ELEKTROTEHNIŠKI VESTNIK 92(3): 123-134, 2025
ORIGINAL SCIENTIFIC PAPER

Implementation of an optimized solution to the problems of asynchronous and synchronous communication between web services

Ahmed Mujić, Nevzudin Buzađija, Samir Lemeš, Denis Čeke

Dept. of software engineering, University of Zenica, Bosnia and Herzegovina
E-mail: nevzudin.buzadjija@unze.ba

Abstract. The paper compares the synchronous and asynchronous communication of services in a notification generating business process. Focusing on a service-to-service communication and demonstrating a migration process from the synchronous to the asynchronous communication. The performance is measured and potential issues are identified for certain approaches throughout the entire process. By comparing the performance and highlighting the potential problems the question is answered which communication type is a better choice for a particular case. The comparison result helps programmers understand the synchronous and asynchronous communication through practical examples and supports them in their making decision. During the migration process, the service remains the same, only the communication pattern changes.

Keywords: synchronous communication, asynchronous communication, Bayesian analysis, data integrity, scalability

Implementacija optimizirane rešitve za asinhrono in sinhrono komunikacijo med spletnimi storitvami

V članku primerjamo sinhrono in asinhrono komunikacijo med spletnimi storitvami na primeru procesa generiranja obvestil. Predstavljen je postopek migracije iz sinhrone v asinhrono komunikacijo, pri čemer spremljamo zmogljivost in identificiramo morebitne težave. Na podlagi rezultatov ocenjujemo, katera komunikacijska strategija je primernejša. Prispevek ponuja praktičen vpogled v delovanje obeh pristopov ter služi kot podpora razvijalcem pri odločanju v realnih sistemih. Poslovna logika storitev ostaja nespremenjena, spreminja se le komunikacijski vzorec.

1 INTRODUCTION

The microservice architecture has become one of the main architectural approaches in the development of modern software products emphasizing scalable, fast, and reliable solutions. However, as the number of microservices grows, the need for an efficient and optimized communication increases to ensure a sufficient system performance.

Nowadays, developers opt for the asynchronous code almost without hesitation, considering it more efficient, even in smaller systems with not many users. This has led the asynchronous code to become a standard today, which cannot be said for the communication of information services/systems.

This paper demonstrates the power of the asynchronous communication in a business case that can be directly improved by it. It also highlights the complexity and hidden problems coming with it and needing to be known. It analyzes particular service communications coping with a certain problem to identify the sources of the entropy and congestion, thus providing the bases for drawing conclusions. The paper primary concern is the performance. The next are potential data integrity breaches and an overall analysis.

For a particular system which uses a synchronous communication, the paper identifies its potential weaknesses and addresses them using asynchronous communication. The aim is to determine the following:

- Is the need for an asynchronous communication driven more by the scalability than by the performance?
- Which are the cases where it is not possible to use the asynchronous communication?
- How does the change in the service load affect the performance, and how do different communication patterns behave during load simulation?

Received: 23 December 2024
Accepted: 28 March 2025

2 PREVIOUS RESEARCH

The first step in understanding the concept of microservices is to review their architectural communication patterns. It is important to take accurate architectural decisions to increase the system efficiency, scalability, and resilience. The most commonly used protocols for the microservices communication are REST (Representational State Transfer), gRPC (g Remote Procedure Calls), AMQP (Advanced Message Queuing Protocol), MQTT (Message Queuing Telemetry Transport) and WebSockets.

In the literature REST and AMQP are often compared and used because they are mostly applied in modern systems. They are considered the best choice for communication because event-driven applications (with either publish-and-subscribe messaging, or HTTP and resource representation) allow decoupling unlike is the case with the point-to-point [1]. Namely, the RPC nature involves the highest coupling compared to HTTP (stateless and resource-oriented) and AMQP (asynchronous message queues).

Decoupling is an important part in developing large-scale applications because systems become more error-prone than they used to be and, it is easier to maintain loosely-coupled systems.

In a system with multiple microservices, the first of them needing the data from another microservice and the others from more microservices, etc., the multiple microservices are blocked to get the needed data. In such case, an asynchronous communication is needed because: "The asynchronous communication is mandatory since any communication often triggers other communications recursively for the receiver. Therefore, the entire tree of microservices cannot be blocked until a final response comes from all the tree leaves, as is the case of the RPC communication [2]" Some research papers focus also on the performance of different communication protocols. [3] experimentally tests the performance of individual protocols (HTTP, AMQP, MQTT, and gRPC) under the same conditions and on the basis of:

- the initial service call,
- the server throughput in terms of the processed requests in a large volume of calls,
- the time taken to transmit one million messages, and
- the throughput in terms of the served memory.

It is shown that the asynchronous communication almost entirely dominates in terms of the number of requests a client can handle and the transmission of a large number of messages. HTTP dominates when large-scale data (>500MB) needs to be sent.

Usually, the asynchronous communication systems refactored to microservices monolith-based systems almost always need an asynchronous microservice communication, at least for the data synchronization. Such approach is used in [4] where a system that has alerts, which defined as communicated events to the system, are stored consistently and notified to interested services. The transactional outbox pattern sends a domain event (a message sent to other microservices about some changes) and atomically updates the database. This proves the asynchronous communication to be the best choice to synchronize the service data and to solve most of the problems in a microservice environment.

[5] provides another example of migrating from a monolith to a microservice. It is migration of the Danske Banks FX Core system. It is shown that a complex and undocumented system has to be divided into multiple microservices. In the monolith system, programmers delegated a received request from external APIs using a RabbitMQ message broker to increase the scalability. Though it works, the issues of maintaining the system and adding new changes, because of the infrastructure of other parts of the system, remain unresolved. After migrating the system to a microservice, RabbitMQ is still used at a higher level (for business services) and for the external API communication. Basically, introducing RabbitMQ in the old system is good and it has not been considerably changed in the new system because the asynchronous communication helps both, the monolith and the microservice architecture.

Using an asynchronous communication doesn't directly imply that it works in a microservice environment. As seen, it helps the monolith systems to scale and use resources more efficiently.

