Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 159 (2019) 1035–1044

Procedia Computer Science

www.elsevier.com/locate/procedia

23rd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems

# Dynamic Microservices to Create Scalable and Fault Tolerance Architecture

Mihai Baboi[a], Adrian Iftene[a,]*, Daniela Gîfu[a,b,c]†

[a]Faculty of Computer Science, "Alexandru Ioan Cuza" University, General Berthelot, 16, 700483, Iasi, Romania
[b]Institute of Computer Science, Romanian Academy - Iasi branch, Bulevardul Carol I, 8, 700505, Romania
[c]Cognos Business Consulting S.R.L., 7, Iuliu Maniu Blvd, 061072, Bucharest, Romania

**Abstract**

One of the industry's most important trends in enterprise architecture is related to the use of microservices, to the detriment of monolithic architectures, which are beginning to no longer be used. Due the cloud-native architectures the deployment of microservices systems is more productive, flexible and cost effective. Anyway, a lot of companies started to migrate from one type of architecture to another, but it is still in the early phase. In this paper we address the challenges raised by the need to develop a scalable and fault tolerance system based on microservices. In our experiments we consider two types of microservices, simple and extended and the proposed solution proves to be an innovative one especially based on its dynamic behavior.

*Keywords:* microservices; dynamic behavior; scalability; fault tolerance

## 1. Introduction

The history of programming languages and computer science paradigms has been characterized over the past decades by an increased attention to distribution and modularization to enhance reuse and robustness of the code.

* Corresponding author. Tel.: +40 232 201091.
*E-mail address:* adiftene@info.uaic.ro
† Corresponding author. Tel.: +40 232 201771.
*E-mail address:* daniela.gifu@info.uaic.ro

There was a need to increase the quantity and quality of the software [1]. One of the key factors in clarifying some misunderstandings related to innovative engineering is how to adequate different tools in order to design and to develop better software systems [2]. The main success in this process was recently demonstrated by systems based on microservices [3] being an architectural paradigm oriented on different applications (e.g. persons with disabilities) [3]. Under the term microservice, there is an increasing interest in architecture and design. Quality attributes (e.g. scalability, performance, and error tolerance) or choice of models such as "contractual de service" [5] or API Gateway no longer violate the YAGNI principle ("*You aren't gonna need it*"), suffering a BDUF error type ("*Big Design Up Front*"). The main research question this paper intends to answer *how we could develop a system based on microservices as easy as developing a monolithic system*? Starting from this concern, *how it could be created an environment that allows for dynamic distribution of computing power between clients*? Our research hypothesis proposes the use of the architecture of a server-client system that combines distributed computing and microservices for solving these challenges.

The paper is structured as follow: Section 2 presents a short overview of the current literature in clarifying the importance of the microservices, including two known services offered by Azure, while Section 3 discusses the proposed architecture. Section 4 discusses the evaluation of this system before drawing some conclusions in the last section.

## 2. Literature Review on Microservices

Due the cloud-native architectures the deployment of microservices systems is more productive, flexible and cost effective [6]. Still, Zimmermann observes that microservices are a sensitive topic investigated especially by academia [7] and industry. For the first time, microservices term was discussed at a workshop of Software Architects in Italy on May, 2011 in order to describe what the participants saw as a common architectural style recently explored by many of them. One year later, the same group confirmed that microservices term is the right name. In fact, the microservices have been developed as a response to the problems in monolithic applications or service-oriented architectures that have put in difficulty the part of scalability, complexity part, and on the part of dependencies of the application under construction, all using lightweight communication mechanisms [8-9]. Since the monolith is a software application whose modules cannot be executed in an independent manner, the solution based on microservices must be regarded as the only one capable of executing independent instructions from one another [10-11]. These large monoliths become in time difficult to maintain and they evaluate with difficult due to their complexity, but a major disadvantage is that they limit the scalability of the product. Another problem comes from the fact that it does not provide fault resistance, and there is no possibility that a system component to work while another component does not work, which is possible with the microservice-oriented architectures.

In SOA (Service Oriented Architecture) the main services are coordinated using two methods: orchestration (where there is a central microservice that will send requests to other services and supervise the whole process by sending and receiving responses) and choreography (which does not suppose any centralization, but each service knows in advance what it has to do) [1]. As with monolithic architectures and SOA architectures, the most difficult issue remains the decomposition of the system into services [12]. Also, at all not to be neglected the issue of providing confidential information through uncontrolled propagation of the services [13].

Our architecture combines distributed computing with microservices to create an environment that allows dynamic computing distribution between clients. By distributed computing, we understand the availability of processing and storing of large amounts of data in the cloud, a key element in today's industry, inside and outside the IT area. Distributed storage systems are designed to meet the requirements of distributed and computationally extended applications with wide applicability, scalability and high performance. A well-known solution is MapReduce [14]. A well-known solution is MapReduce that orchestrates the processing by sorting distributed servers, by managing the different tasks in parallel, all communications and data transfers between parts of the system providing redundancy and fault tolerance.

