Enabling Centralized Access to a Reactive Architecture for Hardware Control Systems

Rob Stortelder
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
rob@stortelder.eu

September 3, 2019

Graduation committee

First supervisor	**Dr. A. Fehnker** University of Twente
Second supervisor	**Dr. L. Ferreira Pires** University of Twente
First external supervisor	**ing. P. Noltes** Thales Netherlands B.V.
Second external supervisor	**M. Horsthuis MSc** Thales Netherlands B.V.

Abstract

This study describes the process of introducing an API gateway with cross-cutting features into a reactive architecture. The aim of this implementation is to introduce these cross-cutting features into the architecture without changing the services of the reactive architecture. Examples of cross-cutting features are: security, tracing, monitoring and request composition. The target design required the translation of an asynchronous protocol to a synchronous protocol. This was achieved by identifying conversations based on sequence diagrams of current communications in the architecture of Thales.

A proof of concept implementation was created to verify the findings of this study. In this implementation an API transformation service was introduced which transform the customer-facing synchronous API to the internal asynchronous message based communication. In this proof of concept implementation the cross-cutting features were successfully implemented without any changes needed to the services in the reactive architecture. Benchmark tests showed a promising performance of the implemented API gateway with cross-cutting features.

Contents

1 Introduction

This chapter presents the motivation, objectives, approach and structure of this thesis. Section 1.1 describes the motivation for this research. Section 1.2 presents the general objectives. Section 1.3 contains the approach that was followed to achieve the objectives. Section 1.4 presents the further structure of this thesis.

1.1 Motivation

There is a trend of moving toward distributed architectures as an alternative to monolithic architectures. These distributed architectures introduce a separation of concerns by dividing the business logic across services. *Reactive systems* can be designed as distributed systems, which are based on a set of architectural design principles. These principles state that a reactive system should be responsive, resilient, elastic and message-driven [8]. Such systems are capable of responding quickly and consistently, stay responsive in the face of failure, adapt to varying changes in input rate and rely on asynchronous message passing.

This study is aimed at a variant of reactive architectures which is focused on control systems for specialized hardware (e.g., radar systems). The architecture should still be responsive, resilient and is message-driven, but the interpretation of elasticity is slightly different. The architecture should still be able to react to a varying amount of load, but there is a fixed hardware limit to scaling possibilities. This is because hardware systems have a fixed amount of computing power available, so that the fixed amount of available resources should be allocated as effectively as possible. Hardware systems mainly receive control commands from a couple of users, so they do not have to handle millions of requests from users. These request can still require a high amount of computing power. An example is in case a high amount of sensor-data needs to be processed.

Customers of hardware control systems preferably have a centralized entry point for uniform communication with the hardware system. To achieve this, an API gateway can be implemented that offers a single entry point to the architecture. Figure 1 shows an example of a reactive architecture with an API gateway that facilitates access to this architecture. There are several identifiable elements in this architecture:

- **Client:** External system communicating with the reactive system. The client accesses the reactive architecture through the API gateway.
- **API gateway:** Entry point to the reactive system. This translates a single incoming API to messages for the reactive system.
- **hardware control service:** Service running inside the reactive system. Multiple of these services control the hardware system together. The services communicate using asynchronous message passing.
- **Reactive system:** Combination of multiple hardware control services and a communication channel.

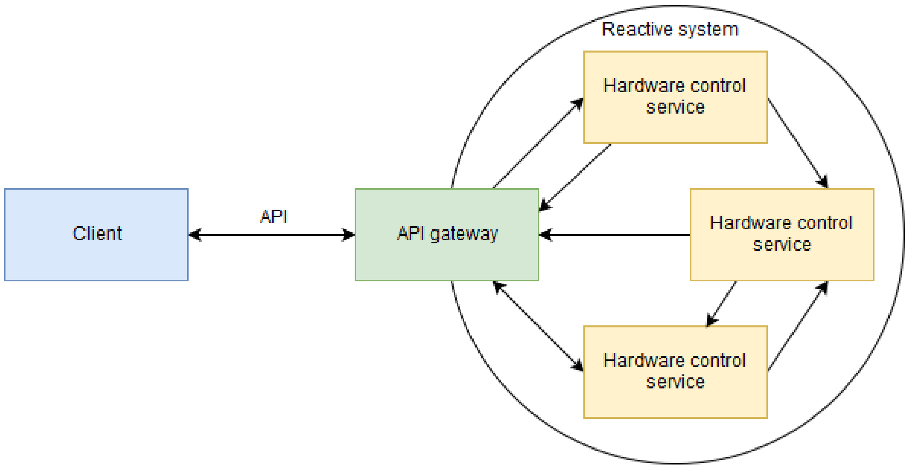


Figure 1: Reactive architecture with an API gateway

Many design choices can be made while implementing such an API gateway. The API gateway can provide different kinds of synchronous or asynchronous APIs to the clients. The gateway can also provide a wide variety of middleware features like security, logging and traceability. The following sections describe the different characteristics of a reactive system and how an API gateway can comply with these characteristics.

1.1.1 Responsiveness

Reactive systems should provide consistent response times to clients. In the case of hardware control systems, control commands and responses from the hardware should arrive in a timely manner. Delays can be introduced by the API gateway, as it mediates access from and to the reactive architecture. For websites, the delays for usability are in the range of a couple of hundreds milliseconds [10]. For hardware control systems, this can even be less due to the possibility of other hardware systems controlling the reactive system. The API gateway should only add a minimal delay to the total delay of the system, preferably in the order of magnitude of 10 milliseconds or less.

1.1.2 Resilience

Reactive systems should be designed to be resilient, stay functional in the face of failure. Failures should be contained in single components and not cascade beyond their boundaries. Besides the architecture being resilient, the API gateway should also be resilient, it should be able to handle failure of services. The API gateway should also be replicated to prevent having a single point of failure.

1.1.3 Elasticity

Elasticity means that the system should stay responsive under a variety of different loads. In case of hardware control systems, the load does not necessarily originate from the amount of client requests, but the high processing load that such a request requires. For example, the client might request the processing of sensor data, which can be a very complex task. The system needs to allocate available resources in order to be able to process these data in time. The API gateway should also be elastic and monitor the reactive system, this monitoring information can be used to prioritize requests in case the system is being overloaded.

1.1.4 Message-driven

Communication between services in a reactive system is based on asynchronous message passing. Services in the reactive architecture can intercommunicate by using some kind of message passing system like a message bus. The API gateway should translate the incoming API to messages that are used inside the reactive architecture. The gateway can provide a variety of API options to clients, among these are synchronous and asynchronous APIs. An asynchronous API is more easily translatable to the reactive architecture as this architecture is based on asynchronous message passing. Providing a synchronous API requires code that is able to translate the synchronous API requests to asynchronous messages.

1.2 Objectives

In this research we worked towards solutions for common challenges when introducing an API gateway into a reactive architecture for hardware control systems. This API gateway should preserve the basic characteristics of the reactive architecture (responsive, resilient, elastic and message-driven). Several challenging areas have been identified for introducing a gateway in a reactive architecture: protocol translations, cross-cutting features and performance.

> *Main objective: Design and implement an API gateway into an reactive architecture for hardware control systems whilst the basic characteristics of the reactive architecture are preserved.*

There are several challenges when implementing an API gateway in a reactive architecture. The first challenge lies in protocol translations, which are challenging due to a couple of factors. There can be a difference between the protocol requirements between the client and the gateway and the protocol from the gateway to the reactive architecture. The reactive architecture is asynchronous in nature, but a client might require a synchronous protocol, which requires the translation from an asynchronous to a synchronous protocol. Message formats from the client can be different to the formats of messages used in the reactive architecture, the gateway should transform these messages. Another element of protocol translations is request composition, where one incoming request is translated to multiple internal requests and their results are combined.

> *Sub-objective 1: Research methods for protocol translations from a protocol designed for a reactive architecture to a protocol suitable for use with an API gateway.*

The second challenge lies in the variety of cross-cutting features that an API gateway can provide for the architecture. Examples of such features are logging, monitoring, authentication and security. The API gateway can provide such features in a centralized manner without having to change individual services significantly. This can be particularly useful when the reactive system is based on legacy technologies and is hard to change. Solving these challenges by using an API gateway can introduce delays on requests to the system. These delays should be minimized where possible, to minimize the impact on the current responsiveness of the architecture.

> *Sub-objective 2: Introduce cross-cutting features using the API gateway without significant code changes to the existing services and minimal overhead*

The third challenge is to validate the findings of this study. For this validation, an API gateway has been implemented in the sensor management system at Thales Netherlands B.V (further in this study referred to as Thales). The aim is to use an off-the-shelf open-source API gateway as an alternative to their currently implemented proprietary API gateway, which provides limited functionality. Off-the-shelf solutions are preferred as they do not require the development and maintenance effort in contrast to in-house development of an API gateway [5]. The solution should preferably be open-source in order for Thales to be able to make changes to the system if needed.

Sub-objective 3: Validate the findings by implementing and testing an API gateway prototype with protocol translations and cross-cutting features at the sensor management system of Thales

1.3 Approach

Several steps have been taken to achieve the objectives described in the previous section:

1. **Domain analysis** The domain of API gateways has been analyzed using a literature study. This analysis consists of two parts: an analysis of API gateway implementations by other software companies and an analysis of open-source API gateways concerning their available protocols and features.

2. **Thales system analysis** An analysis of the current architecture of Thales has been conducted. This analysis shows the challenges and use cases of the current API gateway implementation.

3. **Analyze and implement protocol translations** A literature study into protocol translations has been conducted. This study looks into translating a proprietary protocol to a protocol that is suitable for use with an existing API gateway. After this analysis the protocol translations have been implemented in the reactive architecture of Thales.

4. **Select and implement API gateway** An API gateway has been selected based on results of the domain analysis and this API gateway have been included in the reactive architecture of Thales. Several cross-cutting features have been implemented using this API gateway, some examples are: security, logging and traceability.

5. **Validate implementation** The implementation has been validated on a variety of factors. The first is the degree with which the cross-cutting features could be implemented in the API gateway without significant changes to the services. The performance impact from the implemented API gateway and cross-cutting features has been tested using a benchmarking tool. Lastly the design choices and possible alternatives have been discussed discussed.

1.4 Report Structure

The further structure of this report is as follows:

- Chapter 2 discusses API gateways.
- Chapter 3 introduces the current architecture of the Thales sensor management system.

- Chapter 4 discusses protocol translations from an asynchronous protocol to a synchronous protocol.
- Chapter 5 describes the approach for the new implementation of the synchronous API, with API gateway and cross-cutting features.
- Chapter 6 validates our result of the API gateway implementation.
- Chapter 7 concludes the report and identifies future work opportunities.

2 Background

This section analyses API gateways and their usage in practice. Section 2.1 discusses the usage of API gateway by other companies. Section 2.2 analyses open-source API gateways in terms of their protocols and features. Section 2.3 describes synchronous and asynchronous communication characteristics. Section 2.4 contains concluding remarks for this chapter.

2.1 API Gateway Usage

API gateways are used by companies mostly to provide a single access point to a micro-services architecture. Two well known examples of API gateway usage are those of Netflix and Lyft, which are described in the next sections.These examples are picked as both companies introduce a new approach or technique for using API gateways. Netflix uses the API gateway to provide specific APIs for the different kinds of clients that run the Netflix application. Lyft re-purposes their API gateway as a sidecar service to separate the communication logic from the business logic. A sidecar service separates communication logic from business logic in services

2.1.1 Netflix

Netflix is currently one of the largest media streaming services worldwide, and provides millions of users on a daily basis with media content. Employees of Netflix regularly post entries on the Netflix techblog. This blog contains articles about technical details of the software infrastructure at Netflix. There are several entries on this blog about the API gateway implementation of Netflix [12, 25]. The Netflix app is available on a wide variety of devices, which all need to communicate with the service-oriented architecture of Netflix. Figure 2 shows how they use an API gateway to manage traffic from such devices to their services.

The Netflix API consists of two parts: the client-facing API and the internal API to the services. The API gateway translates between these two parts. Each device communicates with the API gateway using the client-facing API. Request composition is used extensively to reduce the amount of requests from the client to the server. In Figure 2 a Playstation device is making one request to the client-facing API of the API gateway, which forwards the request to a variety of services using the internal API. The responses of these subsequent requests are combined into one response, which is sent back to the Playstation. Request composition can reduce the network overhead by combining multiple API requests into one request.

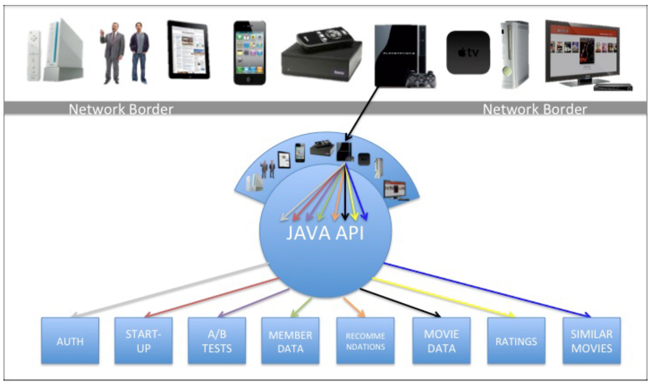


Figure 2: Netflix API gateway usage [25]

The Netflix approach moves the client-server boundary to the API gateway. Client-side developers who program the applications for different devices also develop the connector for translating the client-facing API to the internal API. This connector runs on the API gateway and is written in a server-side programming language. This means that client-side developers also should have some knowledge about server-side development. This approach reduces the dependency from the client teams on the API development teams, but requires the client-side developers to have some knowledge of server-side programming.

Netflix has open-sourced their API gateway implementation in a project called 'Netflix Zuul' [31]. An inbound request passes through a variety of configured inbound filters, then through an endpoint filter, after which the request is forwarded to the destination. After receiving a response from the destination, the response is passed through outbound filters before it is returned to the client. Netflix Zuul offers a variety of integration options with other open-source software projects. Examples of these are integration with Eureka for service discovery (locating services in a distributed architecture) and Ribbon for load balancing (spreading request load among similar services).

2.1.2 Lyft

Lyft is a company that provides ride sharing between its users; at the time of writing they facilitate around one million rides each day. Lyft has been working to transform their monolithic architecture to a distributed architecture. This introduces challenges with service discovery, communication methods and other distributed features. The so called Envoy proxy [29] was introduced to solve these challenges in their distributed architecture. The Envoy proxy can either be used as an API gateway but also as a sidecar service, which is a service that is running besides other services and encapsulates the communication-related features such as service discovery and the use of protocol-specific and fault-tolerant communication libraries from the services themselves. Using the sidecar mechanism, service developers only write business logic in the application and communicate with the sidecar service, this forms an abstraction between business logic and communication logic [26]. Figure 3 shows examples of how Envoy is used, both as an API gateway or as a sidecar service. The 'Front Envoy' services act as an API gateway and provides secure communications between the clients and the architecture. Each application has an Envoy sidecar that facilitates communication between services.

All the communications between components of the system pass through the Envoy proxy. This can be leveraged to implement a variety of cross-cutting features. Examples of these features are service discovery, health checking, load balancing, rate limiting, logging and tracing. Running a sidecar with each service in an architecture comes with a performance overhead. This is why the Envoy proxy is written in a low level programming language (C++), which aims to minimize the performance impact. The developers continually work on improving the performance of Envoy, they argue that the cross-cutting features and abstraction between business logic and networking are worth the (minimized) performance overhead.

2.2 API Gateway Market Analysis

We performed a web search to determine the current state and trends of API gateways. This web search first aimed at creating a list of various open source API gateways. This selection of API gateways resulted from the search queries 'open source API gateway' and 'open source API management. We have only taken open source API gateways into account because these allow for making in-house changes to these API gateways if the need arises. This resulted in

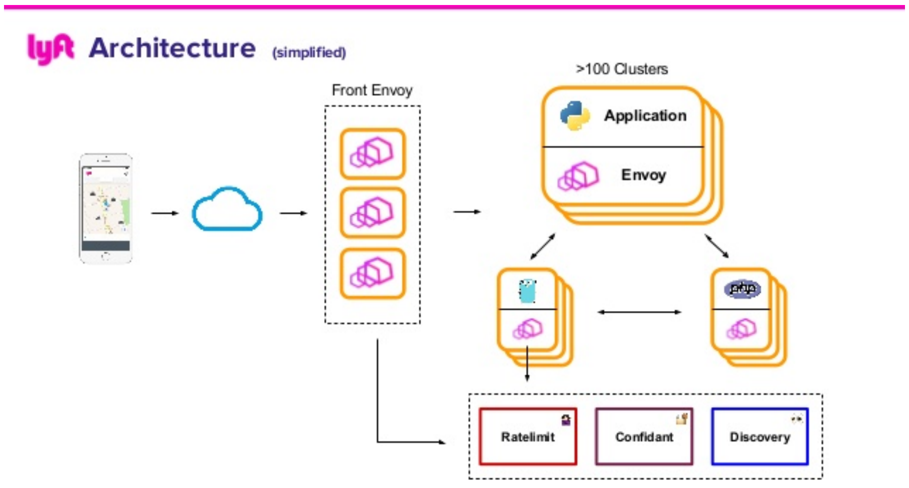


Figure 3: Lyft using Envoy Proxy [30]

a list of 15 API gateways, which is shown in Table 1. This table shows the API gateways, as well as the programming language these have been written in, the release date and whether the API gateway is in active development. In the following sections, these different API gateways are analyzed by looking at the web site, documentation and source code. The first analysis focuses on the protocol support of the different API gateways, the second analysis looks into the features that are provided by the different API gateways.

