Using microservices and event driven architecture for big data stream processing

Cite as: AIP Conference Proceedings **2172**, 090010 (2019); https://doi.org/10.1063/1.5133587
Published Online: 13 November 2019

Svetoslav Zhelev and Anna Rozeva

Using Microservices and Event Driven Architecture for Big Data Stream Processing

Svetoslav Zhelev[1, a)] and Anna Rozeva[1, b)]

[1]*Technical University of Sofia, 8 Kliment Ohridski blv., 1000 Sofia, Bulgaria*

[a)] Corresponding author: sk@goodlightsolutions.com
[b)] arozeva@tu-sofia.bg

Abstract. Choosing the appropriate architecture for a big data project is not a straightforward task. The architect needs to have knowledge in the problem domain, future application requirements, big data technology landscape and architectural patterns. Today's dynamic business environment enforces constantly evolving big data applications, which often need to process thousands of messages per second. Solutions implementing microservices and Event Driven Architecture (EDA) could provide scalability and extensibility to the required application. The article will outline microservices and EDA challenges, advantages and potential problems concerning big data stream processing.

1. INTRODUCTION

Modern software usually needs to process big data streams and has complex business requirements. Software architecture is essential for application's maintainability and extensibility. However, choosing the appropriate architecture is not a trivial task. A careful consideration of application feature requirements, project time schedule, business requirements and development team expertise is essential. Although microservices are a natural fit for complex applications and cloud environment in many cases it may be better to start with a monolithic architecture and move the application to microservices later on. The communication between different microservices in the application is usually implemented by using either REST webservices or Event Driven Architecture (EDA) and stream processing.

Event Driven Architecture (EDA) is a loosely coupled architecture, which is based on asynchronous processing of real-time flow of events. Although the concept originated in the early 2000's, it has gained a lot of attention in the last years. EDA is a preferred architectural style for many cloud-based applications, which demand high availability and high data throughput.

2. MICROSERVICES

Mircroservices are an architectural style in which a single application is built by using multiple small services [1]. Each service runs in its own process and communicates with other services by using lightweight mechanisms, often REST web services. Each service is a small component, which performs a single business purpose - authentication, notification, payment processing, etc.. The different services may use different technology stacks - databases, frameworks or programming languages. They are also independently deployable and scalable. A good way to understand microservices is through a comparison with traditional monolithic architecture (Fig. 1).