Though the performance is important, the data integrity is also a critical issue in systems with an asynchronous communication. [6] shows an example of a microservice system with its own databases. The problem needing to be solved is the database synchronization of each microservice. The following three solutions are offered:

- Using CRON job to make the database synchronize every N minutes. The issue here is the period between two synchronizations. Synchronizing the database every N minute could be acceptable for some systems in which the data is not updated frequently. Though not an ideal approach, it can be used for systems where changes are rare. The author categorizes the approach as **not suggested**.
- Updating the database of another system from initial one. Though easy to implement, the updating is prone to problems of coupling and transactions. When two systems update the same database, this may lead to the issue of handling errors and failures and it might be difficult to know which system has failed to update the database models, handle concurrent transactions etc. The author categorizes the approach as **not suggested**.
- A **suggested** approach is using a message broker when every database action sends a message with the topic. A microservice receives a message about a data change and finds it either interesting or not. This is why coupling between services is reduced to a minimum to simplify the data synchronization.

When data sourcing over events from external systems is needed, it is important to make sure that the data integrity is protected and always true [6]. The data integrity can be violated when sending data or when wrongly assuming that something is true. For example, when a money deduction request in a service is received, it cannot be saved in database that the money has been deducted from an account if no response is obtained from service that will tell us if the user has enough money. In a synchronous environment, we wait a for a response which is then saved in the database. In an asynchronous environment, the state between sending and receiving the data must be handled.

[7] presents SAGA pattern to update data in multiple services in a microservice architecture using a distributed transaction [7]. So, when there is one failure in one of the microservices, each (transaction) "waiting" for the results of the current transaction will be rollbacked. This makes SAGA pattern important when it comes to maintaining the data integrity. The question is how to maintain the integrity of the sent messages? How can one be sure that a message has been sent from one's service? How can one be sure that no duplicate messages have been received?

Making sure that a message has been sent from one's service is usually resolved using a transactional outbox that resolves a dual write operation issue occurring in a distributed system when a single operation involves both a database write operation and a message or event notification [8]. For a database for which a message should be sent to a message broker, there are two reasons why the outbox pattern should be used:

- If a database call fails, the message will be sent, and the data can be corrupted.
- If a database call is successful but sending message fails, the downstream service will not be aware of the change and the system can enter an inconsistent state.

Based on [8] and AWS environment, transactional outbox can be implemented in the following seven steps:

1. Prepare the data to be inserted in the database.
2. Run the database transaction.
3. Do the business-related database work.
4. Save the message to the outbox table.
5. Commit the database transaction.
6. Use AWS lambda (the background job can be used) to read and process the outbox table.
7. Send the notification to the message queue.

In [9] the transactional outbox pattern is part of a group of patterns that maintains a neutral module/service size, decreases the number of the database sharings between modules/services, and increases the service coupling level. Because of the architectural trade-offs in microservice environments, the outbox pattern is analyzed from that standpoint. A message outbox is injected into the persistence command, based on the atomicity of the database command [9]. In this way the transactional outbox provides flexibility and also introduces an additional complexity by adding an additional logic needed to make it working.

[10] shows an example of the SAGA pattern implementation based on an outbox pattern. Debezium is used to track changes in the database and its sending messages to a message bus. It is important to note that a Debezium outbox event routing component is used to implement and execute the outbox pattern logic.

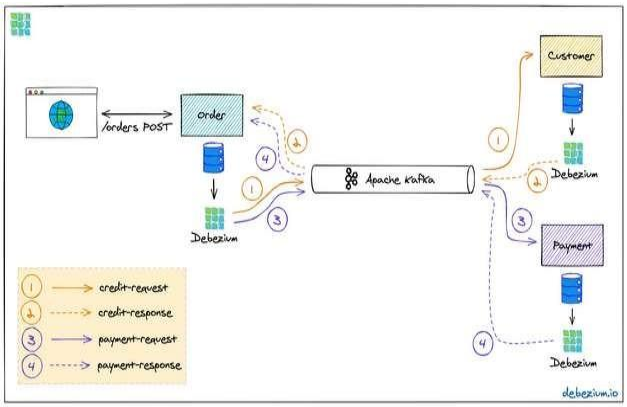

Figure 1. Saga orchestration using the outbox pattern [10]

The high-level perspective of Fig. 1 helps understanding the importance of the outbox pattern in the SAGA pattern. Consider that no outbox pattern is implemented, nor payment-response message has been sent from the Payment service. And no order service approves/declines the order based on sufficient/insufficient funds on the user account. If the user account in the Payment service and the Order service doesn't know about it, we will end up having the order that has been paid but not approved. The conclusion applies when the outbox pattern is a must in almost any case of sending messages, not only in SAGA. This proves that most of the literature rely on the outbox pattern to make sure a message has been sent. An opposite to the outbox pattern is the inbox pattern. It is not often mentioned in research papers, the reason for it is that most of the modern message brokers support at least one delivery, and most of the actions are idempotent. It means that a service will receive at least one message. If it receives multiple messages, it is idempotent and no additional checks are required, even when needed.

[11] presents the inbox pattern as an opposite to the outbox pattern. As soon as a message is received, it is saved to a database and the message broker is notified that the message has been successfully received. Another background process picks the message and processes it. It is important to know, there is no universal solution to any problem, and solutions proposed by researchers are not universally applicable to every problem. The conclusion is that most researchers favourise the asynchronous communication using message brokers whenever possible and when some results can be expected. It is therefore important to analyze both, the system, and the problem to be solved.

Based on the above research, the paper analyzes the asynchronous service communication in terms of its performance and data integrity (detection of duplicate messages using statistical methods).

3 RESEARCH METHODOLOGY

The aim of the paper is to optimize the current solution, based on the existing one for an efficient communication between web services. Its focus is on the migration process from a synchronous to an asynchronous scenario, identification of potential issues during migration, and limitations of the two scenarios without changing the business logic of these services. So, the aim is to provide a fast solution to a non-scalable system. A simulation system is used with no specific business logic for it. Only simple mock methods which can be replaced with a complex logic in a real world example. The results can be applied to any similar situation regardless of tech stack used on the project. So, the focus is not on a code implementation, but improving one process by using another without many implications.

Our hypothesis is that the asynchronous communication gives significantly better results than the synchronous communication, leading to a more efficient and reliable data exchange. It enhances the scalability and performance of the application at specific circumstances.

