Serdica J. Computing **11** (2017), No 2, 115–136

Serdica
Journal of Computing

Bulgarian Academy of Sciences
Institute of Mathematics and Informatics

AN OVERVIEW OF SELF-ADAPTIVE TECHNIQUES FOR MICROSERVICE ARCHITECTURES*

Krasimir Baylov, Aleksandar Dimov

Abstract. Contemporary software systems are continuously growing in size and a large number of users need to deal with new class of problems —complexity and evolution. To overcome this, new technologies and methods in software engineering emerge. One of them is the architectural style of microservices. It tends to provide solutions, however it introduces additional complexity in terms of administration, detecting fault behavior and applying fixes. Self-adaptive systems address the problems of complexity and evolution by providing mechanisms that allow systems to respond to external environmental changes without human interaction. Currently, there is a lack of understanding on how microservices can utilize the notion of self-adaptiveness and in this paper we make an overview of the current solutions in the field.

1. Introduction. The constant evolution of software systems, along with the advance of cloud capabilities, has led to evolution of the existing

ACM Computing Classification System (1998): C.2.4, D.2.11.

Key words: self-adaptive systems, microservices, software architecture.

*The research presented in this paper was partially supported by the DFNI I02-2/2014 project, funded by the National Science Fund, Ministry of Education and Science in Bulgaria.

architectural styles. Companies need to respond quickly to the constantly changing business requirements and they need to do it in a reliable and consistent way. In the past, software systems used to be deployed as a single package or component called monolith. However, this causes many problems such as slow system evolution, low scalability, dependent component development, technology homogeneity, etc. This leads to a new way of designing and deploying software systems. Microservices is an architectural style that addresses those limitations. Microservices are "small, autonomous services that work together" [1]. Contrary to the monolith approach, microservice systems are split into multiple independent and autonomous services that exchange information over hyper-text transfer protocol (HTTP). This allows developers to pick the most suitable technology stack and work independently on each of those services. However, everything comes at a cost. Despite the many benefits of microservices, they introduce a lot of operational challenges. The large number of services requires that all procedures for testing, deployment, monitoring, etc. are automated. This inevitably increases the complexity of such systems. New methods and tools need to be introduced in order to reduce the complexity and keep it at manageable levels. Such methods should exclude humans from the loop and let services adapt and manage themselves.

Self-adaptive systems provide a solution to this problem. Such systems can manage themselves following high-level goals provided by administrators [2]. They are all based on feedback/control loops [3]. Such loops allow gathering information on the external environment, analyzing it, making decisions for future actions and executing those decisions. Self-adaptation provides a huge potential in the microservices architectural style. Although there are industry tools that provide partial support with self-adaptation capabilities, they are only focused on a specific aspect of the problem—for example load balancing, container orchestration and monitoring, etc.

In this paper we provide a detailed overview of the field of self-adaptive microservice architectures. We analyze the key properties of microservice architectures and the points that need to be improved regarding their self-adaptation capabilities. We also point out how self-adaptive mechanisms can complement microservices to keep their complexity at acceptable levels.

The rest of this paper is organized as follows: Section 2 describes the framework of our research; Section 3 and 4 present the key characteristics of microservices.

2. Systematic overview. As part of our overview, we have been following a systematic approach to selecting the papers that are related to the field of self-adaptive microservice architectures. To do so, we have searched the following sources for information:

- **ACM** (`http://dl.acm.org`)—a digital library that provides access to research papers published at ACM (Association of Computing Machinery);
- **Google Scholar** (`https://scholar.google.com`)—Google's search engine for scientific papers, journals, books, etc.

We have searched for papers using a set of key words in their titles. Once the papers were identified, we reviewed the abstracts to determine to what extent they relate to our research. Below, we present the result of the initial systematic selection of papers. We provide the filters we have searched with and the number of papers that matched the criteria.

