FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Remote Function Calls for Haskell Applications
Student:	Martin Bednář
Supervisor:	Ing. Marek Suchánek
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Haskell is a functional programming language that today allows the creation of robust backend applications using various frameworks. For management and testing purposes, it is necessary to run service functions remotely (e.g. clear cache, display diagnostics, change logging, etc.). This work aims to facilitate these operations similarly to Java JMX and MBeans.

- Analyze remote function calls and existing solutions, focus on JMX and MBeans.
- Describe the possibilities in Haskell for exposure and remote calls of functions with the least possible burden for programmers.
- Design a solution for Haskell web applications that will expose tagged or otherwise selected functions through the API. Also, consider the need to secure remote calls.
- Implement and demonstrate the solution. Implement client applications in the form of a simple web UI or use an existing one compatible with your API.
- Test the resulting solution, evaluate it and prepare it for further development as open-source.

Electronically approved by Ing. Michal Valenta, Ph.D. on 14 October 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

Remote Function Calls for Haskell Applications

Martin Bednář

Department of Software Engineering
Supervisor: Ing. Marek Suchánek

May 11, 2022

Acknowledgements

Firstly, I would like to thank my supervisor, Ing. Marek Suchánek, for supervising this work, and for his advice.

Secondly, I would like to thank my partner, my family, and my dog, for providing the support I needed.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on May 11, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Bednář, Martin. *Remote Function Calls for Haskell Applications.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstract

The ability to reliably monitor and manage running applications is crucial in modern software development. This bachelor's thesis is dedicated to remote application management in the Haskell programming language. The current state was examined, and two software projects were created, which extended the possibilities of remote management of Haskell applications. The resulting software consists of a user-facing application management system, and a remote application management framework, that can also be used for remote function calls.

Keywords software library, remote function call, application management, application monitoring, Haskell, Java Management Extensions, MBeans

Abstrakt

Schopnost spolehlivě monitorovat a řídit běžící aplikace je v moderním softwarovém vývoji zásadní. Tato bakalářská práce se věnuje vzdálenému řízení aplikací v programovacím jazyku Haskell. Současný stav byl prozkoumán a byly vytvořeny dva softwarové projekty, které rozšiřují možnosti vzdáleného řízení Haskell aplikací. Výsledný software se skládá ze systému pro řízení aplikací s uživatelským rozhraním a z frameworku pro vzdálené řízení aplikací, který lze také použít pro vzdálená volání funkcí.

Klíčová slova softwarová knihovna, vzdálené volání funkce, vzdálená správa aplikací, monitoring aplikací, Haskell, Java Management Extensions, MBeans

Contents

List of Figures

List of Tables

List of code snippets

Introduction

In recent years, the Haskell programming language has been getting more convenient for production use, as indicated by the number of published libraries as well as reports of its usage in the industry.

However, some important capabilities of remote administration (a crucial aspect of the modern software development lifecycle) are still missing, as opposed to other programming languages. Specifically, Haskell lacks an admin tool that would provide complex monitoring, management, and administration of running applications. Examples of such tools for other programming languages include Spring Boot Admin (Java language, Spring framework) or Django Admin (Python language, Django framework).

This thesis proposes combining existing management tools into a unified platform. That way, administrators can perform various application management tasks from a single user interface. It also minimizes the development overhead needed to combine different application management tools and simplifies the development of new ones by addressing common concerns (user interface, client-server communication, security, API design etc.).

Moreover, this thesis proposes an extensible Remote Function Call (RFC) library compatible with multiple RFC specifications and protocols, following the pattern of Java Management Extensions (JMX). While some RFC protocols and specifications are covered by existing Haskell libraries, none of these libraries are designed to be extended to other protocols. Considering that the core functionality of a remote function call is very similar across all the different solutions, creating a common implementation base is desired.

Java Management Extensions achieve that in terms of Java – they provide an implementation of the core functionality and enable developers to easily add support to new protocols.

Porting the successful design of Java Management Extensions to Haskell simplifies the development of RFC libraries by providing a similar common base for all protocols and specifications.

The proposed solutions (management platform and RFC library) can also

be combined, providing a simple and straightforward way for programmers to expose some functions for remote administration, which is more complicated using previously available solutions.

The thesis is divided into four chapters. Chapter 1 discusses remote function calls, the abilities of Haskell, and previous solutions written in both Haskell and other programming languages. Next, Chapter 2 and Chapter 3 discuss possible solutions to the problems determined above. Finally, Chapter 4 describes the development and testing of both projects (the remote function call library and the unified management platform), their evaluation, and possibilities for further development.

Goals

The ultimate goal of this thesis is the creation of a software solution allowing remote management and remote function calls with the least possible burden for programmers. In accordance with the thesis assignment, the solution will be implemented in the Haskell programming language.

Other goals and subgoals derived from the ultimate goal are divided into analysis, design, and realization, according to the traditional process of software development.

In terms of analysis, there are two goals: First, an analysis of remote function calls and existing solutions in Haskell and other programming languages, focusing on JMX and MBeans. Second, a description of possibilities in Haskell and its type system for exposure and remote calls of functions.

In terms of design, the main goal is to design a solution for Haskell web applications that will expose tagged or otherwise selected functions through the API. Also, the need to secure remote calls will be considered.

In terms of realization, the main goal is to implement and demonstrate the designed solution. Additionally, the solution will be tested, evaluated, and prepared for further development as open-source.

The implemented system will be demonstrated using a dedicated client application in the form of a simple web UI.

Chapter 1

State-of-the-Art

This chapter establishes the theoretical basis of this thesis. It covers the topics of remote application management, remote function calls, the Haskell programming language and related existing software projects. The knowledge contained in this chapter is then used in the following chapters (Chapter 2, Chapter 3, and Chapter 4) to design and realize a software project, that implements remote management in Haskell.

1.1 Remote Application Management

This section discusses the topic of remote application management, emphasizing its relation to functional programming and Haskell.

Remote application management, as referred to in this thesis, stands for any interaction with a running application that is carried out remotely with the purpose of observing or modifying the configuration or behavior of the application. The interaction being carried out remotely means that it doesn't rely on physical access to the device on which the application is running. Instead of physically accessing the device to perform management tasks, a software system is used to perform these tasks from a different device. Examples of management tasks include modification of environment variables or examination of errors logged by the application.

Similarly, **remote application monitoring** also interacts with a running application, but it's limited to passively observing and reporting metrics related to the application's state. Compared to application management, it doesn't actively influence the application's behavior, so only some management tasks are enabled by remote monitoring. For example, remote monitoring enables examination of logged errors, because it's a task that doesn't modify the application's behavior in any way.

1.1.1 Remote Management Architecture

Kreger (2001) describes a typical software architecture that is often used to implement remote application management. [2]

The architecture consists of managed **applications**, their **agents** and a **management system**. [2]

The **application** is simply an arbitrary computer program, running on a server or any other device. Its main responsibilities are "exposing appropriate data for use by a management system" [2] and "responding to requests from the management system" [2]. The application doesn't communicate with a management system directly. Instead, a specialized software component called the *agent* is used to connect the application to the management system [2].

The **agent** mediates the communication between an application and a management system. It provides an interface for the management system to access the application by handling data transfer and other communication. It's "responsible for sending events to the management system, relaying data and command requests from the management system and the application, gathering responses, and returning them to the requester." [2]

The agent can exist as a separate computer program, or as a part of the application. In both cases, it's directly connected to the application and they usually run on "the same host or process". [2]

The **management system** is a user-facing software system responsible for "providing the infrastructure and user interfaces to manage applications." [2] It manages one or more applications through their agents. Apart from taking direct user input, it may execute automated actions or react to events received from the agents. "The management system can initiate a command or data request, send commands to agents to control the application, and request data from the agent (which requests it from the application) for polling, monitoring, trending, or problem determination." [2] It may have the form of an administration dashboard, or something more sophisticated.

The three-level architecture consisting of applications, agents and a management system separates the software solution into independent components. Each component handles a different aspect of remote management, so they can be developed independently, as well as reused in other projects.

A particularly interesting use case of this architecture is the *Java Management Extensions (JMX)* technology [2], which will be further discussed in Section 1.5.1.

1.1.2 Relation to Cloud-Based Applications

In today's world of software engineering, it's a standard and often desirable practice to deploy a backend application in an isolated environment "in the cloud". Moreover, larger systems are typically deployed in a distributed way,

with several different applications or application instances running on different machines. [3]

Compared to an older model, where the application would run on a single server with direct access, the cloud-based approach brings specific challenges, requiring dedicated software tools to be used for remote management of applications. In the older model, the application could be monitored and managed using log files and tools provided by the operating system. Also, the server running the application could be accessed easily. [3]

However, in cloud-based and distributed applications, servers are typically not physically accessible and their locations on the network often change. Also, the large number of application instances that may be running simultaneously makes manual intervention on individual servers impractical.

To conclude, most modern software solutions running in the cloud require a remote management system capable of fast, safe and efficient control over individual application instances.

1.2 Remote Function Calls

This section covers the topic of remote function calls (otherwise known as remote *procedure* calls) and their relation to remote application management.

A **remote function call (RFC)** is "a distributed computing technique in which a computer program calls a procedure (subroutine or service) to execute in a different address space than its own. The procedure may be on the same system or a different system connected on a network." [4] There are two actors working together to perform a remote function call – the *caller (client)*, which initiates the call and provides input data, and the *callee (server)*, which executes the call locally and provides the result. Each of them has its own address space.

As the definition suggests, there is a fundamental difference between a *remote* call and a *local* call (i. e. the native function call mechanism used in a particular language). Local calls are based on a simple mechanism of transfer of control between procedures sharing a memory space. Contrary to that, a remote call involves two different memory spaces (possibly on different devices) – and that comes with the need for data serialization and transfer between the memory spaces [5]. One of the memory spaces belongs to the caller, and the other one to the callee.

Depending on the particular use case, remote calls may be implemented so that, on the caller side, the usage is very similar to local calls. In that case, the particular remote call library used is fully responsible for data serialization and communication with the callee, providing an abstraction for the programmer on the caller side. The programmer then uses remote calls as if they were local calls.

In contrast, remote calls may also be implemented so that the programmer on the caller side is responsible for managing the specific difficulties of RFCs (e. g. handling network errors). The difference between local and remote calls is then more significant.

In the context of functional programming, remote function calls can also be thought of as a remote management technique that enables the caller to cause *side effects*[i] on the callee side.

Considering a functional language like Haskell, remotely calling a *pure function*[ii] without any side effects is not particularly useful, because the call result is guaranteed to be the same as if the function was called locally, and the lack of side effects prohibits the callee to interact with its environment. Additionally, a remote call by itself necessarily causes a side effect, because it's transferred over a network – the data transfer between the caller and the callee causes the state of the network to change. Also, a remote call can fail randomly in case of a network error, so a computation involving remote calls can't be truly pure.

It may, however, be desirable to execute a computation (that would otherwise be pure) on a different machine to share processing power and improve execution time. Further developing this idea, remote calls can be used as the basis of a distributed computing system.

Considering *effectful* computations, RFCs provide a way for the caller to execute side effects on the callee side. That allows the caller to, among other things, observe and manipulate the callee's system environment (system health, system load, available resources, other running processes etc.), making RFCs useful for remote application management.

1.2.1 RFC Protocols

Many distinct implementations of remote calls exist today. They differ in terms of supported programming languages – some are specific to one language and some support multiple languages. They also differ in complexity, communication protocols, data representation and other properties.

Some of the implementations follow dedicated RFC specifications, such as *gRPC* or *JSON-RPC*, that dictate the communication protocols, data representation, error management techniques, etc. Thanks to these specifications, consistent and reliable remote calls are possible even between different programming languages.

[i] A side effect is any operation that uses or modifies the global state of the application or system resources. Modifying a global variable, communicating over a network, or reading from a file are examples of a side effect.

[ii] A function is pure if it doesn't perform any side effects, and always returns the same output for a given input (its output only depends on the input and not on the "outside world".)

1.3 Haskell

Haskell is a statically-typed, purely functional programming language. Being a purely functional language, it puts special focus on handling side effects. Functions aren't allowed to execute arbitrary side effects, which are instead handled by a designated monadic I/O system instead. [6]

Haskell's type system consists of a set of predefined primitive and composite types. It can be extended easily with user-defined algebraic types. [6]

Other relevant features of Haskell include higher-order functions, parametric polymorphism and type classes. [6]

1.4 Previous Solutions in Haskell

This section examines current Haskell-specific tools suitable for remote application management and remote function calls.

All of the further mentioned tools were published to Hackage – the de facto standard archive of open-source packages used by Haskell developers.

