11th ANKARA INTERNATIONAL AEROSPACE CONFERENCE
8-10 September 2021 - METU, Ankara TURKEY

AIAC-2021-137

MIGRATION OF SATELLITE MONITORING AND CONTROL SOFTWARE FROM MONOLITHIC ARCHITECTURE INTO MICROSERVICE ARCHITECTURE

Volkan Ulutaş[1] and Necdet Engin Öztuna[2]
Türk Havacılık ve Uzay Sanayi A.Ş.
Ankara, Turkey

Onurcan Yozgat[3], Necla Ebru Fıratlı Eryılmaz[4], Akın Yılmaz[5]
Türk Havacılık ve Uzay Sanayi A.Ş.
Ankara, Turkey

ABSTRACT

This article reports the experiences and lessons learned during the incremental migration to the microservices architecture of the Monitoring and Control Software (MCS), which is developed in a monolithic architecture by Turkish Aerospace Inc. The problems encountered in the monolithic architecture, the experiences about how these problems are solved with the microservices architecture, the decisions made and technology choices are explained. This study covers the migration steps taken to migrate the MCS to the microservices architecture; it was aimed to explain the separation of the modules (functional decomposition) in the existing architecture into microservices using domain-driven design, flexibility, accessibility, scalability, and distribution.

INTRODUCTION

The Monitoring and Control Software is a complex application that requires high level of domain knowledge, may contain more than one programming language, and is developed using monolithic architecture by more than one team. When software developers need to add a new feature to the system, they need to learn the entire codebase, and analyze its impact on other modules. When the software, designed with monolithic architecture, becomes more complex, maintenance and managing costs increase. [Furda, 2018]

In microservices architecture, by decomposition of the services according to the domain knowledge, it is possible to develop reliable and robust software. Moreover, usage of microservices architecture reduces the adaptation time of a software developer to the project [Evans, 2003].

A new software developer learns the structure of the services divided into small parts instead of learning the entire codebase after joining the project. Another factor that facilitates project

[1] Senior Software Engineer, Email: volkan.ulutas@tai.com.tr
[2] Satellite Ground Software Chief Engineer, Email: eoztuna@tai.com.tr
[3] Senior Software Engineer, Email: onurcan.yozgat@tai.com.tr
[4] Software Design Engineer, Email: neclaebru.firatli@tai.com.tr
[5] Software Engineering Manager, Email: akin.yilmaz@tai.com.tr

management process is that software project managers divide human resources into teams according to the domain [ECSS-E-ST-40C, 2009].

For projects that require expertise such as satellite management systems, dividing the software into small services reduces complexity, cost of development, testing, and maintenance. During the lifetime of the satellite which has limited resources, failures in the ground software must be fixed in a short time so that they do not affect satellite operations. It takes time to resolve the errors and analyze the impacts of the errors on software designed in a monolithic architecture. All the tests of the application should be rerun and all the software should be updated in the production environment. This will cause the system to be interrupted during the system update. In microservices architecture, after the source of the error is identified and the solution is implemented, only the tests of the related service should be run and source code should be deployed to the production environment. During the update of the related service, the rest of the system is not affected and the operations remain uninterrupted.

It may be necessary to use different programming languages for tasks which have different requirements such as performance, security, and environment. It is difficult to use different programming languages in software developed in a monolithic architecture. It becomes possible to develop each service in different languages in the microservices architecture, as the services can work independently from each other. Thus, dependency on a programming language is reduced and selection of problem-specific language and technology can be made. A similar situation applies to databases. Since each service stores its data on its own database, it is possible to select the database technology of the services according to non-functional requirements. For example, using a NoSQL database in one microservice may be efficient, while a relational database for another service may provide better results. In this way, it is possible to use problem-specific technologies by eliminating dependencies on any particular technology.