The basic entropies in the synchronous and asynchronous communication patterns will be presented, and experiments will be conducted to verify that the hypothesis is correct. In the context of the web service communication, the entropy is a measure of the uncertainty or diversity of the data being transmitted or exchanged. Various factors can affect the entropy in this context, contributing to the complexity of the web service communication. Several major entropies often encountered in this area include different data formats, variable data volumes (changing the data quantities), diverse request frequencies, unforeseen requests, response times variety, potential data losses, and security measures and authentication.

The paper analyzes the synchronous and the asynchronous communication in web services to provide programmers with an overview of the challenges that they may encounter during the implementation and help them make decision based on experiments.

4 RESEARCH RESULTS

Before presenting our research results, the technologies and methodologies used for the simulation implementation will be briefly discussed. The results of our testing are not specific for the used technologies and methodologies. They include .NET Core, RabbitMQ, HttpClient, JMeter, and Microservices. The system used for our testing and experimenting addresses a common problem present in almost any major application that requires delivering reports of the performance, work, etc. It simulates a part of the microservice system responsible for solving the issue of generating and sending emails and/or messages to clients. As the application user, we can configure the execution time of our task and other parameters important to us. The process involves the following services as actors: JobEngine, MainService (Core), and ContentService.

The JobEngine sends generated emails to clients and checks the available notifications for execution by calling the MainService (Core). The MainService (Core) acts as an orchestrator in the entire process, deciding whether a specific notification should be sent for the execution and content generation. In a real world, the ContentService receives contents and generates them, while in our simulated system, it simply receives a message or request, simulates the content generation, and responds to the message or request. In our simulations, the communication between the JobEngine and the ContentService is done over the MainService (Core). Why? Let's imagine a case where the MainService (Core) handles the data states and the JobEngine and the ContentService are just helper services that are doing a minimum amount of the logic assigned to them. So, all the data states and updates are done on the MainService (Core) side.

During the experiments, the flow and the steps of the business logic are absolutely the same for the synchronous and the asynchronous communication to obtain results that closely determine which method is more efficient. **Our focus is not on optimizing the steps of the implementation but rather on optimizing the communication pattern.** Therefore, our goal is to compare the communication rather than the implementation. The following parameters are compared:

- the time taken to complete N notifications/tasks (the most impacting parameter),
- the system load during the execution of the background processes.

After completing the experiments, our comparison focuses on the time taken to execute N tasks. This parameter is crucial as it directly indicates the scalability of the system and determines which implementation is better. Imagine that at 2:00 PM there are 100 notifications scheduled for execution. Our goal is that all clients receive their notification by 2:05 PM. Any time beyond 2:10 PM (required time > 10 minutes) is considered inefficient, and clients will be dissatisfied. The system load is measured and compared using various methods and results, such as the average response time, the median and percentile values (90, 95, 99), and the distribution and histogram of the response times.

The following two diagrams show how the experiments are executed for the synchronous and asynchronous communication.

