Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics: Games Engineering

Native Service Interfaces for the Virtual State Layer

Felix Kuperjans

Technische Universität München

Department of Informatics

Bachelor's Thesis in Informatics: Games Engineering

Native Service Interfaces for the Virtual State Layer

Native Dienstschnittstellen für die Virtual State Layer

Author	Felix Kuperjans
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Dr. Marc-Oliver Pahl
	Stefan Liebald, M.Sc.
Date	April 18, 2017

Informatik VIII
Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, April 18, 2017

Signature

Abstract

Current smart spaces often suffer from a lack of interconnectivity between devices or from security issues. The Virtual State Layer is a middleware which mediates between devices and services of different vendors. It is currently written in Java and defines a fixed set of operations which are used by services to interact with the middleware. In this thesis, a RESTful service interface is specified using standardized and interoperable network technologies, for the development of native connectors in other programming languages than Java. The interface uses HTTP and several standardized serialization formats (XML, JSON, CBOR, Google protocol buffers) through content negotiation. Security is required with TLS 1.2 or higher and client certificate authentication, to prevent accidentially insecure deployments. This specification is implemented and evaluated in Java, C and Python to provide native connectors for further service development.

Zusammenfassung

Smart Spaces sind heutzutage oft von fehlender Interkonnektivität oder Sicherheitsproblemen betroffen. Die Virtual State Layer ist eine Middleware, die zwischen den Geräten und Diensten verschiedener Anbieter vermittelt. Sie ist momentan in Java programmiert und definiert eine feste Menge von Operationen, die Dienste nutzen, um mit der Middleware zu interagieren. In dieser Arbeit wird eine RESTful gestaltete Dienstschnittstelle mithilfe von standardisierten und kompatiblen Netzwerktechnologien spezifiziert, um die Entwicklung nativer Anbindungen in anderen Programmiersprachen als Java zu ermöglichen. Die Schnittstelle nutzt HTTP und einige standardisierte Serialisierungsformate (XML, JSON, CBOR, Google protocol buffers) durch die Content Negotiation von HTTP. Sicherheit ist vorgeschrieben mittels TLS 1.2 oder neuer und Authentisierung mittels Klientenzertifikaten, um versehentlichen unsicheren Betrieb zu unterbinden. Diese Spezifikation ist in Java, C und Python umgesetzt und stellt native Anbindungen für weitere Dienstentwicklung bereit.

Contents

List of Figures

List of Tables

Listings

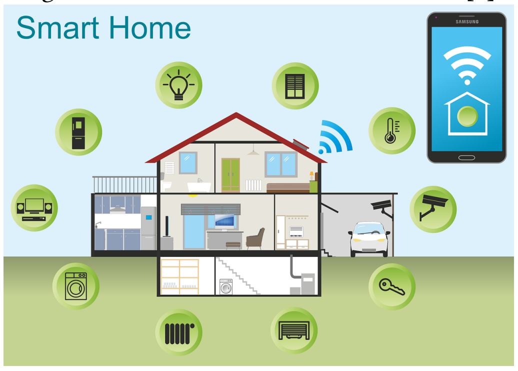
Chapter 1

Introduction

With the emergence of the internet, interconnectivity between electrical devices has grown each year. Starting with the mainframes of research centres, universities and big corporations, taking over all PCs in corporate and also private environments and then going further to mobile phones, televisions, watches and even household devices, more and more devices are connected to the internet. Using the new possibilities of communication between devices, access to these devices from anywhere and the possibility to control the devices from powerful "cloud" servers, interaction with the devices can be automated or driven by more intuitive user interaction flows.

These advanced methods of control over real devices turn a box with a simple on/off switch into a smart device that "knows" what the user wants without direct interaction (i.e. through a switch), by allowing control through voice recognition, presence detection or remote control from other devices. These smart interaction mechanisms provide a seamless and intuitive interaction between humans and the smart devices around us.

Figure 1.1: Smart home illustration from [1]



This vision of a smart space, where every smart device connects to the needs of its owner, knowing its owner's intention also from mimic or gestures, like for example adjusting the room temperature after detecting someone shivering, has not yet arrived at everyone's home. There are however very advanced research projects around specific applications or functions and automation in office buildings is usually already quite advanced.

The term smart space does not just apply to home automation ("Smart Home", shown in figure 1.1), but also for example to industrial automation ("Smart Factory"). This situation of having any kind of device connected to the internet, from television (smart TV), cars (connected car) to everyday devices such as light bulbs, is often referred to with the generic term "Internet of Things" (IoT).

In recent years, the whole market of smart devices or connected things has risen rapidly and is expected to grow even further. Gartner for example predicts, that in 2017 the number of connected IoT devices will grow by 31% to 8.4 billion devices [3]. Based on this rapid growth, a lot of new technologies and challenges emerge, on the one hand to technically handle this large number of devices in the Internet infrastructure and on the other hand to develop devices and device software which can connect to the Internet without using too much power or requiring powerful processors.

In addition to the challenge of getting that many smart devices connected, there are also challenges with the controlling of several connected devices so that they actually do smart interactions with the user. This management of many smart devices is also called orchestration, where complex flows are executed, which involve interactions with many smart devices. Also the configuration of who has access to the device and when or for which interactions certain devices can be used needs to be adjustable in complex smart spaces.

For example, a simple setting like 'the lighting in conference room 1 can only controlled by those employees, which are currently in the room' involves a lot of interaction between the devices: First of all, it needs to be determined who is in the room, for example through a facial recognition camera or an indoor localization service. Then the access rights to the lights in conference room 1 need to be adjusted accordingly and devices which do the interaction, for example a smart phone of an employee, needs to know which lights are in conference room 1 so it can display controls accordingly, like for example for the main lights, front lights and window shutters. The whole scenario gets even more complex when for example an energy saving service additionally fulfills the rule 'the lights in all conference rooms, which are empty, must be turned off'. In that case there is a potential conflict between the setting of the last person which controlled the device (i.e. an employee leaves the lights on when he left the room) and the energy saving service, which sets all lights in the empty room to off. This particular example does not lead to a conflict, because the employee only has access to the lighting when he is in the room and the energy saver is only active when the room is empty, but in some cases these rules can collide. For example if the rule would be 'after 6 PM (end of working hours), all lights should be turned off', a prioritization between these rules is required to allow an employee, which is doing overtime, to still turn on the lights when needed.

This challenge of smart space orchestration can be tackled with middleware, which enables such configurations to be performed, to deploy custom apps which add functionalities or flows to the smart space and to interact with all devices regardless of the vendor or technology. One middleware developed for this purpose is the Distributed Smart Space Orchestration System (DS2OS), which is extended in this thesis and described in more detail in section 2.1.2. It features a general abstraction of devices, their sensor data and actuators, through the Virtual State Layer (VSL, see 2.1.2.1) in which also pure orchestration services can derive data and perform actions. Additionally there is a service management layer, which features an app distribution infrastructure, high level management of the enabled services and tools for closer cooperation between developers to prevent incompatibilities between services and devices.

The next section illustrates common problems of smart spaces today, which led to the goals of my thesis which are presented in the section thereafter. This chapter then concludes with an outline of the thesis.

1.1 Common problems of smart spaces

There are several common problems which arise in smart spaces today and which affect the devices which are on the market or still in use. Some of these are:

Lack of interconnectivity
Currently most devices on the market are connected using vendor specific technology, which often leads to problems with communication and control of the devices across different vendors. In other cases the devices are connected to one specific cloud of a single provider and therefore rely on (1) all devices being connected to the same cloud and (2) the cloud provider to be accessible in order to communicate at all. This leads to isolated applications and a strong reliance on a specific vendor or cloud to still deliver the service, replacement devices and software updates.

For instance, a very popular product nowadays is a smart TV, which delivers video on demand services through the internet and can for example, be controlled remotely using a smartphone app. Additionally, someone could have a robot vacuum which can be controlled using a smartphone app, too. While the user is able to control both devices with his smartphone and both devices are connected to the Internet, these two devices are currently not connected. This leads to annoying user experiences, as for example the vacuum starts cleaning the living room based on a time schedule, although the smart TV in the same room is currently streaming a TV show, which means that a user in that room is watching TV and obviously does not want to be disturbed by the robot vacuum.

This problem is caused by having separate cloud infrastructures for these different devices and although the apps are installed on the same smartphone, they don't interact with each other. Additionally the current software usually lacks support for specifying rules to deny this kind of malfunction or for artificial intelligence or facial gesture recognition to realize on its own that this behavior is not desired by the user. This is exactly the kind of problem which is solved by using an independent middleware with extensibility through smart space apps.

Security issues
Many device vendors are not yet prepared to develop software and embedded device firmware that meets the security requirements of modern Internet applications. Frequent software updates, state of the art encryption techniques and good authentication and access control using a safe setup procedure are often not provided by the vendor. This led to many security incidents in recent times, often affecting thousands of devices, where usually even basic measures to protect the infrastructure were neglected.

A recent example is a security scan performed by security researcher Lucas Lundgren, which scanned for unprotected devices using the Message Queue Telemetry Transport (MQTT) protocol without authentication. The scan revealed in less than two days more than 59,000 insecure devices, ranging from devices in cars over an emergency news distribution system to medical equipment, flight information, ATMs and other critical systems, that could be read and partly controlled by anyone on the internet. The details were revealed at DEF CON 24 in 2016 [4].

Especially in critical public infrastructure, insecure smart devices can pose a serious risk for example if terrorist hackers use the security issues to trigger off a dangerous malfunction of the devices. An example scenario was investigated by researchers at the Georgia Institute of Technology, in which they made a proof of concept malware, which was able to infect the controllers of valves in a water treatment plant. They could then increase the amount of chlorine added to the water while displaying wrong data to the operating staff. This attack, which was performed in an isolated environment with a nontoxic addition instead of chlorine, shows that similar approaches could be used for ransom or terrorist attacks [5].

Privacy concerns
There are many privacy concerns related to smart spaces, first of all due to the aforementioned security issues, many devices provide their data to everyone on the Internet, which is of course unacceptable. But further issues arise for example if devices provide all their raw data to a cloud infrastructure and the provider might have regulations which are too open in its data privacy statement or in some countries the privacy of these services is compromised by statutory requirements to provide data for example to public authorities. This is especially concerning with many of these devices, like cameras or microphones for presence detection or voice recognition, providing audio and video recordings from within private property and therefore potentially violating the sanctity of the home. In order to secure private data, the amount of data exposure to systems of the vendor must be controllable by the user and substantial functionality and data processing should be provided by the devices within the smart space, to make

it independent of third party servers or cloud systems.

1.2 Goals of the thesis

The primary goal of the thesis is to create native connectors to different programming languages and platforms for the Virtual State Layer middleware. A secondary goal that derives from this primary goal is to standardize the network communication between the connector and the middleware (the service interface, more on this term in 2.1.1) using interoperable technologies, so that the implementation of these connectors is possible and maintainable.

Regarding the common problems of smart spaces I mentioned before, this is how my work relates to them:

Interconnectivity between devices is already improved a lot with the Virtual State Layer middleware itself, regarding the cooperation between devices and a vendor independent layer to provide orchestration workflows which can incorporate all smart devices of a smart space. In order to make all these devices actually be able to connect to the VSL middleware, currently they would have to use the Java Virtual Machine as a platform and developers would be limited to this choice. With my extensions to provide native connectors and a standardized service interface which allows other developers to easily create more connectors, the overall interoperability rises and more devices could be connected regardless of their platform.

Security issues are often related to insecure network protocols, for example because they lack encryption or reliable authentication methods. It is part of my requirements in 2.1.3 to only consider technologies that can be deployed securely and to use them with current best practices for Internet communication. Additionally, the Virtual State Layer middleware provides sophisticated access control methods and a service management layer to give authorized users the ability to control which devices and services are allowed to do what.

Privacy concerns are addressed in two ways, first of all because there is a high level of security to prevent unauthorized access and secondly, the Virtual State Layer performs local processing inside your smart space and gives the user control over how and what specific data are for example transmitted to cloud systems. It can also be operated completely local, but still offers the ability to use cloud connectors for advanced features.

To sum up, in this thesis a

- service interface with standard network protocols is designed and evaluated and
- native connector implementations for other languages than Java are implemented and evaluated, to ease the development of services on other platforms.

This work contributes to solving common problems of smart spaces today and the detailed requirements of my work are elaborated in 2.1.3.

1.3 Outline

The thesis is structured into seven chapters, the first of them being this introduction.

The next chapter, chapter 2, contains the analysis of the problem domain and existing technologies which can be used for solving these problems. It begins with the problem statement in 2.1, in which the relevant definitions and the Virtual State Layer middleware are introduced and then the specific problems to solve and requirements of the solution are derived. The rest of the analysis describes existing technologies which can be used for fulfilling parts of the requirements or solving a part of the problem. In each of the sections, a particular family of technologies is introduced and afterwards assessed to what extent they can be used or which technology is more suitable.

Chapter 3 presents related work, where other researchers worked with similar or related problems and analyses the results and findings and how they can be applied to my approach as well.

After this assessment of existing technologies and research, the design of the service interface is specified in chapter 4 based on the technologies which were identified as suitable to fulfill my requirements. This chapter is closely followed by chapter 5, which describes my experiences from actually implementing the native connectors and the problems which were solved during the implementation, as well as giving some insights to implementation details like the used libraries.

Chapter 6 aggregates the details on how I evaluated the implemented connectors if they actually meet the requirements based on quantitative measurements of the serialization modules and qualitative analysis of whether the requirements are fulfilled and how easy to use and interoperable the service interface actually is.

Finally, the results are concluded in chapter 7, where the findings are summarized and put into context with the introduced smart space scenario.

Chapter 2

Analysis

In this chapter, the problem domain and the requirements are introduced and then relevant technologies that propose solutions to individual requirements are analyzed and assessed.

First, the term service interface is defined with relation to middleware and the role of service interfaces in large scale software systems is emphasized. In that part, the motivation why native interfaces for multiple programming languages or operating systems are important for widespread use of a middleware is reasoned. Then the specific middleware for which I create a service interface is introduced and the functions which the interface needs to cover are provided. A specification of the requirements for the native service interfaces is derived and specified in detail to conclude the first part of this chapter.

In the second part, technologies which can be used to fulfill some of the requirements are analyzed. This starts with an introduction to different architectural paradigms which are also compared with regard to their field of appliance and their representation of required functionalities.

As the interface is required to be usable via network connections, different network technologies which are suitable candidates for the creation of this interface are presented. The representation of data structures in network communication, called data serialization, is analyzed in depth with different serialization formats that are assessed for this purpose. Following the serialization formats, the standard protocols which are suited for the requirements are presented and assessed. A deeper look into different ways how callbacks are realized with these protocols rounds up the discussion of suitable network technologies.

In order to create native interfaces to a lot of programming languages, standardized interface description languages and code generation can be useful. The term interface description language is defined and put into the context of code generation. Then, different approaches to code generation with or without the usage of interface description languages are analyzed and compared with a focus on generating code for a lot of programming languages with the required functionalities.

Finally a summary concludes this chapter with an overview of the analyzed technologies and which ones are useful for the development of the service interface.

2.1 Problem statement

This section motivates the topic of this thesis and leads to a definition of the requirements for the interface. The first subsection defines the term service interface and the role of service interfaces for a middleware in large scale software systems.

2.1.1 Service interfaces for a middleware

The W3C working group defines a service interface as

"[...] the abstract boundary that a service exposes. It defines the types of messages and the message exchange patterns that are involved in interacting with the service, together with any conditions implied by those messages." [6]

A middleware is a general-purpose interface between platforms (e.g. operating system) and applications. It can be used by a wide variety of applications and enables these to run on all platforms which the middleware supports. The applications don't interact with the operating system directly and instead only use the middleware's application programming interface (API) [7].

A middleware is also distributed, it can be accessed remotely or enables applications to be remotely accessible. Normally it builds up on standard network protocols like the TCP/IP stack and allows applications to access these remote resources without handling the network connections on their own. So by just using the middleware's API, applications can interact with the operating systems and applications on the local or remote machines [7].

This is a very important property for large scale software systems where a middleware greatly increases the ease of application development and deployment [8]. By interacting with the middleware only, applications can be enabled to operate in a distributed computing environment at any scale which the middleware effectively supports.

The network communication of a middleware is defined by its service interface. Middleware components that communicate to each other on the different platforms and maybe using different programming languages need to implement this service interface. The choice of the service interface is very important for the potential platform coverage and portability of the middleware itself and therefore also the applications that can run on the middleware.

In the next subsection, the specific middleware is presented, for which a service interface is developed in this thesis.

2.1.2 DS2OS - the Distributed Smart Space Orchestration System

The Distributed Smart Space Orchestration System (DS2OS) is a system for orchestration of devices in a smart space.

A smart space is a room or an area, in which embedded computing devices can interact with the space by sensing its state with sensors and interacting with the space by performing actions through actuators. This could be for example temperature sensors in a room measuring the temperature and adjusting the heating to raise or lower the temperature to a desired level. The devices communicate with each other and can interact with the users in the smart space. Even a device or software can change its functions based on the space it is in, like for example an app on a smartphone being able to control the room where the device is located in, but only this room at the time when the device is actually there [9].

Orchestration of these different devices which each run their own software and operating system for controlling their sensors and actuators requires a high degree of data exchange between these devices [9]. In DS2OS, two main components faciliate this: the service management component manages the different services and the the virtual state layer (VSL), a middleware for the interaction between these services. This middleware is the component of DS2OS, which is extended in this thesis and explained in more detail in the next sections.

I also worked as a student research assistant on this project before and contributed for example to the architecture of the knowledge agent, which is described in more detail in 2.1.2.3. Some of the statements in this subsection are based on my knowledge of the code and architecture and are not based on published papers listed in the references.

The most relevant parts of the system are now presented in detail, beginning with the Virtual State Layer (VSL) middleware and its special concept of virtual nodes.

2.1.2.1 Virtual State Layer

The Virtual State Layer (VSL) is a tree structure of nodes containing typed information. The information of a node represents a state of a resource in the smart space, which can be state of a physical entity or a state information inferred by software [10].

This node can be accessed using a set of standard methods (described in 2.1.2.4) with an unique address which is valid on all devices which are connected to this DS2OS instance. In order to change a value in the VSL, it is only necessary to have access to one knowledge agent (KA), which are the entities which each manage a subtree of the VSL. Using the connection to this KA, all knowledge in the VSL is accessible to the client using the VSL middleware, as long as the client has the required access rights [10].

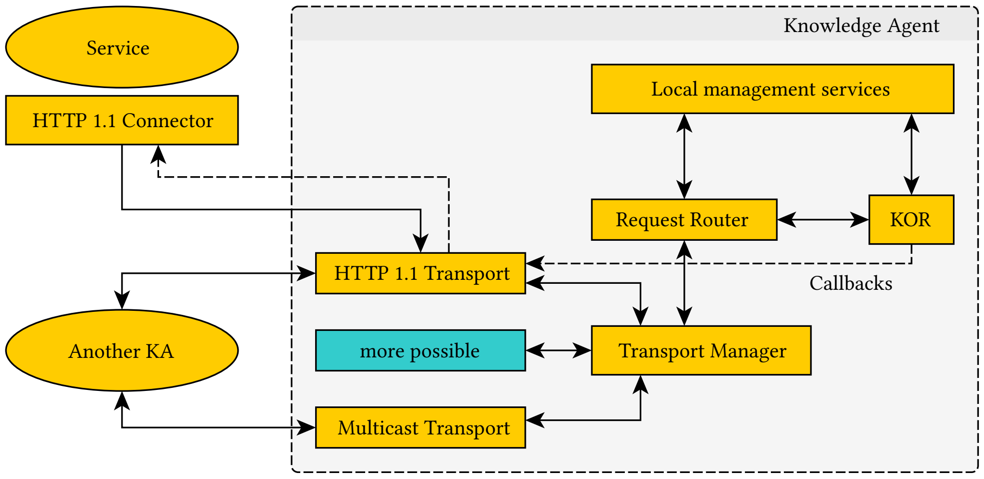
Similar to a file system, the nodes in the VSL form a tree structure based on the addresses, where address components are separated by a slash. Every service connected to the VSL via one of the knowledge agents can register a private subtree of the VSL, which is a node of a specified type. The types are derived from basic types like either primitive types (string, number, ...) or a composed type. In case of the composite type, it declares a set of child nodes with the corresponding child types. This way the node is the root of a subtree where all subnodes are known from the type information. An exception to these statically known children are lists, where nodes can be added or removed at runtime, but in that case the type of the list's root node is list, so the list behaviour of that node is known by its type. In most cases, a service will instantiate a composite node with children for the different attributes or functions of the service which might represent a device or a management functionality [10].

2.1.2.2 Virtual Nodes

The data in the VSL can be dynamically overlayed by the service which provides the data with live data in realtime. This can be transparently achieved by the service by registering a **virtual node** with the knowledge agent. Other services will not be influenced by this action but the knowledge agent forwards the requests directly to the service. The service can then provide live data instead of frequently pushing changes to the knowlege agent's knowledge base, allowing for less interactions as the data is only updated when it is actually requested and also allowing for realtime data access [11].

This is technically achieved by using callbacks to the service which registered the virtual node. Upon registration, the callback is provided to the knowledge agent and the knowledge agent will then use the callback to forward the requests of other services to the service which registered the virtual node. Additionally, the knowledge agent saves the data provided by the callback to the local knowledge base, but only every time the data is requested by another service. This allows the knowledge agent to provide data from the knowledge base again if the callback becomes unavailable, for example because the service which registered the virtual node is suspended or defect. The same happens when the service unregisters a virtual node: the node's data will be provided by the knowledge agent from its knowledge base which stores the value which is the most current value that is available for this node. It is no longer a realtime value but at least the most recent one that is known. Once a service registers the virtual node again

Figure 2.1: Knowledge agent architecture



it will be served directly with the callback again [11].

This design improves the resilience of the smart space especially if services frequently change its availability, for example for power saving or because of network migrations of moving devices. Outdated values can be detected by the requesting services, for example via the timestamps. So services that rely on realtime data can verify whether the data is recent enough or not.

2.1.2.3 Current knowledge agent architecture

The architecture of the most important entity of the Virtual State Layer (VSL) middleware, the knowledge agent (KA), as shown in figure 2.1. It features the **knowledge object repository (KOR)** for storing the local VSL subtree data, but also metadata of the whole VSL and state information of local services like which nodes are registered as virtual. Furthermore, this instance also provides access control checking and invokes callbacks for example on virtual node accesses in the local VSL. All requests that access the local VSL subtree are passed to the KOR.

The knowledge agent also has **local management services**, that deal with different high-level system management functions. While they usually act similar to normal services, some of them use direct access to the KOR to allow for special operations like type search. Requests from these or other services are routed by the **Request Router** first, to destinguish between local and remote requests and to get the request handled by the responsible component of the KA. This part represents the inner working of a KA to get the requests executed and the other part, which is described in more detail, deals with the request communication via network.

In this architecture, support for multiple different network ”transports”, i.e. specific implementations of a protocol, is already provided with the **Transport Manager**. The Transport Manager manages the available transports and delivers requests to other knowledge agents by selecting a suitable transport module for the communication. Secondly, the transports can also receive requests, either from other KAs or from services, and pass them to the Request Router for further processing. There are two kinds of transport, that serve different purposes:

- **unicast transports**: these serve the peer to peer communication of KA to KA or service to KA and allow to execute one of the requests described in 2.1.2.4. This kind of transport is the one to be developed or improved in this thesis.
- **multicast transports**: the purpose of these is to maintain the overlay network, which is formed by the knowledge agents. So it is only used for KA to KA communication and only for specific operations like agent discovery, heartbeats respective keep alive pings and multicast-based exchange of VSL metadata. The current implementation uses IPv4 broadcast and IPv6 multicast for this purpose.

The only unicast module, that is currently available, is an implementation with the HTTP 1.1 protocol and JSON data serialization (see 2.4.2 on HTTP and 2.3.2 on JSON). By adding more transport modules, services are able to use different protocols and even if the request was received using HTTP 1.1 and needs to be sent to another KA for processing, the usage of a different protocol for the KA to KA communication is possible. This makes testing different protocols and implementations easily possible even within the same instance of a KA through configuration or client changes.

For realizing some features like the virtual nodes from 2.1.2.2, callbacks are very essential. As it can be seen in the architecture diagram in 2.1, callbacks that were registered from a transport module at the local KOR will be directly called from the KOR to the transport module. The transport module is then responsible for doing the actual callback invocation and to maintain the callback's availability (i.e. disable it if a service disconnects). Currently, the HTTP 1.1 transport uses a WebSocket (see 2.4.5) for the callbacks.

The next part describes the service interface of the virtual state layer in detail, stating the different methods used by services to access the VSL. These are also implemented in the existing HTTP 1.1 connector.

2.1.2.4 VSL interface

Access to the data in the VSL is provided by a fixed set of methods, which are defined in the VSL interface. Every interaction with the smart space can be done by using one or more calls to these methods.

The methods used by the services - there are only few extensions for the communication between agents - which are used for all VSL accesses are these:

- **get(address): VslNode** - Used to retrieve a VSL node by its address.
- **set(address, data)** - Used to set data on a VSL node by its address.
- **notify(address)** - Used to notify the KA about changes on a virtual node, so it can notify the subscribers.
- **subscribe(address, callback)** - Subscribes to changes to the node at this address.
- **unsubscribe(address)** - Removes subscriptions at this address.
- **lockSubtree(address, lockCallback)** - Locks a VSL subtree to track changes made by this service in a transaction.
- **commitSubtree(address)** - Commits the changes done to a locked subtree.
- **rollbackSubtree(address)** - Rolls back the changes of a locked subtree.
- **registerVirtualNode(address, virtualNodeCallback)** - Creates a virtual node at this address - see 2.1.2.2 for a description of virtual nodes.
- **unregisterVirtualNode(address)** - Removes the virtual node from this address.
- **registerService(manifest): String** - Instantiate this service's instance with the specified service manifest. It returns the service's instantiation address and multiple invocations lead to one instance at one address.
- **unregisterService()** - Remove this service's instance from the VSL.

This interface consists of very simple data manipulation methods and special methods that utilize callbacks for real time communication with the service. The service itself can control whether the data accesses are based on requesting the data explicitly upon need (pull) or whether a callback is triggered upon interaction (push). Advanced functions like the virtual nodes rely on the callbacks, so a reliable callback mechanism is crucial for the VSL interface.

These are the DS2OS features which are most relevant for this thesis and many more features exist and can be discovered in [10] [11]. The next section will provide an overview of the requirements for a service interface which implements these methods to provide an API to the VSL.

2.1.3 Requirements of the service interface

The primary requirements of the VSL service interface derive directly from the functional interface described in 2.1.2.4. The functions listed in the interface must be usable via network connections for connectivity between multiple smart devices. It is the main goal of this thesis to make these functions natively available in many programming languages and on many platforms. This will ease service development as programs written in those languages can be adopted to connect to the VSL middleware natively in that language.

In order to achieve this, a high interoperability of the used technology is crucial for having many existent implementations or libraries on various platforms. The interoperability can already be provided by a standardized network protocol which has already been used in a lot of other projects. The smart devices will communicate via network, which allows the devices to also use different programming languages and still connect to instances written in other languages. It is also easier to implement the connector to the interface using standard libraries of the respective language for standardized network protocols. As part of this thesis, different standard protocols are evaluated and compared with qualitative analysis and quantitative performance metrics.

