On Micro-services Architecture

Dmitry Namiot, Manfred Sneps-Sneppe

Abstract—**This paper provides an overview of micro-services architecture and implementation patterns. It continues our series of publications about M2M systems, existing and upcoming system software platforms for M2M applications. A micro-service is a lightweight and independent service that performs single functions and collaborates with other similar services using a well-defined interface. We would like to discuss the common principles behind this approach, its advantages and disadvantages as well as its possible usage in M2M applications.**

***Keywords*—M2M, communications, software standards, micro-service, middleware.**

I. Introduction

The micro-services approach is a relatively new term in software architecture patterns. The micro-service architecture is an approach to developing an application as a set of small independent services. Each of the services is running in its own independent process. Services can communicate with some lightweight mechanisms (usually it is something around HTTP) [1]. Such services could be deployed absolutely independently. Also, the centralized management of these services is a completely separate service too. It may be written in different programming languages, use own data models, etc.

An opposite approach is so-called monolithic architecture. E.g., for Java web application you can think about a single WAR file. Yes, internally this application may have several services, components, etc. But it is deployed as a united solution. Sure, for the scalability you can run several copies of this application, but they are identical. What are the advantages?

Unless the application is getting too big, it is easier to develop. But there are some limitations connected with the development team we will discuss below.

No doubt, it is easier to deploy. It is the biggest advantage of the monolithic solution.

The path for the scalability is clear. We can run multiple copies of the application behind a load balancer. But let us see the potential problems too.

The monolithic application could be difficult to understand and modify. It is especially true, when the application is getting bigger. With the growing application it is difficult to add new developers, or replace leaving team members.

Article received Aug 20, 2014.

D.Namiot is senior researcher at Open Information Technologies Lab, Lomonosov Moscow State University. Email: dnamiot@gmail.com

M. Sneps-Sneppe is with ZNIIS. Email: sneps@mail.ru

The large code base slows the productivity. Very often we will the declined quality of the code. The original modularity will be eroded. The monolithic application prevents the developers from working independently. The whole team must coordinate all development and redeployments efforts.

It makes the continuous development very difficult. The monolithic application makes the obstacles to the frequent updates. In order to update some small component, we have to redeploy the whole application.

Scaling the application can be actual difficult too. But there is another reason. A monolithic architecture can only scale in one dimension. We can increase transaction volume by running more copies of the application. But on the other hand, this architecture can not scale with an increasing data volume. Each our copy of application instance will access all of the data. It makes caching less effective. Also, this solution increases memory consumption and input/output traffic. At the same time, different application components may have different resource requirements. One might be CPU intensive while another might be memory intensive. With a monolithic architecture, we can not scale each component independently.

The next biggest issue is a technology stack. With the monolithic architecture, it is very difficult (read – impossible) to change it. E.g. there is almost no way to change development framework, etc. It can be difficult to incrementally adopt a newer technology. And all components within the application will be sticking to technology being selected at the beginning.

Micro-services architecture gets our attention in the connection with M2M applications. We declare many times, that in our opinion “no one size fits all” in M2M applications [2]. We think that the unified (monolithic) framework for M2M (IoT) is not a realistic solution [3]. By this reason we think that micro-services are the natural fit for M2M (IoT) development. As an example, we can mention our paper [4].

II. On challenges for micro-services

Of course, the proposed micro-services approach has got an own set of drawbacks.

In practice, micro-services approach means for the developers the additional complexity of creating a distributed system.

Testing is more difficult for distributed systems.

Probably, it is one of the main problems – we must implement the inter-service communication mechanism. Also, very often, we will need some form of distributed transactions.

Of course, multiple services will require us strong coordination within the team of developers. Or, what is more probable, between the teams of developers.

Obviously, the deployment complexity will be increased. We need to deploy and manage many different service types. The next problem is also obvious. The micro-services approach leads to the increased memory consumption. It simply, due to own address space for the each service.