1.4.1 EKG

EKG[iii] is a Haskell library focused on application monitoring, enabling both local and remote monitoring of running applications. It uses HTTP for communication and provides an HTML client, as well as a JSON API. [7]

Primarily, EKG displays basic statistics about resource usage and system health. However, various packages published on Hackage extend its functionality so that it forms a complete solution for application monitoring.

Application monitoring, the focus area of EKG, is closely related to application management. While application management should enable complete *active control* over the running application, application monitoring focuses on the specific area of *passively observing* metrics related to the running application, without interfering with the application state or with the computer system.

EKG isn't suitable as a remote application management tool by itself, because it focuses solely on application monitoring. However, application monitoring is a significant part of application management, so it provides a good starting point for the creation of application management software.

1.4.2 RFC Implementations

Remote calls are implemented by several existing Haskell libraries.

[iii] https://hackage.haskell.org/package/ekg

Some of them implement a remote call protocol according to a published specification, such as **grpc-haskell**[iv] and **json-rpc**[v], implementing the gRPC and JSON-RPC specifications, respectively.

Other packages focus on implementing Haskell-specific remote calls, implementing a custom protocol. An example of that is **Curryer**[vi]. Instead of implementing a well-defined specification, it utilizes existing Haskell libraries for serialization and streaming to implement easy and fast RFC. [8]

Several other RFC implementations exist, providing a wide range of different protocols to choose from. However, nearly all of them reimplement the core functionality of remote calls on their own. It would be more desirable to have a solid common library providing the core functionality of remote calls, that could be adapted for specific use cases. This idea is partly realized by the **mu-rpc**[vii] package, that implements remote calls in a protocol-independent way. However, this package is a part of **Mu-Haskell**[viii] (a microservices framework), and isn't intended for general usage outside of the framework.

In conclusion, there is a large number of existing implementations of remote function calls in Haskell. However, they lack unity and most of them don't make use of a common core. They are also arguably unnecessarily difficult to use – all of the examined implementations require quite a lot of setup before remote calls can be used.

1.5 Previous Solutions in Other Programming Languages

This section describes **Java Management Extensions (JMX)** and **Spring Boot Admin** – two software projects written in the Java language. Both of them focus on remote management and both are widely used by Java programmers.

This thesis takes inspiration from them, and in later chapters proposes a software solution written in Haskell, that is aimed to become the Haskell equivalent of JMX and Spring Boot Admin.

1.5.1 Java Management Extensions

Java Management Extensions is a remote management technology targeting the Java programming language. It defines "an architecture, the design patterns, the APIs, and the services" [9] for remote management and monitoring of Java applications and other resources. [9, 3]

[iv] https://hackage.haskell.org/package/grpc-haskell
[v] https://hackage.haskell.org/package/json-rpc
[vi] https://hackage.haskell.org/package/curryer-rpc
[vii] https://hackage.haskell.org/package/mu-rpc
[viii] https://higherkindness.io/mu-haskell/

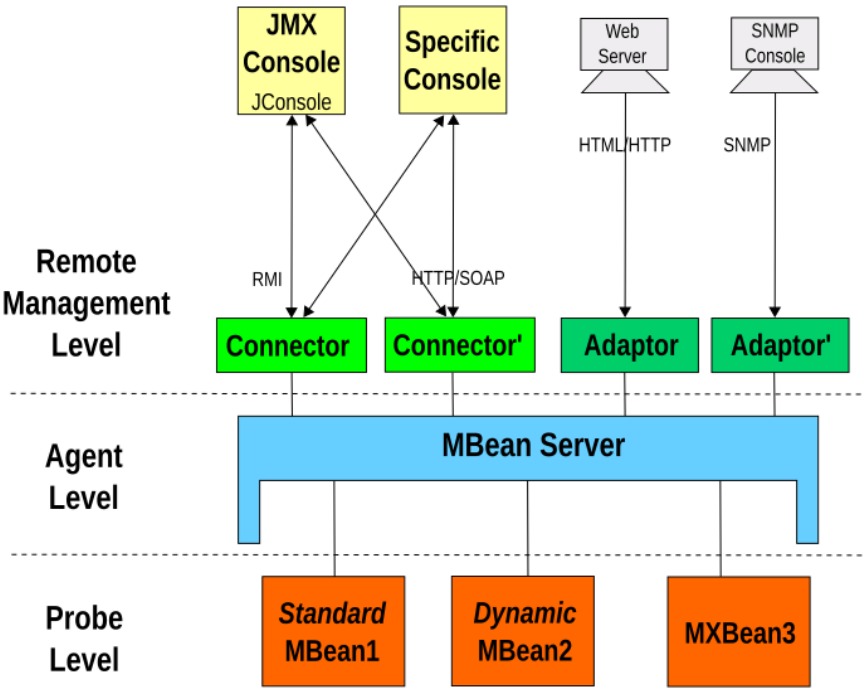

Figure 1.1: JMX Architecture, cited from [1]

JMX as a whole follows the JMX specification, that exists in several consecutive versions. This text considers *Java Management Extensions (JMX) Specification, version 1.4*, which was released in 2006 by Sun Microsystems. [9]

The architecture of JMX is particularly interesting for this thesis, as it is mostly independent of the specifics of Java and can be largely ported to Haskell. It fits into the broadly defined remote management architecture described by Kreger (2001) [2], which has been discussed in Section 1.1.1.

The architecture of the JMX technology consists of three levels:

1. instrumentation level,
2. agent level,
3. distributed services level. [3]

The levels roughly correspond to the architectural components described by Kreger (2001) – the *application*, the *agent* and the *management system*, respectively.

Fig. 1.1 shows an overview of the architectural components and their relations.

1.5.1.1 Instrumentation Level

A *resource* is an entity in the system that needs to be remotely monitored and controlled [3]. Relevant examples include "an application, an implementation of a service, a device, a user" etc. [9]

The first level of JMX, called the **instrumentation level**, is responsible for making individual resources manageable through specialized objects called *MBeans*. [3]

An MBean consists of **attributes**, which completely describe the state of the resource, **operations**, which may be invoked remotely from a management system and cause the resource to perform an action, and **notifications**, that are created by the resource and can be passed to the agent level. [3]

In JMX, MBeans are the elemental units of remote management. Each of them is tied to a specific system resource that it manages. Different types of MBeans exist, which differ in the level of control given to the programmer. [3] The types of MBeans won't be discussed in this thesis, because they are rather irrelevant to the high-level architecture of JMX.

1.5.1.2 Agent Level

The instrumentation level is responsible for creating separate MBeans, as discussed earlier. In contrast, the **agent level** is responsible for maintaining a registry of all the MBeans in an application and making them available to management systems. On top of that, the agent level contains several specialized services, as mandated by the JMX specification. [3, 9]

The agent level defines a standardized agent used to manage JMX resources. The agent consists of two parts – the **MBean server** and the **JMX agent services**. [3, 9]

The **MBean server** contains a registry of available MBeans, associated with their so-called *object names*, that uniquely identify each MBean. The MBean server mediates the interactions between management systems and manageable resources by invoking methods on the registered MBeans. [3]

The **JMX agent services** provide specific functionalities, that are related to the managed application as a whole, rather than to individual MBeans. For example, agent services manage cooperation between several MBeans (the *relation service*), provide a way to dynamically load MBeans from the network (the *M-Let service*), or collect statistics by observing the attribute value of an MBeans at specific intervals (the *monitoring services*). [3]

The JMX agent typically runs on the same machine as the managed application and connects the application and system resources with the outside world. As it may have access to sensitive information and resources, the standard Java security model is used to restrict access to the agent. Different permissions are defined to control access to the individual services and the MBean server. [9]

1.5.1.3 Distributed Services Level

The **distributed services level** (also called the remote management level) is responsible for connecting JMX agents to management systems or any other applications that need to communicate with an agent. It consists of *protocol adaptors* and *connectors*. [3]

A **Protocol adaptor** connects JMX agents to a specific application – for example, a protocol adaptor can connect a JMX agent with a browser-based admin board, to enable an administrator to interact with MBeans. The protocol adaptor is then responsible for handling communication (perhaps over HTTP) and data transfer between the Java-based JMX agent and the remote management application, which may be written in any programming language. [3]

A **connector**, on the other hand, connects a pair of JMX agents in a *client-server* relationship. It enables the client to access MBeans on the server, hiding the communication details, such as the server's network location. [3]

1.5.1.4 JMX – Conclusion

In conclusion, the JMX technology provides a complete bundle for remote management of Java applications, that can be adapted to any management system. The layered architecture enables the different challenges of remote management to be targeted individually, which results in loosely-coupled components. That makes it easy to add new remotely managed resources or to switch to a different management system. The individual levels of the architecture are independent of each other, because they only communicate through well-defined interfaces, in accordance with the JMX specification.

1.5.2 Spring Boot Admin

Spring Boot Admin is an admin interface used to monitor applications built with the Spring Boot framework. It has the role of a *management system*[ix], as described by Kreger (2001) [2].

Spring Boot supports a wide range of features focused on remote management and monitoring, such as:

- showing application health status,
- managing environment variables and JVM system properties,
- interacting with MBeans using JMX,
- downloading heapdump,
- sending notifications on status change (e. g. via e-mail). [10]

[ix] See Section 1.1.1

The list shows the diversity of remote management tasks and responsibilities. Apart from resource usage statistics, notifications, and the ability to configure the running application, it supports interaction with MBeans using JMX. Here, the JMX technology provides a way to extend the abilities of the remote management system with custom management tasks defined as MBeans.

In terms of remote management, every application is different and needs a slightly different remote management solution. It's convenient to be able to easily define additional management tasks focused on the specific needs of the application [2]. The combination of Spring Boot Admin and JMX provides a usable solution to this problem by providing a flexible, safe, and easy-to-use way of remote management.

1.6 Conclusion

This chapter has laid out the theoretical basis needed to create a remote management solution for Haskell applications. It covered the topics of remote application management and remote function calls, introduced a software architecture suitable for remote management solutions, and described previous projects in the field of remote management.

It has been found that a complex remote management software dedicated to Haskell applications currently doesn't exist. The current solutions partially provide support for most of the desired management tasks, but they are not united in a single suite. Also, there isn't any remote management framework, so it's currently difficult to incorporate custom remote management tasks needed to manage a specific application.

Based on this knowledge, a new remote management solution will be created, that will fill in the gaps and provide a unified platform for remote management. The following chapters will document the creation of this solution.

Chapter 2

Analysis

The ultimate goal of this thesis is the creation of a software system allowing remote management and remote function calls for Haskell applications.

This chapter focuses on describing the system from a high-level perspective (Section 2.1), analyzing its use cases (Section 2.2), and identifying specific requirements that the system should meet (Section 2.3).

2.1 Overall Description

This section provides a high-level overview of the software system to be created.

The software system will consist of two independent parts (that can be, however, used together) – *Managed Functions* and *Haskell Admin*.

The first part, called **Managed Functions**, will be a software framework inspired by the JMX technology. It will adopt the concept of the *agent*, a software component that is attached to an application and manages its resources. Similarly to JMX, resources will be encapsulated in a structure with a common API (an equivalent of MBeans). However, contrary to object-oriented Java, Haskell is a purely functional language that lacks the concept of *objects*. Instead of objects, *functions* are the basic building blocks of a Haskell program. So, instead of representing the application resources as objects with certain properties, the Haskell solution will represent them using functions and I/O actions. Following the example of JMX, the Haskell solution's architecture will consist of three levels (instrumentation, agent, and distributed services). The interactions between these levels will be mostly similar to those found in JMX.

The second part of the software system, called **Haskell Admin**, will be a user-facing management system, inspired by Spring Boot Admin and similar projects. A modular architecture will be used to combine existing Haskell solutions into a complete package.

The two parts of the solution will possibly be used together, similarly to how JMX integrates with Spring Boot Admin.

2.2 Use Case Analysis

This section briefly describes the expected use cases of both software projects. UML Use Case diagrams are used to illustrate the findings.

2.2.1 Use Cases – Haskell Admin

Considering **Haskell Admin**, there are two actors occurring in the use cases:

- the **Administrator**, a human user interacting with the graphical user interface (GUI) to perform management tasks,
- the **Connected software**, which doesn't interact with the GUI of Haskell Admin and instead queries the Haskell Admin API directly.

The method of access is different for both actors. However, the nature of their interaction with the system is the same – they both perform *management tasks*.

In this context, management tasks refer to a wide variety of interactions that can be performed with the managed application, such as:

- check application health,
- stop or restart the application,
- examine resource usage (memory, CPU, network, etc.),
- invoke a Managed Function,
- manage environment variables.