The quantitative key performance indicators are used to measure the efficiency of the protocol for usage in a smart space. As the embedded devices usually do not have high computing performance and some of them might even be battery driven, a low protocol overhead is important to reduce the amount of data, that needs to be transmitted, encrypted and processed. Another important aspect is the latency, as the controlling of devices should have an immediate response to user interaction. Therefore, a low latency improves the user experience and is an important metric for the technology comparison.

For the qualitative comparison, the most important aspect is the usability during implementation and its simplicity. With only nine methods of the interface itself, the implementation should not yield a complex client binding. Additionally, a simple binding will involve less maintenance effort and is easier to port to more platforms or languages.

Furthermore, security is very important in a smart space environment, where potentially insecure devices might share a network. This could be for example due to guests in an office or private network, hacked devices or reduced physical network security. The protocol should therefore support strong encryption and sophisticated cryptographic authentication mechanisms. While in some specific cases it might be useful to disable encryption, it should be part of the transport by design and enabled by default.

In addition to the mentioned requirements, these properties are very desirable in specific use cases:

- Protocol support for callbacks, especially through pushing mechanisms. Four methods of the service interface use callbacks, especially the virtual nodes make havy use of callback invocations. The different possiblities how to implement the callbacks efficiently will be an important characteristic for choosing the protocols and the way to implement the callbacks.

Table 2.1: Table of requirements

#	**Requirement**	**Analysis criterea**
1	standardized network protocol	standard accepted by IETF
2	interoperability	supported languages and platforms
3	low overhead of data transfers	actual data per transmitted byte
4	low latency of a full operation	number of round trips processing need
5	simplicity of the implementation	lines of code needed per language easy to use APIs available
6	security: encryption, authentication	strength of supported cryptography effort to include it
7	callback support	efficiency and ease to use callbacks
8	asynchronous operations	effort to achieve them
9	stateless/suspendable protocol	number of round trips to reactivate

- Asynchronous request execution: the data requested from the VSL can in some cases be very large, which leads to substantial transmission times depending on the network speed. If asynchronous operations are not possible in the protocol, services would not be able to execute small requests during the transmission of one large data chunk. A solution could be to establish multiple client connections, but native support for asynchronous operations would be better.
- Stateless or suspendable protocol: smart devices might have an unreliable power source, run out of battery or hibernate for the reduction of power consumption. In these cases, re-establishing the whole communication is far more expensive (in terms of time, energy consumption) than having a stateless protocol or a suspendable protocol that does not need a full reconnection in these cases.

These requirements are structured and numbered in the table 2.1 for later references.

2.1.3.1 Native interfaces to provide

The actual programming languages, for which native interfaces are developed in this thesis are chosen based on popularity and the goal to choose a set of not too similar programming languages. For measuring the popularity, the TIOBE index of 2016 (12 month avarage) [12] gives a good overview based on the number of search engine results for this language.

The top ten of the TIOBE ranking in 2016 (12 month average) is:

1. Java
2. C
3. C++
4. C#
5. Python
6. PHP
7. JavaScript
8. VisualBasic .NET
9. Perl
10. Assembly

For my thesis I choose the following three programming languages for the actual implementations based on the following reasons:

- **Java**: number one on TIOBE and it is the language in which the knowledge agent is already written in.
- **C**: number two on TIOBE and it is also a base language for C++, which is number three, and partly also for C# on the fourth place. C is also commonly used for embedded programming.
- **Python**: having risen to number five on TIOBE in the recent years, it is a popular script language which differs from Java or C.

The next part of the chapter will analyse existing technologies that can be used to fulfill these requirements.

2.2 Service interface architecture

In this section, different approaches to the architecture of service interfaces and service interaction are presented. These architectural principles present an idea of how data is transferred, how methods are invoked and what maintains the state of the system as a whole. Depending on the architecture, the role of certain entities in the system differ and with that the different tasks it has to fulfill to make the system working properly. In each subsection, the relation of this architectural principle to the VSL middleware (cf. 2.1.2.1) and the design of the service interface is shown. The first introduced principle is the well known Service Oriented Architecture.

2.2.1 Service Oriented Architecture

In a Service Oriented Architecture (SOA), a system is split up into services that communicate via network. These services encapsulate a specific functionality into an independant unit that hides its internal working from the service's consumers. In order to achieve a complex task – in SOA terms referred to as ”business process” – the service consumer may also use the functionality of other services, leading to a combination of services that is composed for this task [13].

Among the principles of SOA service design is the loose coupling of services. It implies that the functionality of a service, which is specified by its interface, can be offered by more than one implementation or service provider. Combined with a mechanism to find services that implement a specific interface, this allows services to be added, removed or moved to different network locations. This can be achieved with a service repository or directory service which can be queried for services [13].

Depending on the specific implementation, there are standards how a directory service works. One important example is UDDI which is used in combination with SOAP (see 2.4.7) [14]. Also DS2OS (see 2.1.2) has a service management component which amongst other functionality also serves as a service directory: one way is the App Store, which serves as a global directory for available service implementations and another is the type search functionality which allows to find instances (= service implementation) of a model (= abstract service definition) [15].

In many cases of a SOA implementation, the service interface defines methods that can be called by others in the interface. In contrast, DS2OS uses a fixed set of methods and the interface is the data structure of the service defined by the service's type [15]. This is described by an architectural principle called data-centric design, which is presented in the next subsection.

2.2.2 Data-Centric Design

Data-centric design moves away from the idea of using different methods to interact with different object types, but instead describes all object properties as data, which can be accessed using a set of standard methods. This allows service consumer implementations to only implement a fixed set of methods, but to apply them to different data [15].

There are several technologies based on this design principle, which is sometimes also referred to as ”Information-Centric”, ”Content-Centric” or ”Data-Oriented”. A well-known example is ”Content-Centric Networking”, which is an approach of reorganizing computer networks to make data requests not for a specific location like a URL, but instead for a specific named data object (NDO), which can be provided by any node that cached this object. If a particular data object is requested, the request can be satisfied by any network node knowing the data. Especially even nodes which do not interpret this specific data type can still cache and forward the data, without knowing the meaning of the data or loading code with specific support for this data object [16].

This design principle has several advantages and disadvantages, that are now discussed in relation to the distributed smart space orchestration system (DS2OS, see 2.1.2), which uses a Data-Centric Design with the fixed interface methods described in 2.1.2.4.

One advantage is the location independance of data objects. The data structure from the DS2OS model has the same meaning in different locations of the virtual state layer (VSL), allowing services to be instantiated on any knowledge agent (KA). The type search functionality allows to search for data in the whole VSL tree, to allow finding data instances location-independant in the whole VSL. This enables service mobility and an interesting use case of service composition, in which case the individual services are subnodes of the composed service [15]. A practical example from smart spaces would be a light bulb with an integrated temperature sensor, which could use a subnode "temperature" in its model that is instantiating the model for temperature measurement devices and another subnode "light" which is instantiating the model for light bulbs.

A problem arising from the Data-Centric approach is the realization of inheritance as in the Object-Oriented Programming (OOP, see [17]). The problematic part is extending a data structure without breaking code that does not support this specific extension. Apart from the composition approach from the previous paragraph, DS2OS supports type inheritance to solve this issue, as shown in [15].

Another important benefit is the centralization of access control in a Data-Centric designed software: Access rights and restrictions can be easily attached to data objects without knowing the actual meaning of the data. Everyone accessing the data object with any of the fixed methods is required to have access to the data globally in the system. Compared to this, a Service Oriented Architecture (SOA) with different service interfaces for each service requires every service to implement access control to its data on its own or to restrict access to calling a method, but without control of which data actually is used within this method's execution. This leads to a way easier implementation of reliable access control mechanisms in a Data-Centric designed SOA [10].

2.2.3 Event Driven Architecture

In subsection 2.2.1, the Service Oriented Architecture (SOA) was introduced with services that expose a service interface of methods to other services or consumers. The Data-Centric Design of the previous subsection proposes a SOA implementation where data objects instead of methods are used to represent the individual service behavior. A third popular method of service communication and service design in SOA is presented here, the Event Driven Architecture (EDA).

In the Event Driven Architecture, the communication of services is not modeled by method calls but with events. A service produces events and other services can receive or listen to these events. The program logic of a service is usually triggered by incoming events and then possibly produces further events to other services. In some cases, events might not even be listened to at all, so they eventually have no effect. Data exchange between the services only happens through the data stored in an event, not by any shared data structures or databases [18] [19].

One of the major benefits of an Event Driven Architecture is that because all data is encapsulated into the events, it is very easy to parallelize the processing of events without the need to synchronize data access. This leads to a high scalability of the whole system, as horizontal parallelism can be easily achieved [18]. Also it infers an extremely loose coupling of the services, as the event producing service does not even know how an event will be processed later on [19].

On the other hand, the traditional request and response workflow is not that easy in an Event Driven Architecture: an event can be used to represent a request and another (response) event can be raised by the service that handled to request, but matching the individual requests and responses to each other is more difficult in case many requests of the same type have been raised at the same time. The matching can be performed using the event data and client logic to identify the correct response. If a lot of specific requests to other services are required, it is a good option to mix the Event Driven Architecture with a traditional SOA [20] [19].

Another important application of EDAs are real-time applications, which profit from several properties of the EDA: First of all, events can be reordered by priorities or deadlines in their processing, allowing to meet deadlines more easily. Secondly, an event can be dropped (no longer processed) if its deadline expired without harming the operation of other parts of the system. With the asynchronous and very lightweight behavior of events, very efficient real-time systems can be developed using an EDA [20].

Network communication between the services usually models the events as simple messages, that are passed to other services of the system. The distribution of the events is done by event routers, that distribute the event to the listeners and processors. These simple messages can be passed using almost any network protocol and are very lightweight [18] [20].

The Distributed Smart Space Orchestration System (DS2OS, see 2.1.2) has three components that represent events: the subscription mechanism that notifies other services about changes in the VSL (see 2.1.2.1), the virtual nodes (see 2.1.2.2) that emit notifications on changes and the callbacks of the locking mechanism. So most of the callbacks in DS2OS are used to promote events to other services. Protocols and mechanisms of EDAs are good candidates for the realization of the callback mechanism and subscription's notification emission.

2.2.4 Remote Procedure Call

After the introduction of the Service Oriented Architecture (SOA) and the more specific architectures of service interaction in a SOA, a deeper analysis of network communication architectures is now performed. The first presented approach is the Remote Procedure Call (RPC), which provides a very simple architecture for service communication via network.

The basic idea of the Remote Procedure Call is to do a call similar to a normal call in a high-level programming language, just that the execution of the procedure takes place on another computer in the network. This is achieved by sending a network packet specifying the procedure to call and the arguments to the host of this call. The response is a packet containing either the result of the procedure or an exception, in case there occurred an exception on the callee [21].

The main benefit of this architecture is its simplicity and easy integration into existing programs. On the other hand, the blocking nature of these calls tend to slow down parallel workloads and the very simple architecture does not transparently map the semantics of objects or standard operations, for which extensions exist [22] [23].

For the network architecture of the DS2OS service interface, a more advanced architecture like the Representional State Transfer, which is presented in the next subsection, seems to be more suited. Still, RPC based protocols are analyzed like for example XML-RPC (see 2.4.6), as they are a viable method for the implementation of the service interface. Especially for the callback mechanism, the RPC architecture is highly suited to invoke the callback methods on the service.

The next subsection will present the Representional State Transfer (REST) in detail.

2.2.5 Representional State Transfer

Representional State Transfer (REST) is a term introduced by Roy Fielding to describe the architecture of web services [24]. A web service is considered RESTful if it meets the architectural constrains defined in his thesis. REST is a hybrid architecture which combines several architectural principles:

- **Client-server**: Data is stored on the server and the client is only a user interface. Additionally, both sides can evolve independently, increasing the interoperability.
- **Stateless server**: The request performed by the client contains all necessary information for the server to process the request. The server does not maintain a state for the client.
- **Cacheability**: Each response must state wether it is cachable or not and if it is cachable, a client may reuse the response without further confirmation.

- **Uniform interface between components**: Decoupling of implementations and services, divided into these subconstraints:
  - Identification of resources: Every resource must be indentifiable by a resource identifier, but its concrete format is not specified.
  - Every resource must be representable by data and metadata which describes this data, possibly the state of the resource and control information like the requested cache behaviour.
  - The data format of the representation must be included in the message, known as media type.
  - Hypermedia as the engine of application state: As REST request are stateless, the state must be maintained by the requests and their follow-up requests, making the sequence of requests a finite automaton. This is only possible if the resource contains possible follow-up requests as hypermedia references and clients can locate each resource from a single entry resource. This allows clients without detailed knowledge of the available resources or possible operations to use the service, which improves long-term maintenance [25].
- **Layered system style**: Services may be composed of hierarchical layers but a component does not know what or how many layers are behind the component they are interacting with.
- **Code on demand** (optional constraint): In order to ease the implementation of a client, a server may provide additional code snippets or applets which extend the client software.

The constraint that RESTful interfaces must be uniform is often not adhered, which is why many REST services that implement an RPC (Remote procedure call) architecture are regarded as REST-RPC hybrid services and not properly RESTful [25] [26]. There has been multiple aproaches to judge the ”RESTfulness” of services, for example the Richardson Maturity Model [27] and an analysis of several impacting factors by NORD Software Consulting [28]. The Richardson Maturity Model is explained in more details now.

2.2.5.1 Richardson Maturity Model

The Richardson Maturity Model (explained by [27], based on the slides of Richardson’s talk [29]) arranges HTTP (see 2.4.2) based web services on a scale from zero to three, with zero being not RESTful at all and three being RESTful according to Fielding’s thesis. It is only used for HTTP based services and is an extension to the REST definition, not part of it, i.e. true RESTfulness is only reached at level three, which is full conformance to the definition. The different levels in detail (all higher levels include the conditions of all previous levels, so only the new conditions of each level are listed):

- Level 0: The service is using HTTP, but it is not RESTful. For example, it only uses POST operations to a single URI and all content is put into the POST bodies, like XML-RPC (see 2.4.6) and SOAP (see 2.4.7) do.
- Level 1: Different URIs are used to identify resources, but still only one operation, usually POST, is used. In this Level, resources act like objects in object oriented programming; the URI points to an instance of the object and the POST operations are like calls to a method of this instance.
- Level 2: HTTP operations are used in their intentional meaning and not for tunneling custom methods. This usually includes using many different HTTP operations and POST operations only in cases of complex modification of a resource. In addition, the HTTP rules for these operations, like idempotency of GET, must be adhered and HTTP error codes must be properly used instead of error messages in the response body.
- Level 3: Hypermedia is used as the engine of application state, i.e. all URIs can be discovered by previous operations, so that all resources can be reached from a single entrance URI. Addtionally, the operations (esp. POST semantic) should be documented and this documentation should be delivered by the service.

This model can be used to improve the understanding of REST and assist in the development of RESTful services, by allowing a step-by-step development of the interface, rising the design from level zero (a simple RPC approach) to level three, with clear steps what can be done next (i.e. to go from level zero to one, one must identify resources respectively objects and transform the global RPC calls to object oriented methods) [27].

2.2.5.2 RESTful VSL

By looking at the structure and interface of the Virtual State Layer (VSL, see 2.1.2.1 and 2.1.2.4), many similarities to the REST architecture can be discovered: the addressing of VSL nodes for instance, is very similar to the REST URI addressing, also following the principle of having a unique address per resource or in this case, service instance. Furthermore, interface methods like get and set directly map to the HTTP operations GET and PUT, while the other methods can be done in an RPC style.

In the Richardson Maturity Model of the previous chapter, reaching a Level 2 maturity is straightforward by using the HTTP operations for the VSL operations as applicable. About the other REST criteria, the client-server architecture is already achieved with the knowledge agent holding the data of the service, i.e. the knowledge agent is the server and the services are clients in REST terminology. The statelessness of services (clients) is only achieved as long as no callbacks are registered, as maintaining callback information is a client-specific state. Cacheability and a layered system style are already part of the existing architecture, with a caching mechanism integrated into the VSL and services being agnostic of the connections between different knowledge agents and instead they just communicate with their KA.

Regarding the uniform interface, the criteria of resource identification and metadata to describe it is given with the VSL type information. The data format of the representation would be done at the actual protocol level, which will be discussed in the design of the service interface. Not using Hypermedia as the engine of application state is finally the only real violation of REST in the VSL design, as the linking of one resource to the other does not exist without specific implementation knowledge about the type search, the working of these references and the model of a service (which is not provided with the data yet but only by separate requests).

This shows that the VSL middleware's architecture highly relates to a RESTful architecture, while not being strictly RESTful yet. Therefore, the REST architecture is an important guide for designing the service interface of the VSL. Additionally, protocols and other technologies which are frequently used to implement RESTful services are especially suited for this project. The next sections contain a detailed analysis of these protocols and related technologies such as the data serialization.

2.3 Data serialization formats

Data serialization (or sometimes called object marshalling) describes the process of presenting data from an application in a textual or binary representation. It is used to expose data via input / output to other processes, or to store this information persistently for future instances of the same process. This data is then either represented as a string (textual serialization) or as a byte array (binary serialization) [30] [31].

Inside the program, the data is usually represented using an object in Object-Oriented Programming (OOP) or for example a structure in C. In software engineering design patterns, the Data Transfer Object (DTO) pattern is used to express an object that describes data that can be transferred for example via network or other means in a serialized format [32] [33].

The selection of the presented serialization formats in this section (XML, JSON, CBOR and Protocol Buffers) is based on the following criteria: XML and JSON are chosen based on a very high popularity, in specific implementations as well as in scientific research. They both are the de-facto standard of an era (XML first in the 2000s, switching to JSON in recent times). This marks high interoperability, good availability of libraries on every programming language and lower effort needed by an application developer to adopt this technique. The other two are mainly selected because they show promising results in producing compact results that lead to lower overhead, while still being popular for applications with this specific goal. First, XML is introduced.

2.3.1 XML

The eXtensible Markup Language (XML) standard has been proposed in 1998 and accepted by the W3C. It evolved from the Standard Generalized Markup Language (SGML) with the goal to make it more suitable for the internet. The serialized representation of a data object is called an ”XML document” in XML. It is a textual representation of the data, that is quite extensive in its description of the data structure and therefore also easy to read for humans [34].

The structure of XML consists of elements, that can be empty, contain other elements or contain text data. The document is formed of a root element and optional an XML declaration or the document type, which specifies an XML document type definition (DTD) for the validation of the document. Additionally, the elements may have attributes assigned to it, with the main difference that an element can have an attribute of a specific type either once or not at all, while child elements may occur arbitrarily often, as long as no validity constraint restricts it [34]. After the initial definition of XML with DTD for document validation, newer definitions for checking XML document validity were created, most notably XML schema and RELAX NG. They both focus on allowing additional data validity constraints, i.e. what may be used as text data inside and element and more complex restrcitions of when elements may be used inside another element [35]. The following listing shows an XML example using DTD:

Listing 2.1: XML example with embedded DTD

```xml
<?xml version="1.0"?>
<!DOCTYPE person [
        <!ELEMENT person (name,age)>
        <!ELEMENT name (#PCDATA)>
        <!ELEMENT age (#PCDATA)>
]>
<person>
        <name>John Doe</name>
        <age>18</age>
</person>
```

The information shown in the ”<!DOCTYPE” part can be extracted to a special DTD file, which is then only referenced by the document type tag. This is specifically useful for larger documents or if many documents of the same type exist, which is very common in applications. Also the document type is optional, as also any restrictions on text values are up to the applications using the document or an extra schema definition [35]. That allows to use weak typing in combination with XML, although many developers use strict schemas and definitions, especially in a context like SOAP (see 2.4.7).

Espcially together with SOAP and other technologies which were commonly used in the eraly 2000s, XML gained high popularity and was the lingua franca of many computer systems. Even today, it is very common for human-edited configuration or specification files and it is still used in many applications. In recent years, especially with modern web technologies like JavaScript, application developers started to prefer JSON for machine to machine communication (and especially browser to web server communication). This serialization format is now presented in detail.

2.3.2 JSON

The JavaScript Object Notation (JSON) is a data interchange format originally specified by Douglas Crockford [36]. It is derived from ECMAScript [37] and was proposed to the IETF as RFC7159 [38] in March 2014. The format is text-based, human-readable, language-independent and designed as a lightweight alternative to XML for data transmission on the web.

JSON supports four primitive and two structured types, whereas the whole serialized JSON text should be one of the structured types. The supported types are [38]:

- **null** [primitive]: This is a null reference as known from various programming languages and must be lower case without enclosing quotation marks.
- **false and true** [primitive]: Boolean values which must be lower case without enclosing quotation marks.
- **Numbers** [primitive]: Any numerical value represented using decimal digits, but infinity or NaN values are not permitted (they are usally translated to null). Fractions are separated with a single point or exponential notation and negative values are prefixed by a minus sign (e.g. 1.05, 2e-5 or -1.3e+8). Numbers have no minimum or maximum boundaries and can be of arbitrary precision, but the values might be rounded during deserialization.
- **Strings** [primitive]: Any unicode string, enclosed by quotation marks and using C-style escaping for all control characters, quotation marks and the reverse solidus. Other unicode characters may be escaped as well.
- **Objects** [structured]: A pair of curly brackets surrounding zero or more name/-value pairs (also referred to as members) is an object. Multiple members are comma-separated and the names should be unique within a single object (it is not specified how an implementation behaves on non-unique member names). The name must be a string, though many implementations allow to omit the enclosing quotation marks if the name does not contain any whitespaces. Each name is then followed by a : and a value, which can be of any JSON type (of course, the members of an object may have values of different types).

- **Arrays** [structured]: Zero or more elements surrounded by square brackets form an array, with a comma separating the elements. The elements may be of any JSON type and even multiple elements in the same array may be of different types. **Examples**: [] (empty array), ["foo","bar"] (array containing two string elements "foo" and "bar"), [null, "foo", 2] (mixed array)

Whitespace characters may be placed around separators, between elements or around any of the brackets. They are completely ignored and therefore it is possible to print arbitrarily complex JSON objects on a single line [38]. For improved readability, objects and arrays are often printed with identation and a single member or element on each line, like in this example:

Listing 2.2: JSON example

```json
{
	"name": "John Doe",
	"age": 18,
	"isMale": true,
	"location": {
		"latitude": 13.37,
		"longitude": -4.2e+1
	},
	"driverLicense": null,
	"siblings": [
		"Jane Doe"
	]
}
```

A big advantage of JSON is its schema-free design, which is achieved through the weak typing of the data fields, which works exactly like in JavaScript. Validation of the input values is usually either done during the mapping, i.e. if a JSON field is mapped in Java to an object's field of integer type, the Jackson parser [39] will fail to map if the JSON's value can't be translated to an integer. It could be a string stating ”7” or an integer 7 though, this case would be automatically handled by the weak-typed nature. This allows for an easy extensibility of the format and input validation is solely done at the reader's side, which on the other hand accepts any JSON object, that can be translated to valid input data.

However, although JSON does not need a schema, there are extensions like JSON schema that allow to define the input validation rules in JSON, if a stricter parsing at the expense of less portability is intended [40]. This can also be very useful if multiple implementations are required to do the same validations.

After the JSON format gained high popularity, derivatives of it were created, mostly to add more data types like binary chunks or to achieve a more compact binary storage of the same information. One of these is the Concise Binary Object Representation (CBOR) which is presented in the next section.

2.3.3 CBOR

The Concise Binary Object Representation (CBOR) is described as an evolution from JSON (see 2.3.2). It requires that all JSON documents can be represented as CBOR and this CBOR data can be mapped back to the same JSON without extra information. On the other hand, CBOR supports some extensions (mainly a primitive type for binary blobs) which JSON doesn't support natively [41].

The design of CBOR integrates other goals, mainly from the field of embedded or resource constrained devices. Another important aspect is to foster a very high interoperability, also with future extensions. To cite [41], "the format is designed for decades of use [...] [and] must be able to be extended in the future by later IETF standards."

Specifically, CBOR adopts the weak-typing of JavaScript to allow named fields to map to one of the primitive types, without making fixed assumptions about the binary representation of the value. Parsers of CBOR can read the structure and the values out of the binary blob without the need of a schema or a definition file. The extensibility might lead to reading an unknown value because of not having the implementation for a new type of binary representation, but the rest of the object can still be read dropping just this single field. This leads to having the same possibilites as with JSON, being able to just read an object and then map it to the expected datastructure, ignoring potential new fields or slight type changes, but with even some additional features like mainly the embedding of binary data [41].

For resource constrained applications, CBOR on the one hand aims to have very compact code with a low memory requirement and low code complexity. On the other hand, it also aims to make a compact binary representation with small object sizes, but this goal is declared as secondary to the first one. One big advantage that derives from this compared to JSON is, that parsers don't rely on reading one character after the other, but larger chunks of bytes can be processed at once depending on the binary field representation [41].

Like JSON, CBOR is also schema-free although JSON schema [40] can be used with CBOR as well, but like with JSON it only adds little value. Most other binary formats rely on a definition of the binary data layout, making their usage more cumbersome than CBOR's. A popular example of one of these formats are Google's Protocol Buffers, which are now described in more detail.

2.3.4 Protocol Buffers

Protocol Buffers (often just called "Protobuf") were developed by Google internally for organizing binary data exchange between servers and then published as an Open Source project in 2008. They evolved from the need of having a compact and efficiently parseable binary representation of data, while maintaining forward and backward compatibility through an interface description language (IDL, see more in 2.6), which is used for the definition of the binary structure. While it maintains compatibility if fields are added or removed in a future version, it is not possible to interpret the binary data without knowing the structure [42].

The definition of a protocol buffers message is done with a proto file which contains a definition of required, optional or repeated fields, assigning types and numerical tags to them. These tags must be unique throughout all versions of this message, so the field will never be confused with another field. Required fields will always be required, making it harder to adopt the message to later needs, which is why it is often advisable to use optional in combination with validation logic inside the application if needed. Repeated allows the field to appear any number of times, including zero and preserving order, making it technically a list [43] [44]. An example proto file looks like this:

Listing 2.3: Proto file example

```
message Person {
	required string name = 1;
	optional int32 age = 2;
	repeated Person children = 3;
}
```

The types are similar to those in JSON (see 2.3.2), but with a C type to map the value to, like int32 being an unsigned integer with 32 bit and sint32 the signed variant. For other languages like Java the closest matching type is used. The types are interchangeable as long as they have the same base type like numbers, floating point numbers, strings etc., so it is possible to interchange int32 with sint64 if needed, but not possible to make a string field an int32 in a later version [44]. Compared to CBOR, this leads to a slightly lesser interoperability.

Which worsens the interoperability is the need of the proto schema definition file. An application may use an extended (i.e. newer) schema, but an application not having the schema or using an incompatible schema file is not able to properly deserialize the data [44]. On the other hand, protobuf promises a very high compactness of data, making it suitable for the VSL middleware.

These four analyzed serialization formats are now assessed and compared.

2.3.5 Assessment of data serialization formats

Various criteria is eligible for the assessment of the aforementioned serialization formats. The most obvious is the separation in text and binary serialization formats and looking at the standardization, inception dates and prevalence of the formats. All of the four presented formats are widespread enough to offer various libraries for all programming languages that are used during the implementation, so availability of libraries is not very distinctive. So for these formats, the goal of interoperability is met and a detailed rating of the interoperability is not very useful for the decision.

An interesting aspect is the need and availability of schemas or definition files, which can be useful for code generation as discussed in 2.6. The available and commonly used schemas are referenced here and a more detailed discussion follows in 2.6.