Table 1: Selection of open source API gateways (March 2019)

No.	Name	Language	Release date	Active development
1	Ambassador (Envoyproxy) [1]	Python	March 2017	Yes
2	API Axle [2]	CoffeScript	December 2012	Inactive since July 2017
3	API Microgateway Strongloop [37]	Javascript	March 2016	Yes, but minimally
4	API Umbrella [42]	Ruby	October 2014	Yes
5	APIMan [3]	Java	September 2014	Yes
6	Dreamfactory [15]	PHP	October 2015	Yes
7	Express gateway [16]	Javascript	May 2017	Yes
8	Fusio API Management [4]	PHP	July 2015	Yes
9	Gravitee [21]	Java	October 2015	Yes
10	Kong [27]	Lua	March 2015	Yes
11	Netflix Zuul [31]	Java	May 2013	Yes
12	Traefic [13]	Go	September 2015	Yes
13	Tree Gateway [41]	TypeScript	May 2017	Yes
14	Tyk [39]	Go	August 2014	Yes
15	WSO2 API Manager [44]	Java	March 2015	Yes

2.2.1 Protocol Support

The documentation of the 15 API gateways were analyzed to list the protocol support for each API gateway. The results of this analysis can be found in table 2. Four different protocols were identified that were supported by one or more API gateways. Some API gateways support multiple protocols, as were others are specifically build for 1 protocol. Some gateways provide partial support for some protocols, reasons for the partial support can be found in the footnotes. Descriptions of the identified protocols are as follows:

Table 2: Protocol analysis of 15 open source API gateway solutions (March 2019)

	Name	HTTP/REST	WebSockets	gRPC	SOAP
1	Ambassador (Envoyproxy)	Yes	Yes[1]	Yes	No
2	API Axle	Yes	No	No	Yes
3	API Microgateway Strongloop	Yes	No	No	No
4	API Umbrella	Yes	No	No	No
5	APIMan	Yes	No	No	Yes
6	Dreamfactory	Yes	No	No	Yes
7	Express Gateway	Yes	No[2]	No	No
8	Fusio API Management	Yes	No	No	No
9	Gravitee	Yes	No	No	Yes
10	Kong	Yes	Yes	Yes[4]	No
11	Netflix Zuul	Yes	Yes[1,3]	No[2]	No
12	Traefic	Yes	Yes	Yes	No
13	Tree Gateway	Yes	No	No	No
14	Tyk	Yes	Yes	Yes	No
15	WSO2 API Manager	Yes	Yes	No	Yes

[1]Ambassador supports WebSockets since October 2018 and Zuul since June 2018
[2]Mentioned on roadmap
[3]WebSocket connections to the gateway are supported, but not between the gateway and the services
[4]Basic support, only primitive passive proxy-pass is supported

- **HTTP/REST** [18] is supported by all major open-source API gateways and it is widely used in the open internet. HTTP is a stateless synchronous protocol built on top of TCP. Each command is executed independently and the client is blocked until a response is received. Representational State Transfer (REST) is a software architectural style and approach of communications on top of HTTP. It provides a set of guidelines on how to use HTTP according to the REST style. This architectural style was introduced in 2000 in a PhD thesis by Thomas Fielding [19].Server-sent events (SSE) can be used on top of HTTP to facilitate asynchronous server notifications on top of the HTTP protocol. When the clients initiates a request to the server, the connection stays open in order for the server to send multiple responses (notifications) back to the client. SSE is a W3C standard since 2009 [24], it has gone through several iterations since then. Only few API gateways currently officially support SSE; Zuul officially supports SSE and blog posts indicate that users use Gravitee with SSE. In general, if an API gateway supports HTTP then it likely supports SSE.

- The **WebSocket** [17] protocol is supported by Ambassador, Kong, Traefic, Tyk, WSO2, and partially by Netflix Zuul. The WebSocket protocol is a full duplex asynchronous protocol on top of the TCP layer. This allows clients to send messages to the server at any time and vice versa. The WebSocket definition was standardized by the IETF in 2011. A WebSocket server can make use of the same TCP port as HTTP, and the initial

WebSocket handshake takes place over HTTP. After the handshake over HTTP succeeds, the protocol moves to the WebSocket protocol.

- **HTTP2/gRPC** [23] is supported by Ambassador, Traefic, Tyk and partly by Kong. gRPC is an open source universal RPC framework which was originally released in 2016. This protocol was originally developed at Google, but it is now open source. The protocol is based on the new HTTP/2 standard which is a binary multiplexed protocol, and is designed to be more efficient than HTTP/1.1. gRPC can be used with different data formats, but is was originally designed to be used with Protocol Buffers. Protocol buffers are a language-neutral mechanism for serializing structured data, where data structures are defined once and code can be generated for a variety of languages to use these data structures. HTTP/2 supports server push, in which the server can send data to the clients before a client requests this data.

- **SOAP** [9] is another protocol which is also generally used over HTTP. SOAP is mainly focused on XML-based message exchange. SOAP was originally standardized in 2000 by W3C. API gateways mainly advertise with the translation from a legacy SOAP API to a REST API. This shows that REST based web services are a more popular choice over SOAP.

2.2.2 Functionalities

Just as with the protocol analysis, the features of the list of open source API gateways has also been analyzed. The results of this analysis can be found in Table 3. This shows a list of common features in API gateways and the support of these features by the different API gateways. There are a couple of question marks in Table 3 as it is unclear if the gateways fully support these features. Sometimes the feature is not officially supported by the gateway but underlying technologies can likely provide this feature. In other cases, the community describes complex workarounds or the use of some tooling to support the feature.

Table 3 clearly shows that Monitoring, authentication and rate limiting are very common features in API gateways as all the API gateways support these features. Other pretty common features are header/body transformations load balancing and plug-ins. Traceability and request composition are supported by a couple of API gateways.

Some features can also be supported by tooling outside of the API gateway. An example of this is using a load balancer after the API gateway to balance load between multiple instances of the same service. In this way, the API gateway can simply forward the request to the load balancer. The same principle holds for the lesser supported feature of request composition, since a dedicated service can take care of request composition and the gateway can send a request which is translated to multiple requests through this service. Another solution is to implement plug-ins for API gateways to provide specific features, although this is only possible if the API gateway allows functional extensions or modifications using plug-ins. 9 of the 15 analyzed API gateways support plug-ins to extend the functionalities of the API gateway.

Table 3: Feature analysis of 15 open source API gateway solutions (March 2019)

		Monitoring/analytics	Traceability	Authentication/authorization	Rate limiting	Header transformations	Body transformations	Request composition	Load balancing	Plug-ins
1	Ambassador (Envoyproxy)	Yes	Yes	Yes	Yes	Yes	No	No	?	Yes
2	API Axle	Yes	No	Yes	Yes	No	No	No	Yes	No
3	API Microgateway Strongloop	Yes	No	Yes	Yes	?	Yes	Yes	No	Yes
4	API Umbrella	Yes	Yes	Yes	Yes	No	No	No	Yes	No
5	APIMan	Yes	No	Yes	Yes	Yes	Yes	No	No	Yes
6	Dreamfactory	Yes	No	Yes	Yes	Yes	Yes	No	No	No
7	Express gateway	Yes	?	Yes	Yes	Yes	Yes	No	Yes	Yes
8	Fusio API Management	Yes	No	Yes	Yes	No	No	No	No	No
9	Gravitee	Yes	No	Yes	Yes	Yes	Yes	No	Yes	Yes
10	Kong	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
11	Netflix Zuul[1]	Yes	No	Yes	Yes	Yes	Yes	?	No	Yes
12	Traefic	Yes	Yes	Yes	Yes	No	No	No	Yes	No
13	Tree Gateway	Yes	No	Yes	Yes	Yes	Yes	No	Yes	Yes
14	Tyk	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
15	WSO2 API Manager	Yes	Yes	Yes	Yes	No	Yes	No	Yes	No

[1]Support is mainly based on community plug-ins, out of the box support is minimal

2.3 Synchronous and Asynchronous Communication

Section 2.2.1 described different synchronous and asynchronous protocols which are supported by API gateways. This section describes the different characteristics of synchronous and asynchronous communication methods as described in literature. Literature describes synchronous communications as communications where the client is blocked until the request is known to be accepted. With asynchronous messaging, the client continues after the message has been submitted to the middleware [38]. There are essentially three points where synchronization can take place. The first point is where the sender is blocked until the middle-ware notifies that it will take over transmission of the request. The second point is that the sender synchronizes until its request has been received by the recipient. The third point is where the sender synchronizes until the message has been processed by the recipient and a response is received.

Asynchronous communication can hide communication latencies. A system can continue doing work instead of waiting for a response. The application is interrupted when a handler is called to complete the previously issued request. Asynchronous communication is mostly used in parallel applications in which tasks continue processing while another task is waiting for communication to complete. With both synchronous and asynchronous communication it is important to implement error handling correctly. With synchronous communication the sender can experience errors or timeouts when the receiver is unavailable. With asynchronous communication it is possible that the sender sends a message to a receiver that is unavailable. Mechanisms in both cases are needed to handle such errors.

Within asynchronous communications it is possible to identify conversations [11]. This can be achieved by analysing interactions between services. A conversation is a sequence of (synchronous) messages exchanged among web services recorded in the order they were sent. An example of a recorded conversation can be that a receipt will always follow after a payment. These conversations can be mapped in state machines which can be used for further analysis and verification of software systems [7, 11, 20]. The advantage is that already available methods like linear temporal logic can be used with these conversations.

2.4 Concluding remarks

This chapter shows the analysis of the current landscape and usage of API gateways. Example uses of API gateways at Neflix and Lyft have been described which show unique or special ways in which they implemented an API gateway. Netflix mainly uses the API gateway to provide APIs specific for each supported device. Lyft uses an API gateway as well as sidecar services to abstract communication logic from individual services. The open source landscape of API gateways has been analyzed on features and protocols. This showed that most gateways have been built for HTTP/REST communications but some do support other protocols. Feature analysis of the API gateways showed that request composition and traceability are features which are minimally supported by API gateways. Features like monitoring, analytics and security are supported by all analyzed API gateways. Lastly the different characteristics of synchronous and asynchronous communications have been described.

3 Thales System Analysis

Thales provides a range of radar and sensor solutions to a variety of clients, specifically in the Naval domain. These radars have to process a large amount of sensor data in a relatively short amount of time. During this study, challenges were identified with the current proprietary gateway implementation used at Thales. This chapter will introduce the current architecture of the sensor management system of Thales. After the introduction of the architecture the current challenges and communications through the API gateway will be analyzed. Section 3.1 describes the current software architecture of the Thales sensor management system. Section 3.2 describes the identified challenges with this architecture. Section 3.3 contains sequence diagrams which capture the different types of conversations between the client and the Thales sensor management architecture through the API gateway. Section 3.4 concludes this chapter.

3.1 Thales Architecture

The Thales sensor management architecture is a reactive architecture that matches the definition of a reactive architecture as introduced in Chapter 1. A component-based development method is applied where functionalities are separated into different logical components. Thales develops components with model-driven development using an in-house developed platform. Components are defined using a DSL (Domain Specific Language), which is used to describe the messages the component produces and consumes as well as some other elements like timers and dependencies.This DSL is developed in house and is currently not open source. From these component descriptions, program code can be generated in a chosen target language, currently supported target languages are C and C++. The underlying communication layer is abstracted away from the developer and can be exchanged for other communication layers.

A schematic of the current architecture is shown in Figure 4. This shows the component-based architecture, where in this example four components are shown. The components intercommunicate using a Data-Distribution System (DDS), which is a data-driven real-time machine-to-machine connectivity framework. A DDS is generally used in many high-performance systems, such as aerospace and defence systems. One component shows an interaction with a hardware system as where another component acts as the API gateway. This API gateway component currently facilitates message transformations based on customer requirements for the API.

All communications in the architecture are currently based on asynchronous message passing. This includes the communications from the client with the API gateway component. Communication to and between components is time-sensitive, as responses to situational changes need to be handled quickly.

3.2 Current Challenges

Thales has identified the following challenges for the current implementation of the proprietary API gateway:

- **Asynchronous client-facing API:** Current communication with the API gateway component are based on asynchronous messaging. Developers at Thales have expressed interest in using an off the shelf API gateway to replace the current proprietary API gateway component to reduce development cost on the current proprietary API gateway. As discussed in chapter 2, most modern API gateways mainly provide full functionalities for

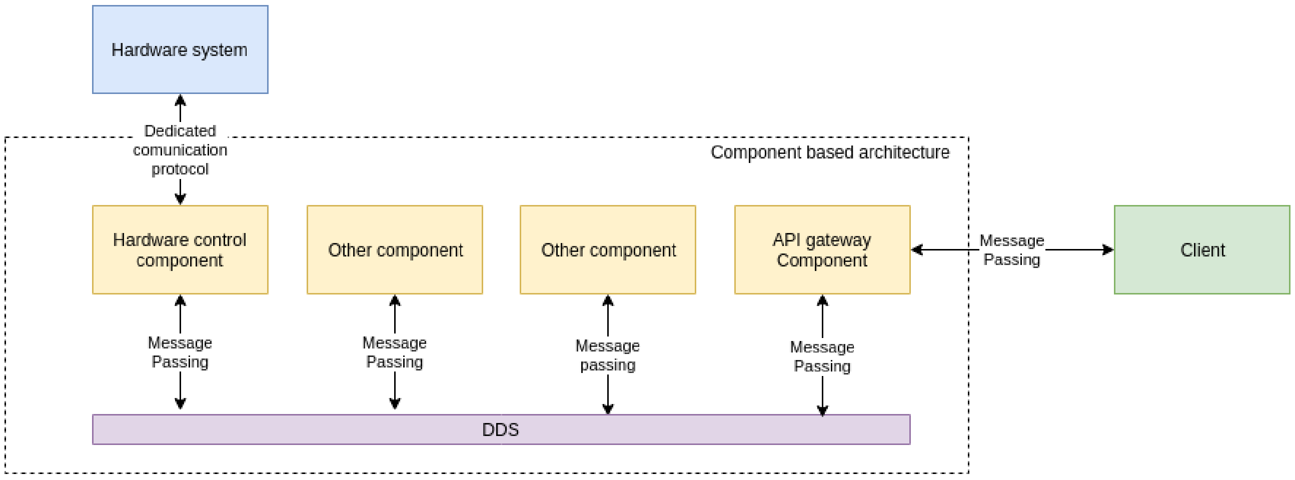


Figure 4: Thales component-based architecture

a synchronous API gateway while support for asynchronous APIs is very limited. Modernization steps towards a more standardized API with modern security features and off-the-shelf support tooling also introduced the interest of an API redesign.

- **Proprietary API:** The API gateway component currently provides a proprietary protocol which is specific for the current reactive system. Modernization steps can be taken in the API gateway to provide customers with a protocol based on open standards. Examples of such protocols are described in Section 2.2.1. The major advantage of using a protocol based on (popular) open standards is that tooling and frameworks are available that work with these standards.

- **User adaptations:** Customers who buy a radar system have specific requirements about the API that Thales provides for the radar. Thales currently programs these adaptations into the API gateway component specifically for these customers. These changes are the so called 'user adaptations'. These adaptations are programmed in stead of configured which can make the adaptations error prone. Off the shelf API gateways generally provide header and body transformations, which can be configured in stead of programmed to handle request or response changes.

- **Request composition:** With request composition, multiple responses can be combined by only sending one request. An example of such a combination is combining wind and temperature information. One component tracks wind measurements, while another component manages temperature measurements. A client may have the requirement that the wind and temperature information is combined into one message. This is achieved by programming the composition logic in the current proprietary API gateway component. An alternative to this is handling composition logic in the new API gateway, but only a couple of API gateways support request composition as described in Chapter 2.

- **Security features:** With changing requirements it is desired that more security features are implemented, like access control, authentication and session management. Due to the high performance nature of the system, security functionality should be performed at the gateway and services can assume the request as trusted from the gateway.

- **State changes:** As mentioned before, the system reacts to changes in its environment and can change state at any point in time. For example, this means that in an extreme case the system can decide that it should be shut down, in which case the internal state changes from 'on' to 'off'. This introduces the challenge that the client should somehow be continually available to receive such changes from the system.
- **Timing constraints:** The API gateway with all the previously described features should not introduce significant delays. Ideally this would be maximum of 10ms according to system experts at Thales. This is the total time of sending a message from the client to the API gateway, which performs all kind of middle-ware features, to the component-based architecture and back.
- **Tracing, logging and monitoring** The API gateway can be used to trace, log and monitor messages passing the API gateway. As the API gateway is a centralized access point it is possible to catch all the communications to and from the reactive architecture.The logging and monitoring features can be used to diagnose problems. Most API gateways provide out of the box logging and monitoring features while some provide tracing features.
- **Maintainability** Thales delivers solutions that sometimes have a life span for up to 30 years. It is very hard to predict whether technologies are still used or available over a 30 year time span. Design choices need to be made that choose technologies that have a likelihood of being supported in the future. This can be achieved by using programming languages and technologies that are currently widely used or are growing in popularity. Another option is to choose open-source software solutions which allow in-house maintenance of the software in the future.

This study focuses on feasibility and proof-of-concept implementation of protocol translations and an off-the-shelf API gateway. The aim is to provide solutions for all the above mentioned challenges, where the aim is to solve most of the challenges using features of the off-the-shelf API gateway. As this research has a focus on feasibility, performance of the system is used as a factor in decision making, but a fully optimized solution is not the aim of this study.

3.3 Sequence Diagrams

This section shows a selection of sequence diagrams of communication types in the current Thales sensor management architecture. These sequence diagrams are selected to showcase communications which are affected by a new API gateway solution. Each section introduces a different kind of conversation between the client, API gateway component and other components. Together these sequence diagrams will cover the different types of conversations that can be identified in the current architecture of Thales. Note that these sequence diagrams do not exactly reflect the implementation of Thales due to the classified nature of the system, but these grasp the essence of the communication types of the Thales system.

The sequence diagrams in this chapter show the currently implemented fully asynchronous communications. As stated by the UML standard [22], asynchronous communications are indicated with an open arrow as where synchronous communications are indicated with a closed arrow. The sequence diagrams in this chapter are solely based on asynchronous communications, thus only open arrows are used.

