proceedings of the Third Symposium on Operating Systems Design and Implementation. — 1999. — Feb.

3 *Chen Y.*, *Chen T.*, *Hu W.* Global Clock, Physical Time Order and Pending Period Analysis in Multiprocessor Systems. — 2009.

4 Good-case Latency of Byzantine Broadcast: A Complete Categorization / I. Abraham [et al.]. — 2021.

5 HotStuff / M. Yin [et al.] // Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing - PODC '19. — Toronto ON, Canada : ACM Press, 2019.

6 *Imbs D.*, *Raynal M.* Trading off t-Resilience for Efficiency in Asynchronous Byzantine Reliable Broadcast // Parallel Processing Letters. — 2016. — Dec. — Vol. 26, no. 04. — P. 1650017.

7 *Isoms K.* Practical Cryptography With Go. — Leanpub, 2014. — URL: `https://leanpub.com/gocrypto`.

8 Modeling LRU cache with invalidation / A. Detti [et al.] // Computer Networks. — 2018. — Vol. 134. — P. 55–65. — ISSN 1389-1286.

9 *Rangapur A.*, *Kanakam T.*, *Jubilson A.* DDoSDet: An approach to Detect DDoS attacks using Neural Networks. — 2022.

10 *Yan H.*, *Sun G.*, *Zhang Y.* Calculation and Simulation of Message Delay on TDMA Short-wave Communication Network // Journal of Communication and Computer. — 2006. — Aug. — Vol. 3, no. 8. — P. 6.

11 HyperLedger [Electronic resource]. — URL: `https : / / www . hyperledger . org / use / distributed - ledgers` (visited on 04/19/2022).

12 Message authentication code [Electronic resource]. — URL: `https://en.wikipedia.org/wiki/Message_authentication_code` (visited on 12/11/2021).

13 Replicated State Machines [Electronic resource]. — URL: https://aeroncookbook.com/distributed-systems-basics/replicated-state-machines/ (visited on 04/17/2022).

14 What is a CDN? | How do CDNs work? [Electronic resource]. — URL: https://www.cloudflare.com/en-gb/learning/cdn/what-is-a-cdn/ (visited on 04/17/2022).

APPENDIX A. SOME OF THE IMPLEMENTED CLASSES LISTINGS

Listing A.1 – Constants and common methods

```
package common

import java.text.SimpleDateFormat

typealias Message = String
typealias HostName = String

class Utils {
  companion object {
    var basePath = "/Users/hakimov/thesis/results"
    var keysPath = "/Users/hakimov/thesis"
    var timestampFormat = SimpleDateFormat("yyyy.MM.dd HH:mm:ss.
       SSS")

    fun padLeftZeros(inputString: String, length: Int): String {
      if (inputString.length >= length) {
        return inputString
      }
      val sb = StringBuilder()
      while (sb.length < length - inputString.length) {
        sb.append('0')
      }
      sb.append(inputString)
      return sb.toString()
    }
  }
}
```

Listing A.2 – Class with Ed25519 sign and verify methods

```
package crypto

import common.Utils
import java.io.File
import java.io.FileOutputStream
import java.nio.file.Files
import java.security.*
import java.security.spec.EncodedKeySpec
import java.security.spec.PKCS8EncodedKeySpec
import java.security.spec.X509EncodedKeySpec

class Ed25519 {
  companion object {
```

```
fun sign(secretMessage: String, privateKey: PrivateKey):
   ByteArray {
  val signature: Signature = Signature.getInstance("Ed25519")
  signature.initSign(privateKey, SecureRandom())
  val message: ByteArray = secretMessage.toByteArray()
  signature.update(message)
  return signature.sign()
}

fun verify(secretMessage: String, publicKey: PublicKey,
   sigBytes: ByteArray): Boolean{
  val message: ByteArray = secretMessage.toByteArray()
  val signature = Signature.getInstance("Ed25519")
  signature.initVerify(publicKey)
  signature.update(message)
  return signature.verify(sigBytes)
}

fun generateKeys(): KeyPair {
  val generator = KeyPairGenerator.getInstance("Ed25519")
  val pair = generator.generateKeyPair()

  val privateKey = pair.private
  val publicKey = pair.public

  FileOutputStream("${Utils.keysPath}/keys/public.key").use {
     fos -> fos.write(publicKey.encoded) }
  FileOutputStream("${Utils.keysPath}/keys/private.key").use {
      fos -> fos.write(privateKey.encoded) }

  return KeyPair(publicKey, privateKey)
}

fun readKeys(): KeyPair {
  val publicKeyFile = File("${Utils.keysPath}/keys/public.key"
     )
  val publicKeyBytes = Files.readAllBytes(publicKeyFile.toPath
     ())

  val keyFactory = KeyFactory.getInstance("Ed25519")
```

```
        val publicKeySpec: EncodedKeySpec = X509EncodedKeySpec(
            publicKeyBytes)
        val publicKey = keyFactory.generatePublic(publicKeySpec)

        val privateKeyFile = File("${Utils.keysPath}/keys/private.
            key")
        val privateKeyBytes = Files.readAllBytes(privateKeyFile.
            toPath())

        val privateKeySpec: EncodedKeySpec = PKCS8EncodedKeySpec(
            privateKeyBytes)
        val privateKey = keyFactory.generatePrivate(privateKeySpec)

        return KeyPair(publicKey, privateKey)
    }
  }
}
```