METHOD

Monolithic Architecture

A monolithic architecture is built as a single, self-contained unit where all business logic and components are combined in a single application which are designed to handle multiple tasks such as processing telemetry data which is retrieved from satellite, handling telecommand data which is sent from ground software to satellite, managing event data and generating space system model (ex. XTCE data). All domain-driven functionality is enclosed into a single application whose modules cannot be run separately. This type of design is tightly-coupled and all code exists inside a single codebase which means that the entire application is deployed at the same time, and scaling is achieved by adding additional machines horizontally with the help of a load-balancer [Lewis and Fowler, 2014]. Although deployment time and complexity is low, lots of drawbacks occur when the application becomes larger and the team grows in size:

- Intelligibility
- Manageability
- Deployment
- Scalability
- Commitment to technology stack
- Learning curve difficulty
- Single point of failures
- Testing activities required long time

Software developed in accordance with monolithic architecture are generally developed using client-server and a common database. A relational database, server application and desktop client applications have been developed with a similar approach in satellite ground software.

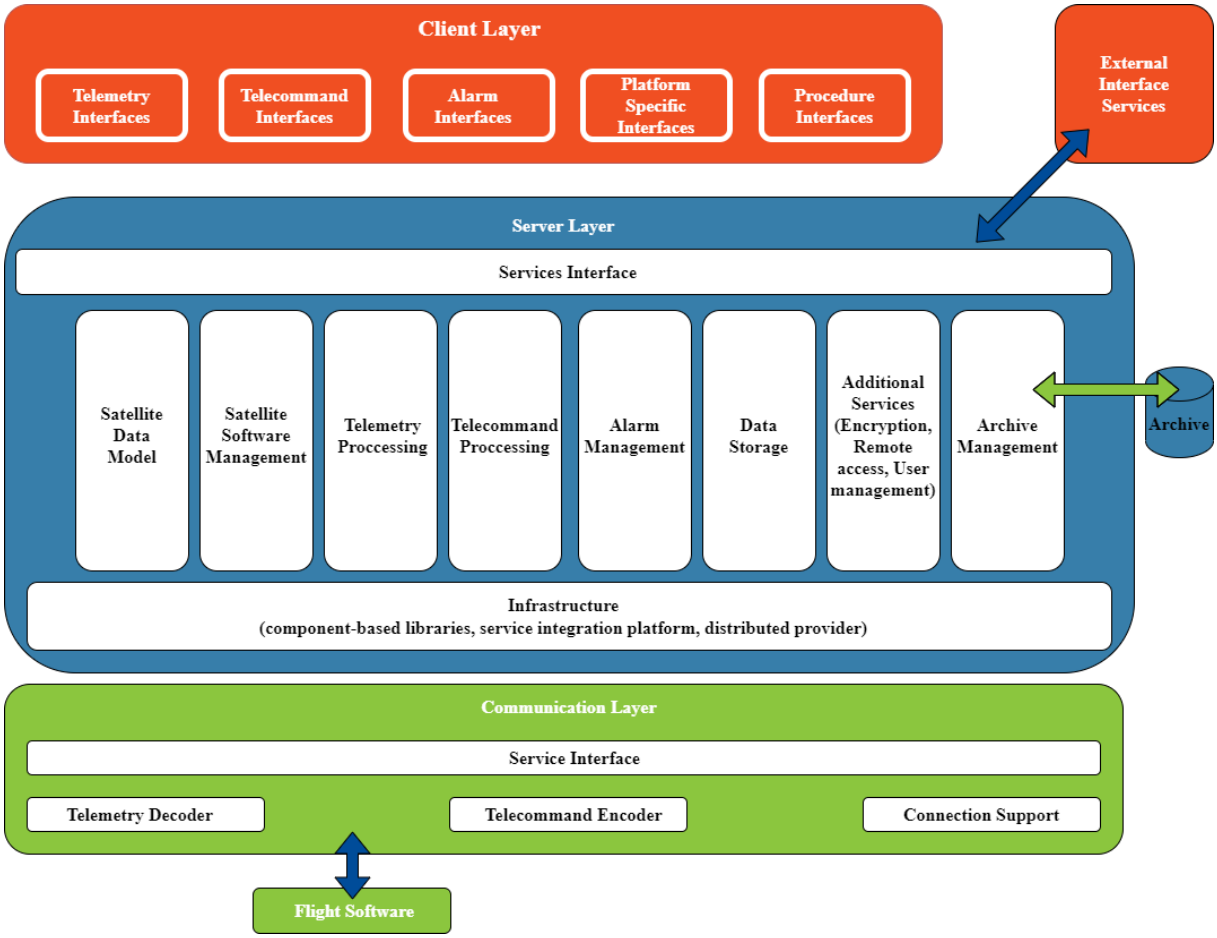

Figure 1: Monolithic architecture of MCS

Microservices

Microservices are an architectural and organizational approach to software development where software is composed of small independent services that communicate with over lightweight technologies (e.g. HTTP API, AMQP.) [Lewis and Fowler, 2014] Because of their small size, they are easier to manage and more fault-tolerant, as a single service failure will not bring the entire system down, as a monolithic design would. As a result, this design style enables for the development of architectures that are flexible, modular, and easy to adapt.