3.3.1 Request Current State

The client can request the current state of the system, the gateway keeps the last system state in memory so the system state can almost directly be returned. New state changes are automatically pushed to the client, this request is mainly used when the client is initializing or had been reset. An example of this conversation is shown in Figure 5 which shows the gateway which listens to state changes of the system and stores these in memory. After the client request the current state a message is emitted by the gateway which contains the state.

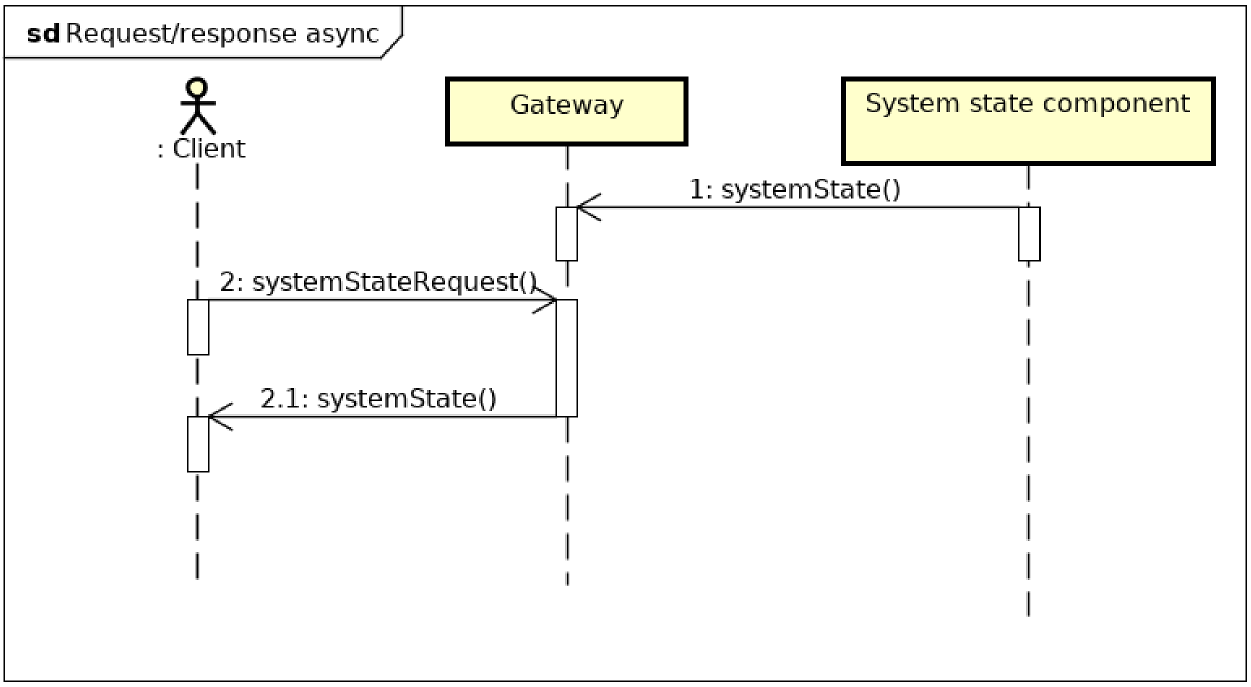


Figure 5: Request state sequence diagram

3.3.2 Request Composition

Request composition is a situation where information from multiple components is combined into one result message. Figure 6 shows an example of request composition which is facilitated by an API gateway. This shows the combination of wind and temperature information into one response message which shows the current weather state. The gateway combines this information into one result message which is then sent to the client.

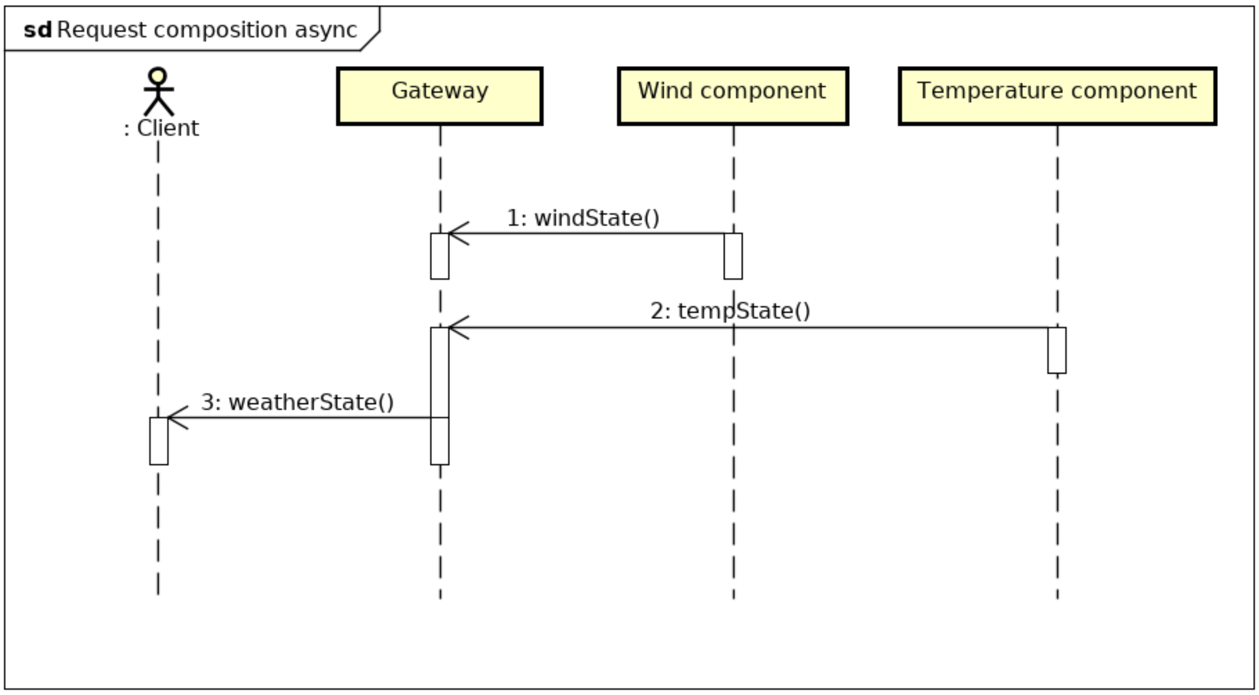


Figure 6: Request composition sequence diagram

3.3.3 Power on Subsystem

A client can request a subsystem to power on. In this case two responses from the gateway are expected. Both responses are that of the state of the subsystem, the first indicates that the current state is off, but the target state is on. After a while a second response is expected that indicates that the current state is on and the target state is on. The sequence diagram of this interaction is shown in Figure 7. The client actor sends an asynchronous message to the gateway with a command to power on the system. The gateway forwards this command to the subsystem after which the gateway returns an asynchronous message with the current state of the subsystem, which is currently off but the target state is on. The subsystem will start turning on and after a while will emit a message that the system has been turned on.

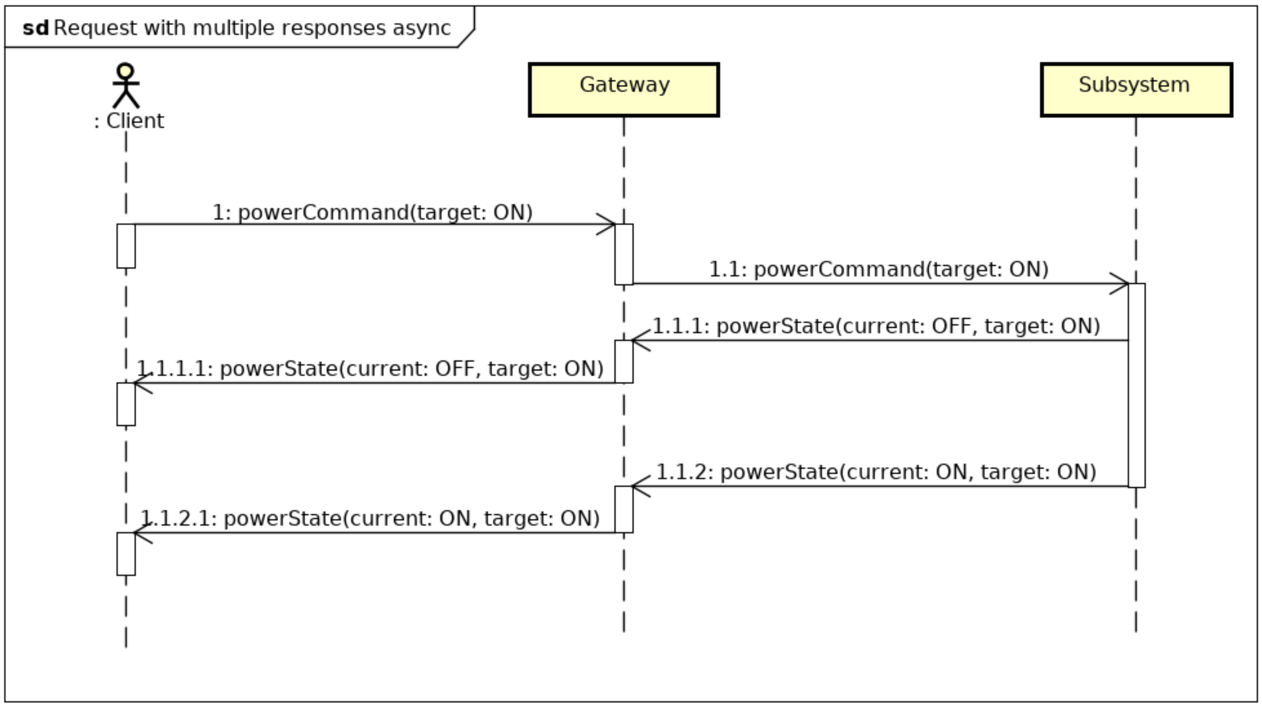

sd Request with multiple responses async
: Client
Gateway
Subsystem
1: powerCommand(target: ON)
1.1: powerCommand(target: ON)
1.1.1: powerState(current: OFF, target: ON)
1.1.1.1: powerState(current: OFF, target: ON)
1.1.2: powerState(current: ON, target: ON)
1.1.2.1: powerState(current: ON, target: ON)

Figure 7: Changing sensor mode sequence diagram

3.3.4 Periodic Task

A periodic task is a task that can be enabled or disabled by a client, multiple predefined periodic tasks are available. It is not possible for users to add custom periodic tasks. Each task is identified with an ID and after enabling a task the task will send updates about the task with a regular interval. The sequence diagram for this interaction is shown in Figure 8, which shows the client making a request to the gateway to start a periodic task. The gateway forwards this message to the task component which starts sending task update messages with a fixed interval. The client can send a cancel task message though the gateway to the task component to cancel this task. This example shows the interaction for the task with ID 1, multiple tasks with different IDs can be enabled simultaneously.

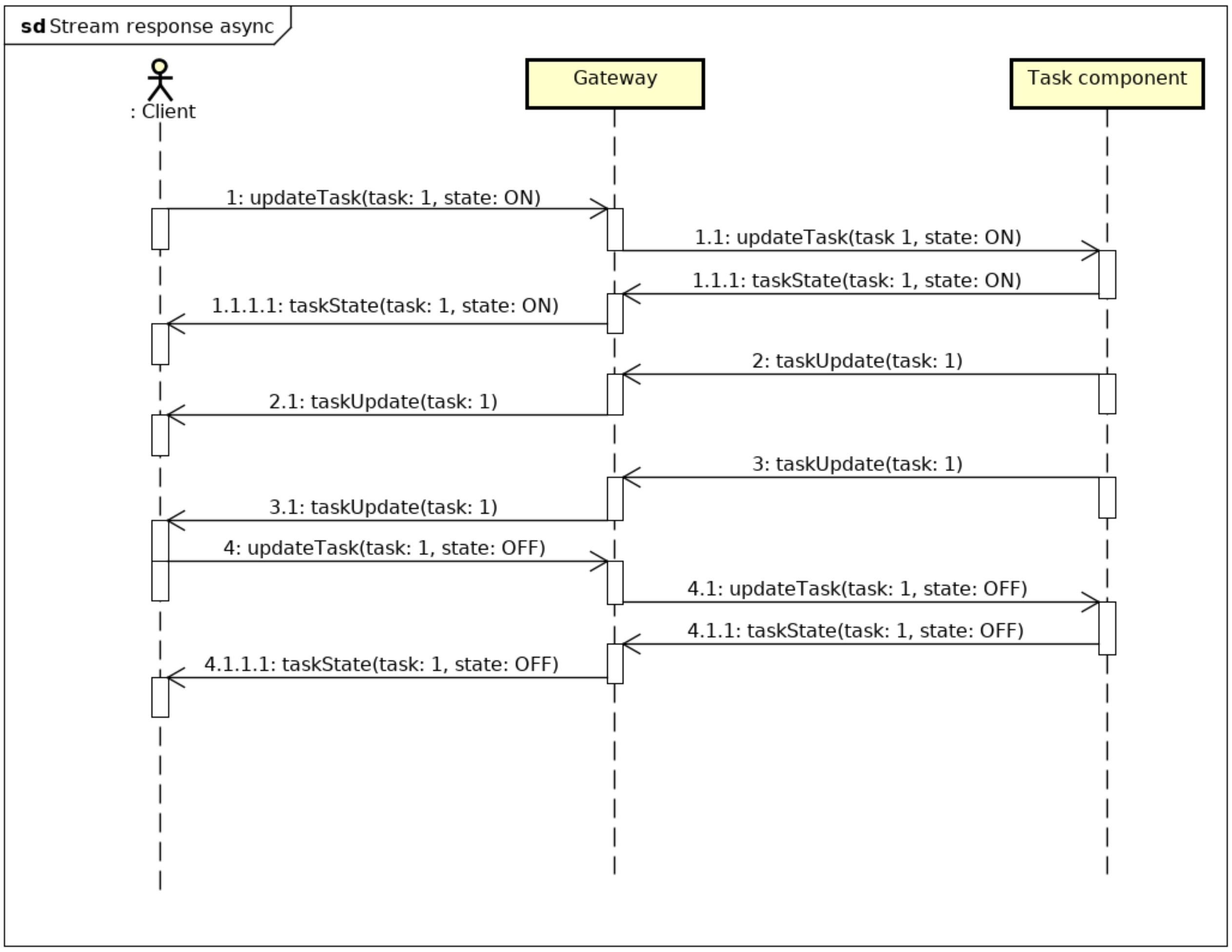

sd Stream response async
: Client
Gateway
Task component
1: updateTask(task: 1, state: ON)
1.1: updateTask(task 1, state: ON)
1.1.1: taskState(task: 1, state: ON)
1.1.1.1: taskState(task: 1, state: ON)
2: taskUpdate(task: 1)
2.1: taskUpdate(task: 1)
3: taskUpdate(task: 1)
3.1: taskUpdate(task: 1)
4: updateTask(task: 1, state: OFF)
4.1: updateTask(task: 1, state: OFF)
4.1.1: taskState(task: 1, state: OFF)
4.1.1.1: taskState(task: 1, state: OFF)

Figure 8: Periodic task sequence diagram

3.3.5 Spontaneous State Change

A sensor can change state at any point in time. This can either be due to environmental factors or hardware failure. This state needs to be propagated to the clients for the states to be synchronized between the hardware monitoring component and the client. This is shown by a sequence diagram in Figure 9. Due to some outside factor, the hardware monitoring components emits a message that the state has changed. The gateway receives this message and subsequently forwards this message to the client after which the client can act upon this state change.

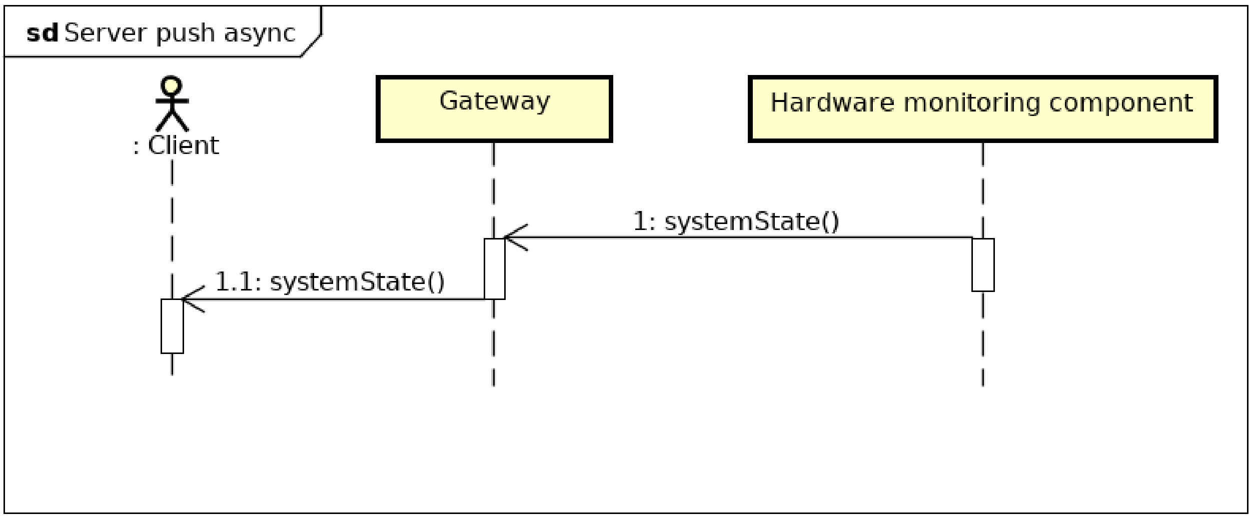


Figure 9: A state change from the hardware monitoring component to the client

3.4 Concluding remarks

This chapter introduced the Thales component-based architecture which is based on fully asynchronous messaging. The communication between the client and the current proprietary API gateway contains several challenges. These challenges can largely be solved with an API redesign and an off-the-shelf API gateway solution. Several sequence diagrams are described which together grasp the main types of conversations between the client and the component-based architecture. The following chapters will introduce translations to a new API design and an API implementation with the aim to solve the above mentioned challenges while still support the conversations as shown in the sequence diagrams.