Table 1. Selection of papers

	ACM	**Google Scholar**
Self-adaptive AND architecture	594	44800
Self-adaptive AND SOA	10	3720
Self-adaptive AND microservice	1	75
Autonomous AND architecture	2604	2010000
Autonomous AND microservice	4	327

The identified papers were further filtered by reviewing their full title and the abstracts. Where the results exceeded 1000 we reviewed the first 1000 starting from the most recent ones. Once the initial set of papers was selected we read their entire content and selected more papers based on the references

they provided. After reading the full paper we finally decided whether it is relevant to our research or not. As a result we reduced the number of papers that were fully reviewed and analyzed to 19.

3. Key characteristics of microservices.

3.1. Monolith approach. Traditional approaches for building big and complex systems are based on packaging the entire (or very big parts of the) system in a single component, called monolith. This is referred to as the "monolith approach" and it has several limitations [4].

- **Hard to understand and modify**—as applications grow bigger and bigger, people find it hard to understand their entire design. Large applications imply that there are multiple dependencies which developers have to trace and analyze before applying any changes.
- **Decreased productivity because of the large code base**—Developers cannot work independently because of application size. Large applications require significant amount of communication when coordinating any development, test or deployment activities.
- **Continuous development/deployment is very difficult**—Updating a single component would require redeploying the entire application.
- **Scaling is difficult**—Scaling can be achieved by running multiple copies of the application, but this can only guarantee increased transaction volume processing. However, scaling in other dimensions like data volume processing is extremely hard. Another problem with scaling is that in large applications usually few components need to scale. What's more, different components may require different type of scaling—some may be CPU intensive, other memory intensive, network intensive, etc. With monolith architectures everything scales at the same time, which results in loss of efficiency and money.
- **Tied to a technology stack**—Monolith architectures are built around a technology stack. The larger the systems get, the harder it is to change the technologies because of the significant efforts required in migration and refactoring.

3.2. Microservices. Microservices is an architectural style that tries to overcome many of the limitations implied by large monolithic applications. However the community lacks a common understanding on the concepts of microservices and microservice architectures. For example, James Lewis and Martin Fowler propose the following definition [5]:

> *The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.*

Sam Newman in his book *Building Microservices* states [1]:

> *Microservices are small, autonomous services that work together.*

Adiran Dockroft refers to microservices as:

> *Service-oriented architecture composed of loosely coupled elements that have bounded contexts.*

Mark Little, however, disagrees with the above definitions by arguing that microservices are just another term for SOA [5]. The lack of unified and precise definitions for microservices may be something normal, especially considering the fact that this is a relatively new pattern that has yet to be formalized. Therefore, it is more reasonable to think of microservices in terms of the common characteristics that they have. In the next sections we cover some of the key characteristics and benefits of using this architectural style.

For the purpose of our research we define microservices as follows:

> *The microservice architectural style is service-oriented architecture composed of a suite of small services, that work together and communicating over lightweight mechanisms like HTTP resource API. Services are built around business capabilities, independently*

deployable, and can be implemented using different technology stacks.

So defined, microservices have both advantages and drawbacks [6]. On one hand, their small size increases developers' productivity. Services can be deployed and scaled independently, which also improves their fault isolation. Additionally, developers are not tied to a specific technology stack—they can pick the most suitable technology for their services. On the other hand, microservices introduce additional complexity of distributed systems. Compared to monolithic applications, they are harder to operate and team coordination becomes more complicated. Transaction management [8] is hard to achieve in such distributed systems. The large number of services makes reporting and data aggregation extremely complex [1]. Last but not least, microservices come with increased initial cost.

4. Microservice characteristics. Microservices is an architectural style and comes with a set of common characteristics that are identical regardless of the projects they are applied in. Although not all microservice architectures will have all of the presented characteristics, it is expected that all architectures will have most of these characteristics. James Lewis and Martin Fowler have come up with a list of 9 characteristics they describe in their article [7]. Below we present some of the key characteristics or microservices. This is not a complete list and depending on the perspective or business context some may be removed or altered, or others added.