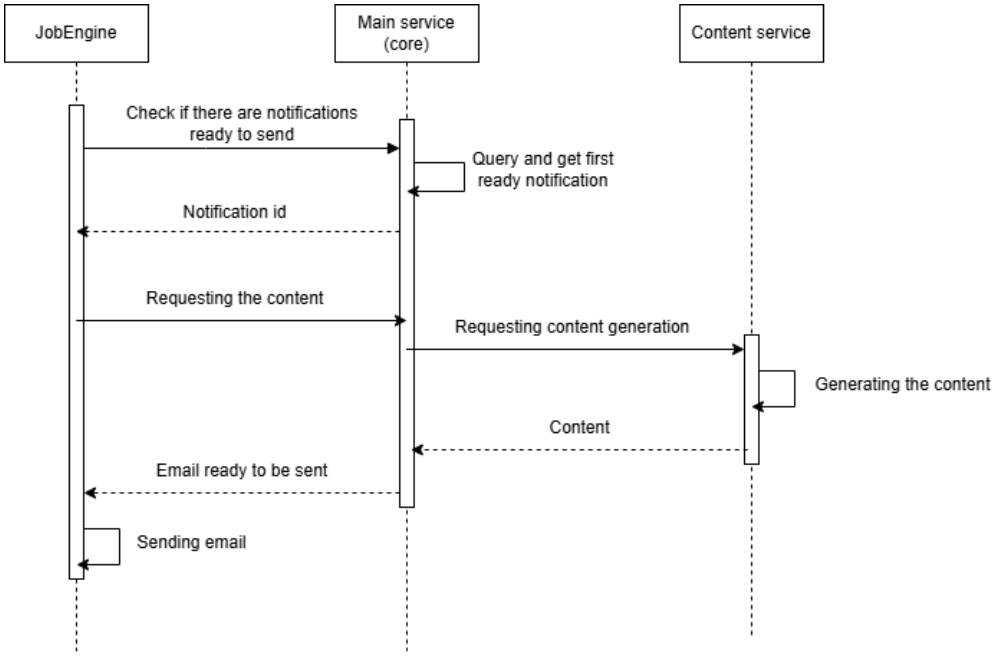

Figure 2. Synchronous process

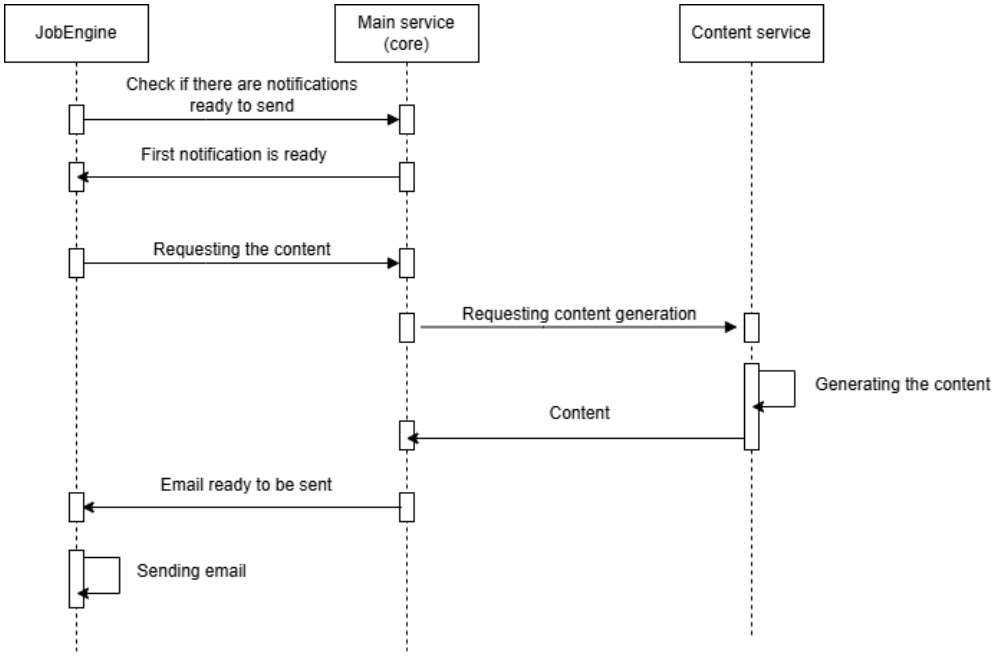

Figure 3. Asynchronous process

Analyzing the above diagrams, shows the first drawback of this communication method. It is the fact that the service, initiator of the entire process, waits for a response from two consecutive services. Only after receiving responses from both services, the MainService (Core) for the status information and notification details sent for the execution, and the ContentService for the content generation, the JobEngine finishes its process.

On the other hand, the problem with the data integrity is not an issue here because each call waits for the next one and does not proceed with its processing until it receives the necessary data. If an error occurs, the service that initiates the call manages a further flow of the program without too much difficulty.

In this approach none of the services waits for a response of their requested processes. This helps the system scale better than the synchronous approach, reduce the time taken for the flow to be finished, but on the other hand, the data integrity is at risk.

4.1 Experiment on the Synchronous Communication Model

The synchronous communication is an excellent choice for scenarios in applications where the scalability and the number of long-running tasks are not too high. The term "too high" is relative in programming and can range from a queue of 100 to 1000 users. This number also depends on the complexity of the tasks executed in the application, and the best way to compare is to use the same number of users in both experiments, which we do throughout our experiments. Users do not affect the business process of the notification generation. Instead, we use it to simulate a parallel system usage while the task generation process is ongoing. Therefore, an additional stress from users making calls to the application provides us with a better understanding of the advantages and disadvantages of a synchronous and/or asynchronous communication.

As to the number of the users, our experiment involves 100 users. They call the system at a one-second interval using the JMeter software, while an automated job in the JobEngine calls the MainService (Core).

Algorithm 1. MainService (Core) – available notifications check

```
1.  [HttpGet("check")]
2.  public async Task<ActionResult<int>>
    GetEventCount()
3.  {
4.  var notificationsToBeExecuted = await _taskDbContext
    .Task .FirstOrDefaultAsync(t => t.Status == "ready for
    execution"); // K-1
5.  if (notificationsToBeExecuted is not null) // K-2
6.  {
7.  notificationsToBeExecuted.Status = "execution";
8.  await _taskDbContext.SaveChangesAsync(); // K-3
9.  return Ok(notificationsToBeExecuted.Id);
10. }
11. return BadRequest();
12. }
```

Algorithm 1 mocks the main logic of getting notifications from the MainService (Core). In our case it's just a simple database query, but in a real world a complex business logic can be implemented. This endpoint is called from the JobEngine at the initial step of the process.
Algorithm 2 shows an example of a mocking notification content by calling the ContentService and returning data to a caller. In a real world, the ContentService generates the content, but here, we are just mocking the flow.

The ContentService and the JobEngine service are too simple to be shown. There is no additional logic except the logic mock. Solving the problems in a synchronous system usually involves constantly calling the service to check the status of the notification to be executed.

Algorithm 2. Mocking the notification content generation

```
1.  [HttpGet("list/{id}")]
2.  public async
    Task<ActionResult<Event>>
    GetEventList(int id)
3.  {
4.  var randomizerEmailBody =
    RandomizerFactory.GetRandomizer(ne
    w FieldOptionsTextWords());
5.  string body =
    randomizerEmailBody.Generate();
6.  var task = await
    _taskDbContext.Task.SingleOrDefaultAsync(t => t.Id
    == id);
7.  HttpResponseMessage response = await
    _httpClient.GetAsync($"{_baseUrl}/event-
    execution/set-execution");
8.  task.Status = "done"; // K-3
9.  await _taskDbContext.SaveChangesAsync();
10. var emailEvent = new Event
11. {
12. TaskId = task.Id,
13. EmailFrom = task.Email,
14. EmailTo = "to" + task.Email,
15. MailBody = body
16. }; // K-4
17. return Ok(emailEvent);
}
```

The JobEngine service doesn't know anything about the state of the notification, so it needs to call the MainService (Core).

In systems with a lower traffic, this approach is quite satisfying and does not pose significant problems. However, as the number of users (in our case, notifications for execution) increases, the efficiency of the implementation decreases. In addition to affecting the flow efficiency, a constant calling of another service also affects the performance of other actions in the system. It is important to note that in a practical example, we focus on communication, while the actual database and external system calls will only be in the MainService (Core). Similar operations in other services will be performed using functions simulating resource loads, while service responses will be fixed.

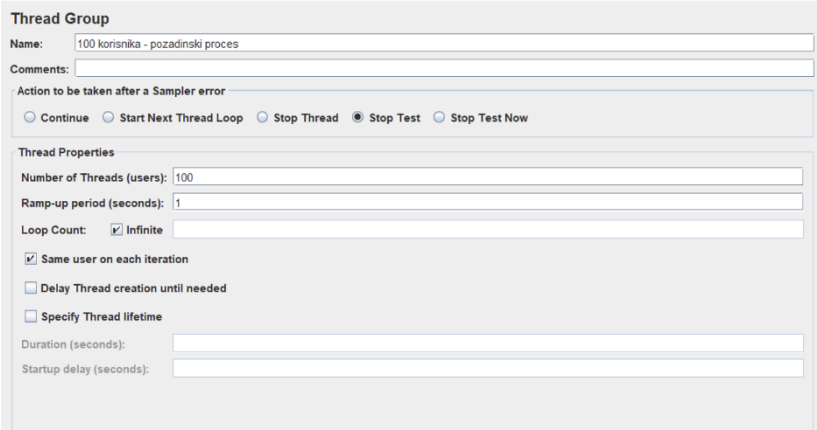

Figure 4. JMeter setup

On the JobEngine side, we set up a background job that calls the MainService (Core) every ten seconds and receives information about notification in a ready to deliver state. After setting up methods that simulate the data generation process and notifications management, it is necessary to set up a method that simulates the process concurrently called by clients. So, in the background, the task process is takes place, while at the same time, the application serves N number of clients. N represents 100 users simultaneously accessing the application. For simulation purposes, we use the JMeter program which tests the system load. The results can be generated and presented also graphically.