Related especially to the need for a schema, but generally very important is the weak parsing of the input data or the possibility to adapt data representation to future needs without breaking the compatibility. The benefits and risks of weak typing in programming languages is still somewhat disputed and the most often used programming languages contain some weak and some strict typed languages, leaving good arguments for both [45]. For data serialization formats however, weak typing and weak parsing has strong advantages over strict parsing and restrictive schema application. Using very weak parsing, the interoperability of reading the data is enhanced, especially if types or data structures are altered over time. These examples can illustrate this:

- Assume a field used to have a single number as a value, but future implementations require the option to accept more than one number as a value there. So the type is changed to an array of numbers, while some legacy applications still send only a single number. Weak implementations can read the single number values as a single element array and accept multi element arrays from modernized applications as well.
- A date field used to be represented as a timestamp for compactness reasons. Now it turns out, the implementation needs to support ancient dates like February 23rd, 1786. Date strings conforming to ISO 8601 should be used now but numeric timestamps were used before. For example JSON parsers can parse both formats into suitable date representations of the corresponding programming language. Even if it does not support automatic conversion between these formats, a simple ”if” after the deserialization could handle both cases properly.
- Adding more fields to an object is a very common extension, but this case is more tricky: Assuming a person object as in some of the examples before contained a required field ”Full name”. For improving the separation between multiple first or last names, it was split into ”First name” and ”Last name”, both being mandatory. In some formats or schema languages, it would be hard to express that it is either mandatory to have the full name but not first and last name or to have last and first name but no full name specified. Weak formats would accept everything first and the application logic can either specify this requirement or convert the older format into the newer, by e.g. splitting the full name at the last space.

Schema definitions are often defined too strict in order to suit for data validation as well. This leads to many problems with interoperability, which is why most experienced software engineers advise against it, e.g. also for protobuf definitions [44]. The input validation should occur after the deserialization and potential conversion of deprecated data formats, but of course before actual usage of the data. This validation should be restricted to the actually used data fields, as unused fields might contain future features and ignored parts of the data don't pose any harm and can be disposed.

A downside of weak parsing can be confusing effects for unaware users, e.g. if "1.0e3" can be interpreted as integer (1000) or "0x3e8" being the same integer value. This is only relevant when user input is checked, which should be done with immediate feedback to the user and more strict to avoid confusion. In case of computer to computer communication, an attacker could use these values but later validation (like it has to be an integer value between 0 and 1000) still applies regardless of how the data was represented, so with reasonable validation after deserialization this does not pose harm.

All of the four discussed data serialization formats allow for weak parsing to some extend, but with differences in detail. Protobuf for example can only handle two of the three examples above if best practices for proto file design are followed (like never using required [44]), but fails at the date/timestamp example. JSON would be able to handle the date/timestamp case as well, even with allowing the old implementation that expected a timestamp to parse the ISO 8601 date if it is in a range that can be converted to a timestamp. XML's weakness highly depends on the best practices used, e.g. many software engineers advise to never use XML attributes because of their bad extensibility [46]. On the other hand, XML contains just text values, leaving the value interpretation very weak.

Schemas with included validation can on the other hand be used to port validation rules to different programming languages and ensure consistent validation. That is a case where validation is unified in a multi language environment (otherwise just the validation code could be shared) but at the expense of reduced extensibility. In case of the Virtual State Layer (VSL) middleware, validation is done inside the KOR (see 2.1.2.3) and services can rely on the knowledge agent to provide valid data, as it validates everything it processes. This makes using a very weak serialization format advisable.

The criteria for the assessment and a structured comparison of the four candidates is done in table 2.2 to summarize the results. Further evaluation of the formats, especially on compactness and performance is done in the evaluation section 6.1. The next section analyses the network protocols that can be used to transport the serialized data.

Table 2.2: Comparison of serialization formats

Format	**XML**	**JSON**	**CBOR**	**Protocol Buffers**
Encoding	various text encodings [34]	UTF-8 or UTF-16/32 [38]	binary [41]	binary [44]
Standards	W3C recommendation [34]	RFC 4627 [47], RFC 7159 (standard) [38]	RFC 7049 [41]	No standard, developed by Google [44]
Published	1996 (draft), 1998 (1.0) [34]	2006 [47], 2014 (standard) [38]	2013 [41]	2001 (internal only), 2008 [44]
Media type	application/xml or text/xml [48]	application/json [38]	application/cbor [41]	not official, application/octet-stream [49]
Intended purpose, design goals	high standardization, reliable data exchange [34]	high portability, simplicity [36]	binary JSON variant, compactness [41]	efficiency, extensibility [42]
Schemas and definition files	DTD, XML schema, others	schema-free, JSON schema inofficial[1]	schema-free, could adopt JSON schema	proto file (always required)
Self-describing[2]	Yes	Yes	Yes	No
Type weakness	No types, everything is just text	Very weak	weak[3]	representation weak, but basetype strict
Human readability	Very good, IDE support for DTDs	Good	Only when converted to JSON	No[4]

1. JSON is designed without versions or schema on purpose [36], but some people disagreeing with this design created JSON schema as an extension [40]
2. Being parseable and somewhat interpretable without further structure or schema definition
3. CBOR is rated less weak then JSON because the different binary representations might be extended and some parsers might be unable to read the new representations
4. Tools could read the protobuf data if the proto file is loaded, and convert it into a custom human readable view; this cumbersome but possible process is not rated as ”human-readable”

2.4 Suitable network protocols

After the data has been serialized, suiteable network protocols are needed to transfer the data to other applications using the architectural design approaches which are presented in 2.2. Applied to the Virtual State Layer middleware (VSL, see 2.1.2.1), the protocol is needed for the communication of services to the knowledge agents, but also for the communication between kowledge agents.

Security considerations (cf. requirement #6 from table 2.1) are very crucial in this section, especially encryption and authentication. That is why the most common security protocol, the Transport Layer Security (TLS), also known as Secure Socket Layer (SSL), is presented first. In this subsection also some special aspects like certificate based authentication or datagram TLS (DTLS) are described in detail, as these are important for the other protocols, which mostly use TLS to secure the communication.

The next subsection focuses on the Hypertext transfer protocol (HTTP), which is on the one hand also used as a base for other high level protocols like XML-RPC, but on the other hand it can also be directly used especially for RESTful architectures. The new HTTP/2 version is compared to its predecessor and the main improvements are highlighted.

Afterwards, the Constrained Application Protocol (CoAP) is presented, which is similar to HTTP but based on datagram communication instead of streams, also utilizing other methods to reduce the protocol overhead. Additionally, the WebSocket protocol is introduced, a technology that builds up on HTTP to provide a bidirectional message based communication similar to plain sockets.

Then, two Remote Procedure Call (RPC) style protocols are analyzed. First the simple XML-RPC protocol and then SOAP, a very common RPC protocol for service oriented applications.

Now the SSL/TLS protocol family for encrpyted and authenticated network communication is indroduced in detail.

2.4.1 SSL/TLS

The Transport Layer Security (TLS) protocol is a very commonly used protocol to add cryptographic security to a network connection on the transport layer, i.e. underneath the actual protocol implementation. It originates from the Secure Socket Layer (SSL) protocol, which was originally invented at Netscape and patented as US patent number 5,657,390 [50]. The current version of TLS is 1.2, which is standardized in RFC 5246 [51]. The whole family of protocols starting with SSL 1.0 in 1994 to TLS 1.2 and further to future version is often referenced as SSL/TLS [52].

The functionality of TLS is provided by two main components: the TLS Record Protocol which serves as a low level channel with encryption and message integrity throughout a TLS session and the TLS Handshake Protocol which performs peer authentication and negotiates the used features and encryption methods. The handshake can also negotiate additional properties like the protocol used on top of the TLS connection. TLS can be used as a transport protocol for any application layer protocol [51].

Due to the negotiation of the actually used TLS features and the high age of some protocol versions and features, it is very important that both peers require certain features like strong encryption or authentication during the handshake, as otherwise the intended security level is not reached [52]. The different approaches for authentication and especially the X.509 certificates are described in more detail in 2.4.1.2. During the handshake, a symmetric key is exchanged and then used for encryption and session integrity, except if encryption is disabled by negotiation of the NULL cipher [51]. Using proper requirements during the handshake, integrity and secrecy of the data transported via TLS is thereby provided [52]. In 2.4.1.1, some additional negotiable session parameters of TLS are introduced.

TLS requires a reliable stream-based communication in which data reordering and losses of application payload must not occur [53]. So usually TLS is based on a TCP or similarly reliable connection. In order to use TLS features in datagram based communication for example via UDP, the Datagram Transport Layer Security (DTLS) protocol provides very similar functionality to TLS while preserving the datagram communication style [53]. This means specifically, that each datagram message can be received individually even if previous messages got lost or arrive at a later time (reordering), but DTLS still provides a reliable handshake mechanism and an optional replay protection [53]. Most of the features work excactly the same from a high level perspective, which is why the further parts do not specifically distinguish between these two protocols.

Now some additional TLS features which can be negotiated during the handshake are described in more detail.

2.4.1.1 Additional negotiable TLS features

While the TLS handshake mainly serves the authentication, key exchange and negotiation of the used cryptographic algorithms, it can additionally negotiate other features. Some of these add additional functionality to TLS while others are even needed by certain application layer protocols [51].

One of these features is session resumption using TLS session tickets. With these tickets, a client which already negotiated a TLS session with a server can reuse the exchanged keys in a future session with the same server. This can be used to speed up multiple connections to the same server by doing a shorter handshake for session resumption and to require less entropy due to less key exchanges [54]. Especially scenarios with lots of reconnections or embedded devices can profit from this extension.

Some features also exist to overcome shortcomings of application layer protocols, like the TLS compression [55] to add compression to TLS in case of uncompressed but highly redundant application layer data or the heartbeat extension [56] to support active keep alive on the TLS layer if the application layer does not support handling of NAT timeouts. While these features on the one hand add valuable functionality to TLS for special applications, it can on the other hand lead to security issues that arise from improper usage or implementation. In case of the compression, the general issue if information leakage through the packet size even on encrypted channels is known for a long time [57] and proved severly bad in case of the TLS encryption feature [58]. The heartbeat extension gained most attention for an implementation error in the OpenSSL library, which lead to many vulnerable servers which did not even actively use the extension [59]. These examples illustrate that these extension can be useful in special cases but also inhibit a risk of security issues which should be considered carefully, especially as these features could also be provided by the application layer protocol.

Another additional feature of the TLS handshake is the optional negotiation of the application layer protocol which will be used on top of TLS. This feature is called Application Layer Protocol Negotiation (ALPN). The major current application of this feature is the upgrade to HTTP/2 via ALPN [60]. In this case, if ALPN is not supported by one of the peers or one of the peers only announces earlier protocol versions, HTTP/2 is not used. HTTP/2 is described in more detail in 2.4.3.

The next section describes the authentication methods during the TLS handshake in more detail.

2.4.1.2 Authentication and X.509 certificates

TLS also supports many different authentication schemes, the main RFC names anonymous authentication, server only and client and server authentication using X.509 certificates with asymmetric cryptography [51]. Additional extensions added features like pre-shared key (PSK) authentication [61] or for example Kerberos authentication [62].

In case of anonymous handshaking or if additional security is wanted, the handshake also includes a Diffie Hellman (or similar) key exchange. This feature is called ”perfect forward secrecy” to emphasize that the session key can not be derived from a leak of the private keys. On the contrary, if no key exchange is applied, the key exchange happens with the client sending a part of the key encrypted using the server’s public key, which allows to decrypt the session if the server’s private key leaks [51].

The certificates, if used, are usually X.509v3 certificates unless explicitly negotiated otherwise [51]. This format allows to specify many details on the authenticated entity, apart from the subject identifier with the common name also alternative names, optional attributes, validity timespan of the certificate and much more. One of the very big advantages is, that the certificate itself can be signed by an issuer, whose X.509 certificate is also delivered by the server or already known to the client. This can be used to validate servers or clients without the need of a list of all valid certificates, as only by knowing the public key of the so called "certificate authority" (CA), which is also part of the certificate, all certificates issued from this CA can be verified by their certificate signature. Additionally, the CA can also distribute certificate revocation lists (CRL) to revoke certificates that should no longer be trusted. This allows very flexible offline verification (i.e. without asking a server for the validity of the information) of even a large number of entities [63].

Unless explicitly negotiated otherwise, the TLS server sends its server certificate and relevant issuer certificates to the client. The client validates the information and does usually not provide a certificate, which is the server only authentication. But the server can also include a CA certificate to challenge the client to authenticate with a certificate from this particular issuer. The client provides a certficate of this CA if possible and the server can decide if a client without a valid certificate may procedd or not. If this client certificate mechanism is used, server and client authenticate against each other and a very high level of integrity of the authentication information is granted [51].

The next section covers the HTTP protocol, which is also commonly used with SSL/TLS.

2.4.2 HTTP

The Hypertext Transfer Protocol (HTTP) was originally developed for the transfer of websites to the browser and is still used for this purpose but additionally for a lot more, like as a transport protocol for other protocols like XML-RPC (see 2.4.6) or SOAP (see 2.4.7) or as a communication protocol for RESTful applications (see 2.2.5 on REST) [64].

HTTP has mainly three versions, 1.0, 1.1 and 2. HTTP 1.0 and 1.1 are compatible to each other in the way, that HTTP 1.0 clients receiving a HTTP 1.1 answer could read relevant information from it and vice versa. The HTTP 1.1 standard was revised and extended in 2014 to the six RFCs 7230-7235 [64]. The new version 2 uses the same high level semantic of HTTP 1.1 but with a very different low level transport mainly to enhance the performance of HTTP. It is standardized in RFC 7540 [65]. The general semantic of HTTP is described first and then in section 2.4.3 the specific changes of HTTP/2 are introduced. An additional extension to HTTP, the WebSocket protocol, which establishes a messaging channel through an HTTP request, is presented in 2.4.5.

The general semantic of HTTP is a client sending a request to the server, which responds with a reply to the client. The request is stating an HTTP method (like GET), a uniform resource identifier (URI, like /index.html) and the HTTP version (like HTTP/1.1). After-

Table 2.3: Common HTTP status codes as described in [2]

#	**Name**	**Description**
101	Switching Protocols	Switching to a new protocol like WebSocket (see 2.4.5)
200	OK	Request is OK with body
201	Created	A new resource was created
202	Accepted	Request accepted but is processed in the background
204	No Content	Request is OK without body
301	Moved Permanently	The resource now resides on another URI
400	Bad Request	Request header or syntax is invalid
401	Unauthorized	HTTP authentication required (see 2.4.2.3)
403	Forbidden	Access to the resource is denied
404	Not Found	Resource does not exist or the server does not disclose its existence
405	Method Not Allowed	The used HTTP method (see 2.4.2.2) is not allowed
406	Not Acceptable	Content negotiation (see 2.4.2.1) failed, no accepted format is available
415	Unsupported Media Type	The client proposed a request body with an unsupported Content-Type or Content-Encoding
500	Internal Server Error	Error in the server's request processing

wards, the request can include HTTP headers and an HTTP body. The response states the HTTP version, a status code (like 404) and a textual reason phrase (like Not found). It can also include HTTP headers and a response HTTP body. While strictly speaking, the headers are not mandatory, it is very usual that both include headers which specify the body or further details of the method. The inclusion of a body mostly depends on the executed HTTP method and the response status code. Each of the methods is executed on its own and therefore stateless, following the principles of the REST architecture (see 2.2.5) [65].

The next sections explain more details on central aspects of the HTTP protocol. Section 2.4.2.1 explains HTTP's content negotiation feature, section 2.4.2.2 describes the purpose and proper usage of the different HTTP methods and section 2.4.2.3 describes important HTTP headers which are also used in the protocol design (see 4.3). Table 2.3 shows a list of common HTTP status codes and their meaning.

2.4.2.1 Content negotiation

One of the features of the HTTP protocol which I highlight for the later usage in the design is the content negotiation feature of HTTP, namely the Accept-* and Content-* request and response headers. The purpose of these headers is to allow the client to specify and weight which data format (specifically: MIME type, charset, encoding and language) of the requested resource is preferred and which formats are also acceptable to the client. The server then communicates the selection which it has made using the headers Content-Type, Content-Encoding and Content-Language. These headers are usually also sent, if no accept headers were used in the request, to allow the client to identify the type of the content (HTTP body) [2].

An example how this actually looks like: if a browser sends the request below (listing 2.4), it specifies very detailed what it accepts using the ";q=weight" parameters to specifiy the weighting of preferences (1.0 = highest, >0 = lowest, 0 = not acceptable). The asterisk * is a wildcard for any, but for example "text/plain, text/*" would give text/plain precedence although text/* has the same weighting. The default weighting 1.0 can be ommited [2].

Listing 2.4: HTTP request with Accept headers

```
GET / HTTP/1.1
Accept: text/plain;q=0.5, text/html, text/*;q=0.5, */*;q=0.1
Accept-Charset: utf-8, *;q=0.8
Accept-Encoding: gzip, identity;q=0.5, *;q=0
Accept-Language: en-gb, en;q=0.8
```

A possible response from a server, which has for example an index.txt and an index.csv file for the path /, would choose index.txt and recode it to UTF-8 if supported. Lets assume the server only supports deflate compression, it would have to deliver the file unencoded (identity is not encoding, if *;q=0 is missing the server would actually be allowed to use any encoding, which is a special rule for Accept-Encoding) [2]. If the file has a language and multiple version exist, the British english version would be preferred over any english version which is still preferred over any other language. Assuming the server has an english version, but only US english, the response would look like in listing 2.5. Note that there is no Content header but only Content-Type, which also includes the charset if and only if the MIME type is not binary, and that Content-Encoding is left out because no encoding was applied.

Listing 2.5: HTTP response to listing 2.4

```
HTTP/1.1 200 OK
Content-Type: text/plain;charset=utf-8
Content-Language: en-US
[...]
```

This content negotiation feature is very important for clients which only support a subset of the HTTP functionality, for example because they are implemented on an embedded device, as they can demand specific formats from the server. It is also an important feature for resources which can be represented using multiple serialization formats (see 2.3), as the format can be negotiated using these headers.

2.4.2.2 HTTP methods

The HTTP protocols allows different methods for the interaction with a resource and each of the methods has its own definition how to interact with the resource. The methods which are commonly used are listed with their properties here:

- **GET**: The GET method is used to retrieve a resource from the server. The request should not contain a body and the execution of a GET request must not change the resource or other resources. This way, the operation is idempotent unless another method was executed in between, which changed the resource. Usually the response to a GET request is cached for later reuse, unless the Cache-Control header (see 2.4.2.3) denies it. There is also the **HEAD** method as a variant of GET which does only request response headers without the actual content. Otherwise the HEAD method works the same [2].
- **PUT**: The PUT method is used to replace an existing resource with a new version or to create a new resource on the server. In the request body, the resource must be included and properly specified using the Content headers (see 2.4.2.1 on the content negotiation). The server can then respond with 201 Created if the resource was newly created or 200 OK/204 No Content in case of replacing an existing resource. Optionally the server can include information on the resource in the response, but it must not redirect to another URI where the resource was create instead of the PUT URI (example: PUT /list/add must not redirect to /list/element1 where it was created; POST must be used instead for this kind of operation). Normally after putting a resource, a subsequent GET will return the same data, but the server might recode or transform the resource for example to a common media type. What usually can be expected is that multiple PUTs with the same data to the same resource will not alter the resource further, i.e. that PUT also behaves idempotent [2].
- **POST**: The more generic POST method can be used for many interactions with a resource or for creation of a resource in a server-controlled location. It is often also used for RPC style interactions (see 2.2.4 on RPC), using the request and response bodies for the invocation and result payloads. POST requests can be used quite freely, but the REST architecture demands it to be only used if the other methods cannot be used to represent the operation. If a POST request creates a resource, it should also answer with 201 Created and specify the new resource's location in the Location header. A POST request may only be cached if it specifies how to be cached, and even that is uncommon and often not implemented [2].

- **DELETE**: This operation deletes the resource if it exists. It may respond with 200 OK and information about the deletion status in the body, 202 Accepted if the deletion will be performed or 204 No content if the deletion is completed. By its nature, a DELETE request cannot be cached, but caches can delete cached information if they encounter a DELETE request [2].
- **OPTIONS**: The OPTIONS request is used to determine possible interactions with a resource and to detect server capabilities. Usually the request and respons only contain HTTP headers, especially also containing special headers like Allow which are used specifically for the OPTIONS method (see 2.4.2.3), it is however allowed to include bodies in both, request and response. There is also a special OPTIONS request on the URI "*" which is used to generally determine server capabilities without a specific resource. The answer however can differ from resource to resource, which is why strictly speaking a request to every interacted resource would be required, but using OPTIONS at all is optional, so it varies between clients how much it is actually used. It is practically useful to determine the possibilities of using PUT, POST or DELETE requests (GET can normally be expected to work or just gives an error and it is as efficient to just probe GET instead of doing OPTIONS beforehand). An OPTIONS request is responded to using 200 OK and a Content-Length header with the value 0, unless there is an actual body content. OPTIONS requests must not change or interact with the resource at all and they are also not cachable [2].

Other methods exist for special purposes and they are listed in [2]. Now those of the HTTP headers, which are important for many applications, are introduced in more detail.

2.4.2.3 HTTP headers

RFC 7231 [2] specifies in detail which standard headers HTTP clients and servers should understand and there is the possibility to add custom headers or for advanced techniques like the WebSocket (see 2.4.5) to specify headers for their specific purposes. This section introduces some of the HTTP headers, which were not discussed in the previous parts but are relevant for many applications which use HTTP or for my protocol design.

One important aspect is dealing with caching and having control over when a cache answers on behalf of the server to the client, without asking the server for updated information. On the one hand, specific resources which are highly volatile need to be excluded from the caching or specific rules for the caching need to be applied, for example to prevent the exposure of an access protected resource to unauthorized clients through the cache. On the other hand, very static resources, especially large resources, can be cached very well and might not even contain any private information, allowing for the cache to be even shared amongst different users [64] [66].

The header which configures how a resource may be cached is the ”Cache-Control” header, which contains a list of caching directives, which control how the resource may be cached. Important values are: no-cache demands to not respond with a cached value, no-store demands to never store the value (privacy/secrecy rule), private tells to never share the resource with another client and public declares the exact opposite. There are more directives on validation of cached resources or maximal ages, but these are only relevant if fine-grained caching is intended. There are more headers associated to caching like the ”Expires” header which sets an absolute date after which a resource must not be delivered from a cache or the ”Pragma” header which an be used with the value ”no-cache” to prevent caching on HTTP 1.0 caches [66].

All of this is explained in high detail in RFC 7234 [66], but for most private RESTful APIs the most important information is to deny caching with ”no-cache, no-store” in the ”Cache-Control” header and to maybe include the HTTP 1.0 ”Pragma: no-cache”.

Another important HTTP header is the ”Authorization” header which allows to pass identity information of the client to the server, which can be used for authentication and authorization. It contains an authorization type (like basic or something else) and a usually base64 encoded authorization, which depends on the type (actually 68 tokens are allowed, but base64 is commonly used). The authorization can either be just presented by the client or challenged by the server by providing a 401 error with an WWW-Authenticate header which specifies the challenge. The whole process is documented in detail in RFC 7235 [67].

There are also special headers mostly used with the OPTIONS method (see 2.4.2.2) or specific kinds of requests. One of these is the ”Allow” header, which lists the HTTP methods which are allowed with the specific resource or the whole server in case of an OPTIONS * request. The Allow header is also included in the HTTP error 405 method not allowed [2].

Additionally there is a set of headers for so called cross-origin requests which are executed inside a browser usually by JavaScript. The ”Cross-Origin Resource Sharing” (CORS) defines in depth how these requests are secured using additional headers and a ”preflight” OPTIONS request before executing the actual request. A cross origin request is a request from a website to another website with a different URL, so for example HTML and JavaScript loaded from example.com and then a request executed to other.com [68].

There are several headers involved in the preflight request, first of all the client specifies the original site without resource path as ”Origin” and the method that will be executed using ”Access-Control-Request-Method”. Additional headers which are explicitly added by the client (e.g. JavaScript, the browser will set more headers implicitly anyway) are named in the ”Access-Control-Request-Headers” list (just naming the headers without values). The server answers with ”Access-Control-Allow-Origin” which specifies the allowed origin (* for all or one specific origin - if the server supports multiple, only the one provided by the request), ”Access-Control-Max-Age” sets the seconds for how long the preflight may be cached for subsequent identical requests and ”Access-Control-Allow-Methods” with a list of methods which are allowed. The methods list is actually the same as in the ”Allow” header, but CORS requests are required by specification to ignore the ”Allow” header and to only use the ”Access-Control-Allow-Methods” header. The server also uses additional headers to further specify allowed HTTP headers using ”Access-Control-Allow-Headers”, if credentials may be included in the request using ”Access-Control-Allow-Credentials” with the only value ”true” and ”Access-Control-Expose-Headers” for the exposition of response headers to the JavaScript, which otherwise would not be allowed to access the response headers [68].

The actual request after the preflight still has to include some of the headers, namely ”Origin” from the client and from the server ”Access-Control-Allow-Origin” and ”Access-Control-Allow-Credentials” if credentials are allowed. The other headers are only used in the preflight [68]. This proper handling of CORS requests is important for modern web applications which connect to different backend APIs and in the specific scenario of the Virtual State Layer middleware (see 2.1.2.1) it is required to allow web applications to access a knowledge agent.

The next section introduces the changes of HTTP/2 compared to HTTP/1.1.

2.4.3 HTTP/2

The HTTP/2 protocol version was designed to overcome known performance issues of HTTP/1.1. These are mainly the lack of asynchronous operations (like in requirement #8 of table 2.1), leading to the head of line blocking problem and the excessive overhead of some verbose HTTP headers, which on the one hand increases the overhead but also leads to worse TCP performance [65].

There are multiple approaches on how to deal with many operations and especially asynchronous operations in HTTP/1.1, but all of them have disadvantages.

First of all, it is possible to use multiple concurrent (TCP or TLS) connections to the server, so that each connection can execute one request and depending on the number of connections, multiple requests can be executed concurrently. There are multiple issues with this approach, first of all multiple connections must be established leading to additional handshakes, more resource consumption and more mangement overhead. In case of TLS connections (see 2.4.1), the handshake is even more complex, altough mechanisms like session tickets (see 2.4.1.1) can reduce the TLS handshake complexity if many connections to the same server are established. A second big issue with this approach is less efficient TCP congestion, as the parallel connections each perform individual congestion control unless there is specific optimization for this use case in the operating system's TCP stack. Especially each connection usually does a slow start, which leads to high inefficiency in fast networks [65] [69].

Another solution is to use HTTP keep-alive, which was introduced with HTTP 1.1 and allows to reuse the connection of a previous request for further requests. This avoids doing a handshake for each of the requests, but it suffers the head of line blocking problem, i.e. a request (or response) can only be transmitted if the transmission of the previous request (or response) is completed, which does not really allow asynchronous operations but can speed up sequential operations a lot [65].

Most advanced HTTP 1.1 clients like browsers implement a mix of these two by using a keep-alive connection pool which can increase the number of connections depending on how many parallel operations are needed. The management of the pool is however very expensive and still suffers from the aforementioned issues.

HTTP/2 solves all these issues by using one connection with binary message frames which support multiplexing, so every request and response is split into small frames and they are sent using the same connection. The frames are associated to streams, in which the order of the individual frames matter while frames of multiple streams can be mixed in any order. This allows processing of HTTP/2 streams similar to concurrent HTTP 1.1 connections. Using this method, TCP's congestion management is utilized better as a single connection can ramp up to the full network performance without being disturbed by parallel connections to the same server. Also asynchronous operations are possible as parallel requests form streams, which can run in parallel, efficiently transmitting multiple requests at once. This is especially useful if one very large request is performed concurrently to a batch of small requests [65].