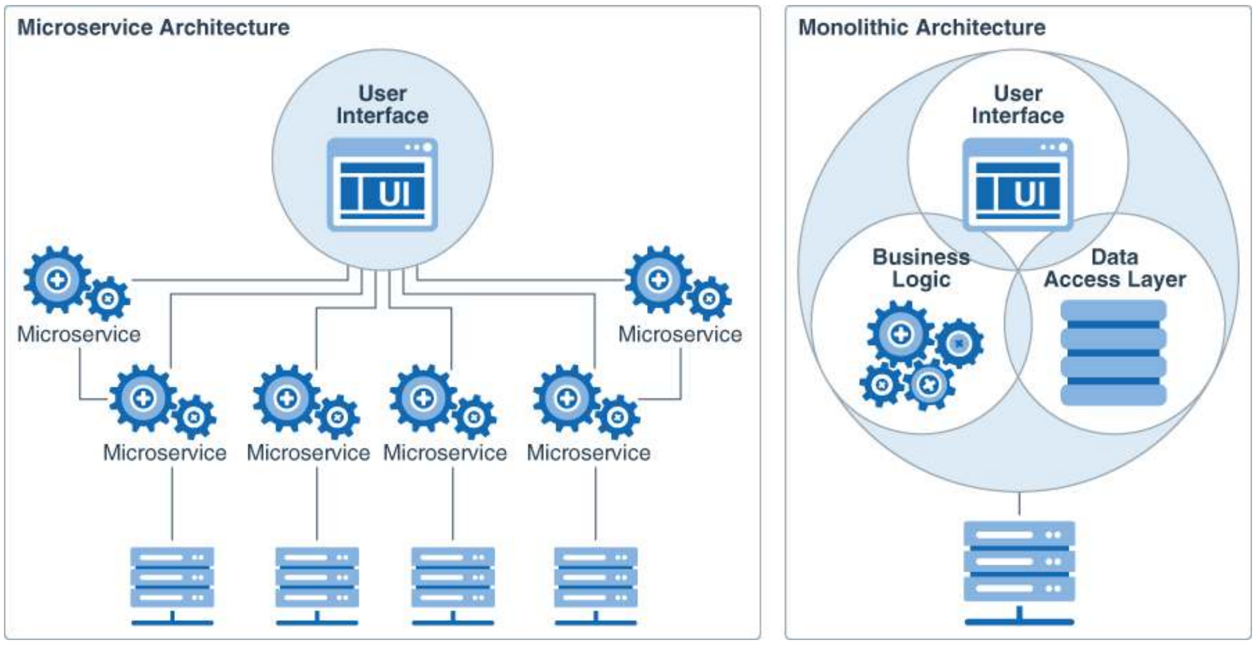

FIGURE 1: Monolithic Architecture versus Microservices Architecture. (image taken from https://docs.oracle.com/en/solutions/learn-architect-microservice)

2.1. Monolithic architecture

If the application to be built is small or the product needs to reach the market fast then probably starting with a monolithic architecture would turn out to be a better choice. Monolithic applications have the following advantages [2, 3]:

- Easy development and testing if the application is relatively small: most enterprise applications have similar layered architecture (presentation layer, integration/service layer, business logic layer, database access layer). Experienced developers and testers could implement the solution easily with such straightforward approach.
- Easy application deployment: there is only one application, which needs to be deployed and configured.
- Easy application scaling: often a load balancer distributes load between several application instances.
- Faster communication between application modules: usually different application modules communicate through procedure calls.

Many of today's successful applications started as monolithic ones. However, when applications grow some disadvantages of the monolithic architecture should be considered [2, 3]:

- If the application becomes too big and complex then it becomes significantly harder for the developers to implement new features or make changes fast.
- Testing becomes much harder because changes may introduce regression defects to any application modules.
- Reliability: a defect in any application module could bring down the whole application.
- Adopting new technologies is very difficult because an update to frameworks or language version will affect the whole application.
- Doing continuous deployment with big monolithic applications is difficult – an update in one module would require a whole application re-deployment, which could be expensive in terms of hardware resources and time.

2.2. Microservice architecture

Microservice architecture splits the application into a set of small, loosely coupled services where each service is responsible for a single business capability. As an illustration, the services for an online shopping application might be: Product Catalog Service, Inventory Service, Orders Service, Delivery Service, User Service, Product Recommendation Service, etc.. Each microservice uses its own database. Although this approach may lead to data duplication, it ensures services' loose coupling. Different services could use different technology stacks – different frameworks, different databases (polyglot persistence [4]) or different programming languages.

Microservices' advantages could be summarized the in the following list [2, 3, 5]:

- The complexity problem of big monolithic applications is solved by decomposing the application into many small services. Each service is developed and maintained by a single dedicated team.
- Easy adoption of new technologies: the small team that maintains the corresponding service is responsible for choosing the implementation technologies.
- Easy and fast deployment: each service can be deployed independently. Continuous deployment is also much easier compared to big monolithic applications.
- Easier scalability: each service can be scaled independently.
- Fault tolerance: the whole application does not crash if one of the services goes down due to a problem.

It is important to list the microservices drawbacks as well [2, 3, 5]:

- Initial implementation cost: microservices architecture works well only for complex applications. Designing and implementing the interconnected services is time consuming at the beginning of the development process and requires experienced architect and team.
- There is a significant communication overhead due to remote calls.
- Business transactions, which need to update multiple business entities have to update different databases owned by different services.
- It is difficult to do changes, which affect multiple services.

