HOCHSCHULE
RAVENSBURG-WEINGARTEN
UNIVERSITY
OF APPLIED SCIENCES

Das gRPC-Framework zur Nutzung in Web APIs

Bachelorarbeit

Verfasser: Matthias Funk
31353

Gutachter: Prof. Dr. rer. nat. Marius Hofmeister
RWU Hochschule Ravensburg-Weingarten
Prof. Dr. rer. nat. Martin Zeller
RWU Hochschule Ravensburg-Weingarten

06. April 2023

Abstract

In dieser Arbeit wurde die Verwendung des gRPC-Frameworks in Web APIs untersucht. Dazu wurde gRPC mit den etablierten API Kommunikationskonzepten REST und WebSocket vergleicht. Als Vergleichskriterien dienten ein Großteil der Merkmale aus der Norm ISO 25010, die ein Modell zur Bewertung von Qualitätskriterien für Software beschreibt. Es werden die Vor- und Nachteile von gRPC, REST und WebSocket hinsichtlich der Kategorien Funktionalität, Kompatibilität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Wartbarkeit und Sicherheit aufgezeigt.

Die Ergebnisse zeigen, dass gRPC insbesondere in den Kriterien Korrektheit, Zuverlässigkeit und Sicherheit besser performt. Allerdings erfordert die Verwendung von gRPC auch ein höheres technisches Verständnis und eine aufwändigere Implementierung als REST oder WebSocket. Insgesamt ist gRPC eine vielversprechende Option für die Entwicklung von Web-APIs.

Inhaltsverzeichnis

Selbständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit eigenständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen, Darstellungen und Hilfsmittel benutzt zu haben. Dies trifft insbesondere auch auf Quellen aus dem Internet zu. Alle Textstellen, die wortwörtlich oder sinngemäß anderen Werken oder sonstigen Quellen entnommen sind, habe ich in jedem einzelnen Fall unter genauer Angabe der jeweiligen Quelle gekennzeichnet. Mir ist bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note 'nicht ausreichend'(5.0) zur Folge hat und dass Verletzungen des Urheberrechts strafrechtlich verfolgt werden können.

Ort, Datum Unterschrift

1 Einleitung

1.1 Einführung

Web APIs sind eine hervorragende Möglichkeit effizient Daten über das Internet auszutauschen. Ein Beispiel für den Nutzen von APIs ist die Single Sign-On Keycloak Authentifizierung, mit der kein Konto auf der Website erstellt werden muss und der Login beispielsweise über die Google Kontodaten ermöglicht wird. Bei der Entwicklung einer API gibt es eine vielfältige Auswahl bei der sich die meisten Entwickler dennoch für eine RESTful API entscheiden. Das Verzeichnis für APIs „ProgrammableWeb“ hat in einer Statistik [1] im Jahr 2017 angegeben, dass 81,5% der hinterlegten APIs REST-basiert sind.
Eine neue Möglichkeit APIs zu implementieren ist das von Google entwickelte gRPC Framework, welches von der Cloud Native Computing Foundation als in der Entwicklungsphase eingestuft wird. Eine Eigenschaft von Projekten in dieser Phase sind wenige Implementierungsbeispiele und die fehlende Hilfestellung aufgrund der kleinen Entwicklergemeinschaft.
Durch das Verwenden von HTTP/2 unterstützt gRPC Multiplexing und binär codierte Übertragung mittels Protocol Buffers. Außerdem sind die Remote Procedure Calls auch asynchron möglich. Verbindungen zum Server können vergleichbar wie bei REST nach dem Request-Response-Modell ablaufen, aber auch bidirektionales oder einseitiges Client-, Server-Streaming ist möglich.
Diese Möglichkeiten in der Implementierung heben gRPC von den herkömmlichen API-Technologien wie WebSockets, REST, GraphQL und SOAP ab.

1.2 Motivation

Ziel der Arbeit ist es, auf dem gRPC-Framework basierende APIs mit herkömmlichen API-Stilen zu vergleichen. Da RPC nicht nur einfache Anfragen vergleichbar mit dem Request-Response-Modell unterstützt, sondern auch Streaming unterstützt, wurde für beide Möglichkeiten jeweils ein API-Stil gewählt. Der Vergleich soll eine Übersicht bieten, wann eine gRPC-API vorteilhafter gegenüber den herkömmlichen API-Stilen ist. Außerdem soll der Vergleich einen Einblick in Best Practises bieten und aufklären, wie eine gut durchdachte Architektur einen positiven Einfluss auf die Entwicklung hat.

1.3 Stand der Forschung

Der derzeitige Stand der Forschung bietet verschiedenste Tests und Vergleiche zwischen den populärsten Web APIs.
Zu gRPC gibt es von [2] einen Vergleich von Antwortzeiten auf Anfragen mit REST. Der Vergleich ergibt, dass für Anfragen mit geringen Daten, wie einzelne Integer oder Strings, REST durchschnittlich 3 ms und gRPC 7 ms benötigt. Sobald jedoch größere Dateien ausgetauscht werden, liegt die Antwortzeit von gRPC unter der von REST. Wird die Verbindung mittels TLS verschlüsselt, ist gRPC in allen getesteten Szenarien ein wenig schneller als REST.
In [3] wurden die Performance, Evolvierbarkeit und Komplexität von Introspected REST mit gRPC und GraphQL vergleicht. Im Gegensatz zu [2] hat der Vergleich von Antwortzeiten auf Anfragen ergeben, dass gRPC bei einer gesetzten Bandbreite von 100KB Up- und Download für eine Client-Anfrage 35ms und REST 88ms benötigte. Dieser Unterschied ist voraussichtlich auf die Größe der angefragten Informationen zu führen. Einer der Hauptkritikpunkte von gRPC sind die Client-Stubs, welche nach nicht abwärtskompatiblen Änderungen der Schnittstelle neu generiert werden müssen. Positiv wurde die Effizienz von Protocol Buffers gelobt und die Fehlerbehandlung bei ungültigen Eingaben.
Die Arbeiten behandeln ausschließlich Anfragen im Request-Response-Modell und somit bleibt der Vergleich für Streaming aus.

2 Grundlagen

2.1 Das gRPC Framework

In den folgenden Unterkapiteln wird die Funktionsweise des RPC Frameworks gRPC beschrieben. Um gRPC zu beschreiben, ist es wichtig, zuerst die Funktionsweise von Remote Procedure Calls zu veranschaulichen. Protocol Buffer entsprechen der Schnittstellendefinition und das HTTP/2 Protokoll dem Transportprotokoll gRPCs. Zuletzt befindet sich ein Unterkapitel mit einer Beispielanwendung.

2.1.1 Remote Procedure Call

Der Grundgedanke von RPCs nach [4] ist, eine Prozedur eines anderen Systems aufzurufen, als wäre sie lokal vorhanden. Ein Vorteil ist die Modularisierung von Prozeduren, in dem Systeme diese nicht selbst implementieren müssen, sondern über einen RPC ein dafür dediziertes System anfragen können. Die Auslagern von Prozeduren verbessert die Skalierbarkeit und ermöglicht das Umsetzen eines Client-Server-Modells.

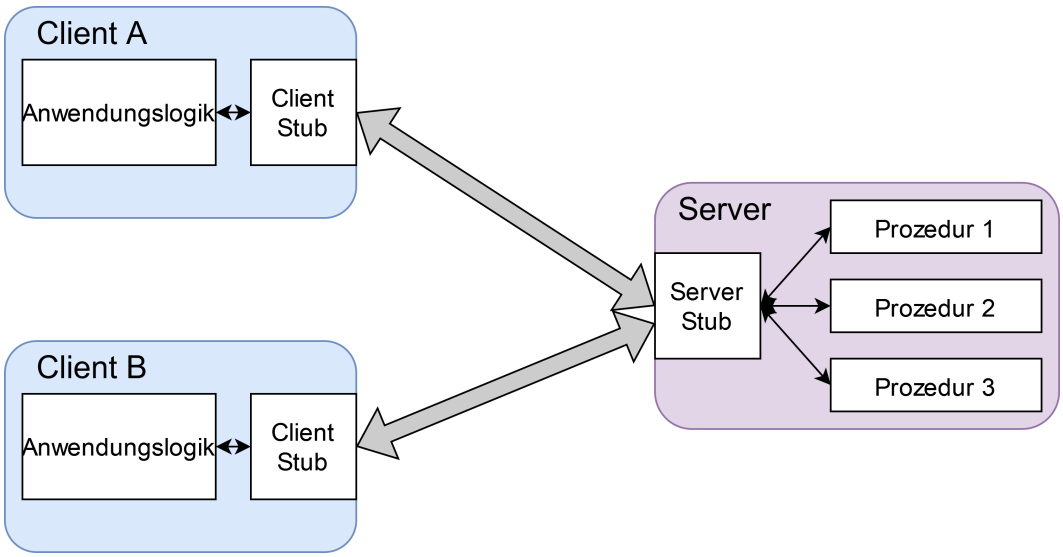

Abbildung 1: Ein Konzept nach [5] über RPCs das Client-Server-Modell umzusetzen.

Um einen Client als Aufrufender und einen Server als Ausführender der Prozedur zu ermöglichen, wird jeweils ein Stub benötigt. Die Inhalte eines Stubs sind unter anderem die Serialisierung der Parameter und die Kommunikation auf Netzwerkebene [6]. Möchte ein Nutzer nun eine Server-Prozedur aufrufen, implementiert er den dafür benötigten Stub. Die aufzurufende Prozedur wird in einer Anfrage dem Client Stub übergeben, welcher die Daten an den Server Stub sendet. Der Server Stub liest die Daten aus und ruft die entsprechende Prozedur vom Server auf. Der Client-Thread ist während des RPCs blockiert.
Da es keinen einheitlichen Standard zum RPC-Konzept gibt, können sich die unterschiedlichen RPC Protokolle und Frameworks stark voneinander unterscheiden.

2.1.2 Protocol Buffer

Die von Google entwickelte Schnittstellendefinitionssprache Protocol Buffer, kurz Protobuf genannt, bietet eine einfache, Möglichkeit Daten in Binärdaten umzuwandeln. Zusätzlich bietet Protobuf die Funktion gRPC Schnittstellen zu definieren, um daraus sprach- oder plattformspezifische Stubs für Client und Server zu generieren [7]. Die Stärken Protobufs sind die hohe Performance durch das Serialisieren der Daten in das Binärformat und die unkomplizierte Definition der Nachrichtenformate.
Definiert wird das Nachrichtenformat sowie die RPC Service Interfaces in einer „.proto“ Datei. Die Inhalte der Protobuf Nachrichten bilden sich aus Feldern, welche Datentypen, Strukturen oder weiteren Protobuf Messages entsprechen. Um Nachrichten in Binärdaten zu Serialisieren oder empfangene Binärdaten zu Deserialisieren, muss jedes Feld mit einer eindeutig zuordenbaren „Field Number“ versehen werden. Ein RPC Service Interface lässt sich mit dem Schlüsselwort `service` und einer Protobuf Anfrage- und Antwort-Nachricht definieren. Wird eine Service-Interface-Nachricht durch das Schlüsselwort `stream` ergänzt, handelt es sich um ein oder mehrere Nachrichten des Typs.

Eine .proto-Datei mit gRPC Schnittstelle, die Nachrichten vom Typ `EchoRequest` entgegennimmt und diese mit einer `EchoReply` beantwortet wird wie folgt definiert:

```
1  syntax = "proto3";
2  package echo;
3
4  service Echo {
5      rpc SendEcho (EchoRequest) returns (EchoReply) {}
6  }
7
8  message EchoRequest {
9      int64 number = 1;
10 }
11
12 message EchoReply {
13     int64 message = 1;
14 }
```

Sind alle Service-Interfaces und Nachrichten definiert, lassen sich mit dem Protokoll-Compiler protoc jeweils der Stub und der Code zum Serialisieren von Proto-Nachrichten generieren. Die neuste Version Protobufs, proto3, kann Nachrichten in den Programmiersprachen Java, Kotlin, Python, C++, Go, Ruby, Objective-C, und C# in Binärdaten serialisieren. Der Server importiert den resultierenden Stub, um ein Objekt vom Typ gRPC-Server zu erstellen und dadurch auf eingehende Anfragen zu reagieren. Clients importieren den Stub, um Protobuf-Nachrichten an den Server zu senden. Um Protobuf-Nachrichten zu erstellen und diese Nachrichten auslesen zu können, importieren beide zusätzlich die generierte Nachrichtendefinition.