An overview of actors and use cases concerning Haskell Admin is shown in Fig. 2.1 in the form of a UML Use Case Diagram.

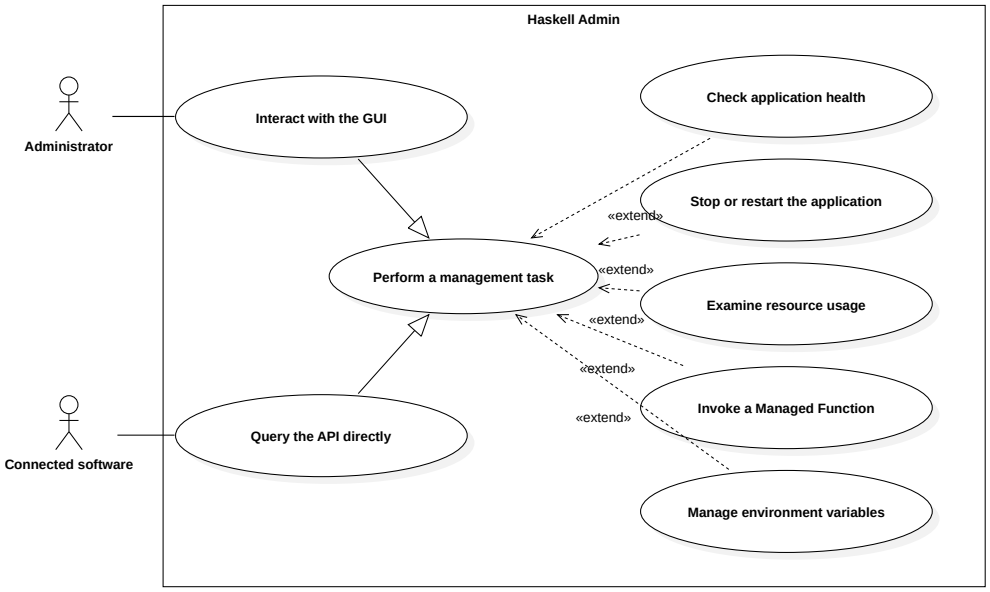

Figure 2.1: Use Case Diagram for Haskell Admin

2.2.2 Use Cases – Managed Functions

Considering **Managed Functions**, there are two actors:

- The **Programmer**, who creates a specific application of the framework, as described below. The Programmer sets up a Connector, that can then be used to interact with the exposed functions.
- The **User**, who interacts with the framework through the interface of the specific Connector. The User can invoke functions with certain arguments and fetch additional information about the functions. The user can either be a human or a computer program.

Managed Functions is a software framework (as opposed to a stand-alone computer program), which provides the basis for a specific remote management solution, but needs to be integrated into an application by the Programmer. To use the framework, the Programmer is expected to:

1. enumerate the functions that should be exposed,
2. wrap the functions into an Agent,
3. choose a suitable Connector or write a new one,
4. and finally, run the Connector, giving it the previously created Agent as an argument.

After these steps, the User can call the selected functions remotely using the interface of the specific Connector. Apart from calling functions, the Connector can also be used to list available functions, and provide additional information, such as the type signature of the function and its parameters.

The use cases described above are summarized in Fig. 2.2 in the form of a UML Use Case Diagram.

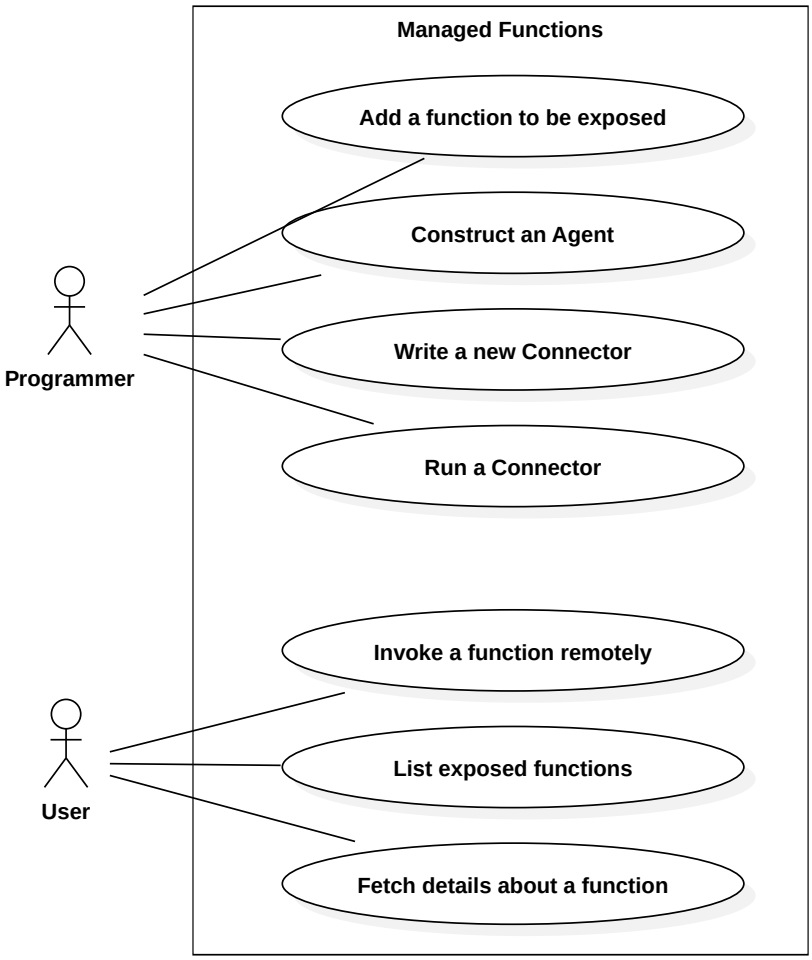

Figure 2.2: Use Case Diagram for Managed Functions

2.3 Requirements Analysis

This section specifies the software requirements for both Haskell Admin and Managed Functions, using the so-called **MoSCoW method**. This method categorizes individual requirements into four categories, based on their priority:

1. *must* have (mandatory),
2. *should* have (important),
3. *could* have (nice to have),
4. *won't* have or *would* have (not a priority). [11]

Based on the assignment and goals of this thesis, the following requirements have been selected for the software system being created. Some of the requirements only apply to one of the components (Haskell Admin or Managed Functions, respectively) – these are annotated with the name of the component. Also, every requirement is annotated with either *F*, or *NF*, based on whether it's a functional, or non-functional requirement.

Must have

- Users can manage applications remotely from a different machine (*F*).
- Custom functions can be used for remote management (Managed Functions, *F*).
- The system is usable for remote function calls (Managed Functions, *F*).
- A graphical user interface is available (Haskell Admin, *F*).

Should have

- The system is tested well (*NF*).
- The system is prepared for further development as open-source (*NF*).
- The system supports basic security by providing an authorization mechanism (*F*).
- The system can be combined with existing solutions easily (Haskell Admin, *NF*).
- Functions can be exposed for remote calls easily (Managed Functions, *NF*).
- The user can choose between various data representations (Managed Functions, *F*).

Could have

- The system supports sending e-mail notifications (Haskell Admin, *F*).
- Streaming (continuous data flow) is supported (Managed Functions, *F*).
- The list of exposed functions can be adjusted dynamically at runtime (Managed Functions, *F*).
- Multiple applications or instances can be managed at the same time (Haskell Admin, *F*).

Won't have

- Apart from Haskell, other programming languages are also supported (Haskell Admin, *F*).

2.4 Conclusion

This chapter provides a high-level specification of the software system that will be created. Use case analysis and requirements analysis were used to find the most important properties that the system should have.

In conclusion, the system will contain two independent parts – Haskell Admin and Managed Functions.

Haskell Admin will be an application with a graphical user interface, used to remotely manage Haskell applications running on a different machine. It should enable integration with existing projects and provide a security mechanism.

Managed Functions will be a software framework for remote management, also suitable for remote function calls. It will enable remote management using custom functions. Additionally, it should be easy to use for programmers and should support various data representations.

Both projects should be tested and prepared for further development as open-source.

Chapter 3

Design

This chapter focuses on designing a high-level architecture of the system being created (Section 3.1), and designing both parts of the system (Section 3.2, Section 3.3)

3.1 General Architecture

Previously in this thesis, a software architecture for remote management was described (Section 1.1.1), originally formulated by Kreger (2001). The architecture consists of *applications*, *agents* and *management systems*.

Both software projects created in this thesis (Haskell Admin and Managed Functions) realize the architecture. However, each of them is focused on a different part of it.

Haskell Admin consists of a *server side*, deployed together with the managed application, and a *client side*, deployed independently on the application (possibly on another device). The **client side** has the role of a management system, as described by Kreger (2001). It's a user-facing software restricted to a specific data transfer protocol (HTTP) and interface (Haskell Admin Web API). The **server side** has the role of an agent, directly managing the application and communicating with the management system.

The other project, **Managed Functions**, focuses on the application and agent level. Similarly to JMX, it provides a framework for adapting the application to various management systems. However, it doesn't provide any specific management system.

When used together, Managed Functions and Haskell Admin form a complete remote management solution, realizing all three levels of remote management (application level, agent level and management system level). A UML Component Diagram with the two projects being used together is shown in Fig. 3.1. The individual components and interfaces shown in the diagram are explained in the following sections.

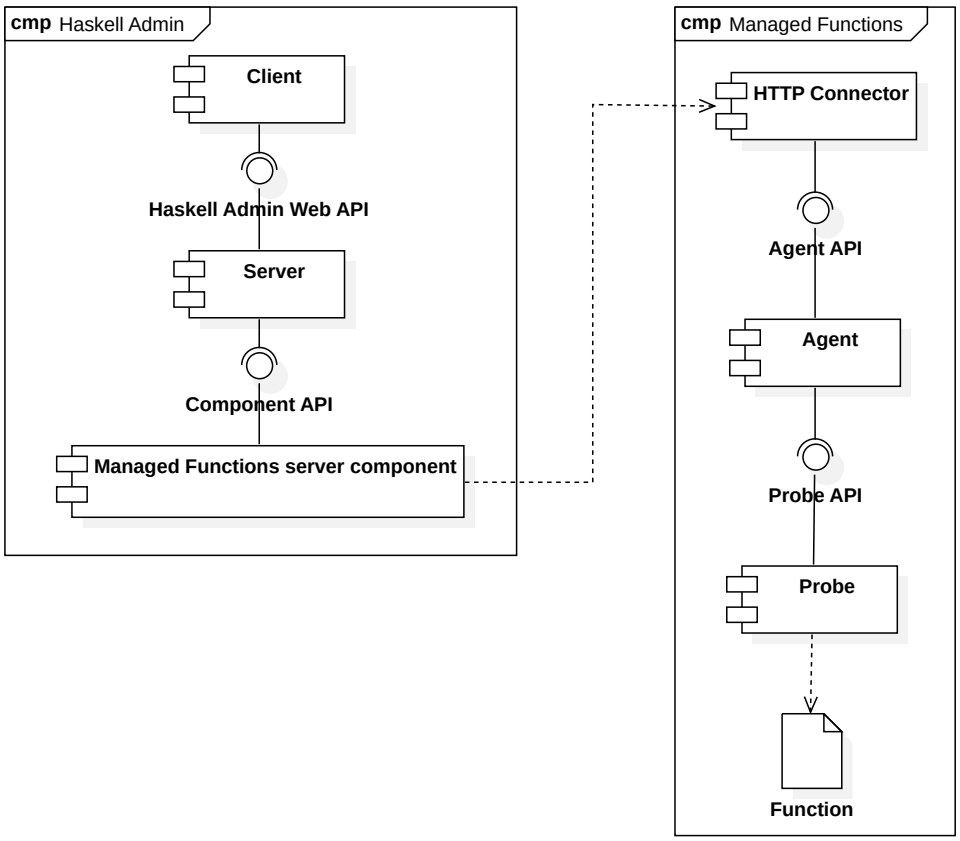

Figure 3.1: Component Diagram for Haskell Admin and Managed Functions

3.2 Haskell Admin

This section discusses the most important parts of Haskell Admin, in terms of design:

- the architecture (Section 3.2.1),
- a system of components on the server side (Section 3.2.2),
- the API exposed by the server side (Section 3.2.3),
- security concerns (Section 3.2.4),
- and the GUI (Section 3.2.5).