- Flexibility
- Resilience
- Scalable
- Deployment
- Free to choose technology stack
- Ease of learning curve

Microservices solve these problems by allowing separate development teams to build and deploy small services; the resulting flexibility allows teams to focus on enhancing each service and improving business value. Thus, DevOps (Development and Operations) and Continuous Delivery are a good fit for microservices architectures in practice. [Fowler, 2016.]

Migrating from a monolithic system into microservices is a type of software modernization, also known as application modernization, platform modernization or legacy modernization that conversion, rewriting or porting of a legacy system to modern architecture, software libraries, protocol or hardware. [Chiang, 2006]

Migration of Monitoring and Control Software to microservices architecture

There are four software modernization techniques which can be applied when migrating from monolithic systems to microservices:

- Big Bang Approach
- Adding New Feature as a Separate Service
- Split Frontend and Backend
- Extract Services

Big Bang Approach:

Big Bang approach is defined as creating a new application from scratch. It seems simple to migrate in this way but there are some problems that need to be taken into account. This approach is based on rewriting the entire system. Therefore, it will take undesired amounts of time and effort. Because of these reasons, there are alternative strategies that propose refactoring the system incrementally which are more common.

Adding New Feature as a Separate Service:

The decision to migrate from a monolith to a microservices architecture is often made because of the difficulties experienced in managing the monolithic system. As the monolithic system grows adding a new feature becomes more difficult. This approach proposes developing the new feature as a separate microservice rather than adding into a monolithic application. With this approach the new system consists of the unchanged legacy monolith and a newly created microservice with the added feature. Such a system requires a request router to route the requests regarding the new feature to the microservice and the rest of the requests to the legacy monolith. The request router handles the HTTP requests similar to the API Gateway. The new microservice can communicate with the monolith by invoking a remote API, access the monolith's database directly or copy of the corresponding monolith data and synchronize between applications. This approach prevents the monolithic application from becoming more complex and allows teams to gain experience with microservices architecture but not solve the problems arising from the monolith. [Richardson, 2016]. This approach can be considered as an initial strategy for migrating from a monolith to a microservices architecture.

Long-term archive service is responsible for the storage and management of long-term data and also serves data to the client application in the MCS. Long-term archive service was a new feature that needed to be developed. It was developed with this technique as a separate microservice and integrated with the monolith.

Split Frontend and Backend:

An enterprise application usually consists of three layers. The presentation layer has the user interfaces that the user can use to interact with the application. It can also handle HTTP requests. Business logic layer is responsible for processing the user inputs. It also serves as a bridge between the presentation layer and the data-access layer. This layer has domain specific logic. The data access layer is responsible for accessing the data which is stored in the database and data from the message brokers and providing this data to the business logic layer.

This modernization strategy suggests separating the presentation layer from the business logic and data access layers. This allows the teams to develop and test easily both the frontend and backend. After splitting the frontend and backend into two applications, remote APIs must be implemented to enable communication between them. As in the first

strategy this is only a step for migrating from a monolith to a microservice because these two applications could still become a monolith.

The client side of the MCS was developed as a desktop application and has common dependencies and data models with the server side. Based on this technique, the client side and the server side were separated from each other and REST APIs were developed for their communication. This separation is also the first step of developing the client side as a web application with the new technologies.

Extract Services:

In monolithic architecture there are many modules in the application. These modules could be microservice candidates while migrating the architecture from monolith to microservice. There are some techniques for choosing and prioritizing the modules that need to be extracted from a monolith as a microservice. As a starting point it will be a good choice to start with the smaller and simple modules rather than the complex ones. Once the extraction starts from the simple modules the monolith starts to shrink and the teams start to gain experience on microservices architecture. Modification frequency on the module is an important sign for choosing the modules to extract. If there are too many changes or planned feature additions on the module, there will be positive impact on development time and effort after extracting that module to a standalone microservice. In addition to these techniques, modules could be chosen according to their system requirements or availability requirements. For example, if the module has a task that requires too much memory or CPU then deploying that module as a standalone service will be important when considering scalability and availability concerns.