2.1.3 Das HTTP/2 Protokoll

Nachdem die Protobuf Nachrichten in Binärdaten serialisiert wurden, senden die Stubs über das HTTP/2 Protokoll an den Server Stub. Das im RFC 7540 definierte HTTP/2 Protokoll baut auf dem HTTP/1.x Protokoll auf und löst dabei drei Punkte des Konzepts, um eine bessere Performance zu erreichen [8]:

- Für jede Verbindung zwischen Client und Server muss ein TCP Three-Way-Handshake durchgeführt werden. Bei Webseiten, die aus mehreren Javascript-Skripten, CSS-Stylesheets und Bildern bestehen, wird für jede Ressource ihre eigene TCP Verbindung aufgebaut.
- Eine weitere Schwäche TCPs ist das Head Of Line Blocking. Das Head Of Line Blocking kann auftreten, falls ein Client mehrere Ressourcen über dieselbe TCP-Verbindung anfragt. Bricht die Übertragung einer Ressource ab, ist die Verbindung blockiert, bis die fehlgeschlagene Ressource erneut übertragen wird. Dies passiert, weil TCP angefragte Daten nur in der Reihenfolge senden darf, in der sie angefragt wurden.
- HTTP Header sind unkomprimiert. Um eine Webseite in Desktop-Ansicht zu laden benötigte man am 01. Oktober 2022 durschnittlich 71 Anfragen [9], wovon zu jeder Anfrage ein Header gehört.

Eine Lösung HTTP/2s ist das Trennen der Header von den Nachrichten und der Umstieg auf binäre Nachrichten-Frames. Die Header Compression, kurz HPACK genannt, trennt die Header von den Paketen, in dem Client und Server jeweils eine Header-Liste führen und somit ein sendender nur noch auf die Indizes der eingetragenen Header Felder referenziert, um einen Header zu rekonstruieren [10]. Des Weiteren unterstützt HTTP/2 Multiplexing, welches dem beidseitigen Senden von Client und Server über dieselbe TCP-Verbindung entspricht. Um Head Of Line Blocking zu umgehen, können Anfragen in beliebiger Reihenfolge beantwortet werden.

2.1.4 Server- und Client-Beispielanwendung

Wurde der Protobuf Compiler protoc beispielsweise über den Python Packetmanager Pip installiert, können Stubs mit folgendem Befehl generiert werden:
`python -m grpc_tools.protoc -I./protos protos/echo_example.proto --python_out=. --grpc_python_out=.`
Der erste Parameter `-I./protos protos/echo_example.proto` gibt im ersten Teil den Pfad zum Ordner an, in dem sich die „.proto“-Datei befindet und im zweiten Teil den relativen Pfad vom Ordner zur Datei. Daraufhin können mehrere Parameter folgen, die den Ausgabeort der Programmiersprachen abhängigen Stubs bestimmen. Der Parameter `--python_out=.` generiert den Python Code zum Serialisieren von Protobuf Nachrichten und der Parameter `--grpc_python_out=.` die Python gRPC Stubs. Der Punkt bei der beiden Parameter gibt als Ausgabeort das aktuelle Verzeichnis des Terminals an. Beiden resultierenden Codedateien sind in den Codelistings 1 und 2 einsehbar.

Ein Python gRPC-Server mit dem Interface `SendEcho`, welches Nachrichten vom Typ `EchoReply` empfängt und mit einer `EchoResponse` beantwortet, könnte wie folgt implementiert werden [11]:

```
1  from concurrent import futures
2  import grpc
3  import echo_example_pb2
4  import echo_example_pb2_grpc
5
6  class Echo(echo_example_pb2_grpc.EchoServicer):
7      def sendEcho(self, request, context):
8          print("Received the number: %i" % request.number)
9          return echo_example_pb2.EchoReply(message="You send the number %i" %
                                             request.number)
10
11 if __name__=="__main__":
12     server = grpc.server(futures.ThreadPoolExecutor(max_workers=1))
13     echo_example_pb2_grpc.add_EchoServicer_to_server(Echo(), server)
14     server.add_insecure_port('localhost:50051')
15     print("Starting gRPC Server")
16     server.start()
17     server.wait_for_termination()
```

Der in Zeile 12 erstellte gRPC Server bietet zunächst keine Schnittstelle für Clients. Durch das Aufrufen der Funktion `add_EchoServicer_to_server` aus dem generiertem Stub wird dem Server die Schnittstelle `Echo` hinzugefügt, um die Prozedur `sendEcho` aufrufbar zu machen. Ruft ein Client die Prozedur `sendEcho` auf, gibt der Server die beinhaltende Zahl der `EchoRequest` in der Konsole aus. Ist die Zahl beispielsweise 25 sendet der Server dem Client als Antwort die `EchoReply` mit dem Inhalt `"You send the number 25"`.

Die Client Implementierung um Anfragen an den Server zu senden ist vergleichsweise klein:

```
1 import grpc
2 import echo_example_pb2
3 import echo_example_pb2_grpc
4
5 if __name__=="__main__":
6    with grpc.insecure_channel('localhost:50051') as channel:
7       stub = echo_example_pb2_grpc.EchoStub(channel)
8       response = stub.sendEcho(echo_example_pb2.EchoRequest(number=7))
9    print(response.message)
```

Um die Verbindung zum Server aufzubauen, benötigt der Client den generierten Stub. Über den Stub lassen sich die vorher definierten Prozeduren aufrufen. Beim Aufrufen der Prozedur `sendEcho` wird die benötigte Nachricht `EchoRequest` mitgegeben, welche in diesem Beispiel aus einem Feld namens `number` besteht. Wird der Client ausgeführt, ruft er die Prozedur `sendEcho` auf und sendet als Inhalt der `EchoRequest` die Zahl sieben. Als Antwort vom Server empfängt der Client die Nachricht mit dem Inhalt `"You send the number 7"`.
Bei der Verwendung von gRPC muss nicht auf die Serialisierung in Binärdaten vor dem senden einer Nachricht geachtet werden, da diese Funktion bereits Teil des importierten Stubs sind. Zusätzlich bleiben Entwicklern bis auf die Adresse und der Port des Servers Informationen über den Ablauf der Kommunikation verborgen.

2.2 Der REST-Architekturstil

Als populärster Architekturstil für Web APIs bietet REST ein simples Konzept für den Austausch von Ressourcen an. Um Ressourcen von einer REST-API anzufragen, wird eine URI benötigt, welche die angefragte Ressource eindeutig identifiziert. Dies wird erreicht, in dem REST auf den HTTP-Spezifikationen aufbaut und die Anfragen meist mittels HTTP oder HTTPS transportiert. Aus den Spezifikationen stammt beispielsweise das URI-Schema oder die Methoden, um Ressourcen zu verwalten.

2.2.1 Die HTTP-Methoden

Die vier Standardmethoden auf persistenten Speicher CRUD, die für `create`, `read`, `update` und `delete` stehen, gibt es auch als HTTP-Methoden. In Kombination mit der URI lässt sich die Operation der Methode auf der identifizierten Ressource ausführen.

GET	Gibt als Antwort eine Kopie der identifizierten Ressource zurück. Die Ressource auf dem Server bleibt unverändert.
POST	Sendet die angehängten Daten an die Ressource, um eine Zustandsänderung oder andere Funktionen zu bewirken.
PUT	Überschreibt die derzeitig hinterlegte Ressource mit den angehängten Daten
DELETE	Löscht die identifizierte Ressource

Tabelle 1: Die gebräuchlichsten HTTP-Methoden und deren Nutzen

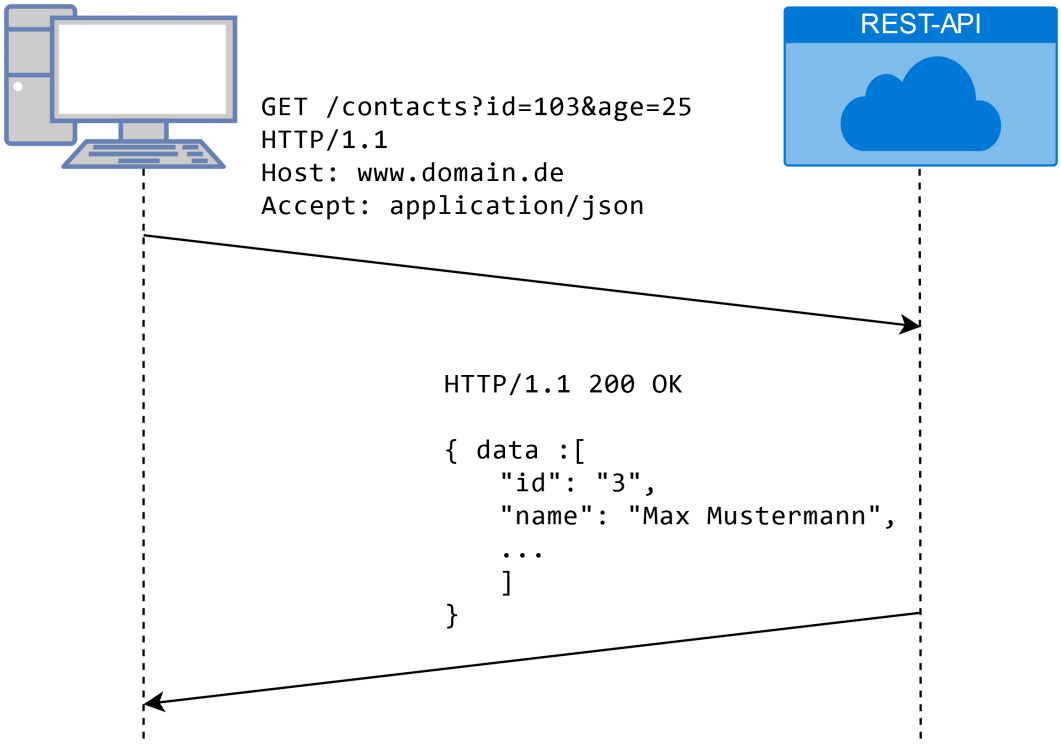

Abbildung 2: Der Client fragt über die GET-Methode eine Ressource an. Das URI-Schema beinhaltet Parameter, die zur eindeutigen Identifizierung der Ressource beitragen.

2.2.2 Kernprinzipien REST

Ob eine API RESTful ist, definiert eine Sammlung von Kernprinzipien, welche auch als architektonische Einschränkungen bekannt sind [12]. Insgesamt gibt es fünf einzuhaltende und ein optionales Kriterium:

- Die REST-API verfügt über eine einheitliche Schnittstelle, durch die jede Ressource durch eine einzige URI gekennzeichnet ist.
- Client und Server müssen entkoppelt funktionsfähig sein. Bis auf den Austausch der URI und der Ressource sollen sich beide Kommunikationspartner nicht gegenseitig beeinflussen.
- Die REST-API ist statusunabhängig. Es müssen keine Sitzungen über mehrere Anfragen aufgebaut werden, da jede Anfrage bereits alle Informationen zum Bearbeiten enthalten sollte.
- Angefragte Ressourcen des Servers sollten von Client und Server zwischengespeichert werden können. Hierfür muss der Server beim Senden einer Ressource mit angeben, ob die Zwischenspeicherung erlaubt ist. Serverseitiges Cachings verbessert die Skalierbarkeit der REST-API.
- Systeme sollen mehrschichtiges System aufgebaut sein. Ein soll Client nicht wissen, mit was für weiteren Systemen die REST-API kommuniziert. Ob für eine Antwort beispielsweise eine Datenbank aufgerufen wurde, ist für den Client unbekannt.
- Als optional angesehen ist die Möglichkeit, über die Antwort der REST-API ausführbaren Code zu versenden.

