Self-Adaptive Agent for Reliable RESTful Messaging

Krasimir Baylov
University of Sofia St. Kliment Orhidski/Department of Software Engineering, Sofia, Bulgaria
krasimirb@uni-sofia.bg

***Abstract*— Service Oriented Architecture (SOA) is the most popular and widely adopted architectural style for implementing and integrating large-scale heterogeneous systems. While it has a lot of forms, microservices were established as the preferred way of building distributed SOA applications. Such architectures tend to have a lot of asynchronous REST communication. Their distributed nature, however, often results in complex design decisions that address the issue of reliable messaging. In this work, we present a self-adaptive agent that guarantees reliable RESTful messaging. The agent can be integrated into existing services, thus reduce the overall complexity of the entire infrastructure.**

I. INTRODUCTION

Microservices [1] is an architectural style based on Service Oriented Architectures (SOA) used for building large distributed systems. They come with a set of characteristics that help companies respond to business changes in a much faster way. However, their distributed nature introduces a new set of challenges to be addressed - network failures, service failures, scalability, etc. Multiple services communicate with each other using Representational State Transfer (REST) [1] over the network and this means that new mechanisms should be introduced to guarantee that all messages are received and processed. Microservice architectures favor the use of lightweight message buses [2] but this requires new infrastructure components that need to be deployed and supported. While message buses decouple the communication between services, they do not fully solve the problem with reliable messaging between the services. This often requires additional custom development to handle the different failure situations - detecting not delivered messages, messages not processed because of service unavailability, retry mechanisms, etc.

The extensive usage of REST in microservice systems makes communication easier and more reliable, but it cannot meet all interoperability requirements [8]. Contracts are not always reliable because of different invocation semantics, message exchange formats. Moreover, the distributed nature of microservices emphasizes the need of reliable message passing mechanisms [9]. The large number of communication channels makes them extremely vulnerable to failures in large and complex systems.

This determines the need of a solution which guarantees reliable messaging between microservices and yet minimizing the need of developing and supporting multiple external components. Such a solution should be flexible enough, so that it can be integrated and reused across multiple services. Most importantly, it should address the risk of networks failures and service unavailability which are common for distributed systems.

II. RELATED WORK

Some of the most common challenges with contemporary microservice architectures is their availability and reliability [3]. This is a result of the distributed components and the underlying infrastructure that supports them.

Authors of [4] propose a distributed and centralized MANO framework designs which result in high availability and fast fault recovery. They use distributed load balancing techniques and dynamic state sharing events to allow quick recovery and increased availability. However, the approach does not fully address the reliability of message delivery in case of service failures.

Sam Newman [2] argues that microservices should rely on lightweight protocols for message communication like REST and message queues. They should avoid using complex middleware like Enterprise service buses. However, this approach requires new infrastructure components that need to be deployed and supported over time. In case of failures, developers need to introduce new mechanisms that handle cases like service failures, message retries, etc. This increased the time for development and requires additional effort in monitoring and supporting such applications.

Authors of [10] propose a solution for reliable messaging using CORBA Notification Service. The notification service is a suitable mechanisms for applications that can detect or tolerate duplication of messages. Microservices, however, rarely face the problem of message duplication.

Simple Object Access Protocol (SOAP) [11] provides mechanisms for ensuring reliability in message delivery. WS-Reliability and WS-Reliable Messaging [12] could be integrated into SOA architectures. There are multiple patterns for reliable messaging based on SOAP. SOAP, however, is considered too heavyweight for microservice systems and is rarely used.

None of the studied approaches is based on self-adaptive systems [5]. Such systems could pursue high-level goals with minimal or no human supervision. This makes them extremely suitable in the context of microservices for several reasons. Microservice architectures are complex and monitoring them should be automatic. Failures are extremely common, so they should be detected and fixed in a timely manner. Self-adaptive systems could provide solution to many of these limitations. The same is valid for ensuring reliable messaging.