The issue of the header overhead is solved by header compression using binary headers which are also packed more efficiently. Still HTTP/2 supports the same headers as HTTP 1.1, especially also arbitrary custom or extension headers [65].

HTTP/2 connections are usually established using TLS with ALPN (see 2.4.1.1)), requiring TLS 1.2 by design. An alternative establishment via the "Upgrade" HTTP header (more on protocol upgrade in 2.4.5) or by knowing in advance, that the server supports HTTP/2, for example via SVC DNS records. The establishment using TLS ALPN is the preferred way of using HTTP/2 [65].

Another noteworthy extension of HTTP/2 is the server push functionality, which allows to push updates to a requested resource to clients whenever they occur without needing another request for this resource. Server push works as follows: if the client sent a request which is cachable without request body (e.g. a simple GET), the server can make push promises on resources it will push. These resources do not need to be requested by the client, as the server keeps its promise and will push these resources. This can be used for example to push images, which are referenced in an HTML file, directly after the HTML file which contained the push promises. The server's promise also includes pushing updates to the resource once it changes, easing the caching of the resources and avoiding the need of an auto refresh from the client. The whole server push functionality can of course also be disabled by the client if it does not want to use it [65].

The next section introduces another RESTful protocol, CoAP.

2.4.4 CoAP

The Constrained Application Protocol (CoAP) is designed to provide a protocol for RESTful designed (see 2.2.5) applications to constrained devices or networks. These could be embedded devices with very low memory or CPU power or networks with low bandwidht and high error rates. CoAP is designed to keep the overhead of the operations very low and to make requests similar to HTTP requests (see 2.4.2), but very compact and with less effort for the parsing of the headers [70] [71].

In order to achieve this, it uses UDP as the underlying transport or DTLS (see 2.4.1) if encrpytion or X.509 certificate authentiation (see 2.4.1.2) is needed. Due to the datagram communication the CoAP nodes interact in a peer-to-peer network style, not relying as much on server and client as for example HTTP. This means on the other hand, that every node must be able to accept incoming messages, which might cause issues with unaware firewalls or NAT gateways. Based on this datagram messaging layer, CoAP builds a request and response layer which can then execute requests and identify responses as in a classical server/client architecture [71].

In order to allow reliable request execution, CoAP nodes have to implement similar things like the TCP protocol stack, for instace congestion control, message deduplication, retransmissions, acknowledgements etc. Also requests and responses need to be mapped based on the individual and potentially asynchronous messages to form the request and response layer [71].

On the request and response layer, CoAP allows very similar operations to HTTP, it also has the same common methods for RESTful interaction, URIs, content negotiation and similar headers. The headers of CoAP are however binary options and do not allow for the addition of custom headers or protocol upgrades. Based on this similarities, CoAP also offers a stateless HTTP mapping which can be done by CoAP to HTTP proxies [70] [71].

CoAP also offers some extra features like a builtin resource discovery feature, which can be used to fulfill the HATEOAS constraint of RESTful design (see 2.2.5). Additionally it supports unreliable requests which might get lost without retransmission or multicast requests which can be answered by any of the servers in the multicast group [71].

The next section describes the WebSocket protocol, which establish a stateful connection for bidirectional communication over HTTP.

2.4.5 WebSocket

The WebSocket protocol was introduced to allow bidirectional communication between client and server using HTTP (see 2.4.2), for stateful connections and message exchange. It uses a normal HTTP request with some extension headers to establish the connection, also using potential HTTP authentication mechanisms (see 2.4.2.3) or underlying TLS security (see 2.4.1). After this initial request which makes use of the HTTP protocol upgrade header, the underlying TCP or TLS connection can be used by the websocket for bidirectional communication [72].

The opening of the WebSocket is done using an upgrade request like in listing 2.6, which is taken from [72].

Listing 2.6: HTTP request with WebSocket upgrade

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

The request is first of all a GET request to /chat on server.example.com, coming from the origin http://example.com (CORS, see 2.4.2.3). The "Upgrade: websocket" header requests a protocol upgrade to the WebSocket protocol and "Connection: Upgrade" declares that the connection will be afterwards used for the upgraded protocol's communication. The WebSocket specific headers are those starting with "Sec-WebSocket-", where the client challenges the server with a key (to avoid confusion with normal GET requests, clients must deny WebSocket connections which do not properly respond to their challenge), provides multiple choices (chat, superchat) for the sub protocol negotiation and requests the standardized WebSocket version 13 instead of previous draft versions [72].

An answer to this request is shown in listing 2.7, which is also taken from [72].

Listing 2.7: HTTP response with WebSocket upgrade

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Here the HTTP status code 101 signals the protocol switch with "Upgrade: websocket" confirmation of the websocket protocol and "Connection: Upgrade" confirmation on using the underlying TCP or TLS connection for the websocket. The ”Sec-WebSocket-Accept” contains a SHA-1 hash of the clients key (without trailing spaces) and the constant GUID ”258EAFA5-E914-47DA-95CA-C5AB0DC85B11” concatenated and then base64 encoded. This security mechanism ensures that the server knows what its doing. ”Sec-WebSocket-Protocol” is the result of the sub protocol negotiation, which is now explained in more detail [72].

As the WebSocket provides a generic messaging layer, the actual purpose and structure of the messages could be anything. To allow for a protocol negotiation similar to TLS' ALPN (see 2.4.1.1), the client sends a list of protocols it would support to the server and the server accepts one of the protocols or denies the request. A sub protocol name should be an ASCII string without whitespaces and contain the reversed domain of the protocol's creator as prefix (similar to Java package names). The names of the sub protocols can be registered with the IANA and may be treated case sensitively, but two different protocols should not just differ in case to allow for case insensitive matching as well [73] [72].

An establishment of a WebSocket connection is indicated by using a ”ws://” or ”wss://” URI instead of ”http://” and ”https://”, respectively. On the wire, WebSocket uses a message framing after the handshake, which can transport several message types. For payload, it supports text or binary messages which can be send in any direction. Additionally there are control frames for ping and pong, which can be used for active keep-alive and the closing message with an optional reason [72].

Using this set of functions, many protocols can be tunneled through a WebSocket and bidirectional communication with a stateful connection including keep-alive is possible. The next section introduces XML-RPC, a protocol for Remote Procedure Calls (RPC).

2.4.6 XML-RPC

XML-RPC is a remote procedure call (RPC) mechanism which uses XML for standardized message bodies (see 2.3.1 on XML). The messages are exchanged with the HTTP protocol using POST operations (see 2.4.2 on HTTP) [74] [75].

One of the most interesting features is that the messages encode their data structure in the message body using XML-RPC types. These offer a variety of common types like integer, double, string and additionally aggregates like arrays and structures. An extension even allows the usage of the ”nil” type, which is referred to as ”null” in many languages [76].

The XML-RPC client (usually ”caller”) sends an XML encoded request in a HTTP POST body, which contains a single XML <MethodCall> element. This element contains a method name and parameters, which are described with their type and value, but not with parameter names unless they are put in a structure. The server then answers with a <MethodResponse> element in the POST response body, containing the XML-RPC response which consists of a set of response parameters which are structured as in the requests [75].

Because of the XML structure, a server can always decode the request but it may not provide a method of this name or require different parameters. In this case, the server sends a XML-RPC response containing a <fault> element with some error value, which is not standardized at all [74].

An example method call which is put in the HTTP POST request body:

Listing 2.8: XML-RPC method call

```xml
<?xml version="1.0"?>
<methodCall>
  <methodName>sum</methodName>
  <params>
    <param>
      <value><int>42</int></value>
    </param>
    <param>
      <value><int>1337</int></value>
    </param>
  </params>
</methodCall>
```

A response for this request, returned in the HTTP response body or with an error:

Listing 2.9: XML-RPC response

```xml
<?xml version="1.0"?>
<MethodResponse>
  <params>
    <param>
      <value><int>1379</int></value>
    </param>
  </params>
</MethodResponse>

<?xml version="1.0"?>
<MethodResponse>
  <fault>
    <value>
      <struct>
        <member>
          <name>errorCode</name>
          <value><int>1</int></value>
        </member>
        <member>
          <name>errorMessage</name>
          <value><string>Invalid parameters.</string></value>
        </member>
      </struct>
    </value>
  </fault>
</MethodResponse>
```

2.4.7 SOAP

Based on the idea of XML-RPC (see 2.4.6), Microsoft developed a more extensive standard which was intended to suit the needs of all web applications, named SOAP. This was initially an acronym for Simple Object Access Protocol, but SOAP 1.1 was submitted to the W3C on May 8, 2000 and on July 9, 2001 the W3C published SOAP 1.2 with the note that it is no longer intended as an acronym [77].

SOAP defines a whole ecosystem for web services, with many components [77]:

- Various operations like: RPC, direct messaging, queued messaging, notifications
- Standardized error messages with well defined content
- The Web Services Description Language (WSDL, see 2.6.1) for well-defined service specifications
- Universal Description, Discovery and Integration (UDDI), a central registry of all web services which is itself accessible by a SOAP interface
- Even other transports than HTTP are supported, but not commonly used (e.g. SMTP)

Some of these components, especially UDDI, are not used by many SOAP users and are even considered dead [78].

Every SOAP message uses a specified envelope, which is the same for requests, responses, error messages and unidirectional messages. This envelope is an XML document which is versioned using XML namespaces (SOAP 1.1 uses "http://schemas.xmlsoap.org/soap/envelope/", SOAP 1.2 uses "http://www.w3.org/2001/09/soap-envelope") and consists of an optional message header and a required message body [74].

The header allows to add some metadata to the messages like e.g. a digital signature or transaction management information [77]. It is possible to set attributes on header elements, like the mustUnderstand attribute which indicates that this message must not be accepted by the reciptient, if it does not handle this header. Big web services use the optional header for authorization and payment purposes, but many, esp. smaller services, do not utilize headers at all [74].

The envelope body can either contain a fault element, or the content of any message like an RPC request or response. In case of a fault, the format of the fault is specified to have one or multiple of the predefined fault fields, e.g. faultCode, which is a predefined code like SOAP-ENV:MustUnderstand. The SOAP message body contains an own XML namespace which can either use a simple type (such as string, boolean, long) or a compound type. Implementations differ in how they provide the type information, it may be supplied by an XML schema definition or directly using xsi:type attributes on each value [74].

Listing 2.10 shows a sophisticated example of a SOAP message including headers, taken from [77]:

Listing 2.10: SOAP message example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header>
    <td:TargetDepartment
      xmlns:td="http://www.skatestown.com/ns/partnergateway"
      SOAP-ENV:actor="urn:X-SkatesTown:PartnerGateway"
      SOAP-ENV:mustUnderstand="1">
        Purchasing
    </td:TargetDepartment>
    <ai:AuthenticationInformation
      xmlns:ai="http://www.skatestown.com/ns/security"
      SOAP-ENV:actor="urn:X-SkatesTown:PartnerGateway"
      SOAP-ENV:mustUnderstand="1">
        <username>PartnerA</username>
        <password>LongLiveSOAP</password>
    </ai:AuthenticationInformation>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <doCheck>
      <arg0 xsi:type="xsd:string">947-TI</arg0>
      <arg1 xsi:type="xsd:int">1</arg1>
    </doCheck>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Now an assessment of the analyzed network protocols follows.

2.4.8 Assessment of network protocols

The Transport Layer Security (TLS, see 2.4.1) protocol can be optimally used to fulfill the security constraints from requirement #6 from table 2.1 also with sufficient authentication methods.

Three protocols which are designed for RESTful applications (see 2.2.5) are good candidates for the implementation of the service interface:

- **HTTP/1.1** (2.4.2): Old traditional HTTP, probably not the fastest but definitely supported everywhere.
- **HTTP/2** (2.4.3): New low level transport for HTTP to overcome performance limitations. On the high level fully compatible with HTTP/1.1.
- **CoAP** (2.4.4): Specialized protocol for constrained applications, but with a mapping to HTTP operations. Can be used instead of HTTP but is more restricted in extensibility and network conditions.

Additionally, two RPC protocols, XML-RPC (2.4.6) and SOAP (2.4.7), can be used for Remote Procedure Call communication. While XML-RPC uses a very simple and straightforward apprpach, SOAP has a massive complexity and lots of additional features, which also strongly recommend the usage of WSDL (see 2.6.1) for the interface specification. SOAP is most likely too complex for the purpose of the service interface, but using XML-RPC for callbacks is a considerable option.

Another analyzed technology is the WebSocket (2.4.5) a bidirectional messaging channel based on HTTP, which can be used to overcoe a strict client server architecture and to provide communication from server to client as well. It is considered in 2.5.3 as a suitable technology for callbacks using a stateful connection.

2.5 Callback handling

In this section, different techniques how callbacks can be realized, using the various network protocols, are described.

The general issue with callbacks is, that clients send requests to a server, and by protocol design, the server sends one response. If the request registers a callback, the server must be able to execute a request against the client, which it can answer back to the server. This is usually not builtin to the protocols, especially not the RESTful protocols (see 2.2.5 on REST and 2.4.8 for the suitable protocols) we are mainly looking at.

2.5.1 Double server

The most straightforward approach to solve the client/server issue of callbacks is to simply run a server on the client as well. The client can then provide its address or URL to the server in the callback registration request and the server can later during callback invocation act as a client and call the client's server using the provided address or URL. This simple solution does usually not have a specific name, it is simply common sense to come up with this solution.

For later references I will refer to this technique as the ”Double server” as instead of having one server (at the server side), we now have two servers (on client and server). Doubling the server is actually also where the problems of this technique arise: Normally, clients can use mobility and come and go as they like, they only execute stateless requests to the server in a RESTful design. This is also demanded by requirement #9 from table 2.1, stateless and suspendable protocol, that the client should be able to suspend its operation.

Thinking of a device like a smart phone, also further issues arise: the device might be behind NAT gateways or firewalls, which causes issues due to blocking incoming connections, and mobility of the device like changing networks and also IP addresses is very common. Maintianing the state of the callback to keep it reachable under these circumstances is very cumbersome.

On the one hand, this issue is the reason why this section exists and the further techniques described here were developed, on the other hand, this ”Double server” pattern is a good option in local networks were no IP mobility is needed.

2.5.2 Long polling

A traditional solution with the HTTP protocol (see 2.4.2) is the so-called long polling. The basic idea behind it is to send a request to the server, but the server does not answer the request yet. The server waits for a callback invocation to happen and keeps the open request open till this happens [79].

In practice, the clients have timeouts which means the request has to be frequently reexecuted to have a fresh connection for the callback. This requires fine-tuning of the server and client to actually identify a good timeframe for how long a request is open [79].

The management of long polling requests is quite cumbersome and the frequet but useless requests if no callback invocation happens creates additional network overhead [79].

2.5.3 Stateful connections

Stateful connections are connections which are actively kept alive by the client so the server can use the connection to monitor the clients availability and send a callback invocation to the client. This is actually one of the main purposes for which the WebSocket protocol (see 2.4.5) was initially created [72].

The effort to maintain this stateful connection is way less than with the long-polling (see 2.5.2) and an open WebSocket for example creates less network traffic than the frequent long polling requests. Also WebSockets and similar connections preserve the client server architecture.

2.5.4 Server push

Newer protocols like HTTP/2 (see 2.4.3) but also CoAP (see 2.4.4), include a feature called server push. The idea behind it is to push resoures to the client which it would otherwise have requested anyway, for example images which are referenced in an HTML file [65].

This feature is originally not designed to create callbacks, but it would be possible to abuse the push promises to push callback invocations to the client, which then executes a reply request using some custom request and response mapping. An implementation of this feature would probably be quite hacky and have issues to maintain a state of the callback, i.e. know if it is still possible to invoke it, without invoking it. Using the push feature for this purpose is not intended by the standard, and should therefore be used with care [65].

2.5.5 Assessment of callback techniques

The ”Double server” approach provides a clean and practical way to provide callback functionality if there is no need to stick with the client server architecture and the client can easily provide a server. The clients needs more resources and changing IP addresses of the client can be an issue, but otherwise the method is usable.

Long polling has a lot of issues and is considered obsolete given the options of WebSockets or similar stateful connections, which provide a clean way to maintain a callback channel within the client server architecture. The client can also reconnect to the websocket after it changed its network, allowing for a smooth handover which is also transparent to the server in terms of callbak state maintenance, i.e. the server knowing if a callback can still be used.

The new server push feature could be used for callbacks, but is not actually designed for this purpose, so it is advisable to stick with the other solutions.

2.6 Interface description for code generation

With the goal to create interfaces for many programming languages, code generation techniques could be useful to automate the development of the bindings for further languages. A common method which is usable with many programming languages is using a dedicated language or tool to describe the interface of the code and then a compiler or other tool can generate the language specific code.

In this section, two suitable techniques are analyed: Using Interface Definition Languages (IDLs) of technologies which are already considered in this chapter and the Simplified Wrapper and Interface Generator (SWIG).

Interface Definition Languages (IDLs) originate from the Common Object Request Broker Architecture (CORBA), a binary communication protocol which allows RPC calls (see 2.2.4 on RPC). It was needed to describe the binary data layout to generate code which properly reads the binary data, similar to the proto files from Google protocol buffers (see 2.3.4) [80].

The description language described now is the Web Services Description Language (WSDL).

2.6.1 Web Services Description Language

The Web Services Description Language (WSDL) can be used to describe a web service (esp. SOAP service via HTTP) with an XML document [81]. It essentially describes three fundamental properties of a webservice: what a service does, how it is used and where it is located [77].

There are six major elements for describing a service:

- **portType** [what]: This is an abstract interface having abstract operations with abstract messages. The bindings describe concrete implementations of a portType.
- **message** [what]: Defines a set of parameters with concrete types, structure and order which are used in the message bodies. This is referenced by the portType, in order to describe the interface of the abstract messages.
- **types** [what]: All types used by any message of this service, therefore referenced by the messages.
- **binding** [how]: A binding is a concrete implementation of a portType, especially what messages form an operation, possible fault messages, etc.
- **port** [where]: Where a binding is deployed, i.e. its specific network location (e.g. URL).
- **service** [where]: A collection of ports which form a service.

WSDL can be used to generate code, e.g. Java code, for the server and client sides of a SOAP service, so that the service is only specified using the WSDL document and the interfaces used by various programming languages match automatically [77]. The usage of WSDL would be recommended for a SOAP interface, otherwise it is not usable.

2.6.2 Simplified Wrapper and Interface Generator

The Simplified Wrapper and Interface Generator (SWIG) is a tool which builds wrappers for native system libraries which are written in C or C++. It can then generatenative language bindings for many popular scripting languages and also native languages like Go or D [82].

It works by creating a native wrapper system library first, which wraps the original library with some SWIG generated helper functions and conversions for error handling, allocation handling etc. Then it automatically creates a native binding in the target programming library which uses methods like the Java Native Interface (JNI) or native library loading in Python to loadthis wrapper system library. The native language wrapper then exposes native functions in the target language and encapsulates all the management logic for loading and using the native wrapper library within the target language wrapper module [82].

The supported target languages of SWIG are: Allegro CL, C#, CFFI, CLISP, Chicken, D, Go, Guile, Java, Javascript, Lua, Modula-3, Mzscheme, OCAML, Octave, Perl, PHP, Python, R, Ruby, Scilab, Tcl, UFFI and some dialects and variants of these [82].

Basically the SWIG mapping can be done by providing an interface description file which simply imports the C header file, but additional instructures for detailed type mapping, translation of C errors to target language exceptions and many more is supported in this file [82]. An example file which simply imports a C header is shown in listing 2.11.

Listing 2.11: SWIG interface example

```
%module simpleswig
%{
#include "some-library.h"
%}

%include "some-library.h"
```

While it provides a very simple starting point, SWIG can also generate flexible wrappers with advanced features for the target language [82].

2.7 Summary

After introducing the problem domain with specific requirements in 2.1.3, various technologies which can solve parts of the problem were analyzed step by step.

The analysis of design principles for the service interface architecture showed that the Representional State Transfer (REST) principle is very suited for the service interface of the Virtual State Layer middleware.

The technology comparison was first performed on serialization formats, where all four analysed formats showed specific fields where it could be applicable. The results of this assessment are summarized in 2.3.5. Afterwards, suitable network protocols were analyzed and it was discovered that the Transport Layer Security (TLS) protocol serves well to fulfill the security requirements of the service interface. The RESTful protocols HTTP (in version 1.x and 2) an CoAP could both be used to implement the service interface. This assessment is summarized in 2.4.8.

Then, different callback techniques were researched and assessed to analyze different methods how the callbaks in the VSL service interface can be realized. The assesmment of the callbacks is summarized in 2.5.5. Finally two methods for automated code generation were analyzed to automate parts of the implementation in this thesis. It was specifically discovered that the Simplified Wrapper and Interface Generator (SWIG) can be used to create a Python connector by wrapping the connector for C.

The next chapter now presents the related work.

Chapter 3

Related work

In this chapter, related work is analyzed where similar problems were identified and their results are compared and incorporated into the design of my service interface. This is grouped based on the main focus of the related work, starting with related work on RESTful design.

Afterwards, related work on the network technologies is analyzed, such as comparison of serialization formats, examples of using or adopting standard protocols for embedded devices and comparisons of different network protocols. Also examples of native interfaces for many programming languages by using network communication and by using the Simplified Wrapper and Interface Generator (SWIG, see 2.6.2) are presented.

The first section analyzes the RESTful design on the example of a high performance computing web interface.

3.1 RESTful design

While there are many applications which chose a RESTful design (see 2.2.5 on RESTful design), the particular project of the National Energy Research Scientific Computing (NERSC) Center used the RESTful design paradigm to create an interface to their existing not RESTful High Performance Computing (HPC) grid. With the resulting "NERSC Web Toolkit" (NEWT), direct access to the HPC resources via a RESTful interface is granted, which coexists with other access methods [83]. This relates to the Virtual State Layer (VSL, see 2.1.2.1) middleware's ability to support different transport connectors (cf. 2.1.2.3) which can also use different protocols and architectures based on their specific requirements. The goals of the NEWT project also show similarities to the requirements defined in 2.1.3, like an easy usability for programmers due to the well-known architecture and technologies or the high portability for support of many platforms [83].

The client of their REST service is a HTML5 web interface which utilizes JavaScript requests for the operations. In order to support normal web browsers, they utilize the HTTP protocol (see 2.4.2) with JSON data serialization (see 2.3.2) and SSL/TLS for encryption (see 2.4.1). For authentication, a form based login with session cookies is used, although this violates the RESTful design, in order to achieve a higher usability for normal users without high technical knowledge [83].

A performance evaluation showed ”very little overhead for most standard Grid operations in the NEWT layer” [83] and therefore proves the efficiency of the design approach and the used protocols. The exposure of the HPC resources via URIs and the mapping of the operations to the HTTP verbs showed to be simple and straightforward, although some operations could only be mapped using the POST operation. The resulting design is very flexible and extensible, for instance, the possibility to include new functionalities in new resource subtrees was emphasized [83].

Specific to the use case of using normal browsers for the access to the RESTful HTTP service, the challenge of doing Cross Origin Resource Sharing (CORS) requests is addressed. It is a security mechanism of modern browsers which denies JavaScript accesses to other URLs unless the specifically allow it in their HTTP headers. The NEWT project addresses these issues and describes their approach for CORS handling using the respective HTTP headers, which is useful for the creation of REST clients inside a browser [83].

In a nutshell, the NEWT project successfully provides a web service interface to the existing HPC grid infrastructure using the RESTful design principles. The challenges are addressed and the performance analysis shows promising results. For the implementation, HTTP and JSON are used because they can be directly used in a normal web browser. Especially native clients that can use a broader variety of formats, can utilize different serialization formats, so the next section analyzes comparisons of serialization formats.

3.2 Comparison of serialization formats

The choice of the serialization format (see 2.3) and the specific library used during implementation is often influenced by the performance of this combination. Many comparisons and measurements have been conducted that provide an overview of the performance to expect. Based on the four analyzed formats in 2.3, I performed my own measurements to evaluate them in 6.1 using my Java implementation which is described in 5.1. The choice of the libraries used in the Java implementation is influenced by the results of this existing research. The papers which are referenced here, [31] and [84], conducted broad measurements of different text and binary formats using multiple libraries and provide a broad overview as a base for my decisions.

While the two papers evaluated some additional formats to those which I presented in 2.3, they both analyzed XML (see 2.3.1), JSON (see 2.3.2) and the protocol buffers (see 2.3.4). As CBOR (see 2.3.3) is very young (published 2013), it was not yet included in these two benchmarks (2011 and 2012). The used programming language of all tests was Java, using different libraries for most of the formats [31] [84].

During the measurements, the throughput performance and the size of the serialized data were captured. While [31] uses size of raw object divided by time needed for serialization to give a throughput of bytes per second, [84] measures the time needed to serialize 500 instances of the same reference object, i.e. the total seconds per test run. The size of the serialized data was measured very differently, [84] took an avarage of ten different reference data objects in bytes and secondly the size in bytes of a very small object. In contrast, [31] used a small reference object with absolute overhead in bytes, being serialized size minus data size, and two larger objects where the overhead is described in percentage which the serialized data is larger than the original object.

Regarding the serialized data size, both papers come to the conclusion that XML is the largest format with the most overhead and that JSON is way more efficient in contrast. Both also conclude, that the binary formats and especially protobuf are more compact than the text formats. More specifically, protobuf also performs very well compared to other binary formats, although compressed formats or Apache Avro can sometimes perform better than protobuf [31] [84].

The results of the throughput vary a lot more depending on the test and evaluation method, library and format. Both papers agree, that binary formats and especially the protocol buffers usually have the highest throughput and in both protobuf is a lot faster than the other binary formats [31] [84]. Potentially due to the special test setup and data structure, XML outperformed JSON by a large factor in [31], while [84] shows a way better performance with JSON than with XML. This is even still the case if only the Jackson library, which performed best amongst the JSON libraries, is compared. Another notable result from [31] is the very high performance of the WoodStox XML library compared to other XML libraries. So these two libraries, Jackson and WoodStox for XML, seem to be good choices on Java for efficient serialization.

On compression of serialization formats, [31] identifies a reduction of the overhead but at the expense of a very big performance impact. So compression can be used to further reduce the data size, but at the expense of a lot more calculations during serialization. Also [31] points out how having separate objects generated from a schema like for example with protobuf leads to copying of objects before the serialization compared to the handy direct serialization of Java objects using reflection functions from libraries like Jackson. This process can be further specified using annotations but provides a slight overhead at runtime during initialization of the library for parsing the object [31]. The parsing itself can however access cached information on the object, which leads to less impact at runtime with repeated serializations. [84] even used a dedicated warm-up phase before the measurements start to get the runtime code generation to perform the optimizations before the measurements.

The key results of these related benchmarks show, that binary formats should be considered if a low size and efficient serialization is intended. The text formats provide a good readability on the one hand [84], but at the expense of more overhead on the other hand. For the measurements, different ways to calculate the throughput and other setup specific details can lead to contradicting results and a warm-up of the Java Virtual Machine is required to get representative data for a real runtime operation.

The next section analyses related work on comparisons of network protocols for the operation on embedded or resource constrained devices.

3.3 Comparison of network protocols

The different protocols which are analyzed in 2.4 are used very frequently and therefore others have already performed comparisons of the protocols.

For example [85] presents the Californium library for CoAP (see 2.4.4) and performed a performance evaluation of the Californium CoAP library compared to common CoAP and HTTP libraries and implementations. The tested other systems are: Initial-Cf, Sensinode, nCoAP, OpenWSN, Vert.x, Jetty, Grizzly, Tomcat, Node.js and Apache with PHP.