Wie die Prinzipien eingehalten werden, bleibt dem Entwickler überlassen.

2.3 Das WebSocket Protokoll

Das im RFC6455 [13] definierte WebSocket Protokoll wird meist für Echtzeitanwendungen im Web genutzt. Der Vorteil von WebSockets ist, dass Client und Server simultan Daten senden können. Um dies zu erreichen, muss ähnlich wie bei TCP ein Opening Handshake durchgeführt werden. Der Client sendet hierfür eine HTTP-Anfrage an den Server, die unter den Upgrade Header mit dem WebSocket Protokoll enthält. Bestätigt der Server die Anfrage, in dem er als Header Switching Protocols setzt, wird die Verbindung zu einer WebSocket Verbindung aufgewertet.
Nach dem Verbindungsaufbau können beide Kommunikationspartner Daten senden und empfangen. Durch das Aufwerten des Protokolls müssen die gesendeten Daten keinen HTTP Header mehr enthalten. Server oder Client dürfen jederzeit einen PING-Frame senden, welcher der Kommunikationspartner frühzeitig mit einem PONG-Frame beantwortet, um einen Heartbeat Mechanismus zu implementieren. Das Schließen der Verbindung erfolgt über den Closing Handshake, welcher aus einem gegenseitigen Austausch des CLOSE-Frames besteht. Die definierten Frames entsprechen jeweils den Opcodes `0x8 (Close), 0x9 (Ping), and 0xA (Pong)`.

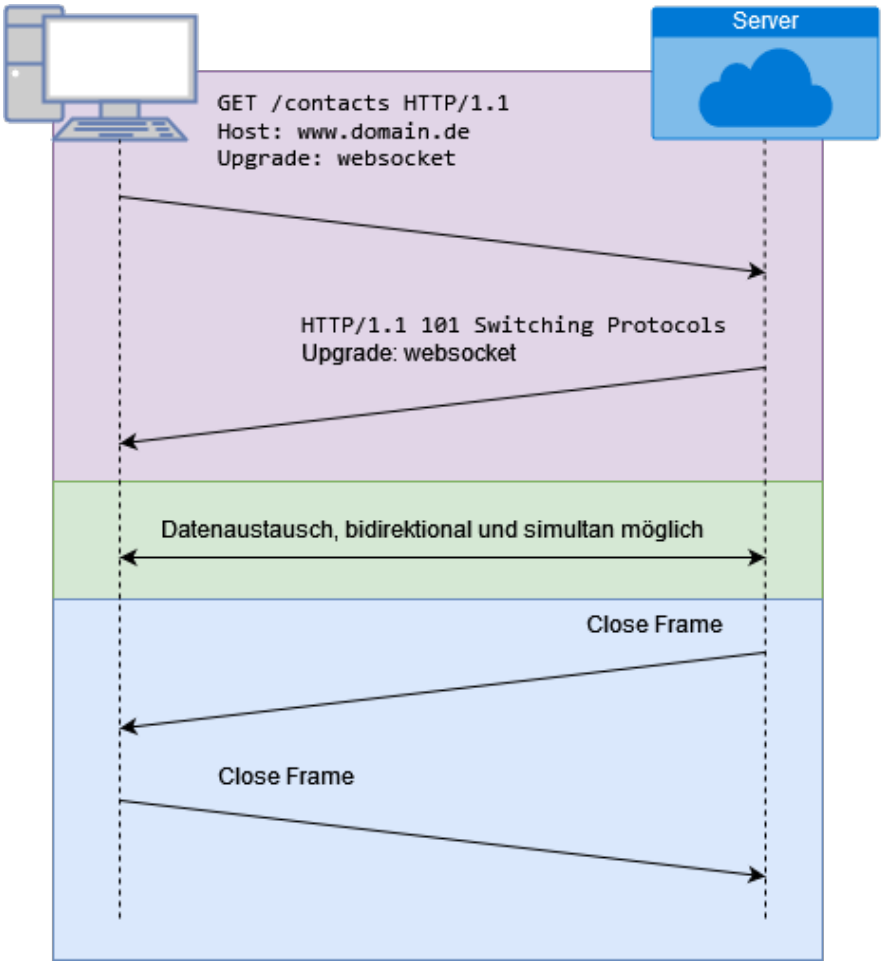

Abbildung 3: Der Opening Handshake mit dem gesetzten Upgrade-Header um von dem HTTP Protokoll auf das WebSocket Protokoll zu wechseln. Der Verbindungsabbau wird durch den Closing Frame signalisiert. (Diagramm erstellt nach [14])

3 Erläuterung der Vergleichskriterien

Um die Vor- und Nachteile von gRPC gegenüber WebSocket und REST basierten Web APIs zu bestimmen, bietet sich ein Vergleich anhand eines Software Qualitätsmodells an, welches die wichtigsten Merkmale beschreibt. Die gängigsten Modelle sind die Norm ISO/IEC 25010, das Vorgängermodell der Norm ISO/IEC 9126 [15], so wie das McCall- [16] und das Boehm-Qualitätsmodell [17], welche bereits in den 1970er Jahren entwickelt wurden. Da das McCall und das Boehm Modell jedoch ohne Ergänzungen nicht alle relevanten Aspekte wie beispielsweise die Sicherheit und Testbarkeit abdecken, werden für den Vergleich die Merkmale der ISO/IEC 25010 verwendet.
Das Modell der ISO-Norm unterteilt die Kriterien in acht Hauptmerkmale, die bei Erfüllung die Softwarequalität sicherstellen. Jedes der acht Hauptmerkmale ist weiter unterteilt in Teilmerkmale, die einzelne Aspekte abdecken.

Hauptmerkmal	Teilmerkmal	Kurzbeschreibung
Funktionalität	Vollständigkeit	Vorhandensein der erwarteten Funktionen
	Korrektheit	Vorkommen von unerwarteten Ergebnissen
	Angemessenheit	Erfüllung der Anforderungen an Funktionen
Effizienz	Kapazität	Erwartete Verarbeitungskapazitäten erfüllen
	Zeitverhalten	Durchsatz, Antwort- und Verarbeitungszeiten
	Verbrauchsverhalten	Auslastung von Ressourcen
Kompatibilität	Koexistenz	Funktionsfähigkeit trotz Teilen von Ressourcen
	Interoperabilität	Kompatibilität mit weiteren Systemen
Benutzbarkeit	Erlernbarkeit	Fähigkeit schnell und einfach erlernt zu werden
	Erkennbarkeit	Nutzer verstehen Zwecke von Funktionen
	Bedienbarkeit	Leicht nutzbare und kontrollierbare Software
	UI-Ästhetik	User-Interface angenehm und zufriedenstellend
	Fehlervermeidung	Vor Fehleingaben schützende Maßnahmen
	Zugänglichkeit	Barrierefreie Gestaltung der Software
Zuverlässigkeit	Fehlertoleranz	Möglichst geringe Beeinträchtigungen bei Fehlern
	Reife	Funktionsfähigkeit über lange Zeiträume
	Wiederherstellbarkeit	Wiederherstellbarkeit von Zuständen bei Fehlern
	Verfügbarkeit	Ständige Verfügbarkeit ohne Ausfälle
Sicherheit	Vetraulichkeit	Schutz vor unautorisiertem Zugriff
	Integrität	Unveränderbarkeit einer Nachricht
	Verbindlichkeit	Ausgeführte Aktionen lassen sich nicht abstreiten
	Authentizität	Identitätsbeweis des Senders
	Verantwortlichkeit	Rückverfolgbarkeit von Aktionen auf Nutzer
Wartbarkeit	Analysierbarkeit	Änderungen und Defizite beurteilen
	Modifizierbarkeit	Aufwand Software zu erweitern oder zu ändern
	Modularität	Unabhängig austauschbare Komponenten
	Testbarkeit	Umfangreiche Testoptionen
	Wiederverwendbarkeit	Wiederverwendbare Module und Komponenten
Portabilität	Adaptierbarkeit	Anpassbarkeit auf neue Anforderungen
	Installierbarkeit	Leichte Installation auf weiteren Umgebungen
	Austauschbarkeit	Austauschen von Parts ohne Beeinträchtigungen

Tabelle 2: ISO/IEC 9126 nach [18]: Die Haupt- und Teilmerkmale um Softwarequalität sicherzustellen mit Kurzbeschreibung

Obwohl die Merkmale generisch formuliert wurden, um eine breite Anwendbarkeit auf verschiedenste Arten von Software zu ermöglichen, kann es dennoch vorkommen, dass Web-APIs für einzelne Teilmerkmale schwierig ausgewertet werden können. Zur Hilfe der manchmal sehr generisch formulierter Merkmale hilft die Interpretation der Merkmale aus [19]. Des Weiteren kann es vorkommen, dass bestimmte Merkmale, die für diese Arbeit von Bedeutung wären, aufgrund des begrenzten Zeitrahmens nicht bearbeitet werden können.
Des Weiteren werden die Teilmerkmale des Hauptmerkmals Sicherheit umstrukturiert, da sich sonst in Bezug auf Web-APIs die Auswertungen in den Teilmerkmalen voraussichtlich überschneiden.

4 Vorbereitung

Für die Durchführung des Vergleichs, werden für manche Merkmale eine Implementierung benötigt, um anhand dieser Messungen zu erstellen. Dementsprechend behandelt dieses Kapitel die Umsetzung von drei Client- und Server-Anwendungen. Die Implementierung der Anwendungen unterscheiden sich jeweils im Kommunikationskonzept, wobei die Server jeweils eine gRPC-, REST- oder Websocket-API und die Clients das jeweilige Gegenstück für die Kommunikation implementieren. Damit der Vergleich ein bestmögliches Ergebnis liefert, müssen Anwendungen zwei Bedingungen erfüllen. Zum einen sollten sich die Anwendungen größtenteils in der Implementierung der Kommunikation zwischen Client und Server unterscheiden und zum anderen muss das Anwendungs-Szenario im Request-Response-Modell als auch für bidirektionales Streaming anwendbar sein.
Als Anwendungs-Szenario wurde die Umsetzung eines Zeichenprogramms gewählt, bei dem der Server den Zustand einer Canvas-Zeichenfläche verwaltet und Clients nach dem Verbinden ihr Canvas mit dem des Servers synchronisieren. Umgesetzt wird das Szenario in der Programmiersprache Python mit der Version 3.10. Ein ähnliches Anwendungs-Szenario ist die Web-Anwendung Miro [20], welche durch das Anbieten eines Canvas mit verschiedenen Farben, Formen und Symbolen die virtuelle Zusammenarbeit von Gruppen fördert.

4.1 Umsetzung der Client-Anwendung

Für das Erstellen einer GUI bietet sich die Python Bibliothek Tkinter an, da diese für MacOS, Windows und die meisten Unix-Plattformen verfügbar ist. Das GUI des Clients besteht aus einem Canvas Widget, einer Maßstableiste um die Dicke von Linien zu steuern und jeweils einem Button um Linien zu zeichnen, die Farbe des Stifts zu ändern und einer Radierfunktion.

