Adaptation of HTTP request-response messaging to arbitrary messaging pattern in RESTful service oriented architectures

Lazar Nikolić, Marko Letić, Bojana Zoranović, Igor Dejanović
Faculty of Technical Sciences, University in Novi Sad, Serbia
lazar.nikolic@uns.ac.rs, marko.letic@uns.ac.rs, bojana.zoranovic@uns.ac.rs, igord@uns.ac.rs

***Abstract*—Service oriented architectures have traditionally been using HTTP messages for client-server communication, as well as inter-service communication. However, request-response pattern of HTTP protocol can be inefficient for messaging patterns such as event-driven messaging. The problem arises for architectures that need to introduce new messaging patterns, but already use HTTP. In this paper we present architecture of an adapter that converts HTTP messages into messages of another protocol. Such protocol can use a messaging pattern other than request-response. We cover synchronous request-response, asynchronous request-response and event-driven messaging patterns. The adapter must also allow transition to a new protocol or messaging pattern without affecting service interface definitions and their application logic. We also present an implementation of such architecture and discuss the results.**

1. MOTIVATION

Despite its age, HTTP protocol is still the most widely used network protocol on the Internet [1]. It is used in two of the most popular web service implementations, SOAP[21] and more recently REST[14]. Although they greatly differ in style, HTTP is mostly used in both approaches as the message carrier. But HTTP was obviously designed with server-client communication in mind; it might be inefficient for complex systems that rely on events, or other messaging patterns other than request-response.

For larger scale projects, migration to a new messaging pattern or messaging protocol is a time consuming and tedious task. Reasons for this migration can be whole system rework with legacy support, introduction of a more specialized messaging pattern (e.g. for stream processing), performance issues caused by inefficient inter-service communication etc. Services must also be able to handle the new message format, meaning that the code must be refactored to accommodate this change. Instead, using an adapter can ensure separation between the services and the messaging protocol. As a result, service interface definitions and application logic should remain intact.

Web services using HTTP require load balancing and service discovery components to be included in the system. One solution is to introduce battle-tested solutions such as Eureka[2], but this can cause two problems. The first problem is that doing so potentially increases complexity and fragility of the system as there are more moving parts that can fail. Other problem comes from the fact that services need to query the registry to get network address of one or multiple destinations. This leads to additional network hops needed to fulfill a request, leading to increased latency. Instead, we try to leverage on built-in service discovery and load balancing of target messaging platform. Solutions such as ZeroMQ, NATS and Apache Kafka [3][4][5] offer these features.

2. RELATED WORK

Transitioning to a new messaging pattern can be seen as a problem of integrating two systems: internal and external. Internal system refers to web services of the system, while external refers to clients outside of that system. While the internal system will change its messaging and protocol, the external system will not. Integrating two systems using different protocols is a known issue. Works in this area have addressed and formally defined the problem [5][6][7]. Solutions have been made for service mapping [8][9][10] mostly SOAP-to-REST or REST-REST service mapping based on WSDL. In this case the underlying protocol stays the same (i.e. HTTP), so protocol conversion boils down to matching XML/JSON elements from the payload of two observed systems. In other approaches, such as NATS proxy[11], two systems truly use different application level protocols (i.e. NATS and HTTP).

A. StoRHm

StoRHm[9] is a protocol adapter, described as best suited for systems to transition from SOAP web services to RESTful HTTP web services. While both approaches use HTTP messages, paradigms are much different. SOAP is a protocol carried on top of HTTP, whereas REST is an architectural style. Another important difference is that SOAP only carries XML documents, while REST does not have such a constraint. StoRHm allows its user to define mappings between WSDL service definitions and RESTful service definitions through a GUI client application. Based on these mappings, adapters convert SOAP messages to HTTP messages between internal and external systems.

