DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2020

Applying Design By Contract to Remote Procedure Call Interface Definition Languages

MATAS KAIRAITIS

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Tillämpling av Design Genom Kontrakt till Gränssnitt av Avlägna Proceduranropp

MATAS KAIRAITIS

Degree Project in Computer Science, DD142X
Date: June 8, 2020
Supervisor: Alexander Kozlov
Examiner: Pawel Herman
School of Electrical Engineering and Computer Science

Abstract

Design by contract, abbreviated as DBC, is a software development methodology that aims to increase the reliability and robustness of software components. While a fair amount of research has been done around how DbC can be utilised in an in-process object-oriented system, not much is known about how DbC concepts can be applied to systems that predominantly communicate over the network by way of remote procedure calls. With the recent increase in popularity of service-oriented and microservice system architectures, the ability to develop robust networked components at scale is highly relevant. This study applies the DbC notion of software contracts to remote procedure calls by developing an interface definition language that can be used in conjunction with JSON-RPC and JSON Schema. The results demonstrate that it is possible to leverage DbC concepts when implementing networked software services, but that it may in many cases be impractical to do so due to the resulting concurrency issues and increased complexity.

Sammanfattning

Design genom kontrakt, abbrevierat ned till DbC, är en metodik för mjukvaruutveckling vars syfte är att öka pålitlighet och robusthet av programvara. Medan en ansenlig mängd har forskats för att bedöma hur DbC kan utnyttjas i ett objekt-orentierad sammanhang, det är fortsatt ovetandes om hur DbC konceptet kan appliceras till system som huvudsakligen kommunicerar över nätverket. Med den ökade populariteten av service-orienterad mjukvaruarkitektur, förmågan att utveckla robusta nätverkskomponenter är högst relevant. Denna studie applicerar en DbC förståelse av mjukvarukontrakt till gränssnittav av avlägna proceduranropp genom att använda JSON-RPC och JSON Schema. Resultatet visar att det finns möjlighet att verkställa DbC koncepter när man implementerar mjukvara som kommunicerar över nätverket, men detta kan vara opraktiskt pga ökad komplexitet och resulterande samtidighetsproblem.

Abbreviations

API Application Programming Interface

DBC Design by Contract

IDL Interface Definition Language

RPC Remote Procedure Call

Contents

Chapter 1

Introduction

Design by contract, abbreviated as DbC, is a software development methodology that aims to increase the reliability and robustness of software components by applying a systematic approach to specifying and implementing object-oriented software elements and their relations in a software system. While a fair amount of research has been done around how DbC can be utilised in an in-process object-oriented system, not much is known about how DbC concepts can be applied to systems that predominantly communicate over the network by way of remote procedure calls. With service-based and microservice architectures gaining in popularity, understanding how DbC concepts can be expressed in interface definition languages that are used to define the application programming interfaces of networked software services may have a considerable positive impact on the relative correctness and reliability of modern software systems, as well as increase the utility and productivity of software application developers.

1.1 Problem Statement

Traditional software development workflows consist of a group or multiple groups of software developers contributing software components such as functions, classes and methods to a single codebase. The separate components then interact with each other by way of in-process procedure calls. To allow for a more disciplined and resilient software development process, ideas such as object-oriented programming and DbC were developed and widely adopted, resulting in an increase in software reliability and developer productivity (Meyer 1992).

In recent years, to address the growing complexity and size of both large software systems and software developer teams, a new trend has emerged in large-scale software system design: service-oriented and microservice architectures (Bossert, Ip, and Starikova 2020). In such a system two or more software components interact with one-another through a communication protocol over a network as opposed to in-process procedure calls (Newman 2018). While some concepts of interface design have been carried over from the object-oriented paradigm, DbC concepts and ideas seem to have largely been omitted. As such, we state the research question: To what extent, and what benefit, can DbC concepts be applied to networked service interface definition languages?

1.2 Scope

This study does not aim to advocate for or against a specific in-process or networked system architecture. It simply observes current trend of splitting out complex, monolithic software systems into smaller networked components in order to determine if there exist any potential productivity gains that can be achieved by software developers and organisations when applying DbC concepts that are well established in the world of object-oriented programming to interface definition languages commonly used to define the application programming interfaces of networked software components.

While there are many ways in which software components can communicate with one another over the network in a service-oriented or microservice architecture, this study focuses solely on applying DbC concepts to synchronous, request-response remote procedure calls. Mechanisms such as queues, publish-subscribe, and streams will not be considered.

Chapter 2

Background

2.1 Design by Contract

The fundamental idea of DbC is that explicitly stating the desired behaviour of software component greatly contributes to ensuring that the desired behaviour is achieved thus attaining increased software robustness and reliability. Robust software does not require as much maintenance and can be reused more readily which is an attractive proposition for those seeking to reduce costs associated with developing software applications (Benveniste et al. 2015).