After determining the modules to be extracted, defining an interface between the extracted service and the monolith is the first step. Implementing and refactoring domain-specific parts may be difficult because of the dependencies between modules and data models but when the interface is defined clearly these parts become easier to implement.

All the modules in the MCS monolithic architecture were analyzed, prioritized and extracted with this technique while migrating from monolithic architecture to microservices architecture.

The migration from monolithic to the microservices architecture is planned in four stages.

In the first stage, researches have been conducted on usage of the auxiliary services and technologies in the microservices architecture. As a result of these researches, it was decided to use REST (Representational State Transfer) API for simultaneous messaging and AMQP (Advanced Message Queue Protocol) technology for asynchronous inter-microservice communication.

Unlike the monolithic architecture, multiple services on the server side need to communicate with each other and the client side in the microservices architecture. A router is needed to avoid the uncertainty of which service should respond to a request sent by the client. As a result of the researches carried out at the first stage, it was decided to use API-Gateway as a router. API-Gateway allows the requests received from the client to be routed to the relevant microservices through the REST interface. Thanks to the identification and authorization capability of API-Gateway, the requests from the client are authorized and directed to the relevant microservice. The requests from unauthorized users are rejected.

API Gateway is responsible for request routing. All the requests made by the client go through the API Gateway. After that, the API Gateway routes requests to the appropriate microservice. However, at this point, a problem about checking whether the services were up or not was encountered. Service Discovery allows us to get the health information of the microservices on the server-side and routes the requests to the microservices that are running. When the microservices are ready to operate, they are registered to Service Discovery, which runs redundantly in the system as an independent service. Service

Discovery reports server states by periodically checking whether applications are running or not.

In the second stage, the MCS was developed according to the principles of domain-driven design, and the microservices architecture shown in Figure 2.