B. NATS proxy

NATS proxy is a framework written in Golang that provides a bridge between NATS and HTTP, or more specifically, NATS and RESTful HTTP. It aims to migrate REST based architecture to an architecture using NATS messaging platform. In the current version, NATS proxy only supports synchronous request-response messaging. Besides protocol adaptation, it brings another benefit of leveraging on service discovery and load balancing of the NATS platform itself, eliminating the

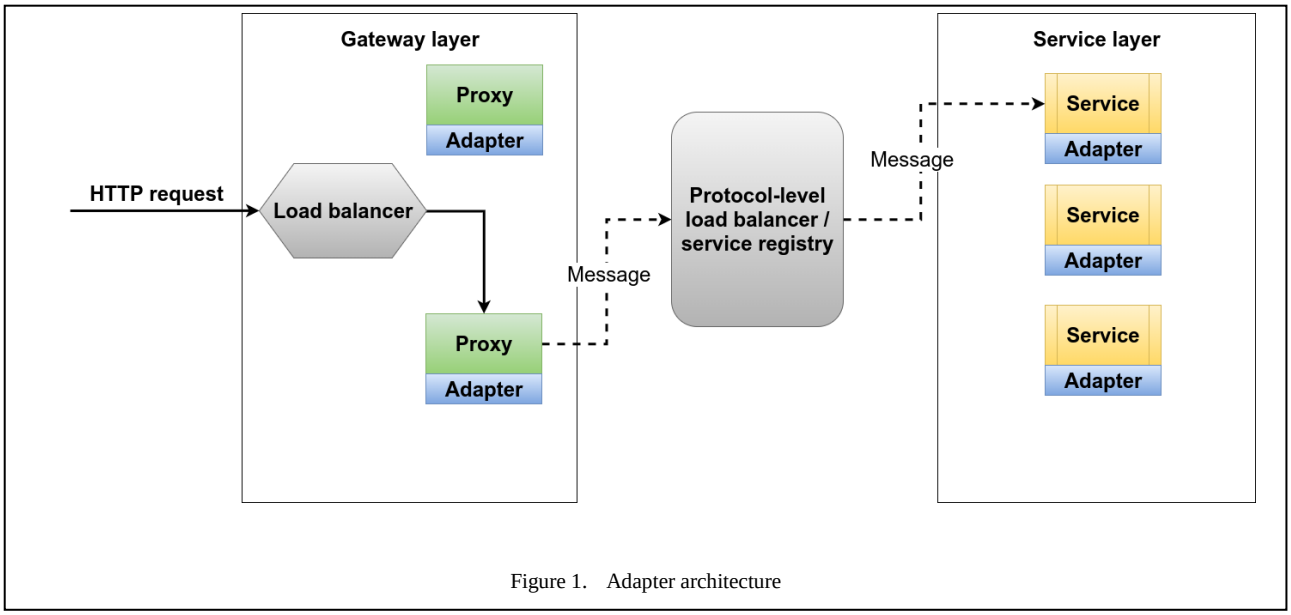

Figure 1. Adapter architecture

need for additional network components, thus reducing complexity of the system.

In both described approaches only one part of the problem is being addressed. Web services still need to be refactored to accommodate the changes in protocol and messaging format. It is easy to see that interface has to change when the underlying protocol or payload type changes; two protocols will most likely differ in both structure and concepts. Our goal is to provide both protocol adapter and service interface adapter. This way we attempt to also solve the other part of the problem, which is making new messaging format understandable to existing web services without changing them.

3. Architecture

In the previous chapter we discussed that in order to fully migrate web service architecture from one messaging pattern or protocol to another, two adapters are needed. The first adapter acts as the protocol adapter, converting HTTP messages to another protocol. This adapter is integrated into the entry point of the system, forming the **gateway** or **proxy layer** of the protocol adapter. The second adapter serves to translate payload to a format web services understand (e.g. POJO for Java). It is integrated into web services and needs to understand both the language and data format of web services used in the system. This adapter is part of the **conversion layer** with the web services it is integrated with. Conversion layer is naturally a part of the service layer. Overview of the architecture is shown in Fig. 1.

Web service architecture commonly have reversing proxies as gateway into the system. Since an inbound request first meets these proxies, it is reasonable to include protocol adaptation logic there. This is done by providing a library that is specific to language of the service and target protocol. Potential problems are that there might not be library support for used languages, and increased difficulty in introducing new languages. The other approach is to create a separate conversion service that is not language specific, but doing so leads to increased complexity and latency. We have chosen the former approach, as the adapter should strive to improve or at least keep the same performance.