In general, there is no lightweight approach which takes advantage of self-adaptive systems and introduces reliable messaging that eliminates the need of deploying new infrastructure components.

III. Core Requirements

In order to address the identified problem, we have determined a set of core requirements that fill the gaps in the field of reliable messaging in microservice systems. A list of the main requirements is presented below:

A. Lack of External Components

The solution should not require the deployment of any new infrastructure components outside the existing microservices. This reduces the overall complexity of the entire solution. Lack of external components results in increased performance, lower risk of integration issues, reduced time for development, etc.

While using external components for addressing reliable messaging is a common practice, they increase the overall complexity of the entire system. Developers have more flexibility, but they also must support a larger set or infrastructure components.

B. Configurable and Extensible Solution

The solution should be easily configurable and extensible, so that developers could easily enhance its functionality. Configuration parameters allow developers to adjust the behavior of the solution without having to introduce any programmatic changes. Such changes could be applied both at compile and at runtime.

The architecture of the solution shall be easily extensible. It should have a clearly defined modular architecture. Each of the designed modules should provide stable integration points, so that developers could enhance them based on their specific need. In addition, they could also add new components or modules to address more complex or additional problems

C. Simple Application Programming Interface (API)

The solution should provide simple to use API so that developers could easily integrate it into existing microservices. Such an API should allow access to the key functionalities and modules. Moreover, it should have two layers of abstraction.

High Level API should allow using the solution by focusing on the high-level features. Such features are sending messages, obtaining responses, setting key configuration parameters, etc. This API should hide the details regarding the networking protocols, headers manipulation, etc.

Low Level API should provide fine grained control over the solution. This API allows defining specific network or RESTful libraries as well as direct manipulation of messages payload.

D. Message Buffering

The solution should provide a mechanism for message buffering. This mechanism should be self-adaptable based on the non-functional characteristics of the component receiving messages. E.g. if the destination component cannot handle all of the sent messages because of reduced throughput, these messages should be buffered and sent at a later stage. Such an approach has the following key benefits.

No lost messages. If the destination point cannot process the message because of high load, it will be buffered until the load is normalized and it can be processed.

Potential Denial of Service of the destination point. Overloading the destination point with more messages that it can handle could easily result in denial of services. This could have significant impact on the entire systems that it supports.

E. Retry Mechanism

The solution should provide a retry mechanism for handling messages that failed the delivery process. If a message has not been successfully delivered to the destination point, it should be marked for retry later. The retry mechanisms should be self-adaptable based on the analyzed behavior of the destination point.

While there are many additional requirements, the above ones define the core principles that the solution should comply with. In addition, we have incorporated some of the principles used for building self-adaptive systems [5]. We have incorporated an autonomic control loop which adapts the behavior of the solution based on external factors like load of the current service, availability of the external system, network latency, etc.

IV. Implementation

As a result of the current research we have built a Java agent which can be integrated into existing services. The agent addresses the above-mentioned requirements and does not require any additional infrastructure components in order to guarantee reliable messaging. The overall architecture of the agent is presented in figure 1.