After setting up the environment to perform stress testing on our system, a system to monitor the efficiency of the initial notification generation flow is also set up. The goal is to show and prove that in a synchronous mode and under a load, the distribution of the execution time increases, and the overall experiment time is significantly longer.

4.1.1 Experiment Flow and Monitoring

The first task is to start the three services and prepare the database with the tasks. Before starting the experiment, we generate 9468 rows in the database (with the status "done") out of which we randomly prepare 100 for the execution (set the status to "ready for execution"). After preparing the experiment, we launch the application and the system load testing software. The software concurrently simulates 100 users accessing the system until at least one of them receives an error as a response from the system. This is the main reason for stopping the experiment. The experiment starts at 21:30 and ends at 22:02, resulting in a total duration of 32 minutes. This time represents the time it takes for all 100 background tasks to be successfully completed in a synchronous manner.

Since the experiment takes 32 minutes, and we simultaneously have 100 users accessing the application, we have many samples to observe. The table shows that the average request execution time is moderately high, and the maximum request is high, too. The percentiles show that there are spikes and drops in the requests, indicating a potential instability. The system response time is uniform throughout the entire experiment, with two significant spikes. The reason for this could be a potential temporary lack of the memory, threads, connection interruption to the database etc. The total duration of this type of the system behavior change is about two minutes. This can be quite satisfactory in systems that do not require an excessive tolerance. The issue with the disruption is not that it occurs but that the response times increase dramatically at that moment, causing frustration for users.

To determine whether those two requests are just exceptions to the rule, we will examine the percentile distribution graph in the following figure. The percentiles represented on the graph are the 99th percentile (in red), the 95th percentile (in green), and the 90th percentile (in blue). The last two percentiles are less visible and are expressed in the last minute of the execution. The highest value belongs to the red percentile, followed by the 95th and the 90th percentiles. The red percentile demonstrates that the spikes in the execution times are exceptions (1% of responses). We can also observe that in the last minute, there are a few responses with longer times, but in a smaller ratio. These spikes in the 95th and 90th percentiles as exceptions to the rule, but they should not be ignored.

Table 1. Summarized Experiment Results

Requests	Executions			Response Times (ms)							Throughput	Network (kB/s)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	36591	30	0.08%	5222.12	179	1E+06	1188.00	5007.00	6399.80	28193.24	19.12	1.72	2.37
HTTP Request	36591	30	0.08%	5222.12	179	1E+06	1188.00	5007.00	6399.80	28193.24	19.12	1.72	2.37

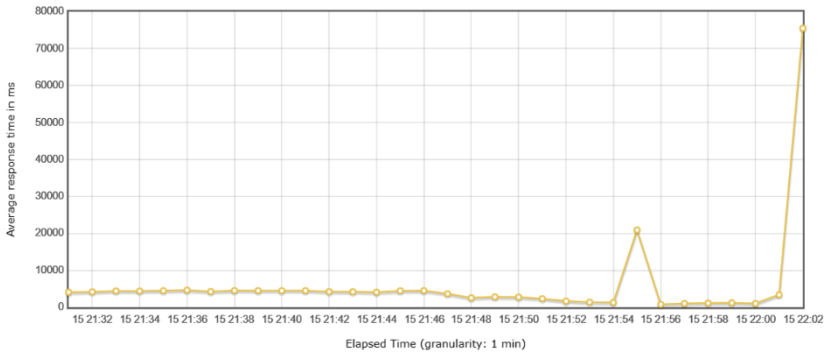

Figure 5. Average response times during the experiment duration

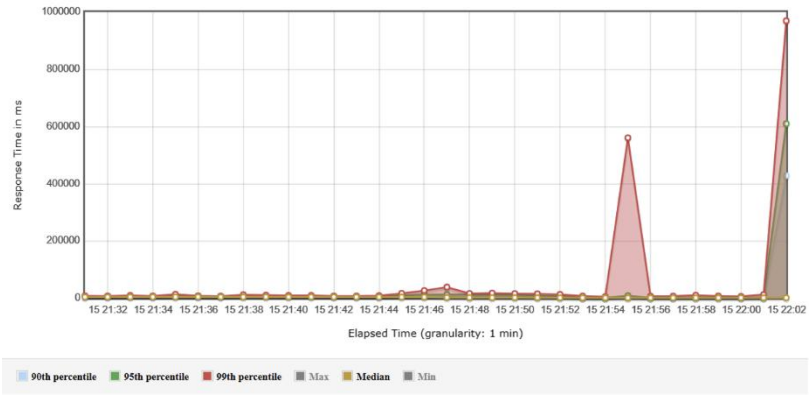

Figure 6. Median and Percentile Times of Synchronous Communication

4.2 Experiment using an Asynchronous Communication Model

The asynchronous communication is used in scenarios where the scalability is important, and long-lasting actions pose a threat to the application performance. In our case, the asynchronous method enables a parallel execution of 100 tasks, delivering them to clients in a significantly shorter time compared to the synchronous execution. The method allows free threads to execute background processes independently of the main processes initiated by users in the system. On the other hand, we must be prepared for an impact on the execution times (response times) by main application processes, as the asynchronous communication will definitely cause a higher system load during its execution. As to the number of the users, the experiments are conducted with a set of 100 concurrent users. Each user calls the system at intervals of one second using the JMeter software, at a parallel monitoring of experiment results that are visually presented and used for the analysis.

In our asynchronous approach we use message broker-based technologies. They enable systems to communicate with each other without waiting for a response from the other system, thereby reducing its blocking. So, each call is done over a RabbitMQ message broker, there are no HTTP calls here. In other words, the communication occurs based on the current capabilities of the system to handle the request from the other side. On the side of the JobEngine, background job or a simple infinite loop that runs every n seconds is set up. The ContentService has only one method that simulates the notification content generation, lasting for ten seconds. Therefore, rendering each notification takes a minimum of ten seconds using the simulation method for the same process.