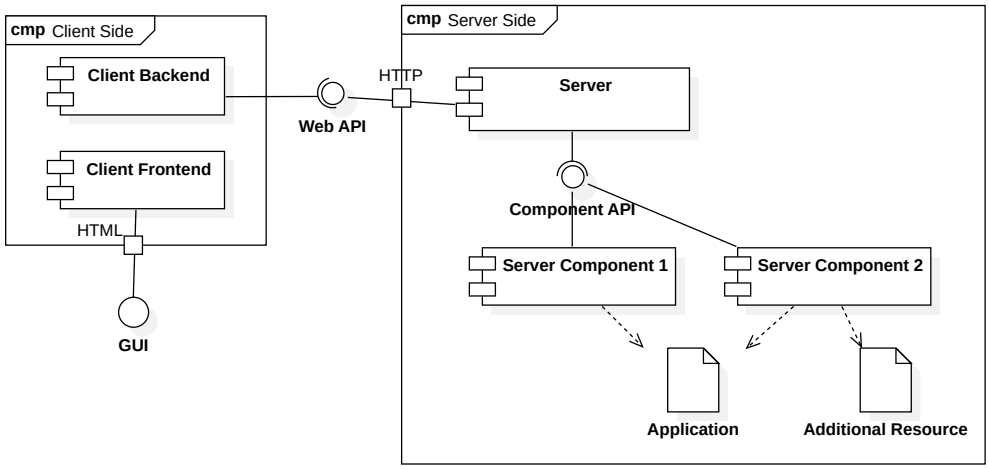

Figure 3.2: Component Diagram for Haskell Admin

3.2.1 Architecture

Fig. 3.2 shows the architecture of Haskell Admin using a UML Component Diagram.

The project is based on a traditional web application architecture. The client is a JavaScript web application running in the browser. The server exposes a RESTful API over HTTP.

The server is written in Haskell and is deployed together with the managed application. It contains a common core and various components dedicated to specific management tasks. It exposes an HTTP API containing both common and component-specific endpoints.

The client, written in HTML and JavaScript, runs in the user's browser and consumes the server's API. It's composed of two layers – the backend, which handles communication with the server, and the frontend, which handles the graphical user interface.

3.2.2 Components

The server side of Haskell Admin is composed of various *components*, each of which corresponds to a different management task or specialization.

For example, one component can be responsible for reporting application health and the ability to stop or restart the application, while another component can be used to edit environment variables.

All components are defined in terms of a common Component API, which determines the component's unique name, its API endpoints, and the linked management tasks. Also, every component can have its own representation in the client side GUI.

The selection of components can be easily adjusted for the specific use case – some applications may require specific components, that are not relevant for other ones.

The component-based approach enables existing solutions (such as EKG) to be easily included in Haskell Admin. On top of that, it makes Haskell Admin easily extensible by making it simple to create new components that can be included in the deployed solution.

3.2.3 Haskell Admin Web API

This section describes the HTTP API exposed by the Haskell Admin server.

The API contains two common endpoints – the root endpoint, and the component list, as shown in Table 3.1.

URL	Description	Type (REST)	Methods
`/`	Root endpoint	Resource	GET
`/components`	List of available components	Collection	GET

Table 3.1: Common API endpoints

The root endpoint doesn't contain any data. It's used to test the connection. The list of available components contains unique names of the components being used. It's used by the client to determine which endpoints are available.

Apart from the common endpoints, each component defines its own sub-API containing arbitrary endpoints. To prevent clashing URLs between different components, the sub-API is exposed under `/components/{name}/`.

3.2.4 Security

Haskell Admin provides an authentication mechanism that can be used to restrict access to the server API.

The HTTP protocol, supports a variety of standardized authentication schemes. Each scheme provides a different way to confirm the client's identity. [12]

One of the authentication schemes is the **bearer token** scheme. Originally, it's a part of the *OAuth 2.0* specification. However, it can be used on its own. In this scheme, every request made by the client contains an *access token* – a secret string, usually issued to the individual client. [13]

This scheme has been chosen for Haskell Admin, because it's not difficult to implement and provides reasonable security for an HTTP API. However, it's vulnerable to interception when used on HTTP (as opposed to HTTPS), because the secret bearer token is sent to the server unencrypted [13]. For this reason, and for general flexibility of the software, it may be desirable to support additional authentication schemes in the future. Notably, the *Oauth 2.0*

specification could be fully adopted, which would extend the bearer token scheme to a more robust and secure solution.

Because of security concerns, Haskell Admin must also support HTTPS, apart from simple HTTP.

In the following section, the bearer token is referred to as the *API key*.

3.2.5 Client and GUI

As previously stated, the client side of Haskell Admin is a browser-based application written in JavaScript and HTML. Contrary to a native application, this solution is cross-platform and easy to implement.

The client application consists of a *backend* part, which communicates with the server API, and a *frontend* part, which provides the GUI and handles user input.

The GUI has two screens:

1. the **Connection Screen**, used to establish a connection with a server,
2. the **Main Screen**, used to perform management tasks.

The Connection Screen contains a simple form, prompting the user to enter details about the server (host, port, and API key). Then, it allows the user to test the connection, confirm the entered data and proceed to the Main Screen.

The Main Screen takes inspiration from the GUI of Spring Boot Admin. It contains the sidebar, and the main display area. The main display area shows the UI of the currently selected tab. The sidebar enables the user to select a different tab. Tabs are grouped into *components*, which refer to the server-side components of Haskell Admin. Additionally, the Main Screen displays connection status and enables the user to disconnect from the server.

The proposed graphical design of the Connection Screen and the Main Screen is shown in Fig. 3.3 and Fig. 3.4, respectively.

Figure 3.3: Haskell Admin GUI – Connection Screen

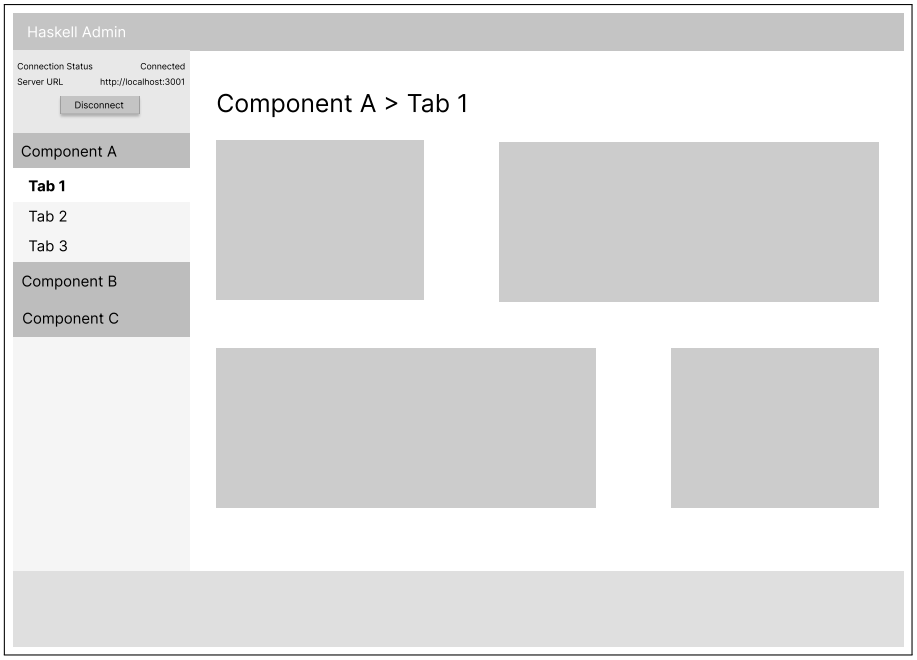

Figure 3.4: Haskell Admin GUI – Main Screen

3.3 Managed Functions

This section discusses the most important parts of Managed Functions design:

- the architecture (Section 3.3.1),
- the suitability for remote function calls (Section 3.3.2),
- the Haskell functions that can be used (Section 3.3.3),
- data representation concerns (Section 3.3.4, Section 3.3.6),
- the individual components of the framework, and the relations between them (Section 3.3.5),
- and security concerns (Section 3.3.7).

3.3.1 Architecture

The architecture of Managed Functions is heavily inspired by Java Management Extensions, and follows the remote management architecture discussed in Section 1.1.1, originally described by Kreger (2001).

It consists of three levels – the *Probe level*, the *Agent level*, and the *Connector level*, which correspond to JMX's *Instrumentation*, *Agent* and *Distributed services* levels, respectively.

On **Probe level**, individual functions are encapsulated with a common data type called a *Probe*. **Agent level** then serves as a registry of all the Probes, that can find and invoke them. It also handles any exceptions thrown during a Probe invocation. Finally, **Connector level** makes the Agent accessible to clients. Similarly to JMX, different Connectors exist to enable communication over various protocols and interfaces.

Fig. 3.5 shows the architecture of Managed Functions using a UML Component Diagram.

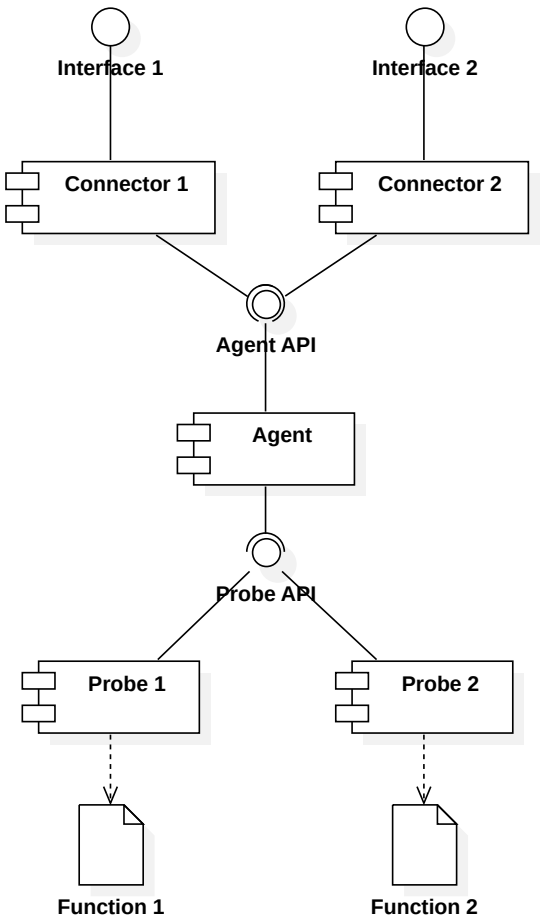

Figure 3.5: Component Diagram for Managed Functions

3.3.2 Remote Function Calls

This section discusses the suitability of Managed Functions for remote function calls.

A remote function call typically contains:

- a function name that the server (callee) can use to uniquely identify the function that should be called,
- serialized input data (function arguments).

In general, a remote function call with this structure can be executed as follows by the server:

1. Based on the function name, the server chooses a Haskell function `f`.

2. The input data are deserialized from the representation used in the remote call into a Haskell value `p` (with regards to the arity and type signature of the previously selected function `f`).

3. Once the function `f` and its argument `p` are determined, the RFC is executed by evaluating the expression `f p` (function `f` with argument `p`).

4. Depending on the exact RFC protocol used, the result may then be serialized and sent back to the client.

In Managed Functions, functions are encapsulated in Probes. The Agent then provides a common interface that can be used to call any of the functions in a uniform way, based on the function's name and arguments.

The common interface for calling functions makes Managed Functions applicable for implementing remote function calls – the framework provides data serialization, function lookup based on its name, argument application and error handling.

3.3.3 Suitable Types

This section determines the types of functions suitable for Managed Functions in terms of Haskell's type system.

A function encapsulated by a Probe can be either pure (without side effects), or it can execute side effects using the IO monad (the standard way of handling side effects in Haskell). As discussed in Section 1.2, enabling side effects in remote function calls (and application management in general) is generally desirable, as it allows the caller to work with the callee's environment.

Based on that, the function can take any number of input parameters and its output type is either a simple value or an IO action. In a Haskell-like pseudocode, the following type signatures are suitable for Managed Functions (Code snippet 1):

```
f_1 :: o
f_2 :: IO o
f_3 :: i_1 -> i_2 -> ... -> i_n -> o
f_3 :: i_1 -> i_2 -> ... -> i_n -> IO o
```

Code snippet 1: Type signatures suitable for Managed Functions

where `i_x` is the type of an input parameter and `o` is the output type.

The suitable data types are further constrained by the needs of communication between the caller and the callee, as explained below in Section 3.3.4.