4.1. Service componentization [5]. Service componentization refers to the degree to which microservices can be independently replaced and updated. Components are software units that are both independently replaceable and independently upgradeable [9]. This means that they can be deployed independently from other services. For example, suppose that we have an environment with 10 services. Using service componentization, each of those services can be replaced or updated without affecting any of the other services. In addition, all other services should continue to operate while we update/replace the selected one.

4.2. Organization around business capabilities [5]. Microservices support business capabilities rather than individual components of the system. Teams are responsible for working on the individual features of the applications, regardless of the number of components that are affected. In other words, if such a team needs to develop a "shopping cart" functionality for their portal, they would build the code for all layers—starting from user interface to the database.

Note that Conway's law [10] is valid in both cases for dividing the teams:

> *Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.*

4.3. Product development (not project development) [5]. Microservices are best suited for product development. The team that has developed the system takes responsibility for supporting and evolving it, too. Unlike project development, which has a defined duration and budget, development teams are the owners of the systems until they are decommissioned and not supported anymore.

4.4. Smart endpoints and dumb pipes [5]. Microservices rely on lightweight protocols for information exchange. They avoid complex middleware like Enterprise Service Bus (ESB) that make message translation and routing. This lays a solid foundation for using service choreography. It is achieved by using lightweight REST protocols. Dumb pipes play a key role in composing services into more complex structures. Every service is responsible for receiving requests and producing a response, therefore they can be combined with pipes (UNIX-style).

4.5. Decentralized governance. Decentralized governance refers to the ability of using a wide set of processes, methodologies and technology stacks within the services that comprise an IT system.

IT governance addresses the definition of implementation of processes, structures and relational mechanisms in the organization in order to allow business and IT staff to easily execute their responsibilities to support business

and IT alignment. Although the decentralized governance gives a lot of freedom, teams should be careful with the technologies they select. A big number of different technologies may negatively impact the support capabilities of the teams. Therefore, some companies prefer to limit the technology options into a certain range.

4.6. Decentralized data management [5]. Decentralized data management refers to the ability of splitting the data so that each service is responsible for its own data. This allows using different technologies for storing and processing the data. Multiple services may operate with the same entities (like customers, billing, shipping, etc.) and therefore special attention should be given to data modelling. Data modelling refers to the way that entities are modeled, their relationships and constraints. Another factor for modelling the data is that different services may have a different view or understanding for those shared entities. For example, the finance, sales and support department may have different views on the customer entity. This brings up the notion of bounded context used in the Domain-Driven Design (DDD) [11]. DDD focuses on splitting a domain into a set of bounded contexts that relate to each other.

Another effect of data decentralization is using separate stores for each service. This means that each service is responsible for its own data. A great benefit of this separation is that services can also use different technologies to store their data depending on the specific operations that they support. For some services it may be more suitable to use relational databases, for others a document or graph database. Any communication, however, should be done only though the services application programming interfaces (API). Database communication is not allowed except in some cases where explicit design decisions are taken in this direction.

4.7. Automated infrastructure [5]. Microservices rely on the DevOps [12] culture by adopting nearly full automation in terms of deployment, testing and monitoring. Teams that build microservices have vast experience with Continuous Integration (CI) and Continuous Delivery (CD). Today many of the cloud infrastructure platforms provide solid support for CD, which makes them a preferred choice for deployment infrastructure by the microservice teams.

4.8. Design for failure [5]. Microservices add an additional layer of complexity compared to monolith deployments because of their distributed nature. This results in multiple points of potential failures. In fact, every service is likely to fail at any time. There are also many additional factors that may cause failures at any point—delay in response, network outages, slow network infrastructure, undelivered messages, etc. They are something that goes along with distributed systems and the only way to counteract is to make designs that incorporate failures.

