Making the Move to Microservice Architecture

Arne Koschel
Faculty IV
Department of Computer Science
University of Applied Sciences and Arts
Hannover, Germany
akoschel@acm.org

Irina Astrova
Department of Software Science
School of IT
Tallinn University of Technology
Tallinn, Estonia
irina@cs.ioc.ee

Jeremias Dötterl
Faculty IV
Department of Computer Science
University of Applied Sciences and Arts
Hannover, Germany

Abstract—**Microservice architecture is an architectural style suitable for large software projects. The main goals of microservices are flexible on-demand scalability and short release cycles. Microservice architecture decomposes applications into multiple independent components (called microservices), each running in its own process. This sets microservice applications apart from monoliths, which run in a single process. This paper describes the characteristics of both architectures and explores under which circumstances a shift from a monolith towards the more costly microservice architecture is justified. Furthermore, the paper describes (non-)technical challenges that arise in that shift.**

Keywords-monolithycal architecture; microservice architecture; large software projects

I. INTRODUCTION

When the popularity of a distributed software application increases, it has to serve a growing number of clients, thus requiring the application to be designed with scalability in mind. These clients, especially if they are paying customers, often expect regular updates, which fix errors or introduce new features. To meet those expectations, release cycles have to be short and the application has to be maintainable, even when the application is feature-rich and lots of people involved in the development process. To avoid the development speed to slow down, active measures beyond simple code refactoring have to be taken. To keep the quality high, it is desirable to replace whole software parts, instead of just rewriting them, without having to adapt the rest of the application to the replacement. Microservice architecture is a software architecture style, which strives to support all those goals.

Microservice architecture can be seen as a "coming together of a bunch of better practices" [1], which has been covered by several books, articles and papers about service-oriented architecture (SOA), continuous delivery, domain-driven design, etc. However, little has been written about the combination of those practices (called microservice architecture) and their impact on the software projects.

II. MOTIVATING EXAMPLE

Consider a company launching an online shop for its customers. Typical activities of visitors of the online shop are searching for products, buying the products and writing comments on the products. These activities can generate a lot of data that have to be stored persistently and can consume a lot of server resources.

Assume that the company identifies several requirements for the online shop. First of all, this application should support a wide range of different clients. On the one hand, these are web browsers installed on the customers' desktop computers. On the other hand, mobile versions of the most popular web browsers should also be supported by the application because the number of users visiting web pages from the comfort of their mobile devices steadily increases [2]. However, the structure of web pages viewed on the mobile devices can greatly differ from that on the desktop computers.

Next, the application should be maintainable. That is, the architecture of the application should be easy to understand, change and extend. This is especially important if the company wants to operate the application for a long period of time and wants to be able to make changes frequently and rapidly. Furthermore, the company wants to be independent of concrete technologies as much as possible. This is because if the support of a technology stops, the security problem can occur. To avoid this problem, the architecture should enable to replace the old technology with a new one as easy as possible. In addition, this replacement should be inexpensive so that the company can experiment with new and innovative solutions.

Since the online shop has many visitors and expects a further increase in the number of visitors, the application should be scalable. That is, it should be able to cope with an increased number of requests. Furthermore, high availability – 24 hours a day, 7 days a week and 365 days a year – is required. In the case of the online shop, where web pages are the only source of income, every minute of unavailability of a web page is expensive. While the page is unavailable, no product can be sold. If failures become frequent, customers can lose their trust into the company or get frustrated, which lets them switch to competitors. First-time visitors (prospective customers) who cannot reach a web page will probably not come back. This makes high availability an important requirement too.

Assume that the company searches for an architectural style that meets all those requirements. One possible solution could be monolithic architecture.

978-1-908320/86/5/$31.00 © 2017 IEEE

III. MONOLITHIC ARCHITECTURE

Figure 1 gives an overview of the monolithic architecture. On the left side, different types of clients are shown that want to interact with the application. The application itself (shown in the center) runs on the server-side in a single process as the whole "monolith". All modules of the application use the same shared database (e.g., a relational database), which is shown on the right side.