Abbildung 4: Das GUI der ausgeführten Client-Anwendung

Beim erstellen der Elemente kann mit dem Argument `command` eine Callbackmethode angebunden werden, die sich bei Interaktion aufruft.

```
class DrawClient(object):
    def __init__(self):
        self.root = Tk()
        self.canvas = Canvas(self.root, bg="white", width=600, height=600)
        self.pen_button = Button(self.root, text="Draw", command=self.use_pen
                                 )
        self.color_button = Button(self.root, text="Color", command=self.
                                   choose_color)
        self.eraser_button = Button(self.root, text="Eraser", command=self.
                                    use_eraser)
```

Das Zeichnen auf das Canvas erfolgt durch das Anbinden des Callbacks `paint` und der Tastenbelegung `<B1-Motion>`, was einer Bewegung der Maus während gedrückter Linksklick-Taste entspricht. Zusätzlich wird der Callback `drawedLine` mit der Tastenbelegung `<ButtonRelease-1>` angebunden, was für die Freigabe der Linksklick-Taste steht.

```
        self.canvas.bind("<B1-Motion>", self.paint)
        self.canvas.bind("<ButtonRelease-1>", self.drawedLine)
```

Die Callbackmethode `paint` wird im Client verwendet, um ein Datenobjekt mit den Koordinaten der gezeichneten Linie zu erstellen. Inhalte des Objekts sind die Farbe der Linie, die Breite der Linie und zwei Listen für alle x-y-Koordinatenpaare, um die Linie zu Zeichnen. Wird die Callbackmethode `drawedLine` aufgerufen, ist die Linie gezeichnet und das Objekt ist bereit zum Versenden an den Server. Der Aufbau eines Datenobjekts, dass eine gezeichnete Linie beschreibt, sieht wie folgt aus.

```
{
    'x': [98, 97, 96, 95, 94, 93],
    'y': [35, 35, 35, 35, 35, 35],
    'color': 'black',
    'width': 3
}
```

Die Kommunikation mit der API wird in einem seperaten Thread gehandhabt, da die `mainloop` der GUI blockierend ist. In allen Client-Anwendungen ist der Ablauf zum synchronisieren des Canvas mit dem Server gleich. Jede Zeichnung auf dem Canvas entspricht einem Datenobjekt, welches der Liste `lines_to_send` hinzugefügt wird. Sind Einträge in der Liste `lines_to_send` vorhanden, sendet der Client diese an die API. Anschließend überprüft der Client die API auf noch nicht gezeichnete Elemente. Gibt es keinen Eintrag in der Liste `lines_to_send`, wird der Schritt übersprungen und die API wird direkt nach noch nicht gezeichneten Elementen angefragt. Dieser Ablauf wird wiederholt bis zur Beendigung des Clients.

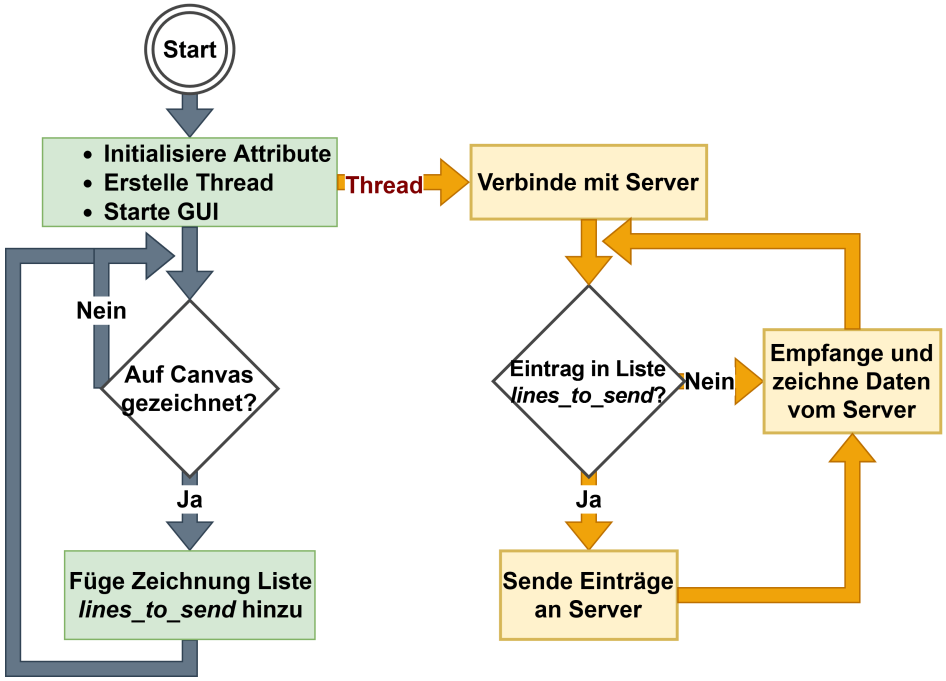

Abbildung 5: Ablauf der Synchronisation des Client Canvas mit dem Server

4.2 gRPC

Eine Kommunikation zwischen eines gRPC Servers und Clients setzt eine kompilierte Schnittstellendefinition voraus. In der Kommunikation zwischen Client und API gibt es zwei verschiedene Fälle die zu beachten sind. Im ersten Fall wurde ein Client neu gestartet und fragt alle bisher gezeichneten Linien anderer Clients von der API an. Der zweite Fall ist, dass der Client das Objekt einer gezeichneten Linie an die API sendet. In beiden Fällen empfängt der Client als Antwort einen Stream, welcher die gesendeten Objekte der anderen Clients beinhaltet. Die daraus folgende Schnittstellendefinition `canvas_server_client.proto`:

```
1  syntax = "proto3";
2
3  service CanvasSync {
4      rpc getServerCanvas (Empty) returns (stream Canvas) {}
5      rpc updateServerCanvas (stream Canvas) returns (stream Canvas) {}
6  }
7
8  message Canvas {
9      repeated uint32 x = 1;
10     repeated uint32 y = 2;
11     string color = 3;
12     uint32 width = 4;
13   }
14
15 message Empty {}
```

Die Proto-Nachricht `Empty` entspricht einer Nachricht ohne Inhalt, die verwendet wird, falls ein Client neu gestartet wurde und deshalb noch keine Daten zu senden hat.
Der Thread `start_grpc_client` implementiert den gRPC-Client-Stub. Zu Beginn speichert der Thread die Prozess-ID, damit diese als Metadaten an jede Nachricht beigefügt werden kann. Daraufhin wird die Verbindung über den Stub mit dem gRPC-Server aufgebaut und die gRPC-Methode `getServerCanvas` aufgerufen, um das Canvas mit dem Zustand der API zu synchronisieren. Nun kann in bestimmten Abständen die `updateServerCanvas`-Methode des Stubs aufgerufen werden, um zuerst Objekte der Liste `lines_to_send` einzeln zu übertragen und als Antwort einen Stream von noch nicht gezeichneten Objekte zu empfangen. Der Code der gRPC-API befindet sich in Codelisting 3.

4.3 REST

Zur Umsetzung der REST-API wird das Python-Web-Framework FastAPI verwendet. Im Gegensatz zu anderen REST Frameworks wie Flask bietet FastAPI von sich aus das `BaseModel` als Schema zur Validierung der erwartenden Daten auf den Endpunkten. Stimmen empfangene Datentypen nicht mit dem des `BaseModel`s überein wird eine `RequestValidationError` Antwort an den anfragenden Client gesendet. Das `BaseModel` für eine gezeichnete Linie sieht wie folgt aus.

```
class Line(BaseModel):
    x: list = []
    y: list = []
    color: str
    width: int
```

Um neu gezeichnete Linien an die REST-API zu senden und die bisher von anderen Clients gesendeten Linien anzufragen, reicht der Endpunkt `/canvas/lines` aus. Mit der HTTP-Methode GET können Linien durch die API angefragt und mit POST hinzugefügt werden.

```
@app.get("/canvas/lines")
async def get_lines():
    return lines

@app.post("/canvas/lines", status_code=201)
async def add_line(line: Line):
    lines.append(line)
    return line
```

Auf Client Seite wird die `drawedLine` Callbackmethode erweitert, um bei jedem Aufruf eine POST-Anfrage mit der gezeichneten Linie zu senden.

```
def drawed_line(self, event):
    requests.post("http://127.0.0.1:5000/canvas/lines", json=self.line_dict)
```

Das Abfragen des Enpoints mit der GET-Methode erfolgt in einem seperaten Thread. Mittels Polling wird in regelmäßigen Zeitabständen von mindestens einer Sekunde der Server auf noch nicht gezeichnete Linien anderer Clients überprüft. Als Antwort erhält der Client eine Liste von Linien-Objekten, iteriert durch die Einträge der Liste und fügt sie dem Canvas zu.

```
while True:
     response = requests.get("http://127.0.0.1:5000/canvas/lines")
     old_x, old_y = None
     width=response.json()[index]["width"]
     color = response.json()[index]["color"]
     for x,y in zip(response.json()[index]["x"],response.json()[index]["y"]):
          if old_x and old_y:
               self.canvas.create_line(old_x, old_y, x, y,
                    width=width, fill=color, capstyle=ROUND, smooth=TRUE)
          old_x = x
          old_y = y
     sleep(0.1)
```

Der Code der REST-API befindet sich in Codelisting 4.

4.4 WebSocket

Die WebSockets-Bibliothek in Python3 beinhaltet die Python Implementierung zum Erstellen von WebSocket Servern und Clients. Auch hier benötigt die Kommunikation nur einen Endpunkt. Beim Start eines Clients verbindet sich dieser mit dem Server, sendet eine leere Nachricht, worauf mit allen Objekte geantwortet, die noch nicht gezeichnet wurden, um das Canvas auf den aktuellen Stand zu synchronisieren.
Um bidirektionale Verbindungen zu unterstützen, können zusätzlich die `asyncio` oder die `gevent` Bibliothek verwendet werden. Serverseitig wird die Bibliothek genutzt, um über einen Thread mehrere Verbindungen zu verwalten, anstatt jeder Verbindung einen eigenen Thread zuzuweisen.
Die Verwendung der `asyncio` Bibliothek in der WebSocket API:

```
1  import asyncio
2  import websockets
3
4  async def sync_canvas(websocket):
5      async for message in websocket:
6          user_id = process_message(message)
7          await websocket.send(get_response(user_id))
8
9  async def main():
10     while True:
11         async with websockets.serve(sync_canvas, "192.168.0.1", 50051):
12             await asyncio.Future()
13
14 if __name__ == "__main__":
15     asyncio.run(main())
```

Der ungekürzte Code des WebSocket Servers befindet sich in Codelisting 5.

5 Vergleich

5.1 Funktionalität

Das Kapitel Funktionalität befasst sich zum einen mit dem Erfüllen von benutzerdefinierten Anforderungen, sowie dem Bereitstellen von Daten im erwarteten Format. Die Punkte Vollständigkeit und Angemessenheit, welcher sich mit dem Erfüllen der Anforderungen sowie dem Bereitstellen von erwarteten Funktionen befassen, werden aufgrund des beschränkten Zeitraums nicht weiter ausgeführt.

5.1.1 Korrektheit

Die Korrektheit von APIs bezieht sich auf die Fähigkeit, erwartete Ergebnisse an den Anfragenden bereitzustellen und die Richtigkeit der Daten zu garantieren. Um dies zu ermöglichen, muss die Anfrage an die API erfolgreich verarbeitet werden und die Antwort dem erwarteten Format entsprechen. Das erfolgreiche Verarbeiten von Anfragen kann sichergestellt werden, in dem Fehler behandelt werden. Abgesehen von der Fehlerbehandlung, die je nach Implementierung unterschiedlich ausfallen kann, lassen sich die verschiedenen Kommunikationskonzepte in Bezug auf der Korrektheit vergleichen.
gRPC nutzt zum Serialisieren der Nachrichten Protobuf, was eine vorab festgelegte Struktur der Nachrichten voraussetzt. Somit können die API und anfragende Nachrichten nur senden, wenn sie auch in der Schnittstellendefinition spezifiziert sind. Beim Senden oder Empfangen von unerwarteten Formaten wird der Statuscode `INVALID_ARGUMENT` [21] zurückgegeben.
Mit Schemas bietet REST eine vergleichbare Maßnahme, um die Korrektheit zu gewährleisten. Über REST-Schemas können für Ressourcen-Endpunkten und den HTTP-Methoden die Strukturen von Anfragen und Antworten definiert werden [22]. Jedoch sind Schemas kein Teil der Kernprinzipien RESTs und somit nicht in jedem Framework gegeben. Für die meisten Frameworks ohne Schema gibt es die Möglichkeit eine Validierungsbibliothek zu integrieren, welche meist auch noch weitere Funktionalitäten bieten.
Die Einhaltung der Korrektheit bei WebSockets ist in den meisten Fällen im Vergleich zu gRPC und REST mit dem größten Aufwand verbunden. Durch die freie Auswahl der Datenformate gibt es die Möglichkeit sich auf ein festes Format wie JSON zu beschränken, um von JSON-Schemas zu profitieren. Jedoch gibt es für beliebige Formate, keine bekannten Bibliotheken oder Frameworks, die bei der Validierung unterstützen, weshalb die Validierungschritte eigenhändig implementiert werden müssen.