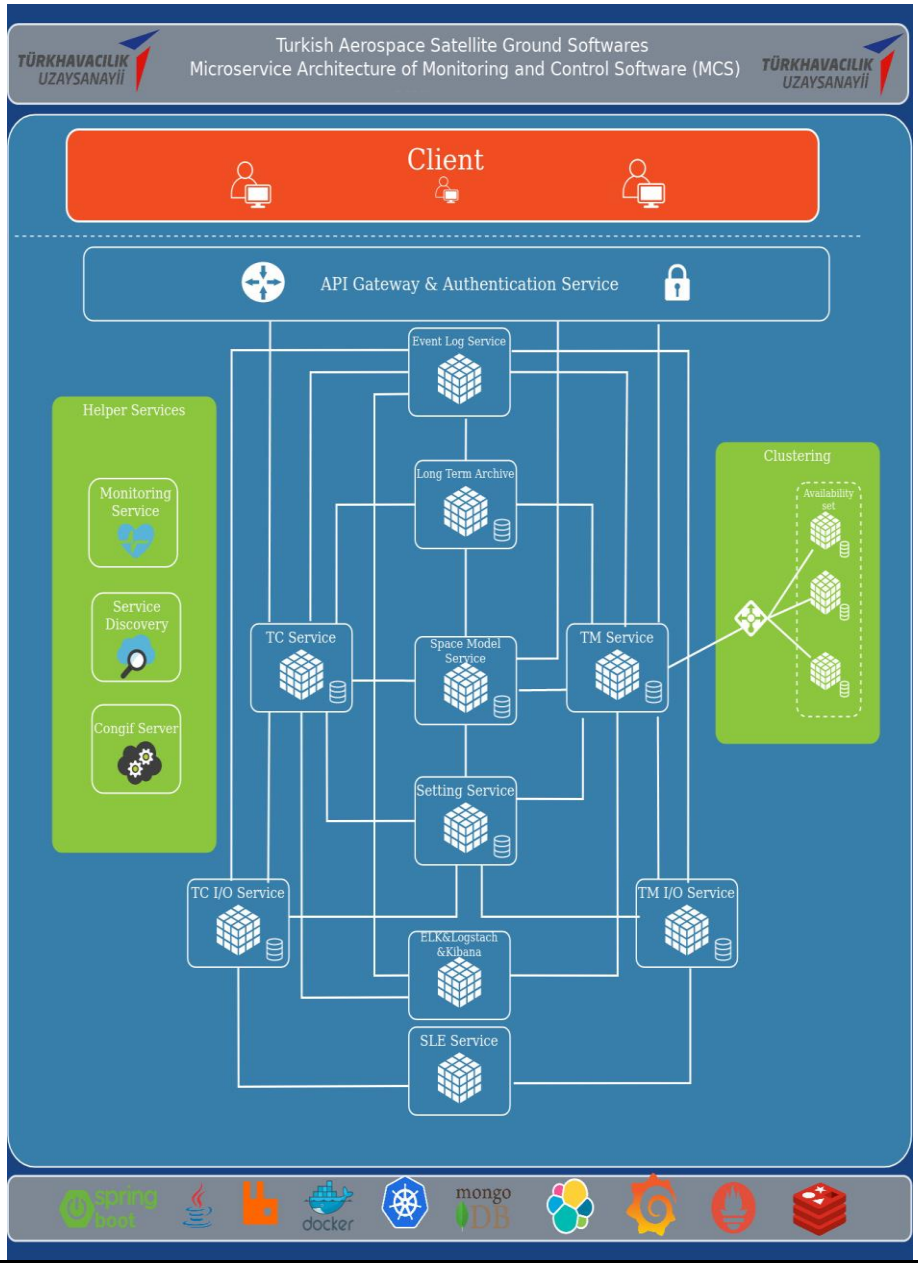

Figure 2: Microservices architecture of the MCS and it's technology stack

The determination of the complexity of the services to be included is another problem encountered during the migration from a monolithic architecture to a microservices architecture. The main goal of microservices architecture is to separate the system into software components focused on a single purpose that is independent of each other [Newman, 2015]. In systems such as MCS where multiple tasks (e.g. telemetry processing, telecommand generation, logging, archiving) are performed sequentially in a chain structure. It is necessary to decide which stages of these tasks should be developed as separate services. One solution is to develop a separate microservice for all the stages. However, in this way, there may be excessive amount of services in the system. This will increase the complexity of management, distribution, and inter-communication with other microservices. It will contradict with the principle of increasing simplicity and clarity, which is one of the starting points of the microservices architecture approach. Therefore, it would be more appropriate to divide monoliths according to the purpose of the use of the software. Using the domain-driven design model and the migration strategies which are mentioned previously, the MCS is divided into the following microservices:

TM Service (Telemetry Service): It is the service where the telemetry data is received from the TM I/O service (Telemetry Input / Output service). The telemetry is decoded, processed, and served to the client application and other services that need this data according to the definitions that are provided by the Space Model Service.

TM I/O Service (Telemetry Input / Output Service): It is the service in which raw data are received from the satellite. TM I/O service establishes a connection with the SLE (Space Link Extension) service to communicate with the satellite. TM I/O service is also responsible for the storing of raw data.

SLE Service (Space Link Extension Service): SLE Service implements the SLE standard which is a reference model that identifies a set of SLE Transfer Services that enable missions to send forward space link data units to a spacecraft and to receive/return space link data units form a spacecraft [CCSDS, 2015].

TC Service (Telecommand Service): It is the service where the telecommands are to be sent to the TC I/O Service (Telecommand Input / Output Service). The telecommand is encoded, processed, and presented to the client application and other services that need this data according to the definitions in the Space Model service.

TC I/O Service (Telecommand Input / Output Service): It is the service which establishes a connection between the satellite and the SLE service. The service is also responsible for saving the raw telecommand data.

Event Log Service: It is the service that serves the event and warning notifications to the client application generated by the services and the satellite.

Settings Service: It is a service where satellite-specific and domain-oriented settings are kept. It retrieves settings information from other services and stores them in it's database. It also provides settings to the client application and informs the related services in case of changes on settings made by the client application.

Space Model Service: It creates data structures (telecommand, telemetry, and parameter models) necessary for decoding and encoding operations between satellite software and the MCS, storing and versioning changes, and it is the service that notifies these changes to the client application and the related services.