3.3.4 Data Representation Concerns

Considering that the caller and the callee exchange data over a medium using some data representation (such as JSON, XML, or binary), the callee must additionally be able to convert between the representation and the input and output types. Two functions are needed for that – one for encoding (`encode`), and the other for decoding (`decode`). Code snippet 2 shows the type signatures of these functions.

```
encode :: o -> rep
decode :: rep -> i_x
```

Code snippet 2: `encode` and `decode`

That places a constraint on the input and output types, because some Haskell types cannot be serialized to a common data representation. For example, a function (`a -> b`) or a value wrapped in the IO monad (`IO a`) cannot be encoded in any suitable way, because `encode` and `decode` cannot be defined for these types.

On the other hand, this way of defining the input and output types makes it simple to utilize existing solutions for data serialization and deserialization. For example, if a JSON representation is used, `encode` and `decode` can be implemented in a very simple manner. Code snippet 3 demonstrates that by using the `ToJSON` and `FromJSON` type classes defined by the popular JSON serialization library *aeson*[x].

```
import Data.Aeson

encode :: ToJSON o => o -> ByteString
encode = Data.Aeson.encode

decode :: FromJSON i => ByteString -> i
Just decode = Data.Aeson.decode
```

Code snippet 3: `encode` and `decode` using Aeson

Similarly, the `encode` and `decode` functions can be implemented for other existing data representations using the appropriate type classes. Apart from *aeson*, another example is the *binary*[xi] library and its type class `Binary`, which contains very similar `encode` and `decode` functions.

Section 4.1.3 shows how this common pattern can be used to support multiple data representations in Managed Functions.

[x] https://hackage.haskell.org/package/aeson
[xi] https://hackage.haskell.org/package/binary

3.3.5 Framework Design

This section discusses the individual levels of Managed Functions and the relations between them. It also proposes Haskell types that may be used in the implementation.

3.3.5.1 Initial Assumptions

The following general assumptions about remote management are made to limit the scope of Managed Functions:

- In the context of any remote management solution, there is always a **client-server relationship** between the one who requests some tasks to be completed (client), and the one who executes them (server).
- The remote management process is composed of a series of tasks executed by the server on the client's demand. Task execution is coordinated by **transactions**.
- Every transaction consists of three steps:
  1. The client sends a **request** to the server.
  2. The server executes a **task** described in the request.
  3. The server sends a **response** to the client.
- **Requests** and **responses** are represented by a consistent generic format (e. g. JSON, plain text, or binary data).

Fig. 3.6 shows an overview of the communication scheme between the server and the client.

Properties of requests, responses, and suitable communication protocols will be examined below.

3.3.5.2 Communication Protocols

A communication protocol suitable for remote management as specified above has the following properties:

1. **client-server architecture**,
2. **transactional communication** – the communication is carried out as a series of requests and responses (as opposed to e. g. a communication based on a continuous bidirectional stream of data),
3. **uniform representation of data** – messages are sent in a uniform, consistent format, that can be either binary or text based. For simplicity, the rest of this section will assume a text based representation, and the generalization to other formats will be discussed in Section 3.3.6.

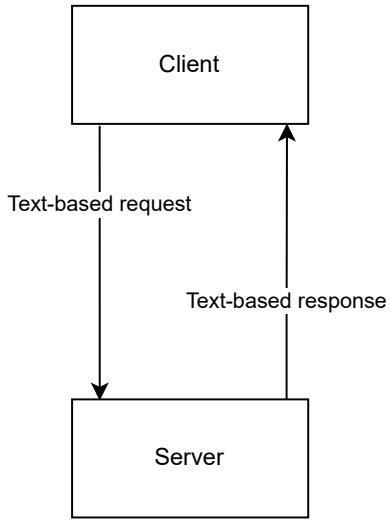

Figure 3.6: The transaction scheme

For example, HTTP is suitable, as it is a text based protocol that distinguishes between the client and the server, and the communication consists of individual requests and responses.

3.3.5.3 Request

In remote management, the managed application has a clearly defined set of tasks that can be executed remotely by sending a request to the remote management server.

The request must uniquely **identify** the task that should be executed, and may contain some **input data** for the requested task, unless the task can be executed without any input (a function with no input parameters)

The request must therefore contain two things:

- a unique identifier of the task,
- input parameters specific to the requested task, encoded as text.[xii]

Taken into Haskell terms, the following type can be used to describe a request (Code snippet 4):

```haskell
type Request = (ProbeID, String)
```

Code snippet 4: Type signature of a request

[xii] As discussed above, plain text encoding is considered here, and other options are described in Section 3.3.6

Where the first parameter is the task identifier (ProbeID can be a type synonym of String or a more complex structure), and the second parameter contains input data for the task.

3.3.5.4 Response

Similarly, a Haskell type for the server's response can be defined:

Considering that the response would eventually be sent back to the client encoded as text, it's convenient to use the String type. In order to determine the response, the requested server task must be executed, so the response type must be wrapped in the IO monad to enable side effects of tasks (Code snippet 5):

```
type Response = IO String
```

Code snippet 5: Type signature of a response

This type can be used to do any sort of computation with the result being representable as text.

3.3.5.5 Agent

As previously discussed, the Agent serves as a registry of Probes, and provides a common interface to invoke (call) them. The Probe invocation interface can now be defined using the Request and Response types, as shown in Code snippet 6:

```
invoke :: Request -> Agent -> Response

-- Without the Request and Response type synonyms:
invoke :: (ProbeID, String) -> Agent -> IO String
```

Code snippet 6: `invoke` function

The `Agent` type used in the type signature can simply be a container that can find a `Probe` based on its `ProbeID`, such as a `Map` from the *containers*[xiii] package:

```
type Agent = Map ProbeID Probe
```

Code snippet 7: Type signature (Agent)

The `invoke` function can be used for any kind of task execution that falls into the general transaction scheme described earlier. However, the Agent

[xiii] https://hackage.haskell.org/package/containers

can't be directly called by a client, because it's not tied to a certain communication protocol and cannot decode client requests on its own. To form a complete server solution, it needs the help of a Connector.

Apart from calling Probes, the Agent supports at least two more operations:

- listing Probes together with their `ProbeID`s,
- and providing additional information about a specific Probe.

Type signatures of the corresponding functions are shown in Code snippet 8.

```
toList :: Agent -> [(ProbeID, Probe)]
describe :: ProbeID -> Agent -> ProbeDescription
```

Code snippet 8: Type signatures of `toList` and `describe`

The `Probe` and `ProbeDescription` data types will be explained in Section 3.3.5.7.

3.3.5.6 Connector

A Connector mediates the communication between the Agent and a client (management system), by connecting the Agent API with a suitable communication protocol (HTTP, command-line interface, etc.)

It takes the Agent as an input parameter, and runs a server dedicated to the specific protocol. Its Haskell type signature can therefore be defined as follows (Code snippet 9):

```
newtype Connector
= Connector
{ run :: Agent -> IO ()
}
```

Code snippet 9: Type signature (Connector)

Fig. 3.7 shows an updated overview of the transaction scheme with Agent and Connector.

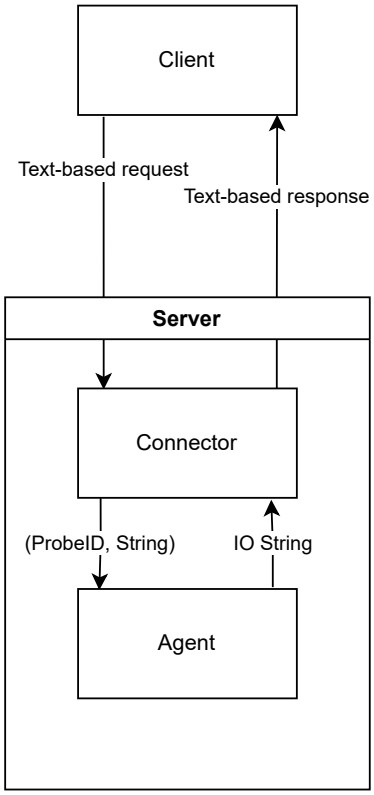

Figure 3.7: Transaction scheme (level 1)

3.3.5.7 Probe

As previously discussed, a server can handle requests using a Connector and an Agent.

As the next step, this section describes the *tasks* that can be requested, and a data structure that is used to encapsulate them – the Probe.

In Haskell, a management task is represented by a function that may:

- make use of input data, represented as a String,
- execute side effects in the IO monad,
- return the computation result in a way that can be sent back to the client – again, represented as a String.

Considering that, the Probe may have the type signature shown in Code snippet 10.

This type signature is sufficient for all the possible server tasks in the transaction scheme described above. Also, the type signature allows several

```haskell
type Probe = String -> IO String
```

Code snippet 10: Type signature (Probe)

common types of Haskell functions to be easily converted to a Probe. This will be further explained in Section 4.1.4.

To enable the Agent to provide details about a specific Probe, a Probe must additionally contain some information about the original function encapsulated in them. This is done using the `ProbeDescription` data type, that contains the original type of the function, and possibly some additional information. To enable that, the `Probe` type is extended to also contain the original type representation, and record syntax is used, as shown in Code snippet 11.

```haskell
data Probe =
Probe
  { call :: String -> IO String
  , typeRep :: TypeRep
  }

data ProbeDescription =
ProbeDescription
  { probeID :: ProbeID
  , probeType :: String
  }
```

Code snippet 11: Extension of the `Probe` type

3.3.5.8 Full Design Overview

Fig. 3.8 shows the complete transaction scheme of a server composed of the constructs described in the previous sections (Connector, Agent, Probe).

This architecture resembles Java Management Extensions (discussed in Section 1.5.1). It can be split into three levels analogous to those used in JMX (Instrumentation, Agent, Remote Management), as shown in Fig. 3.9.

3.3.6 Generalized Data Representation

The previous section assumed that the input and output types of a Probe are always `String` and `IO String`, respectively. However, in some cases, it is not practical to enforce these exact types. For example, if the Connector uses JSON for communication, and the exposed functions work with data types that have `FromJSON` and `ToJSON` instances, it's unnecessary to convert

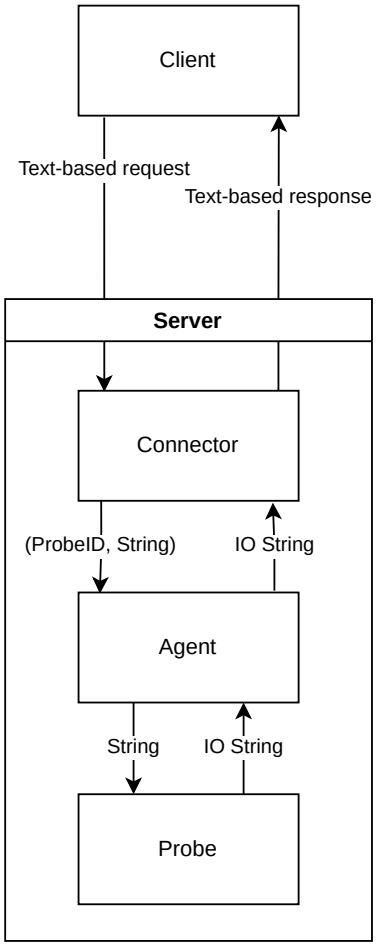

Figure 3.8: Transaction scheme (level 2)

the data from JSON to a `String`, and then back to JSON. The additional conversion may degrade performance, or be difficult to implement.

For this reason, the actual types used as input and output parameters are polymorphic – the data representation is determined at the type level. The Connector, the Agent and all Probes used together must support the same data representation, so that the relations between them still hold. The exact technique used to achieve that is based on type families, and is described in Section 4.1.3.

3.3.7 Security

As discussed multiple times in previous chapters, remote application management comes with two inherent security threats – unauthorized access to resources, and network interception.

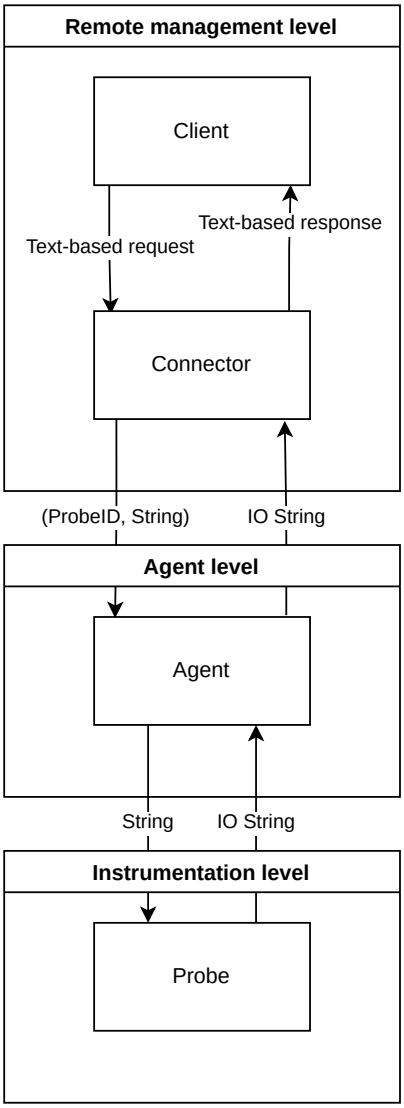

Figure 3.9: Transaction scheme (JMX analogy)

Often, the remote management system has access to sensitive resources or data through the agent, so it must be secured against unauthorized access.