5.2 Effizienz

In dem Kapitel Effizienz werden die vorbereiteten Implementierungen jeweils auf ihr Verbrauchsverhalten und dem Zeitverhalten getestet. Es ist aber wichtig zu beachten, dass die Ergebnisse nicht nur auf das gesamte Framework oder auf das Kommunikationsmodell zu folgern sind, da die Ergebnisse sich je nach Anwendungs-Szenario unterscheiden können.

5.2.1 Verbrauchsverhalten

Das Verbrauchsverhalten umfasst die von der API erzeugten Auslastung auf Prozessor, Speicher und Netzwerk. Eine API mit einem guten Verbrauchsverhalten sollte schnell auf Anfragen reagieren, in dem die zur Verfügung stehenden Ressourcen genutzt werden. Das Gegenteil dazu sind schlechte APIs, die Ressourcen grundlos belasten und langsame Antworten zurückgeben. Um das Verbrauchsverhalten zu testen und auszuwerten, werden eine API und zwei zugehörige Clients jeweils auf einer eigenen Virtual Machine mit 2 Prozessoren und vier Gigabyte Arbeitsspeicher ausgeführt. Beide Clients senden in Abständen von einer Sekunde eine neue Zeichnung an die API und fragen zusätzlich alle 100 Millisekunden nach noch nicht gezeichneten Linien anderer Clients. Über einen Zeitraum von zehn Minuten werden nun beide Clients sowie die API ausgeführt und die Auslastung der VMs in unregelmäßigen Abständen zwischen 100 und 1000 Millisekunden aufgezeichnet. Da die Anzahl der Anfragen und die Größe der gesendeten Daten über den gesamten Zeitraum konstant bleiben, zeichnet sich ein angemessenes Verbrauchsverhalten in diesem Testfall dadurch aus, dass auch die Auslastung der VM mit der API möglichst konstant bleibt.
Beim Messen der gRPC-API gab esüber den gesamten Zeitraum nur geringe CPU Schwankungen. Die Auslastung des Arbeitsspeichers blieb sehr konstant.

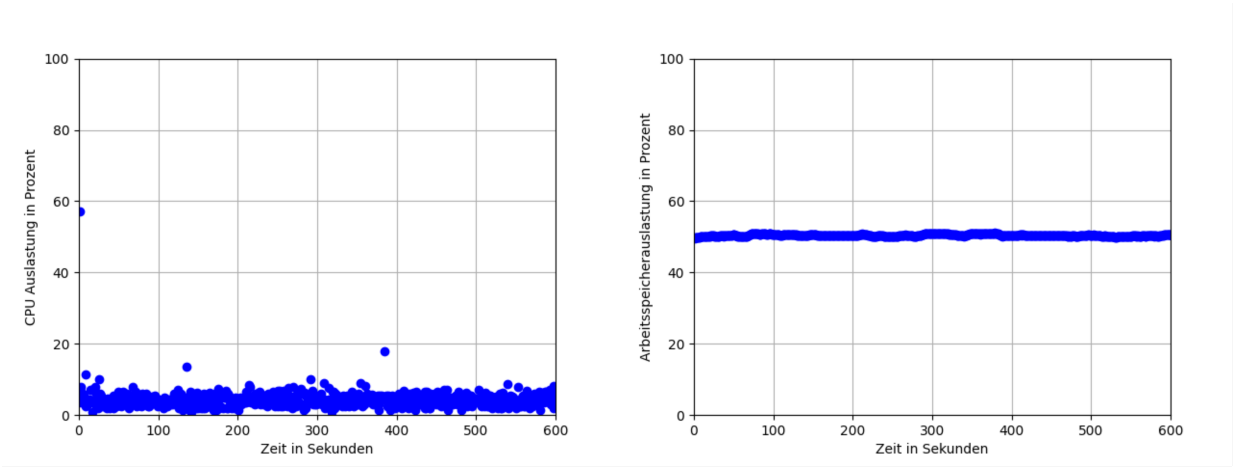

Abbildung 6: Ressourcenauslastung der gRPC-API zwei anfragende Clients

Auch bei REST ist die Auslastung insgesamt konstant geblieben. Die CPU-Auslastung ist im Vergleich zu gRPC ein wenig höher, gestreuter und auch Ausschläge kommen häufiger vor. Die Arbeitsspeicherauslastung ist auch hier konstant.

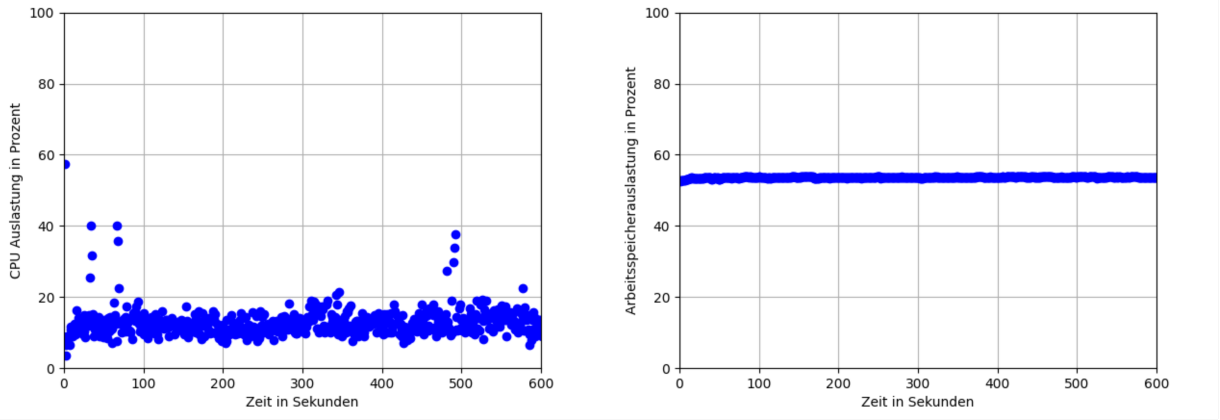

Abbildung 7: Ressourcenauslastung der REST-API zwei anfragende Clients

Die Auslastung der WebSocket-API ist insgesamt konstant geblieben und beansprucht den Prozessor etwas weniger als gRPC.

Für eine Messung, bei der die Unterschiede voraussichtlich genauer erkennbar sind, fehlten

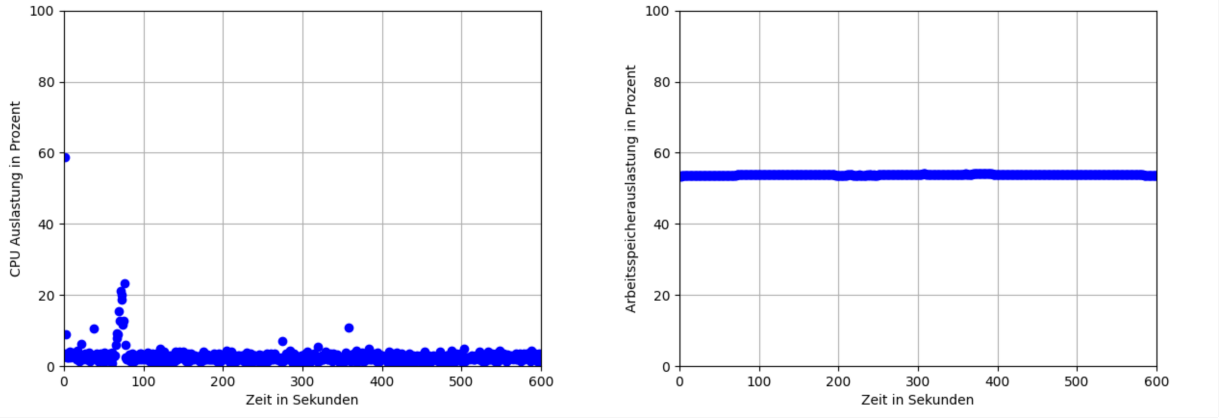

Abbildung 8: Ressourcenauslastung der WebSocket-API für zwei anfragende Clients

auf der Testumgebung die Ressourcen, um weitere Client zu nutzen. Dennoch ist zu erkennen, dass die Streuung, sowie die Auslastung der Prozessorleistung am größten ist. Es ist auch zu vermuten, dass sich bei höheren Auslastungen dies genauer erkennbar macht. In allen drei APIs blieb die Auslastung des Arbeitsspeichers konstant. Grund dafür ist, da die verwaltete Liste von Canvas-Zuständen persistent abgespeichert wurde, damit die wachsende Liste sich nicht als lineares Wachstum auf dem Arbeitsspeicher abbildet. In dieser Arbeit nicht behandelt, aber dennoch interessant wäre eine Messung, inwieweit sich die Auslastung des Arbeitsspeichers für eine steigende Anzahl von Clients verhält.

5.2.2 Zeitverhalten

Zum Beurteilen wie lang eine Anfrage insgesamt benötigt bietet die Round Trip Time, kurz RTT, einen aussagekräftigen Wert. Die RTT berechnet sich aus der Summe der Zeiten T_S, die ein Paket benötigt, um vom Sender an den Empfänger zu gelangen, T_E der Dauer, bis der Empfänger bereit ist, eine Antwort zu senden und der Zeit T_R bis das Antwortpaket am Sender angekommen ist.

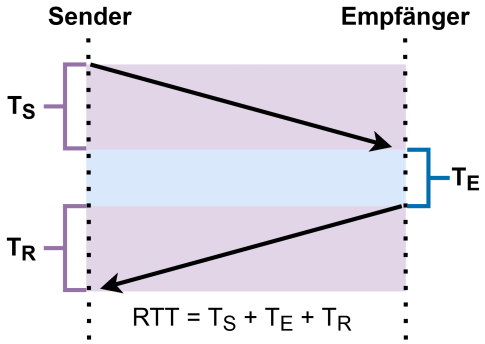

Abbildung 9: Berechnung der Round Trip Time (RTT) nach [23]

Für die Messung der RTT werden jeweils ein Rechner mit der Client-Anwendung und ein weiterer mit dem Server-Skript über einen Router verbunden. Somit können beeinflussende Faktoren wie die Distanz zwischen den Kommunizierenden und die Anzahl der Hops als konstant angesehen werden.
Da unterschiedlich große Zeichnungen auf dem Canvas auch unterschiedlich großen Datenpaketen entsprechen, zeichnet der Client zwei gleich große zufällig positionierte Punkte pro Sekunde auf dem Canvas, um eine möglichst gleichmäßige RTT zu erzeugen. Die Messung erfolgt durch einer ungefähr 60-sekündigen Aufzeichnung der Kommunikation mittels des Packet-Sniffers Wireshark.

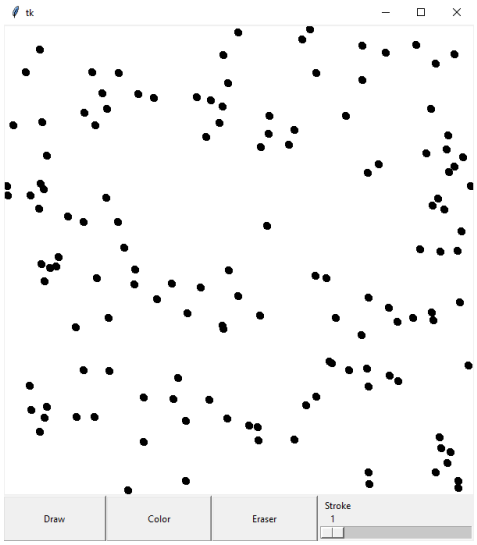

Abbildung 10: Der seit 60 Sekunden ausgeführte Client beinhaltet ungefähr 120 Punkte

Im Durchschnitt liegt die RTT bei WebSockets zwischen fünf und zehn Millisekunden. Bei den Messwerten gibt es einzelne Ausreißer, was voraussichtlich darauf zu führen ist, dass die Rechner mit dem Router über WLAN verbunden sind.