In software engineering, design pattern [6] is a good general solution to a common problem. API Gateway and Service Discovery are two of the most commonly used patterns in microservice architecture.

API Gateway

Problem description: implement product details page of online shopping application and show the following information: shopping cart overview, order history, customer reviews, product recommendations, shipping options. If using monolithic architecture then the client will most likely need to make only one call to a REST service, which will gather the required information. However, when using microservices then the following two options exist:

- Call each service directly to get the necessary information. This approach has a number of potential problems:
    - In our simple example the client needs to make small amount of calls. However, for big applications the client might need to call hundreds of services.
    - Some of the services might use different protocols of communication, which will make client implementation harder.
    - Refactoring services becomes immensely more difficult if clients call them directly.
- Use of API Gateway (Fig. 2). API Gateway is similar to Facade [7, 8] object-oriented design pattern. It encapsulates the internal system architecture and provides a specific API to each client. It is a single entry point to the system and often handles incoming request by calling multiple services and aggregating the results. It simplifies the client's code and reduces the round trips between the client and the application [9, 10]. The potential drawbacks of API Gateway are:
    - It is additional application component that needs to be developed, deployed and managed.
    - It may become a development bottleneck in a big system. Developers have to update API Gateway in order to change or expose system functionality. They may be forced to wait in order to make the necessary changes to the API Gateway.

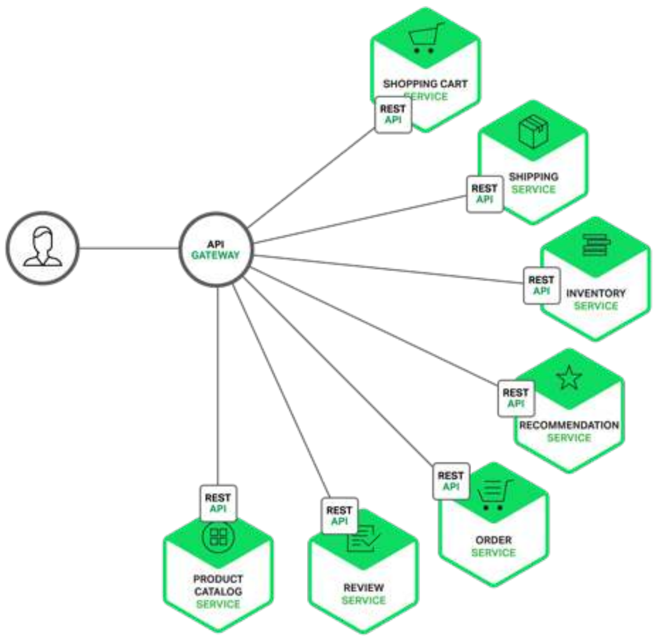

FIGURE 2: API Gateway. (image taken from https://www.nginx.com/blog/building-microservices-using-an-api-gateway/)

Service Discovery

Problem description: in traditional enterprise applications usually the network locations of the different services are static and they can be stored in application configuration files. However, in cloud-based applications the IP addresses of the different services are dynamically assigned and change often.

Service Discovery pattern provides a solution for this problem. It uses a Service Registry where information for all available service endpoints is stored. The Service Registry needs to be highly available and provides management and query APIs. Netflix Eureka [11] or Apache Zookeeper [12] are good choices for Service Registry implementation. There are two service discovery patterns:

- Server-Side Discovery (Fig. 3): clients use load balancer for making requests. The load balancer queries the service registry and routes the requests to the appropriate service instance [10, 13].