Also, the management system communicates with the agent over a computer network, so the communication channel is potentially vulnerable to interception.

In the context of Managed Functions, the preferred way to secure a managed application against these threads is to use a secured Connector. For example, the Connector used by Haskell Admin supports HTTPS with the bearer token authentication scheme, which prevents both unauthorized access and interception.

In JMX, a system of permissions is used to restrict access to individual services and MBeans [9]. Creating a similar system for Managed Functions would make it more robust in terms of security. However, for the sake of simplicity, it isn't currently included, because secure Connectors alone can mitigate the risks sufficiently.

3.3.8 Conclusion

The previous sections described in detail how the levels of Managed Functions work together and how the framework can be used for remote management.

Also, some Haskell data types were mentioned, that could be used in the code. However, the actual implementation uses slightly more sophisticated data types to enable different data representations to be used, and to solve some implementation-specific problems, that are not that closely related to the overall design. The actual data types used are described in Section 4.1.5.

Chapter 4

Realization

Based on all the topics covered in the previous chapters, Haskell Admin and Managed Functions have been implemented. The result is included in the attachment of this thesis.

This chapter discusses the most important parts of the implementation and the decisions made during the realization process.

4.1 Managed Functions

This section covers the key parts of Managed Functions implementation. Mainly, it focuses on how Probes have been defined, because that is the most complex and innovative part of the project.

4.1.1 Project Structure

Managed Functions are implemented in the form of a Haskell package, that can be easily published to a package repository (such as the *Hackage*[xiv]), and used in other projects as a dependency.

There are two Haskell build tools available, that could have been used to create the package – *Cabal*[xv] and *Stack*[xvi]. The package was created using Stack, because it provides better support for managing local dependencies, that haven't yet been published to a package repository, and for building a project composed of several packages. Managed Functions is a dependency of Haskell Admin, and the packages have been developed simultaneously, so it meant a significant difference.

The package consists of a **library**, a **test suite**, and an **example executable**. The executable demonstrates how the framework should be used.

[xiv] https://hackage.haskell.org/
[xv] https://www.haskell.org/cabal/
[xvi] https://docs.haskellstack.org/

However, it can be omitted when the package is published, or alternatively moved to a separate package, as it's not a part of the framework.

The library is divided into small modules, dedicated to specific parts of the implementation, in accordance with the single-responsibility principle (SRP).

4.1.2 Probe API

Several options were considered for the type signature of the `call` function used to call a Probe. Calling a Probe is the most fundamental operation of the Managed Functions framework, so a significant amount of work has been put into defining the most suitable implementation. The following observations have been made:

- All Probes used together by a specific Agent should have the exact same type signature. Otherwise, the Agent would have to implement polymorphic functions for interacting with Probes, meaning that working with Probes wouldn't be any simpler than working directly with the underlying functions.
- The type signature should be as generic as possible, so that there is no unnecessary restriction on the type of the encapsulated function.
- Functions should be easily convertible to Probes.
- The user should be able to use any data representation (binary, JSON, XML, etc.), as long as it's defined for the input and output types of the encapsulated function.
- The type signature should respect that Haskell functions can have any number of parameters.

Based on that, several options for the type signature have been identified. The individual options are listed in Code snippet 12, and explained below:

The first option was to "hard-code" a certain data type `a` (such as `String`), that would serve as both input and output. In that case, the type signature could be `String -> String`, or perhaps `String -> IO String`. Additionally, multiple parameters could be supported by changing the input type to a list – `[String] -> IO String`. However, this approach is not very suitable for the different data representations, as it forces all data to be converted to a `String`.

A different approach would be to make the Probe data type generic by introducing type variables `a` and `b`, determining the input and output types, respectively. The type signature would then be simply `a -> b`. The main disadvantage of this approach is that the Probes wouldn't have the same type any more, and the Agent would have to use polymorphic functions to interact with them.

To ensure that all the Probes are of the same type, while enabling various input and output types for the `call` function, a combination of type classes and universal qualification could be used. Precisely, two type classes would be defined – `ProbeInput` and `ProbeOutput`. Every data representation would have an instance of these classes, and the `call` function would accept any of them. The resulting type signature is shown in Code snippet 12. The downsides of this option are that every Probe would have to support every data representation, making it more difficult to define Probes, and that there could be only one instance of the class for every type used to represent the data – for example, if the `ByteString` type was used to store binary-encoded data, it would be "taken" and couldn't be used to e.g. store JSON.

The most advanced option, which is used in the actual implementation, is based on a combination of generic data types, type classes, and type families. It follows all the expectations written above by providing a balance between genericity and uniformity of the different Probes. Here, the Probe data type is generic. It has one type variable `e`, which determines the data representation used. The type variable must have an instance of the `Encoding` type family. Based on the instance declaration, the exact input and output types are determined. The `Encoding` type family is further discussed in Section 4.1.3. This approach fits the expectations very well – Probes can be defined easily, the encapsulated functions can be of any type, as long as their individual parameters can be represented with the chosen encoding, and all Probes with matching encodings have matching interfaces.

```
-- Hard-coded types
Probe = Probe {call :: String -> IO String}
-- Multiple parameters
Probe = Probe {call :: [String] -> IO String}
-- Generic data type
Probe a b = Probe {call :: a -> b}
-- Using type classes
Probe = Probe {
  call :: forall a b. (ProbeInput a, ProbeOutput b) => [a] -> b
  }
-- Using the Encoding type family
Probe e = {call :: [In e] -> IO (Out e)}
```

Code snippet 12: Different options of a Probe call type

In conclusion, a rather complex, but very powerful approach was selected for the Probe's `call` function. As long as all Probes used together in a specific application are based on the same data representation (which is usually the case), the selected type annotation is the most practical of all the considered options.

4.1.3 Data Encoding

This section describes the approach that was taken to handle different data representations, and the related problems of converting various data types between the Haskell internal representation, and the representations that can be used for data transfer.

Existing Haskell libraries provide sufficient ways of encoding and decoding values of various data types to a common representation, such as JSON, binary, or plain text. These libraries typically define some data type that can store encoded data, and two type classes to handle the encoding and decoding, respectively. Instances of some basic types, such as the predefined primitives, are typically included in the library, and users can define their own instances as needed. Also, the `Generic` type class exists, which provides a mechanism to automatically derive instances for most data types. The type system used in Haskell is based on algebraic data types, so it's usually possible to automatically generate encoding functions.

The described scheme, consisting of the two type classes for encoding and decoding, and a common representation data type, can be generalized, as shown in Code snippet 13.

```
class Encode t rep where
encode :: t -> rep

class Decode t rep where
decode :: rep -> Maybe t
```

Code snippet 13: `Encode` and `Decode` type classes

In the context of Managed Functions, this scheme can be taken a step further. A problem with using the `Encode` and `Decode` classes directly is that there can be two distinct use cases of the framework, that both use the same type for representation (e.g. `ByteString`), but the logical data representation is different (a `ByteString` is commonly used to store binary-encoded data, as well as JSON-encoded data). The `toProbe` type class provided by Managed Functions should respect this ambiguity, and users of the framework should be able to easily specify which encoding they want to use. Additionally, a Probe could theoretically accept arguments encoded in one representation, and return a result encoded in a different one, which may be beneficial for some use cases.

To satisfy these cases, Managed Functions define the `Encoding` type family, which is used to indicate the exact data representation. The class contains two types – the `In` type for Probe input (call arguments), and the `Out` type for Probe output (call result). Code snippet 14 shows the implementation of `Encoding`.

```
class Encoding a where
  type In a :: Type
  type Out a :: Type
```

Code snippet 14: `Encoding` type class

The previously discussed `Encode` and `Decode` type classes can then be redefined using the `In` and `Out` types – see Code snippet 15. Example instances of `Encoding`, `Encode` and `Decode` are shown in Code snippet 16 – notice that the `JSON` data type doesn't contain any data, and only serves as a symbol denoting the data representation used.

```
class Encode t rep where
  encode :: t -> Out rep

class Decode t rep where
  decode :: In rep -> Maybe t
```

Code snippet 15: `Encode` and `Decode` using `In` and `Out`

```
data JSON

instance Encoding JSON where
  type In JSON = Value
  type Out JSON = Value

instance (ToJSON a) => Encode a JSON where
encode = toJSON

instance (FromJSON a) => Decode a JSON where
  decode v =
  case fromJSON v of
    Error _ -> Nothing
    Success x -> x
```

Code snippet 16: Example instances of `Encoding`, `Encode`, and `Decode`

The next section shows how the `Encoding` data family is used to elegantly generate a Probe from any suitable function.

4.1.4 ToProbe Type Class

The previous sections described how the `Probe` data type was defined using the `Encoding` type class. This section discusses a mechanism that is used to convert functions to Probes with minimal effort.

The mechanism is based on the `ToProbe` type class. The type class has two parameters, `fn` and `e`. `fn` represents a function type, and `e` represents an `Encoding`. The class has only one method called `apply`. The method reads arguments from a list, converts them to the appropriate data types using the provided encoding `e`, applies them to `fn`, and converts the result to the desired representation, again using the encoding `e`. Code snippet 17 shows the exact definition of the class.

```
class ToProbe fn e
  where
  apply :: Proxy e -> fn -> [In e] -> IO (Out e)
```

Code snippet 17: `ToProbe` type class

The `ToProbe` class has two base instances. One is for a constant function with no parameters `a`, that has an instance of `Encode a e`, meaning that it's a constant value that can be encoded to the output representation. The other one is for `IO a` – an effectful action that yields a value of type `a`.

A third instance of the class exists, which recursively defines the `apply` function for functions with parameters, using partial application. It takes a single argument from the argument list, decodes it, partially applies it to the function, and calls `apply` on the result of the partial application, passing the remaining arguments.

The instances are shown in Code snippet 18. The code has been simplified to show the general idea. The full implementation can be found in module `Managed.Probe.ToProbe` of the *managed-functions* package.

```
instance (Encode a e) => ToProbe a e where
  apply _ c [] = return $ (encode @a @e) c

instance (Encode a e) => ToProbe (IO a) e where
  apply _ c [] = (encode @a @e) <$> c

instance (Decode a e, ToProbe b e) => ToProbe (a -> b) e where
  apply _ f (x:xs) = do
    r <- decode @a @e x
    apply (Proxy @e) (f r) xs
```

Code snippet 18: `ToProbe` instances

Additionally, the `toProbe` function is defined, which is used to create Probes in practice. It uses the `apply` function at its core, and creates a Probe of the appropriate type. Finally, this function can be used to very conveniently convert any suitable function to a Probe. A usage example is shown in Code snippet 19 – note that the `JSON` type in the Probe's type signature is an instance of `Encoding`, and denotes what data representation, and which `encode` and `decode` functions are used.

```
myFn :: Int -> Char -> String
myFn = Prelude.replicate

myProbe :: Probe JSON
myProbe = toProbe myFn
```

Code snippet 19: Usage example of `toProbe`

In conclusion, the `ToProbe` type class provides an elegant mechanism for creating Probes without any unnecessary burden for the programmer.

4.1.5 Actual Data Types

This section covers the actual data types that are used for Probes, Agents and Connectors in the implementation. All three of them are generic with one parameter, that denotes the data representation used. The generic parameter ensures that the data representation is consistent across all Probes exposed by an individual Connector.

The `Probe` data type (Code snippet 20) contains two fields:

- the `call` function, which is used to call the encapsulated function,
- the `typeRep` field, which contains the type of the encapsulated function, accessible at runtime.

```
data Probe e =
Probe
  { call :: [In e] -> IO (Out e)
  , typeRep :: TypeRep
  }
```

Code snippet 20: Actual `Probe` data type

The `Agent` type (Code snippet 21) is a synonym of `Map ProbeID (Probe e)`, and `ProbeID` is a synonym of `String`.