Another programming model used to run efficiently the computerized applications in parallel or large-scale, without manual configuration or infrastructure management, with more powerful high performance computing (HPC) clusters become available in Azure Batch [15]. To illustrate these ideas, we recall SaaS (Software as a service) or client applications where it is necessary widely execution [16]. In fact, different ITC companies are exhibiting an increased interest in SaaS, being attractive in reducing their operational costs and in consequence increasing their business agility [17]. Another service offered by major cloud service providers is Azure Functions which allow on-demand run without the infrastructure having to be explicitly provided or managed [18].

It also gives increased interest to the applications to easily run small pieces of code or "functions" in cloud. The growing interest for Internet-of-Things (IoT) makes that Azure Functions [19] to become an excellent solution for data processing, system integration and the construction of simple APIs and microservices.

## 3. Methodology

Structural, the proposed system could be split in 3 different areas: (1) the client - the one who will execute tasks assigned by the server; (2) server - the interface with the client, the brain of the monoliths application (3) the client-server relationship management area that encapsulates all the details connected by the transfer of the server-to-client execution. All information sent over the network between client and server is encrypted using the DES (Data Encryption Standard) algorithm and the key is changed through the Diffie-Hellman protocol [20] which, although is vulnerable under certain conditions, it is still implemented in various Internet security solutions.

### *3.1. System Architecture*

Our system respects to a large extent based on the architecture of a dynamic system microservices. The architecture is based on a server-client type in which a server corresponds to more clients. Both the server and the client run web microservices, the communication protocol being HTTP, the data format being in JSON. This architecture is useful in distributing and dynamically redistributing resources between clients. Such an architectural model is used to build large, complex and scalable applications on horizontal consisting of small, independent and decoupled processes communicating with each other using application programming interfaces (API) [21].

In Fig. 1 it is describes how the server distributes packages with functionalities to its clients. Depending on the number of clients, there may be instructions that are not assigned to any client or the same set of instructions assigned to multiple clients.

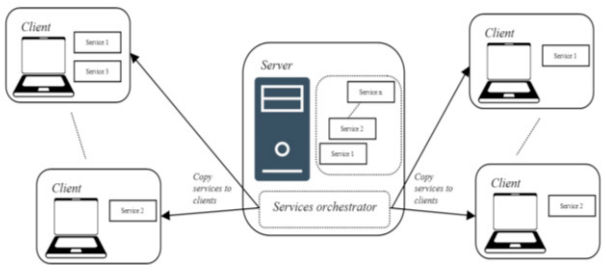


Fig. 1. Distributing services to clients.

The application architecture was built using the ASP.NET MVC framework from Microsoft. In the central part, we can see server-side microservices on the server, and in the left and right, more clients that are waiting to run tasks from the server. Orchestrator service component ensures on the one hand the communication between the server and the clients, sending tasks from the server to the clients, and on the other hand it monitors the status of these requests.

This architecture offers the possibility that a microservice to call another microservice (obtaining an extended microservice) or to call each other, which can lead to a circular dependence, which should be prevented at the user level. The server-client communication protocol is carried out in the following stages:

1) The client connects to the server and initiates the key exchange protocol. He will also give to the server and to the port they will correspond to.
2) The server notifies the client with the following task to execute (a task is represented by a pair (microservice, input data)).

3) The client receives the task and then notifies the server that the transmission and uploading of it has succeeded or failed successfully.
4) Once the connection between the two entities is established, the server sends data in JSON format, encrypted with DES to the client for processing.
5) The client receives the data and returns the response (the output data obtained from the running microservice on the input data) also in JSON format encrypted with DES.
6) To every minute the client notifies the server that it is available or not to receive new tasks.
7) The server can initialize any time the procedure to reset the client, in which case it will reset itself and will be able to accept new tasks.

A special case of this communication is the scenario where a client is running a task needs the response of another client to complete the execution. In this case, two existing possibilities were assessed: orchestration and choreography.

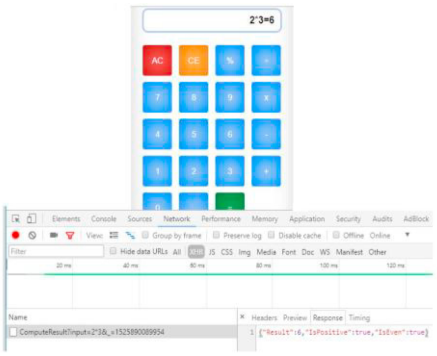
In the case of choreography, we identified several impediments: (a) the list of clients available for executing the external task had to be sent by the server to the client, and keeping this list of updated values would put pressure on the network by frequently exchanging information; (b) the connection between the two clients was vulnerable to attacks. The two situations were solved by using orchestration. Basically, the server has all the problems to manage, the clients being just simple entities that are easy to handle.