Inevitably, this brings us to the point of service monitoring. With the growing number of services it gets more important to detect failures as quickly as possible and take appropriate actions. Some companies use separate systems that perform external monitoring and provide instant status of the deployed services.

4.9. Evolutionary design [5]. Microservices allow rapid development of new features and obtaining user feedback very quickly. This allows companies to respond to marked needs quickly and constantly evolve their products and services. One of the main reasons for distinguishing microservices as a separate architectural style is the need of systems evolution. Business requirements are always changing and the reason for this is the dynamic market. Therefore, the microservices architectural style put evolutionary design in the core of its principles and characteristics.

4.10. Automated service monitoring. Microservices rely on automated services monitoring to detect anomalies in the behavior of services failures. Automated service monitoring can detect any service failures immediately. Even more, any deviations from the established Service Level Agreements (SLA) can be detected within reasonable time, allowing developers to take timely actions before incidents are reported.

Service monitoring can be done as a part of the service implementation or as a separate external process that collects performance data independently. Such data can be visualized in dashboards that provide a general overview of the overall health of the system.

4.11. Summary of the characteristics. The aforementioned characteristics provide a solid foundation for introducing self-adaptation capabilities

in microservice architectures. Service componentization allows for such capabilities to be applied independently per service. This way, each service would take advantage of the adaptations it needs. Automated infrastructure and service monitoring can provide a sufficient amount of data and connection endpoints to interfere with the services once failures are detected.

Other characteristics point to some of the microservices pains that self-adaptive systems can improve. Designing for failure can be addressed to determine network and component failures and provide self-healing mechanisms for the services.

5. Self-adaptive software systems. The concept of services (and microservices, especially) introduces a new layer of complexity that developers and administrators have to deal with. They operate in an environment that is highly distributed and there are multiple points of failure. Service-based systems need to be constantly monitored and whenever a failure or major deviation from the agreed SLA is detected, corrective action should be taken. This introduces the need of design mechanisms that help in reducing system complexity and support the systems in adapting themselves at runtime. Such systems are called self-adaptive systems.

Self-adaptive systems are types of systems that can manage themselves without direct intervention of humans. The concept of self-adaptation, however, is not, however, a new one. Its origins go back to 2001 when IBM's senior vice president of research, Paul Horn, introduced the idea of autonomic computing—systems that are able to manage themselves given some high level objectives [2].

In the literature self-adaptive systems can be named in various ways. Authors usually use the terms self-adaptive, autonomic computing and self-managing interchangeably.

5.1. Software complexity. Although self-adaptive systems could be useful in solving multiple problems, they initially emerged as a cure for one single major issue—complexity. In 2001 complexity was recognized as one of the main obstacles for further progress in the IT industry [2]. Today software systems are even more complex. Software engineers need broader and deeper knowledge to perform basic technology operations. Therefore, reducing

complexity is one of the main driving forces for the development of self-adaptive software systems [10].

5.2. Self-managing aspects. Self-Adaptive systems can manage themselves in different directions. We would call them self-managing aspects. They come in four major flavors [2]:

- **Self-configuration**—components and systems can configure themselves following high level policies. Such systems would change their configuration parameters when starting or while operating, so that they comply with initially specified goals.
- **Self-optimization**—components and systems continuously monitor themselves and try to identify opportunities for improvement. For example, such components could constantly check for updates and apply then once they are released.
- **Self-healing**—components and systems automatically detect and repair problems. Once the system has identified the failure, it would search for and apply software patches and retest itself.
- **Self-protecting**—components and systems protect themselves against software attacks and cascading failures. Self-protecting comes in two different flavors. First, systems could take measures to protect themselves from external attacks. Second, they could take actions to reduce the effect of the attack.