Long-Term Archive Service: There is a need for short-term and long-term archiving of certain database information specific to the domain and the requirements. This service is

responsible for the storage and management of long-term data. It also serves data to the client application.

Auxiliary Services: It aims to increase the efficiency of development and maintain processes by integrating services such as configuration management service, service discovery, API-Gateway, identification and authorization, load balancing, circuit-breaking, data observability, and other technologies [ECSS-Q-ST-80C, 2009].

All of these services, which include messaging, monitoring, and logging make up the architecture's infrastructure. To ensure high availability and performance, all of these infrastructure systems run in clusters. This is called load scalability.

In the third stage, the details of the microservices architecture design were developed using the chosen programming language and appropriate design patterns. In this stage, difficulties were encountered such as communication among microservices, integration of developed services, communication of user interface and services.

In the fourth stage, it is planned to discuss the approaches for managing the additional testing complexity of multiple independently deployable components as well as how to have tests and the application remain correct despite having multiple teams each acting as guardians for different services [Lewis, 2014]. Unit testing, integration testing, component testing, contract testing and end-to-end testing scenarios will be applied.

CHALLENGES

Microservice design seeks to address the challenges associated with monolithic systems. In addition to the advantages of microservices architecture, there are also some challenges that need to be overcome. These challenges are described as follows.

Service Discovery: In the monolithic architecture, the addresses and ports to be reached do not change dynamically and are kept statically in some properties or settings files. Due to the use of static addresses in the monolithic architecture, service discovery is not needed, but the situations are different in microservices architecture. Since the addresses of the services are determined dynamically in the microservices architecture, when a service communicates with another service, it cannot know the address of the service with which it will communicate directly. The mechanism by which these dynamically changing service addresses are managed is called service discovery. Service registry is the main part of service discovery. The service registry works like a database, that is, it is the section where the addresses of the services are stored. [Richardson, 2015]

There are two service discovery patterns: client-side discovery and server-side discovery. In the client-side discovery, the requesting service obtains the address directly by querying the service registry. [Richardson, (n.d.)] In the server-side discovery, there is a router between the service and the service registry, so the requesting service obtains the address from the registry via the router [Richardson, (n.d.)] We preferred to use server-side discovery instead of client-side discovery in our application.

Service instances must be registered with and deregistered from the service registry. There are two ways for this: self-registration pattern and third-party registration pattern. In the self-registration pattern, the service instances register and deregister themselves to/from the service registry so this responsibility is on the services themselves. In the third-party registration pattern, the other component is responsible for the registration and deregistration of the services. [Richardson, 2015] We preferred to apply self-registration pattern instead of the third-party registration pattern in our application.

We used Netflix Eureka and Apache Zookeeper technologies while developing the concept of service discovery in our application.

Interservice Communication: There are two basic communication types between microservices. Synchronous communication is a communication pattern where one microservice calls an API which is exposed by another microservice and waits for the response. HTTP and gRPC protocols are used for this type of communication. On the other hand asynchronous communication is a communication pattern where one microservice sends a message to other microservices without waiting for any response. There are both advantages and disadvantages between these communication types. [Smid, Wang and Cerny, 2019] According to the requirements both patterns are used in the MCS.

As a synchronous communication example in the MCS, when microservices need telemetry or telecommand definitions as a data structure they need to request these information from Space Model Service. After a request is received via an HTTP request, Space Model Service provides the information synchronously. This type of communication is not complex and is easy to trace. Because of the easy tracing, error handling is much easier than asynchronous messaging. For synchronous messaging, both services must be available and must work properly.

Telecommand status updates must be notified by Telecommand Service to other microservices after commands are sent to the satellite. These telecommand status update notifications are sent asynchronously. In the MCS, RabbitMQ is used as an AMQP protocol for asynchronous messaging. Because the sender microservice does not need to know the receiver microservice, coupling is reduced in this type of communication. These status updates must be notified to more than one microservices therefore pub/sub model is used to notify subscribers to handle this requirement. Queue brokers could be scaled easily according to the workload. When handling failure cases in services, asynchronous messaging protocols have some advantages over synchronous messaging protocols. For an asynchronous messaging protocol, it is not important whether the receiver microservice is available or not. Because when the failure in the receiver microservice is fixed or revised with updates on that microservice, messages will be received from the queue broker. But installing and managing message broker makes another operational complexity to handle and could be a bottleneck of the system therefore it must be configured properly. [“Designing interservice communication for microservices”, 2019] As another advantage of asynchronous messaging, when the request needs data from more than one service latency will be increased in synchronous messaging. For example, if Service A needs data which is stored in Service B and Service C partially, then Service A sends a request to Service B and after that request Service B must send a request to Service C. Number of services could be more than two according to stored data types or the complexity of the data. As the number of requested services are increased, the amount of latency may also increase. On the other hand, latency could also be a problem for asynchronous messaging when message queues becomes overloaded.