4 Protocol Translations

This chapter defines and translates the current asynchronous message passing protocol to a protocol that works with commercial off-the-shelf API gateway solutions. Most API gateway solutions focus on REST communications and most of the features that are supported by the API gateways are designed for REST/JSON protocols as described in Section 2.2. Besides API gateways there is tooling available that allows for client and server side code to be generated from RESTful API descriptions [33]. Several publications in the literature analyses the conversations among asynchronous web services [6, 7, 11, 20]. Section 4.1 identifies the different types of conversations in hardware control systems. Section 4.2 introduces several push technologies based on HTTP that support methods for pushing data from the server to the client. Based on the conversation types and chosen push technologies, translations to synchronous HTTP/REST are defined in Section 4.3. Section 4.4 shows that similar conversations can be applied to other hardware control systems. Concluding remarks are described in Section 4.5.

4.1 Conversation Types

This section presents the different conversation types that have been identified by analysing the sequence diagrams in Chapter 3. Four different conversation types were identified, these are 'request/response, 'request with fixed amount of responses', 'request with stream of responses' and 'server push'. Each conversation types introduces a different challenge for protocol translations. The following sections will describe the conversion types in more detail.

State machines for each conversation type have been created. These state machines are used to help illustrate the conversations for the following sections. These state machines are shown in Figure 10 [6]. Each conversation is between the client and the server, where the server in this case is a reactive architecture. In the figure, the reactive architecture is depicted as a single server, but in reality the server consists of multiple asynchronous services. State transitions are indicated with an arrow from one state to another state, where the final state is indicated with a circle around the state. The start state is indicated by a small circle with an arrow. State transitions are labeled with a question mark (?) or an exclamation mark (!), which indicates an incoming or an outgoing message, respectively. In the first conversation (request/response), the initial state has a transition labeled with '!r' to another state, this indicates an outgoing request 'r' to the server. From that state a transition labeled '?a' indicates that the client waits for an answer 'a' from the server. The following sections describe each identified conversation type separately.

4.1.1 Request/Response

The first type is blocking request-response messaging, where the client sends a request to the server and waits until a response is received. With asynchronous message passing at the server side, this is implemented by the client waiting for a response message on the communication channel. The client finalizes the conversation when a response is received or when a time-out occurs. An example of an synchronous request/response in the Thales case is that of the client requesting the state from the system. The client sends a message which requests the current state of the system as described in Section 3.3.1. Eventually, the client receives a message that contains the state of the system. The client is blocked while waiting for this synchronous response.

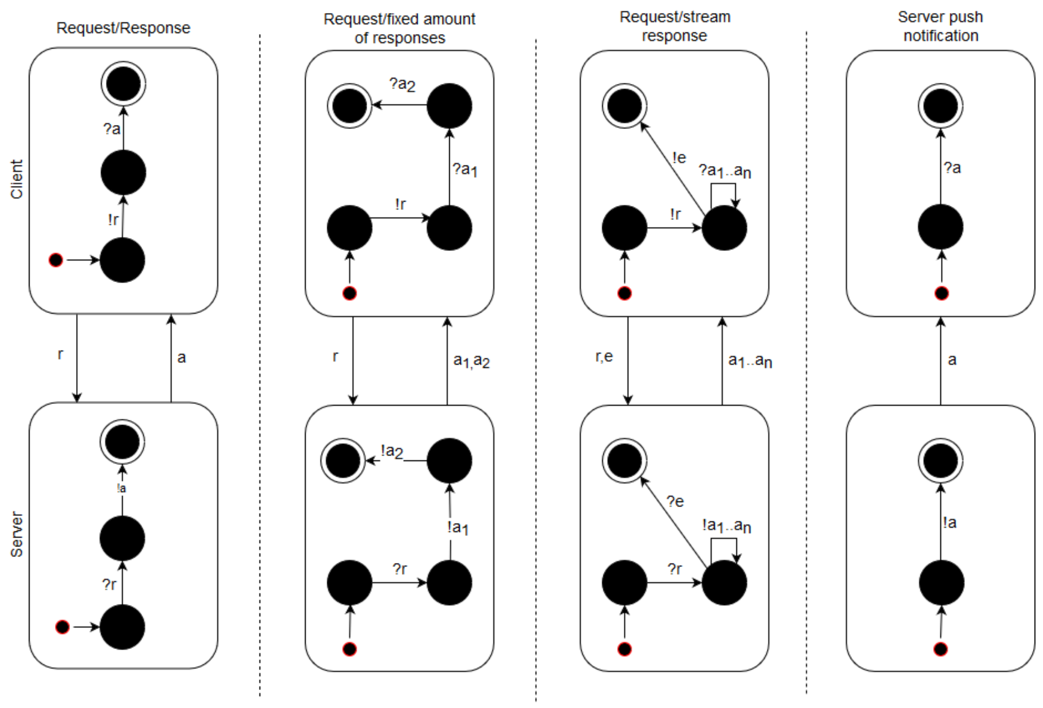


Figure 10: Conversation state machines

4.1.2 Request with fixed amount of Responses

This is a request to the server in which multiple responses are expected by the client. The amount and types of the responses are known beforehand by both the client and the server. An example of such a request in the Thales system is that in case of powering on a subsystem, as described in Section 3.3.3. In this example the client expects two responses, which are known by the client and server beforehand. The client blocks until at least the first response has been received. Figure 10 illustrates this with one outgoing request from the client, indicating with !r and two responses, a1 and a2. In this image the order of the responses is fixed, but it can also be designed in way that responses can arrive in any order.

4.1.3 Request with Stream of Responses

This is a request that yields a stream of responses with a regular interval. An example of this in the Thales system is the execution of a periodic task, as described in Section 3.3.4. The system will then emit messages with a regular interval. The messages will continue from the server to the client (messages a1 till an) until the client ends the conversation, in Figure 10 indicated with the label 'e'. The client listens for these messages and processes these messages. This can be in a separate blocking thread, but does not necessarily have to be blocking.

4.1.4 Server Push

The last type of conversation is the server push. An example of this is a state change due to environmental factors or hardware failure, after which system can automatically turn off in case of hardware failure. This is described in Section 3.3.5. In this case the client controlling the system should be informed of this state change. This communication is initiated by the server and does not require an acknowledgement from the client. Figure 10 shows this by a server initiated message which the client receives. As this is a server push without acknowledgement from the client, this communication is non-blocking.

4.2 Push Technologies

As some of the conversations require the server to push data to the client, specifically with the server push conversation, some kind of push technology is needed. There are three categories of push technologies which can be used to propagate asynchronous messages to clients. These examples are derived from push technologies found in literature [28, 35]. Each of the three categories in the following sections contain one or more examples of such push technologies.

4.2.1 Requesting changes

The first category is **requesting changes**. In this case, the clients initiates a request to the server for new available changes. This can either be done by **polling** with a set interval or by using **long polling**. By polling the clients requests the information continually, when the previous request completes. The server responds directly even if no change is available. With long polling the client makes a request to the server and the server responds when a change is available. This means that the request will stay open until a timeout occurs or if there is new data available, in both cases a new long poll will be opened when the previous one closes. Polling in both cases consumes resources for unnecessary polls or unnecessary long open standing connections. An advantage of (long)polling is that is generally works with existing firewall configurations and uses the same technologies as with normal API requests. Long polling is a fully synchronous method, the client makes requests and the server answers these requests.

4.2.2 Continuous open connections

The second category are **continuous open connections**. In this approach, the client initiates a connection to the server that stays open in which multiple responses can be received. Examples of such protocols are **WebSockets**, **Server-Sent Events (SSE)** and **HTTP/2**. In all cases, there is a connection to the server on which multiple responses can be received. With WebSockets the communication is based on a bi-directional TCP connection, messages can be passed between both parties at any time. SSE is unidirectional, the client opens a request to the server and the server can send multiple responses over this open connection. The client can not sent multiple requests using the same connection. HTTP/2 is the new version of the HTTP/1 protocol released in 2015. This is based on a binary protocol over TCP, which also allows the server to push data to the client using the opened TCP connection.

4.2.3 Client-as-a-server

The third and last category is that of **client-as-a-server**. The client in this case acts as a server and is reachable by the server, the server can send messages directly to the client. Examples of techniques using this are **WebSub** (formally known as PubSubHubbub) or **Webhooks**. In both cases, the client needs to provide an HTTP server for the server to send messages to. This introduces some addition code and configuration, specifically in the firewall.

4.2.4 Applying to hardware control system

As we are dealing with hardware systems, it can be assumed that the number of clients is limited, the network with which the hardware system is introduced is outside of the control of the hardware system and the push technology should be supported by API gateways. The choice has been made to implement a mechanism based on long polling, due to four main reasons as listed below.

1. The first reason is that long polling is a fully synchronous method, in contrast to something like WebSockets. The aim of this study is to deliver a fully synchronous protocol, so long polling perfectly fits this requirement.
2. The second reason is that long polling is naturally supported by API gateways, in contrast to many other methods. Some gateways support pass-through of WebSocket connections but do not provide message transformations or other features on WebSocket connections. Support for SSE in API gateways is in most cases not officially documented but SSE may work as it is closely based on HTTP communications.
3. The third reason is that the amount of clients is very limited, which means that the long polling overhead is also limited. Clients continuously poll the server, which can be a drain on resources. With only a couple of clients this will not have a significant impact.
4. The last reason is that long polling does not require any specific firewall configurations besides the already made configurations for HTTP communication. The client-as-a-server approach does require network configurations for running a reachable HTTP server and introduces implementation complexity at the client side.

4.3 Translations

This section will describe translations to a synchronous protocol for each conversation type as described in Section 4.1. The sequence diagrams in this section are synchronized variants of the sequence diagrams described in Section 3.3. In this section the conversations between the client and the gateway are synchronous, which means a closed arrow is used to indicate a request and a dotted arrow is used to indicate a response. The communication between the gateway and the components is still asynchronous in nature, thus indicated with the open arrows.

The design follows the REST principles. A REST API is designed around resources, a client can modify these resources. A resource is available through an URI, an such as: example.com/products/1. A REST API is designed to be stateless, the server does not store any state about the client between requests. There are several operations for resources, which are: GET, POST, PUT, PATCH, and DELETE. Operations are called idempotent when multiple identical requests have the same effect as making one single request. Examples of idempotent operations are GET, PUT and DELETE. GET and DELETE are idempotent as getting the same object does not change it, and deleting the same object multiple times will have the same result as deleting it once. PUT is idempotent because it fully replaces a resource, so that executing the full replacement multiple times yields the same result. PATCH can be idempotent but it is not necessarily idempotent. An idempotent example of the use of a PATCH request is to change the name of user 1 to 'Rob'. This can be executed multiple times and this will have the same result. PATCH is not idempotent, for example, when the request appends the text 'working at Thales' to the description of Rob. If this request is executed multiple times, the text will be appended multiple times. A POST request is not idempotent as it is only used to create a new resource. Executing the same POST request multiple times will result in multiple resources being created.

The following subsections describe synchronous sequence diagrams for each different conversation type. These sequence diagrams are synchronized versions of the sequence diagrams as described in Section 3.3. Where possible the REST communications are set up in an idempotent way, which will reduce the risk of undesired modifications when a request is accidentally sent multiple times.

4.3.1 Request/Response

The system can make a synchronous GET request to get the current state of the system. Just as described in Section 3.3.3, the gateway is subscribed to the system state updates and will have the latest system state in memory. The current state can directly be returned when a GET request is received by the gateway, this is illustrated in Figure 11. An alternative is that the gateway does not keep the current state in memory. In that case the state is requested from the system state component by the gateway when the GET request is received. Then the response message from the system state component is returned to the client. A time-out in the request from the client to the gateway can occur when no response is received or when the response is delayed. Both approaches are fully idempotent as these are only based on GET requests and will not change any data. Multiple executions of these requests will yield the same result.

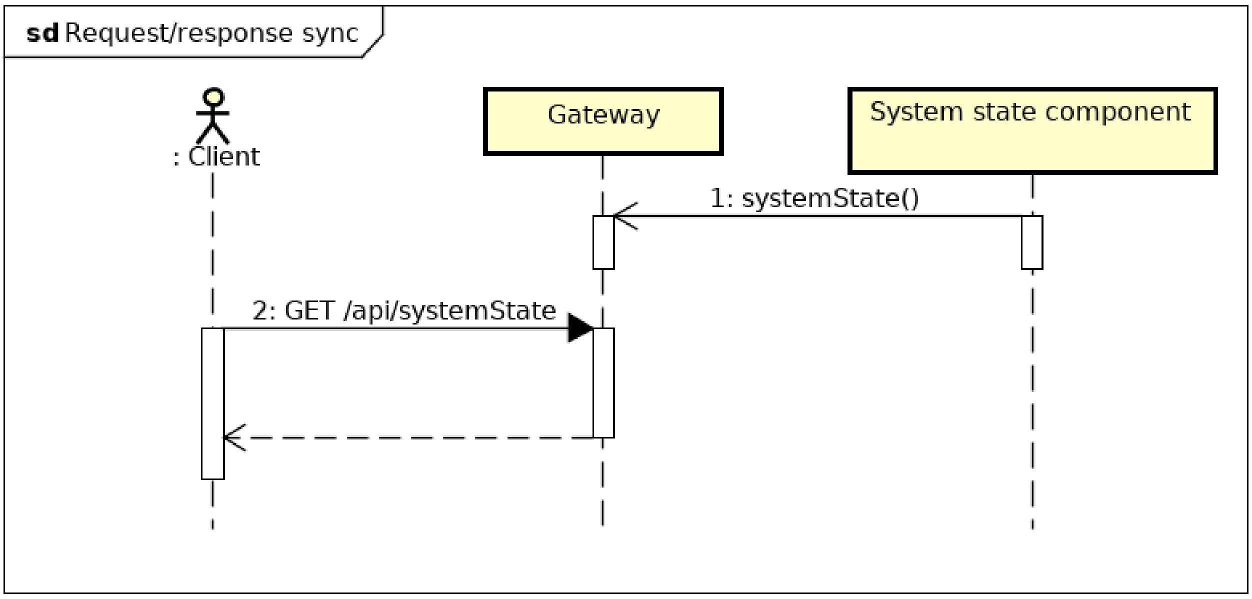


Figure 11: Synchronous request/response sequence diagram

4.3.2 Server Push

Server push is when data on the server have changed and this information should be forwarded to the client. In asynchronous communications, this change can be directly transmitted to the client, as described in Section 3.3.5 where a hardware monitoring component changes the system state which is immediately sent to the client. Figure 12 shows an synchronous example of server push using long polling. The client continually executes GET requests to the API gateway to get the current state of the system. The first request returns the state directly as the gateway has the current state already stored (similar to the synchronized request/response). The second GET request explicitly request version 2 of the state. However, the state has not yet changed, this will trigger a time-out after which the client sends another GET request for version 2. During the second GET request for version 2, the state has changed to version 2 and this is returned to the client. This whole conversation is naturally idempotent as it is only based on GET requests. These GET requests do not change any data on the server.

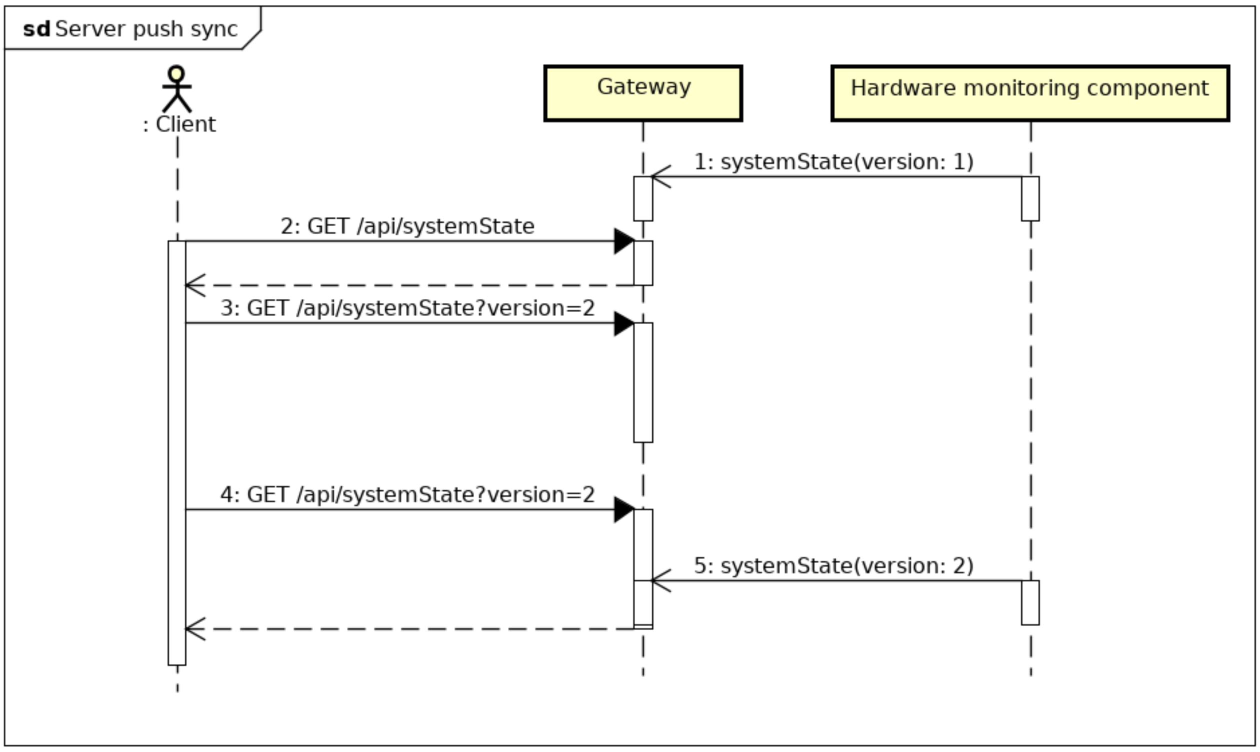

sd Server push sync
: Client
Gateway
Hardware monitoring component
1: systemState(version: 1)
2: GET /api/systemState
3: GET /api/systemState?version=2
4: GET /api/systemState?version=2
5: systemState(version: 2)

Figure 12: Synchronous server push sequence diagram

4.3.3 Request with Fixed Amount of Responses

A client can initiate a request that yields multiple responses. An example in which two responses are returned with one request is shown in Figure 13. This is the example of powering on a subsystem, as described in Section 3.3.3. The first request is a PUT or PATCH, which will request to change the target state of the subsystem to ON. In response to this request the state of the subsystem is changed, of which the current state is still OFF and the target state is ON. Sometime in the future a message can be expected with the current state ON and the target state ON. This state can be requested with a GET request using long polling, which is similar to the server push conversation, which is described in Section 4.3.2. This conversation can be implemented in a fully idempotent way, multiple requests to turn the system on will still result in the system turning on.