For the extended microservice side, the client-to-client communication phases are:

1) The client initializes a call to the server requesting the result for a particular task. He generates a unique password to be used once. This password is encrypted with all the data package sent using DES algorithm.
2) The server receives the call, checks if there is a client available. If there exists, he is calling back, if it does not, it will not return anything, and the task will be retrieved for execution by the initiating client of the call. It receives, besides the package with direct instructions, additional packages (dependencies, etc.).
3) If the server interrogates a client for an extended microservice, it will also receive the unique disposable password (used as encryption-encryption optimization) created by the original client, which can encrypt the result with this key.
4) Once the result is received, the server will only redirect it to the first client.
5) The client decrypts the result with the disposable password and continues the execution.

### 3.2. Application

For testing and evaluating this architecture, we implemented several microservices that we called on what we wanted to check at one time.


2*3=6
AC
CE
%
7
8
9
x
4
5
6
-
1
2
3
+
0
=
Elements Console Sources Network Performance Memory Application Security Audits AdBlock
View: Group by frame Preserve log Disable cache Offline Online
Filter Hide data URLs All XHR JS CSS Img Media Font Doc WS Manifest Other
20 ms 40 ms 60 ms 80 ms 100 ms 120 ms
Name
Headers Preview Response Timing

```
ComputeResult?input=2*3&_=1525890089954
```

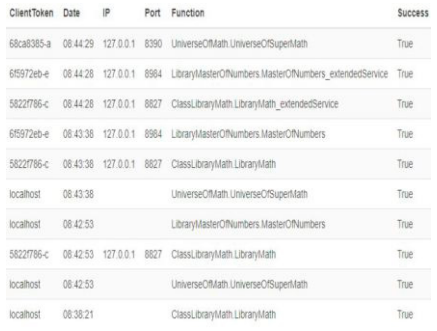
```
1 {"Result":6,"IsPositive":true,"IsEven":true}
```

Fig. 2. The interface.

In the first set of experiments we used 3 microservices as follows: (1) a microservice that performs a mathematical operation between two numbers (using *LibraryMath*), (2) a microservice that tells us if a number is prim-positive (*MasterOfNumbers*) and (3) an extended microservice that will call the first microservice when it receives two numbers, the result will be sending it to the second microservice to find out information about this number (*UniverseOfSuperMath*).

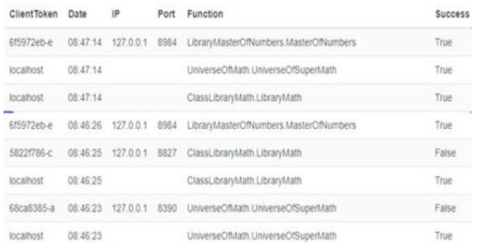
Figure 2 shows how the mathematical computation is achieved through the micro services presented. At the interface level, only the result of the mathematical operation is displayed, the rest of the information can be seen in the result received from the server after the AJAX call launched by pressing the equal key (the result is positive and the result is positive).

Next, we will look at a basic functionality of the application that is centered on *what happens when there are one, two or more connected clients*? In Figure 3, we can see how in the experiments we did we started more clients on the local computer, using different ports for each of them.



| ClientToken | Date | IP | Port | Function | Success |
|---|---|---|---|---|---|
| 68ca8385-a | 08:44:29 | 127.0.0.1 | 8390 | UniverseOfMath.UniverseOfSuperMath | True |
| 6f5972eb-e | 08:44:28 | 127.0.0.1 | 8984 | LibraryMasterOfNumbers.MasterOfNumbers_extendedService | True |
| 5822f786-c | 08:44:28 | 127.0.0.1 | 8827 | ClassLibraryMath.LibraryMath_extendedService | True |
| 6f5972eb-e | 08:43:38 | 127.0.0.1 | 8984 | LibraryMasterOfNumbers.MasterOfNumbers | True |
| 5822f786-c | 08:43:38 | 127.0.0.1 | 8827 | ClassLibraryMath.LibraryMath | True |
| localhost | 08:43:38 | | | UniverseOfMath.UniverseOfSuperMath | True |
| localhost | 08:42:53 | | | LibraryMasterOfNumbers.MasterOfNumbers | True |
| 5822f786-c | 08:42:53 | 127.0.0.1 | 8827 | ClassLibraryMath.LibraryMath | True |
| localhost | 08:42:53 | | | UniverseOfMath.UniverseOfSuperMath | True |
| localhost | 08:38:21 | | | ClassLibraryMath.LibraryMath | True |

a)