Algorithm 3. Simulation of rendering a task on the "Content service" side

```
1. consumer.Received += async (model, eventArgs) =>
2. {
3. var body = eventArgs.Body.ToArray();
4. var message = Encoding.UTF8.GetString(body);
5. var taskRenderingResult =
   JsonConvert.DeserializeObject<int>(message);
6. await EventController.HeavyLoadOperation(10, 2);
7. channel.BasicPublish( exchange: "",
   routingKey: "email_task-ready-to-render_finished",
   body:
   Encoding.UTF8.GetBytes(taskRenderingResult.ToStrin
   g()) );
8. };
```

Algorithm 4. JobEngine caller simulation

```
1. while (true)
2. {
3. Channel.BasicPublish(exchange: "",
   routingKey: "email_check_task_id"); // K-1
4. Thread.Sleep(TimeSpan.FromSeconds(1)); // K-2
5. }
```

Algorithms 3 and 4 show simulations of the basic published messages in the async communication. Compared to the synchronous model, the asynchronous model in a real world keeps the business logic and only replaces the communication pattern. So, we try to improve the scalability of the system only by replacing its way of communication. It should be noted that there are different kinds of the possible optimization, but in our paper analysis only one is shown.

4.2.1 Experiment Flow and Monitoring

The experiment started at 18:40 and finished at 18:43, giving us the total time taken to complete the 100 tasks in impressive three minutes. During this time, 100 users accessed the system concurrently, creating a considerable stress on the system, on external systems used in the experiment, and on those used outside the experiment (database, mail provider, notification content generator, etc.)

The distribution of the percentiles and the average execution time is not mutually aligned, meaning none of the percentiles corresponds to the projection of the average execution time. This means that the 99th percentile represents requests that occurred as exceptions, basically outside the average. The average execution time is significantly higher than the medians, indicating that the majority of the requests (50%) are faster than the average. The reason for the disrupted average lies in the high percentiles with high values. As seen, the 95th percentile is considerably higher than the medians, indicating a significant number of requests, or a smaller number of requests with long times, disrupting the statistics. Consequently, users may experience occasional slow execution times, even if the average time is relatively low. Initially, the projection of the average and medians is uniform, but as time passed, the median remained stable while the average increased. This tells us that the system becomes more stressed as time goes on.

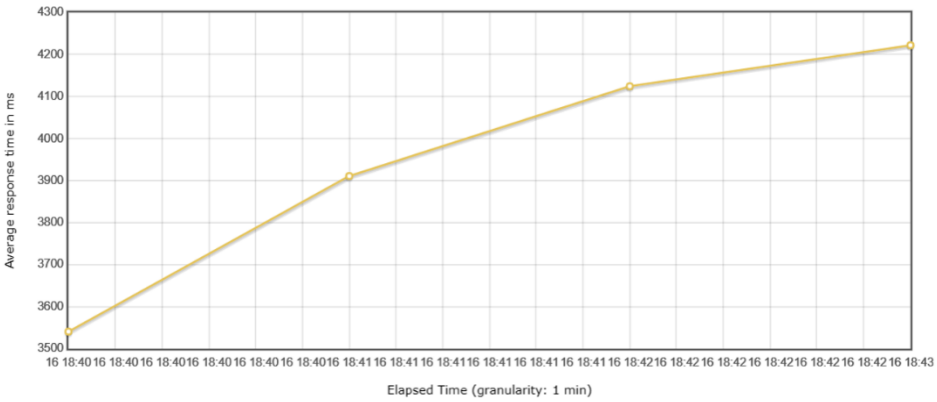

Figure 7. Average response times during the experiment

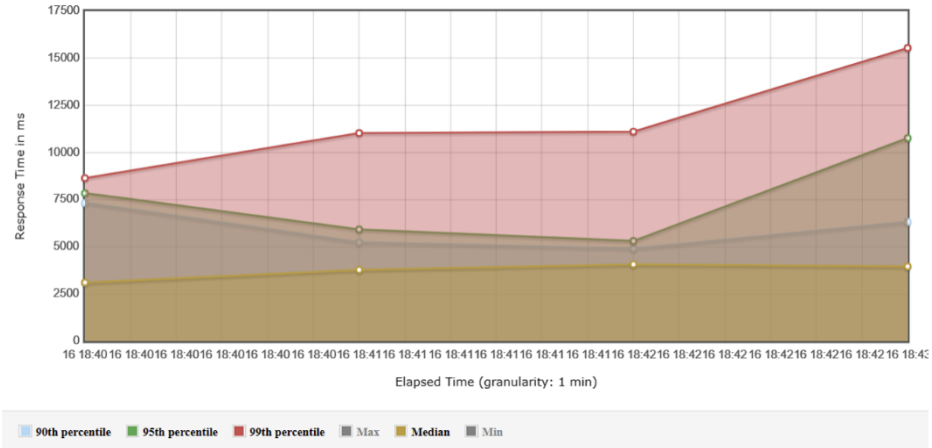

Figure 8. Median and percentile values during the experiment

4.3 Bayesian Analysis

As mentioned above, the data integrity in asynchronous systems is one of the main challenges those systems are facing. Compromising the data integrity in asynchronous systems can occur due to the receiving of duplicate messages, leading to a multiple processing of a single message. To assess the likelihood of receiving a duplicate message, we use the Bayesian statistical methods. In our case, a success means that the received message is not a duplicate, while a failure indicates that we have received a duplicate message.

The Bayesian analysis begins by defining the prior values using the beta distribution. The reason for using the beta distribution lies in its mathematical foundation and its ability to represent the probability distribution within the interval [0, 1]. Therefore, if we have experiments with two possible outcomes (success/failure), the beta distribution can be used for the analysis [12].

In our asynchronous task processing example, we can assume that the system, upon receiving messages, will receive approximately two duplicate messages out of 100 sent messages. Although, by definition, the values of alpha and beta for the beta distribution can be arbitrary, we will set them to 2 and 98. Therefore, the beta distribution has the following form: Beta (2, 98).