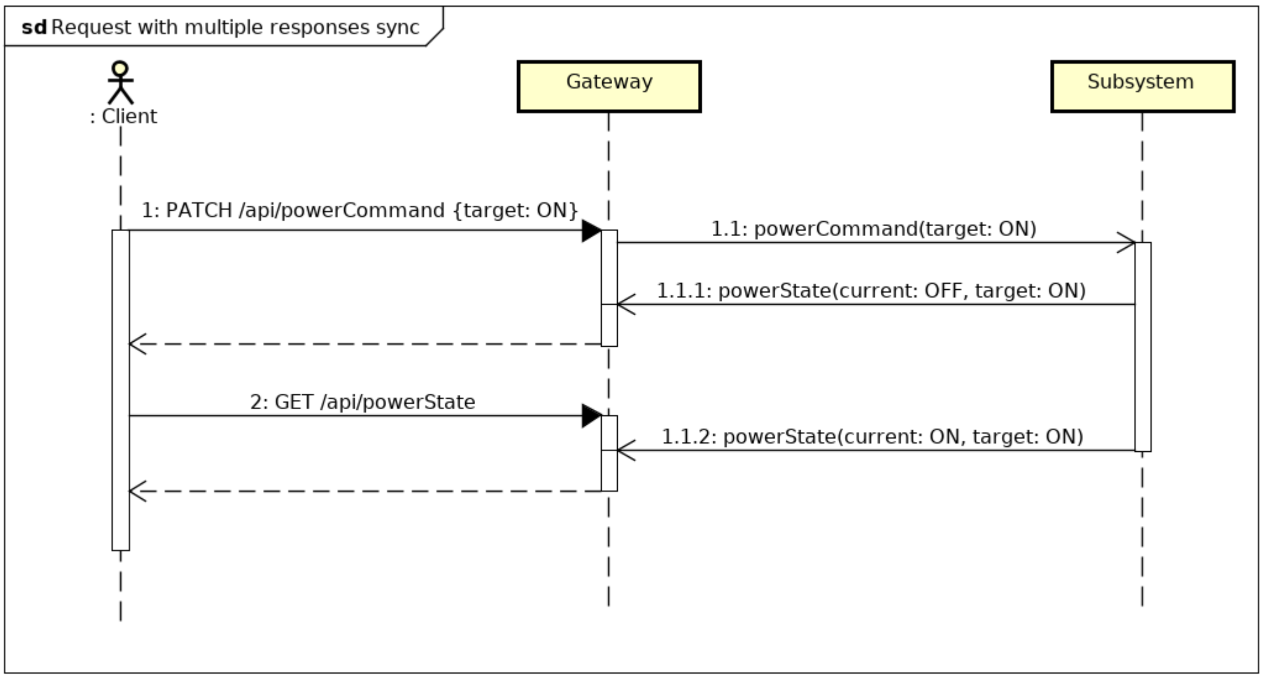

sd Request with multiple responses sync
: Client
Gateway
Subsystem
1: PATCH /api/powerCommand {target: ON}
1.1: powerCommand(target: ON)
1.1.1: powerState(current: OFF, target: ON)
2: GET /api/powerState
1.1.2: powerState(current: ON, target: ON)

Figure 13: Synchronous request with multiple response sequence diagram

4.3.4 Request with Stream of Responses

This is a single request with a stream of responses, as described in Section 3.3.4. The synchronous version of this sequence diagram is shown in Figure 14. This shows an initial PATCH which can change the periodic task to ON. Subsequently taskUpdates are emitted by the task component, which can be requested by the client using GET requests. The client can send a new GET request when the previous one finishes to continuously receive task updates. A version ID is used to request the next task update, to prevent receiving the same task update. The task can be canceled by a PATCH request to the API gateway, which changes the task to off Task component. All the requests are idempotent, sending multiple identical patch request will still ensure that the task is on or off. GET request are always idempotent.

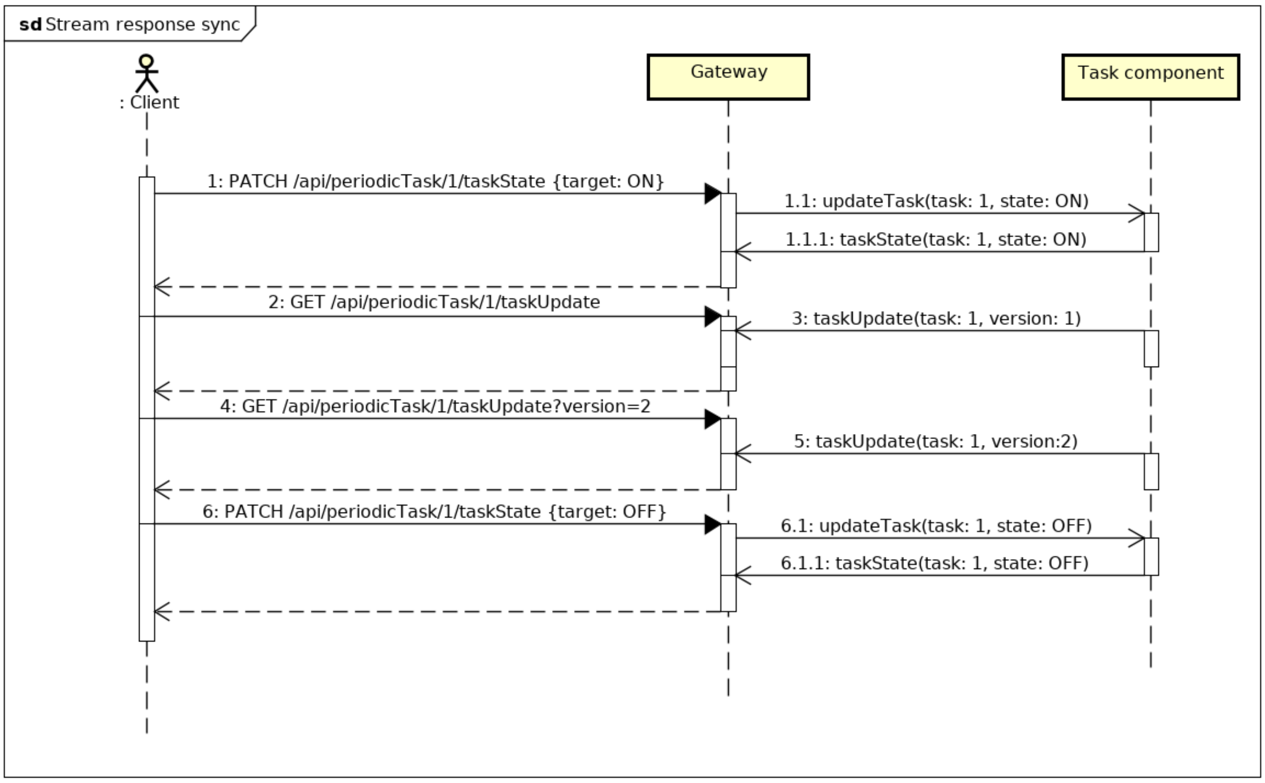

sd Stream response sync
: Client
Gateway
Task component
1: PATCH /api/periodicTask/1/taskState {target: ON}
1.1: updateTask(task: 1, state: ON)
1.1.1: taskState(task: 1, state: ON)
2: GET /api/periodicTask/1/taskUpdate
3: taskUpdate(task: 1, version: 1)
4: GET /api/periodicTask/1/taskUpdate?version=2
5: taskUpdate(task: 1, version:2)
6: PATCH /api/periodicTask/1/taskState {target: OFF}
6.1: updateTask(task: 1, state: OFF)
6.1.1: taskState(task: 1, state: OFF)

Figure 14: Synchronous stream response sequence diagram

4.4 Other Hardware Control Systems

Throughout this study examples of the Thales use case are used. The results of this study can also be applied in a similar fashion to other hardware control systems. To illustrate this, we use the example of a high precision medical operating table. This table can be controlled by sending commands to the table, the table can be turned on and off and the hight of the table can be adjusted. The different conversation types can be translated for controlling the table:

- **Request/response:** Request the current hight of the table. The table will respond with one message indicating the current hight of the table.
- **Request with fixed amount of responses:** Turn on the table, similar to the Thales case a message is emitted when the table is turning on and when the table has turned on.
- **Request with steam of responses:** Command the table to rise, the table will emit messages about the current hight of the table with a fixed interval.
- **Server push:** When something is blocking the table a message can be emitted to notify the client that the table is not functioning properly.

4.5 Concluding Remarks

This chapter identified several conversation types which were translated to synchronous communications using REST. Chapter 5 describes the actual implementation of these conversation types. For push technology, the choice has been made to implement a mechanism based on long polling, but in practice it is very well possible to use another method by following a similar design such as server-sent events. Instead of opening a long poll to the server, the connection of the get request can be kept open and the response which would be sent over the long poll would instead be sent over the still open connection. The same holds for the 'request with multiple responses' example.

5 Implementation

This chapter describes the implementation of the protocol translations and the API gateway in the service-oriented architecture of Thales. The overall target architecture is described in Section 5.1. The implementation of the API translator component with protocol translations is described in Section 5.2. The implementation of the API gateway is described in Section 5.3. Concluding remarks are given in Section 5.4.

5.1 Target architecture

This section describes the target architecture of the proof of concept implementation. The aim of the implementation is to introduce minimal changes to the original asynchronous component-based architecture while implementing a HTTP/REST interface with which the gateway/client can communicate. A diagram of the proof of concept architecture is shown in Figure 15, of which the green elements are the same as in the current existing architecture as shown in Figure 4 in Section 3.1. This means that the existing components are unchanged and still communicate using asynchronous messaging with a DDS. The proprietary API gateway component is replaced with an API translator component, which is indicated in yellow in Figure 15. This API translator component translates incoming HTTP/REST communications to internal messages. A HTTP/REST server is implemented in the API translator component which can be accessed by the API gateway. The blue elements in Figure 15 indicate the elements which communicate using HTTP/REST. For this research the Kong API gateway is used,which is justified Section 5.3.

The following sections describe the implementation process of the different new elements of this architecture. These sections describe the implementation of the API Translator component as well as the installation and configuration of the Kong API gateway.

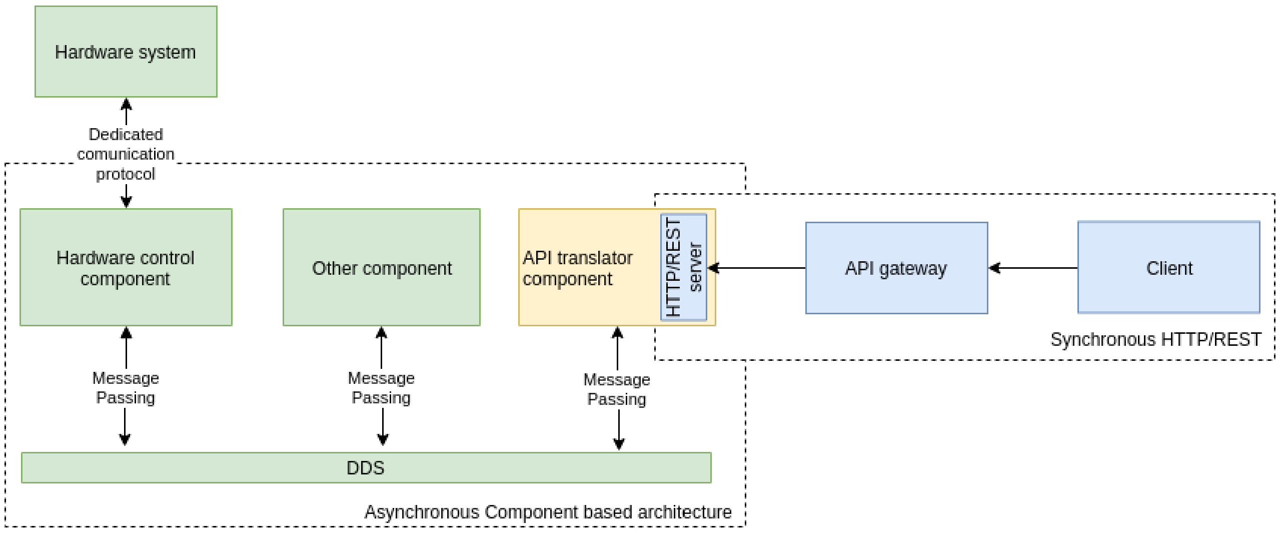


Figure 15: Target architecture

5.2 API translator component

This section describes the implementation of the API translator component which includes a HTTP/REST server, which is indicated with the yellow block with the blue HTTP/REST server block in Figure 15. This component handles the translations of the conversation types as described in Chapter 4. This section starts by describing the OpenAPI descriptions used to generate the server stub code and follows with the implementation of each conversation type in the translator component.

5.2.1 API Descriptions and Code Generation

Client and server side code can be generated by using an API description. For REST APIs this is commonly achieved by using OpenAPI descriptions [33]. Using this format, a REST API can be formally described, including its endpoints, operations and message schemas. An example OpenApi definition is described in Appendix A. An OpenAPI specification enables server and client code generation. For this project, the server stub is generated and is integrated in the API translator component. Customers can use the OpenAPI definition to generate client stub code in their language of choice, which facilitates the client side implementation and makes them less error prone. Due to the readability of the OpenAPI specification, it can also be an asset to the API documentation. Tooling is available to generate documentation based on the OpenAPI specification [40].

There are multiple server libraries written in C++ for which server stubs can be generated. Initially the choice was made to generate code for the Restbed library [14], which is an open source HTTP/REST server library written in C. Restbed was chosen as it supports a high number of features, but unfortunately it proved to be a challenge to integrate with the Thales architecture. This was mainly due an error in the library that occured with the configuration of Thales for which was not a solution at the time. Eventually the switch was made to the Pistache library [32], which proved to be significantly easier to integrate in the Thales architecture. An added benefit of Pistache is that is has no external dependencies and is developed with the aiming at high performance. Performance is not addressed in this chapter, but overall performance tests are described in Chapter 6.

5.2.2 Integrating Generated Server Code

The first step of the implementation was to integrate the generated server code into the Thales component-based architecture of Thales. During start-up of the system, the HTTP server is started by the API translator component. The generated server code provides skeleton functions which were implemented in this study. The implementation of the skeleton transforms incoming requests to messages to other components, or returns values based on (previous) messages. The API component should only provide basic transformations from REST to internal messages. Other features like logging, security and tracing are provided by the API gateway, of which the implementation is described in Section 5.3.

The generated server stub code automatically parses message bodies, request headers and URL parameters. These values are provided as variables to the generated stub functions. Examples of generated functions are shown in Listing 1, which shows generated stub code for a get and a patch request. In this example, the get request is that of a periodic task, which has an id value in the URL (e.g. /api/taskUpdate/**1**) and an optional message parameter version (e.g. /api/taskUpdate/1?**version=3**). These values are passed as parameters to the function, where the version URL parameter is an optional value. The patch request also has the id in the URL, and a parsed object of the request body called taskStatePatch is provided. Both functions also have response object parameters, which are used to construct and send a response back to the client.

Listing 1: Generated code stub examples

```
void DefaultApiImpl::periodictask_id_taskupdate_get(
        const int32_t &id,
        const Pistache::Optional<int32_t> &version,
        Pistache::Http::ResponseWriter &response) {

void DefaultApiImpl::periodictask_id_taskstate_patch(
        const int32_t &id,
        const Task_state_patch &taskStatePatch,
        Pistache::Http::ResponseWriter &response) {
```

5.2.3 Request/Response Implementation

The first implementation of a synchronous conversation is that of blocking request/response messaging. This means the client sends a request to the HTTP server and blocks until a response is received from the HTTP server. In this case the HTTP server always stores the latest version of the systemState, so that the state be directly returned. This is schematically shown in Figure 16. The state component emits the systemState to the API translator component, which stores this state in memory. This memory object is accessible by the HTTP/REST server, which can send the state as a response to the GET request.

An alternative to this implementation is that the API translator component does not store the systemState in memory. In this case, the systemState should be requested from the state component when a GET request is received from the client. After a request from the client is received, a message is emitted to the state component to request the systemState. After a while the systemState is received by the API translator component and it is then sent to the client. The client is blocked while this communication takes place.