The behaviour of software components can be stated in a variety of different ways, the most simple being plain English. However, a more formal expression of this idea is that of a "contract" for a software component. Much like an agreement or contract between individuals or organisations in a business setting, the software component contract defines the obligations that each caller of the component must achieve in order to receive the desired benefits of the component (Meyer 1992). More concretely, each software component be it a function, method, or class, must specify the 'pre' and 'post' conditions that have to be met in order for the component to function correctly, as well as the outcomes that the software component promises to deliver, given that all of the preconditions are satisfied when the component is invoked. Consider the following example of a verbal software contract for retrieving an element from a key-value data structure:

Table 2.1: Example of a software component contract

Entity	**Obligations**	**Benefits**
Caller	Provide a non-void key for which the value should be retrieved. Ensure that the key is present in the key-value store.	Have a value in the store that can be accessed by using the key at a later time.
Component	Return the value for the provided key if the key has an associated value within the store. Do not remove the key-value pair from the store.	No need to return a value if the provided key does not have a corresponding value within the store.

While the example in table 2.1 serves well as a simplistic view, the real value of DbC is realised when software contracts can be defined using a formal system based on discrete mathematics thus making it possible to verify and enforce them at program runtime.

One example of where contracts are central to the entire model is the Eiffel programming language in which specifying contracts for a component is built into the language (Eiffel Software 2019). To illustrate how such contracts function, consider the API of a key-value store component that supports the *contains(key)* and *get(key)* methods that check if the given key string is present in the store, and retrieve the string value associated with the given key respectively. In the Eiffel programming language, the API containing contracts for the methods is expressed as follows:

Listing 2.2: Software contracts in the Eiffel programming language

```
contains (key: STRING): BOOLEAN
    -- Check if the store contains a value for ‘key‘
    require
        key /= Void
    do
        -- Actual implementation goes here
        ...
    end


get (key: STRING): STRING
    -- Get the value associated with
    -- ‘key‘ from the store.
    require
        key /= Void and then contains(key)
    do
        -- Actual implementation goes here
        ...
    ensure
        contains(key)
    end
```

Eiffel guarantees that the contracts of the components are checked at runtime. In the example above, the *require* clause is guaranteed to run before the execution of the *do* clause, meaning that when the implementation of the method is reached, the key is guaranteed to be a non-void string, and have an associated value within the store. After the execution of the method, the *ensure* clause is guaranteed to run which checks that the key-value pair was not removed from the store. If either the precondition or postcondition checks fail, assertion errors will be raised, thus terminating the execution of the program (Meyer 1992).

Much like the pre/post conditions for individual methods showcased in listing 2.2, DbC also enables us to specify requirements for entire classes of components. This is achieved by specifying a class invariant. A class invariant constrains objects of a class to preserving a certain condition at all times. The

class invariant must be true before a method of the class is invoked, and must be true after the method call completes. Building atop the key-value store example in listing 2.2, consider two more methods named *insert*, and *remove* that insert a key-value pair into the store, and remove a key-value pair from the store respectively. A possible invariant in a class such as this is one ensuring the number of elements is equal or greater to zero at all times.

Listing 2.3: Class invariant in the Eiffel programming language

```
class
        MAP
    ...

feature -- Members

        size: INTEGER
        -- Number of items in the key-value store

feature -- Methods

        contains (key: STRING): BOOLEAN
        -- Check if the store contains a value for 'key'
        ...

        get (key: STRING): STRING
        -- Get the value associated with 'key' from the store.
        ...

        insert (key: STRING, value: STRING)
        -- Insert the 'key'-'value' pair into the store
        ...

        remove (key: STRING): STRING
        -- Remove and return the value associated with
        -- 'key' from the store.
        ...

invariant
        size_greater_than_zero: size >= 0
```

```
end
```

The *invariant* section in the bottom guarantees that for each object of the class MAP, before and after the execution of its methods, the size will always be equal to or greater than zero. While the methods of the class can break this invariant throughout the duration of their execution, they must always restore it before returning. If the invariant is found to be invalid before or after the execution of any of the methods within the class, an assertion error will be raised.

The fundamental ideas of DbC advocate that if a precondition for a method is specified, it is the responsibility of the caller to check said precondition before invoking the method, as any runtime violation of an assertion is not treated as a special case, but the manifestation of a software bug (Meyer 1992). While not many languages choose to treat software contract validation as a first-class concept, it is possible to apply the idea in most major programming languages with the only requirement being that some form of assertions are supported.