5.3. Evolution of autonomic operations. In order to get to autonomic systems we need to take an evolutionary approach. We need to use the existing systems and introduce self-managing capabilities without having to completely replace them. Deprecating an existing system and starting it from scratch may not be an option for big enterprise software. It may require too much time until it gets implemented and could hide significant risks of integrating the new systems with the existing ones. Therefore, all manual effort should gradually be replaced by the system in small steps. To do this, the authors of [11] discuss a model for evolving autonomic computing operations in five levels.

- **Basic level**—Each module is maintained and configured by highly skilled IT professionals. They monitor the system and take any corrective actions in case of misbehavior. In the basic level there are multiple sources of system generated data which requires significant effort from IT staff to analyze the data.
- **Managed level**—System generated data is consolidated, so that IT professionals can use fewer consoles. This reduces the total amount of time that administrators need in order to synthesize and analyze the data. This level provides better system awareness and improves the overall productivity of the IT staff.
- **Predictive level**—The system is able to recognize patterns and make predictions for the optimal configuration parameters. The system can monitor itself and provide recommendations of what course of actions can be taken. IT professionals need to approve or reject the proposed changes. This level reduces the need of deep technical skills and makes decision making faster and more efficient.
- **Adaptive level**—In addition to the predictive level, the system can take the prescribed action itself. This level can be reached once the system is able to provide a solid number of suggestions that are valid (i. e., get approved by people). The system is guided by the service level agreements (SLAs) that IT staff need to specify in advance.
- **Autonomic level**—The system operates on the basis of business policies and objectives. At this level IT professionals focus on enabling business needs. Their interaction with the system is mainly limited to monitoring or changing the business processes or updating the system goals.

Each of the described levels is dependent on the previous ones. Existing systems are classified as basic level (level 1). They need to evolve through all levels sequentially until they get transformed to fully autonomic.

5.4. Autonomic control loop. Engineering self-adaptive systems is extremely challenging. In other engineering disciplines there is a widespread notion, which could be applied in software systems, too. This is the notion of feedback. It is meant to provide feedback to the system, so that it knows what

the effect of the applied change is. In self-adaptive systems it is called control loop [12]. It provides constant feedback to the system, so that the system could improve its decision-making process. The autonomic control loop is illustrated in Fig. 1.

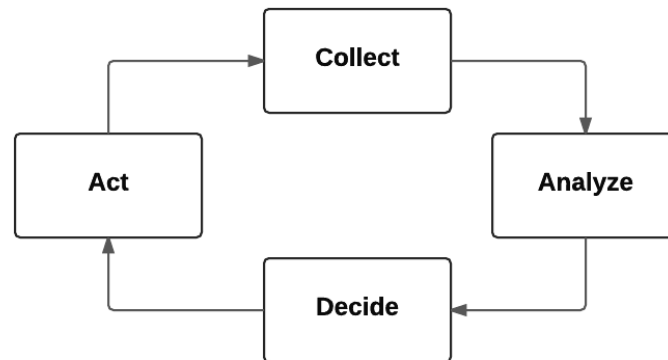

Fig. 1. Autonomic control loop

An autonomic control loop has four main activities. They follow one after another and this is a never-ending process. The reason is that components constantly need feedback on the actions they have taken. Once they collect and analyze the gathered feedback data, they can proceed with further improvement actions or revert their changes. Below is an overview of the four activities in the control loop.

- **Collect**—gather data from environmental sensors or other sources that reflect the overall state of the system.
- **Analyze**—analyze the collected data and try to map it to existing models or compare it to the established business rules. For this the raw data needs to be properly structured. The goal of this process is the system- or component-wide understanding of the system state, based on which a change decision can be taken.
- **Decide**—process the analyzed data and make a decision on what needs to be changed within the system so that it reaches the desired state. Usually, there may be many approaches for reaching the desired state of

the system and they need to be thoroughly analyzed. The decide activity would include risk analysis of the possible directions of change as well as some predictions on what the expected system state would be.