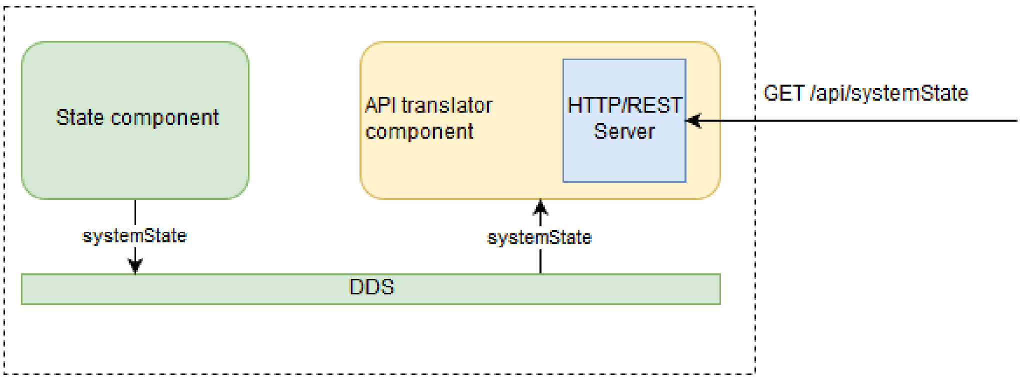


Figure 16: Request response implementation

5.2.4 Server push Implementation

The server push implementation architecture is identical to the architecture as displayed in Figure 16. With server push the client request a certain version of the systemState by using the URL parameter *?version=x*. Somewhere in the future, when the version of the state changes, a response is sent to the client. In this case, the client can send the request */api/systemState?version=4*, after which the client starts waiting for a response (long polling). If the version of the systemState is already higher than than 4, the response is returned directly. If this is not the case, the response is blocked until the systemState is at the desired version. It is possible that the systemState does not change for a while and a timeout occurs on the connection. In this case the client will start a new request to receive version 4.

The server has to keep the connection open and wait for the systemState to change to the desired value. The implemented approach uses a condition variable with a lock to block the connection until the value has changed. The code for this approach is shown in Listing 2. This shows the request thread waiting for the modification_id to be larger or equal than the requested version 'v'. The main thread notifies the request thread that the systemState has changed.

Listing 2: Long poll lock example

```
//Request thread
if(api.platform_taskUpdate_condition_variable.wait_for(
  lk,
  std::chrono::duration<int>(60),
  [this, v]{return  api.platform_taskUpdate.modification_id >= v;})
){
   response.send(Pistache::Http::Code::Ok, getTaskUpdateString());
} else {
   // handle timeout
}

//main thread
systemState_condition_variable.notify_all();
```

In this design, the value is returned if the version of the value is higher than the requested value. This means that when version 3 is requested, a newer version (e.g., version 5) can be returned. This is a design choice, as in this case the newest version is always the most relevant. In a system where older versions are still relevant it is of course necessary to store and retrieve these past versions. This can also be very important if all versions need to be received by a client to allow for correct processing of all the information.

5.2.5 Request with Multiple Responses Implementation

The schema of this architecture for this conversation type can be found in Figure 17. This shows two incoming requests, namely a get request the get the state of the subsystem and a patch request with a powerCommand to turn the system on or off. The response of this patch request is a HTTP status code indicating that the patch was received successfully. A code snippet of the PATCH request is shown in Listing 3, which shows that an incoming PATCH request is simply translated to a message which is sent by the API component. After the patch is received successfully, the state changes multiple times. The changed state can be requested using the GET request, which implements the same long-polling mechanism as described in Section 5.2.4 to wait for changes.

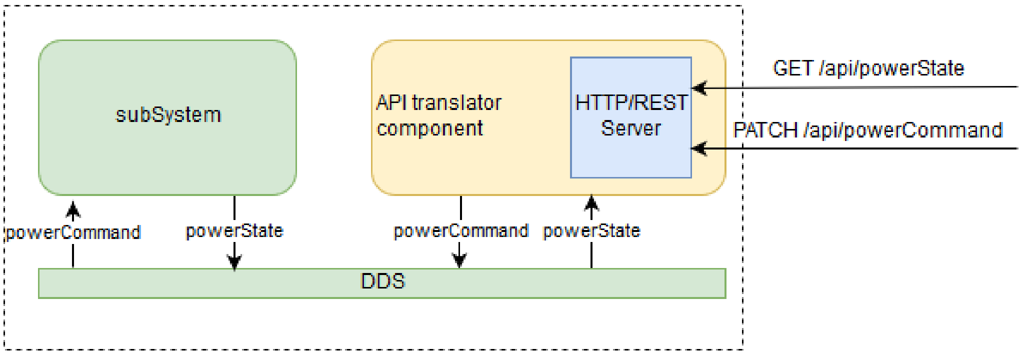


Figure 17: Request with multiple responses implementation

Listing 3: powerCommand patch

```
void DefaultApiImpl::powercommand_patch(
                       const State_patch &statePatch,
                       Pistache::Http::ResponseWriter &response) {
      api.sendPowerCommand(statePatch.getTargetState());
      response.send(Pistache::Http::Code::Ok);
}
```

5.2.6 Request with Stream Responses Implementation

The request with a stream of responses implementation is based upon the periodic task example, and the architecture for this implementation is shown in Figure 18. This conversation starts with a PATCH request to a task with a specific id. There are multiple predefined tasks, which can all individually be controlled by providing their id in the URL. There are two GET requests implemented, where one GET request is used to get the current state of the task. This indicates if the task is currently running or not. The other GET request is to get a taskUpdate for a specific task. In our example only one taskUpdate can be requested at a time. It is also possible to design a system in which multiple task updates can be returned using a single request. Once a task is enabled with a PATCH request the task will emits taskUpdates with a fixed interval.

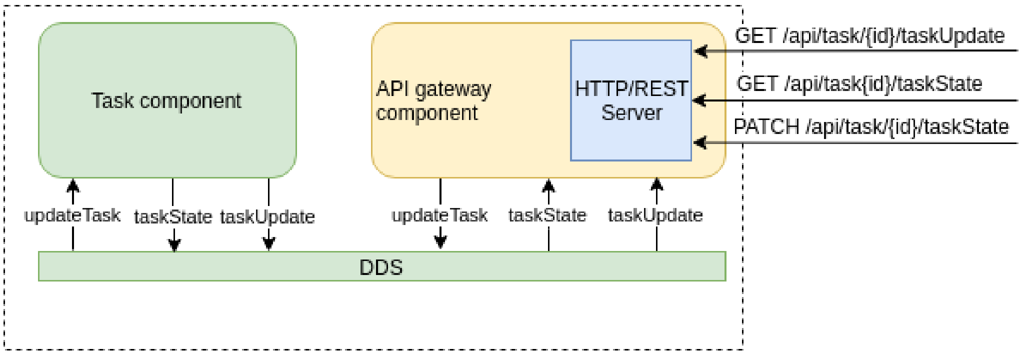


Figure 18: Request with stream responses implementation

Once a PATCH request is received, an updateTask message is sent to the task component to update the task. The task component then emits a taskState message to indicate that the task is updated. When the task is enabled, the taskUpdate message is emitted with a regular interval. For this proof of concept, this is simulated by a thread which emits a message every 4 seconds, of which the code can be seen in Listing 4. This code sets a random number in each message to simulate a changing value. A simple client can be created to test this implementation, which is a bash script with a loop that increases the version id, as can be seen in Listing 5. Sample output of this client is shown in Listing 6.

Listing 4: Task update thread

```
void Task::task_executor() {
    while(task_state.current_state == platform_ON) {
        task_state_mutex.lock();
        task_update.modification_id += 1 % INT_MAX;
        task_update.task_variable = rand() % 100;
        task_state_mutex.unlock();

        this->platform_task_update_sendTaskUpdate(task_update);
        std::this_thread::sleep_for(std::chrono::seconds(4));
    }
}
```

Listing 5: Task update client

```
for i in {0..10000}
do
  curl http://localhost:8000/unauth/task/1/taskupdate?version=$i
done
```

Listing 6: Task update client output

```
{"task_id": 1, "task_variable": 96, "modification_id": 2}
{"task_id": 1, "task_variable": 6, "modification_id": 3}
{"task_id": 1, "task_variable": 65, "modification_id": 4}
{"task_id": 1, "task_variable": 66, "modification_id": 5}
```

5.3 API Gateway

The features of API gateways have been analysed in Section 2.2. This showed that Tyk and Kong are the most feature rich, where Tyk only lacks traceability functionality and Kong only lacks request composition. In our preliminary research, the Tyk API gateway was tested, which showed that the Tyk gateway relies on many different scripting techniques and languages [36]. Examples are request composition, which is achieved with JavaScript, and request translations, which are done with Go templates. This requires developers and maintainers of the API gateway to be acquainted with these different techniques. For this research, an implementation with the Kong API gateway has been tested, as it also offers a large selection of features, and this allows for a comparison between Tyk and Kong.

5.3.1 Installation

The API gateway, management interface and database are running in Docker containers. For the Kong API gateway the open source management interface called Konga [34] is used. It is also possible to configure the Kong API gateway directly using the provided HTTP/REST API. The overall architecture is shown in Figure 19. This shows a simplified version of the component-based architecture on the right side with the new API translator component. Components in the component-based architecture can also be run as Docker containers, which allows for the whole system to run in Docker. The Kong API gateway communicates with a Postgres database to store its configuration and is accessible by two APIs on port 8000 and 8001. Kong can be configured using the API accessible on port 8001 and the client-facing API is accessible at port 8000. Konga communicates with the API on port 8001 to configure Kong. Konga provides a website on port 1337 which can be accessed with a browser. This website can be used to configure Kong.

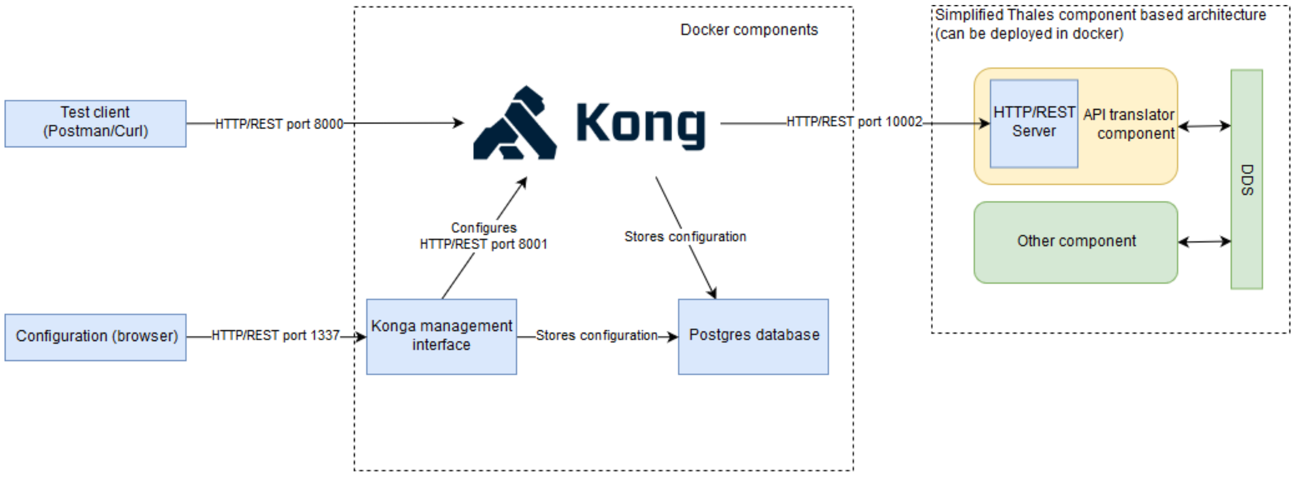
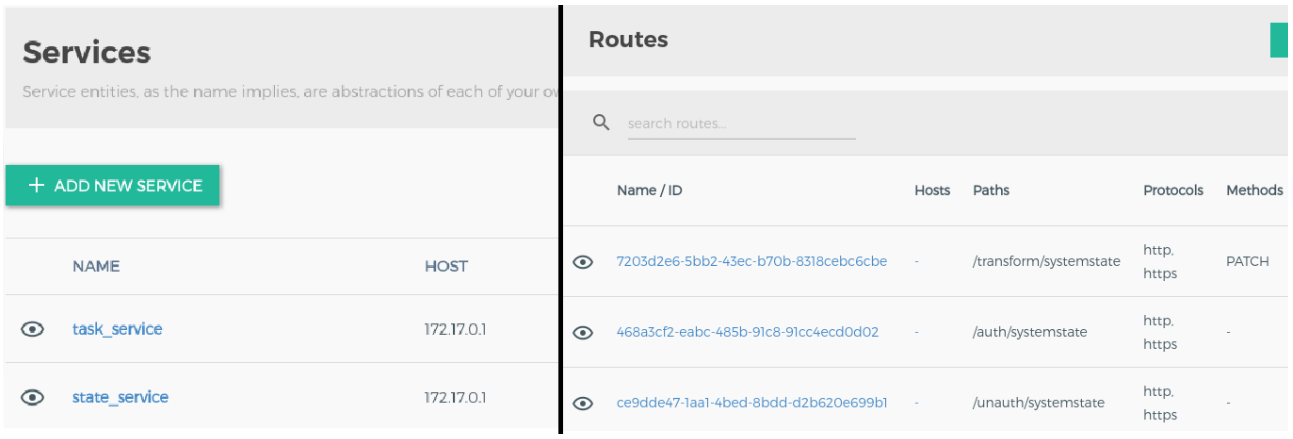


Figure 19: Overall architecture

5.3.2 Configuration

The gateway can be configured based on services and routes. A service describes a certain endpoint at which multiple routes can be defined. Figure 20 shows two configured services on the left side, one is the task service and the other is the state service. The routes for the state service are shown on the right side. In this example, there are transformation, auth and unauth routes available. Plug-ins can be enabled on different levels. It is possible to enable system-wide plug-ins, plug-ins for a specific service or plug-ins on a specific route of a service.


Services
Service entities, as the name implies, are abstractions of each of your ow
+ ADD NEW SERVICE
Routes
search routes...

NAME	HOST
task_service	172.17.0.1
state_service	172.17.0.1

Name / ID	Hosts	Paths	Protocols	Methods
7203d2e6-5bb2-43ec-b70b-8318cebc6cbe	-	/transform/systemstate	http, https	PATCH
468a3cf2-eabc-485b-91c8-91cc4ecd0d02	-	/auth/systemstate	http, https	-
ce9dde47-1aa1-4bed-8bdd-d2b620e699b1	-	/unauth/systemstate	http, https	-

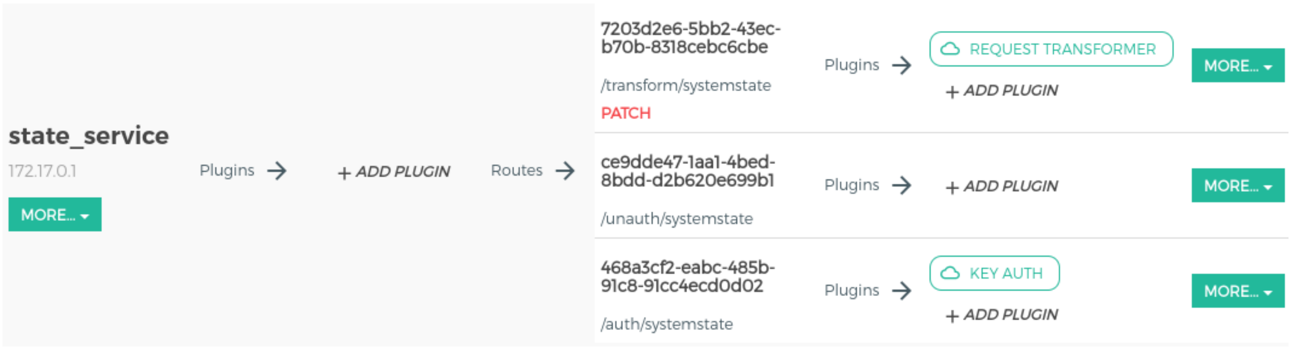
Figure 20: Routes and services configuration

5.3.3 Security

Both Tyk and Kong provide a similar approach to security. In both, clients can be configured, and this configuration can be used to authenticate and authorize requests. They offer support for different kinds of authentication methods, such as: basic authentication (username/password), key authentication and OAuth. For this example, key authentication was implemented to authenticate and authorize request. Key authentication requires a valid API key to be provided within requests. An authentication key should be provided in an URL parameter, such as for example: example.comapistate?key=some_key. The API gateway then validates this API key on every request that passes through the gateway. In both Kong and Tyk, it is also possible to configure security certificates to encrypt traffic using SSL certificates. Both API gateways provide security features which are fully handled by the API gateway and fully transparent to the server.

A route was configured for authentication, which is shown in Figure 20. One client was created which has one authentication key configured. The gateway responds with an error message when no key is provided or when a wrong key is provided. In case of a correct key, the client is authenticated and the request is forwarded. The plug-in allows for the authentication credentials to be forwarded to upstream services, which can use the credentials if some use case requires this. For our use case it is sufficient for the gateway to only handle the authentication.

Each client has a page with which the it can be configured in detail. It is possible to add a client to specific groups, add credentials and enable plug-ins specific for this client. It is also possible to get a detailed overview of all the accessible routes for the client. Figure 21 shows part of this page, which shows the detailed information about the access this client has to the state service.



state_service
172.17.0.1
Plugins → + ADD PLUGIN Routes →

7203d2e6-5bb2-43ec-b70b-8318cebc6cbe
/transform/systemstate
PATCH
Plugins → REQUEST TRANSFORMER + ADD PLUGIN

ce9dde47-1aa1-4bed-8bdd-d2b620e699b1
/unauth/systemstate
Plugins → + ADD PLUGIN

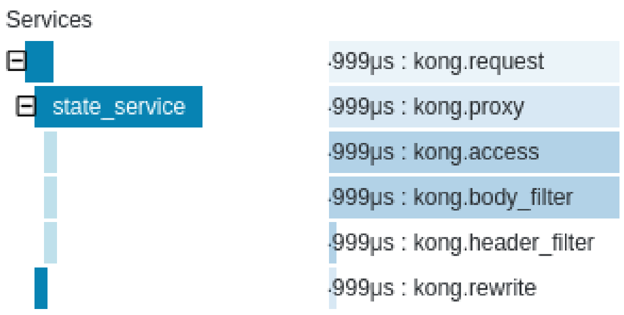
468a3cf2-eabc-485b-91c8-91cc4ecd0d02
/auth/systemstate
Plugins → KEY AUTH + ADD PLUGIN

Figure 21: Accessible routes for a client

5.3.4 Monitoring/Logging/Tracing

Both Kong and Tyk have in depth monitoring and logging features. Both provide plug-ins to log data to external services. Examples of available logging services (in Kong or Tyk) are file, TCP, HTTP or a service like Loggly. Monitoring plug-ins are also available, like Galileo, Datalog and Prometheus. Both management interfaces of Kong and Tyk provide a dashboard to get a quick indication of the state of the API gateway.