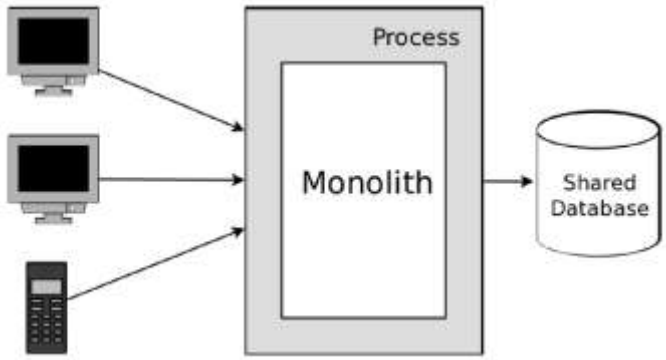

Figure 1. Monolithic architecture.

The monolithic architecture has several advantages. The application is easy to develop. Modules within the application are built with classes or packages; they communicate with each other via method calls, which are reliable and have a very little delay. In addition, the application is easy to deploy. There is only a single artefact that has to be deployed to the server. Scaling the application is also easy (see Figure 2). If one instance of the application is not capable handling all the requests on its own, copies of the application can run on multiple servers. A load balancer is responsible for routing the incoming client request to the different servers. For the clients, this process is transparent as they do not have to know which instance of the application they are talking to.

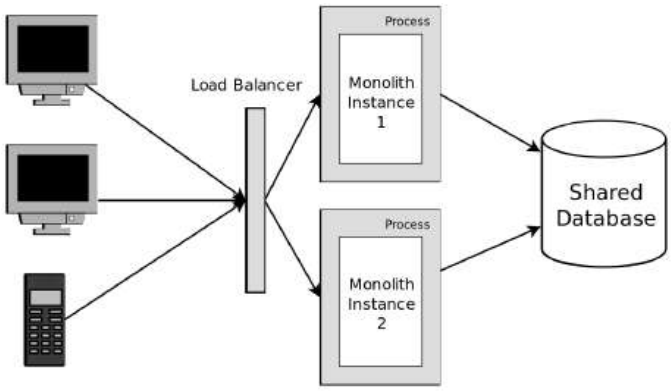

Figure 2. Scaling of monolithic architecture.

However, the monolithic architecture becomes problematic if the application gets too big. "Big" can mean a high number of lines of code or a high number of people involved in the development of the application. Often the former implies the later – if the application consists of many lines of code, lots of people are necessary to develop the application. Both dimensions of "big" can cause problems. First of all, large software projects tend to slow down in the development speed [3] and new features get shipped less frequently. This is because the lines of code start to depend on each other, whereas changes and enhancements get more difficult. Moreover, the more people are involved, the more communication is necessary. As a result, time spent for communication cannot be spent on adding new features.

In our example, the company wants to ship new features, changes and enhancements rapidly. But not only does the decreased productivity contradict this goal of short release cycles. Even if the development can take place with acceptable progress, the code has to be deployed first to become visible to the customers. In the monolithic architecture, the change of a single line requires a re-deployment of the whole application. Because deployments are risky and can cause problems, frequent deployments are often avoided. This means that even more changes get shipped with a next deployment, which makes it even riskier and more problematic.

In the monolith, components can share a database or database cluster. But a shared data store implies that different teams have to agree on a shared data schema. This is problematic because changes can affect potentially all other components that use the same data store. For this reason strong dependencies are formed and high communication effort between all teams is required. For example, if a team responsible for the business logic wants to make a change in the database, this change often has to be coordinated with the user interface (UI) team.

Microservice architecture strives to solve the problems described above.

IV. MICROSERVICE ARCHITECTURE

Figure 3 gives an overview of the microservice architecture. Here the application is decomposed into multiple components called microservices, each of them providing a limited set of functionality. For example, in the case of the online shop, there can be a microservice that handles searching for products and another microservice that handles buy requests. Each microservice runs in its own process and can be deployed and scaled independently.