| ClientToken | Date | IP | Port | Function | Success |
|---|---|---|---|---|---|
| 6f5972eb-e | 08:47:14 | 127.0.0.1 | 8984 | LibraryMasterOfNumbers.MasterOfNumbers | True |
| localhost | 08:47:14 | | | UniverseOfMath.UniverseOfSuperMath | True |
| localhost | 08:47:14 | | | ClassLibraryMath.LibraryMath | True |
| 6f5972eb-e | 08:46:26 | 127.0.0.1 | 8984 | LibraryMasterOfNumbers.MasterOfNumbers | True |
| 5822f786-c | 08:46:25 | 127.0.0.1 | 8827 | ClassLibraryMath.LibraryMath | False |
| localhost | 08:46:25 | | | ClassLibraryMath.LibraryMath | True |
| 68ca8385-a | 08:46:23 | 127.0.0.1 | 8390 | UniverseOfMath.UniverseOfSuperMath | False |
| localhost | 08:46:23 | | | UniverseOfMath.UniverseOfSuperMath | True |

b)

Fig. 3. The interface.

We have 6 fields: *ClientToken* – a unique token associated to each client (when the call is local and it has the localhost value); *Date* – the moment when the request was made; *IP & Port* = the client's IP & the port through which communication is made; *Function* – the name of the called function; *Success* – a true or false message if the call was or not finalized with success. For example, we notice that at the first call (h: 8:38:21, no client is connected to the server, the process being executed by the server). At the second call, we notice the dynamic behavior of the system, one of the tasks being executed by one of the clients, while the other two are executed by the server. Concretely, *UniverserOfSuperMath* is called (local - there is no client available for this task) which in turn calls two other microservices, one local, and one via the client delegated to use the specific instruction, etc.

***Fault tolerance***

Another functionality I took into account when I built this architecture was related to the system's fault tolerance. Starting from the previous scenario, we can see what happens if one or more clients want to leave the system.

In Figure 3 on the right, the call at 8:46 AM shows this scenario. Clients on ports 8390 and 8827 have a local or network problem, or simply shut down the connection to the server, and the server is not notified in time to remove them from the list. The server will try to contact clients and launch commands, but if they do not respond in a timely manner, the server takes over their tasks and returns the requested result. The clients are asked once again for verification for a while, and if they continue not to respond, they will be removed from the list of available clients. The next call (8:47 AM) will no longer needlessly query clients that are no longer available, and the tasks that are missed by the available clients will be executed by the server.

***Advantages / disadvantages of the offered solution***

The benefits of this architecture are obvious: low hosting costs, microservices offered in distributed network, being dynamic and auto scalable (when clients also offer computational power as their number grows, the computational power of the system increases).

Equally, the limitations must be emphasized: when the computation power curve does not follow that of clients. We also have a restriction on the ability to run this application on any operating system. For this we decided to transform the available solution in .NET into Java. But this solution has some drawbacks to the initial solution (Java offers lower data processing speed and less dynamic packet transmission than we have in .NET). We currently use this solution because .Net Core offered by Microsoft to run on multiple platforms is not yet mature and does not offer all the functionalities on the .NET standard platform).

## *3.3. Client-Server components*

### *3.3.1. Client*

In this architecture, the client is a WPF (Windows Presentation Foundation) desktop application specially built to communicate with the server and to run various tasks received from it. Since the application is an executable that does not require installation, the operating system must run .Net Framework. Basically, a web microservice will communicate with another web microservice.

Initially, the client starts a job scheduler on a parallel thread that every minute will try to notify the server with his presence. The task can take two states: (1) either has a job to execute (initializing a code pack already done) - in which case it only notifies the server with its presence; (2) or requires an initialization with the server.

Initializing with the server involves, in the first instance, the arbitrary choice of a code and port that the server will run that are sent to it by a Diffie-Hellman key exchange protocol (IKE). Once the link between the two entities is established, the server will notify the client with the instruction package to be installed. The primary role of a client is to receive a package of instructions from the server, load it in memory, process the information received from the server and then return the result obtained from the execution of that instruction pack. The first step executed by the client is to contact the server in order to receive the instruction package. This instruction pack comes in the form of a ZIP archive.

Before extracting this package, delete the previous directory with instructions from the “process” folder (if it exists), then extract the new content in this folder and load it into memory. This load in memory is run once, no matter how many hits the client will have. This is possible because the three properties are kept static in session: assembly, *methodInfo* and type. The assembly property stores the reference to the loaded DLL, the *methodInfo* property keeps in mind the method called from the DLL, and the type describes the type of the DLL. The install.zip file is the instruction package received from the server that contains (DLLs, XMLs, images, configuration files, etc.) and all the compiled code to be executed in the future process.