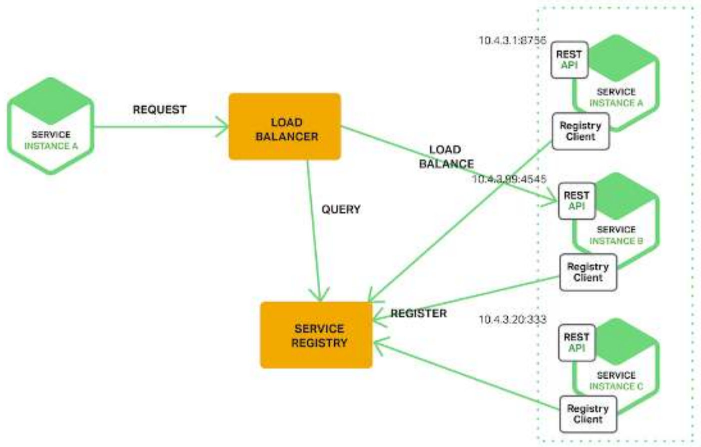

FIGURE 3: Server-Side Discovery Pattern. (image taken from https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/)

- Client-Side Discovery (Fig. 4): clients are responsible for finding the network locations of the available services and for load balancing requests across services. When services start they register their network locations with the Service Registry. By using client-side discovery the clients can implement applications with specific smart load balancing decisions [10, 13].

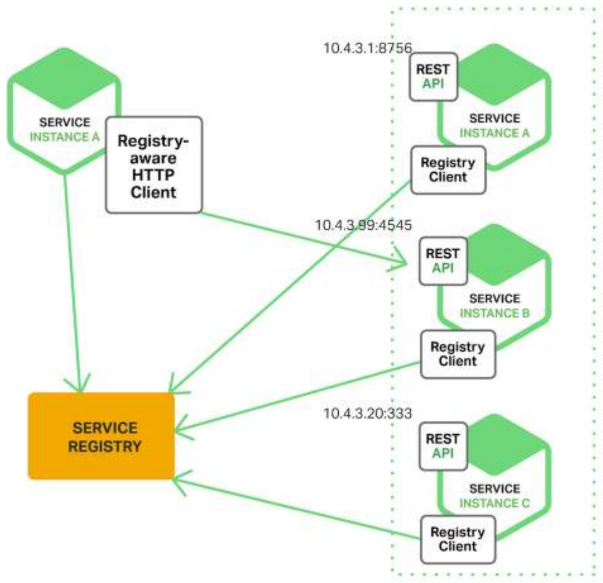

FIGURE 4: Client-Side Discovery Pattern. (image taken from https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/)

3. EVENT DRIVEN ARCHITECTURE

In monolithic applications data management is relatively easy – there is only one database and it is usually relational. When using relational databases, the applications can use ACID (Atomicity, Consistency, Isolation, Durability) model [14, 15] and change multiple rows and tables in one transaction. However, in microservices architecture each service data is private to that service and can only be accessed through the API that the service provides [16]. It introduces the following distributed data major challenges:

- Retrieve data from multiple services.
- Implement business transactions that maintain data consistency across multiple services.

Event-driven architecture (EDA) is a popular distributed asynchronous architecture pattern, which can be used to overcome the distributed data challenges. It is highly scalable and flexible [17, 18]. In EDA each microservice publishes an event when something notable happens, i.e. Order Service would publish new event when order has been created or modified. The other microservices subscribe to the events, which they are interested to (for example Inventory Service would subscribe to new order events because it needs to reduce the appropriate goods count in the inventory database). An event can be defined as "a significant change in state" [19]. Events can be used to implement business transactions that span multiple services. Transactions can be represented by a series of steps where each step is a microservice, which updates or creates a business entity and publishes an event that triggers the next step. EDA has two main topologies – mediator and broker:

- Mediator Topology: useful for events with multiple steps. Mediator topology has four main components: event queues, an event mediator, event channels, and event processors. The event flow starts with a client sending an event to a queue, which transports it to the event mediator. Then the mediator sends additional asynchronous events to event channels in order to execute each step of the process. Event processors listen on the event channels and execute specific business logic to process the event (Fig. 5) [17].