For the measurment, many clients were used which execute requests parallely on the test system. The Californium library showed the highest performance of all tested library and exceeded the other CoAP and HTTP libraries. HTTP/2 (see 2.4.3) was not enabled on any of the tested HTTP servers, so the test compared to HTTP/1.1 only [85].

Amongst the tested Java web servers, Vert.x, Jetty, Grizzly and Tomcat, Vert.x and Jetty showed the best performance. Vert.x was faster for smaller number of clients but shows a certain instability with high performance deviations. Jetty performed very stable and could handle the maximum number of 10,000 concurrent clients very well [85].

A comparison of SOAP (see 2.4.7) and RESTful HTTP (see 2.2.5 and 2.4.2) using XML (see 2.3.1), JSON (see 2.3.2) and Google protocol buffers (see 2.3.4) has been performed by [86]. SOAP only used XML as per definition of SOAP and four example operations were tested using the four protocol and serialization format combinations [86].

Every combination with RESTful HTTP performed better than SOAP, using less round trips and a smaller operation latency. The different serialization formats which are used with RESTful HTTP showed similar performance for XML and JSON and a better perofrmance using Google protocol buffers [86].

The measurement of [86] clearly shows that SOAP is not recommendable if low latencies or low overhead is intended. Both papers show the high practical fitness of the RESTful desing with HTTP or CoAP, where compared to the old HTTP/1.1 CoAP performed way better [85] using the Californium library and comparable using the other CoAP libraries. Also [86] showed as in 3.2, that Google protocol buffers are more efficient than JSON or XML.

3.4 Native interfaces for multiple programming languages

The goal of transferring an existing implementation in a single language to many programming languages by providing a native interface for the other programming languages has also been pursued by other research projects as well.

In [87], a cloud runtime for High Performance Computing (HPC), which originally only supported Java, was extended to support native bindings to C, C++, C#, Python and R. The selection of languages is very similar to my case. To allow the accessibility of the Java runtime in these languages, a bridge was created which either used a TCP network transport or a system pipe for the communication [87].

While they considered the usage of SWIG (see 2.6.2) in their related work, the implementations for the languages were all developed individually. The reason for this decision was the wish to have a more direct control over the typing in the different languages [87].

Another project which provided a native connector to a C++ cheminformatics tollkit to Python actually used SWIG to perform the mapping [88]. In this case SWIG was used to map a C++ library, where SWIG is also capable to map C++ classes to Python objects. The issue of having less control over the types was not considered an issue there [88].

The performance measurements of the network bridge show a higher performance with local pipes than using TCP and a lot higher performance with compiled languages than script languages like Python and R [87]. The SWIG binding for Python was not evaluated for its performance, bu worked as expected in the tests. Also the usage of SWIG was seen as convenient for the mapping [88].

Chapter 4

Design

This chapter explains the design considerations for the service interfaces developed in this thesis and shows the structure of the components that are implemented. The suitable approaches and technologies identified in chapter 2 and the results of related work from chapter 3 are incorporated into these considerations.

First, the intended architecture of the service interface is explained and reasoned. The architectural principles from 2.2, especially the RESTful design (see 2.2.5), plays an important role in this process. Additionally, the required components for the Knowledge Agent (see 2.1.2.1) and the different service interfaces are described.

Afterwards, the first required component is specified, the data serialization unit which translates the data structures into transmittable transport data (see 2.3 on serialization). The most important data structure, the VSL node (see 2.1.2.1) which stores the actual VSL data and metadata, is precisely specified. Multiple serialization formats can be used with this specification and are compared in the evaluation section 6.1.

Then the network protocol for the actual communication using HTTP (see 2.4.2) and HTTP/2 (see 2.4.3) is described. The concept of this RESTful HTTP interface is also transferrable to other RESTful protocols like CoAP (see 2.4.4), for which the relevant transfer steps are outlined in 4.3.9. Multiple protocols can even be deployed to a single knowledge agent instance, as the Transport Manager (see 2.1.2.3) handles the presence of multiple transport modules. An important detail of this specification is the callback handling following the techniques described in 2.5, which is described in detail in 4.3.7. The HTTP interface with different serialization formats is then implemented in chapter 5 and evaluated in 6.2.

The first section now proceeds with the architectural design of the service interface.

4.1 Service interface architecture

As already discovered in 2.2.5.2, the current design of the Virtual State Layer (VSL, see 2.1.2.1) has many similarities with the Representional State Transfer (REST) architecture (see 2.2.5). Following the Richardson Maturity Model (see 2.2.5.1), VSL reaches level two (with level three being RESTful), mainly missing the "Hypermedia as the engine of application state" (HATEOAS, see 2.2.5) principle, which is also discussed in 2.2.5.2.

One of the possible advances which could be made in the direction of being RESTful is to adopt the HATEOAS principle. This would require a service to be able to reach all VSL information, that is accessible by the service, by following links from a single entry URL. Locating data in the VSL, especially available services, normally works by using either the type search to search for instances of a specific service type or by enumerating all child nodes of a parent node, which could also be a whole knowledge agent. The current functions however do not allow to enumerate all knowledge agents, making enumerating all services by requesting each knowledge agent a difficult process. In addition, it would be preferable to organize services by type and not by location, as the latter is randomly decided by where the service connected and services can also move from one knowledge agent to another. For searching by type however, an enumeration of all available types is currently not easy to reach[1], so an extension to the VSL is needed to fulfill the HATEOAS principle. It needs an entry point where either available types or all available knowledge agents can be listed. Using the type list for service discovery seems more reasonable.

On the other hand, it might not even be useful in practice to have this kind of functionality as services usually know the types of other services with which they can interact. So they can use the type search directly, even without listing all available types, getting an empty result if a type is not available. Still the addtional funtionality of listing available types can be useful for certain purposes, and it can simply stay unused in other cases. Another detail to add to this is the enumeration of search providers, as also other ways to search for services can exist. For now it seams reasonable to leave the current state as it is and accept a little violation of the HATEOAS principle.

Another design consideration which also leads to violations of the REST principle is the callback handling. Once a service registers a callback with the knowledge agent, by using one of the operations (listed in 2.1.2.4) that provides a callback as argument, the service starts to have a state by providing the callback. This functionality is an important part of the VSL design but the state is kept transparently without changing the behaviour towards other services, so the VSL can be used stateless as long as no callbacks are registered by the service which wants to stay stateless.

[1]Actually, there is the site-local context model repository, which can be queried for all available types if the access rights are granted. However, it does not state whether services actually use this type and it has to be located via type search first, making this process cumbersome and not suitable for this purpose.

Once callbacks are registered, the execution of a callback from the knowledge agent to the service is a Remote Procedure Call (RPC, see 2.2.4). So at this specific detail, the RPC architecture is important for understanding the callback mechanism and its implications, especially with regard to state. Both sides need to maintain a state about the callback, the knowledge agent that the callback is registered at all, what for and how the service can be reached for the callback and the client where the callback function is and the registration state so it can be registered again after connection losses. It is also very useful if both sides monitor the availability of the callback, so that the service can register it again after the knowledge agent changed or restarted and for the knowledge agent to cleanup callbacks that are not reachable anymore, for example because the service is no longer running. The specific callback handling techniques used and the general considerations for maintaining this state is discussed in detail in 4.3.7.

The next section provides details about the used data structures and the design of their serialized representaions in different serialization formats.

4.2 Data structures and serialization

The most important data structure of the Virtual State Layer (VSL) is the knowledge node (or VSL node), which is a node in the knowledge tree as described in 2.1.2.1. These nodes store data and metadata about the virtual state of services in the smart space. Additionally, as the structure is a tree, each node can have children and is usually the child of a parent node, unless it is the root. The different possibilities how these nodes can be structured are discussed in 4.2.1, with a resulting design that is most useful in practice.

Other data structures exist as well, most notably the callback invocation and callback response messages used by the callback handling, which is described in more detail in 4.3.7. Addtionally, there is the description of post operations used for extended VSL operations that are mapped to HTTP POST and a description of VSL exceptions in case an error occured. The detailed handling of the HTTP operations and errors is designed in 4.3.2 and 4.3.3. These additional data structures and even some more with their corresponding serialized forms are discussed in 4.2.2.

The last part of this section describes some specialities of certain serialization formats and how I dealt with them. This applies mostly to XML and protobuf, as for XML there are differences between attributes and child elements and some structures like lists or maps require special handling. Protobuf uses the proto files for schema description (see 2.3.4), which also needed some special considerations which are discussed there. Now the serialization of the VSL node structure is desribed in detail.

4.2.1 Serialization of VSL nodes

The node of the virtual state layer has two basic types of content, data and metadata. The data is a simple string which represents the value of the node, with any further semantic of this string value being described by the VSL node's type. This type is part of the node's metadata and represented as a list of the type and its supertypes (see 2.1.2.1 for more on the types).

Addtional to types, the metadata contains a version number, which counts up for newer versions of the node, and a timestamp of when this value was set. Furthermore, there is a map of restrictions with restriction types and their restriction value, represented as a map of strings to strings with the former being unique in this map. These restrictions apply to the value of the node and are derived from the type internally. The access to the node is also described as either "r" (read only), "w" (read and write) or "-" (no access), which differs depending on the service which accessed the node, as every service can have different access rights to the node.

The data is stored in a field called "value", the metadata in the fields "types", "version", "timestamp", "restrictions" and "access" respectively. The metadata as well as th value itself can also not be present, most specifically if only data (or metadata) was requested or for example if a node does not contain a value, which is also possible. To optimize the compactness of the nodes, fields that are empty or contain the default value are not serialized at all and the deserializer just inserts the default or null value instead.

Now the tree structure has to be considered; requests to the VSL might request a node with all its children or with its children up to a certain depth as sepcified in 4.3.1, so the result should always be a VSL subtree structure of nodes, which might also be just a single node. To represent this tree structure, two main methods are favored in computer science, listed below.

- **Recursive embedding of children**: the recursive approach is based on serializing the node which is the root of the (sub)tree and to include the direct children of it in a relative address to child map. These children may then include further children, also by their relative address. Written with brackets surrounding a node, the example would look like this: (root: path1->(child1: sub1->(child 1.1), sub2->(child1.2)), path2->(child2: sub1->(child2.1)), path3->(child3)).
  This recursive structure is on the one hand quite intuitive to read and use in the code, but on the other hand it requires a recursive implementation of the parser as well. A huge benefit of this structure is that each substructure (like "child1" in the example above) is by itself a valid subtree, that can be easily passed to functions that deal with this subtree as if it was requested directly. Another benefit is the easy iteration of the children with different iteration patterns (like depth-first or breadth-first) and their intuitive implementation and handling. But the downside of this approach is, that in order to reach a certain path like "path1/sub2", multiple

resolutions of children are required, namely one for each path component.

- **Address to data map**: this approach does not nest children into their parents, but just provides a mapping of all addresses (which need a path separator to add multiple path components to one path; mostly / is used for this purpose) to the node data. The example would look like this in an address to data map: /->(root), /path1->(child1), /path1/sub1->(child1.1), /path1/sub2->(child1.2), /path2->(child2), /path2/sub1->(child2.1), /path3->(child3).
  The benefit of this structure is an easy access to all subpaths, especially the deep ones (like /a/b/c/d/e/f/g) and a flat parsing of the structure (one map, then only flat nodes without children). On the other hand, the extraction of a subtree is a very cumbersome process: an iteration of the whole structure needs to select all childs that meet the subpath's prefix (e.g. /path1) and then this subpath prefix must be stripped from all paths (making /path1/sub1 only /sub1 and so on). All of this would have to create a new address to data map, also leading to additional allocations. Another pitfall of this strcuture is the access to the requested node (the ”/” node; in some cases this is even the only requested node), as a resolution of this path is required to access its data, so instead of ”deserialize(rawData).getValue()” one would need ”deserialize(rawData).get("/").getValue()” to access a single value request's value. This is important to consider as just accessing one value is a common use case of the VSL middleware.

After the evaluation of these two structural approaches, I considered to build a hybrid using the root node directly and then providing an address to data map for all children, regardless of the depth inside the root node. This would avoid the usage of a recursive parser as the parsing depth would always be one and solve the issue of accessing the requested node directly. On the other hand it showed, that using a recursive parser is even more handy than having two ways of parsing the root with children and then the children as leaves without further recursion and especially it showed, that translating the tree strcuture used for internal processing to a flat children map needs copying of all nodes which have children. This is due to the issue, that the serializer would serialize a node with children as a root node, effectively repeating the children, and that the paths need to be built with nodes that do not have any children to prevent this.

Finally I abandoned this idea and decided that the recursive embedding of children is the most usable approach for the use cases of the VSL middleware, where each subtree in its own has a valid type and meaning, which can be delegated to other components that deal with this specific subtree only. Avoiding any copying of the tree structure avoids redundant memory allocations and potential race conditions if a node is removed or added to the tree (which is however a rather rare case in the VSL, as it can only happen with lists). Also this format is more intuitive and human-readable, easing the debugging of VSL data accesses.

The plain node data and metadata without children would look similar to listing 4.1 in JSON format (see 2.3.2 for the description of the JSON format):

Listing 4.1: Example VSL node data as JSON

```
{
	"value":"1234",
	"types":[
		"/basic/number"
	],
	"version":123456789,
	"timestamp":1356217200000,
	"restrictions":{
		"regex":"[0-9]+"
	},
	"access":"w"
}
```

A node structure with children is presented in listing 4.2 also in the JSON format, but in this case without metadata:

Listing 4.2: Example VSL node structure as JSON

```
{
	"value":"this is root",
	"children":{
		"child1":{
			"value":"child1 value",
			"children":{
				"child1.1":{"value":"foo"},
				"child1.2":{"value":"bar"}
			}
		},
		"child2":{
			"value":"child2 value",
			"children":{
				"child2.1":{"value":"value of child2.1"}
			}
		},
		"child3":{"value":"123"},
	}
}
```

The real node serializers will create less whitespace (as it does not have meaning in JSON) and probably change the order of the fields, but apart from that, these example are real JSON examples. Now some other data structures are briefly discussed.

4.2.2 Serialization of other data structures

There are a bunch of other objects used by the VSL middleware, for example for the callback invocation (see 4.3.7.1) or in the KA to KA communication. They are described in this document as ”objects” with fields as almost all of them are just flat datastructures. These objects can be canonically translated to JSON or the other serialization formats with the general rules and considerations of this section.

One datastructure which is more complex are the KOR updates which are used in KA to KA synchronization (see 4.3.8). They contain the VSL node structure metadata, which is very similar to the VSL nodes but containing the internal metadata. The same serialization rules as for the nodes in 4.2.1 are applied.

Now the serialization in the other formats than JSON is specified.

4.2.3 Special handling of serialization formats

The aforementioned design considerations are valid not just for JSON, but also for all other serialization formats. However, some of the formats require special considerations that are discussed in detail here.

For instance, the XML document (see 2.3.1) lacks a distinction between value fields and array fields, with both being represented by a tag when following the advice from [46] not to use attributes. For the array representation, one option is to repeat the tag just multiple times for each array element, which is called the unwrapped mode. The other option is to make a tag for the array which contains the array elements and then include the array elements each with a separate element tag. This is called the wrapped mode then and usually regarded as cleaner design, because an empty array has an explicit representation (just empty wrapper tags) and the array elements must be grouped together and cannot spread everywhere inside the node (like in the unwrapped exampe below, listing 4.3).

Listing 4.3: XML unwrapped array

```
<?xml version="1.0"?>
<root>
        <somefield>foo</somefield>
        <arrayfield>1</arrayfield>
        <arrayfield>2</arrayfield>
        <anotherfield>bar</anotherfield>
        <arrayfield>3</arrayfield>
</root>
```

Listing 4.4: XML wrapped array

```
<?xml version="1.0"?>
<root>
        <somefield>foo</somefield>
        <arrayfield>
                <value>1</value>
                <value>2</value>
                <value>3</value>
        </arrayfield>
        <anotherfield>bar</anotherfield>
</root>
```

Another structure which needs special considerations for XML is the map. Maps should be always wrapped, but inside it could use a tag which then has the subtags key and value or it could use key as the tag type and value as the value inside the key tag. While the former is a very exact representation of the map in XML, the latter is nicer to read for humans and a more natural representation of a key to value map. Two pitfalls are important when using key-tags: document type definitions (DTD) have to mention all possible map keys as tags as the document does not conform to it otherwise and the map keys must be valid XML tag names. If these two are not a problem, the more natural key-tag representation (see listing 4.6) is favorable and otherwise the clean representation (see listing 4.5) is needed.

Listing 4.5: XML map (clean)

```
<mapfield>
	<item>
		<key>key1</key>
		<value>foo</value>
	</item>
	<item>
		<key>key2</key>
		<value>bar</value>
	</item>
</mapfield>
```

Listing 4.6: XML map (key-tags)

```
<mapfield>
	<key1>foo</key1>
	<key2>bar</key2>
</mapfield>
```

In order to make this design very standardized, interoperable and easy to implement with different languages and libraries, the clean map solution and the wrapped arrays are used. An example XML node similar to the JSON example in listing 4.1, but with an additional child node with just a value, is shown in listing 4.7.

Listing 4.7: Full XML example of a VSL node

```
<?xml version="1.0"?>
<node>
	<value>1234</value>
	<timestamp>1356217200000</timestamp>
	<version>123456789</version>
	<access>w</access>
	<restrictions>
		<item>
			<key>regex</key>
			<value>[0-9]+</value>
		</item>
	</restrictions>
	<types>
		<type>/basic/number</type>
	</types>
	<children>
		<child>
			<address>childpath</address>
			<node>
				<value>1234</value>
			</node>
		</child>
	</children>
</node>
```

Regarding the binary formats, CBOR (see 2.3.3) uses exactly the same structure as JSON without any changes or special adoptions. However, adopting the protocol buffers (see 2.3.4) is not so trivial, as the proto files for the schema definitions need to be created.

The proto schema needs explicit field names which are not written to the serialized data, so weak map structures like an XML key-tag map or the JSON representation of the children is not possible. To solve this, a very explicit structure like in the XML design above is needed, just with the difference that the ”repeated” type is used for arrays and map items. For example for the VSL node structure, the proto file is shown in listing 4.8.

Listing 4.8: Proto file for VSL nodes

```
message VslNode {
    repeated ChildAt children = 1;

    optional string value = 2;
    repeated string types = 3;
    optional sint64 version = 4;
    optional uint64 timestamp = 5;
    optional string access = 6;
    repeated MapEntryStringString restrictions = 7;

    message ChildAt {
        required VslNode node = 1;
        required string address = 2;
    }

    message MapEntryStringString {
        required string key = 1;
        required string value = 2;
    }
}
```

With these additional considerations, the design of the data strcutures can be used with all four implemented serialization formats (and potentially a lot more). The considerations for the VSL nodes presented here are also applied to the other data structures from 4.2.2. In the next section, the RESTful service interface protocol is specified.

4.3 Protocol of the service interface

This section specifies the protocol of the RESTful service interface using HTTP (see 2.4.2) for the Virtual State Layer (see 2.1.2.1). Both HTTP versions 1.1 and 2.0 (see 2.4.3) are supported because the high level protocol is the same and HTTP/2 mainly promises a higher efficiency for the same operations. The security requirements (requirement #6 from table 2.1) are provided by using TLS 1.2 only (see 2.4.1) with ALPN (see 2.4.1.1) support for HTTP/2. Authentication is done using X.509 client certificates under a common certificate authority (see 2.4.1.2) of the knowledge agent and the services (see 2.1.2.3 for an overview of the knowledge agents and services).

In the first step, the resource paths and URIs are defined, followed by the definition of the methods which can be used with these resources and which VSL operations they represent (see 2.1.2.4 on the service interface operations). Afterwards, some specific details are defined mmore precisely, like the error handling, the usage of HTTP content negotiation (see 2.4.2.1 on HTTP content negotiation) and some details on the usage of important HTTP headers (see 2.4.2.3 for an introduction to these headers).

The next section describes the usage of SSL/TLS with client certificate authentication. Then, the implementation of callbacks using the WebSocket protocol (see 2.4.5) and how they are managed is elaborated.

Finally the to last sections describe the extensions which are needed to use the same transport module also for KA to KA communication and what steps and considerations are needed to implement a RESTful interface using other protocols than HTTP.

Now the resource paths and URIs are defined.

4.3.1 Resource paths and URIs

The heart of the VSL middleware is the tree of VSL nodes, which are already described in detail in the serialization section 4.2.1. The tree is built up from a root nodes and each child has relative addresses, which are also compatible to URI paths. See also the examples of this addressing in 4.2.1.

This make the indentification of URIs for the VSL nodes very straightforward, the only extension is to add a /vsl/ prefix, which is done to allow other resources to exist without potentially colliding with nodes in the VSL tree. So the VSL address "/agent1/service/someValue" is represented in the resource path "/vsl/agent1/service/someValue". The VSL addresses can also have parameters in the style of URL query strings. The parameters are depth for how many children should be included (<0: all, 0 the default: no children, >0: this many level of children) and scope ("metadata": only metadata and no values, "value": the default to just get the value, "complete": value and metadata) [89]. The parameters are simply included as query string in the resource path. An example URL with parameters: "/vsl/agent1/service/someValue?scope=complete&depth=2"

Additional resources are used for every operation which is not executed on a specific VSL node, i.e. it does not have an address parameter. This is for example the case with the registerService and unregisterService methods (see 2.1.2.4). These methods get specific resource paths, in this case "/service/register" and "/service/unregister". The usage of these resources is explained in section 4.3.2 and further special resources are defined in the next sections for specific purposes like the callbacks (see 4.3.7).

Another kind of extra resources is the ability to deliver static files or resources through the same interface, in order to fulfill the RESTful principle of code on demand (see 2.2.5). They must reside in the path "/static/" or use common HTML names like "index.html" to avoid collisions with future extensions to the additional resources. Also the path "/" may deliver the index.html if the accept header specifies it, but this special case of "/" is defined more detailed in 4.3.4. The usage of paths with the prefix "/static/" or paths like "index.*", "robots.txt" or "favicon.*" is denied for other purposes than static file delivery. While the actual inclusion of static files is optional like also the whole code on demand REST constraint is optional, it is an interesting possibility to serve for example a webinterface directly from the KA to common browsers.

In order to provide HATEOAS (see 2.2.5) links to all resources, the resource ”/resources” provides an index of all available resources with their intended usage (”service” for resources for the services, ”ka” or resources which are only used in KA to KA communication and ”static” for static files). With the subresources like ”/resources/client” or ”/resources/static”, only resources of this type are listed, using the same listing format. The format is an object which is serialized as described in 4.2.2, containing the list ”links” to resource description objects with the fields ”path” (like ”/service/register”), ”type” with the intended usage and ”name”, which provides a name that can be matched for resource discovery. Optionally this per-entry object may contain a ”description” field, which must be ignored by implementations but can be used for developers to identify the usage of the resource.

These links can also be specifically queried using ”/resource/name” with the name which is also stored inside the resource description object, to allow clients to specifically query the path of a named resource. The paths defined in this chapter for the basic service interface can always be accessed to the known paths directly without querying for the path, but additional developed extensions should require the services to query the path, so that collisions between independently developed extensions can be avoided. Also the names should always have a unique extension prefix which includes the vendor to avoid collisions in the names. Lookups of the resource paths can be cached by the clients for the whole service lifetime. An extension might be hotplugged, so it could be added or removed, but it must never change its path within the lifetime to allow for the caching.

Nonexistent resources (also requests to ”/resource/name” in case no resource with this name exists) raise the error 404 not found to allow clients to detect missing functionality (more on errors in 4.3.3).

The URIs are composed with a base URL, which must be known to the client in order to communicate with the knowledge agent. It is always an https URL with usually the IPv4 or IPv6 address or a host name of the knowledge agent. In many cases the knowledge agent will use a nonstandard port, which must also be added unless the standard port 443 is used. A subpath inside the base URL should normally not be needed, but can be done. Examples how knowledge agent URIs to the VSL address ”/agent1/service/someValue” could look:

- https://agent.example.com/ka/vsl/agent1/service/someValue
  global DNS name agent.example.com, port 443, subpath /ka/

- https://192.168.0.1:8080/vsl/agent1/service/someValue
  local IPv4 address 192.168.0.1, port 8080, (no) subpath /

Table 4.1: Mapping of VSL operations to PostOperation names

Operation	**Name**	**Callback?**
notify	NOTIFY	no
subscribe	SUBSCRIBE	yes
unsubscribe	UNSUBSCRIBE	no
lockSubtree	LOCK_SUBTREE	yes
commitSubtree	COMMIT_SUBTREE	no
rollbackSubtree	ROLLBACK_SUBTREE	no
registerVirtualNode	REGISTER_VIRTUAL_NODE	yes
unregisterVirtualNode	UNREGISTER_VIRTUAL_NODE	no

- https://[::1]:8088/vsl/agent1/service/someValue
  loopback IPv6 address ::1, port 8088, (no) subpath /

The next section describes the HTTP methods which can be used with these resources and to which operations they map.

4.3.2 HTTP operations

The HTTP methods (see 2.4.2.2) on VSL tree resources (see 4.3.1) are mapped to VSL operations (see 2.1.2.4) as follows: GET on one of the VSL nodes executes the ”get(address)” operation and PUT executes the ”set(address, node)” operation. The node data gets serialized in the GET response body and PUT request body according to the negotiated content type (see 4.3.4) and the serialization as described in 4.2.1. The address (including parameters) can extracted from the request URI by removing the prefix. These VSL operations conform also to the HTTP an REST restrictions (see 2.4.2.2), so this mapping is acceptable.

The other operations on VLS nodes of the service interface are all mapped to the POST method, using a ”PostOperation” object in the request body to further describe the executed VSL call. The PostOperation object contains the fields ”operation” which is the operation name from table 4.1 and ”callbackId” or ”callbackUrl” depending on the callback technique which is used. The callback techniques are described in detail in 4.3.7 and the specification of callback is only done, if the operation includes a callback parameter. All of the methods don’t return any data, so the response body is left empty using 204 no content.

The operations performed on the additional resources outside of the VSL tree (see 4.3.1) highly depend on the resource and its purpose, of course. For instance, ”registerService(manifest)” on ”/service/register” and ”unregisterService()” on ”/service/unregister” are both using the POST method. The registration call serializes the manifest in the request body and sends the address string in the response body, while the deregistration sends an empty object in the requests and gets no content in the response (the empty object is needed to make it a not simple POST request for CORS; see 4.3.5 for details). Still, the POST method is used as DELETE would be inadequate because the resource still persists and only the service deregisters. The HATEOAS resource routes and all static resurces only support the GET method and the interaction with the other special resources is described where they are defined.

Every resource which supports the GET method also supports the HEAD method, executing excactly th same request just without delivering the content. This is however practically useless with the VSL nodes. The OPTIONS requet is also supported for all of the resources to allow clients to identify the possible interactions. The result of the OPTIONS request does not change depending on the node, if it is executed on the VSL tree. It actually does not even check the existence of the node and does not need to be executed on each of the nodes, as its reponse can be seen as valid for all nodes in the VSL tree.

Using this mapping, all operations of the service interface as listed in 2.1.2.4 are mapped to corresponding HTTP methods using the resource path scheme from 4.3.1. The next section describes the error handling and the used HTTP error messages.