2.2 Remote Procedure Calls

A remote procedure call, abbreviated as RPC, is a form of inter-process communication where a software component invokes a procedure call that may execute in a different process, possibly on a different machine (Birrell and Nelson 1983). This is achieved by a request-response type exchange where the caller, referred to as the client, sends the name and parameters of the operation in an agreed-upon format in a chosen protocol over the network to a destination, referred to as the server or service, that is able to execute the procedure and return the resulting output to the client. While this approach allows functionality to be moved out of process, it is a more complex interaction model compared to an in-process procedure call as it requires additional logic to be incorporated into both the client and server components to facilitate the exchange of information over the network. Popular approaches to RPC include, but are not limited to, gRPC (The Linux Foundation 2020) and JSON-RPC (JSON-RPC Working Group 2013).

2.3 Interface Definition Language

In order for two or more software services to exchange data using RPC an agreed-upon format for data interchange has to be established. An interface

definition language, or IDL for short, is a structured way of expressing the application programming interface, abbreviated as API, of an RPC service in a programming-language agnostic manner. IDLs are commonly used to communicate the structure and type information of the data that networked services exchange with one-another over RPC. Commonly used IDLs include Google Protocol Buffers (Google Developers 2020), as well as the OpenAPI Specification (Swagger 2020). It is worth mentioning that not all RPC implementations require the use of an IDL. Some approaches such as JSON-RPC only outline the protocol and leave the choice of using an IDL entirely up to the user.

In addition to providing a common understanding between the server and client of the structured data to be exchanged, IDLs also enable service producers and consumers to employ automated tools for code generation that can make development, testing, as well as payload structure validation at runtime easier by greatly reducing the amount of so-called "boilerplate" code that is needed for successful data exchange (Lämmel and Jones 2003).

2.4 JSON, JSON Schema and JSON-RPC

JavaScript Object Notation, abbreviated as JSON is a data interchange format based on a subset of the JavaScript programming language (ECMA International 2017). It is commonly used as the data serialization format for interprocess communication over RPC. An example of a JSON document partially describing this study can be seen in listing 2.4 below.

Listing 2.4: Example of a JSON Document

```
{
    "title": "Applying Design by Contract...",
    "sections": [
        {
            "name": "Introduction"
        }
    ],
    "sectionCount": 1
}
```

JSON Schema is a vocabulary that makes it possible to annotate and validate JSON documents (Wright, Henry, and Hutton 2019). JSON Schema definitions are expressed in JSON. They include structure and type information, as well as constraints that a JSON document must conform to. A minimal example of a JSON Schema document that validates the JSON structure in listing 2.4 is shown in listing 2.5 below.

Listing 2.5: Example of a JSON Schema Document

```
{
    "additionalProperties": false,
    "required": [
        "title",
        "sections",
        "sectionCount"
    ],
    "properties": {
        "title": {
            "type": "string"
        },
        "sections": {
            "type": "array",
            "items": {
                "type": "object",
                "additionalProperties": false,
                "required": [
                    "name"
                ],
                "properties": {
                    "name": {
                        "type": "string"
                    }
                }
            }
        },
        "sectionCount": {
            "type": "integer"
        }
    }
}
```

JSON-RPC is a lightweight RPC protocol (JSON-RPC Working Group 2013). The JSON-RPC 2.0 standard defines two objects: the *request* and the *response*. The request object must define the required *jsonrpc*, *method* and *id* members containing the version of JSON-RPC, the name of the operation to invoke, and the unique identifier of the object respectively. In addition, it may also define an optional *params* member containing the parameters of the operation. Similarly, the response object must define the required *jsonrpc* and *id* members that maintain the same semantics as the request object. In addition, the response object must define one, but not both, of the following members: *result* containing the *result* if the RPC operation executed successfully, or *error* if any error was triggered when processing the request. An example of a successful request/response payload can be found in listing 2.6 below.

Listing 2.6: Example JSON-RPC request/response objects

```
{
    "jsonrpc": "2.0",
    "id": 12345,
    "method": "get",
    "params": {...}
}
```

```
{
    "jsonrpc": "2.0",
    "id": 12345,
    "result": ...
}
```

2.5 Previous work

Extensive research has been done on topics such as DbC, RPC as well as the various IDLs and data serialization formats that can be used to interchange data over the network. However, there has not been any formal investigation into how the DbC notion of contracts can be communicated and/or enforced using RPC. When beginning to ponder such an approach new questions such as:

- How should software contracts be expressed and serialized?
- Should the client, the server, or both, validate the software contracts?

arise. To the best of the author's knowledge, possible answers to such questions have not been formally investigated and require further inquiry that will be carried out by this study.