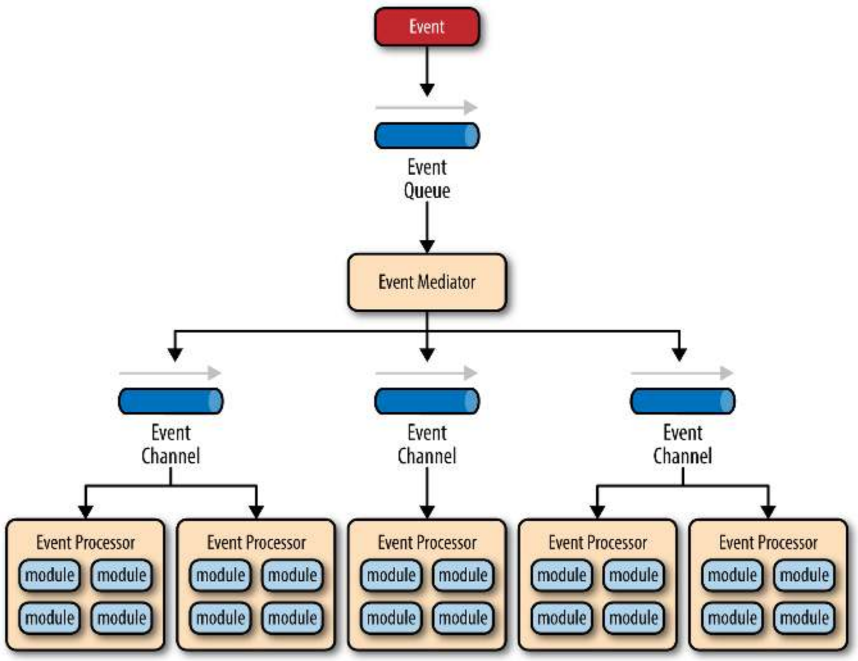

FIGURE 5: Mediator Topology. (Image taken from https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch02.html)

- Broker Topology: in this topology there is no central event mediator to orchestrate the initial event. There are two component types: broker and event processors. Each event processor is responsible for processing an event and publishing a new event to notify others for the performed action. The broker provides the following event channels: message queues, message topics, or a combination of both. The topology is useful when there is a relatively simple event processing flow (Fig. 6) [17].

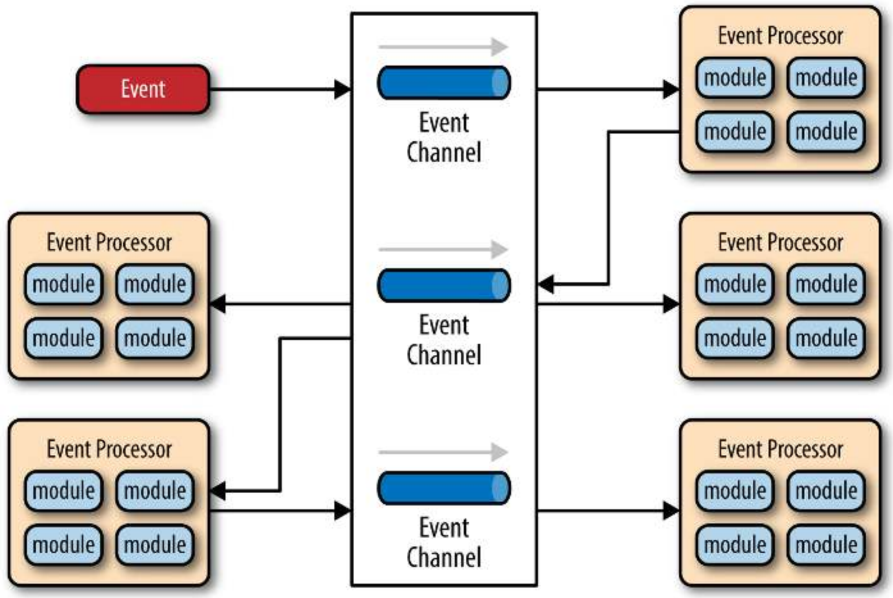

FIGURE 6: Broker Topology. (image taken from https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch02.html)