Configuration Management Service: This service mainly serves a purpose of updating service settings without the need of redeployment. The source code and program configurations should be stored in two different repositories. Although it may be necessary to synchronize these repositories when the configuration keys change, they should be handled independently of one another. Besides, any updates on the configuration repository should be notified to the running services, which they should then respond accordingly.

Circuit Breaker: A client or a requester service usually make remote calls to a target service running in different machines across a network. One of the big difference between in-memory calls and remote calls can fail, or hang without a response until some timeout limit is reached. If there are high number of unresponsive calls, the system may run out of critical resources leading to cascading failures across more than one service. It may cause a single

point of failure thus that can be overcome. The purpose of the circuit breaker is to solve this problem. The remote call is wrapped with a circuit breaker object, which monitors for failures. Once the failures reach a predefined threshold, the circuit breaker forwards remote calls to an alternative service, or returns an error message. This will make sure the service is responsive. There are three states in the circuit breaking mechanism. When everything is normal, the circuit breaker is in the closed state. When the failures exceed the predefined threshold, it goes into the open state or half-open state. When the circuit breaker is in the open state, the remote calls are forwarded to an alternative service or a default error message is returned. When the circuit breaker switches to a half-open state, it tests if the underlying problem persists. [Lewis, 2014]

Load Balancing: There are two types of load balancing methods used in the MCS, internal and external load balancing. In internal load balancing, each server should have an internal load balancer that retrieves a list of available instances of a requested service from the service registry using a service registry client. The internal load balancer will then use local metrics, such as the response time of the instances, to manage the load between the accessible instances. An internal load balancer reduces the need to set up an external load balancer and requires various load balancing methods to be used by different service clients. In external load balancing, an external load balancer may be established, which retrieves the list of available instances from the service registry and balances the load between service instances using a centralized algorithm. This load balancer will act as a proxy (which is not suggested) or an instance address locator. Another option is to use a service registry with built-in load balancing capabilities and the ability to serve as a load balanced address locator.

FUTURE WORK

The containerization technology is widely used to deploy applications in microservices architecture. By containerization, the developed applications can run independent from the platform and their dependencies can be used in isolation. In this way, the developed applications exhibiting a different operating behavior in different environments can be prevented. Docker is used to manage and deploy the MCS microservices we have developed to the target environment. Docker images of microservices are created, versioned and stored in the artifactory manager.

Creating Docker images is not enough to run applications. A container orchestration tool is needed for operations such as checking the health status of running applications, restarting them in case of failures, and scaling applications. At this point, we use Kubernetes to manage MCS microservices. When it is desired to run services with different parameters in different environments such as development, test and production, there is a need to make changes in the Kubernetes configuration files. For future work, it is planned to use Helm Charts or a similar tool to resolve this issue.

Non-functional needs such as traceability, load balancing and monitoring appear as services and software parts that are both critical and time-consuming to code. Developing these non-functional but necessary code increases the complexity and maintainability of the overall system. At the point where we want to get rid of this complexity, we come across the concept of service mesh.

Service Mesh is an architectural layer that offers the solution of many non-functional problems that a service should consider. It is mainly used for observability, traffic management and security. The fact that service mesh can perform operations such as discovery, load balancing, failure recovery, metrics, encryption, traceability, authentication/authorization and monitoring without the need for any coding creates a very advantageous situation for developers. [Calcote and Butcher, 2019]

There are many products that offer service mesh, we have found it appropriate to use Istio for our future work. Istio works with Docker and Kubernetes. It has two parts, the data plane and the control plane. The data plane manages the network traffic between services. Istio uses the proxy to control all network traffic and it is called sidecar or envoy proxy. The control plane is responsible for controlling the proxy according to the values of the configuration. It allows proxy to be reprogrammed when the configuration is updated. Istio has its own add-ons that allow direct use of Zipkin, Grafana, Jeager, Kiali and Prometheus. [“The Istio service mesh”, (n.d.)]