Chapter 3

Method

In order to illustrate how DbC concepts can be applied to RPC, an interface definition for a service exposing a key-value store was developed as a practical example. This section outlines the steps and tools that were used to do so. In order to re-use existing technologies as opposed to introducing a new RPC protocol, JSON-RPC and JSON Schema were leveraged as the basis of the IDL as they are well established and simple to understand. All of the schema validations were done using the Python programming language as it is widely adopted and has readily available built-in or open-source tooling for working with both JSON and JSON-Schema (Python Software Foundation 2020).

3.1 Defining a JSON Schema for JSON-RPC

A prerequisite for a successful RPC interaction is an agreement between server and client on the format and structure of data to be exchanged. The JSON-RPC protocol and JSON Schema were used as the definition of the rules for the exchange and the data definition language respectively. The two technologies combined constituted an interface definition of a RPC service.

As a first step, a formal schema for JSON-RPC 2.0 was defined using JSON Schema. This provided the capability to programatically validate that JSON documents conform to the JSON-RPC 2.0 specification. To ensure that the JSON Schema was defined correctly, it was tested against a series of valid and invalid JSON documents as defined by the JSON-RPC specification. A Python example of how to validate a JSON document against a JSON Schema using the open-source *jsonschema* package is shown in listing 3.1 (Berman 2020).

Listing 3.1: Validating a JSON document using the jsonschema Python package

```
from jsonschema import validate

json_document = ...  # Some JSON document
json_schema = ...  # Schema to validate the JSON document agaist

# Errors will be raised if validation fails
validate(instance=json_document, schema=json_schema)
```

Next, the JSON Schema for JSON RPC was extended to contain the domain specific request and response object definitions for the following methods in the key-value store interface:

- *size()*: Retrieves the number of elements in the store
- *contains(key)*: Checks if the store contains a value with the given key
- *get(key)*: Retrieves the value with the given key from the store
- *insert(key, value)*: Inserts the key-value pair into the store
- *remove(key)*: Removes the value with the given key from the store

The enhanced JSON Schema was once again tested against a series of valid and invalid JSON documents, this time containing payloads specific to the domain.

3.2 Extending JSON-RPC to JSON-RPC-DbC

The next step was to extend the interface definition to support DbC concepts. To accomplish this, two groups of pre/post conditions, or constraints, were considered. The first group consisted of simple parameter validations such as value ranges, enumerations and patterns. Our choice of IDL, JSON Schema, provided built-in support for such constructs. For example, defining a string that is guaranteed to be within a certain length range could be accomplished as follows:

Listing 3.2: Example of JSON Schema Constraints

```
{
  "type": "string",
  "minLength": 5,
  "maxLength": 10
}
```

The schema constraints were added to the domain specific payload validation schema from section 3.1 where necessary.

The second group consisted of constraints that may require calling other methods that the interface exposes. Considering the key-value store example in section 2.1, in order to be able to get a value for a given key, we must first ensure that that key-value pair exists in the store. Therefore, the interface must in some way express the fact that *contains(key)* must be true before the invocation of *get(key)* can succeed. To accomplish this, a way of specifying pre/post conditions and invariants was defined as a JSON structure in a separate file, containing the pre/post conditions for each method call within the interface as well as the invariants of the system. In order to have an easy way of referring to a JSON-RPC interface that supports DbC using a separate file, the term JSON-RPC-DbC was introduced. The file containing the component contract information was dubbed as the "RPC-DbC file" for brevity. A JSON Schema defining the structure of the RPC-DbC file was also created so that its contents could be validated programatically.

Chapter 4

Results

The resulting interface definition for the key-value store API consists of two JSON files: the schema for the payloads of the RPC interaction and the RPC-DbC file containing the software contract information of the key-value store API. Together, the two files enable both the client and server to programatically ensure than an RPC call is valid based on the constraints specified by the interface, and the current system state. Both files contain information for the following RPC methods:

- *size()*: Retrieves the number of elements in the store
- *contains(key)*: Checks if the store contains a value with the given key
- *get(key)*: Retrieves the value with the given key from the store
- *insert(key, value)*: Inserts the key-value pair into the store
- *remove(key)*: Removes the value with the given key from the store

The first part of the interface is the payload schema that defines the structure of the domain specific JSON RPC objects to be sent between server and client. The schema can be used to validate the structure of the JSON payloads that are sent between server and client. The validation can be performed by either the server, the client, or both using existing tools such as the *jsonschema* package for Python as shown in listing 3.1. As the payload schema is simply an application of JSON Schema, it is not described in detail and can be found in full in appendix A.

The second part of the interface is the RPC-DbC file containing the pre/post conditions for the methods in the interface as well as the system invariants. The core concept at the heart of the RPC-DbC file is that of a constraint object. The constraint object is used to indicate that a method that is exposed on the API should be called as a part of a system invariant check or a pre/post condition of a different method. An extract from the RPC-DbC file of such an object is shown in listing 4.1 below.