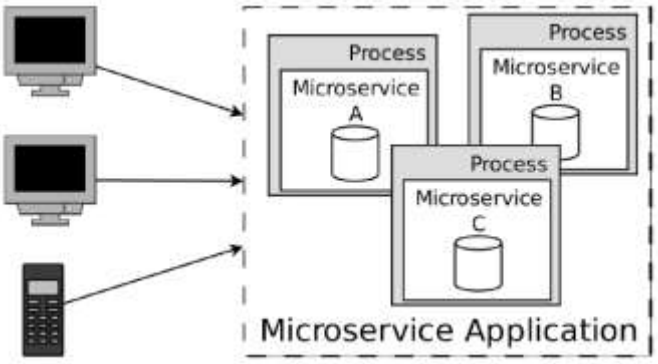

Figure 3. Microservice architecture.

Every microservice has a team assigned to it; this team is fully responsible for the microservice. For this reason, teams

are no longer formed regarding technical expertise (as it was done in the monolithic architecture). For example, there is neither a team solely responsible for the UI nor a team solely responsible for the business logic. Rather, every team has its own experts for different tasks like UI and business logic so that UI and business logic experts work in the same team. As a result, a change within a microservice can now be made mostly independent of other teams. Even more, the team is not only responsible for developing but also for operating the microservice. One microservice stays in the same team for its whole life cycle and does not get transferred to other teams. This way the microservice architecture strives to reduce the communication effort between teams as much as possible. While teams can make decisions within their microservices rather autonomously, there still has to be a shared architectural vision.

Obviously, for the online shop, the microservice architecture should be preferred to the monolith. However, making the move from the monolith towards the microservice architecture is a challenging task.

V. Moving to Microservice Architecture

There are a number of challenges project teams (especially, architects) commonly face when moving to the microservice architecture. Therefore, every team should at least spend some time thinking about those challenges and decide how the challenges need to be addressed.

A. Decomposition

The term microservice suggests that those components are rather small but there is no definition how small a microservice should be. Of course, they should not be too small because then the overhead that is necessary to operate a microservice would be greater than its benefits. On the other hand, microservices should be so small that they can be replaced easily with little costs. While in the monolithic architecture refactoring is applied to make sure the initial architectural vision keeps being followed, the microservice architecture provides a more radical approach against code rotting. Whenever plain refactoring seems not to be sufficient, the implementation of a microservice is replaced completely with a new one. This is possible without other microservices noticing as long as the interface keeps unchanged.

B. Deployment

The microservice architecture enables to deploy components independent of each other, and to publish features and changes rapidly. Since deployment is not executed for a single monolith every once a while, but frequently and individually for different microservices, the deployment process should be automated. Deployment also should be as easy as possible. Otherwise, it will not be executed regularly and the goal of short release cycles will not be reached. Additionally, the deployment should work reliably and without any errors.

When designing the interfaces one should also keep in mind that microservices should be coupled loosely. Changes in one microservice should ideally require no changes in other microservices. This guarantees that microservices can be deployed independently. It is not necessary to deploy all microservices at once if the deployment of one microservice does not require the deployment of a new version of another microservice.

C. Technology Heterogeneity

Because microservices should be as independent of each other, the microservice architecture strives to minimize the interfaces and dependencies as much as possible. The application works with data that have to be stored persistently. Every microservice has its own data store that the other microservices have no access to. If another microservice wants to get access to the data, it has to request the data via the public interface of the microservice that owns the data. This has an advantage that every team can change the internal schema as long as the public interface keeps stable. A further advantage is that it enables to exploit polyglot persistence – different database paradigms can be used. For every microservice, the paradigm can be chosen that fits best to the concrete use case. For example, the online shop might use key-value stores for the virtual shopping carts and relational databases for storing the products.

Not only different database systems can be used but it is also possible that every microservice uses a different programming language or framework. This works as long as all the languages support the used communication technology. From the outside, each microservice can be seen as a black box. Other microservices do not have to know which programming language the implementation uses to provide a certain service. However, this freedom should be used carefully. If it is overused, problems will occur. The complexity of the application increases and transferring developers between teams becomes more difficult. Team members may not be able to switch to another team because they may not have the necessary knowledge about the technology the other team uses. Moreover, if a microservice is implemented in a rarely used language that only a small number of employees are able to productively write with, the company becomes dependent of those employees. If the employees leave the company, the microservice cannot be longer maintained and developed and new employees have to be found that take over these tasks. Such a problem can be avoided if a software project restricts itself to a fixed set of programming languages and frameworks.