One of the biggest challenges is deciding how to split (partition) the system into micro-services. One obvious approach is to partition services by use case. For example, the M2M ETSI model is a typical example (Fig. 1):

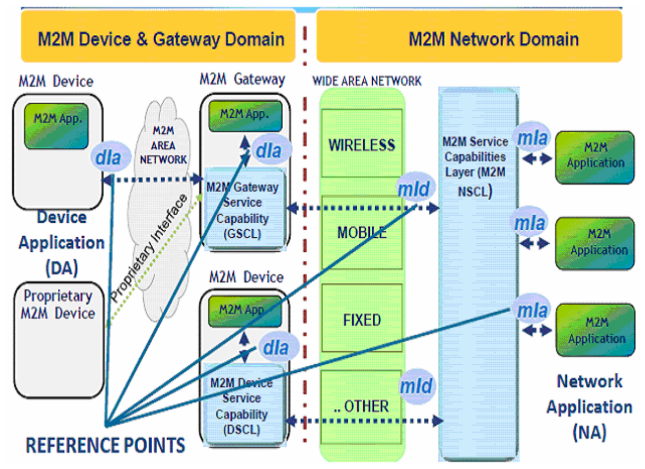

Figure 1. M2M ETSI [5]

Some of the authors also mentioned partitioning strategy by the verbs. E.g., service implements the Login sub-system, Backup sub-system, etc. [6]

Another partitioning approach is to partition the system by nouns or resources [7]. This kind of service is responsible for all operations that operate on entities/resources of a given type. For example, Figure 2 presents FI-WARE data model:

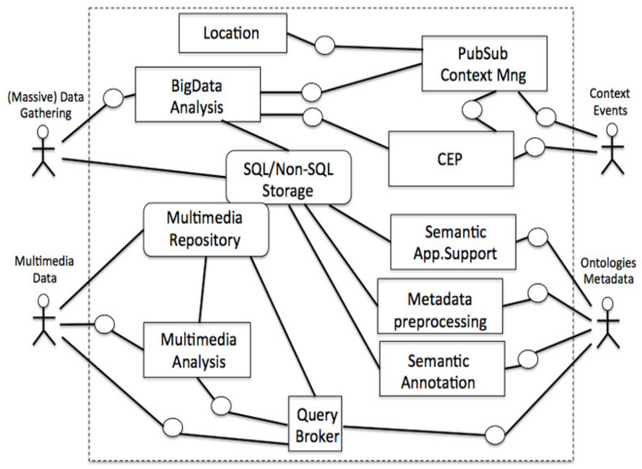

Figure 2. FI-WARE data model [8]

Ideally, each service should have only a small set of responsibilities. We should mention in this case the Single Responsible Principle (SRP) pattern [9]. The SRP defines a responsibility of the class as a reason to change, and that a class should only have one reason to change.

Another widely used illustration of various approaches for partitioning monolith applications is scaling cube [10] (Figure 3).

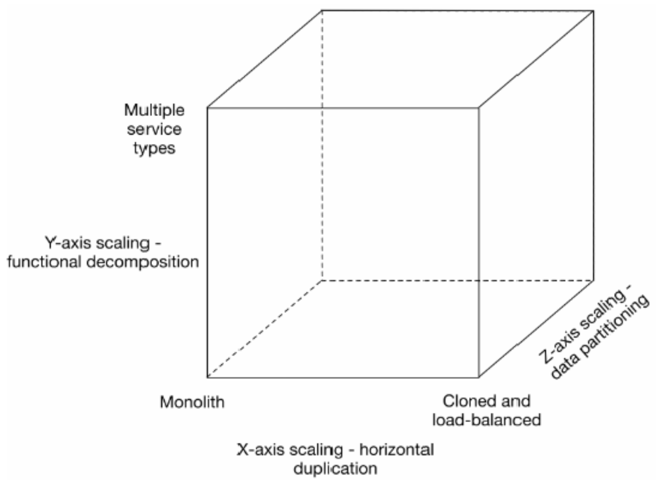
Multiple service types
Y-axis scaling - functional decomposition
Z-axis scaling - data partitioning
Monolith
Cloned and load-balanced
X-axis scaling - horizontal duplication

Figure 3. The scale cube [11]

Here X-axis scaling is so-called horizontal scaling. We scale our application by running multiple identical copies of the application behind a load balancer.

For Z-axis scaling each server runs an identical copy of the code. But in this approach, each server is responsible for only a subset of the data. Some proxy (a special component of the system) is responsible for routing each request to the appropriate server. For databases, for example, we can use the primary key as a main source for this routing. It is so-called sharding [12]. Another widely used example of this approach is a service division for free/payable users. The code base for service is the same, but servers may have different capacities (depends on bill).

Both Z-axis and X-axis scaling improve the application's capacity and availability. But in the same time they can increase the application (and development) complexity too. So, for dealing with the increased complexity we can follow to the Y-axis scaling. It is a functional decomposition exactly. For example, Z-axis scaling splits things that are similar, where Y-axis scaling splits things that are different. At the application tier, Y-axis scaling splits a monolithic application into a set of services. Each service implements a set of related functionality (sub-set of the application's functionality).