Listing 4.1: RPC-DbC file constraint object extract

```
{
   "call": "contains",
   "paramMappings": [
      {
         "currentParam": "key",
         "targetParam": "key"
      }
   ],
   "expect":{
      "eq": true
   }
}
```

The constraint object above communicates that a method named *contains* must be called with the currently bound parameter of *key* mapped to the target method's *key* parameter with the expected return value being equal to *true*. In this case, the expected return value is a boolean, however, it can also be any other primitive JSON value or even another constraint object. The *expect* object uses the *eq* key to indicate what the return value must be equal to, however, *gt*, *lt*, and *neq* can also be used to ensure that the returned value is greater than, less that, or not equal respectively.

The root of the RPC-DbC file contains two fields: *invariants* and *constraints*. The *invariants* field contains a list of constraint objects that the system should uphold as DbC invariants, as described in section 2.1. The *constraints* section contains objects describing the pre/post conditions for the methods exposed by the API. Such objects consist of the *methodName* field containing the name of the method on which the condition should be placed, as well as the *require* and *ensure* fields containing lists of constraint objects for the pre and post conditions respectively. A trimmed down extract from the RPC-DbC file outlining

the structure is shown in listing 4.2 below.

Listing 4.2: RPC-DbC file illustrative extract

```
{
  ...
  "invariants": [
    ... constraint objects
  ],
  "constraints": [
    {
      "methodName": "contains",
      "require": [
        ... constraint objects
      ],
      "ensure": [
        ... constraint objects
      ]
    }
  ],
  ...
}
```

The full RPC-DbC file for the key-value store example can be found in appendix B. As this is a new file specification, a JSON Schema defining the structure of the RPC-DbC file is also included, and can be found in appendix C.

To leverage the full benefits of this approach the client and server must carry a copy of both the JSON Schema for the payload structure as well as the RPC-DbC file. Validating the payload structure schema can be done by using existing JSON Schema tooling. Validating the constraints in the RPC-DbC file requires custom tooling to be built that understands the structure and contents of the RPC-DbC file.

Chapter 5

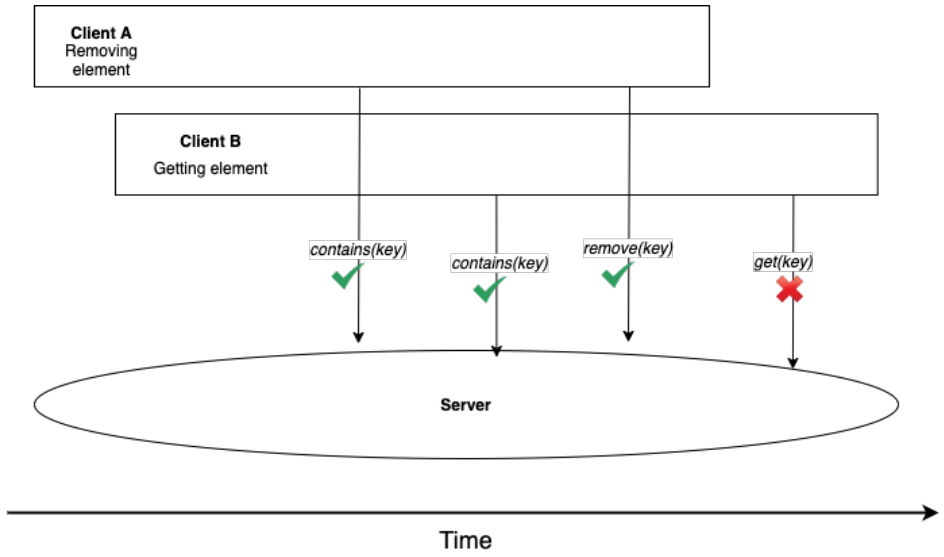
Discussion

5.1 Results

The results show that it is possible to apply the DbC idea of software component contracts to remote procedure calls. Pre/post conditions as well as class invariants can be expressed in the RPC-DbC file and used to ensure that the contracts are upheld by both the client and server. Following the mantra of DbC where "stating the desired behaviour of a software component contributes greatly to ensuring that the behaviour is achieved", the ability to apply DbC to RPC is a useful tool in any API design process and can go a long way to increasing software robustness.

One major limitation are the concurrency issues that arise when using this approach. The rules of DbC dictate that the client is responsible for ensuring that all relevant preconditions are met prior to performing an operation. While this works well in a single-threaded in-process model, nontrivial issues present themselves when multiple clients invoke a series of non-atomic operations on the server. As an example, in the key-value store we must check that *contains(key)* is true before invoking *get(key)*. Each of these operations requires the client to invoke a separate RPC call over the network. Consider what would happen if two clients, A and B, were to perform operations on the key-value store simultaneously on a single server, as shown in figure 5.1 below.