D. Scalability

A central aspect of microservices is that every microservice runs in its own process. This has an important advantage. As desired, the components can now be scaled independently. A microservice that experiences high load can run on more servers than a microservice that experiences low load. Furthermore, if one of the servers hosting the high-load microservice fails, the other one is still up and can serve incoming requests. In the case of the online shop, a microservice that lets visitors view a product needs probably more scaling than a microservice that lets visitors write a comment on the product.

Figure 4 illustrates examples of different scaling options in microservice applications. Assume that A, B and C represent different microservices. In the first example, all microservices run on the same machine. In the second example, microservices B and C still run on the same server but a microservice A runs on a different one. The purpose might be that the microservice A requires many resources that are not available if it shares the resources with B and C as it was in the first example. The third example is an extended version of the second one. There are now two instances of the microservices A and C. The machine hosting the microservice C might be less powerful and cheaper than the machine hosting the microservices B and C. The scenario shown in the third example has another advantage compared to the first and second examples: the microservices A and C are now better protected against server failures.

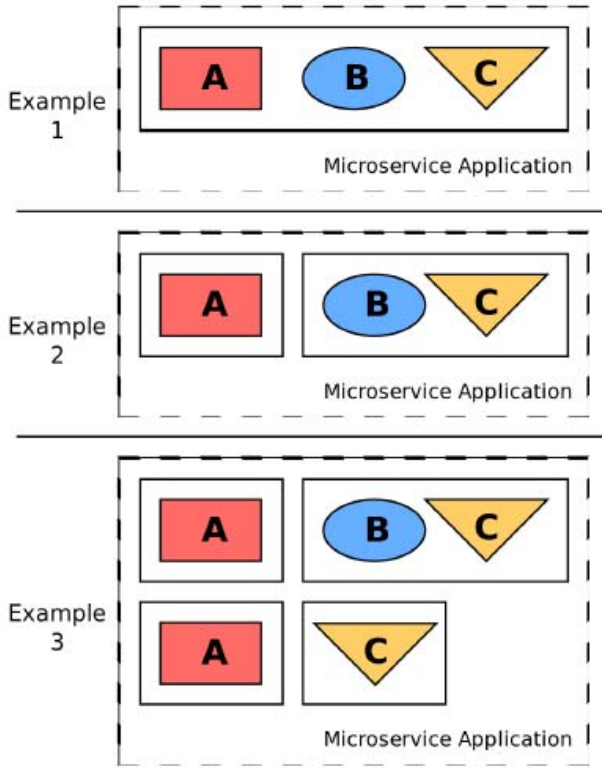

Figure 4. Scaling of microservices.

However, scalability comes with costs – communication between microservices becomes more complex. It is no longer possible for components to communicate with each other via simple method calls. Instead, inter-process communication mechanisms are required. This has impact on how to design the interfaces. Method calls are fast and can be made often without running into any problems. But remote calls are expensive and have high latency compared to simple method calls. For this reason interfaces should not be designed too fine-grained. With remote calls one of the primary goals is to reduce the number of necessary calls.

E. Client-Server Interaction

So far we looked at the microservice architecture as a collection of independent microservices. But how does a client access the services that the collection of microservices provides? Users want to interact via a UI. Users expect their actions resulting in quick responses, even though many microservices may be involved in generating the result. Different possibilities exist how control and data flow between front-end and back-end can be realized.

The first option is *Application Programming Interface (API) composition* [4]. Client applications directly access the fine-grained interfaces of microservices. If a microservice provides, e.g., a JSON REST API, client applications will pull the JSON data and process it to display it via the UI. However, this solution has some disadvantages.