Requests sent from clients run through a following pipeline:

1. Inbound HTTP request is received by a single service instance from the gateway layer.
2. Gateway can cancel the request if it doesn't match a certain set of criteria (e.g. authorization). Request that passes this check will be converted into a message of the chosen protocol.
3. Newly created message is sent to the messaging platform that does load balancing and service discovery. If services are unavailable or request times out, error response is back to client via gateway layer.
4. Inbound message is received by a service and is then converted into format that the service understands. This conversion is done by the adapter integrated with the service.
5. Service computes a response in its native format. The adapter takes this response, converts it into a message of the target protocol and sends it to a gateway service.
6. The gateway service takes the response message, converts it into an HTTP response and sends it back to the client.

4. Gateway Layer

Gateway layer is composed of edge services (proxies or gateways) serving as the entry point into the architecture. Here is where authorization, authentication, routing etc. is usually done. Logic for these actions rely on HTTP elements, in particular headers, method and URL. In two of the most popular authorization protocols, OAuth and JWT [12][13], authorization data is carried by *Authorization* header. For RESTful web services, requested action is expressed by a combination of HTTP method and URL as described in [14]. Request payload is carried by either body or URL, depending on the method. All these things must be taken into consideration when creating a protocol mapping. It is also imperative that the mapping provided as a simple drop-in for the gateway logic. To accomplish this, mapping library should have

the same interface as popular HTTP libraries used by language of the service.

As an HTTP request is received from the outside of the system, its validity is immediately checked in order to avoid using unnecessary resources. For example, a request can be directed at an endpoint not found in the routing list, or the client is not authorized to access requested resource. In other words, request first passes through the existing code before reaching our adapter, meaning that only valid requests should reach it. Its interface is made to mimic popular libraries of the language gateway services are implemented in. For example, Java adapter accepts `javax.servlet.http.HttpServletRequest` which is associated with Apache Tomcat [15] and commonly used by Spring [16] applications. The idea is to keep code modifications at minimum. Ideally, only one line of code will be modified: the one that forwards the request to services. Depending on the target protocol or messaging platform, code responsible for service discovery might have to be removed. However, removal of code can be a welcome change as it reduces complexity.

A. *Message pattern conversion*

Architectures using this adapter should be able to transition to a messaging pattern best suitable for its needs. As HTTP messaging is request-response pattern, the adapter must ensure that there is a mapping available for the new pattern. In this chapter we discuss how to map HTTP to synchronous request-response, asynchronous request-response, and event based messaging.

As an inbound HTTP request arrives and a new message is created, it is assigned an 128-bit Universally Unique Identifier (UUID). This applies to all mappings as it is important to trace messages and keep have a correspondence between request messages and response messages.

1. **Synchronous request-response** conversion is rather simple. HTTP messages are simply mapped according to mapping rules and forwarded to the messaging platform. UUID can be unnecessary here, because it is more efficient to keep the TCP connection open as it uniquely identifies a HTTP request (TCP connection can carry only a single HTTP request).
2. **Asynchronous request-response** conversion heavily relies on UUID. As an HTTP message is received, client receives a "200 OK" response containing UUID of the message. There are two possible ways of implementing callbacks:
   a) HTTP request contains an *X-Callback* header containing the URL of the callback endpoint. This endpoint will be called when the response is ready. It must be able to accept the message that is normally returned via response.
   b) Client establishes a WebSocket connection with the gateway service. Once the response message arrives to gateway layer, it is sufficient to simply emit a WebSocket event. However, each client then keeps an open TCP connection that the server must maintain. A large number of clients could potentially use up all of the services' resources.

In both cases, a request timeout is chosen that can differ from the timeout for synchronous requests. Service that originally received the message generates the expiration date of the request, and starts a timer. Once this timer expires, an error response is sent to the client. Services also push this expiration date into a shared, distributed storage such as Redis[17]. This way, expired messages can be discarded if encountered by another service, either if the original crashed or the load balancer forwarded the response message to a service other than the original. Asynchronous conversion is not fully supported in the current version.

   c) Conversion to **events** is done by emitting an event to a subscription group that equals the base URL of the received message. Lets consider an example containing *user service* with base URL */users*. Upon receiving an HTTP message with URL */users/findByName*, gateway emits an event to the subscription group "users" containing action "findByName". Since events do not require any responses, client receives an "200 OK" response as soon as an event is successfully emitted.