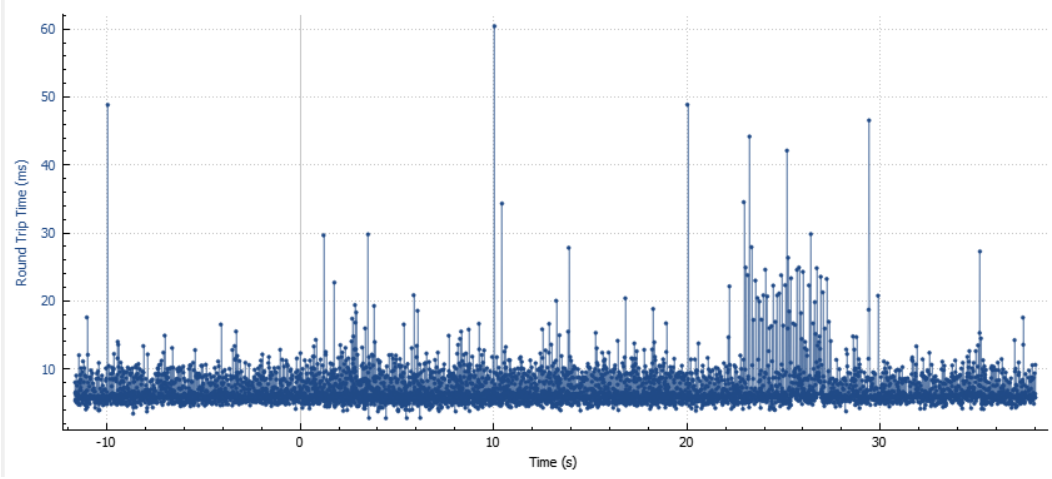

Abbildung 11: Die Messungen der Round Trip Time des WebSocket Clients

Auch bei der Messung der RTT für den gRPC Client gibt es Ausreißer. Die durchschnittliche RTT liegt hier zwischen fünf und 15 Millisekunden.

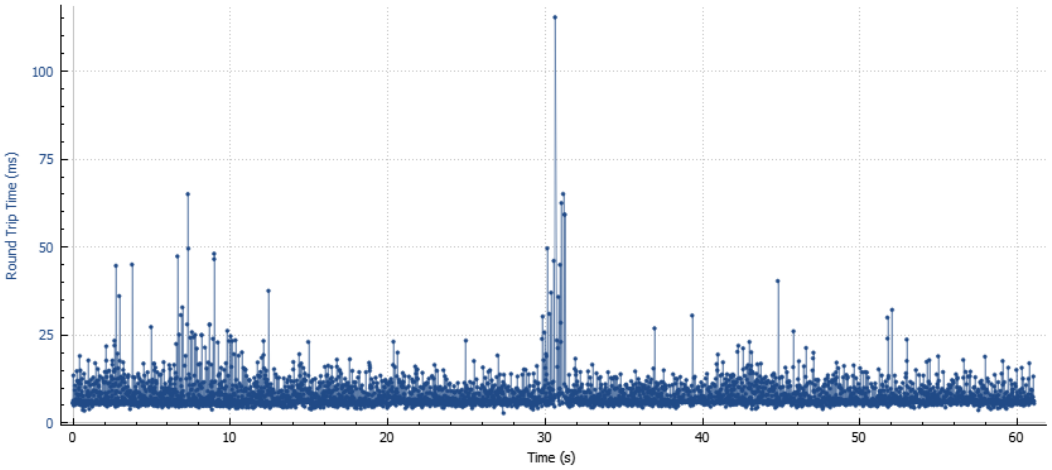

Abbildung 12: Die Messungen der Round Trip Time des gRPC Clients

Die Messung der REST-basierten API lässt sich im Vergleich zu den gRPC und WebSocket Messungen schwieriger darstellen, da für jede HTTP-Anfrage eine neue TCP-Verbindung geöffnet wird und eine Anfrage mehrere RTTs liefert [24]. Die erste RTT entsteht beim TCP-Verbindungsaufbau und die zweite bei der Ausführung der eigentlichen HTTP-Methode, weshalb die Summe der beiden Werte für den Vergleich gewählt wird. Dieser Wert beinhaltet zwar die Antwort der HTTP-Methode, aber nicht die Übertragung des Response-Bodies, welcher die angefragten Ressourcen enthält.

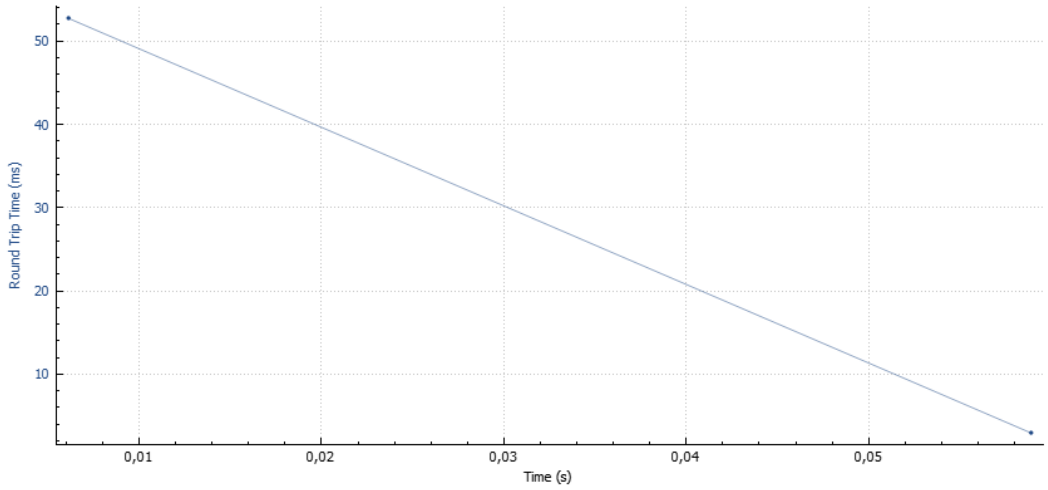

Abbildung 13: Eine einzelne HTTP-Anfrage bestehend aus mindestens zwei RTTs

Daraus lässt sich folgern, dass REST mit einer RTT von mindestens 53 Millisekunden plus der Datenübtragung im Vergleich langsam ist. Die Reduzierung der RTT ist auch einer der Hauptgründe für die Entwicklung von HTTP\2 gewesen [25].

5.3 Kompabilität

Die Kompabilität befasst sich mit der Zusammenarbeit von Systemen und ob diese Daten austauschen können. Die Koexistenz ist ein Teilmerkmal der Kompabilität und beschreibt, dass eine Software ihre Funktionen erfüllt, ohne die Funktionalitäten anderer Software auf derselben Umgebung einzuschränken. Im Rahmen dieser Arbeit wird die Koexistenz jedoch aufgrund des beschränkten Zeitraums nicht weiter ausgeführt.

5.3.1 Interoperabilität

Die Interoperabilität beschreibt die Fähigkeit, mit anderen Systemen kommunizieren zu können, ohne dafür spezifische Anpassungen vorzunehmen. REST bietet durch das Einhalten der REST-Prinzipien eine gute Interoperabilität, da jede Ressource durch eine eindeutige URI gekennzeichnet ist und mit den HTTP-Methoden bedient werden kann. Ein weiterer Punkt, für die gute Interoperabilität ist die weite Verbreitung der Kommandozeilen-Anwendungen cURL und wget, die auf vielen Betriebssystemen vorhanden sind. Durch sie lassen sich Anfragen mit den HTTP-Methoden an die API senden. Nur wissen die anfragenden Systeme weder, wie die Struktur der Daten aussieht, noch welche Endpunkte vorhanden sind. Als Lösung bietet sich eine SwaggerUI an, die als Webanwendung eine Dokumentation, sowie ein User-Interface zum Bedienen der API vereint. Durch die SwaggerUI kann somit jeder berechtigte Nutzer mit seinem Webbrowser alle Endpunkten und die dazu gehörigen Datenformaten einsehen [26]. Das Erstellen der SwaggerUI erfolgt über dafür spezialisierte Tools, welche eine einbindbare HTML-Datei zurückgeben. Die Datei kann anschließend für den Aufruf einer URI, wie beispielsweise `https://localhost:8080/api/docs` mit der GET-Methode zurückgegeben werden.
Für gRPC wird bereits durch das Implementieren der Stubs die Interoperabilität sichergestellt. Beide Stubs nehmen nur Daten an, die auch in der Schnittstellendefinition angegeben wurden. Des Weiteren gibt es eine von der Community entwickelte Sammlung von Tests, um die Interoperabilität zwischen Programmiersprachen und Plattformen zu testen [27].
Websocket-basierte APIs haben im Vergleich wenige Optionen die interoperabel mit anderen Systemen zu sein. Durch das beschränken des Datenformates auf beispielsweise JSON, lässt sich ein Datenformat bestimmen, dass von vielen Systemen unterstützt wird. Ansonsten gibt es auch hier durch das Autobahn-Framework [28] eine Sammlung von Tests um die Interoperabilität zu testen.
Implementieren jedoch die gRPC-, Websocket- oder REST-API einen Teil der genutzten Kommunikationsprotokolle falsch, ist die Interoperabilität nicht mehr gegeben [29]. Würde zum Beispiel die REST-API eine veraltete Version HTTPs implementieren, könnten die meisten Webanwendungen und Systeme nicht mehr mit der API kommunizieren.

5.4 Benutzbarkeit

Der Aufwand, den ein Entwickler benötigt, um die Software zu entwickeln und inwieweit sie barrierefrei und intuitiv bedienbar ist, beschreibt die Benutzbarkeit. Die Teilaspekte Erlernbarkeit, Erkennbarkeit, UI-Ästhetik, Fehlervermeidung und Zugänglichkeit können aufgrund der beschränkten Zeit nicht bearbeitet werden.

5.4.1 Bedienbarkeit

Eine bedienbare Software zeichnet sich dadurch aus, dass sie einfach und schnell zu bedienen ist. Im Aspekt der Einfachheit bietet REST durch die Operationen der HTTP-Methoden und dem klaren Ablauf des Request-Response-Models leicht implementierbare Aufrufe mit einem nachvollziehbaren Ablauf. Bei Websockets hingegen ist die Implementierung ein wenig aufwändiger, da es sich um eine persistente Verbindung handelt, über die je nach Anwendungsfall beidseitig Nachrichten verschickt werden können. Aufgrund der Client-Stubs, welche die entsprechende Prozessur aufrufen und die Daten serialisieren, sind die gRPC-Clients im Vergleich am schwersten zu bedienen. Jedoch bieten gRPC und WebSocket die schnellste Möglichkeit, um Daten zu übertragen. Dementsprechend muss je nach Anwendungsfall gewählt werden, ob eine bessere Einfachheit oder Geschwindigkeit der Bedienbarkeit mehr hilft.

5.5 Zuverlässigkeit

Die Zuverlässigkeit umfasst die Fähigkeit einer Software, Fehler zu vermeiden oder diese möglichst schnell zu beheben. Des Weiteren sollten durch Fehler verlorene Daten möglichst wiederherstellbar sein und die Software sollte trotz auftetenden Fehlern ihre Funktionalitäten erfüllen. Das Teilmerkmal Wiederherstellbarkeit, was sich mit der Wiederherstellung von Datenintegritäten bei Fehlern befasst, wird im folgenden nicht weiter ausgeführt.