III. MICRO-SERVICES AS COMPONENTS

Traditionally, a component is a unit of software that is independently replaceable and upgradeable [13]. And libraries are components that are linked with a program. Usually libraries are called via in-memory functions calls.

Services are out-of-process components. And for communications developers should use some forms of remote procedure calls. By this reason, services are components, rather than libraries. Another reason for this conclusion is the deployment. Services (by the definition) should be independently deployable. Vice versa, for changing library within the application (e.g., update it to a new version) we need to redeploy the whole application.

Unfortunately, this does not work in 100% of cases. What if the interface for some individual service is changed too? In this case we will need more to efforts in the redeployment than simply updating a new service. It is one of the requirements of the micro-service architecture – to minimize the possible influence in case of interface changes. It is about proper design for service contracts.

Of course, remote calls are more expensive than in-process calls. So, developers have to pay more attention to its development due to high price of changes. But this physical isolation is the main strength of micro-services approach. This isolation is a key to scaling. The physical isolation lest define the key components for scaling (as per the standard 80/20 rule [14]).

And the discipline required for the developing service contracts is a yet another strength of this approach. Any development without the proper boarding between components sooner or later leads to the un-maintainable code.

We can mention the following primitives need for micro-services architecture [15]:

1) Request/Response calls with arbitrary structured data

2) Asynchronous events should be flowing in real-time in both directions

3) Requests and responses can flow in any direction,

4) Requests and responses and can be arbitrarily nested. The typical example is a self-registering worker model

5) A message serialization format should be pluggable. So, developers may use, for example, JSON, XML, etc.

IV. COMMUNICATIONS IN MICRO-SERVICES ARCHITECTURE

In this paragraph, we would like to discuss communication patterns. Really, as soon as we talk about distributed systems and remote calls in micro-services architecture the network part of the system becomes crucial. We would like to present some patterns and discuss the related challenges. The first communication pattern is obvious. Our application can use each service directly (Figure 4).

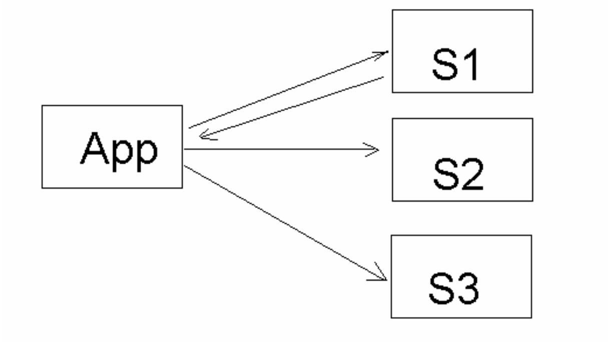

Figure 4. Direct calls

It is, no doubts, the most flexible way. Think, for example, about web server being able to call various services before rendering the output page for some particular request. The biggest problem, of course, is the potential delays for remote calls. So, the next step is almost obvious. We need to decrease the amount of remote calls. It leads us to the various forms of cache and to the solutions, similar to transaction monitors in databases [16], middleware (3-tier) applications [17], etc. It is illustrated in the Figure 5.

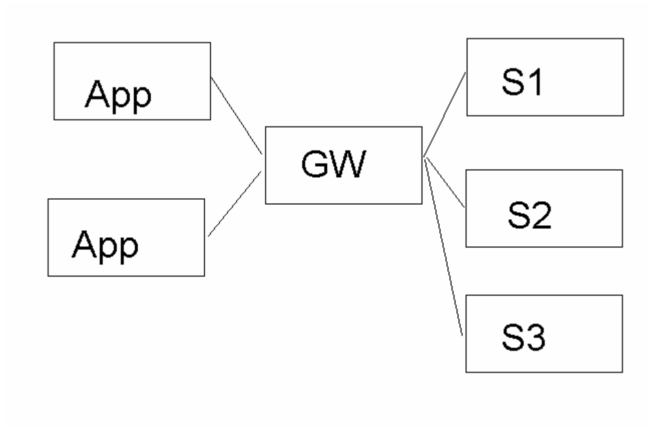

Figure 5. A gateway for micto-services

Note, that this pattern is more traditional for M2M (IoT) applications, because this gateway can also hide some limitations for legacy devices, for example (e.g., for service == device mapping).