Our event refers to receiving a duplicate message, and accordingly, we form a beta distribution. As already known, the beta distribution has two parameters, alpha and beta, which control the shape of the distribution. The alpha parameter controls the probability of the success of our event (receiving a duplicate message), and the beta parameter represents the probability a of failure (not receiving a duplicate message).

After defining the prior values and conducting the experiment, the likelihood function is defined. Let k be the number of the messages in the experiment (in our case 100). According to the assumption (beta distribution) defined at the beginning of the experiment, the binomial distribution is:

$$P\,(y\,|\theta) = \binom{100}{y} * \theta^{y} * (1-\theta)^{100-y} \qquad (1)$$

The binomial distribution is a good way to model the number of successes (the number of duplicate messages) in the fixed number of trials. By combining it with the prior distribution, a posterior distribution and is obtained our assumption is compared with the actual results.

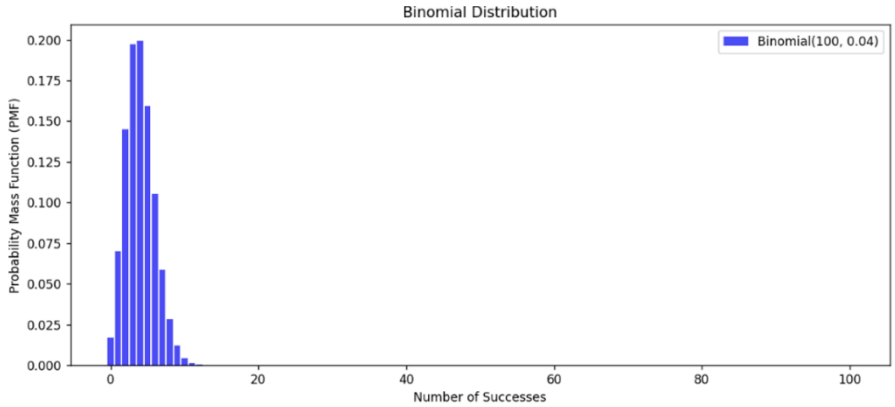

Figure 9. Binomial Distribution

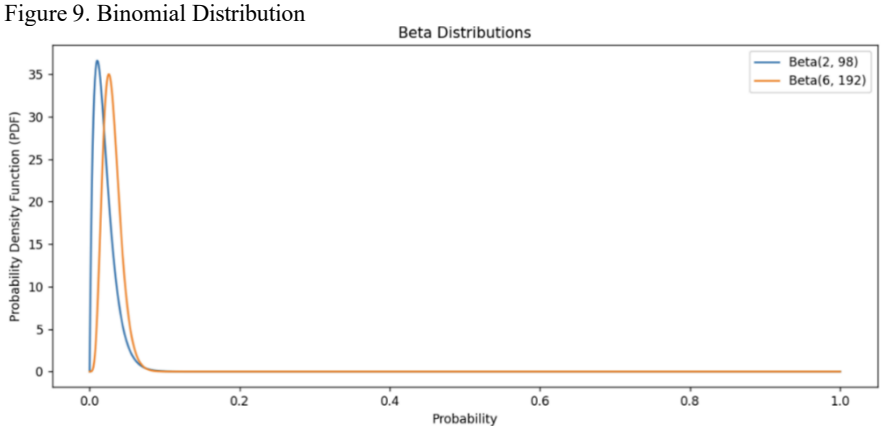

Figure 10. Beta Distribution Posterior Value

The graph shows a binomial distribution that tells us the likelihood of n duplicated messages. That the distribution is the highest around five duplicated messages out of the 100 received. The distribution decreases towards 0 and 10, indicating that the number of the possible duplicated messages ranges from 0 to 10, with the highest probability around the number of 5.

In the final step of the Bayesian analysis the posterior values are calculated. They show that our assumption differs from the experimental results. The posterior value is obtained using a conjugate prior:

$$Beta\ (p + \alpha, n - p + \beta) = Beta\ (6, 192) \quad (2)$$

The posterior distribution closely resembles the initial definition, and the probability of recurring messages complies with the assumption. The posterior distribution is slightly narrower than the prior distribution, indicating that we are slightly more confident about the number of repeating messages in this step. The previous analysis and its results show that the mechanisms to avoid duplicated messages should be implemented. In an asynchronously operated system, there is no guarantee only one message will be received because most message brokers use the "at least once delivery" policy, which in some cases may result in delivering a message after once or several times.

5 DISCUSSION

The paper presents important experiments to compare the synchronous and the asynchronous communication in order to understand the strengths and weaknesses. It is important to note that the system background process load time in asynchronous mode takes only three minutes and in the synchronous mode 32 minutes. This could be due to the results of the average response times since the number of the requests in the synchronous mode is 52,737, whereas in asynchronous mode, it is only 3,770. The conclusion can be that a larger number of samples could have a greater impact on the result.

To confirm assumption, images 5 and 6 in the analysis of the asynchronous communication are compared. The graph shows that the request median remains relatively stable while the server response average increases. Thus, the average response increases over time while the median remains the same, indicating that the system becomes more overloaded over time, and the number of the requests above the median increases. There is no such behavior observed in the synchronous communication of the same volume as in asynchronous communication. Figures Images 3 and 4 showing the synchronous communication results, the average execution time is quite uniform, except for deviations occurring in a few cases.

In the percentile graph, the deviation from average results is due to a 1% increase in the average time, which reached high values at that moment and this disrupts the overall average. This might be due to randomized delays during the execution. Therefore, we can assume that the average execution time is significantly lower in the asynchronous communication, because the system load is higher but occurs over a shorter period of time, not resulting in excessive implications for users utilizing the system at that moment.

Following the above, the asynchronous communication gives better results in terms of reducing the system load, even though the number of messages processed per time unit is significantly higher than the number of requests in the synchronous communication. However, the difference is negligible when comparing the benefits gained in terms of the task execution. The strength of asynchronous communication is not in improving the system performance. On the contrary, it may even reduce it in some cases. Its strength is in the increased scalability as demonstrated in our study. The time taken to execute 100 tasks in synchronous mode takes only 32 minutes, while in the asynchronous mode for the same scenario, it takes around three minutes. The difference of 29 minutes makes approximately 90.6% more efficient than before, and our clients receive data emails within three minutes from the time the task is scheduled.