4.3.3 Error handling

If errors occur during the execution of a VSL request, a corresponding HTTP error response is generated (see table 2.3 on common HTTP status codes). This of course happens if the corresponding HTTP error conditions directly apply (for example: ”400 Bad Request” if the request is actually malformed), but also if a VSL exception is similar to the HTTP error condition (for example: NodeNotFoundException being represented as ”404 Node not found”, as the error message is customizable in HTTP). Other internal exceptions which do not have an equivalent error in the HTTP protocol are mapped to ”500 Detailed error message” (instead of ”Internal Server Error” the actual error message is provided, if possible).

Additionally, in order to allow detailed parsing of errors and to improve debugging, the response body of the error message contains an ”ErrorMessage” object. This object contains the error ”code”, ”message” and ”type” being the Java exception type. Stack traces are omitted to prevent leaking of sensitive information in production environments and to save bandwith. The object is then serialized using the negotiated content type (see 4.3.4), if the content type negotiation could already take place, otherwise JSON is used as a default fallback. Outside of the VSL tree (the ”/vsl/” resource path, see 4.3.1), the error message may be presented as HTML if the content negotiation leads to an explicit allowance of HTML. This can be used by static file resources which present a web interface.

Using these ”ErrorMessage” objects, clients can automate the interpretation of errors automatically and even if clients don’t parse the object, the HTTP status code will always reflect that an error occured and at least what generic kind of error it was. The next section defines how content negotiation is handled by the server and clients.

4.3.4 Content negotiation

The content negotiation is based on HTTP’s content negotiation feature (explained in detail in 2.4.2.1), but using some extended rules for cases in which multiple formats are acceptable.

Generally the service interface contains objects in the HTTP bodies, which are serialized using one of the serialization formats from 4.2 or potentially a future extension might even add more formats, which are not yet considered. A client can accept as many of these formats as it wants and also put a weighting on the different formats to express its preference. The server can then choose from these formats, honoring the clients preference and its own preference if the client did not prefer one of the top choices to the other.

More special conditions apply if a client did either not include an accept header (which means accept anything) or it accepts ”*/*” as the highest supported format. For example some browsers send accept headers like ”text/html,*/*;q=0.8”, and as ”text/html” is not provided as a serialization format, the ”*/*” would be considered. This would allow the server to choose any format of its choice according to the HTTP specification, but the problem is that these clients are not able to parse data of any kind. Especially to save bandwidth, many servers would have a preference on the binary formats like CBOR or protobuf, but especially protobuf is not even self-descriptive. So these are only a good preference if the client explicitly accepts these formats, which means it would also have the ability to actually parse them. To prevent this issue, the server has to prefer JSON as the fallback format, if the client does not specifically accept one of the supported formats. This implies, that every server implementation (i.e. the knowledge agents) must support JSON if it uses this HTTP interface.

It is usually advisable to have a lot of supported formats on the server side and only the needed ones at the client side. The question which remains is, in what format a client sends its data for example with PUT or POST operations, because no content negotiation happens with the request body. The solution is that a client may use any format which was previously accepted by the server, for example on a GET request. This implies that a server must also accept data in the format in which it delivers data, which usually should not be a problem but it also implies that serialization formats may only be used if they allow for complete serialization and deserialization of all objects. If the client did not have any content negotiation beforehand, it coud either try its preferred format and try another one if ”415 Unsupported Media Type” is replied by the server or it uses JSON as a safe fallback. This JSON fallback is also relevant in KA to KA communication, which is described in detail in 4.3.8.

The choice for JSON as the fallback is based on its self-descriptiveness, human readability (useful in browsers) and its very high popularity, especially with RESTful HTTP services. A client is however not required to accept the JSON format at all, but it then needs a connection to a knowledge agent which is able to accept the format, which the client is using.

The used MIME types of the formats are those from table 2.2, with one exception for protbuf: Normally, protobuf does not have a real MIME type and uses ”application/octet-stream”, but this also applies to other binary formats, which is why a specific type for the explicit negotiation of protobuf is required. Looking at some discussions on Stack Overflow [90] [91], the usage of ”application/x-protobuf” or application specific MIME types is quite common. To identify also the proto files which are needed for the communication (i.e. those for the VSL objects as described in 4.2.3), I decided to use ”application/x-protobuf-vsl” to clearly identify the format in the content negotiation.

Static resources (see 4.3.1), if they exist, are only delivered with their respective MIME type and no transformation is done. If the type is not acceptable based on the clients ”Accept” header, the HTTP error ”406 Not Acceptable” is used. The JSON default does of course not apply to these static resources.

The last exception is done for requests to ”/”. It should devliver the HATEOAS resources index (see 4.3.1) to API clients but an ”index.html” (or similar) static resource to browsers, which is solved using these rules: If the client includes ”text/html” or the MIME type of the index document (could also be ”application/xhtml+xml” for example) in its Accept header, it is delivered. Of course the index document must not have a type which is also a serialization format. Otherwise the ”/resources” index is serialized using the format which is negotiated using the aformentioned rules.

The next section describes the usage of important HTTP headers in detail.

4.3.5 HTTP headers

Some of the important HTTP headers (see 2.4.2.3) require special considerations for the design of the HTTP service interface. The most important consideration is to secure Cross-Origin requests (Cross-Origin Resource Sharing, CORS, see 2.4.2.3). Additionally, the Cache-Control header can be used to prevent caches from answering requests on the behalf of the knowledge agent and to prevent insecure storage of private data.

The caching of all VSL data should be denied with ”Cache-Control: no-cache, no-store” (see 2.4.2.3). Including ”Pragma: no-cache” is only needed if a HTTP 1.0 cache could be used, but given our TLS 1.2 usage with client certificates (see 4.3.6) effectively only allows caches which also support TLS 1.2 and they are extremely unlikely to be HTTP 1.0 only, given the ages of these technologies. So the ”Pragma” header is considered redundant in this case. Static resources (see 4.3.1) can usually be cached and don’t require specific authentication, so to them a ”Cache-Control: public, must-revalidate” could be applied. It is up to user configuration and the specific implementation if such fine tunings are desirable.

The Cross-Origin request handling is done by having a whitelist of acceptable origins configured in the knowledge agent. Whenever a client sends a request which includes the ”Origin” header, the CORS specific headers are included in the response. If the origin of the client is whitelisted, it is sent in the response ”Access-Control-Allow-Origin” header, otherwise one of the whitelisted origins is mentioned to provoke a CORS violation. The allowed methods are always set to the same as in the normal OPTIONS ”Allow” response and ”Access-Control-Allow-Credentials: true” must be used to allow the SSL client certificate. The allowed and exposed headers can be adjusted to the actual client need, also the maximum age can be set to a suitable value depending on how likely the allowed origins will change. Using multiple origins with whitelisting actually raises a chaching issue [68], which is solved with the header ”Vary: Origin” in the response to make the cache aware of the reliance of the response on the actual origin. These rules mainly apply to the handling of the preflight request (OPTIONS), but also to the actual request except that only the response headers ”Access-Control-Allow-Origin” and ”Access-Control-Allow-Credentials: true” are needed in the actual request’s response.

These headers deal with the proper allowance of valid Cross-Origin requests, but it is also required to consider invalid Cross-Origin requests as well for security. If the request requires a preflight, the actual request will not be performed because the preflight response does not allow the origin. But there are so called simple requests [68], where the actual request is executed without prior preflight. The response is only exposed to the origin if it is afterwards allowed, but the tricky part is, that the request was already executed and only the answer is not exposed if the origin is denied. This is why it is important to check, that simple requests are not able to execute privileged operations but only read data which is then not exposed.

The request is simple if it is a GET, HEAD or POST, so for GET and HEAD it is acceptable because they never change a resource and the answer, i.e. the actual data, is not exposed. POST however is very dangerous, but it is only a simple request if the included request body has a Content-Type of ”application/x-www-form-urlencoded”, ”multipart/form-data” or ”text/plain” [68].

It is therefore important, that every POST requires a request body object (even if it is just an empty object) with a proper serialization format which must not be one of these above. Not obeying these rules would lead to Cross-Site Request Forgery (CSRF) vulvnerabilities [92], as after a user once accepted the client SSL certificate in a browser tab, malicious websites in other tabs could execute simple Cross-Origin requests to modify for example a node in the VSL.

The next section now defines how TLS is used, when the client authentication is required and how the services are identified using their certificates.

4.3.6 TLS usage and authentication

The usage of TLS is always required to use this service interface, a plain HTTP version is not offered. Additionally, to ensure the usage of modern and secure encrpytion methods, the usage of TLS 1.2 or higher is mandatory. The requirements are derived from those defined in the HTTP/2 protocol specification [65], except that ALPN (see 2.4.1.1) is only required for HTTP/2. Additional rules for the TLS usage and how the X.509 client certificates are used for authentication is now described in detail.

TLS has suffered from a long list of security issues due to bad configuration of the servers or the improper usage of extensions like the TLS compression like already mentioned in 2.4.1.1 and [58] [59]. This is why additionally to the forced usage of TLS 1.2 or newer, features like compression or heartbeats should generally not be used. Also the usage of perfect forward secrecy (see 2.4.1.2) is highly recommended to reduce the impact of a private key compromise. Certificates should have at least 2048 bit RSA public keys and only the secure hash family of SHA-256 and newer is allowed. These rules need to be frequently updated to reflect the current state of research, but also potential compatibility issues with older devices may arise. In question, having a minimal security level which is considered secure at the current state of the art is more important.

For the authentication, a whole DS2OS instance uses one CA certificate which is known to all services and agents (see 2.1.2). Every service and agent gets an individual X.509 certificate (see 2.4.1.2) for authentication towards the others. During the TLS handshake, the server has to provide a valid SSL certificate and demands a certificate from the client. The connection is actually allowed if the client does not provid a client certificate, but only special unprivileged operations are allowed in this case. These include the OPTIONS request and the fetching of static resources (see 4.3.1) if they allow to be publicly fetched. Another exception is the establishment of WebSocket connections (see 2.4.5), as some clients do not properly support client certificates. In this case the client has to provide a token in the Authorization header of type ”WSAUTH” (see 2.4.2.3), which it previously fetched using a POST operation with an empty object in the body to ”/websocket/auth-token” while using the client certificate there. This workaround is added to overcome the shortcoming of some Websocket clients. The server must store the X.509 certificate information per token which it generates at least for the short time till the token is used. The client always has to request a new token directly before establishing the Websocket connection. More on the usage of the Websockets is discussed in 4.3.7.

Either way, the server always has access to the X.509 certificate information when accepting a request which is not one of the public resources or operations. In the X.509 certificate, the common name is used to identify the service's or agent's name and additional fields provide this extra information:

- **IsKA** (OID 1.3.6.1.4.1.0): true or false whether this is a KA. Needed to safely identify KAs which may use the extra methods of the KA to KA communication specified in 4.3.8.
- **Manifest** (OID 1.3.6.1.4.1.1): the service manifest hash, if this is not a KA.
- **AccessIDs** (OID 1.3.6.1.4.1.2): the service access IDs, if this is not a KA.

The mentioned OIDs are currently taken from a development range and should be replaced with OIDs which are officially registered with the IANA.

The next section defines how callbacks are registered and invoked and how the callback's state is maintained.

4.3.7 Callback handling

Based on the analysis of callback techniques in 2.5.5, the two recommended methods for callback handling is either following the ”Double server” approach (see 2.5.1) or to establish a stateful connection (see 2.5.3) like a WebSocket (see 2.4.5).

As the WebSocket protocol builds up on HTTP it can be very well used in this scenario. Also the double server approach can be used, if its constraints are acceptable to the client. This leads to the first design decision, that it is up to the client to decide how it wants the callbacks to be handled.

This section is further devided into four parts: The first part illustrates the high level design of the callback invocation following the RPC architecture (see 2.2.4). Afterwards, the high level management of the callback state is discussed for services and the Knowledge Agent. The third part specifies how the callback invocations using the double server approach works. Finally the fourth part specifies the usage of a WebSocket for the callbacks.

4.3.7.1 Callback design

The callbacks follow the design paradigm of the Remote Procedure Call (RPC), in the reverse direction, i.e. the server invokes a client procedure. During a PostOperation (see 4.3.2), the client either specifies a ”callbackId” (which is the procedure identifier) or a ”callbackUrl”. If the client specifies a URL, the double server technique is used and the knowledge agent relies on this URL to be available for future invocations of the callback. If the client specifies a callback identifier, it has to establish a WebSocket for the callback to be invoked.

The general invocation could be based on a standard RPC protocol like XML-RPC (see 2.4.6) or SOAP (see 2.4.7), but the methods are all quite similar and so a custom RPC message is the easier approach. The callback is always invoked using a ”CallbackInvocation” object (no matter which of the functions is used - they feature only two kinds of parameters anyway: an address and in case of virtual node set the node data), and then replied to with a ”CallbackResponse” object.

The ”CallbackInvocation” object has the following fields: ”invoke” the method to invoke, ”address”, ”identity” is the identity of the requesting service which caused the callback to be invoked and in case of a set operation on a virtual node, the node data as ”data”. The ”CallbackResponse” object has the field ”error” which is an error message like in 4.3.3 and only set if the callback failed and ”data” which contains the response data in case of a get operation on a vitual node.

Depending on the callback technique, more fields are used on this object but this is the common base of all callback implementations.

4.3.7.2 Callback state maintenance

The Knowledge Agent has to maintain the callback state, as all other requests of a service are stateless and the service may suspend during operation, so the callback can be unreachable without the service deregistering from the Knowledge Agent. In order to clean up unreachable callbacks, everytechnique allows to check the reachability of the callback frequently and the Knowledge Agent cleans up the callback. In some cases, the callbacks unreachability is only discovered during the invocation, usually with a timeout. As this slows down the operation because they are waiting on the callback to timeout, the state maintenance is important to reduce occurances of timeouts.

Once a timeout occurs or a callback is known to be dead, the transport module tells the Knowledge Agent that this happend and the Knowledge Agent can take the measures like removing the callback. Clients are allowed to reconnect if they use a WebSocket, so new WebSocket connections which are made within the callback timeout lead to an invocation of the callback on the new channel. More on the WebSocket handling in

4.3.7.4.

4.3.7.3 Callback invocation via URL

The callback invocation via URL following the double server technique is very easy: The Knowledge Agent executes an HTTP POST operation on the URL with the ”Callback-Invocation” object in the request body and gets back a ”CallbackResponse” object. An error during the callback execution is indicated using the ”error” field in the response, not using the HTTP status code. Getting an HTTP error is considered a failure of the reachability of the callback, not an error during invocation (which is technically still an invocation which worked).

Callbacks must be managed by the client so it actually knows via the URL which callback to invoke. The client is advised to use the paths below ”/callbacks/” for the callbacks. If the client is another KA, it has to store it under ”/callbacks/” but does not enumerate callbacks in the resource index (see 4.3.1). More on callbacks between KAs in 4.3.8.

Keep-alive checks can be done using HTTP OPTIONS method, to detect unreachable callback URLs early.

4.3.7.4 Using a WebSocket for callbacks

If a WebSocket is used to transfer all callback invocations, first of all the ”CallbackIn-vocation” and ”CallbackResponse” objects need to be extended to perform request and response mapping. For this purpose, the two fields ”callbackId” which is the identifier defined at registration and ”serial” which is a serial number used by the server to match request and response.

A client usually uses one WebSocket for all callbacks, identifying the callbacks by id. If a client has to reconnect, it can simply establish a new WebSocket. The server must resend all outstnding callback invocations using the same serial, so the client can do deduplication. The callback timeout is always awaited even if no WebSocket is established, to avoid races between socket establishment and callback registration. To avoid always hitting timeouts wih dead clients, the Knowledge Agent does frequent keep-alives using WebSocket ping and pong messages (see 2.4.5). If a client did not establish any WebSocket connection for the duration of the callback timeout or if all WebSockets are found dead by alive ping for that amount of time, all WebSocket callbacks of that client are considered dead.

The client is identified only by its SSL client certificate which it uses for the WebSocket connection. If a client establishe multiple connection (for example during roaming), the Knowledge Agents sends callback invocations via all connected WebSockets and treats the invocation as successful once one of the connections sends a response (with or without error). This can lead to issues if the certificate is used multiple times by different services, which should be avoided.

Another issue with the WebSocket is the head of line blocking problem similar to the case of HTTP/1.1 vs. HTTP/2 (see 2.4.3). It actually only occurs during transmission of data in virtual node get and set operations, as otherwise the message is very small and no further steps are needed. The Knowledge Agent can decide if at a certain data size or other criteria, it wants to deliver or receive the data not inside the message, but via a **drop zone**. In case a drop zone is used, the ”data” field is replaced with a ”dropzone” field containing an URL to the drop zone at the calling Knowledge Agent. The URL is under the subpath ”/dropzone/” and not listed in the resource index (see 4.3.1). Per invocation, the knowledge agents reates a sub URL of the drop zone where the client can receive the data for virtual node set from the invocation using HTTP GET or deliver the whole ”CallbackResponse” object using HTTP PUT on virtual node get invocation. Only the virtul node get and set callbacks may use the drop zone. The HTTP operation on the drop zone can be executed in parallel to the WebSocket and does not block the callback channel in this case. The Knowledge Agent has to authenticate the client which acceses the drop zone using its SSL client certificate.

4.3.8 Extensions for KA to KA communication

The specified service interface is also usable for the communication between Knowledge Agents, using these small additions.

To authenticate a service in a KA to KA request which is forwarded on behalf of the service, the KA’s internal ”ServiceIdentity” object derived from the client certificate is serialized as JSON, base64 encoded and put into the HTTP Authorization header (see 2.4.2.3) with type ”VSL”. As there is no content negotiation possible there, JSON is always used. KAs can trust each other to have performed sufficient authentication with the service, which is part of the VSL middleware design.

Two additional requests are needed for the KA to KA communication only: the handshake and the request for KOR updates. These are executed as usual using HTTP POST on ”/ka/handshake” and ”/ka/requestKorUpdate”’.

As KAs need an HTTP server implementation anyway, they can use the double server pattern for callbacks, but they are also allowed to use a WebSocket (see 4.3.7). Additionally, KAs may use the virtual node notify operation instead of a provided subscription callback, if a KA subscribed.

For the KA to KA requests, content negotiation can be used as usual but every KA has to accept JSON, not necessarily as the favorite, but it has to be included in the Accept header and the other KA must accept it if the negotiation does not yield a better choice that works for both.

4.3.9 Usage of other RESTful protocols

The specification of this chapter is mostly focused on HTTP, but a transfer to other RESTful protocols like for example CoAP (see 2.4.4) should be easy to perform based on this specification. Especially CoAP already provides a CoAP to HTTP mapping, which should be applicable for this purpose. In case of CoAP, DTLS would be required instead of TLS for the encryption.

Chapter 5

Implementation

This chapter summarizes how the implementation of the service interface designed in chapter 4 is implemented. First, the Java modules for the Knowledge Agent (see Knowledge Agent architecture in figure 2.1) are described. They are split into the serialization unit (see 4.2) with extensions to support for the four supported formats XML (see 2.3.1), JSON (see 2.3.2), CBOR (see 2.3.3) and protobuf (see 2.3.4) and the transport modules, which are used for the connector and the server side for the KA.

For each of the three required languages (Java, C and Python, see 2.1.3.1), a connector to the VSL is implemented and an example code using the connector is presented. Each of the sections also describes the used implementation approach, libraries and other development tools.

5.1 Java modules

The different modules of the Java implementation are individual Maven submodules of the existing VSL parent project. Maven allows to manage the dependencies of a Java project and its build process using a central repository of the available dependencies and build modules [93]. For the VSL project, each component is an own Maven module to allow modular building and replacement of KA components for different purposes. Using the VSL parent project, the modules can inherit common configuration for the VSL project.

For the serialization with the different formats, the module ”databind-mapper” provides generic functionality. The actual serialization is done using the Jackson library [39] with its databind functionality, which uses Java introspection to automatically read an object’s structure and then serializes it. Annotations allow the fine-tuning of the process and a runtime cache for the object structure is used to speed up subsequent serialization of the same object type. The ”databind-mapper” module supports JSON by default, as it is already included in the Jackson core and also it is anyway the fallback format (see 4.3.4). The other formats are available as extra modules, which are loaded dynamically at runtime by the ”databind-mapper” module, if they are available on the Java classpath.

To allow for the proper serialization of the VSL nodes according to the specification in 4.2.1, some Jackson annotations are needed on the ”VslNodeImpl” class. These annotations are available separately without using the ”databind-mapper” module and are directly added to the main VSL code, as these annotations take no effect without using Jackson and also don’t add much overhead.

The other serialization formats are supported by the three optional modules ”databind-mapper-xml”, ”databind-mapper-cbor” and ”databind-mapper-protobuf”. They use the respective Jackson plugins for XML, CBOR and protobuf generation. Additionally, the XML module is configured to use the WoodStox XML generator, which was already recommended in the related work in 3.2. Simply adding or removing these modules on the Java classpath automatically enables or disables the respective format.

Based on these serialization modules, the actual implementation of the service interface is done in the modules ”rest-connector” and ”rest-transport”. The connector implements the client connector for services and the transport is based on some common code in the connector, but adds the whole server side as defined in 4.3 and extensions for KA to KA communication (see 4.3.8).

The transport uses the Jetty webserver implementation [94] for the HTTP server (see 2.4.2) and the Jetty HTTP client library for the connector. Jetty allows for asynchronous I/O implementations, has client and server extensions for the WebSocket protocol (see 2.4.5), which is used for the callback implementation as defined in 4.3.7.4. Jetty uses thread and resource pools for efficient management of many parallel connections. Support for TLS (see 2.4.1) is already builtin to the Java Virtual Machine (JVM), but also integrated to the Jetty library.

Using the Jetty library, a full HTTPS server with TLS client auth, special handling of the content negotiation (see 4.3.4) and additional headers (see 4.3.5), which also supports callback channels via websockets (see 4.3.7.4), is running. Additionally, the ”rest-transport” module is able to execute KA to KA requests using a modified connector variant and to register callbacks to other KAs.

The next section describes the ”rest-connector” module in more detail.

5.2 Java connector

The ”rest-connector” Java module is also based on the ”databind-mapper” module and uses the Jetty client library only without the server parts (see 5.1). This allows the connector to also load additional serialization formats at runtime using the extension modules to the ”databind-mapper” (see 5.1).

Listing 5.1 shows an example how the Java connector can be used:

Listing 5.1: Example using the Java rest-connector

```java
// wire everything
KeyStore keystore = SSLUtils.loadKeyStore("service.jks", "jkspassword");
JettyContext jettyContext = new JettyContext(keystore, "jkspassword");
JettyClient jettyClient = new JettyClient(jettyContext);
VslNodeFactory nodeFactory = new VslNodeFactoryImpl();
VslMapperFactory mapperFactory = new DatabindMapperFactory(nodeFactory);
VslRestConfig config = StaticConfig.DEFAULT_REST_CONFIG;
VslConnector connector = new RestConnector(jettyClient, "https://localhost:8080/",
    config, mapperFactory, nodeFactory);

// start Jetty thread pool
jettyClient.start();

// do example request
VslNode node = connector.get("/agent1/service/test");
System.out.println("Test value: " + node.getValue());

// cleanup
jettyClient.stop();
```

The next section describes the C connector and its usage.

5.3 C connector

The C connector uses CMake [95] to manage the build process and its libraries in a platform independent way. For the serialization, JSON is used with the json-c library [96], and for the HTTP client, the curl library [97] provides great functionality.

The main part of the project is the ”vslconnector” library, which boxes the whole handling of the connector inside the library and never exposes headers, data structures or functions of the used libraries like curl. This way it is possible to build different versions of the library, which for example use a different HTTP client than curl or other serialization formats like CBOR, without breaking the API or ABI. Also the memory management of the very dynamic VSL node strucuture is completely done inside the library using atomic reference counting and accessor methods for the node’s values and children. This allows future versions to even change the in-memory representation of the VSL node without breaking compatibility.

Listing 5.2 shows an example how the C connector (installed as libvslconnector.so on Linux) can be used:

Listing 5.2: Example using the C libvslconnector.so

```c
#include <stdio.h>
#include <stdlib.h>
#include <vsl-connector.h>

int main() {
    vsl_connector *connector = vsl_new_connector();
    if(vsl_select_ka(connector, "https://localhost:8080/", NULL)) {
        fprintf(stderr, "Error: %s!\n", vsl_error(connector));
        return 1;
    }
    if(vsl_load_identity(connector, "ca.crt", "service.crt", "service.key")) {
        fprintf(stderr, "Error: %s!\n", vsl_error(connector));
        return 1;
    }
    vsl_node *node = vsl_get(connector, "/agent1/service/test");
    if(node == NULL) {
        fprintf(stderr, "Error: %s!\n", vsl_error(connector));
        return 1;
    }
    printf("Test value: %s\n", vsl_get_value(node));
    vsl_put_node(node);
    vsl_put_connector(connector);
    return 0;
}
```

The next section describes how SWIG was used to generate a Python binding for the C connector.

5.4 Python connector

The Python connector is also built by the CMake project of the C connector as a separate submodule. It uses SWIG (see 2.6.2) to create a wrapper library and Python module which load the original libvslconnector.so into the running Python instance. The public interface of the library (from vsl-connector.h) is exposed as native Python methods in this module.

The SWIG generated wrapper automatically takes care of all required management mechanisms for type conversion (for example the C char arrays are native strings in Python), memory management and loading and unloading of the library itself. In case of Python, the effort for the memory management is considerably low and the boxed approach of the C connector eases this even further. The result is a native Python library which exposes the same functions and data types (less relevant as Python does not require explicit types) as the C connector.

Listing 5.3 shows an example how the python vslconnector module is used:

Listing 5.3: Example using the Python vslconnector module

```python
import sys
from vslconnector import *

connector = vsl_new_connector()
try:
    if vsl_select_ka(connector, "https://localhost:8080/", None) != 0:
        raise Exception(vsl_error(connector))
    if vsl_load_identity(connector, "ca.crt", "service.crt", "service.key") != 0:
        raise Exception(vsl_error(connector))
    node = vsl_get(connector, "/agent1/service/test")
    if node is None:
        raise Exception(vsl_error(connector))
    print("Test value: %s" % vsl_get_value(node))
    vsl_put_node(node)
except Exception as e:
    print("Error: %s!" % str(e))
finally:
    vsl_put_connector(connector)
```

The Python scripting style allows for a more compact and readable representation of the above C code, although on the low level it uses exactly the same C libraries, which should also lead to an equal performance.

The Python connector is just one example out of many languages for which SWIG can easily generate bindings (see 2.6.2 for a list), so the required steps to add more language bindings are minimal.

Chapter 6

Evaluation

This chapter presents the evaluation which I performed to verify that the solution implemented in chapter 5 following the design of chapter 4 actually fullfills the requirements of 2.1.3.

For each of the sections, the taken approach is reasoned with a prediction of the expected results. Then the setup of the measurements or the methodology of the qualitative analysis approach is described. Afterwards, the data is summarized, analyzed and interpeted in the context of the previous expectations. To summarize the key results, a small comparison table concludes each of these sections.

The first section evaluates the four serialization formats, which are described in 4.2, and compares them to provide further guidance when which of the formats is preferrable. Two measurements analyze the size of serialized data to discuss the overhead of the format and analyze the serialization performance with the provided Java implementations (see 5.1) to get insights into the required processor power for each of the formats. Based on the results of these measurements and the conducted assessment in 2.3.5, recommendations for the usage of the different formats with different use cases are given.

The second section performs a qualitative evaluation of the whole service interface design by first reviewing the requirements from 2.1.3. Then the effort which was required to implement a connector in another programming language using the provided interface design from 4.3 is evaluated. Also the practical benefit gained from using additional tools like SWIG is reviewed in this part.

Now the evaluation of the serialization formats is presented in detail.