Figure 5.1: RPC-DbC concurrency issues



Client A attempts to remove a key-value pair from the store. Client B attempts to retrieve the same key-value pair from the store. To ensure that the precondition is satisfied, client A first invokes *contains(key)* to ensure that the pair exists. Subsequently, client B does the same. Both calls return *true*, and both clients proceed with the next operation. Client A invokes *remove(key)* and succeeds, however when client B invokes *get(key)* the operation fails with an assertion error, even though client B has performed all of the necessary steps to ensure that the precondition is valid. While such issues can be circumvented using some form of distributed transactions, this adds additional complexity both from the system design, and implementation standpoint (Thomson et al. 2012). Further research is needed to see how such concurrency issues can be circumvented.

An additional limitation is that more RPC calls are needed within this paradigm. Instead of performing one call for a single operation, clients may have to invoke multiple RPC methods to ensure that DbC preconditions are satisfied. Communication over the network is considerably slower than in-memory procedure calls, meaning that checking preconditions and postconditions would incur a hefty performance penalty when compared to the server handling all of the error cases, such as in the defensive programming paradigm.

5.2 Method

While there are many RPC protocols and serialization formats available, JSON and JSON Schema were used in this study to illustrate how DbC concepts can be applied to RPC operations. JSON and JSON Schema were chosen as the format is easy to read, widely understood, and has readily available tooling in most major programming languages. It is worth mentioning that the choice of serialization format and IDL does not restrict the ability to utilise DbC concepts alongside RPC operations. While not explicitly explored in this study, using a similar method it should be possible to apply DbC concepts can be applied to other data formats, such as XML, or even binary formats.

A separate file was used to communicate DbC contract information. A more simplistic approach might be to simply include that information in the payload of the request/response object of the RPC operation, however, that would mean that each time data is transmitted over the wire static, redundant contract data would be transmitted alongside it. Defining the pre/post conditions in a separate file allows for smaller payload sizes and allows either the client or server components to choose when they want to refer them. An additional advantage of using a separate file as opposed to sending contract information in the payload is that it allows for complete backwards compatibility with JSON-RPC. If a client or server component chooses to provide contract information, but not to enforce it, it will not impact the way JSON-RPC calls function.

One limitation of the chosen method is that there are multiple files that need to be distributed to clients of the service. If a client component wishes to invoke RPC procedures on a server component using JSON-RPC-DbC they must have access to both the service interface definition file, containing the JSON schema for the request/response payload structures, as well as the constraints file containing the pre/post conditions and invariants of the interface. Having two files that need to be distributed implies additional risk as the files may fall out of sync and become incompatible with one another if the distribution process is not coordinated carefully.

Chapter 6

Conclusion

This study set out to investigate if design by contract concepts could be applied to interface definition languages that are used to specify the application programming interfaces of software components that communicate using remote procedure calls. It has demonstrated that it is indeed possible to do so by enhancing existing RPC protocols, however, it has also shown that certain concurrency and performance issues that arise when attempting to have the client component check the constraints before executing an RPC operation. While these issues can be circumvented to an extent, the workarounds often add additional complexity. To conclude, while entertaining the DbC paradigm when designing RPC interfaces is a useful exercise, it might in many cases not be pragmatic to expose it within the API of an RPC service due to the added complexity and concurrency issues that require further research.

Bibliography

Benveniste, A. et al. (2015). "Contracts for Systems Design: Theory". In: *Inria Rennes Bretagne Atlantique*, p. 86.

Berman, Julian (Sept. 2020). *jsonschema*. [Online; Accessed 25-Mar-2020]. URL: https://pypi.org/project/jsonschema/.

Birrell, D. A. and B. Nelson (1983). "Implementing Remote procedure calls". In: *ACM SIGOPS Operating Systems Review* 17.5, pp. 39–59.

Bossert, O., C. Ip, and I. Starikova (Sept. 2020). *Beyond agile: Reorganizing IT for faster software delivery*. Ed. by mckinsey.com. [Online; Accessed 25-Mar-2020]. URL: https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/beyond-agile-reorganizing-it-for-faster-software-delivery.

ECMA International (Dec. 2017). *The JSON Data Interchange Syntax*. [Online; Accessed 25-Mar-2020]. URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

Eiffel Software (2019). *Building bug-free O-O software: An Introduction to Design by Contract*. Ed. by Eiffel Software. [Online; Accessed 25-Mar-2020]. URL: https://www.eiffel.com/values/design-by-contract/introduction/.

Google Developers (2020). *Protocol Buffers*. [Online; Accessed 25-Mar-2020]. URL: https://developers.google.com/protocol-buffers.