The `Connector` data type (Code snippet 22) contains a single field – the `run` function, that takes an Agent as a parameter. That enables an intuitive syntax, as shown in Code snippet 23.

```
type ProbeID = String

type Agent e = Map ProbeID (Probe e)
```

Code snippet 21: Actual `Agent` data type

```
newtype Connector e =
Connector
  { run :: Agent e -> IO ()
  }
```

Code snippet 22: Actual `Connector` data type

```
myConnector :: Connector JSON
myConnector = Connector { run = \agent -> ... }

myAgent :: Agent JSON
myAgent = ...

main :: IO
main = run myConnector myAgent
```

Code snippet 23: Connector usage example

Additionally, there is a `ProbeDescription` data type (Code snippet 24), used to describe a `Probe` and its underlying function. It contains four fields:

- `probeID` identifies the `Probe` in the context of a specific `Agent`,
- `probeType` is a human-readable type signature of the encapsulated function,
- `probeParams` describes the types of the parameters of the function,
- `probeReturs` describes the output type of the function.

The `Agent` can be queried for a `ProbeDescription` of a specific Probe.

```
data ProbeDescription = ProbeDescription
  { probeID :: ProbeID
  , probeType :: String
  , probeParams :: [String]
  , probeReturns :: String
  }
```

Code snippet 24: Actual `ProbeDescription` type

4.1.6 Expected Usage

To use the framework for remote management, several steps are generally taken:

1. A data representation is selected, such as JSON, binary, or plain text.
2. Target functions, that should be later exposed by the Agent, are converted to Probes, preferably using the `toProbe` function.
3. The resulting Probes are passed to a newly created Agent.
4. The Agent is passed to a Connector that supports the selected data representation.
5. The Connector can then be run.

A concrete example is shown in Code snippet 25. The example uses the predefined `SR` data representation, which encodes the data as a simple `String`, and uses the widely used `Show` and `Read` type classes. As a Connector, the `httpConnector` is used, which is also used in Haskell Admin. Code snippet 26 and Code snippet 27 show the HTTP request and response, respectively, of calling the `plus` Probe.

```haskell
-- Probe definitions
probes :: [Probe SR]
probes =
  [ toProbe (reverse @String)
  , toProbe readFile
  , toProbe ((+) @Float)
  ]
-- Agent creation
ag :: Agent SR
ag = fromList $ zip ["reverse", "readFile", "plus"] probes
-- Running a Connector
main :: IO ()
main = run httpConnector ag
```

Code snippet 25: Usage example of Managed Functions

```
POST /probes/plus/invoke HTTP/1.1
Content-Type: application/json
Content-Length: 11

["3.4","1"]
```

Code snippet 26: HTTP request to invoke the “plus” probe

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=utf-8

"4.4"
```

Code snippet 27: HTTP response to Code snippet 26

4.2 Haskell Admin

This section covers the key parts of Haskell Admin implementation, and discusses the problems encountered during development. The server side and the client side are covered by Section 4.2.2 and Section 4.2.5, respectively. Also, the topic of server side components is explained in Section 4.2.3, and security concerns are discussed in Section 4.2.4.

4.2.1 Project Structure

Haskell Admin consists of a Haskell package on the server side, and a standalone application written in JavaScript on the client side.

Haskell wasn't used for the client, even though there is a Haskell-to-JavaScript compiler, called *GHCJS*[xvii], that could have been used to write the browser-based application in Haskell. Using GHCJS would have enabled the client application to reuse the data types and functions defined by the server side. However, GHCJS comes with several challenges – most importantly, compilation is non-trivial, not all Haskell libraries are supported, and there is a limited offer of usable front-end frameworks.

So, instead of using GHCJS, the client has been written in JavaScript, using the *Svelte*[xviii] frontend framework. Svelte was chosen, because it provides handy features that the project uses (state management, binding variables to UI elements, UI components, scoped CSS), but it doesn't introduce unnecessary complexity to this rather small project.

[xvii] https://github.com/ghcjs/ghcjs
[xviii] https://svelte.dev/

4.2.2 Server Side

The server side of Haskell Admin consists of the web server exposing an HTTP API, and individual components, that interact with the managed application in various ways.

A major part of the implementation is dedicated to the web API server. For that reason, it was important to choose a robust web framework.

The *Servant* framework has been chosen for this task. The main distinction of this framework is that the API is defined at the type level [14]. This feature brings several advantages relevant to Haskell Admin:

- The individual request handlers are forced to correspond to the API [14].
- Type-level programming can be used to create a consistent system of server components, as further discussed in Section 4.2.3.
- API documentation, and even functions for querying the API, can be automatically generated [14].

4.2.3 Components

This section discusses how a system of server side components has been implemented (Section 4.2.3.1), and individual components that have been implemented so far (Section 4.2.3.2, Section 4.2.3.3, Section 4.2.3.4).

4.2.3.1 Server Side Components

The server components have been implemented with extensibility in mind – defining new components is very simple, and existing software can be easily integrated. Also, the type-level definition of a Component enforces the HTTP API to be stable, and to separate the endpoints of different components in a systematic way.

A component consists of

- a name, defined as a `Symbol` (type level string literal),
- an API, defined as a Servant API, also at the type level,
- a server, defined as an expression, whose type depends on the component's API, and makes use of Servant's `Server` type.

The resulting data type of a component is shown in Code snippet 28.

```
newtype Component (name :: Symbol) api
  = Component { server :: Server api }
```

Code snippet 28: `Component` data type

A similar data type can be defined for a list of components (Code snippet 29).

```
newtype ComponentList (names :: [Symbol]) apis =
  ComponentList
    { serveAll :: Server apis
    }
```

Code snippet 29: `ComponentList` data type

Additionally, type-level programming was used to enforce the API structure – all endpoints defined by a component's API must be located under `/components/{name}/`. The exact implementation can be found in the module `Admin.Components.Internal`.

What's worth mentioning is the `with` function, used to compose components in accordance with the expected API structure (Code snippet 30). The function uses type-level operators to compose the API with the expected structure, and to collect a list of the component names. It creates a simple interface for the user to choose which components are used in the specific Haskell Admin instance.

```
with ::
     (Components a names apis)
  => Component name api
  -> a
  -> ComponentList (name : names) ((name :> api) :<|> apis)
with new lst = ComponentList (server new :<|> serveAll' lst)
-- Usage:
-- >>> componentA `with` componentB `with` componentC
```

Code snippet 30: `with` function

4.2.3.2 System Component

The System component is responsible for monitoring the state of the application. If an exception is raised in the application, the program doesn't crash, and Haskell Admin shows the error message in the GUI. This is achieved by running the application in a separate thread, utilizing the *async*[xix] library.

The component exposes a single endpoint, `/system/status`, that represents the immediate application state (running, error, or finished).

The type-level definition of the component is shown in Code snippet 31.

[xix] https://hackage.haskell.org/package/async

```
type SystemAPI = "status" :> Get '[ JSON] SystemState

type SystemComponent = Component "system" SystemAPI
```

Code snippet 31: System component

4.2.3.3 Managed Functions Component

The Managed Functions component is a thin wrapper around the `httpConnector` defined in the Managed Functions framework. The component can be used to invoke arbitrary functions in the managed application.

Haskell values are encoded with the `SR` encoding, which uses the `Show` and `Read` type classes – this encoding is appropriate for interaction with a human user, because the exact form of the encoded data is usually identical to how the data would be written in Haskell code.

The type-level definition of the Managed Functions component is shown in Code snippet 32.

```
type ManagedAPI = HTTPConnectorAPI

type ManagedComponent = Component "managed" ManagedAPI
```

Code snippet 32: Managed Functions component

Fig. 4.1 shows how the resulting GUI looks. The screenshot shows the result of using Managed Functions to read the contents of a file (the function `readFile` was called with the argument `/tmp/log.txt`).

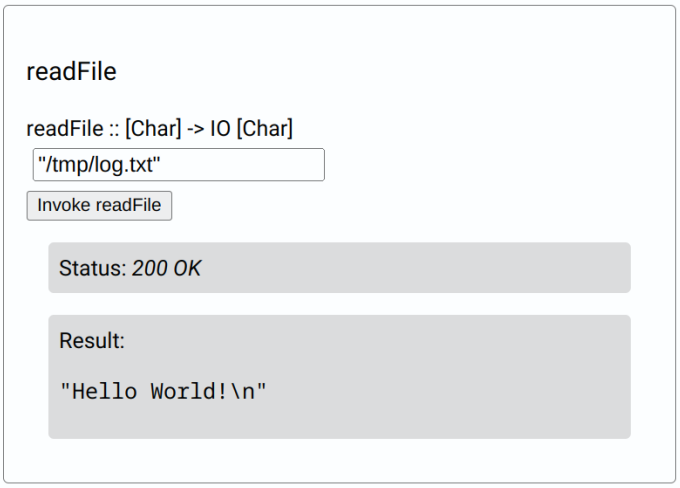

Figure 4.1: Haskell Admin screenshot (Managed Functions)

4.2.3.4 EKG Component

The EKG component demonstrates how existing software can be integrated into Haskell Admin. It uses the previously mentioned *ekg* library to monitor various metrics about the application. Similar to the Managed Functions component, the EKG component is a thin wrapper around existing software. The API, which is identical to the original JSON API exposed by EKG, consists of a single endpoint, that contains the immediate values of all the metrics.

The type-level definition of the component is shown in Code snippet 33.

```
type EkgApi = Get '[ JSON] Value

type EkgComponent = Component "ekg" EkgApi
```

Code snippet 33: EKG component

4.2.4 Security

In the implementation of Haskell Admin, two security concerns have been dealt with – bearer token authentication, and Cross-Origin Resource Sharing (CORS).

4.2.4.1 Bearer Token Authentication

Bearer token authentication was implemented according to the specification [13]. Every HTTP request made by the client must contain the `Authorization` header with a bearer token. The token is then checked on the client side for correctness. In response, the request is either passed to the appropriate handler, or an error page is returned with the `401 Unauthorized` HTTP response status code.

Servant provides a library called *servant-auth*[xx], that can be used to embed the authentication logic into the server definition. An attempt was made to utilize this library for Haskell Admin. However, it was found to be unnecessarily complicated. So instead, the authentication logic was implemented as a *WAI*[xxi] middleware. The `Network.Wai.Middleware.HttpAuth` package was used as a starting point, and the code was adapted to fit the use case. To conclude, a simple bearer token authentication solution was implemented on the entire API, that works on the basis of comparing the `Authorization` HTTP header with a list of allowed bearer tokens (API keys), provided by the user.

[xx] https://hackage.haskell.org/package/servant-auth

[xxi] Web Application Interface (WAI) is the underlying protocol used by Servant, as well as many other Haskell libraries (see https://hackage.haskell.org/package/wai).

4.2.4.2 Cross-Origin Resource Sharing

The client side of Haskell Admin runs directly in the user's browser. As a browser-based application, it must follow some specific security rules, dictated by the browsers. Most importantly, Cross-Origin Resource Sharing (CORS) must be properly configured on all servers that the application communicates with.

Initially, the application must follow the so-called same-origin policy, which means that it can only request resources from its own *origin* – the resources must be located on the same protocol, port, and host. [15]. Generally, that means that all requests to external APIs are forbidden.

To enable requests to a different origin, the server must include the application's origin in the `Access-Control-Allow-Origin` HTTP header.

The Haskell Admin server allows requests from any origin, because the client application can be hosted anywhere, or even run locally from any device. So, the appropriate HTTP header (shown in Code snippet 34) is sent with every request, allowing the client application to have any origin.