6.1 Evaluation of data serialization formats

The four serialization formats XML (see 2.3.1), JSON (see 2.3.2), CBOR (see 2.3.3) and protocol buffers (see 2.3.4) are compared in this section using measurements of the overhead and the serialization speed. These measurements are very similar to those performed by the related work on serialization which is presented in 3.2, but using the actual VSL node datastructure from 4.2.1. As this data structure is the only payload of the main operations of the Virtual State Layer, the actual efficiency when using this particular data structure is very important, wich is why these measurements are conducted with all four candidates.

Depending on the use cases of the services and smart devices which are eventually using the format, the smallest size on the wire or the lowest power consumption during serialization might be more criticial, while in other cases a very common format might be easier to support or human readability might be wanted for debugging purposes. Based on the results of the measurements, the subsection 6.1.4 provides an overview when each of the formats is suitable or not recommended.

The measurement setup and the expected results are presented first, with a comparison to the setups and results of the related work. Afterwards the results are presented and analyzed and then a result assessment concludes this section with specific recommendations when to use which of the formats. Now the setup is explained in detail.

6.1.1 Measurement setup

The VSL node data structure can host a wide variety of information, depending on what was actually requested from the middleware. The different types of information stored in the node are described in 4.2.1. Additionally, a node may contain child nodes, which may even contain further child nodes, depending on the information depth that was requested (see 4.3.1 for the address syntax with parameters like depth).

In order to test different variants of nodes, the following information is either present or left out of the example node data:

- **Value**: the actual value of the node, which can be data payload of any size. It is tested with a small and a considerably big value (1MB, random text).
- **Metadata**: additional metadata of the node, which can either be explicitly requested or sent together with the value. The metadata's complexiy can also vary, for example a simple type would be "/basic/number" and a complex type can be derived multiple times and have long type names. Also the restrictions could be just one restriction or a set of many restrictions that apply.

Table 6.1: Example VSL nodes for serialization measurements

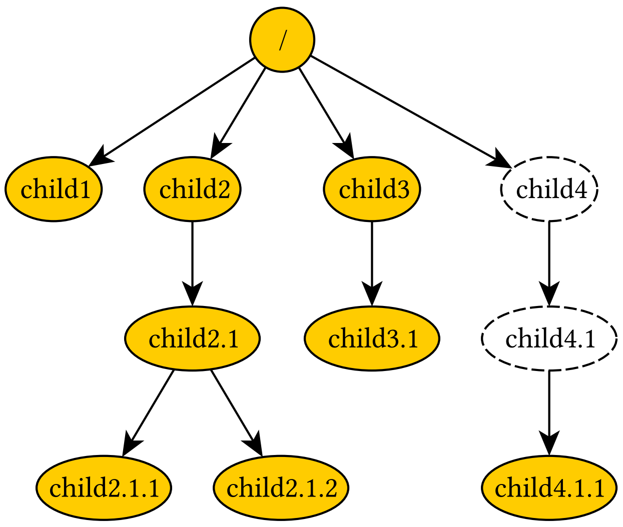
#	**Name**	**Data (value)**	**Metadata**	**Children**
1	simple data node	simple value	none	none
2	big data node	big data (1MB)	none	none
3	simple node	simple value	simple metadata	none
4	metadata node	no data	complex metadata	none
5	big node	big data (1MB)	complex metadata	none
6	simple data structure	simple value	none	child structure
7	big data structure	big data (1MB)	none	child structure
8	simple structure	simple value	simple metadata	child structure
9	metadata structure	no data	complex metadata	child structure
10	big structure	big data (1MB)	complex metadata	child structure

- **Children**: the node can additionally have child nodes, which ususally contain the same data as the parent (so also metadata only, data only, etc.).

These different possibilities are combined to ten example node structures, which are listed in table 6.1. The child structures which are added to some of the examle nodes always use the structure presented in figure 6.1. The filled nodes in this figure are the children which also contain the same data and metadata as the parent node (denoted with "/" in the figure). The nodes which have a dashed line around them and no filling are just path components without any data or metadata. This can happen in some cases like for example if only data is requested and the nodes do not have data (in this case, the node would have metadata which was not requested) or if the requesting service does not have the access rights to query the node itself, but has access rights on one of its children. In order to construct a node's path, the separator "/" is added. So the path of "child2.1" in figure 6.1 would be "/child2/child2.1".

Each of the ten example nodes from table 6.1 is tested separately with each of the four serialization formats. This way, the impact of the data to metadata ratio can be analyzed with each of the formats and it is possible to spot performance problems that arise with specific elements of the node like the child structure or having complex metadata. With the 1MB data chunk in the big (data) nodes, specific problems that arise when handling larger data structures can be identified, although this data chunk is intentionally still small enough for efficient in-memory storage on the test system, as VSL nodes should currently not contain data chunks that are too large to handle for the services which use these. A streaming extension is considered to handle node values, which are larger than the memory of the services which use it, in future versions of the VSL.

Figure 6.1: VSL child node structure



With regard to the related measurements of serialization formats that are presented in 3.2, there are multiple approaches to measure and compare the overhead and throughput or serialization speed. For the overhead, I compare the absolute sizes of the reference data structures for the four evaluated serialization formats and the relative sizes compared to one reference format. As the reference format I choose JSON because it will most likely have an average size and is highly popular and therefore also generally a good reference. The final recommendation table lists the average of this relative overhead averaged across all ten example data structures.

The throughput or serialization speed can be defined in many different ways, as also the two related works show (this is described in more detail in 3.2): One measurement, [31], used the raw data size per second as the throughput metric, while the other, [84], measured the time to serialize 500 instances of the same reference object. For my measurements I use the JMH framework in Java [98], which provides very detailed support for benchmarking of Java operations, inluding things like JVM forking for multiple runs, warm-up phases to get Java's code cache and run time optimizations applied and different measurement and data collection variants. The mode which I use for the performance measurements executes several iterations of each of the benchmark methods which each use one of the example structures with one of the formats, writing the content to a preallocated byte array. This way only the actual serialization time without any I/O performance impact is measured. JMH is configured to run the method in a loop for a whole second in each of the iterations and then average then needed time per serialization call in microseconds per iteration.

The measurements were performed on a laptop with an Intel® Core™ i7-4750HQ CPU

@ 2.00GHz with 16 GB DDR4 memory. The Linux 4.4.50 system was idle except for the benchmark, which was performed on Java 1.8.0_121 using an icedtea 3.3.0 build.

6.1.2 Expected results

I expect the results to be similar to the related measurements from 3.2. The overhead of the formats ranging from highest overhead to lowest will be: XML, JSON, CBOR, Protobuf. For XML I expect a massive overhead, while JSON and CBOR will be quite ok with CBOR being smaller because it is binary. The protocol buffers will be slightly smaller than CBOR, because the provided schema replaces the self-describing parts which are still needed in CBOR.

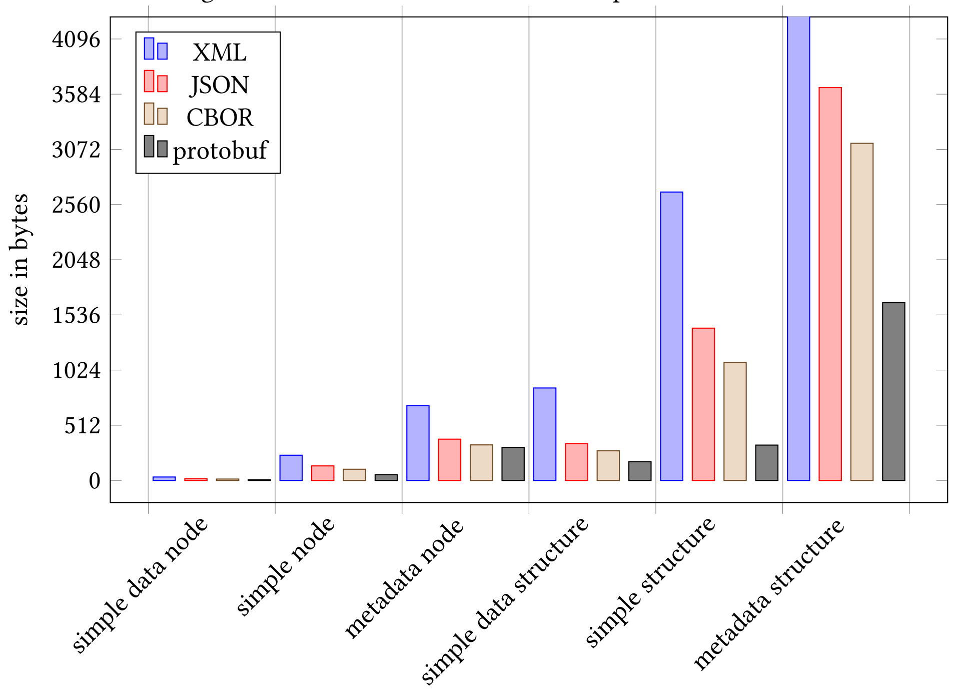
For the serialization performance, I expect the Jackson library and the WoodStox XML generator to perform overall very well after the JVM warm-up, as they utilize runtime code generation and structure caches to enhance repeated serializations of the same data structures, even with different data values. The complexity of XML will likely make it the slowest format, with JSON being already way faster than XML, but still significantly slower than the binary formats. The two binary formats will likely perform very similar, although protobuf could be faster because it generates less output data and the actual data copying can already have an impact at that level.

6.1.3 Measurement result analysis

The first surprising result that came up during the benchmarks is an instability in the Jackson protobuf implementation, causing exceptions on successive invocations (not the first invocation) when serializing the structures with big data, i.e. big data structure and big structure. I tried different versions of the library but the problem persisted. The size of the protobuf serialized structures can be measured as the first invocation succeeds, the performance measurement of these two out of ten structures was not possible on protobuf. Additionally, this is by itself already a result and the project page of the Jackson protobuf plugin states: ”With release 2.6.0 (Jul-2015), this module is considered stable, although number of production deployments is still limited” [99]. So it seems to be not mature enough for this application. Other libraries for protobuf were not tested but are likely to produce more stable results.

The total run time of the benchmark was almost two hours (01:57:28). The results are now presented in detail, starting with the measurements of the overhead.

Figure 6.2: Absolute sizes of small example VSL nodes



6.1.3.1 Results of the overhead measurement

Figure 6.2 shows the abolute sizes of the four measured formats for all example data structures that do not contain the big data, as otherwise the scale would be too large. For example, the values of the simple data node are: XML 32 bytes, JSON 16 bytes, CBOR 13 bytes and protobuf 6 bytes. The value of the metadata structure for XML was too large for this graph, it is actually 6816 bytes compared to 3645 bytes of JSON.

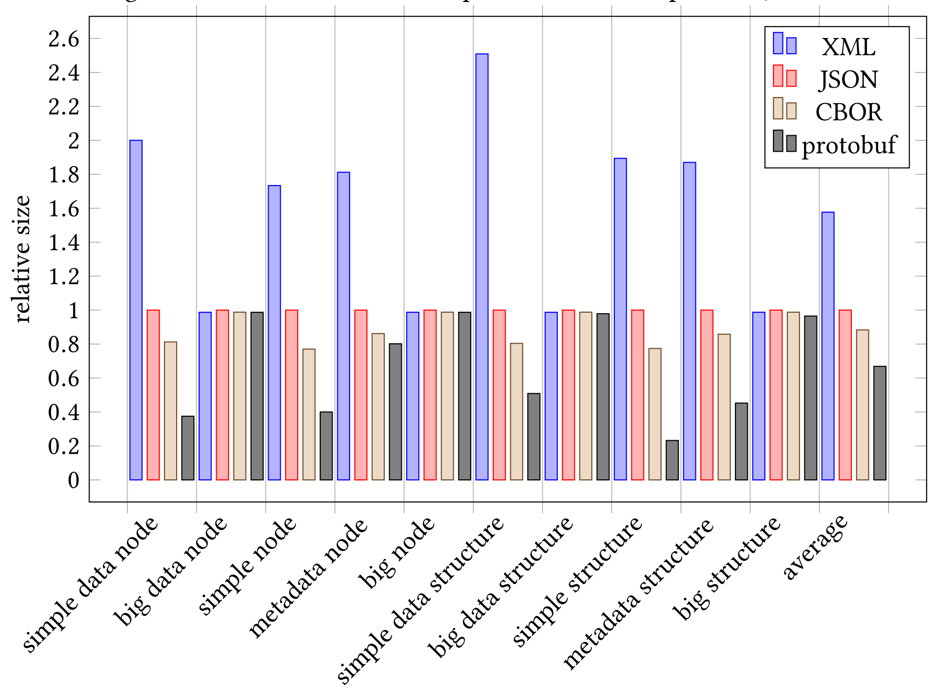
What the measurement clearly shows is that the amount of structural information leads to increasing differences between the serialization formats. So the more metadata or child structures are involved, the differences between the formats get bigger. It is also clearly visible that without large payloads, the formats clearly order from XML with the highest overhead, over JSON and CBOR being quite close yet CBOR always being a bit smaller, to protobuf with the lowest overhead.

The examples with big values stored inside the node are listed as raw data in table 6.2. Intersting is, that JSON and CBOR perform worse because they need to escape special characters like line breaks. This even applies unexpectedly to CBOR, which still performs better than JSON but still worse than XML. XML can embed almost any string data, as long as it does not contain "<", without escaping. Protobuf was again the format with the lowest overhead, but shows an odd result with the big data structure and the big structure: although the big data structure contains less metadata, its total size is

Table 6.2: Absolute sizes of big example VSL nodes

Name	XML	JSON	CBOR	protobuf
big data node	1048604	1062206	1049375	1048580
big node	1049285	1062588	1049703	1048887
big data structure	9438006	9560052	9444533	9355002
big structure	9444135	9563490	9447485	9227356

Figure 6.3: Relative sizes of example VSL nodes compared to JSON

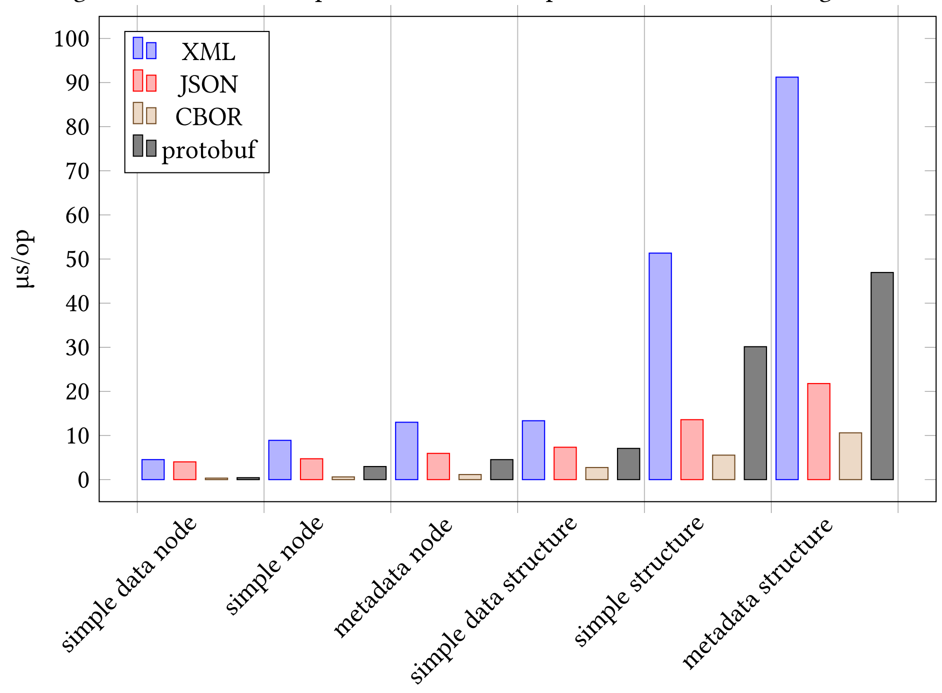


greater, but only for protobuf.

Additionally an evaluation of the relative sizes compared to a reference format is performed. As the reference format I choose JSON due to it being middle-sized and very popular and therefore a good reference. The relative sizes are shown in figure 6.3.

The last column in figure 6.3 shows the average of the relative sizes of all ten example VSL nodes. These avarages will be used in the recommendation table in 6.1.4. The percentual values for the formats are: XML 158%, JSON 100%, CBOR 88% and protobuf 67%. Now the results of the performance measurement are presented.

Figure 6.4: Serialization performance of example VSL nodes without big data

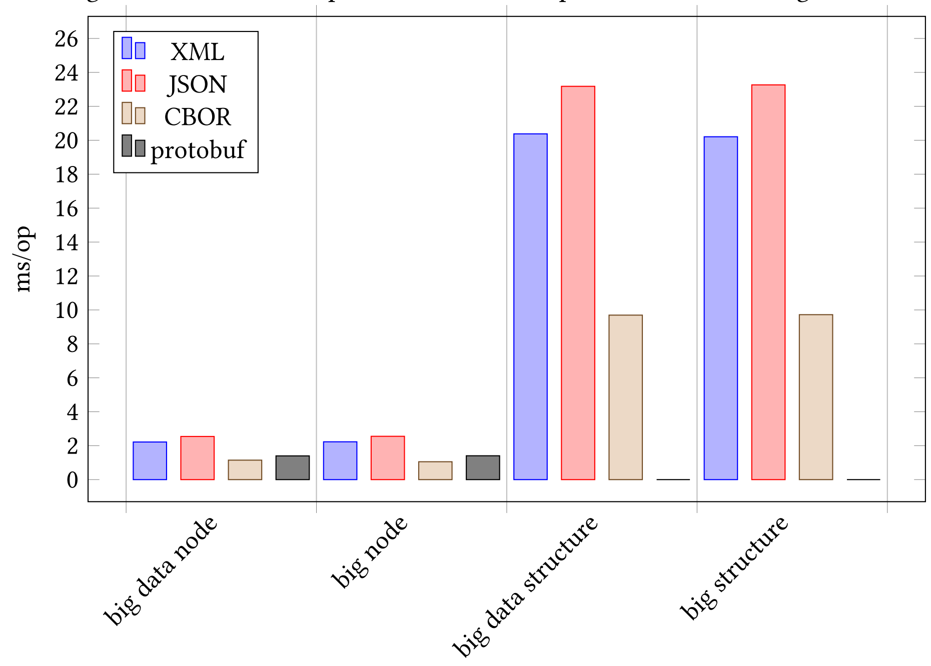


6.1.3.2 Results of the serialization performance measurement

The performance measurement effectively performed 100 samples per serialization format and data structure after the JVM warm-up. Each sample ran a loop to serialize the data structure for one second into the same byte array, to avoid reallocations of the I/O buffer, and then the average serialization time in microseconds was averaged per sample and then the average of these 100 samples are used. Figure 6.4 shows the results for the data nodes and structures without big data and figure 6.5 shows the results for the big data nodes and structures. Note that in figure 6.5 for the big data structures, protobuf results are missing due to the aforementioned bug in the library. The confidence interval of all values is less than +/- 1.5 %, which means that even the minima and maxima are so close to the average value, that a further look at the value distribution does provide valuable insights.

The actually most impressive result is the extreme performance of the CBOR serialization, exceeding all other formats significantly. For example the simple node was serialized with CBOR in 0.613 µs, while JSON needed 4.729 µs and protobuf needed 2.964 µs. Also interesting is that JSON even takes longest (even longer than XML) when serializing the big data nodes, most likely because of the need to properly escape the string, while XML needs way less escaping. This can be seen in figure 6.5, while for all other data structures in figure 6.4 JSON is always faster than XML. But this effect could also be just caused by the output size, as the size measurement in 6.1.3.1 already

Figure 6.5: Serialization performance of example VSL nodes with big data



revealed a similar effect: While JSON is usually more compact than XML, in case of the big data nodes, it actually uses more space than XML.

Protobuf also shows an interesting result in figure 6.4: Usually it is performance wise in between of JSON and CBOR, but with higher complexity of the node (structure instead of just one node) and higher complexity of the metadata (especially metadata structure), it gets even worse than JSON and relatively performs worse than it did with the simpler structures. This could maybe be already related to the bug in the library, although it did not occur even once with these small structures, or it is related to the way how protobuf handles recursive data structures. Anyway, even for the simple data node, CBOR already proofs that it is the better choice for those aiming at lowest power consumption.

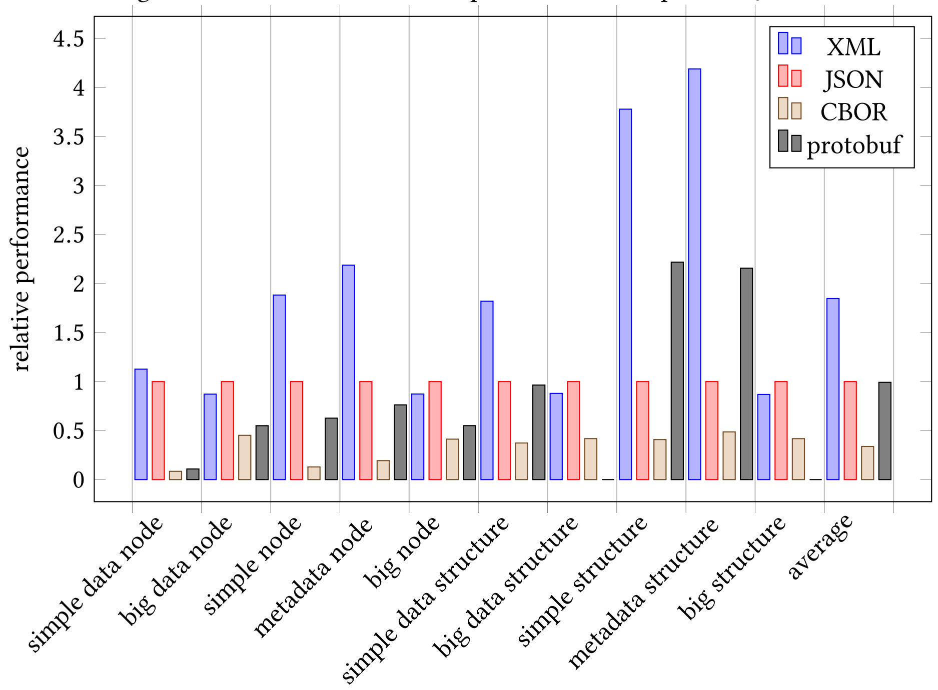
As for the sizes, also the performance is shown relative to the reference format JSON in figure 6.6. Also in this graph the two values for protobuf which could not be measured are excluded from the comparison and also from protobuf's average value, which is why this value has to be taken with care in case of protobuf.

6.1.4 Result assessment and recommendations

The presented results from 6.1.3 mostly match the expectations from 6.1.2.

For the sizes, the predicted order from XML being the worst to protobuf being the best is also the order which I actually measured. One small exception is related to the

Figure 6.6: Relative serialization performance compared to JSON



rather uncommon use case of very long 1 MB data strings inside the nodes, where XML surprisingly performed better than JSON due less escaping in the value.

The performance evaluation showed that protobuf does actually not perform comparable to CBOR but actually way worse and even on a similar level to JSON. Apart from this, the expectations were met, as all formats performed quite good due to the high level of optimization in the Jackson and WoodStox libraries and the performance of XML, JSON and CBOR being in the expected order although CBOR suprised with even better values than expected.

The protobuf implementation shows serious issues and also some odd results like the smaller size of the big structure compared to the big data structure (see 6.1.3.1 and table 6.2). Also some performance issues with node structures could be related to the implementation instead of the format in general (see 6.1.3.2). At its current state the implementation is too unstable to be actually used.

Compared to the measurements from the related work in 3.2, most of the results are in line with the expectations which I derived from their results except for the protobuf results. The results were reproduced for simple data structures but the more extreme examples of deeply nested nodes and big node values behaved differently for protobuf. Also the importance of a JVM warm-up was visible in the live output during the measurement, showing extreme variations of the values during the first few warm-up iterations.

Finally it can be concluded that JSON and CBOR are both good choices as the serialization format for an interoperable service interface. Their self-descriptiveness, good (or in case of CBOR, acceptable) human readability and a good performance supports extensibility, compatibility and the usage on constrained devices. It can be said that while JSON has a higher usage degree on the web, CBOR can improve the resource usage a lot which is particularly useful for embedded devices.

In a browser environment, for example with JavaScript, or on languages where no robust CBOR implementation exists yet, JSON can be used as a safe default which is always supported. Also the translation of the CBOR data to JSON is needed for debugging, so that humans can actually read the data without additional tools.

Binary formats which use a schema like protobuf can be used to reduce the serialized size even further, but loosing a lot of interoperability and compatibility, which is why their usage should be considered carefully. XML mostly serves for improved human readability but its high overhead makes it a bad choice for on the wire serialization.

In consequence, the implementation of the transport module for the knowledge agent (see 5.1), is configured to prefer CBOR over JSON where applicable, but leaving JSON as the fallback format for cases where CBOR support is not explicitly advertised (see 4.3.4 for details on the negotiation of the formats and the JSON fallback). XML support is included for requests which explicitly want to use XML, but not actively used by the services or in KA to KA communication (see 4.3.8). The protobuf module is disabled and would need to be replaced by a reliable version of it.

Table 6.3 summarizes key results and extends table 2.2 with the measured results and the actual usage of the formats. The next section evaluates the protocol on a real network connection.

6.2 Evaluation of the service interface

The Java modules from 5.1 are already integrated into development builds of the Knowledge Agent and showed a stable behaviour (except for the protobuf module, as already discovered in the previous section). All three connectors have been successfully tested with running knowledge agents to execute several get requests. None of the implementations showed particular deficits in performance or stability during these requests.

The following two subsections evaluate the two main goals of the thesis, the creation of a service interface using standardized technologies and the creation of native connectors. Section 6.2.1 performs a review of the requirements which are defined in 2.1.3. The next section 6.2.2 evaluates the native connectors, how much effort it was to create them and if proper interoperable libraries could be used.

Table 6.3: Recommendations for serialization formats (extends table 2.2)

Format	**XML**	**JSON**	**CBOR**	**Protocol Buffers**
Media type	application/ xml	application/ json	application/ cbor	application/ x-protobuf-vsl
Size of simple node	32 bytes	16 bytes	13 bytes	6 bytes
Serialization time of simple node	8.899 µs	4.729 µs	0.613 µs	2.964 µs
Relative overhead[1]	158 %	100 %	88 %	67 %
Relative serialization time[1]	185 %	100 %	34 %	99 %
Strengths	high human readability	compact text format, good human readability	compact, very fast	very compact, fast
Weaknesses	very high overhead, slow	string escaping needed for string values	young, not so popular	schema needed, not self-describing, broken libraries
Used by the service interface	only when explicitly requested	as a fallback if no specific format is requested or if CBOR is not supported	as the favorite format when available	currently never
Currently supported implementations	Java	Java, C, Python, JavaScript[2]	Java	No stable implementation

1. compared to JSON as the reference format, average value of all tested example nodes
2. JavaScript implementation in the VSL WebUI project, developed by Andreas Hubel

6.2.1 Review of the requirements

The requirements from 2.1.3 are summarized in table 2.1. They are now analyzed individually and what part of the service interface design reflects the requirement.