We explore a case of receiving duplicate messages. It is an often concern and proves that, if developers are not familiar with all aspects of the asynchronous communication, they can cause problems that are difficult to detect and can result in significant losses. Using the Bayesian analysis, we prove that our prediction is accurate and, by implementing protection mechanisms, such as the "Outbox pattern" and "Inbox pattern," the system is protected. This type of the issue does not occur in the synchronous communication because the data is updated in a predefined order and the strong consistency is synonym for this scenario.

6 CONCLUSION

Based on the experiment results of background tasks execution, we can conclude that scalability is the primary factor affecting the choice between the synchronous and asynchronous communication. However, for a well-organized asynchronous environment there are besides this factor several others, such as team coordination, eventual consistency, message delivery assurance, changes in the business logic, and using an appropriate broker system. As noted, the implementation of synchronous methods is simpler and often used by programmers. Therefore, team coordination is recommended through practices such as rigorous code reviews via pull requests, comprehensive documentation, workshops, and brief training sessions.

The use of an eventual consistency is one of the main features of the systems based on the asynchronous communication which almost always involves dealing with it. It is a very complex process to implement and requires a complete understanding of the system and asynchronous communication concepts. The Bayesian

analysis of duplicate system messages shows that it is mandatory to implement mechanisms to manage duplicate messages. The same applies to mechanisms ensuring the message delivery. They ensure that each sender sends at least one message, and the receiver receives and completes the message processing. Before introducing asynchronous communication, it is necessary to determine how it should behave in an asynchronous environment. Asynchronous processes can introduce additional complexities, especially in scenarios where an immediate response is required. Therefore, in the process of migrating from a synchronous to an asynchronous systems, we can start with individual cases. The message broker selection must be carefully analyzed in term of its strengths and weaknesses. This hypothesis is that asynchronous communication provides much better results than synchronous communication, thus resulting in a more efficient and reliable data exchange, improving the scalability and performance of the application in certain circumstances. These research results confirm our hypothesis. Performance is improved by 90.6%. Regarding application performance, the medians and average execution times are improved, too. The overall system stress was reduced. So, using asynchronous communication reduces system stress and improves average performance, thereby minimizing user frustration. This is achieved by simply replacing the communication pattern without changing the business logic which is very similar to the experience from [5].

The possibility of compromising the data integrity and threats is evident. We prove the existence of the duplicate messages using the Bayesian method, we show the possibility of duplicate messages is higher than assumed. The asynchronous communication is definitely more efficient, while the reliability of the data exchange depends on the implementation of mechanisms to protect it. Synchronized solutions provide a better control over the data in terms of managing the risk of data loss and/or creating inconsistency in the system.

Future research will focus on the data consistency management and the mechanisms through which it can be managed. Testing asynchronous systems requires complex levels of abstraction, and research in this area can contribute to testers who are encountering systems based on asynchronous communication for the first time.

REFERENCES

[1] J. Thönes, "Microservices" in IEEE Software, vol. 32, no. 1, pp. 116-116, Jan.-Feb., 2015, https://doi.org/10.1109/MS.2015.11

[2] G. Baptista and F. Abbruzzese, Software Architecture with C# 10 and .NET 6, Packt Publishing Ltd., 2022.

[3] I. V. Zakutynskyi and I. E. Rabodzey, Microservice Communication for IoT-based Systems. Architecture Review and Performance Test, Telecommunications and radio engineering, Vol. 4 No. 74, 2022. https://doi.org/10.18372/1990-5548.74.17311

[4] R. Laigner et al., From a Monolithic Big Data System to a Microservices Event-Driven Architecture, 2020. https://doi.org/10.1109/SEAA51224.2020.00045

[5] A. Bucchiarone et al., From Monolithic to Microservices: An experience report, 2017. https://doi.org/10.13140/RG.2.2.34717.00482

[6] A. Smid et al., Case study on data communication in microservice architecture, 2019. https://dl.acm.org/doi/10.1145/3338840.3355659

[7] E. Daraghmi et al., Enhancing Saga Pattern for Distributed Transactions within a Microservices Architecture, 2022. https://doi.org/10.3390/app12126242

[8] Transactional outbox pattern, Available: https://docs.aws.amazon.com/prescriptive- guidance/latest/cloud-design-patterns/transactional- outbox.html (accessed by January, 04, 2024).

[9] T. Rosa et al., Guerra A Method for Architectural Trade- off Analysis Based on Patterns: Evaluating Microservices Structural Attributes, 2020. https://doi.org/10.1145/3424771.3424809

[10] G. Morling and T. Betts, Saga Orchestration for Microservices Using the Outbox Pattern, 2021. https://www.infoq.com/articles/saga-orchestration- outbox/ (accessed by January, 18, 2024)

[11] K. Atłasik, Microservices 101: Transactional Outbox and Inbox, 2022. https://softwaremill.com/microservices-101/ (accessed by January, 18, 2024)

[12] M. Taboga, Beta distribution, https://www.statlect.com/probability-distributions/beta-distribution#:~:text=The%20Beta%20distribution%20can%20be,success%2C%20with%20probability (accessed by January, 18, 2024)

Ahmed Mujić is a software engineer at Isatis Software Solutions, Sarajevo, Bosnia and Herzegovina. He is also an assistant at the University of Zenica. His research focuses on software engineering. He currently works on a weather application project that helps clients schedule and organize their work. Extensively works with .NET Core, Angular, MSSQL, and xUnit as part of the core technology stack on the project.

Nevzudin Buzađija is an Associate Professor at the Department of Software Engineering, Polytechnical Faculty, University of Zenica, Bosnia and Herzegovina. He is the Vice Dean and Head of the Department of Software Engineering. He has written four books and has published several papers in the field of software engineering.

Samir Lemeš is a Full Time Professor at the Department of Software Engineering, Polytechnical Faculty, University of Zenica, Bosnia and Herzegovina. He is a Dean and Professor at the Polytechnical Faculty. His fields of interest are computer graphics, computer-aided design (CAD), finite element analysis, and mechanical measurements. He received his Ph.D. degree as an educated professional with a PhD from the University of Ljubljana, Slovenia in mechanical engineering.

Denis Čeke is an Associate Professor at the Department of Software Engineering, Polytechnical Faculty, University of Zenica, Bosnia and Herzegovina. His research interests are in Machine Learning, Internet of Things and very large databases.