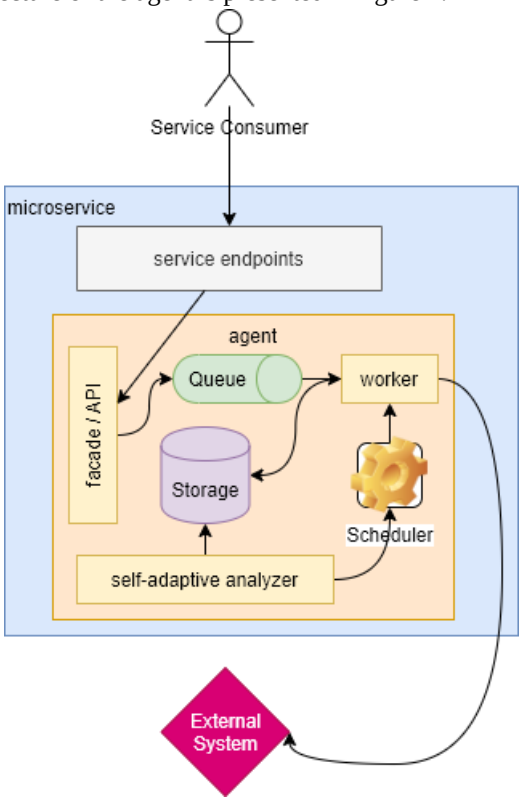

Figure 1. Architecture of Self-adaptive agent for reliable messaging

The agent can easily be integrated into existing microservices. Instead of making direct calls against the external system, developers should use the provided "facade/API". No further changes are required in order to use the agent. The architecture consists of a "Queue" which is built-in message queue that buffers the messages. This increases the throughput of the microservice while guaranteeing that the messages will be processed even if the external system processes them much slower. The "worker" component is responsible for processing the queued messages. It uses a local storage where all information regarding the message execution is stored. If the external system is unavailable, the local storage is used for logging the failed executions. In this case a "self-adaptive analyzer" parses the data in the local storage and creates schedulers which can retry sending the messages until the external system is available again. The self-adaptive analyzer could update the rules of the scheduler based on multiple external factors. In addition, the agent could be further configured to use external storage which ensures that state is persisted over service redeploys. Below is presented a detailed overview of the main architectural components.

A. *Façade/API*

The "Façade/API" component is the entry point for the agent. It provides the main interface what should be used by developers to integrate the agent into their existing microservices.

B. *Queue*

The "Queue" component is a message queue that buffers messages. Every message is initially stored in the queue. This way all messages are processed with the pace at which the destination endpoint (external service) can process them.

Developers could easily use external message queue or adjust the characteristics of this one by updating the configuration settings of the agent. In case external queue is used, the relevant destination parameters should be provided.

C. *Worker*

The "Worker" is responsible for processing the messages. It handles every single message from the queue and sends it against the destination point. This component has some additional responsibilities. It analyses the response. If the message was not delivered, or some error was encountered, the message may be marked as "failed". This means that it should be resent at some later point of time using the retry mechanism.

In general, the worker component should collect any data related to operational characteristics of the endpoints. This data can later be used by the self-adaptive analyzer to optimize the work of the entire agent.

D. *Storage*

The "Storage" component persists all the data related to failed messages as well as operational metrics of the destination points (external system). By default the storage is provided as an in-memory storage. This eliminates the need of increasing the complexity by introducing external storages. However, if the service is restarted, all the data could be lost. Using the in-memory storage is suitable for cases where microservices are running long enough, or the destination point process most of the messages successfully.

External storages can be used by updating the configuration settings of the agent. In this case, the destination details of the external storage should be provided. Such an approach is suitable for environments where the reliability of the destination points is extremely unstable. If the microservices are restarted on a regular basis, this guarantees that all failed messages will be retried.

E. *Self-Adaptive Analyzer*

The "Self-adaptive Analyzer" is a component which introduces self-adaptive capabilities of the agent. It uses a feedback loop [6] to analyze the existing data and update the behavior of the agent. This results in updates related to the types of messages that are processed and retries, the intervals of retries, etc. In some cases, the Analyzer may introduce updates to the messages. For it could update the *content-type* header of the message based on the payload and returned errors from the destination points.

The Analyzer may update the intervals of retries. If all messages against the same destination endpoint fail for a short period of time, the retries intervals could be increased. This way, the load of the agent and the destination point is reduced.

F. *Scheduler*

The "Scheduler" module is responsible for handling all schedules related to retrying the different messages. Every retry message should be scheduled for execution at a certain time and the Scheduler guarantees that it will be processed.

V. TEST RESULTS

The implemented agent is tested in a microservice system and the results show that it satisfies the core requirements. In order to evaluate its efficiency, we have tested two microservices using an external system which is not reliable in terms of processing messages. The first microservice (MS 1) does not use the proposed agent, while the second (MS 2) one does.