JSON-RPC Working Group (2013). *JSON-RPC*. [Online; Accessed 25-Mar-2020]. URL: https://www.jsonrpc.org/specification.

Lämmel, R and S. P. Jones (2003). "Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming". In: *ACM SIGPLAN Notices* 38.3.

Meyer, B. (1992). “Applying ’design by contract’”. In: *Computer* 25.10, pp. 40–51.

Newman, S (2018). *Building Microservices*. 1st edition. O’REILLY MEDIA, INC.

Python Software Foundation (2020). *Python*. [Online; Accessed 25-Mar-2020]. URL: https://www.python.org/.

Swagger (2020). *OpenAPI Specification*. [Online; Accessed 25-Mar-2020]. URL: https://swagger.io/specification/.

The Linux Foundation (2020). *gRPC*. [Online; Accessed 25-Mar-2020]. URL: https://grpc.io/.

Thomson, A. et al. (May 2012). “Calvin: Fast Distributed Transactions for Partitioned Database Systems”. In: *SIGMOD ’12*, pp. 1–12.

Wright, A., A. Henry, and B. Hutton (Sept. 2019). *JSON Schema: A Media Type for Describing JSON Documents*. [Online; Accessed 25-Mar-2020]. URL: https://json-schema.org/draft/2019-09/json-schema-core.html.

Appendix A

JSON Schema for key-value Store Payloads

```
{
  "definitions": {
    "jsonrpc": {
      "type": "string",
      "enum": [
        "2.0"
      ]
    },
    "error": {
      "type": "object",
      "properties": {
        "code": {
          "type": "integer"
        },
        "message": {
          "type": "string"
        },
        "data": {}
      },
      "required": [
        "code",
        "message"
      ],
      "additionalProperties": false
```

```
    },
    "Request": {
      "type": "object",
      "properties": {
        "jsonrpc": {
          "$ref": "#/definitions/jsonrpc"
        },
        "id": {
          "type": [
            "string",
            "integer",
            "null"
          ]
        }
      },
      "required": [
        "jsonrpc"
      ]
    },
    "Response": {
      "type": "object",
      "properties": {
        "jsonrpc": {
          "$ref": "#/definitions/jsonrpc"
        },
        "id": {
          "type": [
            "string",
            "integer",
            "null"
          ]
        },
        "result": {},
        "error": {
          "$ref": "#/definitions/error"
        }
      },
      "required": [
        "id",
```

```
    "jsonrpc"
  ],
  "oneOf": [
    {
      "required": [
        "result"
      ]
    },
    {
      "required": [
        "error"
      ]
    }
  ]
},
"ContainsRequest": {
  "allOf": [
    {
      "$ref": "#/definitions/Request"
    },
    {
      "properties": {
        "method": {
          "type": "string",
          "enum": [
            "contains"
          ]
        },
        "params": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
            "key": {
              "type": "string"
            }
          },
          "required": [
            "key"
          ]
```

```
          }
        },
        "required": [
          "method",
          "params"
        ]
      }
    ]
  },
  "GetRequest": {
    "allOf": [
      {
        "$ref": "#/definitions/Request"
      },
      {
        "properties": {
          "method": {
            "type": "string",
            "enum": [
              "get"
            ]
          },
          "params": {
            "type": "object",
            "additionalProperties": false,
            "properties": {
              "key": {
                "type": "string"
              }
            },
            "required": [
              "key"
            ]
          }
        },
        "required": [
          "method",
          "params"
        ]
```

```
    }
  ]
},
"InsertRequest": {
  "allOf": [
    {
      "$ref": "#/definitions/Request"
    },
    {
      "properties": {
        "method": {
          "type": "string",
          "enum": [
            "insert"
          ]
        },
        "params": {
          "type": "object",
          "additionalProperties": false,
          "properties": {
            "key": {
              "type": "string"
            },
            "value": {
              "type": "string"
            }
          },
          "required": [
            "key",
            "value"
          ]
        }
      },
      "required": [
        "method",
        "params"
      ]
    }
  ]
```

```
    },
    "RemoveRequest": {
      "allOf": [
        {
          "$ref": "#/definitions/Request"
        },
        {
          "properties": {
            "method": {
              "type": "string",
              "enum": [
                "remove"
              ]
            },
            "params": {
              "type": "object",
              "additionalProperties": false,
              "properties": {
                "key": {
                  "type": "string"
                }
              },
              "required": [
                "key"
              ]
            }
          },
          "required": [
            "method",
            "params"
          ]
        }
      ]
    },
    "request": {
      "oneOf": [
        {
          "$ref": "#/definitions/GetRequest"
        },
```

```
        {
          "$ref": "#/definitions/ContainsRequest"
        },
        {
          "$ref": "#/definitions/InsertRequest"
        },
        {
          "$ref": "#/definitions/RemoveRequest"
        }
      ]
    },
    "response": {
      "$ref": "#/definitions/GetRequest"
    }
  },
  "oneOf": [
    {
      "$ref": "#/definitions/request"
    },
    {
      "$ref": "#/definitions/response"
    }
  ]
}
```