Most API gateways provide limited support for request tracing. Tyk does not support tracing features out of the box, but Kong does. To enable tracing in Kong, a Zipkin tracing service needs to be running. A Docker container is available with the zipkin tracing service, which works out of the box. The URL of the zipkin tracing service can be configured in the Kong tracing plug-in, which is everything needed to configure tracing. Figure 22 shows an example trace of a GET request to the state service. This trace shows the various stages the request passes through and the total time of the request. A sample ratio can be set, which only samples a percentage of the requests. This is useful to reduce delays when a lot of requests pass the gateway.



```
999μs : kong.request
999μs : kong.proxy
999μs : kong.access
999μs : kong.body_filter
999μs : kong.header_filter
999μs : kong.rewrite
```

Figure 22: Zipkin trace example

5.3.5 Request/Response Transformation

Tyk and Kong provide a different approach to message transformations. Tyk allows message transformations based on GO templates. The message structures are described in templates and transformation rules are defined on these templates. Go templates can be used in a variety of ways, for changing/moving/editing fields to if/else statements based on data. Kong has two versions of the transformation plug-in, a free version and an enterprise version. The free version can only perform remove, rename, replace, add and append operations on the body, headers or query string.

Content-based transformations in the Kong gateway are only available with the enterprise plug-in. This reduces the applicability of the transformations in the free version significantly, since it is for example not possible to make changes based on request content. An example use case could be that internally the responses 'OK' and 'NOK' are used, but the customer requests that these responses should be 'successful' and 'unsuccessful', respectively. In Tyk this can be configured using the GO templates, in Kong this is not possible with the free version of the plug-in. It is possible to add the rule 'if a field called target_state exists, set the value to 'successful'', but not 'if a field called target_state exists and the value is 'OK', change it to 'Successful'.

5.3.6 Request Composition

Tyk offers request composition which can be programmed in JavaScript files while Kong does not provide any request composition functionality. Ideally, request composition should be configurable, but in practice this is not available in open-source tooling. Both programming in Tyk or programming in C++ provide the disadvantage of programming over configuration which can introduce programming errors.

5.4 Concluding remarks

This chapter describes the full implementation of the conversation types and the API gateway in the reactive architecture of Thales. This shows that the defined protocol translations of Chapter 4 can be implemented with relative ease and are functional with off-the-shelf available tooling. With knowledge about the code language, implementation of the conversation types can be achieved in a couple of days work. The OpenAPI specification with a code generation tool is a useful asset, which increases the ease of implementation significantly. API Gateways like Kong and Tyk can be set up using Docker and are configurable with relative ease. Both Kong and Tyk are powerful API gateways, but Tyk depends more on scripting and templating tooling, while Kong is more based on configuration. For in depth request transformations, the enterprise version of the Kong API gateway is needed, as the free version of the requests transformation plug-in provides very limited functionality.

6 Results

In this chapter, the synchronous implementation is evaluated on several factors. The first factor is suitability, i.e. whether the new implementation addresses (most of) the challenges described in Section 3.2. The second factor is performance, in which benchmark tests have been executed to determine the performance of the new implementation. Lastly, design choices are discussed with possible alternative approaches to the problem. Section 6.1 describes the solutions for the different challenges, Section 6.2 analyses the performance of the system and Section 6.3 discusses the alternatives to the chosen solution.

6.1 Solutions to Challenges

Section 3.2 introduced the list of challenges with the current proprietary API gateway implementation. These challenges have been tackled with the new API gateway implementation. Below we discuss to what extend the implementation solves the challenges.

- **Asynchronous client-facing and proprietary API:** This study has introduced a synchronous API as an alternative to the currently asynchronous API. Using HTTP/REST as the communication protocol allows for widely available open-source tooling to be used to design and support the API. Examples of tooling that was used during this research are: the API gateway, OpenApi specifications, Server and client code generation, HTTP server framework in C++ and the benchmarking tooling. Client code generation allows for customers to easily generate HTTP client code in almost any language or framework of choice, which can reduce a large amount of implementation time.

  The main challenge of the design for the synchronous API is to handle data pushed from the server to the client using synchronous methods. This was needed in multiple conversation types, specifically in the 'server push' conversation type, but also in the 'request with multiple responses' conversation type. There are several synchronous and asynchronous methods for pushing data from a server to the client. For this research the 'long polling' method was chosen but other technologies like WebSockets or SSE are available alternatives for pushing data from the server to the client, but support for these technologies is limited in API gateways.

- **User adaptations:** User adaptations can now be configured instead of programmed by using the Kong Request and Response transformer plug-ins. Depending on the API gateway, there is a limit to the number of changes that can be made to the requests. In case of the Kong API gateway, changes can be made to the body, header and query string using the Remove, Rename, Replace, Add and Append methods. The Kong gateway only provides transformations based on the message payload for enterprise customers. The Tyk API gateway provides more in depth transformations by applying a template-based method using GO templates. The Kong API gateway has the advantage that transformations can be configured using the GUI whereas in the Tyk API gateway knowledge about the GO template language is required.

- **Request composition:** Request composition turned out to be poorly supported by API gateways. Kong does not provide request composition while Tyk does provide this functionality. When using request composition in the Tyk API gateway, it is required to implement the request composition using JavaScript. This only moves programming the request composition in a component to programming request composition in the API gateway. Ideally there should be some kind of configurable option for request composition, but this seems to be unavailable in off-the-shelf systems. We think that this is because that is is very hard to provide a generic method for configuring all kinds of request compositions for a variety of use cases.

- **Security features:** An API gateway is perfectly suited to implement security features between the client and the gateway. Examples are shown in this study, where key authentication is implemented, but multiple authentication options are available ranging from basic username/password authentication to external OAuth authentication. It is also possible to configure SSL security certificates to encrypt communication to the API gateway.

  These security options can be enabled without any changes to upstream services, which in our example case was the API translator component. It is possible to forward the authentication credentials to upstream services if some specific use case requires this, but this is not necessary.

  Consumers (users) can be defined in the API gateway and access control rules can be added for these consumers. This allows fine grained authorization for specific routes per user.

- **State changes:** Pushing data from the server to the client using synchronous methods introduces complexity on the client side. For example, in this study a mechanism was implemented where the client continually requests changes from the server using long polling. The complexity here is that a client needs to implement a mechanism that issues a request and waits for a response, and if there is a timeout a new request needs to be opened. The advantage is that a long poll is similar to a normal HTTP request, which means that the additional features of the API gateway can be used on these requests.

  Long polling can be a drain on server resources, as connections need to be held open in order to handle long polling requests. This disadvantage is minimized because hardware control systems have a limited amount of clients. This means that also a limited amount of simultaneous long polls can be open.

- **Timing constraints:** The performance of the system is analysed in Section 6.2.

- **Tracing, logging and monitoring:** API gateways have extensive support for logging and monitoring features. This is because all traffic passes the API gateway, which allows for a single entry point to monitor all the traffic. API gateways can send logs to local file output or external services using plug-ins, which allows for more in depth (automated) analysis of logs.

  API gateways provide limited support for request tracing. Tyk does not provide request tracing plug-ins while Kong does. Request tracing traces requests to the system and stores these traces in an external service. In this study the Zipkin tracing service was used, of which only the URL needs to be configured in the Kong tracing plug-in. Traces of requests through the different parts of the Kong API gateway were automatically stored in the Zipkin service.

- **Maintainability:** Maintainability and trends in IT are hard to predict, especially over a span of 10+ years. HTTP/REST was chosen for this study as it is supported by all major API gateways, and tooling is currently widely available. All the used tooling is opensource, which allows a company to maintain the solution in the future if projects are no longer frequently maintained.

Looking at the future, HTTP/2 is a promising development which has certain advantages for companies. HTTP/2 allows for the server to push data to the client before the client actually requests these data. This can be used in several conversation types, especially the 'request with multiple responses' conversation type. Support for HTTP/2 is currently growing. Some API gateways and some C++ REST libraries already support HTTP/2.

6.2 Performance

Performance tests were executed on the test implementation to determine whether the performance of the solution is within the bounds set at the start of this research. These bounds were that a total delay of a request, from the client, through the API gateway, to the server, and back through the API gateway should be shorter than 10 ms. This includes all functionalities like message transformation, authentication, logging, monitoring etc.

A HTTP/REST benchmarking tool called 'wrk' [43] was used for the benchmark test. The benchmark ran on the same system as where the test set-up ran. This system is a Fedora Unix system with kernel version 5.1.6 on a Intel Xeon E-2176M CPU with 32 GB of RAM. There was no special sandbox environment, so other system processes could interfere with the benchmark. The test set-up was a proof of concept and did not have any optimization. The result of the benchmark should show an upper bound of the performance of the system on this hardware, further optimizations can improve these results. The benchmarks ran multiple times and the results were validated with a second benchmarking tool. The benchmark ran 30 seconds for each case on two treads, which kept two HTTP connections open. Two treads and two connections are chosen because hardware control systems generally only have a couple of clients. Figure 23 shows the elements included in the benchmark test. The tests measure the full path from the WRK benchmarking tool, through Kong, to the HTTP/REST server, the protocol translations and back.

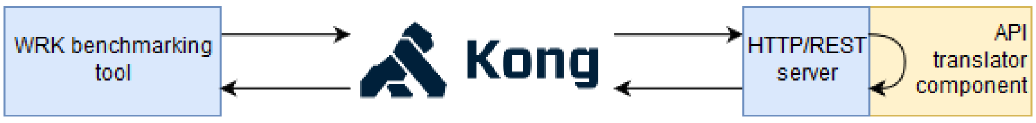


Figure 23: Benchmark path

The visualized results are shown in Figure 24, the raw textual results can be found in Appendix C. Benchmark tests are executed on 6 different requests, of which 4 are GET requests and two are patch requests. The GET request is that of the system state, of which 4 different configurations of the API gateway were tested. The first configuration is without authentication, the second is with authentication, the third and fourth are with request tracing enabled on different sample sizes. One with request tracing enabled with a sample size of 0.01, which traces one in every 100 requests and one with a sample size of 1, which traces every request. The results are as expected, the more complex the enabled plug-ins are, the longer the round trip times become. There is a clear increase with authenticated requests over unauthenticated requests and also a clear increase in request times with a larger tracing sample size. Two benchmarks of PATCH requests are also included, which is a request to change the system state, that includes sending a message to the DDS. One is a normal patch request while the other is based on a request transformation that adds a field to the patch request. Also here a slight increase can be noticed when request transformation is enabled. Overall the response times are well below one millisecond, which is well within the bounds of 10 milliseconds as was the originally stated requirement. This does not include network delays, as the benchmarking tests were executed on the same system in which the other services were running.

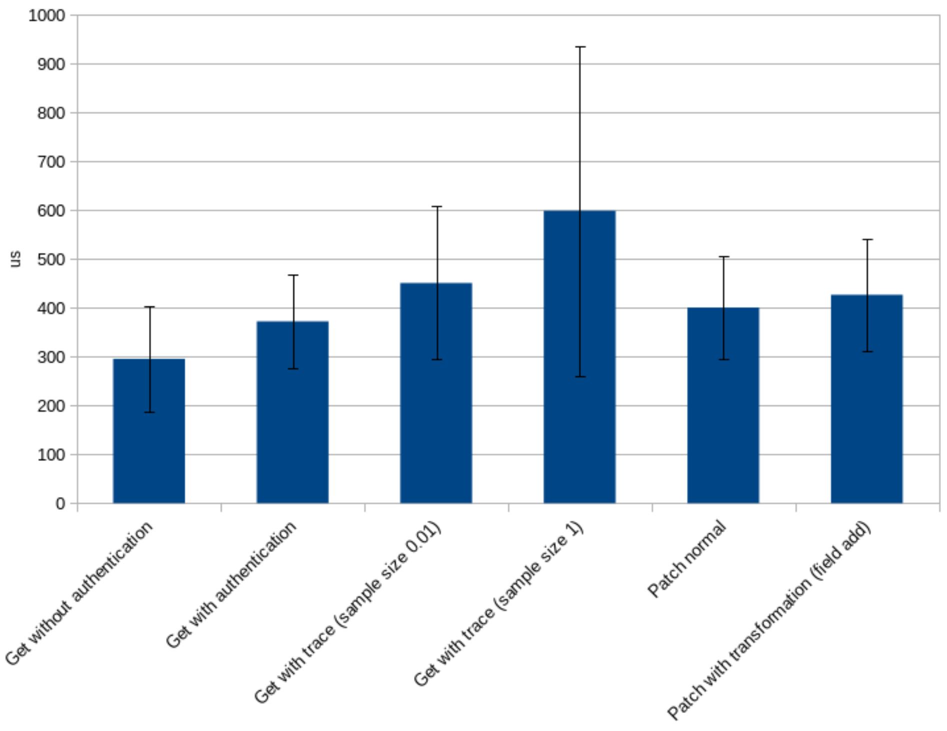


Figure 24: Performance measurements

6.3 Discussion

A variety of decisions were made during the implementation, examples of these decisions are the protocol, API gateway and push technology. There are many API gateways, but we picked two for the analysis, Tyk and Kong, which are both viable open-source gateways for the challenges tackled in this study. Finding an API gateway that is a perfect fit for this study is a challenge as there is no API gateway that both supports request composition and traceability features. Custom plug-ins can be created for an API gateway, both Kong and Tyk support custom plug-ins, which make it possible to adapt the API gateways to be a more perfect fit for the specific use cases of companies. Plug-ins for API gateways can mostly be created as middle-ware, meaning that they can inject or use request information before it is passed to the next level. This means it is generally not possible to use plug-ins to create request composition as this requires more than only injecting or extending existing requests.

We chose HTTP/REST as the communication protocol as this was supported by all API gateways. Support for other protocols had too many limitations to be a feasible alternative, but this might change in the future. An example was already shown, where HTTP/2 can be a protocol that is very suitable in this context in the future. This study implemented a fully synchronous protocol, as an alternative to the currently fully asynchronous protocol.

A hybrid solution between a synchronous and asynchronous protocol can also be a feasible approach. In this way, part of the communication is synchronized through the API gateway, but server push and stream responses can be sent over an asynchronous protocol. A commonly used combination between synchronous and asynchronous protocols is HTTP/REST with the WebSocket protocol. Depending on the API gateway there is limited or pass-through support for WebSockets.

Long polling was used as the push technology during this study. Long polling has the disadvantage that continuous connections to the server have to be kept open in order to push data to the clients. The advantage of hardware control systems is that only a couple of clients are connected with the system at any point in time. A challenge with open connections on the server side is that server threads are blocked while the connection is open. The Pistache library that is used is multi-threaded, and starts a thread for each request. A more efficient design could be to manage multiple connections with one thread, which Pistache is not designed for. This can be a reason to use some other implementation for the HTTP/REST server like Restbed. Restbed mentions a single threaded variant of their HTTP/REST server [14] but the link to the resource containing information for the single threaded server is broken currently.

7 Final remarks

This section gives our general conclusions and describes opportunities for future work. Section 7.1 describes the general conclusions of this report. Section 7.2 describes possible future work.

7.1 Conclusions

This study describes an approach to introduce synchronous communications and an API gateway to a reactive architecture for hardware control systems. The main objective of this research was to *design and implement an API gateway into an reactive architecture for hardware control systems whilst the basic characteristics of the reactive architecture are preserved.* This objective was divided in multiple sub-objectives, which are discussed in the following sections, followed by general conclusion.

7.1.1 Protocol Translations

The first sub-objective was to *research methods for protocol translations from a protocol designed for a reactive architecture to a protocol suitable for use with an API gateway.* The first step to solve this objective was to analyze the currently available open-source API gateways on their supported protocols. An analysis of 15 open-source API gateways showed that API gateways are mainly designed to work with HTTP/REST APIs, and support for asynchronous protocols was limited or recently implemented in experimental releases. The available tooling for HTTP/REST like OpenAPI specifications, server and client code generation and elaborate protocol standards were also the reasons for us to develop an REST API for the reactive architecture.

By analysing the component-based architecture and communication sequence diagrams, several communication types between the client and the reactive architecture were identified, like: 'request/response', 'request with multiple responses', 'request with stream responses' and 'server push'. Synchronous variants of the asynchronous communication types are defined based on HTTP/REST. The main challenge here lied in replacing asynchronous pushing of data from the server to the client with a synchronous method. Multiple push technologies were analyzed, such as: long polling, Server Sent Events and WebSockets. As a synchronous solution, 'long polling' was chosen as the amount of clients is limited so long polling will not have a significant overhead. Long polling behaves exactly as any other synchronous REST request, which allows using the same cross-cutting features for every request.

Synchronization of requests can take place at different points. 1: the sender is blocked until the middle-ware notifies that it will take over transmission of the request. 2: the sender synchronizes until a confirmation from the recipient is received. 3: the sender synchronizes until the message has been processed by the recipient and a response is received. The original asynchronous implementation synchronizes at point 1, when the middle-ware receives the message and takes over transmission. Our new synchronous implementation synchronizes upon receipt of the message at the API translator component with a PATCH request and synchronizes until a response is received with a GET request. This means that in some cases the API translator component has a direct response available, which means that the client is blocked until the request is processed (3). In other cases the API translator component forwards the message to the DDS in which case the client is synchronized upon approval of the message by the server (2). Overall the synchronization point is moved from the client to the server.