- **Act**—put the change in place, as once the decision is made is needs to be applied to the system. A common approach to this is to use some sort of effectors or actuators that can change the behavior or the state of the system.

The described control loop can be referred to as Measure-Analyze-Plan-Execute over a Knowledge base (MAPE-K) [21]. MAPE-K is adds an additional component in the closed loop—the knowledge base. It may store information that is needed for any of the steps in the loop.

6. SOA and self-adaptation. There is much work in existence in the field of self-adaptation. Many of the concepts that they propose are either aimed at software systems in general, or they could easily be applied for SOA. The reason for this is that most of them are based on control feedback loops to achieve self-adaptation.

6.1. MORPH [13]. MORPH [13] is a reference architecture for configuration and behavior self-adaptation. It allows adaptation of system configurations and behavior in an independent and coordinated way. MORPH emphasizes on having reconfiguration and behavior control as first-class architecture entities. It is structured in three main layers: Goal Management, Strategy Management and Strategy Enactment. They all share a common Knowledge Repository. All of the presented layers are based on the MAPE-K loop. However, they have different responsibilities. The Goal Management layer is responsible for setting and adjusting the overall goals for the entire system. The Strategy Management layer is responsible for the adaptation based on a set of predefined strategies. Once the most suitable strategy is selected, the Strategy Enactment layer executes it.

6.2. SASSY [14]. SASSY (Self-Adaptive Software Systems) is a model-driven framework for self-architecting distributed systems. The framework can change its settings in a dynamic way based on the changing

requirements. SASSY can generate a software architecture following several steps. First, domain experts define Service Activity Schemas (SASs), which reflect system requirements. This includes Quality of Service goals, too. Then, SASSY generates a base System Service Architecture (SSA) which is comprised of views that represent its structure and behavior. Once the base architecture is defined, SASSY derives a near-optimal architecture. This phase selects the most suitable service providers and architectural patterns related to QoS. Finally, SASSY generates a running system. It uses the previously derived architecture by binding the identified services and deploys service coordination layer. A key characteristic for SASSY is that it uses QoS architectural patterns. Each QoS architectural pattern relates to a software adaptation pattern that defines how the system adapts its configuration.

6.3. Rainbow [15, 16, 17]. Rainbow is framework which allows engineering software systems with self-adaptive capabilities at runtime. It provides mechanisms for monitoring target systems, detecting opportunities for adaptation, deciding what actions to take, and implementing those actions. Rainbow uses external adaptation mechanisms, which allows developers to specify adaptation strategies for multiple characteristics of the systems.

The framework uses architecture-based self-adaptation [18]. An architectural model represents the architecture of the entire system as a graph of components that interact with each other. Rainbow consists of three main reusable units—system-layer, architecture-layer, and translation.

6.4. Cross-layer self-adaptation of service-oriented architectures. The authors of [19] propose an approach for cross-layer self-adaptation in SOA systems. They emphasize on the two main layers described by Erl [20] – service interface and application layers – and argue that current approaches for self-adaptation in SOA are based mainly on the service interface layer. The authors present the QUA adaptation framework and provide further details on how it can be used with SOA systems.

The QUA middleware has been designed to be technology agnostic. It provides a clear separation between the three main layers—Adaptation framework, adaptation mechanisms, and adaptation targets.

6.5. A comparison of frameworks. All of the identified self-adaptation frameworks are based on the MAPE-K loop. They mainly use an external management component that monitors and adapts the managed one. However, not all of the discussed works are specifically designed for SOA systems. The following table provides a structured comparison of the frameworks.