And the third pattern is some service-bus. It is suitable for M2M (IoT) applications due to the asynchronous nature for the most of the services. E.g., for the most of sensors, data reading requests are asynchronous. So, service (message) bus lets application post requests and read response later (Figure 6)

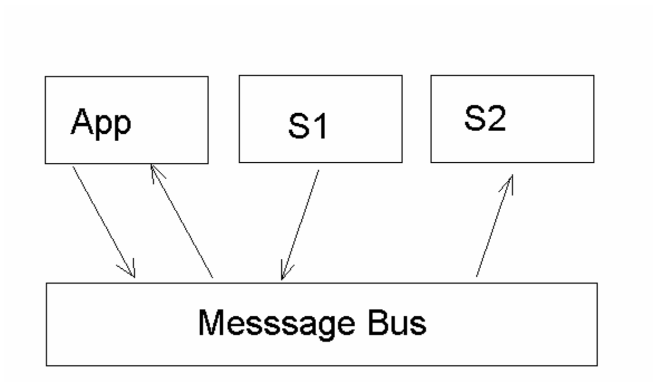

Figure 6. Message Bus

Actually, Publish/Subscribe model is widely used in IoT applications. The reasons for this are obvious too. It is easy with this model to add new components (read - a new functionality) without any changes to the existing components. Any new functionality could be deployed step by step and this process does affect the already deployed components. And the service-bus deployment itself can use clustering and load balancing to improve scalability by distributing the workload across nodes.

REFERENCES

[1] Uckelmann, Dieter, Mark Harrison, and Florian Michahelles. "An architectural approach towards the future internet of things." Architecting the internet of things. Springer Berlin Heidelberg, 2011. 1-24.

[2] Namiot, D., & Sneps-Sneppe, M. (2014). On M2M Software Platforms. International Journal of Open Information Technologies, 2(8), 29-33.

[3] Namiot, D., & Sneps-Sneppe, M. (2014). On M2M Software. International Journal of Open Information Technologies, 2(6), 29-36.

[4] Schneps-Schneppe, M., Namiot, D., Maximenko, A., & Malov, D. (2012, October). Wired Smart Home: energy metering, security, and emergency issues. In Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2012 4th International Congress on (pp. 405-410). IEEE.

[5] “ETSI Machine-to-Machine Communications info and drafts” http://docbox.etsi.org/M2M/Open/ Retrieved: Jul, 2014.

[6] Hassan, M., Zhao, W., & Yang, J. (2010, July). Provisioning web services from resource constrained mobile devices. In Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on (pp. 490-497). IEEE.

[7] Microservices http://microservices.io/patterns/microservices.html Retrieved: Aug, 2014

[8] Elmangoush, A., Al-Hezmi, A., & Magedanz, T. (2013, December). Towards Standard M2M APIs for Cloud-based Telco Service Platforms. In Proceedings of International Conference on Advances in Mobile Computing & Multimedia (p. 143). ACM.

[9] Martin, Robert Cecil. Agile software development: principles, patterns, and practices. Prentice Hall PTR, 2003.

[10] Abbott, Martin L., and Michael T. Fisher. The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise. Pearson Education, 2009

[11] The scale cube http://akfpartners.com/techblog/2008/05/08/splitting-applications-or-services-for-scale/ Retrived: Aug, 2014

[12] Stonebraker, M. (2010). SQL databases v. NoSQL databases. Communications of the ACM, 53(4), 10-11.

[13] Heineman, George T., and William T. Councill. "Component-based software engineering." Putting the Pieces Together, Addison-Westley (2001).

[14] Gorton, Ian, Anna Liu, and Paul Brebner. "Rigorous evaluation of COTS middleware technology." Computer 36.3 (2003): 50-55.

[15] Libchan https://github.com/docker/libchan Retrieved: Aug, 2014

[16] Dayal, U., Garcia-Molina, H., Hsu, M., Kao, B., & Shan, M. C. (1993, June). Third generation TP monitors: A database challenge. In ACM Sigmod Record (Vol. 22, No. 2, pp. 393-397). ACM.

[17] Reijers, N., Lin, K. J., Wang, Y. C., Shih, C. S., & Hsu, J. Y. (2013). Design of an Intelligent Middleware for Flexible Sensor Configuration in M2M Systems. In SENSORNETS (pp. 41-46).

[18] Sneps-Sneppe, M., & Namiot, D. (2012, April). About M2M standards and their possible extensions. In Future Internet Communications (BCFIC), 2012 2nd Baltic Congress on (pp. 187-193). IEEE.