This stage marks the beginning of the relationship between the client and the server in order to execute a particular task. Once the client is successfully initialized with a certain task to execute, the server will only send the packet of data that needs to be processed in an encrypted form and will wait for the reply in an encrypted form.

By executing the code received from the server, a “Locked In” system is not formed, the client being able to connect to databases, to call other APIs, a special case being to call other clients running the same or other instructions. This connection is performed in an orchestration system where the server searches for the next available client, is interrogated for the result, and its response is redirected back to the client by the server. This microservice orchestration is called “*ExtendedService*” and the only difference at the client level for this is that there is an encryption optimization.

A technical problem encountered was to reinitialize the client with another package of instructions to execute. Because the memory load is static in a special context (the web server), this was only possible by restarting the whole process to deal with the DLLs loaded into memory. To accomplish this, we created events in Windows that we triggered from the web application running in a desktop application. This is necessary because we are dealing with two different contexts on two different execution threads.

### *3.3.2. Server*

The built-in micro-service has an *ILibraryMath* interface that provides the *SimpleMath* method, and the implementation of the interface is done by the *LibraryMath* class. The *LibraryMath* class extends the generic abstract class *MicroCore*<I, O> that has two matching parameters for input and output. Extending this abstract class, the *ProcessTask* method must be implemented where all the code to be executed is written, and the Run function in the extended abstract class is called to execute this code in the *SimpleMath* method. In this way, it is possible to define interfaces and methods without being constrained by any particular name, but by passing the code through the abstract class we will obtain the total control of the code that we can distribute to different clients. Within this class we can have more functions, imported libraries, being no problem as long as they are grouped in the same package.

The next step was to record this interface in *SimpleInjector*, a library that facilitates the deployment of dependency injection pattern with freely coupled components. In addition to class-interleaved recording in the Simple Injector container to break the dependence between application levels (Pattern Dependency injection), we need to register the class into a microservice storage container that will be scaled by the application. After this step, we will be able to use the function provided by the interface for the created purpose.

Service1 implements IService1 and extends the *MicroCore* abstract class, and then is registered with *MicroContainer.RegisterMicro* in that container. It is worth mentioning the existence of APIs available at *localohst/DynamicMicros/{Service}*, through which clients communicate with the server, the important actions being made available by this API: the client connects, the client notifies the server about its activity, microservices extended, and so on. Next, we will introduce the *MicroCore* and *MicroContainer* classes, which together form the framework of our application.

The *MicroCore* class is an abstract, generic class and it is responsible for call the code from the *ProcessTask* virtual method. This is done by calling the Run method, which in turn calls a public method called *TaskManager*. We mention that the microservice will call in turn also this method. When the ZIP package is sent to the client to be loaded into memory and executed, it is sent with all its dependencies including this class through which the client's microservice management is performed. Execution management involves de-serializing/serializing the data packet to be sent, calling the code itself, calling other APIs, etc.

Returning to the server side, code execution management consists of the steps below:

1) If it is an *ExtendedService* call, the server will be called for the answer.
2) If there is a client available for the query, it will be sent to it to process the result; in the negative case the server will process the data.
3) We query the client for data processing.
4) If the client is encounter problems, we query once again to confirm availability, but we send the server's response (to avoid dead times and high waiting times).
5) We log the ongoing activity.

The *MicroContainer* class is the management space of the entire built microsystem. Here, clients who connect the application (the server) connect and here come the calls of functions that extend the *MicroCore* abstract class for the "extended service" part. This is a static class where the list of tasks to be executed in the microservices, the connected clients list and the client task list tasks that are running those tasks are stored in a dictionary.

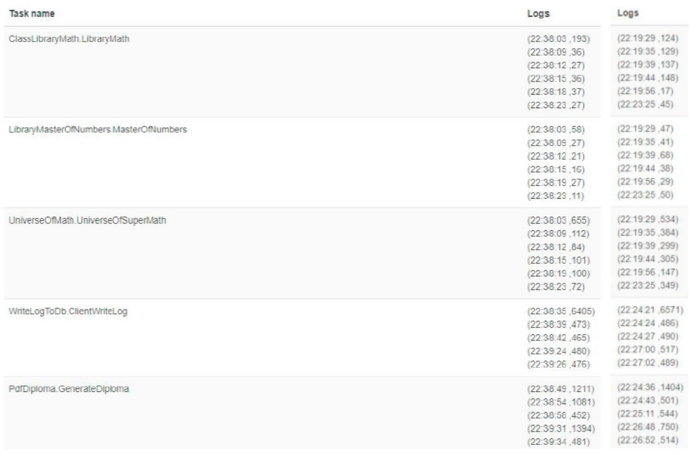
At startup, the class will be registered to be integrated into a microservice with *RegisterMicro*. This will happen only once at initialization. The *AddNewClient* method will provide us with the registration of a new client, the exchange of keys, registration of the IP address of the server and the port it will run. The token received by the new client will be checked before inserting into the client list to verify its uniqueness. Once the connection with a client is established, the server will call the *InstallService* method that packs the data, sends it, and after the client's response, it will be added to the dictionary for that task. The service time that will be allocated to each client depends on the strategy used. In running the *MicroCore* abstract microservice, called on both the server and the client (with *ExtendedService*), querying available clients for the requested task with the *GetNextClient* function. This operation will be executed very frequently and its complexity will have a direct impact on the application's response time. That's why our approach was to randomly choose a client. This is done quickly and from our experiments ensures a uniform distribution of calls.