```
Access-Control-Allow-Origin: *
```

Code snippet 34: CORS HTTP header

The header is implemented using a specialized WAI middleware from the *wai-cors*[xxii] library.

4.2.5 Client Side

The client side of Haskell Admin has been developed as a browser-based JavaScript application using the Svelte framework. It can be deployed locally on a personal device, or it can be deployed on a server and accessed using the browser. Contrary to the server side, it is a complete application that doesn't need any further setup to be used.

The GUI was implemented in accordance with the design proposed in Section 3.2.5. The only significant difference is that in the implementation, the user can choose between HTTP and HTTPS on the Connection screen.

Fig. 4.2 and Fig. 4.3 show screenshots of the Connection screen, and the Main screen, respectively.

[xxii] https://hackage.haskell.org/package/wai-cors

Figure 4.2: Haskell Admin screenshot (Connection screen)

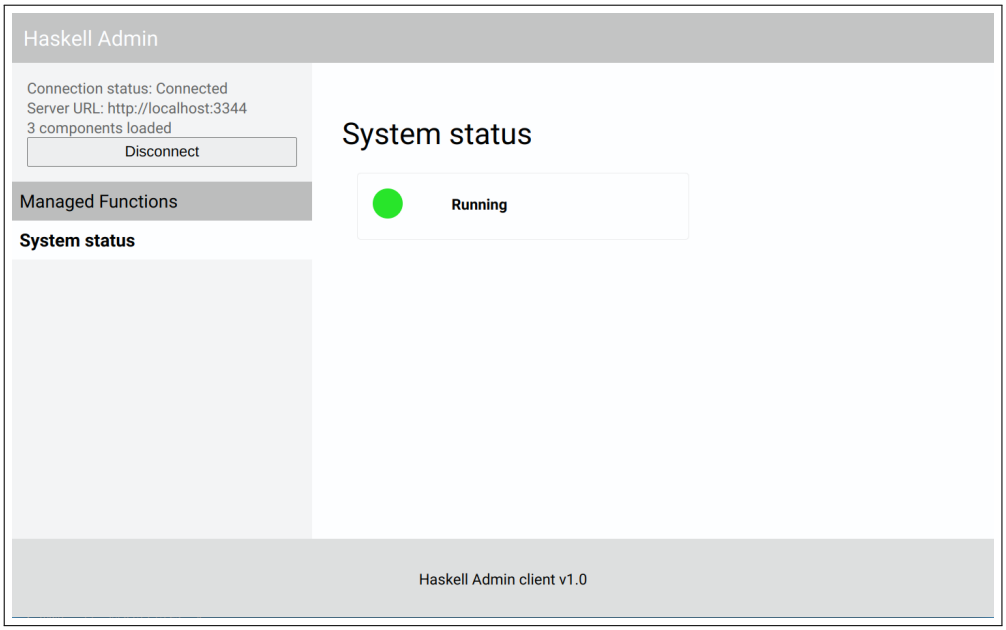

Figure 4.3: Haskell Admin screenshot (Main screen)

4.2.6 Expected Usage

The primary usage scenario assumes that there is an existing Haskell application `app :: IO ()`, that is supposed to be managed using Haskell Admin.

A minimal complete usage example is shown in Code snippet 35. As shown in the snippet, integrating Haskell Admin is quite simple, and, more importantly, it doesn't require any change in the original definition of `app`.

After starting the server, the client application can be connected. The GUI first shows the Connection screen, where the connection can be configured and tested. After that, the Main screen is shown and supported components are automatically loaded. Management tasks can then be performed.

```haskell
{-# LANGUAGE OverloadedStrings #-}
module Main where

import Admin
import Network.Wai.Handler.Warp (run)
import Admin.Components.System (system, async)

-- The original 'main' function of the app
app :: IO ()
app = undefined

main :: IO ()
main = do
  -- Run the app in the 'async' wrapper
  formerMain <- async app
  -- Define an API key for the client application
  let apiKeys = ["myApiKey"]
  -- Only the System component is used in this example
  let components = system formerMain
  -- Run the Haskell server on port 3344 in the main thread
  run 3344 $ admin apiKeys components
```

Code snippet 35: Minimal example of Haskell Admin server

The Haskell Admin server has the form of a WAI application – it provides a standard interface commonly used for Haskell web applications. Thanks to that, the user can conveniently change low-level details about the deployment, such as the port, use a custom web server, or set up HTTPS. Typically, WAI applications are served using the *Warp*[xxiii] server – this approach is used in Code snippet 35.

[xxiii] https://hackage.haskell.org/package/warp

4.3 Testing

Both projects have been tested using a suite of unit tests. The *Hspec*[xxiv] framework was used. Additionally, dedicated test programs have been written for both projects, that were used to manually verify the correct behavior. The test suite and programs can be found in the source code.

4.4 Evaluation

This section describes how the projects were evaluated, and whether there were any problems found with their expected real-world usage.

4.4.1 Haskell Admin

To evaluate Haskell Admin, the following use case scenario has been executed:

- A sample Haskell application was created as a new package.
- The *haskell-admin* package was added as a dependency to the sample application, and the `main` function was modified to use Haskell Admin.
- The sample application was deployed on a remote server.
- The Haskell Admin client application was deployed on a different remote server.
- The connection and interactions between the two applications were manually tested.

Both applications were deployed in a Docker container, which is the expected way of deploying the software in real-world usage.

Both HTTP and HTTPS protocols were tested. HTTP usage was very simple and straightforward. Contrary to that, using HTTPS required a significant amount of preparation. Most importantly, two valid TLS certificates had to be issued and set up, one for each of the servers. Setting up HTTPS on any server is quite tedious. However, it only has to be done once at the beginning.

The evaluation deployment confirmed that there are no unexpected obstructions for real-world usage of Haskell Admin.

[xxiv] https://hspec.github.io/

4.4.2 Managed Functions

Managed Functions were evaluated by creating various sample usages of the framework, including Haskell Admin itself. It has been found that functions can be indeed very simply exposed using Managed Functions, and there is no problem with using the common data representations.

Additionally, the suitability for remote function calls was evaluated by creating a Connector for the *JSON-RPC* specification. Only part of the specification was implemented, to evaluate the core functionality. Compared to existing Haskell libraries, Managed Functions provide the most practical API for defining the functions exposed to remote calls.

4.5 Requirements Fulfillment

This section evaluates whether the requirements identified in Section 2.3 have been fulfilled by the implementation.

All the requirements rated as "must have" have been fulfilled.

The "should have" requirements have all been generally fulfilled as well. However, there are two requirements that should be further worked on:

- "The system is tested well" – unit test coverage can certainly be improved.
- "The user can choose between various data representations" – there are two data representations available in Managed Functions – `JSON`, and `SR`. Additionally, a binary representation can be implemented.

None of the "could have" requirements have been fulfilled. However, all of them could be implemented without any significant changes to the system architecture.

The "won't have" requirement of supporting additional programming languages isn't likely to ever be implemented, because the system is heavily based on the specifics of Haskell.

4.6 Further Development

Both projects have been prepared for further development as open-source.

In terms of Managed Functions, the core functionality is finished, but only one meaningful Connector has been created so far. There is a significant potential in extending the spectrum of Connectors, especially for RFC specifications, such as *JSON-RPC* or *gRPC*. Also, additional data encodings could be provided – currently, there is `JSON` and `SR`.

In terms of Haskell Admin, there is a potential in creating more server components, and improving the client application. Additionally, more authorization options could be provided. The current implementation is indeed

usable, but it's very limited in the actual management tasks that can be performed. However, the System and Managed Functions components have been fully implemented, and are ready to be used in production environments.

Apart from that, the unrealized "could have" requirements can be implemented – most importantly, a notification system for Haskell Admin would be desirable.

Chapter 5

Conclusion

This work was dedicated to examining, evaluating, and improving the state of remote application management in the Haskell language. Two software projects have been created in the process – Managed Functions and Haskell Admin.

In the theoretical part of the work, it has been found that there is a lack of remote management software that can be used for Haskell applications. Also, in the context of functional programming, a significant link between remote management and remote function calls has been found. Existing software was analyzed, and a successful software architecture, that is used for remote management by the JMX technology, was adapted to functional programming.

The Haskell Admin remote management system was created. The system has great potential for real-world use, because there isn't any other similar software dedicated to remote management of Haskell applications. Also, it has been shown that the system is easily extensible, and existing Haskell software can be integrated quickly.

Also, the Managed Functions framework was created. An innovative approach to data representation was used, that could be potentially used in other projects as well. Compared to other RFC libraries available for Haskell, Managed Functions offer a simple interface – particularly, exposing many different functions is simple, as the framework can fully handle data conversion between the internal representation and an explicitly defined encoding.

Both projects have been tested, evaluated, and prepared for further development as open-source.

5.1 Goals Fulfillment

The main goal of this thesis was to create "a software solution allowing remote management and remote function calls with the least possible burden for programmers." This goal has been fulfilled, the software has been evaluated as ready for production use.

In Chapter 1, remote management and remote function calls have been analyzed, with a focus on existing Haskell software, as well as JMX. Also, the specifics of Haskell, and functional programming in general, were discussed.

Chapter 2 provided a high-level overview of the software system to be created, and how it can improve remote management of Haskell applications. Specific requirements have been identified, that the system should fulfill.

Based on the analysis, Chapter 3 designed the general software architecture, as well as the specific components of the software system to be created.

Then, Chapter 4 described selected parts of the implementation, that are particularly interesting or innovative. Apart from that, it described the implementation in general, so the text can serve as a complementary documentation of the source code.

In conclusion, the thesis fulfilled all of the goals, that had been set, in terms of analysis, design, and realization.

5.2 Future Work

Future work could focus on the specifics of functional programming for remote application management, as most existing work focuses on the object-oriented paradigm.

Also, there are many existing publications dedicated to the topic of distributed computing in Haskell, so the relation between distributed computing, remote function calls, and remote application management could be examined by future work.

Finally, a unique approach to managing different data representations in Haskell was used in the Managed Functions project. This topic could be further explored. Perhaps, a separate Haskell library could be created, that would define a common interface for all the existing data representations.

Bibliography

[1] Donsez, D. The JMX 3-level architecture. [online], 2021, [accessed 15.4.2022]. Available from: https://commons.wikimedia.org/wiki/File:JMX_Architecture.svg

[2] Kreger, H. Java Management Extensions for application management. *IBM Systems Journal*, volume 40, no. 1, 2001: pp. 104–129, doi:10.1147/sj.401.0104.

[3] Perry, J. *Java Management Extensions*. Beijing Cambridge Mass: O'Reilly, 2002, ISBN 0-596-00245-9.

[4] Edpresso Team. What is a remote procedure call (RPC)? [online], [accessed 15.4.2022]. Available from: https://www.educative.io/edpresso/what-is-a-remote-procedure-call-rpc

[5] Nelson, B. J. *Remote Procedure Call*. Dissertation thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1981.

[6] Marlow, S.; et al. Haskell 2010 language report. [online], Jul 2010, [accessed 11.4.2022]. Available from: https://www.haskell.org/onlinereport/haskell2010/

[7] Tibell, J. EKG: Remote monitoring of running processes over HTTP. [online], 2017, [accessed 11.4.2022]. Available from: https://github.com/tibbe/ekg/blob/b6dcfd3fa2da837c440517c8850fedc1923bcfd7/README.md

[8] AgentM. Curryer. [online], Dec 2020, [accessed 15.4.2022]. Available from: https://github.com/agentm/curryer/blob/13ed23a6a6b63af84354d3b2a059ca0fd02c1a29/README.markdown

[9] McManus, E.; et al. Java management extensions (jmx) specification. Technical report, Sun Microsystems, 2006, [accessed 11.4.2022]. Available

from: https://docs.oracle.com/javase/8/docs/technotes/guides/jmx/JMX_1_4_specification.pdf

[10] Edmeier, J.; et al. codecentric's Spring Boot Admin. [online], Sep 2021, [accessed 11.4.2022]. Available from: https://github.com/codecentric/spring-boot-admin/blob/e58184726652aae3712a83f8d246fa4f8a729307/README.md

[11] Haughey, D. Moscow method. [online], 2011, [accessed 2.5.2022]. Available from: https://scholar.googleusercontent.com/scholar?q=cache:yCVMKQGrX04J:scholar.google.com/

[12] MDN contributors. HTTP authentication. [online], Feb 2022, [accessed 6.5.2022]. Available from: https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication

[13] Jones, M.; Hardt, D. The OAuth 2.0 Authorization Framework: Bearer Token Usage. [online], Oct. 2012, doi:10.17487/RFC6750, [accessed 6.5.2022]. Available from: https://www.rfc-editor.org/info/rfc6750

[14] Servant Contributors. servant – A type-level web DSL. [online], 2018, [accessed 9.5.2022]. Available from: https://docs.servant.dev/en/stable/

[15] MDN contributors. Same-origin policy. [online], Apr 2022, [accessed 9.5.2022]. Available from: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

Appendix A

Acronyms

API Application programming Interface

CORS Cross-Origin Resource Sharing

CPU Central Processing Unit

CSS Cascading Style Sheets

GUI Graphical User Interface

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

I/O Input/Output

JMX Java Management Extensions

JSON JavaScript Object Notation

REST Representational State Transfer

RFC Remote Function Call

RPC Remote procedure Call

TLS Transport Layer Security

UI User Interface

UML Unified Modeling Language

URL Uniform Resource Locator

WAI Web Application Interface

XML Extensible Markup Language

Appendix B

Contents of enclosed SD card

- `readme.txt` ..... file with SD card contents description
- `exe` ..... directory with executables
- `src` ..... directory of source codes of the implementation
- `thesis` ..... directory of LaTeX source codes of the thesis
- `text` ..... thesis text directory
  - `thesis.pdf` ..... thesis text in PDF format
  - `thesis.ps` ..... thesis text in PS format