5. CONVERSION LAYER

Conversion layer is responsible for bridging web service interface with the new message format. It is integrated with web services as a client library. Ideally, each language would have an implementation of this library. Main purpose of an adapter from this layer is to accept a message and create a corresponding object native to the language it was written in. For example, in RESTful web service architecture, message payload is usually either XML document or a JSON object. Both of these formats are not difficult to convert to POJO, especially as Java environment already offers many libraries for this purpose. Web frameworks such as Spring already have this mapping logic, which can be leveraged on in a custom implementation for Spring.

Web service adapter must accept messages, which implies that it must include a server. This server is

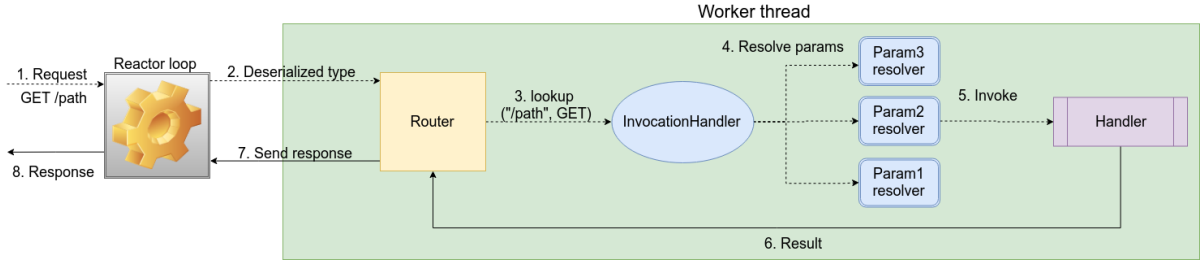

Figure 2. Conversion layer architecture

specific to the target protocol or messaging platform and replaces HTTP server of the web service. We have decided on a reactor-loop based server.

Adapter is composed of several components:

1. **Reactor** that accepts and enqueues messages, and delegates work to available worker threads.
2. **Worker thread** is a thread that executes service logic. Reactor component contains a configurable number of worker threads in its thread pool.
3. **Router** is a component that relies on a structure similar to a radix tree[18] to lookup handlers for a given path. Main difference between a true radix tree and the one router uses, is that nodes are formed based on common path segments instead of common prefixes.
4. **InvocationHandler** encapsulates information on how to invoke a function handler. It is composed of **ParamResolver** subcomponents per formal parameter of its function handler. Each resolver resolves an argument for the parameter it represents. Paramresolver is created for each data type used in the service. InvocationHandler is the component that separates service logic from the embedded server, ensuring no changes are required in function handlers.
5. **Handler** is a function that implements service logic. It is also a part of service interface definition as it describes a single endpoint (i.e. path and parameters).

Inbound message goes through the following sequence of events presented in Fig. 2:

1. Request message arrives to the server. Reactor loop then checks the worker thread for an available worker thread. If there are no available threads, message is put into a queue. In case the queue is full, circuit breaking logic is triggered and an error message is sent as a response. Service is unavailable until queue is cleared or its size falls under a certain threshold. Either way, once an available thread is found, it is launched with the received message sent as an argument.
2. Router component receives the deserialized message and uses its path value as input to its routing tree.
3. If the lookup was successful, router will pass the message to the corresponding InvocationHandler. Otherwise, a “404 Not Found” error message is generated.
4. InvocationHandler passes parts of the payload to each corresponding ParamResolver. If any resolver fails, an “400 Bad Request” message containing an error message is generated.
5. InvocationHandler passes resolved arguments to the Handler function and invokes it.
6. After executing its logic, Handler returns a result back to InvocationHandler, which returns it back to Router.
7. Router returns the message to the Reactor.
8. Reactor serializes the result and generates a success message with UUID of the original message. Message is formed based on chosen messaging pattern and algorithms described in chapter 4.A.

6. CASE STUDY: HTTP TO NATS ADAPTER

In this section we present an adapter implementation that bridges HTTP and NATS protocols. We applied the adapter on a microservice architecture containing 10 different services implemented in Java Spring framework. Gateway services are implemented in NodeJS[19]. Our main goal was to migrate inter-service communication from HTTP to NATS messaging platform. We have chosen to work with NATS mainly because the architecture needed event-based messaging. Another thing is that NATS uses a simple text-based protocol and is easy to setup.