CONCLUSIONS AND FUTURE WORK

Microservices and event-driven architecture are preferred techniques for implementing modern scalable cloud applications. This architecture is a perfect fit for building a platform for collecting and processing big data streams. Autonomous vehicles will change our lives, just as steam trains and motor cars did before. They will shape the future of road transport and will lead to significantly reduced transport costs. They will pave the way for new services and offer new ways to address the ever-increasing demand for mobility of people and goods. Autonomous vehicles will generate large amount of data which will have an enormous potential to create new and personalized services and products, revolutionize existing business models (e.g. roadside assistance, vehicle insurance, vehicle repair, car rental) or lead to the development of new ones [21].

By 2020 the data created annually will reach 44 zettabytes, or 44 trillion gigabytes [20]. With the advancement of autonomous vehicles even more data is to be expected. Platform architecture that will reduce significantly costs and efforts for the development of future autonomous vehicles management software is proposed and shown in Fig. 7.

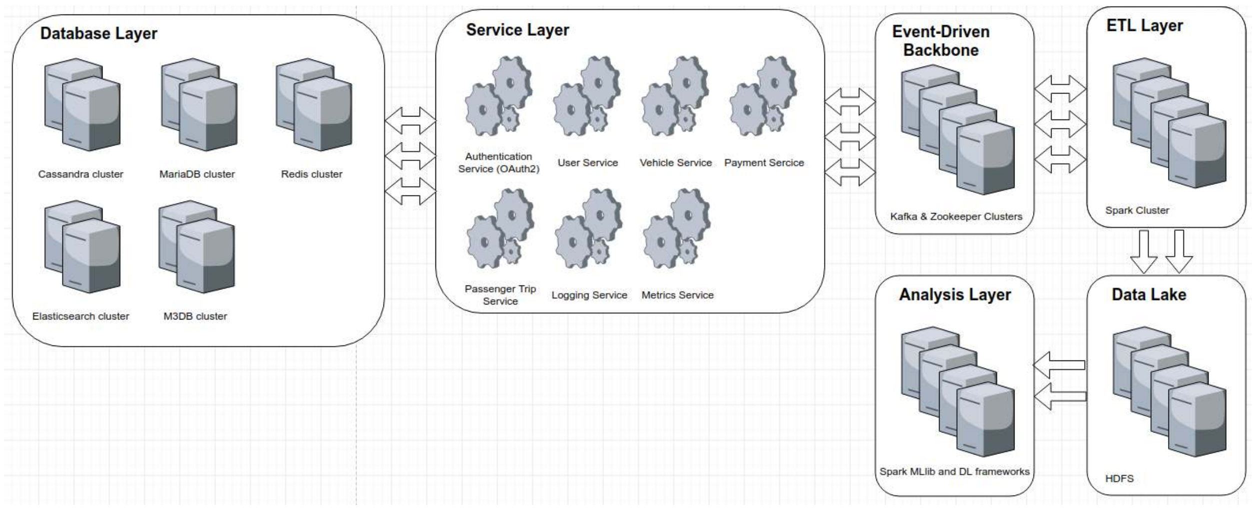

FIGURE 7: Proposed Platform Architecture for big data stream processing

The proposed platform for big data stream processing outlines possible microservices in autonomous vehicles application. Depending on the use cases of such application more or different services may be required. The service database choice is strongly coupled with the service's business requirements. MariaDB [22] or other relational database is a good choice for the Payment Service or other services which operate with financial data. Redis [23] is suitable for storing application cache entries or other temporal data. Cassandra [24] could be used by Vehicle Service for storing sensor data from autonomous vehicles. Elasticsearch [25] can be used by services, which require full text search. M3DB [26] is a distributed time series database, developed by Uber and used by M3 [26] open source metrics platform. Logging and metrics are essential for distributed applications and the use of such platform is strongly recommended. Most likely complex event flows will not be required so broker topology will be sufficient for the application. Kafka [27] is a good choice for event-driven broker. Kafka requires Zookeepr [12], which can also be used for implementation of server side service discovery. Spark [28] is a mature stream processing framework which can be used by both Analysis and ETL layers. HDFS [29] is a common choice for Data Lake implementation.