Table 2. A comparison of frameworks

	MORPH	**SASSY**	**Rainbow**	**Cross-layer Self-adaptation of SOA**
SOA support	Yes	Yes	Yes	Yes
Designed specifically for SOA	No	Yes	No	Yes
Organized around MAPE-K	Yes	Yes	Yes	Yes
Based on adaptation patterns	Yes	Yes	Yes	Yes
Support for runtime adaptation	Yes	Yes	Yes	Yes
Software architecture generation	No	Yes	No	No
Adaptation support	Configuration, Behavior	Configuration	Configuration	Configuration, Partial behavior
Support for QoS	No	Yes	No	Partial

All of the frameworks support runtime adaptation and can leverage predefined adaptation patterns. Although not all frameworks were designed to

explicitly support SOA systems, they are architected in a way that can provide at least partial web services support. However, "SASSY" and "Cross-layer Self-adaptation of SOA" provide explicit support for SOA systems and they are the only ones that can handle QoS adaptation.

A key distinction property of the frameworks is the way that they support their self-adaptation capabilities. Configuration adaptation is supported by all frameworks but "MORPH" and "Cross-layer Self-adaptation of SOA" provide support for behavior adaptation. However, only "SASSY" can generate architecture based on predefined system goals.

7. Microservices and self-adaptation challenges. The field of microservices provides a huge field for introducing self-adaptation capabilities. As we have already discussed, microservices provide an additional layer of complexity that makes it difficult for people to handle in a reliable way. Microservices are characterized by a high level of automation and self-adaptation techniques and patterns can be built on top of them. Based on an analysis of microservice and self-adaptive systems, we have identified the following major fields where autonomous features can be introduced.

- **Service health monitoring and self-healing**—The large number of services makes health monitoring extremely difficult. The complexity can be additionally increased if services use different technology stacks. In this case self-healing capabilities should be introduced. They allow detecting service health problems and bringing back services to a state of health.
- **Container/platform monitoring and self-healing**—Microservices are usually deployed on cloud platforms and containers. Such platforms do not guarantee 100% up-time. Therefore, they should be actively monitored and self-healed in case of failures.
- **QoS support**—Microservices use lightweight protocols based on HTTP and need to provide QoS support. As all the traffic goes through networks it is very likely that on some occasions there are delays when exchanging messages. Services need to track their QoS metrics and take corrective actions in case of deviations.

Besides the major fields for improvements pointed above, there are many minor ones that are specific to the microservices domain. Such adaptation patterns should be identified based on the domain of the entire system. For example financial systems can take regular backups and automatically restore from them in case of service failures. They could also benefit from self-protecting capabilities to detect potential external or internal attacks and take proper counter measures. Safety-critical systems can autonomously run multiple copies of the services to avoid downtimes.

8. Conclusions. Microservices is a new architectural style that has emerged from SOA. It allows fast and reliable software deployment and support of large distributed systems. However, it introduces a new layer of complexity and the large number of services makes them extremely difficult to operate and support manually. Therefore, self-adaptive systems provide mechanisms that can be integrated in microservice architectures and exclude humans from making any manual operations for supporting the services at runtime. In this work we offer an overview of microservice architectures and self-adaptive systems. We analyze their capabilities and how they can be combined for optimizing the benefits that they provide. Such results are important, as in this way we point out how key aspects of microservices, like QoS support, service health monitoring and self-healing can benefit from self-adaptation capabilities.

Microservices still have many areas for research in optimization of their self-adaptation capabilities. The high level of automation lays a solid foundation for building new adaptation patterns and frameworks. Additionally, the tight connection between microservices and cloud platforms allows applying such adaptation mechanisms at various layers like infrastructure, platform and services.

REFERENCES

[1] NEWMAN S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Inc., 2015.

[2] KEPHART J. O., D. M. CHESS. The vision of autonomic computing. *Computer*, **36** (2003), No 1, 41–50.

[3] BRUN Y., G. D. M. SERUGENDO, C. GACEK, H. GIESE, H. KIENLE, M. LITOIU, H. MÜLLER, M. PEZZÈ, M. SHAW. Engineering Self-Adaptive Systems through Feedback Loops. In: Software Engineering for Self-Adaptive Systems. *Lecture Notes in Computer Science*, **5525** (2009), 48–70.