Another option was to implement a circular list, a solution that comes with the drawback that, in the event of a large flow of clients input/output, updating the circular list will require more time and complexity, things we have tried we avoid them. The *RecordClientError* method is called when a client does not respond to a received query. Following the answer to this point, it is decided to retain or remove that client. Uniquely, clients are identified by a token code sent by the client upon initialization, and each microservice is identified by a namespace and class name. The management of all resources (clients, code) is done through this unitary block that provides support for the required operations.

Regarding the security of the system, measures have been taken to prevent attacks, interceptions and to guarantee data protection. All messages sent between the server and the clients are encrypted with the DES symmetric key algorithm, and a Diffie-Hellman key exchange between the client and the server, that takes place at the initialization of a client. Available clients and running programs are stored in the server's memory. We have chosen this solution because it represented the best option in our opinion as it provides a high speed of access to data, information can change very frequently, and the memory area is very difficult to attack by cyber attackers.

### *3.4. Dynamic behavior of the microservices system*

First of all, all computers which clients will run may be located on the same network or on different networks. Two elements are targeted: (a) the time spent with the data transfer; and (b) the overhead added by a system to manage data (e.g., finds client, encrypts, decrypts, treats errors, etc.). Mainly, we were interested in the behavior of our system in LAN and WAN (Fig. 4).



| Task name | Logs | Logs |
|---|---|---|
| ClassLibraryMath.LibraryMath | (22:38:03 ,193)<br>(22:38:09 ,36)<br>(22:38:12 ,27)<br>(22:38:15 ,36)<br>(22:38:18 ,37)<br>(22:38:23 ,27) | (22:19:29 ,124)<br>(22:19:35 ,129)<br>(22:19:39 ,137)<br>(22:19:44 ,148)<br>(22:19:56 ,17)<br>(22:23:25 ,45) |
| LibraryMasterOfNumbers.MasterOfNumbers | (22:38:03 ,58)<br>(22:38:09 ,27)<br>(22:38:12 ,21)<br>(22:38:15 ,16)<br>(22:38:19 ,27)<br>(22:38:23 ,11) | (22:19:29 ,47)<br>(22:19:35 ,41)<br>(22:19:39 ,68)<br>(22:19:44 ,38)<br>(22:19:56 ,29)<br>(22:23:25 ,50) |
| UniverseOfMath.UniverseOfSuperMath | (22:38:03 ,655)<br>(22:38:09 ,112)<br>(22:38:12 ,84)<br>(22:38:15 ,101)<br>(22:38:19 ,100)<br>(22:38:23 ,72) | (22:19:29 ,534)<br>(22:19:35 ,384)<br>(22:19:39 ,299)<br>(22:19:44 ,305)<br>(22:19:56 ,147)<br>(22:23:25 ,349) |
| WriteLogToDb.ClientWriteLog | (22:38:35 ,6405)<br>(22:38:39 ,473)<br>(22:38:42 ,465)<br>(22:39:24 ,480)<br>(22:39:26 ,476) | (22:24:21 ,6571)<br>(22:24:24 ,486)<br>(22:24:27 ,490)<br>(22:27:00 ,517)<br>(22:27:02 ,489) |
| PdfDiploma.GenerateDiploma | (22:38:49 ,1211)<br>(22:38:54 ,1081)<br>(22:38:56 ,452)<br>(22:39:31 ,1394)<br>(22:39:34 ,481) | (22:24:36 ,1404)<br>(22:24:43 ,501)<br>(22:25:11 ,544)<br>(22:26:48 ,750)<br>(22:26:52 ,514) |

Fig. 4. Recording the system running in the LAN (first column of logs) and WAN (second column of logs).

The column Task name contains all registrations made by the client call for each task and the columns Logs, the hour and the duration in ms for each task processing (left in LAN and right in WAN). We note that the tasks have the highest response time to the first call, after which time it decreases. Naturally, because all the memory loads, address retention, etc. are usually made at the first call. The first three tasks are simple mathematical operations that usually run under a millisecond, time which is also required for our system.

For LAN, we have an average of 20-30 milliseconds per job, resulting in encryption, logging, and network transfer (even if it is local). This LAN communication model is also used at cloud, where computers are located in the same location (Data Centers), and the communication between them is carried out by optical fiber, the latency in the network having minimum values. The results are shown in Fig. 4 left logs column.