REFERENCES

1. Microservices, Retrieved 10.06.2019 from https://martinfowler.com/articles/microservices.html
2. Microservices Introduction, Retrieved 10.06.2019 from https://dzone.com/articles/microservices-1-introduction-monolithic-vs-microse
3. Learn About the Microservices Architecture, Retrieved 10.06.2019 from https://docs.oracle.com/en/solutions/learn-architect-microservice/
4. P. J. Sadalage and M. Fowler, NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence, Addison-Wesley Professional, ISBN: 9780133036138 (August 2012)
5. Microservices, Retrieved 10.06.2019 from https://en.wikipedia.org/wiki/Microservices
6. E. Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, ISBN: 978-0201633610, (1995)
7. E. Freeman, B. Bates, K. Sierra and E. Robson, Head First Design Patterns, O'Reilly Media, ISBN-13: 978-0596007126, (June 2014)
8. Facade pattern, Retrieved 10.06.2019 from https://en.wikipedia.org/wiki/Facade_pattern
9. Building Microservices: Using an API Gateway, Retrieved 10.06.2019 from https://www.nginx.com/blog/building-microservices-using-an-api-gateway/
10. C. Richardson, Microservices Patterns, Manning Publications Co, ISBN-13: 978-1617294549, (2019)
11. Eureka, Retrieved 10.06.2019 from https://github.com/Netflix/eureka
12. Apache ZooKeeper, Retrieved 10.06.2019 from https://zookeeper.apache.org/
13. Service Discovery in a Microservices Architecture, Retrieved 10.06.2019 from https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/
14. A. L. Kooijmans, E. Ramos, N. De Greef, D. Delhumeau, D. . Dillenberger, H. Potter and N. Williams, Transaction Processing: Past, Present, and Future, IBM Redguide publication, ISBN-13: 9780738450780, (September 2012)
15. ACID, Retrieved 10.06.2019 from https://en.wikipedia.org/wiki/ACID
16. Event-Driven Data Management for Microservices, Retrieved 10.06.2019 from https://www.nginx.com/blog/event-driven-data-management-microservices/
17. M. Richards, Software Architecture Patterns, O'Reilly Media, ISBN-13: 978-1-491-92424-2, (2015)
18. Event-driven architecture, Retrieved 10.06.2019 from https://en.wikipedia.org/wiki/Event-driven_architecture
19. K. M. Chandy Event-Driven Applications: Costs, Benefits and Design Approaches, California Institute of Technology, (2006)
20. Data Growth, Business Opportunities, and the IT Imperatives, Retrieved 10.06.2019 from https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm
21. On the road to automated mobility: An EU strategy for mobility of the future, Retrieved 10.06.2019 from https://ec.europa.eu/transport/sites/transport/files/3rd-mobility-pack/com20180283_en.pdf
22. MariaDB, Retrieved 10.06.2019 from https://mariadb.org/
23. Redis, Retrieved 10.06.2019 from https://redis.io/
24. Cassandra, Retrieved 10.06.2019 from http://cassandra.apache.org/
25. Elasticsearch, Retrieved 10.06.2019 from https://www.elastic.co/products/elastic-stack
26. M3 and M3DB, Retrieved 10.06.2019 from https://m3db.io/
27. Apache Kafka, Retrieved 10.06.2019 from https://kafka.apache.org/
28. Apache Spark, Retrieved 10.06.2019 from https://spark.apache.org/
29. Hadoop Distributed File System (HDFS), Retrieved 10.06.2019 from http://hadoop.apache.org/