HTTP messages are converted into Flatbuffers[20] objects that are sent as payload via NATS messages. Request messages are converted into NatsRequest objects with the following fields:

- **method** – HTTP method of the original request.
- **path** – All path segments except of the first, as it indicates subscription group of targeted service.

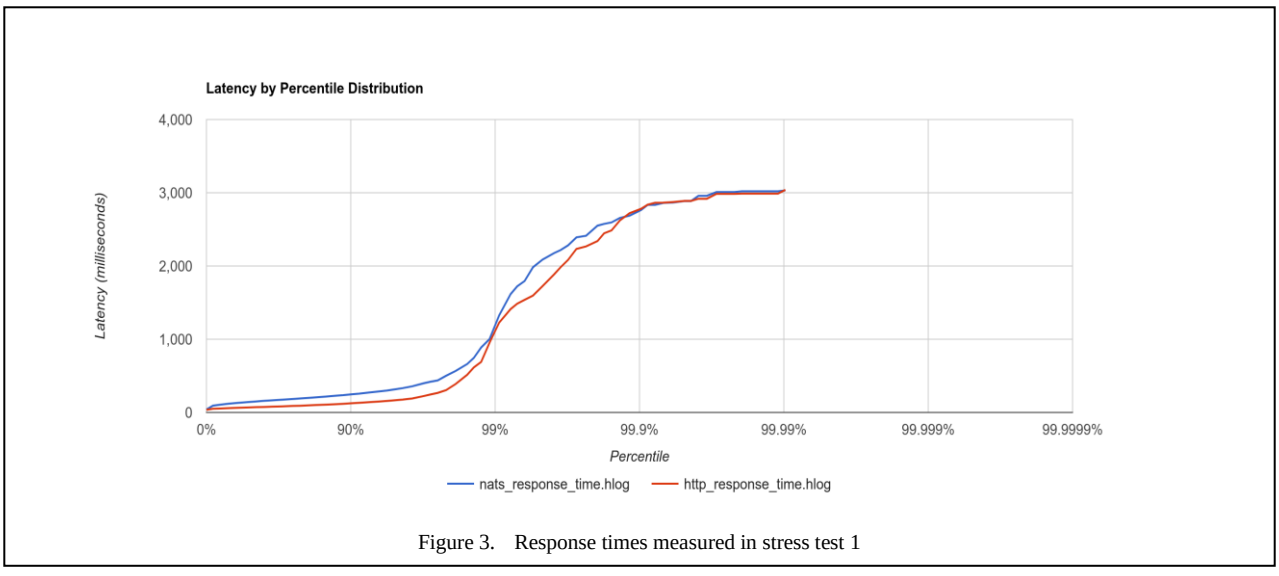

Figure 3. Response times measured in stress test 1

Figure 4. Spring framework endpoint declaration (top) and a message it can accept (bottom)

- **params** – Path parameters of the original request.
- **auth –** Authorization header of the original request.
- **id** – Message UUID.
- **data** – Request body in binary form.

Response messages are converted into NatsResponse objects with the following fields:

- **statusCode** – HTTP response status code.
- **error** – Has error message if there was an error, otherwise has null value.
- **data** – Request body in binary form.
- **id** – Message UUID .
- **rid** – UUID of the corresponding request message.

Message payload is always a JSON object.

Java adapter works with annotation based endpoint declaration. An example is shown in Fig. 4. Users can configure the adapter by specifying which annotation refers to which part of HTTP message. In this particular case, @RequestMapping declares path and method, and @RequestParam declares parameter name and type. If a message fits the specification (Fig.4. bottom), it is translated and passed to the handler.

Two stress tests were used to measure performance of the old system using HTTP and the new system using NATS. Test 1 sends POST requests carrying 390KB of JSON payload data, and test 2 sends small GET requests with no payload. Response times for test 1 are shown in Fig. 3.