All types of clients have to use the same interface. The interface is typically not adjusted to the different requirements of the different devices. This means that a mobile client may have to pull as much data as a desktop client even though it will not display so much information. Mobile clients should also be especially economical because unnecessary tasks consume battery power and may result in costs for data usage. Furthermore, it is possible that there are many microservices involved. The client would have to send requests to every participating microservice. Furthermore, this solution requires that there should be a special UI team that is responsible for pulling the data and visualizing it. Changes are only possible if the UI team and the microservice team coordinate the change.

Since it is possible to use different programming languages and technologies in different microservices, the APIs have to be technology-agnostic [4]. Even if in the beginning just one language is used, it is good not to constrain oneself to such a decision. Technology diversity is an important feature of microservice architecture, which should not be ruled out from the beginning. Microservices themselves are designed to be replaceable, but this is not so much true for the communication links between them. Therefore, the project team should start with a technology-agnostic API from the beginning.

If a project wants to organize its team so that every microservice team contains people responsible for the UI, an option called *User Interface (UI) fragment composition* [4] can be used. Instead of just delivering data in form of, e.g., JSON, a microservice could answer requests with UI fragments that are already assembled. A web client would no longer process the received data and assemble the web page itself, but simply include the received HTML components into the web page. The representation of the response of a microservice would be the responsibility of the microservice team. This raises the question how a consistent user experience can be achieved with this approach. Users do not want to see that different parts of the UI originate from different microservices. The problem can be reduced with style guides and an asset server that hosts shared images and CSS styles. A further problem is native apps on mobile devices, which do not use HTML, but native widgets. Microservices cannot serve native widgets. These devices would have to fall back to API composition, pulling the data first and then using it to assemble the UI. Also, UI Fragment Composition still requires calls to all involved

microservices. This last issue can be solved with an API gateway.

An *Application Programming Interface (API) gateway* [4] provides interfaces that are more coarse-grained than those of the microservices. The gateway talks to the fine-grained interfaces of the different microservices and provides an aggregated answer to the client. The clients send their requests to the gateway that collects the responses of the individual microservices, instead of talking to all the microservices directly. This reduces the required communication effort for the clients. A project can use multiple API gateways, one for each client type. This way each gateway can be adapted to the special requirements of each client type. However, the solution can cause problems if the API gateway becomes too large and complex. The isolation can vanish if several teams work on this gateway. Microservices can lose their independence and then cannot be deployed individually any longer.

F. Inter-Process Communication between Microservices

Applications based on microservice architecture are decomposed into multiple components called microservices, each of them providing a limited set of functionality. Each microservice runs in its own process and can be deployed independently. Microservices have to communicate with each other to work as a whole. Since they are running in different processes, in-process method calls are not an option. Rather some kind of inter-process communication mechanism has to be used.

The first option is *request-response* (sometimes also called *request-reply*). If one microservice wants another microservice to run a certain task and wants to get an immediate response, this method is a natural fit. More specifically, a microservice provides functionality via an API. A microservice that wants to use this functionality sends a request to this API and receives an immediate response (see Figure 5).

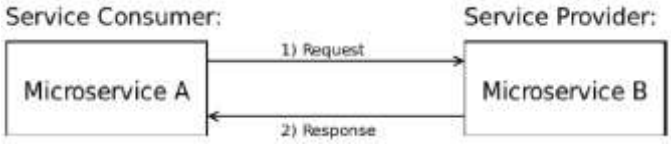

Figure 5. Request-response method.

A popular style, which is often used to realize this type of communication, is REST (Representational State Transfer). REST is an architectural style for creating web services. RESTful systems mostly communicate over HTTP. In this case, the request would be an HTTP request and the response would be a resource represented as a document in, e.g., JSON. There are many frameworks and libraries that let the project team provide and consume RESTful APIs.

The request-response method is simple and intuitive but has its limits. Sometimes a microservice wants to get informed about events taking place in another microservice. Using the request-response method, the microservices would have to regularly poll the other microservice for its state. In large scale systems, this is usually not an option. Or the microservice in which the event is taking place has to call all microservices that are interested in this kind of events explicitly. But this leads to coupling between the microservices.