1. **standardized network protocol**: all protocols which are used in the design (chapter 4) are standardized in a respective RFC of the IETF. Protocol Buffers would be an exception here but they are no mandatory part of the interface and are currently disabled anyway.
2. **interoperability**: all parts of the service interface in 4 are designed independent of programming language or platform.
3. **low overhead of data transfers**: the RESTful design has shown to have a low overhead in the related work (3.3), additionally is the desing not fixed to HTTP, so CoAP (see 2.4.4) could be adopted to further reduce the overhead. The content negotiation feature allows for clients to select a suitable data serialization format, so if low overhead is the main goal of the client it can use CBOR or protobuf serialization following my recommendations in 6.1.4
4. **low latency of a full operation**: using a follow up HTTP request after the first request with HTTP/1.1 keep alive or HTTP/2, a full operation only requires one round trip, sending the request and receiving the response as per definition in the HTTP standard (see 2.4.2). The implementations also reuses existing connections properly. The first operation requires full authentication with the TLS handshake and a TCP handshake, which is a lot more but only for the first request.
5. **simplicity of the implementation**: the next section 6.2.2 evaluates this in detail.
6. **security: encryption, authentication**: the usage of TLS 1.2 or newer with client certificates (see 2.4.1) is required by the design (see 4.3.6) and used in the connector and knowledge agent implementions. TLS provides a very high level of encrytion and secure authentication with these settings.
7. **callback support**: the callbacks were one of the tricky parts during the adoption of the RESTful design. The section chapter 4.3.7 shows very detailed how callbacks be realized, with one simple to use approach following the "Double server" technique from 2.5.1 and one approach which preserves the client server architecture of HTTP using a WebSocket (see 2.4.5 and 4.3.7.4) for the callbacks following the technique with stateful connections from 2.5.3.
8. **asynchronous operations**: are provided by HTTP and even handled very efficiently in HTTP/2 (2.4.3) with frame multiplexing. For the callbacks based on WebSockets, the drop zone provides asynchronous operations of virtual node invocations (see 4.3.7.4).

9. **stateless/suspendable protocol**: a RESTful service interface is by design stateless (see 2.2.5), but if a full TLS handshake is needed the initial request still takes a few more roundtrips, but at least no complex registration process is required.

To summarize, the requirements from table 2.1 are met by the service interface design in chapter 4 and the implementations from chapter 5. The next section takes a deeper look at the native connectors and requirement #5, the simplicity of the native implementation.

6.2.2 Evaluation of the native connectors

The implementaion of the native connectors was an overall straightforward task. For HTTP and JSON, libraries in any programming language can be found within seconds of googling and thousands of examples and other projects using them are available. Especially for C, the process of finding the libraries and connecting the components was surprisingly easy. The C implementation is also considerably compact, the code to execute a GET request and perform the JSON mapping has a total of 68 lines of code handling CURL and JSON-C, although some initialization and other stuff is done in other functions before that.

With using SWIG for the creation of a Python wrapper around the connector for C, there was literally no development effort involved for Python. The SWIG interface definition for the Python wrapper is excactly like the example in 2.6.2, only including the main header file in a total of five lines of code. Even creating the example program in Python was a more complex task (18 lines of code).

So it can be concluded that the implementation of native interfaces for the service interface design is simple and the generation of wrappers for the C library using SWIG a very successful approach for fast extension of the number of supported programming languages.

Chapter 7

Conclusion

In this thesis, a RESTful service interface for the Virtual State Layer middleware is specified and implemented with support for multiple serialization formats based on content negotiation and flexible callbacks using websockets or callback URLs following the ”double server” technique. Service connectors are implemented and tested on different programming languages, using Java, C and Python. With the SWIG wrapper generation, the language binding for Python was automatically created and the same approach can be used with a lot more programming languages.

For the design of the service interface, different architectural principles were considered and the RESTful design was identified as the best fitting architecture. Other architectural principles could be used as well, like the Remote Procedure Call (RPC) architecture, but I think they are less suitable to represent the VSL service interface. I still used RPC in one part of my service interface design, the callbacks. This combination of mainly RESTful design with RPC for callbacks showed to be useful for the design.

As part of the analysis, different formats for data serialization are compared and all of them are available using content negotiation. A detailed evaluation of these formats showed when which of the formats is more suitable for the needs of the service like lower network overhead, less processing power or higher interoperability. The implementation of the Google protocol buffers turned out to be practically too unstable, although it could still be evaluted with small example data structures.

Multiple protocols and callback techniques are analyzed, but based on the architectural decision, only some of them are actually usable for the service interface design. To reach the highest interoperability and to use a well standardized protocol, the choice was made to use HTTP as the protocol for the RESTful service interface. CoAP is an acceptable choice as well, and it can be implemented as future work (see 7.2). The design of the service interface should be easy to reuse for CoAP as well, as CoAP provides a direct mapping to HTTP. CoAP can potentially provide better results in a local network infrastructure, but there are less NAT boxes, proxies, etc. which can deal with CoAP.

Also the number of libraries and existing deployments is far lower than for HTTP, leading to potential interoperability issues.

The most important operations of the service interface are working and have been tested for obvious performance issues. These implementations for Java, C and Python serve as a working base for the further development of services and the specialization of the connectors for particular service needs. Adding more extended or optional features, like more serialization formats for the C connector or improving the handling of special cases with callbacks like the drop zone are the next steps to make the connectors more usable in extended use cases. Especially for the usage of the C connector on embedded devices, the reference implementation's API design should be a good choice, but the actual implementation currently relies on having a kernel, which is not suitable for very small microprocessors. More optimized variants of the C connector can be developed for this specific use case, even using the same API design.

The now completed design specification is the starting point for the development of further connectors in lots of programming languages, with specific platform optimized variants, to allow service development in all common programming languages. The existing implementations can be used to verify the new implementations against this reference design. Furthermore, the creation of the C connector was straightforward, showing that the portability of the interface is very high, especially as many languages have more handy libraries and builtin functions for HTTP and serialization than C.

Finally, the requirements which are listed in table 2.1 are met by the design and implementation. The protocol is usable with the VSL middleware and already integrated into the latest development builds of the Knowledge Agent and connectors.

7.1 Goal achievement and resulting artifacts

Looking back at the declared goals in 1.2, the two main goals of the thesis were to design and evaluate a service interface with standard network protocols for the Virtual State Layer middleware and to implement and evaluate native connector implementations for other languages than Java. These two goals were fully achieved, with the extensive design in chapter 4 which uses only thoroughly standardized and interoperable technologies and with the native implementations of C and Python connectors.

What I did not do is to provide full implementations of all possible combinations of RESTful low level network protocols, serialization formats and programming languages, as all these combinations expand to a very large list. Those which are implemented provide some of the most common combinations with reasonable choices to prove that it is possible to implement all of these combinations, given enough time and effort. Also the testing of the different operations and especially callback methods in different networks, like mobile networks, wireless LAN, etc., would provide valuable insights for which network conditions which of the choices are more appropriate. Detailed testing in real smart space deployments is therefore valuable future work.

This list concludes the main **resulting artifacts** of my thesis:

- **Specification of the RESTful service interface** for the VSL: specified in detail throughout the whole chapter 4, providing detailed information on aspects like the serialization using different formats (4.2) and the protocol specification (4.3) with amongst other things specifies approaches on callback handling (4.3.7).
- **Java reference implementation**: provided for the Knowledge Agent and services, the Java reference implementation can be used to test other connectors and of course it can be used in Java development (see 5.1 and 5.2).
- **Native connector for C**: the native connector for C extends the Distributed Smart Space Orchestration System (DS2OS) beyond the boundaries of the Java Virtual Machine and can also serve as a blueprint for other native connectors (see 5.3).
- **Native Python connector** (SWIG wrapped): proved that the C connector can actually also be used in common scripting languages, to allow the development of services with several popular languages like for instance, Python. The approach is described in 5.4.
- **Recommendation guide on serialization formats**: backed by the analysis and evaluation from 2.3 and 6.1, recommendations on the usage of the four selected serialization formats are given in 6.1.4.

Regarding the common problems of smart spaces which are discussed in 1.1, here is how my work contributes to solving these issues as part of the Virtual State Layer middleware:

By making the VSL middleware more usable to developers which use different programming languages and platforms, the VSL's goal of **privacy preserving** smart spaces is faciliated by raising the motivation for developers to use the system, which would lead to a higher adoption rate and more control over data which leaves the local instance.

The integrated **high level of security** using Transport Layer Security (TLS) with client certificates for authentication as a mandatory part of the service interface design avoids accidential deployments with insufficient security settings like in the mentioned examples in the introduction. Additionally, the TLS protocol is extendable and frequently updated to the current state of the art in cryptography, so there is an easy upgrade path to newer algorithms and more secure authentication methods.

Regarding the **interconnectivity** of smart spaces, the usage of a standardized and interoperable service interface increases the ease to actually use the VSL middleware and with the native connectors, more devices and services can directly connect to the VSL, even if they use other programming languages than Java.

Now some ideas on future work are summarized.

7.2 Future work

Some further research can be conducted in this field and also more connectors and protocols could be implemented and evaluated. This list summarizes some ideas for future work:

- Creation of more native connectors in more languages, either using SWIG or native libraries of that language
- Implementation and evaluation of the RESTful service interface with other RESTful protocols, like especially CoAP
- Further evaluation of the different callback methods, showing when they are recommendable to use
- More testing and evaluation of the service interface in different networks like WLAN, mobile networks, Bluetooth, etc.

As the direct next steps, the connector implementations are further tested and polished, including more source code level documentation and examples.

Bibliography

[1] Pixabay, “Smart Home.” [Online]. Available: https://pixabay.com/de/smart-home-haus-technik-multimedia-2005993/

[2] Fielding, Roy and Reschke, Julian, “Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content,” RFC 7231 (Proposed Standard), Internet Engineering Task Force, 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7231.txt

[3] van der Meulen, Rob, “Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016,” Gartner, Inc., 2017. [Online]. Available: http://www.gartner.com/newsroom/id/3598917

[4] Lundgren, Lucas, “Light Weight Protocol! Serious Equipment! Critical Implications!” DEF CON 24, 2016. [Online]. Available: https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEFCON-24-Lucas-Lundgren-Light-Weight%20Protocol-Critical-Implications.pdf

[5] Toon, John, “Simulated Ransomware Attack Shows Vulnerability of Industrial Controls,” Georgia Institute of Technology, Atlanta, GA, 2017. [Online]. Available: http://www.rh.gatech.edu/news/587359/simulated-ransomware-attack-shows-vulnerability-industrial-controls

[6] W3C Working Group and others, “Web services architecture,” 2004. [Online]. Available: https://www.w3.org/TR/2004/NOTE-ws-arch-20040211/

[7] Bernstein, Philip A., “Middleware: A Model for Distributed System Services,” *Commun. ACM*, vol. 39, no. 2, pp. 86–98, 1996. [Online]. Available: http://doi.acm.org/10.1145/230798.230809

[8] Northrop, Linda and Feiler, Peter and Gabriel, Richard P and Goodenough, John and Linger, Rick and Longstaff, Tom and Kazman, Rick and Klein, Mark and Schmidt, Douglas and Sullivan, Kevin and others, “Ultra-large-scale systems: The software challenge of the future,” DTIC Document, Tech. Rep., 2006.

[9] Satyanarayanan, Mahadev, “Pervasive computing: Vision and challenges,” *IEEE Personal communications*, vol. 8, no. 4, pp. 10–17, 2001.

[10] Pahl, Marc-Oliver and Carle, Georg, "The missing layer—Virtualizing smart spaces," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2013 IEEE International Conference on.* IEEE, 2013, pp. 139–144.

[11] Pahl, Marc-Oliver and Carle, Georg and Klinker, Gudrun, "Distributed smart space orchestration," in *Network Operations and Management Symposium (NOMS), 2016 IEEE/IFIP.* IEEE, 2016, pp. 979–984.

[12] TIOBE software, "TIOBE Index for ranking the popularity of programming languages," 2016. [Online]. Available: http://www.tiobe.com/tiobe-index/

[13] Papazoglou, Mike P and van den Heuvel, Willem-Jan, "Service-Oriented Computing: State-of-the-Art and Open Research Issues," *IEEE Computer. v40 i11*, 2003.

[14] MacKenzie, C Matthew and Laskey, Ken and McCabe, Francis and Brown, Peter F and Metz, Rebekah and Hamilton, Booz Allen, "Reference model for service oriented architecture 1.0," *OASIS standard*, vol. 12, 2006.

[15] Pahl, Marc-Oliver, "Data-centric service-oriented management of things," in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM).* IEEE, 2015, pp. 484–490.

[16] Ahlgren, Bengt and Dannewitz, Christian and Imbrenda, Claudio and Kutscher, Dirk and Ohlman, Borje, "A survey of information-centric networking," *IEEE Communications Magazine*, vol. 50, no. 7, pp. 26–36, 2012.

[17] Cox, Brad J, *Object-oriented programming: an evolutionary approach*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[18] Bruns, Ralf and Dunkel, Jürgen, *Event-driven architecture: Softwarearchitektur für ereignisgesteuerte Geschäftsprozesse.* Heidelberg, Berlin, Germany: Springer-Verlag, 2010.

[19] Michelson, Brenda M, "Event-driven architecture overview," *Patricia Seybold Group*, vol. 2, 2006.

[20] Taylor, Hugh and Yochem, Angela and Phillips, Les and Martinez, Frank, *Event-driven architecture: how SOA enables the real-time enterprise.* Pearson Education, 2009.

[21] Birrell, Andrew D and Nelson, Bruce Jay, "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 1984.

[22] Goldsmith, Amy M and Goldsmith, David B and Pettus, Christopher E, "Object-oriented remote procedure call networking system," 1996, us patent 5,491,800.

[23] Bal, Henri E and Van Renesse, Robbert and Tanenbaum, Andrew S, "Implementing distributed algorithms using remote procedure calls," in *Proc. AFIPS Nat. Computer Conf*, vol. 56. Citeseer, 1987, pp. 499–506.

[24] Fielding, Roy T and Taylor, Richard N, "Principled design of the modern Web architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002.

[25] Fielding, Roy T, "REST APIs must be hypertext-driven," *Untangled musings of Roy T. Fielding*, 2008. [Online]. Available: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[26] Richardson, L. and Ruby, S., *RESTful Web Services.* O'Reilly Media, Inc., 2008.

[27] Fowler, Martin, "Richardson Maturity Model," 2010. [Online]. Available: http://martinfowler.com/articles/richardsonMaturityModel.html

[28] Algermissen, Jan, "Classification of HTTP-based APIs," 2010. [Online]. Available: http://nordsc.com/ext/classification_of_http_based_apis.html

[29] Richardson, L, "The Maturity Heuristic," 2008. [Online]. Available: http://www.crummy.com/writing/speaking/2008-QCon/act3.html

[30] Sumaray, Audie and Makki, S Kami, "A comparison of data serialization formats for optimal efficiency on a mobile platform," in *Proceedings of the 6th international conference on ubiquitous information management and communication.* ACM, 2012, p. 48.

[31] Aihkisalo, Tommi and Paaso, Tuomas, "A performance comparison of web service object marshalling and unmarshalling solutions," in *Services (SERVICES), 2011 IEEE World Congress on.* IEEE, 2011, pp. 122–129.

[32] Fowler, Martin, *Patterns of enterprise application architecture.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[33] Crawford, William and Kaplan, Jonathan, *J2EE design patterns.* O'Reilly, 2003.

[34] Bray, Tim and Paoli, Jean and Sperberg-McQueen, C Michael and Maler, Eve and Yergeau, François, "Extensible markup language (XML)," *World Wide Web Consortium Recommendation*, vol. 16, p. 16, 1998. [Online]. Available: http://www.w3.org/TR/1998/REC-xml-19980210

[35] Murata, Makoto and Lee, Dongwon and Mani, Murali and Kawaguchi, Kohsuke, "Taxonomy of XML schema languages using formal language theory," *ACM Transactions on Internet Technology (TOIT)*, vol. 5, no. 4, pp. 660–704, 2005.

[36] Crockford, Douglas, "JSON: The fat-free alternative to XML," in *Proc. of XML*, 2006. [Online]. Available: http://www.json.org/fatfree.html

[37] "262: ECMAScript language specification," *ECMA (European Association for Standardizing Information and Communication Systems)*, 1999.

[38] Bray, Tim, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 7159 (Proposed Standard), Internet Engineering Task Force, 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7159.txt

[39] FasterXML, "Jackson JSON Processor Wiki." [Online]. Available: http://wiki.fasterxml.com/JacksonHome

[40] Droettboom, Michael and others, "Understanding JSON Schema," 2015. [Online]. Available: http://spacetelescope.github.io/understanding-jsonschema/UnderstandingJSONSchema.pdf

[41] Bormann, Carsten and Hoffman, Paul, "Concise Binary Object Representation (CBOR)," RFC 7049 (Proposed Standard), Internet Engineering Task Force, 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7049.txt

[42] Varda, Kenton, "Protocol buffers: Google's data interchange format," *Google Open Source Blog*, 2008. [Online]. Available: https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html

[43] Seifert, Daniel, "Protocol Buffers - Google's data interchange format," 2011. [Online]. Available: https://spline.de/talks/2011/01/12/protocol-buffers/

[44] Google Developers, "Protocol Buffers." [Online]. Available: https://developers.google.com/protocol-buffers/

[45] Cardelli, Luca, "Type systems," *ACM Computing Surveys*, vol. 28, no. 1, pp. 263–264, 1996.

[46] w3schools, "XML Elements vs. Attributes." [Online]. Available: https://www.w3schools.com/XML/xml_dtd_el_vs_attr.asp

[47] Crockford, Douglas, "The application/json Media Type for JavaScript Object Notation (JSON)," RFC 4627 (Informational), Internet Engineering Task Force, 2006, Obsoleted by RFCs 7158, 7159. [Online]. Available: http://www.ietf.org/rfc/rfc4627.txt

[48] Thompson, Henry S. and Lilley, Chris, "XML Media Types," RFC 7303 (Proposed Standard), Internet Engineering Task Force, 2014. [Online]. Available: https://www.ietf.org/rfc/rfc7303.txt

[49] rafek, "Protobuffer uses octet-stream as its default content type," Git commit eb03145a6a7c72ae6cc43867d9635a5b8d8c4545 of protorpc project. [Online]. Available: https://github.com/google/protorpc/commit/eb03145a6a7c72ae6cc43867d9635a5b8d8c4545

[50] Elgamal, Taher and Hickman, Kipp EB, "Secure socket layer application program apparatus and method," 1997, US Patent 5,657,390.

[51] Dierks, Tim and Rescorla, Eric, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[52] Turner, Sean, "Transport layer security," *IEEE Internet Computing*, vol. 18, no. 6, pp. 60–63, 2014.

[53] Rescorla, Eric and Modadugu, Nagendra, "Datagram Transport Layer Security Version 1.2," RFC 6347 (Proposed Standard), Internet Engineering Task Force, 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6347.txt

[54] Salowey, Joseph and Zhou, Hao and Eronen, Pasi and Tschofenig, Hannes, "Transport Layer Security (TLS) Session Resumption without Server-Side State," RFC 5077 (Proposed Standard), Internet Engineering Task Force, 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5077.txt

[55] Hollenbeck, Scott, "Transport Layer Security Protocol Compression Methods," RFC 3749 (Proposed Standard), Internet Engineering Task Force, 2004. [Online]. Available: http://www.ietf.org/rfc/rfc3749.txt

[56] Seggelmann, Robin and Tuexen, Michael and Williams, Michael Glenn, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension," RFC 6520 (Proposed Standard), Internet Engineering Task Force, 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6520.txt

[57] Kelsey, John, "Compression and information leakage of plaintext," in *International Workshop on Fast Software Encryption.* Springer, 2002, pp. 263–276.

[58] C. Meyer and J. Schwenk, "Lessons Learned From Previous SSL/TLS Attacks-A Brief Chronology Of Attacks And Weaknesses." *IACR Cryptology EPrint Archive*, vol. 2013, p. 49, 2013.

[59] Durumeric, Zakir and Kasten, James and Adrian, David and Halderman, J Alex and Bailey, Michael and Li, Frank and Weaver, Nicolas and Amann, Johanna and Beekman, Jethro and Payer, Mathias and others, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference.* ACM, 2014, pp. 475–488.

[60] Friedl, Stephan and Popov, Andrei and Langley, Adam and Stephan, Emile, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension," RFC 7301 (Proposed Standard), Internet Engineering Task Force, 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7301.txt

[61] Eronen, Pasi and Tschofenig, Hannes, "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)," RFC 4279 (Proposed Standard), Internet Engineering Task Force, 2005. [Online]. Available: http://www.ietf.org/rfc/rfc4279.txt

[62] Medvinsky, Ari and Hur, Matthew, "Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)," RFC 2712 (Proposed Standard), Internet Engineering Task Force, 1999. [Online]. Available: http://www.ietf.org/rfc/rfc2712.txt

[63] Cooper, David and Santesson, Stefan and Farrell, Stephen and Boeyen, Sharon and Housley, Russell and Polk, Tim, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280 (Proposed Standard), Internet Engineering Task Force, 2008. [Online]. Available: http://www.ietf.org/rfc/rfc5280.txt

[64] Fielding, Roy and Reschke, Julian, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing," RFC 7230 (Proposed Standard), Internet Engineering Task Force, 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7230.txt

[65] Belshe, Mike and Peon, Roberto and Thomson, Martin, "Hypertext Transfer Protocol Version 2 (HTTP/2)," RFC 7540 (Proposed Standard), Internet Engineering Task Force, 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7540.txt

[66] Fielding, Roy and Nottingham, Mark and Reschke, Julian, "Hypertext Transfer Protocol (HTTP/1.1): Caching," RFC 7234 (Proposed Standard), Internet Engineering Task Force, 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7234.txt

[67] Fielding, Roy and Reschke, Julian, "Hypertext Transfer Protocol (HTTP/1.1): Authentication," RFC 7235 (Proposed Standard), Internet Engineering Task Force, 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7235.txt

[68] Van Kesteren, Anne, "Cross-Origin Resource Sharing," *World Wide Web Consortium (W3C)*, 2014. [Online]. Available: https://www.w3.org/TR/2014/REC-cors-20140116/

[69] Devresse, Adrien and Furano, Fabrizio, "Efficient HTTP based I/O on very large datasets for high performance computing with the libdavix library," in *Workshop on Big Data Benchmarks, Performance Optimization, and Emerging Hardware*. Springer, 2014, pp. 194–205.

[70] C. Bormann, A. P. Castellani, and Z. Shelby, "Coap: An application protocol for billions of tiny internet nodes," *IEEE Internet Computing*, vol. 16, no. 2, pp. 62–67, 2012.

[71] Shelby, Zach and Hartke, Klaus and Bormann, Carsten, "The Constrained Application Protocol (CoAP)," RFC 7252 (Proposed Standard), Internet Engineering Task Force, 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7252.txt

[72] Fette, Ian and Melnikov, Alexey, "The WebSocket Protocol," RFC 6455 (Proposed Standard), Internet Engineering Task Force, 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6455.txt

[73] Hardie, Ted, "Clarifying Registry Procedures for the WebSocket Subprotocol Name Registry," RFC 7936 (Proposed Standard), Internet Engineering Task Force, 2016. [Online]. Available: http://www.ietf.org/rfc/rfc7936.txt

[74] Cerami, Ethan, *Web services essentials: distributed applications with XML-RPC, SOAP, UDDI & WSDL.* O'Reilly Media, Inc., 2002.

[75] Winer, Dave, "Xml-rpc specification," http://xmlrpc.scripting.com/spec, 1999.

[76] Yankowski, Fred, "XML-RPC Extensions," http://ontosys.com/xml-rpc/extensions.php, 2001.

[77] Graham, Steve and Simeonov, Simeon and Boubez, Toufic and Davis, Doug and Daniels, Glen and Nakamura, Yuichi and Neyama, Ryo, *Building Web services with Java: making sense of XML, SOAP, WSDL, and UDDI.* SAMS publishing, 2001.

[78] Tilkov, Stefan, "UDDI R.I.P." http://www.innoq.com/blog/st/2010/03/uddi_rip.html, 2010.

[79] Loreto, Salvatore and Saint-Andre, Peter and Salsano, Stefano and Wilkins, Greg, "Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP," RFC 6202 (Informational), Internet Engineering Task Force, 2011. [Online]. Available: http://www.ietf.org/rfc/rfc6202.txt

[80] Vinoski, Steve, "CORBA: integrating diverse applications within distributed heterogeneous environments," *IEEE Communications magazine*, vol. 35, no. 2, pp. 46–55, 1997.

[81] Christensen, Erik and Curbera, Francisco and Meredith, Greg and Weerawarana, Sanjiva and others, "Web services description language (WSDL) 1.1," 2001.

[82] "SWIG-3.0 Documentation," SWIG developers, 2017. [Online]. Available: http://swig.org/Doc3.0/SWIGDocumentation.pdf

[83] Cholia, Shreyas and Skinner, David and Boverhof, Joshua, "NEWT: A RESTful service for building High Performance Computing web applications," in *Gateway Computing Environments Workshop (GCE), 2010.* IEEE, 2010, pp. 1–11.

[84] Maeda, Kazuaki, "Performance evaluation of object serialization libraries in XML, JSON and binary formats," in *Digital Information and Communication Technology*

and it's Applications (DICTAP), 2012 Second International Conference on. IEEE, 2012, pp. 177–182.

[85] M. Kovatsch, M. Lanter, and Z. Shelby, "Californium: Scalable cloud services for the internet of things with coap," in *Internet of Things (IOT), 2014 International Conference on the.* IEEE, 2014, pp. 1–6.

[86] T. Aihkisalo and T. Paaso, "Latencies of service invocation and processing of the rest and soap web service interfaces," in *Services (SERVICES), 2012 IEEE Eighth World Congress on.* IEEE, 2012, pp. 100–107.

[87] K. Ericson and S. Pallickara, "Adaptive heterogeneous language support within a cloud runtime," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 128–135, 2012.

[88] N. M. O'Boyle, C. Morley, and G. R. Hutchison, "Pybel: a python wrapper for the openbabel cheminformatics toolkit," *Chemistry Central Journal*, vol. 2, no. 1, p. 5, 2008.

[89] Liebald, Stefan and Hubel, Andreas, "Parametrized requests," 2016. [Online]. Available: https://redmine.dev.ds2os.org/projects/ds2os/wiki/Parametrized_requests

[90] Nazgob and others, "Google Protocol Buffers and HTTP." [Online]. Available: https://stackoverflow.com/questions/1425912/google-protocol-buffers-and-http

[91] Dias, Jader and others, "What is the correct Protobuf content type?" [Online]. Available: https://stackoverflow.com/questions/30505408/what-is-the-correct-protobuf-content-type

[92] Jovanovic, Nenad and Kirda, Engin and Kruegel, Christopher, "Preventing Cross Site Request Forgery Attacks," in *Securecomm and Workshops, 2006.* IEEE, 2006, pp. 1–10.

[93] The Apache Software Foundation, "Welcome to Apache Maven." [Online]. Available: https://maven.apache.org/

[94] The Eclipse Foundation, "Jetty - Servlet Engine and Http Server." [Online]. Available: https://www.eclipse.org/jetty/

[95] Kitware, "CMake." [Online]. Available: https://cmake.org/

[96] Haszlakiewicz, Eric, "JSON-C - A JSON implementation in C." [Online]. Available: https://github.com/json-c/json-c/wiki

[97] Stenberg, Daniel and others, "curl." [Online]. Available: https://curl.haxx.se/

[98] Oracle Corporation, "OpenJDK: jmh," 2017. [Online]. Available: http://openjdk.java.net/projects/code-tools/jmh/

[99] Saloranta, Tatu, “jackson-dataformats-binary/protobuf at master · FasterXML/jackson-dataformats-binary · GitHub.” [Online]. Available: https://github.com/FasterXML/jackson-dataformats-binary/tree/master/protobuf