Test 1 shows a difference between response times in HTTP favor. The difference is around 100ms and is pronounced in lower percentiles. It becomes nonexistent for high percentiles (over 99.9%). The problem with this test is that HTTP requests directly targeted services without going through service discovery. NATS requests had to go through gateway service and implicitly used service discovery. We conclude that this is the probable reason for the performance drop in test 1. Test 2 shows a large increase in single-node throughput, going from ~3000 requests per second for HTTP to ~10000 requests per second for NATS. We attribute this performance boost to NATS' affinity towards smaller messages[21, 22].

Code changes were minimal. Gateway services included the adapter library and only the function that forwards requests further into the system was changed. Java services also included the adapter, but also had to turn off Tomcat embedded server and to start the adapter server. Removal of Tomcat reduced memory usage of service instances by ~50MB, totaling ~550MB across all services.

7. CONCLUSION

In this paper we presented an adapter that enables smooth transition from one messaging pattern and/or protocol to another. Main goals were to minimize changes to code during this transition, while also preserving performance. We also strive to reduce complexity of the architecture by leveraging on service discovery and load balancing of the new message platform, instead of using separate components for this purpose.

Proposed architecture of the adapter has two layers: gateway layer and conversion layer. Gateway layer consists of existing gateway services that use an adapter library for the language they are written in. We chose this approach over having a separate conversion service in order to reduce the number of network hops in the architecture. Main drawback is that every language requires its own implementation of the adapter library. Conversion layer acts both as a web server, that replaces the old HTTP server, and as a wrapper for existing handler functions. This way logic of web services requires no changes during the transition.

HTTP to NATS adapter for Java Spring web services is presented in this paper. Transition to NATS required minimal code changes. Result of stress tests show an increase in both throughput and latency, both of which can be attributed to the nature of NATS.

Further work includes: 1) full asynchronous messaging support, 2) adapter libraries for more languages and 3) support for more messaging patterns such as scatter-gather.

REFERENCES

[1] C. Labovitz, D. McPherson, S. Iekel-Johsnon, M. Hollyman, Internet Traffic Trends https://www.nanog.org/meeting-archives/nanog43/presentations/Labovitz_internetstats_N43.pdf, accessed in January 2018.

[2] https://github.com/Netflix/eureka, accessed in January 2018.

[3] http://zeromq.org/, accessed in January 2018.

[4] https://nats.io/, accessed in January 2018.

[5] https://kafka.apache.org/, accessed in January 2018.

[6] K. Okumura, A formal protocol conversion method, SIGCOMM '86 Proceedings of the ACM SIGCOMM conference on Communications architectures & protocols, pp 30-37, ACM New York, NY USA, ISBN 0-89791-201-2, New York, USA, 1986.

[7] P. Green, Protocol Conversion, IEEE Transactions on Communications vol 3, issue 3, pp 257-268, 1986, ISSN 0090-6778

[8] S.S. Lam, Protocol Conversion, IEEE Transactions on Software Engineering, vol. 14, pp. 353-362, 1988, ISSN 0098-5589

[9] S. Keneddy, R. Steward, P. Jacob, O. Molloy, StoRHm: a protocol adapter for mapping SOAP based Web Services to RESTful HTTP format, Electron Commer Res, vol 11, issue 3, pp. 245-269, 2011, ISSN 1389-5753

[10] Le K.D.H Towards Rules-Based Mapping Framework for RESTful Web Services, In: Drira K. et al (eds) Service-Oriented Computing – ISCOS 2016 Workshops, ISCOS 2016. Lecture Notes in Computer Science, vol 10380, Springer, Cham, 2017.

[11] https://github.com/sohlich/nats-proxy, accessed in January 2018.

[12] https://oauth.net, accessed in January 2018.

[13] https://jwt.io/, accessed in January 2018.

[14] R.T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, PhD dissertation, Chapter 5, University of California, Irvine 2000.

[15] http://tomcat.apache.org, accessed in January 2018.
[16] https://spring.io/, accessed in January 2018.
[17] https://redis.io/, accessed in January 2018.
[18] R. de la Briandais,File searching using variable length keys. Proc. Western J. Computer Conf. pp. 295–298, 1959
[19] https://nodejs.org/en/, accessed in Jaunary 2018.
[20] https://google.github.io/flatbuffers/, accessed in January 2018.
[21] https://www.w3.org/TR/soap/, accessed in January 2018.