To test our WAN application, we configured the router to direct the call from port 80 to: http://192.168.1.160/ (network address), and with IIS (Internet Information Services) running the application, it being accessible from anywhere outside the local network. In order to run the application at the client' level, the right to use the 8000: 9000 ports (arbitrary ports) was required. Clients were located anywhere, the connection settled for the public IP with the API: https://api.ipify.org/. The results are shown in Fig. 4 logs column on the right.

In the results presented in Fig. 4, the values from the right log column in comparison with values from the left log column are 16-17% higher for the first three tasks (not communicating with other microservices) and ± 10% for microservices that download documents from the Internet or interact with a database on a particular server.

## 4. Evaluation

In this study we followed the behavior of the system both in the LAN (connecting 5 computers via Wireless) and WAN (using the name space http://mihaidm.ddns.net/), comparing our system to a monolithic system, these operations being executed on the same computer (see Table 1).

Table 1. System evaluation for networks.

| | Numerical calculation (ms) | Write in BD (ms) | Generate PDF (ms) |
|---|---|---|---|
| localhost | 1 | 4.458 | 15.449 |
| LAN | 25 | 4.408 | 16.415 |
| WAN | 54 | 4.826 | 29.309 |

Testing took place successively on the same device with 5 connected clients for network testing. Each task was executed 100 times, evaluating the total number of milliseconds in all calls.

In case of numerical calculation, it is done the product of two numbers. Microservice does not interact with other microservices, the amount of information sent on the network is small and complexity is minimized to strictly study the time spent with server, client, and network management tasks. If the calculation is executed by the server (localhost), first check if there is an available client, and since there is no client connected, the server processes the result. For the next case, the existence of clients on the LAN shows the execution of the task while networking is very fast, and the processing side is encryption/decryption, finding client response. For the 100 executions, the average time required to complete the operation was 25 ms, promising value considering the flexibility / speed ratio. In the case of WAN, the time is double to the LAN (54ms), this being explained by the encryption process, the transport costs, but the actual execution requires half a millisecond.

Another task we've been following was writing into a database. Specifically, a word that will be written in the database is received as a parameter. We are interested in how quickly the client will communicate with the database, located outside of the local area (for this study, the database was hosted on https://www.my.gearhost.com/). Note that runtime values in LAN and localhost are close. At WAN, the difference is noticeable as there is not much of the time added to processing, client and data management, but with the client's band that connects to the database to insert a value.

Last task done in this study, was the generation of a PDF file, our focus was on evaluating the data transfer time within the system. For this we download a PDF file from https://www.pdf-archive.com/2018/05/14/diploma/diploma.pdf that is loaded into memory. The system will write a name to a specific position and return the result (in the form of byte vectors) back to the server. For localhost and LAN, the difference of nearly 1000 ms represents the time required for encryption and local transport of PDF files. For WAN, the recorded value is higher because the cost of the byte vector transfer is very high.

## 5. Conclusions and Future Work

The generic and abstract nature of the system architecture presented in this work on the server side has made it difficult to design because the same code is executed by both the server and the client. We can state that the present architecture is compact, simple, and easy to understand and expand; the client can execute tasks assigned by the server, the server being the monolith and the client interface.

The proposed architecture allows for the creation of new microservices in a very simple way, which are then automatically integrated into the built system. The innovative elements of this architecture are: it can scale very easily, each new client being assigned a task by the server according to the strategy pursued (the most expensive tasks, the more common, and a combination of the two previously listed or purely and simply arbitrary). Basically, we have a monolith that has the flexibility of a microservice based system. The server handles the dynamic distribution of tasks between clients, providing a dynamic scaling based on a number of parameters (the number of calls to a task, its execution time, or combinations thereof).

One of the future directions takes into account that this system can successfully integrate into a website or API system with a pronounced applicative character. The proposed architecture can be improved and expanded at any time, with availability for multiple platforms (e.g., mobile phone).

Another future direction that we are looking at is considered to be extremely attractive today is that the user provides the processing power in exchange for remuneration (e.g. the BITCOIN system), our application being developed to run microservices on certain computers.

## Acknowledgements

This survey was published with the support by POC-A1-A1.2.3-G-2015 program, as part of the PrivateSky project (P_40_371/13/01.09.2016) and by the README project "Interactive and Innovative application for evaluating the readability of texts in Romanian Language and for improving users' writing styles", contract no. 114/15.09.2017, MySMIS 2014 code 119286.

## References

[1] Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. (2017a) "Microservices: Yesterday, Today, and Tomorrow." Mazzara M., Meyer B. (eds.), *Present and Ulterior Software Engineering*. Springer.