5.5.1 Fehlertoleranz

Ein fehlertolerantes System zeichnet sich dadurch aus, dass bei Fehlern die Funktionalität des Systems so wenig wie möglich eingeschränkt werden. Für APIs bedeutet dies, dass Fehler die Erreichbarkeit der API nicht beeinträchtigen und dem Client eine Fehlermeldung übermittelt wird, die das Problem beschreibt, um weitere Probleme vorzeubeugen.
REST-APIs bieten dies, in dem Fehler im Code der Serveranwendung zu abgefangen werden und in der Antwort entsprechend den Statuscode `4xx` für einen Client-Fehler oder `5xx` für einen Server-Fehler [30] zu setzen. Manche Frameworks wie Flask bieten auch die Möglichkeit einen sogennanten Errorhandler, als Callback-Methode zu registrieren, der bei Fehlern aufgeruft wird, um bei unerwarteten Fehlern dem Client zu antworten.
Tritt ein Fehler in einer Websocket-API auf, kann ähnlich wie bei REST eine beschreibende Fehlermeldung an den Client gesendet werden. Treten jedoch Probleme mit der Netzwerkverbindung auf, versucht der Client die Verbindung mit dem Server wieder aufzubauen. Auch hier helfen manche Frameworks durch das Registrieren von Callback-Methoden, die bei verlorener Verbindung aufgerufen werden. Das Behandeln und Übertragen von Fehlern innerhalb der API muss jedoch selber programmiert werden.
Im Vergleich zu REST und Websocket basierten APIs bietet gRPC mehr Mechanismen, um die Fehlertoleranz sicherzustellen. Wie bei REST, antwortet gRPC auf ungültige Anfragen beispielsweise mit dem Statuscode `INVALID_ARGUMENT` oder `NOT_FOUND`. Für Anfragen, die aufgrund von Störungen der Netzwerkverbindung fehlgeschlagen sind, gibt es eine integrierte Retry Logik, die bei Aktivierung fehlgeschlagene Anfragen erneut versucht. Zusätzlich lassen sich Anfragen mit einem Timeout versehen, um diese nach einer gewissen Zeit als fehlgeschlagen zu betrachten.

5.5.2 Reife

Um eine Funktionsfähigkeit über lange Zeiträume zu gewährleisten, dürfen keine unerwarteten Fehler in der API auftreten. Um dies zu erfüllen, muss das verwendete Framework ausreichend dokumentiert und Entwickler über Best Practices aufgeklärt sein. Die Cloud Native Computing Foundation betreut beispielsweise das gRPC Framework und bewertet die Reife und Stabilität mit dem Status `Incubating`, wobei dies der zweithöchste Status ist. Für REST und WebSocket finden sich viele Frameworks, wobei die Größe der Fangemeinde, sowie die Bekanntheit einem groben Schätzwert der Reife entspricht. Es muss jedoch angemerkt werden, dass die Größe einer Community und die Menge an Anleitungen keine exakten Werte liefern, um die Reife zu bewerten. Da aber keine unabhängige Organisation eine der

populären REST oder Websocket Frameworks auf ihre Reife geprüft haben, werden diese Metriken zur Bewertung verwendet. Das Django REST-Framework bietet eine gute Reife aufgrund der großen Community, die Tutorials veröffentlicht oder sogar Bücher über Best Practices schreibt. Ein bekanntes Framework für Websocket ist SocketIO, welches sich auch seit Jahren etabliert hat.

5.5.3 Verfügbarkeit

Mit dem Behandeln von Fehlern und dem Schutz vor Angriffen, kann sichergestellt werden, dass eine API stets verfügbar ist. Treten dennoch Fehler auf, die sich auf die Erreichbarkeit auswirken, lassen sich die meisten APIs mittels Load-Balancing skalieren, um weiterhin Verfügbar zu sein. Das Konzept des Load-Balancings besteht darin, dass Clients Anfragen an einen Server senden, der als Load-Balancer bezeichnet wird. Der Load-Balancer verteilt anschließend die Anfrage auf einer Gruppe von Servern, welche jeweils die Funktionalität der API anbieten und leitet die Antwort an den Client weiter. Um den derzeitigen Stand der Auslastung eines Servers zu wissen, frägt der Load-Balancer diesen regelmäßig bei den Servern an. Da das Load-Balancing jediglich Datenpakete an APIs mit derselben Funktionalität weiterleitet, müssen Streaming-APIs zuvor einen Synchronisierungsmechanismus implementieren, um zu kommunizieren, welche Daten bereits übertragen wurden. Bei dieser Art von Load-Balancing handelt es sich um serverseitige Load-Balancer, die sich bei den meisten Cloud-Diensten einrichten lassen.

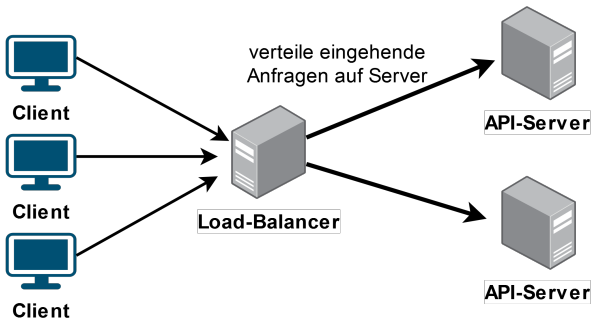

Abbildung 14: Das serverseitige Load-Balancing nach [31]

Dadurch, dass gRPC die Schnittstelle der Clients über die implementierbaren Stubs realisiert, bietet das Framework Implementierungen für clientseitiges Load-Balancing an. Der Unterschied zum serverseitigen Load-Balancing ist, dass der Client sich einen verfügbaren Server aus einer Gruppe aussucht und einen Bericht mit der aktuellen Belastung anfragt. Als Synchronisierungsmechanismus verwendet gRPC normalerweise einen konsistenten Hashing-Algorithmus, um sicherzustellen, dass alle Anfragen eines Clients an denselben Server gesendet werden. Somit wird bei clientseitigen Load-Balancing kein zusätzlicher Server benötigt, der die Rolle des Load-Balancers übernimmt, was durch das Entfernen des extra Hops eine höhere Performance bietet [31]. Die Nachteile des clientseitigen Load-Balancings sind eine höhere Komplexität des Clients durch das Konfigurieren einer Load-Balancing-Policy und dem nötigen Vertrauen in den Client, dass dieser nicht die Informationen der aktuellen Belastung dafür verwendet um die Erreichbarkeit von Servern zu stören.

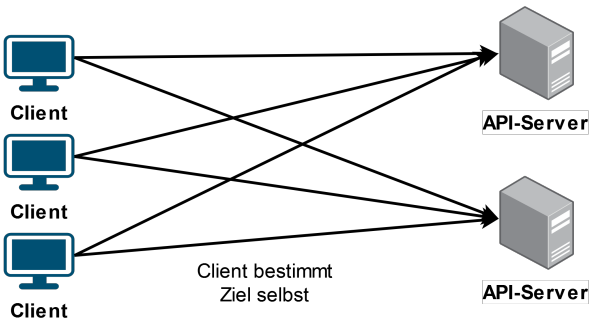

Abbildung 15: Das clientseitige Load-Balancing nach [31]

Eine Protokoll gRPCs zur Lösung für das Stören von bereits belastetet Servern ist das Lookaside-Load-Balancing, bei dem die Clients nicht mehr die Server direkt anfragen, sondern einen Lookaside-Load-Balancer, der den anfragenden Clients mit einer Liste mit mindestens einem gering belasteten Server antwortet. Das Konzept der Lookaside-Load-Balancer übertragt sich auch auf REST und Websocket basierte APIs, wobei die Client-Implementierungen komplexer sind als bei gRPC. Vorteile des Lookaside-Load-Balancings gegenüber dem serverseitigen Load-Balancings sind wie beim clientseitigen auch der reduzierte Netzwerkverkehr und eine bessere Flexibilität, da jeder Client seinen eigenen Algorithmus zur Auswahl des Servers anwenden kann.

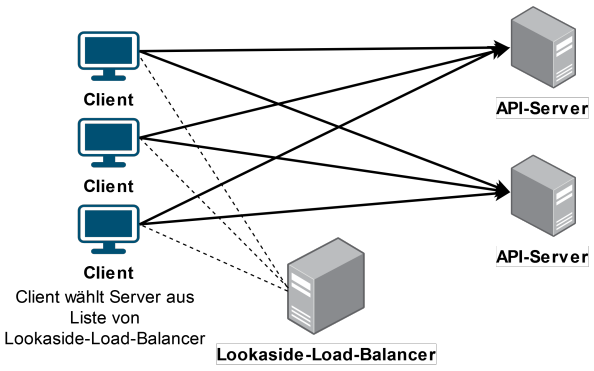

Abbildung 16: Das Lookaside-Load-Balancing nach [31]

5.6 Sicherheit

Die Sicherheit von Web APIs bezieht sich auf Aspekte wie die Autorisierung von Zugriffen auf Ressourcen, die Authentifizierung von Benutzern oder die Verschlüsselung von Datenübertragungen.
Des Weiteren gibt es auch Angriffe wie das Parameter Tampering, bei dem ein bösartiger Nutzer URL Parameter oder Daten der Payload vor dem Absenden manipuliert. Gibt es beispielsweise bei der Canvas API den Endpunkt `newCanvas`, welcher eine Sitzungsnummer zurückgibt und den Endpunkt `session/id=<Sitzungsnummer>`, mit dem das Canvas der Sitzungsnummer aufgerufen wird, könnte ein Angreifer die Canvas anderer Nutzer durch Ausprobieren von zufälligen Sitzungsnummern aufrufen.
Das STRIDE-Modell [32] ist eine Methode zur Identifizierung potenzieller Sicherheitsrisiken bei Web APIs und unterteilt diese in sechs Kategorien.

Risiko	Kurzbeschreibung
Spoofing	Vortäuschen einer anderen Identität.
Tampering	Manipulieren von Daten oder Informationen.
Repudiation	Die Abstreitbarkeit einer Aktion. Ein Nutzer führt eine Aktion aus und behauptet später diese nicht ausgeführt zu haben.
Information disclosure	Unautorisiertes Veröffentlichen von sensiblen Daten.
Denial of service	Ein absichtlicher Versuch einen Dienst zu blockieren.
Elevation of privilege	Die Erhöhung von Zugriffsrechten oder Privilegien um auf Funktionen zuzugreifen, auf die ein Benutzer normalerweise keinen Zugriff hat.

Tabelle 3: Die Kategorien des STRIDE-Modells mit Kurzbeschreibung

Die Maßnahmen für gRPC-, WebSocket- und REST-basierte APIs, um die Risiken abzudecken, werden in den folgenden Unterkapiteln erläutert.

5.6.1 Autorisierung, Authentifizierung und Verschlüsselung

Die Autorisierung stellt sicher, dass Nutzer nur auf die Ressourcen Zugriff haben, für die sie auch berechtigt sind. Die Authentifizierung dient zur Überprüfung, ob ein Benutzer tatsächlich der ist, der er vorgibt zu sein. Damit Unbefugte den Datenaustausch nicht mitlesen oder möglicherweise manipulieren, sollte der Datenverkehr verschlüsselt stattfinden.
Ein Weg, die Autorisierung, Authentifizierung und Verschlüsselung zu erfüllen ist das Transport Layer Security Protokoll [33], kurz TLS genannt. TLS wird manchmal auch SSL/TLS genannt, da SSL der Vorgänger von TLS ist. Das Protokoll beginnt, nachdem eine erfolgreiche TCP-Verbindung aufgebaut wurde, mit der Client-Hello-Nachricht. Die Client-Hello-Nachricht besteht beinhaltet eine Liste der unterstützten kryptografischen Algorithmen zum Verschlüsseln von Nachrichten. Wenn der Server einen der Algorithmen unterstützt, wählt er diesen in seiner Server-Hello Antwort. Daraufhin sendet der Server seinen öffentlichen Schlüssel und ein TLS-Zertifikat, mit dem der Client die Authentizität des Servers überprüfen kann. Der Client berechnet anschließend einen Sitzungsschlüssel, verschlüsselt diesen mit dem öffentlichen Schlüssel des Servers und sendet ihn an den Server. Nun besitzen Client und Server den Sitzungsschlüssel und können Nachrichten mit diesem verschlüsseln und austauschen.