Other protocol designs are possible, such as for example a method based on SSE, which allows the server to send multiple responses on one HTTP request. Support for SSE is limited and mostly not listed in API gateways. HTTP/2 is another solution introduced in 2015 of which support is growing. In the future, this will possibly provide a viable solution to push data from the server to the client. Using SSE, the server can push multiple responses over the open TCP connection. Currently the support of HTTP/2 in tooling is limited but according to many product roadmaps likely implemented in the near future.

7.1.2 Cross-cutting Features

The second sub-objective was to *Introduce cross-cutting features using the API gateway without significant code changes to the existing services and minimal overhead.* Besides the analysis of supported protocols, the feature support of API gateways was also analyzed. Some features are supported by all the API gateways in the selection, which are monitoring/analytics, authentication/authorization and rate limiting. Some features are barely supported by any API gateway, such as traceability and request composition. Two gateways are the most feature-rich, namely Tyk And Kong. Tyk only lacks support for traceability and Kong only lacks support for request composition.

Our preliminary research has analyzed the Tyk gateway as this has support for a wide variety of features. For this research the Kong API gateway was used in order to make a comparison between the two most feature rich API gateways. Almost all features of the API gateways can be enabled without any changes needed to the upstream services. An exception to this is the use of traceability features, as tracing can be implemented at different levels. Tracing only in the API gateway can be implemented without any changes to downstream services, while more in depth tracing of requests through services requires changing downstream services.

In general, open source API gateway supports all features required in this research. Most API gateways can be extended by creating plug-ins. These plug-in systems are mainly based on middle-ware, in which a plug-in is part of a chain of plug-ins that use or alter the request or response. For example, with tracing functionality, the plug-in can inject tracing information in the request. This also means that it is hard to develop a plug-in that can perform request composition, as this means that the middle-ware needs to execute a separate request which, is not the goal of the middle-ware. Support for request composition is also very limited, most API gateways do not support request composition. Tyk does support request composition, but the request composition logic needs to be programmed using JavaScript. It is preferable to be able to configure request composition instead of programming the request composition logic, because programming the request composition can introduce programming errors.

7.1.3 Implementation

The third sub-objective was to *Validate the findings by implementing and testing an API gateway prototype with protocol translations and cross-cutting features at the sensor management system of Thales.* Implementation of the identified conversation types were created as a proof of concept in the component-

based architecture of Thales. An API translator component in the Thales architecture was created that accepts HTTP/REST requests and translates these requests to internal messages.

The API translator component includes a HTTP/REST server, which is generated from an OpenAPI description. The generated code contains stub functions for each REST call as described in the OpenAPI description. These functions contain the parsed requests, including the body, headers and request parameters. These functions have been implemented, and transform the request into internal messages or respond with values that are already present. The implementation complexity lies for the most part in the long polling mechanism. This mechanism needs to manage open connections with the clients and send messages to these clients as these arrive in the component. Customers have to implement the client-side mechanism for long polling. Overall, generating client and server side code can help reduce programmer errors and also automatically parses the messages that are exchanged between the client and the server. The OpenAPI description can also be used as an asset to generate documentation.

Our implementation validated that cross-cutting features can be implemented in the API Gateway with minimal code impact on the server-side. All features chosen for this study could be implemented without any code changes to the services in the reactive architecture. Benchmarking tests were executed to determine the round trip delays of a request to the reactive architecture that passes the API gateway and the HTTP/REST server. Several configurations of the API gateway were tested, which contain authentication, traceability and request transformation. These tests showed that the API gateway and REST library introduce a delay that is well within the bounds of the requirements that were set at the beginning of the research.

7.1.4 General conclusions

Overall this study analysed and introduced protocol translations and an API gateway for an asynchronous component-based reactive architecture. The original aim of the study was to preserve the basic characteristics of the reactive architecture. These characteristics were responsiveness, resilience, elasticity and message-driven. The responsiveness of the system has been affected in a minimal way, as the delay introduced by the API gateway and the HTTP server are well within the bounds set during this study. Resilience can be achieved by running multiple instances of the API gateway and the resilience of the original component-based architecture has not been affected. Elasticity in the system has minimally changes as most of the processing takes place between components and not between the client and the reactive system. The 'message-driven' element of the architecture has been affected on the front-end. Internally the system is still based on message passing, but on the communication between the client and the server this is no longer the case.

The designed solution has several advantages for the reactive architecture, mainly because off-the-shelf software can now be used to facilitate communication between the client and the server. Examples of the off-the-shelf solutions are the OpenAPI specification, code generation, API gateway and benchmarking tools. The API gateway provides many out-of-the box features but also has a couple of disadvantages. The main disadvantage is that support for protocols is limited in the open-source API gateways as these are mainly focused on HTTP/REST communications. The second disadvantage is that an API gateway is probably never a perfect fit and may require plug-ins to be programmed to tailor the API gateway to specific needs.

Overall the design is a matter of moving complexity. In the current situation, the complexity lies in the proprietary API gateway, which requires a lot of maintenance work. Asynchronous communication in the current architecture fit the internal communication directly, but is based on a proprietary protocol which does not facilitate the use of off-the-shelf tooling as these generally only support open protocols. In the newly designed architecture, the complexity lies in the limits set by the off-the-shelf API gateway, as discussed in the previous paragraph. In addition the customers now need to implement the client-side of the long polling mechanism, which also introduces complexity. Overall the decision about where to put complexity is up to software architects. This study should provide insight in the different available options, advantages and disadvantages of an API gateway implementation and protocol translations for an asynchronous reactive architecture.

7.2 Future Work

The future work opportunities we identified are discussed in the following sections.

7.2.1 Apply Directly to Thales API

The implementation created during this study is a proof of concept and is based on general interaction patterns of the current Thales API. To fully test the mechanism, a complete implementation on the customer API should be created. This requires a full API redesign, which can reveal practical issues that have not been foreseen during this research. Using this new design, it will be also possible to fully configure and test the API gateway in a practical scenario.

7.2.2 Analysis with Customer Implementation

As the currently asynchronous protocol was translated to a synchronous protocol, customers need to implement this new synchronous API. Analysis of current and new implementation can show whether the client implementations have been simplified based on customer feedback. Current API descriptions can be compared to the new (Open)Api descriptions, with code generation and other available tools. Documentation can be generated from API descriptions which can make it easier for customers to understand the API. Customers need to implement the long polling mechanism, which can introduce some complexity in the system.

7.2.3 Experiment with http/2

HTTP/2 is an upcoming protocol, which allows the server to send messages to the client before the client actually requests these. Also, the HTTP/2 protocol aims to be more efficient in comparison to the previous versions of the protocol. This protocol might be a more appropriate fit for this research, but most tooling does not support HTTP/2 at the moment. Experiments with HTTP/2 can reveal if this is a viable approach and whether there is currently enough support for the protocol to create a functional implementation.

7.2.4 Test more API gateways

During this study, the Tyk and Kong API gateways were tested. Other API gateways can be tested, as well as their plug-in systems. Example of other viable API gateways are Zuul, Gravitee and Express gateway. The plug-in systems can be used to implement features that are not available out of the box,like traceability and where possible request composition.

7.2.5 Performance Optimization

The current implementation has been created as a proof of concept without concerning high performance. Performance tests have shown that the performance of the current implementation is well within the bounds set by this research. Even though, performance improvements are possible. Performance should also be tested in a real-life scenario, as it is currently only tested in a simulation environment.

7.2.6 Validate Solution with Other Hardware Control Systems

This study is based on the Thales hardware control system, other companies which create hardware control systems can run into similar problems. Applying this solution in other systems may reveal limitations of the described solution.

8 Appendix

A OpenAPI Definition Example

```
openapi: 3.0.0
info:
  version: '0.0.1'
  title: 'State␣management'
  description: 'REST␣API␣for␣state␣management'
paths:
  /state:
    get:
      summary: Get the current state of the system
      responses:
        200:
          description: "Normal␣response"
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/State'
        401:
          description: "Unauthorized␣response"
    patch:
      summary: Request a state change
      responses:
        200:
          description: "The␣change␣request␣is␣accepted"
        401:
          description: "The␣change␣request␣is␣rejected,␣unauthorized"

components:
  schemas:
    State:
      required:
        - current_state
        - target_state
        - modification_id
      properties:
        current_state:
          type: string
          enum: [on, off]
        target_state:
          type: string
          enum: [on, off]
        modification_id:
          type: integer
```

B Docker Compose File

```
version: "3.4"

services:
  #######################################
  # Postgres: The database used by Kong
  #######################################
  kong-database:
    image: postgres:9.6
    restart: always
    ports:
      - "5432:5432"
    environment:
      POSTGRES_USER: kong
      POSTGRES_DB: kong
    healthcheck:
      test: ["CMD", "pg_isready", "-U", "kong"]
      interval: 5s
      timeout: 5s
      retries: 5
    networks:
      - kong-net

  #######################################
  # Kong database migration
  #######################################
  kong-migration:
    image: kong:latest
    command: "kong␣migrations␣bootstrap"
    restart: on-failure
    environment:
      KONG_DATABASE: postgres
      KONG_PG_HOST: kong-database
    depends_on:
      - kong-database
    networks:
      - kong-net

  #######################################
  # Kong: The API Gateway
  #######################################
  kong:
    image: kong:latest
    restart: always
    tty: true
    environment:
      KONG_DATABASE: postgres
      KONG_PG_HOST: kong-database
      KONG_PROXY_LISTEN: 0.0.0.0:8000
      KONG_PROXY_LISTEN_SSL: 0.0.0.0:8443
      KONG_ADMIN_LISTEN: 0.0.0.0:8001
      KONG_ADMIN_ACCESS_LOG: '/dev/stdout'
      KONG_ADMIN_ERROR_LOG: '/dev/stderr'
      KONG_PROXY_ACCESS_LOG: '/dev/stdout'
      KONG_PROXY_ERROR_LOG: '/dev/stderr'
    depends_on:
      - kong-migration
      - kong-database
    healthcheck:
      test: ["CMD", "curl", "-f", "http://kong:8001"]
      interval: 5s
      timeout: 2s
      retries: 15
    ports:
      - "8001:8001"
      - "8000:8000"
    networks:
      - kong-net

  #######################################
  # Konga database prepare
  #######################################
  konga-prepare:
    image: pantsel/konga
    command: "-c prepare -a postgres -u postgresql://kong@kong-
        database:5432/konga_db"
    restart: on-failure
    depends_on:
      - kong-database
    networks:
      - kong-net

  #######################################
  # Konga: Kong GUI
  #######################################
  konga:
    image: pantsel/konga
    restart: always
    environment:
      DB_ADAPTER: postgres
      DB_HOST: kong-database
      DB_USER: kong
      TOKEN_SECRET: removed
      DB_DATABASE: konga_db
      NODE_ENV: production
    depends_on:
      - kong-database
    ports:
      - "1337:1337"
    networks:
      - kong-net

networks:
  kong-net:
    driver: bridge
```

C Raw Benchmark Results

```
##### GET without authentication #####
Running 30s test @ http://localhost:8000/unauth/systemstate
  2 threads and 2 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   295.59us  108.30us   4.51ms   91.52%
    Req/Sec     3.40k   678.88     4.57k    60.07%
  203448 requests in 30.10s, 37.63MB read
Requests/sec:   6759.21
Transfer/sec:      1.25MB

##### GET with authentication #####
Running 30s test @ http://localhost:8000/auth/systemstate?key=
   test123
  2 threads and 2 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   372.44us   96.40us   4.05ms   91.01%
    Req/Sec     2.69k   215.43     3.44k    80.03%
  160806 requests in 30.10s, 29.74MB read
Requests/sec:   5342.37
Transfer/sec:      0.99MB

##### GET with trace (sample size 0.01)  #####
Running 30s test @ http://localhost:8000/trace2/systemstate
  2 threads and 2 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   451.04us  156.62us   5.30ms   92.54%
    Req/Sec     2.24k   291.81     2.73k    79.50%
  133853 requests in 30.00s, 24.76MB read
Requests/sec:   4461.64
Transfer/sec:    845.05KB

##### GET with trace (sample size 1)  #####
Running 30s test @ http://localhost:8000/trace/systemstate
  2 threads and 2 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   598.80us  338.37us  11.56ms   92.91%
    Req/Sec     1.74k   301.91     2.24k    73.33%
  104048 requests in 30.00s, 19.25MB read
Requests/sec:   3468.06
Transfer/sec:    656.87KB

##### PATCH normal #####
Running 30s test @ http://localhost:8000/unauth/systemstate
  2 threads and 2 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
```

```
    Latency   400.70us  105.42us   4.95ms   92.61%
    Req/Sec     2.50k   197.29     2.80k    81.53%
  149643 requests in 30.10s, 18.83MB read
Requests/sec:   4971.66
Transfer/sec:    640.64KB

##### PATCH with transformation #####
Running 30s test @ http://localhost:8000/transform/systemstate
  2 threads and 2 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   426.83us  113.94us   4.65ms   92.22%
    Req/Sec     2.35k   203.36     2.82k    80.03%
  140593 requests in 30.10s, 17.69MB read
Requests/sec:   4670.99
Transfer/sec:    601.89KB
```

References

[1] Ambassador. Ambassador api gateway. https://www.getambassador.io. Accessed: 2019-08-17.

[2] ApiAxle. Apiaxle gateway. https://github.com/apiaxle/apiaxle. Accessed: 2019-08-17.

[3] ApiMan. Apiman gateway. http://www.apiman.io. Accessed: 2019-08-17.

[4] Apioo. Fusio api management. https://www.fusio-project.org/. Accessed: 2019-08-17.

[5] D. Badampudi, C. Wohlin, and K. Petersen. Software component decision-making: In-house, oss, cots or outsourcing - a systematic literature review. *Journal of Systems and Software*, 121:105 – 124, 2016.

[6] S. Basu and T. Bultan. Choreography conformance via synchronizability. In *Proceedings of the 20th international conference on World wide web*, pages 795–804. ACM, 2011.

[7] S. Basu, T. Bultan, and M. Ouederni. Synchronizability for verification of asynchronously communicating systems. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 56–71. Springer, 2012.

[8] J. Bonr, D. Farley, R. Kuhn, and M. Thompson. The reactive manifesto. https://www.reactivemanifesto.org/. Accessed: 2019-02-08.

[9] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (soap) 1.1, 2000.

[10] J. Brutlag. Speed matters for google web search, 2009.

[11] T. Bultan, J. Su, and X. Fu. Analyzing conversations of web services. *IEEE Internet Computing*, 10(1):18–25, 2006.

[12] B. Christensen. Optimizing the netflix api. *Netflix Tech Blog.*, 2013.

[13] Containous. Traefic edge router. https://traefik.io. Accessed: 2019-08-17.

[14] Corvusoft. Restbed. https://github.com/Corvusoft/restbed. Accessed: 2019-07-03.

[15] Dreamfactory. Dreamfactory api gateway. https://www.dreamfactory.com. Accessed: 2019-08-17.

[16] Express. Express gateway. https://www.express-gateway.io/. Accessed: 2019-08-17.

[17] I. Fette and A. Melnikov. Rfc 6455: The websocket protocol. *IETF, December*, 2011.

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol–http/1.1, 1999.

[19] R. T. Fielding and R. N. Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine, USA, 2000.

[20] X. Fu, T. Bultan, and J. Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, 2005.

[21] Gravitee. Gravitee api gateway. https://gravitee.io/. Accessed: 2019-08-17.

[22] O. M. Group. Omg unified modeling language specification. In *Version 2.5.1,© 2017 Object Management Group, Inc*, 2017.

[23] GRPC. Original grpc release. https://grpc.io/blog/gablogpost. Accessed: 2019-03-12.

[24] I. Hickson. Server-sent events original draft. *W3C Working Draft WD-eventsource-20091222*, page 18, 2009.

[25] D. Jacobson. Embracing the differences: Inside the netflix api redesign. *Netflix*, 9:51, 2012.

[26] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018.

[27] Kong. Kong api gateway. https://konghq.com/kong/. Accessed: 2019-08-17.

[28] W.-T. Lo, R.-K. Sheu, Y.-S. Chang, and Y.-W. Chang. A restful web notification service. *Journal of the Chinese Institute of Engineers*, 39(4):429–435, 2016.

[29] Lyft. Envoy proxy. https://www.envoyproxy.io/. Accessed: 2019-08-17.

[30] Lyft. Open-source infrastructure at lyft. slideshare. Accessed: 2019-08-17.

[31] Netflix. Netflix zuul. https://github.com/Netflix/zuul. Accessed: 2019-02-21.

[32] Oktal. Pistache. https://github.com/oktal/pistache. Accessed: 2019-07-03.

[33] OpenApi. Openapi. https://www.openapis.org. Accessed: 2019-05-28.

[34] Pantsel. Konga. https://github.com/pantsel/konga. Accessed: 2019-09-03.

[35] C. Pautasso and E. Wilde. Push-enabling restful business processes. In *International Conference on Service-Oriented Computing*, pages 32–46. Springer, 2011.

[36] R. Stortelder. Introducing a legacy system to the service-oriented paradigm. 2019.

[37] Strongloop. Microgateway. https://github.com/strongloop/microgateway. Accessed: 2019-08-17.

[38] A. S. Tanenbaum and M. Van Steen. *Distributed systems*. Prentice-Hall, 2017.

[39] T. Technologies. Tyk api gateway. https://tyk.io/. Accessed: 2019-08-17.

[40] O. tools. Openapi tools. https://openapi.tools. Accessed: 2019-05-28.

[41] Treegateway. Tree api gateway. http://treegateway.com/. Accessed: 2019-08-17.

[42] A. Umbrella. Api umbrella gateway. https://apiumbrella.io. Accessed: 2019-08-17.

[43] wrk. wrk benchmarking tool. https://github.com/wg/wrk. Accessed: 2019-07-02.

[44] WSO2. Wso2 api gateway. https://wso2.com/api-management/. Accessed: 2019-08-17.