Further studies will be carried out to improve on issues such as redundancy, fault tolerance, and durability in the microservices architecture. In addition to service redundancy, studies will be carried out to ensure that databases in each service work with backup. It is planned to work on the efficient use of logging and visualization infrastructure and security architecture. Future work will consist of experience on containerization, continuous integration and delivery approaches of services.

DISCUSSION

There are some disadvantages of the microservices architecture. The biggest disadvantage of a microservices architecture is it’s increased complexity over a monolithic application. It is difficult to manage a large number of services in the microservices architecture. As microservices rely on messaging, it increases communication cost (network latency, message processing). It is more difficult to test and monitor the system. It is also more difficult to maintain transaction management (security, consistency, atomicity, rollback etc). Microservices architecture provides eventually consistency because of its distributed structure. Therefore, if the system requires transactional consistency, then the microservices architecture is not a suitable design solution. Despite these disadvantages, the migration of monolithic architecture to microservices architecture is aimed due to the reasons that it provides better scalability, higher availability, increased fault tolerance, support of any development platform, different language and technologies for the services.

CONCLUSION

In this paper, we have analyzed a paradigmatic case study of a mission-critical system: the Monitoring and Control Software of Turkish Aerospace. We have investigated both the legacy monolithic architecture and the new microservices architecture. Both architectures have been documented in terms of their domain-driven design, implementation, scalability, flexibility, accessibility, and distribution.

This has resulted in a thorough investigation of how to divide a monolithic service into many services that leverage domain-driven knowledge about space systems. Re-engineering of the transformation of monolithic service into microservices led to reduced complexity, lower coupling, higher cohesion, and simplified integration.

References

Chiang, C., and Bayrak, C. (2006), "*Legacy Software Modernization*," 2006 IEEE International Conference on Systems, Man and Cybernetics, pp. 1304-1309, doi: 10.1109/ICSMC.2006.384895, Oct 2006.

Calcote, L., Butcher Z. (2019) *Istio Up and Running*. O'Reilly Media, Inc., 2019.

Designing interservice communication for microservices. (2019), available on https://docs.microsoft.com/en-us/azure/architecture/microservices/design/interservice-communication

Evans E. (2003) Domain-Driven Design*: Tacking Complexity in the Heart of Software.* *Addison-Wesley Longman* Publishing Co., Inc., Boston, MA, USA, 2003.

European Cooperation for Space Standardization (2009), ECSS-E-ST-40C Space Engineering – Software.

European Cooperation for Space Standardization (2009), ECSS-Q-ST-80C Software Product Assurance.

Fowler, S. J. (2016) *Production-ready Microservices: Building Standardized Systems Across an Engineering Organization*. O'Reilly Media, Inc., 2016.

Furda, A., Fidge, C., Zimmermann, O., Kelly, W., Barros, A. (2018) *Migrating Enterprise Legacy Source Code to Microservices*, IEEE Software.

Lewis, J. and Fowler, M. (2014). *Microservices,* available on http://martinfowler.com/articles/microservices.html

Newman, S. (2015). Building Microservices. O'Reilly.

Richardson, C. (n.d.) *Pattern: Client-side service discovery*, available on https://microservices.io/patterns/client-side-discovery.html

Richardson, C. (n.d.) *Pattern: Server-side service discovery*, available on https://microservices.io/patterns/server-side-discovery.html

Richardson, C. (2015) *Service Discovery in a Microservices Architecture*, available on https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

Richardson, C. (2016) *Refactoring a Monolith into Microservices*, available on https://www.nginx.com/blog/refactoring-a-monolith-into-microservices/

Space Link Extension – Internet Protocol for Transfer Services (2015), Blue Book CCSDS 913.1-B-2.

Smid, A., Wang, R., and Cerny, T. (2019*). Case study on data communication in microservices architecture.* In Proceedings of the Conference on Research in Adaptive and Convergent Systems (RACS '19). Association for Computing Machinery, New York, NY, USA, 261–267.

The Istio service mesh, (n.d.) available on https://istio.io/latest/about/service-mesh/