Appendix B

Key-value Store RPC-DbC File Contents

```
{
  "invariants": [
    {
      "call": "size",
      "paramMappings": [],
      "expect": {
        "gt": −1
      }
    }
  ],
  "constraints": [
    {
      "methodName": "contains",
      "require": [],
      "ensure": []
    },
    {
      "methodName": "size",
      "require": [],
      "ensure": []
    },
    {
      "methodName": "get",
      "require": [
```

```
    {
      "call": "contains",
      "paramMappings": [
        {
          "currentParam": "key",
          "targetParam": "key"
        }
      ],
      "expect": {
        "eq": true
      }
    }
  ],
  "ensure": [
    {
      "call": "contains",
      "paramMappings": [
        {
          "currentParam": "key",
          "targetParam": "key"
        }
      ],
      "expect": {
        "eq": true
      }
    }
  ]
},
{
  "methodName": "insert",
  "require": [
    {
      "call": "contains",
      "paramMappings": [
        {
          "currentParam": "key",
          "targetParam": "key"
        }
      ],
```

```
      "expect": {
        "eq": false
      }
    }
  ],
  "ensure": [
    {
      "call": "contains",
      "paramMappings": [
        {
          "currentParam": "key",
          "targetParam": "key"
        }
      ],
      "expect": {
        "eq": true
      }
    }
  ]
},
{
  "methodName": "remove",
  "require": [
    {
      "call": "contains",
      "paramMappings": [
        {
          "currentParam": "key",
          "targetParam": "key"
        }
      ],
      "expect": {
        "eq": true
      }
    }
  ],
  "ensure": [
    {
      "call": "contains",
```

```
          "paramMappings": [
            {
              "currentParam": "key",
              "targetParam": "key"
            }
          ],
          "expect": {
            "eq": false
          }
        }
      ]
    }
  ]
}
```

Appendix C

RPC-DbC File JSON Schema

```
{
  "definitions": {
    "Constraint": {
      "type": "object",
      "required": [
        "call",
        "paramMappings",
        "expect"
      ],
      "additionalProperties": false,
      "properties": {
        "call": {
          "type": "string"
        },
        "paramMappings": {
          "type": "array",
          "items": {
            "type": "object",
            "required": [
              "currentParam",
              "targetParam"
            ],
            "additionalProperties": false,
            "properties": {
              "currentParam": {
                "type": "string"
```

```
      },
      "targetParam": {
        "type": "string"
      }
    }
  }
},
"expect": {
  "type": "object",
  "required": [],
  "additionalProperties": false,
  "properties": {
    "gt": {
      "oneOf": [
        {
          "type": "number"
        },
        {
          "type": "string"
        },
        {
          "type": "boolean"
        },
        {
          "$ref": "#/definitions/Constraint"
        }
      ]
    },
    "lt": {
      "oneOf": [
        {
          "type": "number"
        },
        {
          "type": "string"
        },
        {
          "type": "boolean"
        },
```

```
            {
              "$ref": "#/definitions/Constraint"
            }
          ]
        },
        "eq": {
          "oneOf": [
            {
              "type": "number"
            },
            {
              "type": "string"
            },
            {
              "type": "boolean"
            },
            {
              "$ref": "#/definitions/Constraint"
            }
          ]
        },
        "neq": {
          "oneOf": [
            {
              "type": "number"
            },
            {
              "type": "string"
            },
            {
              "type": "boolean"
            },
            {
              "$ref": "#/definitions/Constraint"
            }
          ]
        }
      }
    }
```

```
    }
  }
},
"type": "object",
"required": [
  "invariants",
  "constraints"
],
"additionalProperties": false,
"properties": {
  "invariants": {
    "type": "array",
    "items": {
      "$ref": "#/definitions/Constraint"
    }
  },
  "constraints": {
    "type": "array",
    "items": {
      "type": "object",
      "required": [
        "methodName",
        "require",
        "ensure"
      ],
      "additionalProperties": false,
      "properties": {
        "methodName": {
          "type": "string"
        },
        "require": {
          "type": "array",
          "items": {
            "$ref": "#/definitions/Constraint"
          }
        },
        "ensure": {
          "type": "array",
          "items": {
```

```
              "$ref": "#/definitions/Constraint"
            }
          }
        }
      }
    }
  }
}
```

TRITA -EECS-EX-2020:343