[2] Mazzara, M., Khanda, K., Mustafin, R., Rivera, V., Safina, L. and Silitti, A. (2018) "Microservices Science and Engineering". In: P. Ciancarini, S. Litvinov, A. Messina, A., Sillitti, G. Succi (eds.) *Proceedings of 5th International Conference in Software Engineering for Defence Applications, SEDA 2016*, Springer, 10-20.

[3] Dragoni, N., Lanese, I., Larsen, S. T., Mazzara, M., Mustafin, R., and Safina, L. (2017b) "Microservices: How To Make Your Application Scale". In: Petrenko A., Voronkov A. (eds.) *Perspectives of System Informatics. PSI 2017*. Lecture Notes in Computer Science, **10742**. Springer, Cham.

[4] Melis, A., Mirri, S., Prandi, C., Prandini, M., Salomoni, P., and Callegati, F. (2016) "A Microservice Architecture Use Case for Persons with Disabilities". At the *2nd EAI International Conference on Smart Objects and Technologies for Social Good*, DOI: 10.1007/978-3-319-61949-1_5.

[5] Zimmermann, O. (2017) "Microservices Tenets: Agile Approach to Service Development and Deployment, Computer Science - Research and Development", **32 (3-4):** 301-310.

[6] Xia, C., Zhang, Y., Wang, L, Coleman, S., and Liu, Y. (2018) "Microservice-based cloud robotics system for intelligent space". In: *Robotics and Autonomous Systems* **110**, DOI: 10.1016/j.robot.2018.10.001.

[7] Bogner, J., Fritzsch, J., Wagner, S., and Zimmermann, A. (2019) "Microservices in Industry: Insights into Technologies, Characteristics, and Software Quality". At the *2019 IEEE International Conference on Software Architecture Workshops (ICSAW)* At: Hamburg, Germany.

[8] Akentev, E., Tchitchigin, A., Safina, L., and Mzzara, M. (2017) "Verified type checker for Jolie programming language", https:// arXiv.org/pdf/1703.05186.pdf.

[9] Černý, T., Donahoo, M.J., and Trnka, M. (2018) "Contextual understanding of microservice architecture: current and future directions". *ACM SIGAPP Applied Computing Review* **17 (4)**: 29-45, DOI: 10.1145/3183628.3183631.

[10] Larucces, X., Santamaria, I., Colomo-Palacios, R., and Ebert, C. (2018) "Microservices". In: *IEEE Software*, **35/3**: 96-100.

[11] Kalske, M. (2017) "Transforming monolithic architecture towards microservice architecture". M.Sc. Thesis, *Univ. of Helsinki*.

[12] Lenarduzzi, V., and Taibi, D. (2016) "MVP Explained: A Systematic Mapping Study on the Definitions of Minimal Viable Product". At the *42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 112-119.

[13] Taibi, D., Lenarduzzi, V., Janes, A., Liukkunen, K., and Ahmad, M. O. (2017) "Comparing Requirements Decomposition within the Scrum, Scrum with Kanban, XP, and Banana Development Processes". In: Baumeister H., Lichter H., Riebisch M. (eds.) *Agile Processes in Software Engineering and Extreme Programming*. Lecture Notes in Business Information Processing, **283**. Springer, Cham.

[14] Gómez, A., Benelallam, A., and Tisi, M. (2015) "Decentralized Model Persistence for Distributed Computing". At the *3rd BigMDE Workshop*, L'Aquila, Italy.

[15] Kandave, K. R. (2018) "High performance computing on Azure". Nanette Ray (ed.), *AzureCAT, Microsoft Corporation*.

[16] Sreenivas, V., SriHarsha, S., and Narasimham, C. (2012) "A Cloud Model to Implement SaaS". In: *Advanced Materials Research* **341-342**, Trans Tech Publications, Switzerland, 499-503.

[17] Badidi, E. (2013) "A Framework for Software-As-A-Service Selection and Provisioning". In: *International Journal of Computer Networks & Communications (IJCNC)*, **5 (3)**: 189-200.

[18] Lynn, T., Rosati, P., Lejeune, A., and Emeakaroha, V. (2017) "A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms". At the 2017 *IEEE 9th International Conference on Cloud Computing Technology and Science*, 162-169.

[19] Adzic, G. and Chatley, R. (2017) "Serverless Computing: Economic and Architectural Impact". At: *ESEC/FSE'17,* September 4-8, 2017, Paderborn, Germany, ACM.

[20] Diffie, W. and Hellman, M. (1976) "New directions in cryptography". In: *IEEE Transactions on, Information Theory*, **22 (6)**: 644–654.

[21] Kratzke, N. (2015) "About Microservices, Containers and their Underestimated Impact on Network Performance". At the *CLOUD Comput. 2015*, **180** https://arxiv.org/abs/1710.04049.