[4] NAMIOT D., M. SNEPS-SNEPPE. On Micro-services Architecture. *International Journal of Open Information Technologies*, **2** (2014), No 9, 24–27.

[5] LEWIS J., M. FOWLER. Microservices: A definition of this new architectural term. In: MartinFolwer.com, 25 March 2014. http://martinfowler.com/articles/microservices.html, 18 February 2018.

[6] RICHARDSON C. Microservices: Decomposing Applications for Deployability and Scalability. In: InfoQueue, 25 May, 2014. http://www.infoq.com/articles/microservices-intro, 18 February 2018.

[7] ABBOTT M. L., M. T. FISHER. The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise. Pearson Education, 2009.

[8] BERNSTEIN P. A., S. DAS. Rethinking eventual consistency. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. New York, USA, 2013, 923–928.

[9] RICHARDS M. Microservices vs. Service-Oriented Architecture. O'Reilly Media, Inc., 2015.

[10] Salehie M., L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, **4** (2009), No 2, Article No 14.

[11] Ganek A. G., T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal,* **42** (2003), No 1, 5–18.

[12] Cheng B. H. C., R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. di M. Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, J. Whittle. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Software Engineering for Self-Adaptive Systems, Springer, Berlin, Heidelberg. *Lecture Notes in Computer Science*, **5525** (2009), 1–26.

[13] Braberman V., N. D'Ippolito, J. Kramer, D. Sykes, S. Uchitel. Morph: A reference architecture for configuration and behaviour self-adaptation. In: Proceedings of the 1st International Workshop on Control Theory for Software Engineering. Bergamo, Italy, 2015, 9–16.

[14] Menasce D., H. Gomaa, S. Malek, J. Sousa. Sassy: A framework for self-architecting service-oriented systems. *IEEE Software*, **28** (2011), No 6, 78–85.

[15] Garlan D., S.-W. Cheng, A.-C. Huang, B. Schmerl, P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, **37** (2004), No 10, 46–54.

[16] Cheng S.-W., A.-C. Huang, D. Garlan, B. Schmerl, P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. In: International Conference on Autonomic Computing. Proceedings. IEEE, 2004, 276–277.

[17] Cheng S.-W., D. Garlan, B. Schmerl. Evaluating the effectiveness of the Rainbow self-adaptive system. In: ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems SEAMS'09. IEEE, 2009, 132–141.

[18] Oreizy P., M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, A. L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems and Their Applications*, **14** (1999), No 3, 54–62.

[19] Gjørven E., R. Rouvoy, F. Eliassen. Cross-layer self-adaptation of service-oriented architectures. In: Proceedings of the 3rd workshop on Middleware for service oriented computing, Leuven, Belgium. ACM, 2008, 37–42.

[20] Erl T. Service-Oriented Architecture: Concepts, Technology, and Design. Pearson Education, 2005.

[21] De Lemos R., H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, D. B. Smith, J. P. Sousa, L. Tahvildari, K. Wong, J. Wuttke. Software Engineering for Self-Adaptive Systems: A Second Research Roadmap. In: Software Engineering for Self-Adaptive Systems II. Springer, Berlin, Heidelberg. *Lecture Notes in Computer Science*, **7475** (2013), 1–32.

Krasimir Baylov
Department of Software Engineering
Faculty of Mathematics and Informatics
St. Kliment Ohridski University of Sofia
5, J. Baurchier Blvd
Sofia, Bulgaria
e-mail: `krasimirb@uni-sofia.bg`

Aleksandar Dimov
Department of Software Engineering
Faculty of Mathematics and Informatics
St. Kliment Ohridski University of Sofia
5, J. Baurchier Blvd
Sofia, Bulgaria
e-mail: `aldi@fmi.uni-sofia.bg`

Received September 16, 2017
Final Accepted November 15, 2017