Publish-subscribe is a method for decoupling the service provider and the service consumer. A microservice A, which is interested in events of type E, subscribes at a message broker for events of type E. A microservice B, which publishes E, notifies the broker about it. The broker then notifies all subscribers, which are interested in E, that E has taken place (see Figure 6).

The publisher and the subscriber do not have to know about each other – they are decoupled in the space dimension [5]. In fact, there can be an arbitrary number of subscribers as well as an arbitrary number of publishers. Subscribers do not know which and how many publishers exist while publishers do not know which and how many subscribers exist. Furthermore, publishers and subscribers can also be decoupled in the time dimension [5]. That is, the publisher can publish events while the subscriber is disconnected and the subscriber can be notified about an event while the publisher that created the event is disconnected. Unlike to the request-response method, in the publish-subscribe method no polling is necessary if one microservice wants to get notified about events. But this also means event notifications can arrive at any time and the subscriber has to be capable to handle this. This is usually more complex than the simple request-response solution because it introduces asynchrony.

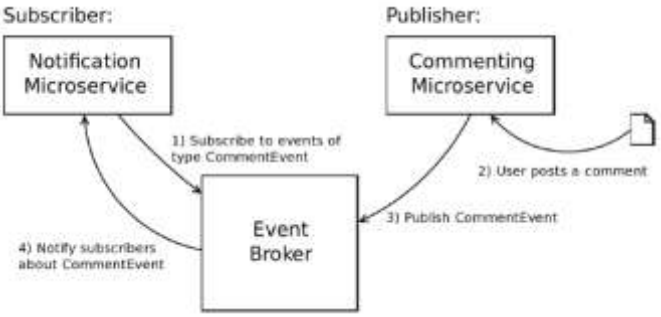

Figure 6. Publish-subscribe method.

G. Monitoring

Monitoring also plays a big role in the microservice architecture because of the distributed nature of the application. In particular, each microservice runs in its own process. These processes usually do not all run on the same physical machine. Instead, often several machines are operated. If the application fails in some way or performs poorly, it might not be obvious which machine is causing the problem. To find the responsible hardware where to look for the error and to get alerted if a critical operation goes wrong a monitoring infrastructure should be established. Therefore, monitoring is one of the tasks in the microservice architecture, which should be standardized – all metrics, regardless which microservices provide those metrics, should be accessible in one place and under a single name [4].

Metrics can be monitored on both system and application levels. System-level metrics measure the behavior of the hardware (the application is running on) and include CPU load

and memory consumption. Application-level metrics measure the behavior of the application and include response times and error rates. Typically, in the microservice architecture both types of metrics should be monitored but not all monitoring frameworks support the two. This fact should be considered when choosing a monitoring system.

Moreover, it should also be kept in mind that a monitoring system itself can fail. Therefore, a failure strategy should be established especially for monitoring systems that are based on single server architecture. A fault on the primary (or single) monitoring server should not result in an unmonitored application environment.

VI. CONCLUSION

This paper explained the simpler and less costly monolithic architecture and its limitations in large software projects, the projects with many lines of code and many developers. To solve the problems that can arise with monoliths, the microservice architecture was introduced. Microservices run in their own processes and form independent deployment units. Each of them owns its own data store. Microservices enable independent scaling and short release cycles.

Microservices have powerful properties, but come with high costs that can outweigh their benefits in small and medium sized software projects.

ACKNOWLEDGMENT

Irina Astrova's work was supported by the Estonian Ministry of Education and Research institutional research grant IUT33-13.

REFERENCES

[1] J. Thönes, "Microservices," IEEE Software, January/February 2015.

[2] "We Are Social, Percentage of all global web pages served to mobile phones from 2009 to 2015," 2015, (Accessed: 2017-03-08). [Online]. Available: http://statista.com/statistics/241462/global-phone-website-traffic-share/ (Access Date: 9 July, 2017)

[3] F. Brooks, Jr, The mythical man-month: essays on software engineering. Addison-Wesley, 1995.

[4] S. Newman, "Building Microservices - Designing Fine-Grained Systems," Sebastopol, CA: O'Reilly, 2015.

[5] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," in ACM Computing Surveys, vol. 35, no. 2, June 2003, pp. 114–131.