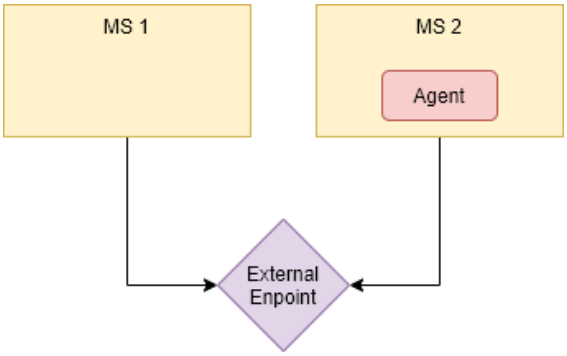

Figure 2. Testing environment

The "External Endpoint" service is configured to fail in processing messages at a rate of 20%. In addition, the throughput if the service is 5 messages per second

TABLE I. TEST RESULTS

	MS 1	MS 2 (Agent)
Average time for processing a message	133ms	135 ms
Delivered messages (overall)	72%	100%
Delivered messages on high load (10 messages per second)	58%	100%

The results show that performance overhead of the java agent is minimal. On the other hand, the rate of successfully delivering messages to the External Endpoint remains 100%. The longest number of retires for a single message is 5.

The agent is tested in simulated environment and the results show that it is capable of achieving 100% reliable messaging. However, next steps in our research would include further testing the agent in large systems.

VI. CONCLUSIONS

Microservices architectural style is widely used for building large distributed systems but often results in reduced reliability in message communication between the services. While there are approaches to address this problem, they all require custom development and introducing new infrastructure components. This paper proposes a Java agent for reliable RESTful messaging. The agent is component which could be integrated into existing microservices without the need of configuring external infrastructure components. Results show that the agent is capable of achieving 100% reliable messaging in small microservice environments. Next steps in our research would be focused in the following directions:

- Further testing the agent in larger and more complex systems
- Extending the self-adaptive capabilities of the agent
- Developing agents or other programming languages

REFERENCES

[1] Thönes, Johannes. "Microservices." IEEE software 32.1 (2015): 116-116.

[2] Newman, Sam. Building microservices: designing fine-grained systems. " O'Reilly Media, Inc.", 2015.

[3] Dragoni, Nicola, et al. "Microservices: yesterday, today, and tomorrow." Present and ulterior software engineering. Springer, Cham, 2017. 195-216.

[4] Soenen, Thomas, et al. "Optimising microservice-based reliable NFV management & orchestration architectures." 2017 9th International Workshop on Resilient Networks Design and Modeling (RNDM). IEEE, 2017.

[5] Cheng, Betty HC, et al. "Software engineering for self-adaptive systems: A research roadmap." Software engineering for self-adaptive systems. Springer, Berlin, Heidelberg, 2009. 1-26.

[6] Brun, Yuriy, et al. "Engineering self-adaptive systems through feedback loops." Software engineering for self-adaptive systems. Springer, Berlin, Heidelberg, 2009. 48-70.

[7] Fielding, Roy T., and Richard N. Taylor. Architectural styles and the design of network-based software architectures. Vol. 7. Irvine: University of California, Irvine, 2000.

[8] Zimmermann, Olaf. "Microservices tenets." Computer Science-Research and Development 32.3-4 (2017): 301-310.

[9] Dragoni, Nicola, et al. "Microservices: yesterday, today, and tomorrow." Present and ulterior software engineering. Springer, Cham, 2017. 195-216.

[10] Ramani, Srinivasan, Balabrishnan Dasarathy, and Kishor S. Trivedi. "Reliable messaging using the CORBA Notification service." Proceedings 3rd International Symposium on Distributed Objects and Applications. IEEE, 2001.

[11] Box, Don, et al. "Simple object access protocol (SOAP) 1.1." (2000).

[12] Buckley, Ingrid, et al. "Web Services Reliability Patterns." SEKE. 2009.