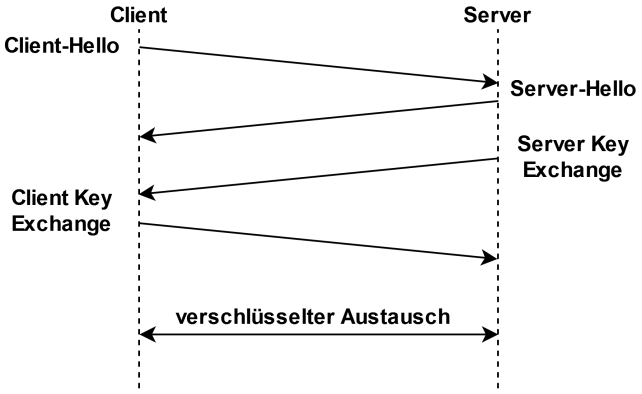

Abbildung 17: Der TLS-Handshake nach [33]

Um die Identität des Servers zu bestätigen und den Datenverkehr zu verschlüsseln, implementiert das gRPC Framework TLS und nutzt dies auch standardmäßig. Optional können Clients dem Server auch ein TLS-Zertifikat senden, um ihre Identität zu verifizieren. Ist die Verbindung bereits verschlüsselt kann die Autorisierung des Clients auch über das hinzufügen von Anmeldeinformationen oder einem Zugriffstoken wie OAuth2 zu den Metadaten der Nachricht erfolgen. Beim Empfangen der Nachricht wertet der Server die Metadaten aus und überprüft somit die Identität des Clients.

REST-basierte Web APIs nutzen oft das HTTP over SSL/TLS Protokoll, kurz HTTPS-Protokoll, um die Sicherheit zwischen Client und Server zu gewährleisten. Im Wesentlichen ist HTTPS eine verschlüsselte Version des HTTP-Protokolls, das TLS zur Verschlüsselung verwendet. Die meisten REST-API-Frameworks bieten Unterstützung für HTTPS-Verschlüsselung. Eine Autorisierung kann des Weiteren wie bei gRPC über Tokens oder dem OAuth2 Protokoll stattfinden.
Ähnlich wie bei REST erfolgt die Authentifizierung von WebSocket-basierten Web APIs durch die Überprüfung von Anmeldedaten, Tokens oder TLS. Der Unterschied zu REST besteht jedoch darin, dass bei WebSockets eine dauerhafte Verbindung zwischen Client und Server besteht und somit der Server den Authentifizierungsstatus des Clients beibehält. Der gängigste Weg WebSocket-Verbindungen zu verschlüsseln ist TLS, das von den meisten WebSocket-Bibliotheken und Frameworks unterstützt wird.

5.6.2 Rate-limiting

Rate-limiting ist das Begrenzen der Anzahl von Anfragen, die innerhalb eines bestimmten Zeitraums von einem Client gemacht werden können. Erreicht der Client das Limit, so wird dieser für eine gewisse Zeit blockiert, um eine Überlastung der API zu verhindern.
Das gRPC Framework bietet zum Begrenzen die Möglichkeit, einen Rate-Limiting-Interceptoren zu konfigurieren, welcher zu Beginn jedes eingehenden RPCs überprüft, ob die definierte Anzahl von Anfragen pro Sekunde überstiegen wurde. Ist dies der Fall, erhält der Client den Status-Code `RESOURCE_EXHAUSTED`. Alternativ könnte die Rate-Limiting Funktion in einer gRPC-Load-Balancer-Implementierung umgesetzt werden [31], welcher voraussetzt, dass mindestens zwei gRPC-Server existieren. Bei eingehendem RPC leitet der Load Balancer je nach Scheduling-Verfahren die Anfrage an einen der Server weiter.
WebSockets bieten im Gegensatz zu gRPC keine Rate-Limiting Funktion. Grund dafür ist die bidirektionale persistente Verbindung, über die kontinuierlich Daten ausgetauscht werden. Durch die Einführung eines Rate-Limits könnte die Echtzeitfähigkeit von WebSockets nicht mehr dauerhaft gegeben sein. Trotzdessen lässt sich eine Rate-limiting Funktionen Implementieren, indem der Code der WebSocket-API mithilfe eines Zählers und Zeitstempeln erweitert wird. Für jede Anfrage wird ein Zeitstempel gespeichert und der Zähler hochgezählt. Ist ein gespeicherter Zeitstempel älter als die Mindestwartezeit, wird er gelöscht und der Zähler runtergezählt. Ist die maximale Anzahl von Anfragen im Zeitraum erreicht, wird die einkommende Nachricht nicht bearbeitet. Für REST-basierte APIs gibt es für die meisten Frameworks Erweiterungen, welche die Rate-limiting Funktion implementierten. Die Erweiterung der FastAPI namens `fastapi-limiter` ermöglicht es, Rate-Limits auf Basis der IP-Adresse oder der Benutzer-ID zu erstellen. Auch hier ist es möglich, das Rate-Limiting selbst zu implementieren, falls es keine Erweiterung für das genutzte Framework gibt.

5.7 Wartbarkeit

Die Unterkapitel der Wartbarkeit befassen sich mit Maßnahmen, um die langfrisige Funktionalität der Software sicherzustellen. Die Teilargumente Analysierbarkeit, Modifizierbarkeit, Modularität und Wiederverwendbarkeit werden aufgrund des beschränkten Zeitraums nicht ausgewertet.

5.7.1 Testbarkeit

Das Testen von Software stellt sicher, dass sie zuverlässig und robust funktioniert, aber besonders bei Web APIs kann dies zeitaufwändig sein. Um das korrekte Verhalten einer API sicherzustellen, müssen nicht nur die Endpunkte durch Anfragen überprüft werden, sondern oft müssen erst die richtigen Daten für interne Status oder Datenbanken eingerichtet sein. Interagiert die API mit weiteren Services, so muss der Tester diese zusätzlich noch nachahmen. Um diesen Aufwand zu reduzieren, wurden für REST-basierte Web APIs in [34] testbare Metriken aufgelistet, Verfahren zum Testen erläutert und beschrieben, welche Tools dabei helfen können.
Nach [34] lassens sich die Metriken einer REST-API in die Gruppen Coverage, Fault Detection und Performance einteilen. Coverage umfasst die Abdeckung von Code und Schemas. Der häufigste Weg Code abzudecken ist über das Überprüfen einzelner Zeilen und Anweisungen. Beim Schema der REST-Api sind der HTTP-Statuscode, der Pfad von Endpunkten, die HTTP-Methoden, die Anfragedaten und die Antwort testbar.
Fault Detection beschreibt das Testen von Fehlerfällen der API. Der HTTP-Statuscode 500 weist meist auf einen internen Fehler hin. Manche internen Fehler der API, wie der fehlgeschlagene Aufruf einer Datenbank, sind testbar. Zusätzlich lassen sich festgelegte Regeln testen, wie die Idempotenz der HTTP-Methode DELETE. Das mehrfache Ausführen der Methode sollte zum gleichen Ergebnis resultieren wie ein einmaliges Ausführen.
Für APIs mit erwarterter Antwortzeit gibt es Performance Tests, um die RTT oder die Latenz auszuwerten.
Zu einem der meistverbreiteten Tests, die sich automatisieren lassen, gehört der Systemtest, welcher als Ziel hat, alle Anforderungen an eine API vollständig zu testen. Systemtests werden meist auf einer Umgebung durchgeführt, die den produktiven Gebrauch nachstellt, um die Fertigstellung der API zu gewährleisten.
Das offizielle Testframework gRPC Test `pytestgrpc` bietet die Möglichkeit Tests zu schreiben, um die Funktionweise sicherzustellen. gRPC Test ist jedoch noch nicht so weit entwickelt, weshalb mit dem Testframework alle Tests selbstständig entwickelt werden müssen. Des weiteren unterstützt Postman als manuelle Testmöglichkeit seit dem Januar 2023 [35] gRPC. Anders ist es bei dem Websocket Testframework Autobahn, welches über 500 Testfälle bietet.
Insgesamt lässt sich zusammenfassen, dass alle drei APIs Testmöglichkeiten anbieten, wobei für gRPC noch empfohlene Best Practices im Bereich Testbarkeit fehlen.

6 Fazit und Ausblick

6.1 Fazit

Insgesamt zeigt der Vergleich von gRPC, REST und WebSocket als mögliche Technologien für Web-APIs, dass jedes der Kommunikationskonzepte ihre eigenen Vor- und Nachteile hat. Vorteile wie die Plattformunabhängigkeit, eine einfache Implementierung und die weite Verbreitung, sind eine der Hauptgründe, warum Entwickler sich für eine REST-API entscheiden. Müssen Daten aber schnell übertragen werden und können verschiedene Formate annehmen, so punkten WebSockets mit einer effizienten Echtzeit-Kommunikation.
Im Vergleich der drei Technologien zeigt sich jedoch, dass gRPC in gewissen Aspekten Vorteile gegenüber REST und WebSocket bietet, insbesondere in Bezug auf die Aspekte der Korrektheit, Zuverlässigkeit und Sicherheit. Darüber hinaus bietet gRPC durch die Schnittstellendefinition mittels Protobuf eine entwicklerfreundliche API-Definition mit Unterstützung für vieler Plattformen und Programmiersprachen.
Allerdings erfordert gRPC eine stärkere Einarbeitung und benötigt mehr Aufwand in der Entwicklung als REST oder WebSocket. Es ist auch nicht für alle Anwendungsfälle geeignet, da ein Client-Zugriff erst erfolgen kann, sobald die neuste Version des Client-Stubs implementiert ist.

6.2 Ausblick

Ein Ausblick auf die zukünftige Entwicklung zeigt, dass gRPC weiter an Bedeutung gewinnen wird. Zwei große Schritte, die viel Aufmerksamkeit auf gRPC richten werden, sind zum einen die Integration der Browser-Unterstützung`grpc-web` in die offizielle gRPC-Bibliothek und das Aufnehmen aus dem `Incubating` in den `Graduated`-Status. Es ist auch zu erwarten, dass weitere Verbesserungen und Erweiterungen für gRPC veröffentlicht werden, um die Unterstützung für verschiedene Plattformen und Programmiersprachen zu erweitern.
Dennoch ist zu erwarten, dass REST und Websocket auch weiterhin wichtige Rollen im Bereich der Web-APIs spielen werden, insbesondere für Anwendungen mit geringerer Entwicklungszeit oder für Anwendungen, die eine höhere Flexibilität oder Einfachheit erfordern. Insgesamt kann gRPC als eine vielversprechende Alternative zu REST und Websocket betrachtet werden, die jedoch sorgfältig auf ihre Eignung für den jeweiligen Anwendungsfall geprüft werden sollte.

Literatur

[1] Wendell Santos. Which api types and architectural styles are most used? https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26, 2017. [Online; accessed: Sep. 07th, 2022]. 5

[2] Marek Bolanowski, Kamil Żak, Andrzej Paszkiewicz, Maria Ganzha, Marcin Paprzycki, Piotr Sowiński, Ignacio Lacalle, and Carlos E Palau. Eficiency of rest and grpc realizing communication tasks in microservice-based ecosystems. *arXiv preprint arXiv:2208.00682*, 2022. 6

[3] Florian Gerlinghoff. Vergleich von introspected rest mit alternativen ansätzen für die entwicklung von web-apis hinsichtlich performance, evolvierbarkeit und komplexität. *Qucosa urn:nbn:de:bsz:l189-qucosa2-748915*, 2020. 6

[4] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017. 7

[5] Introduction to grpc. https://grpc.io/docs/what-is-grpc/introduction/. [Online; accessed: Feb. 15th, 2023]. 7

[6] Oracle. Stubs and skeletons. https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmi-arch2.html, 2010. [Online; accessed: Oct. 26th, 2022]. 7

[7] Interface definition language (idl). https://learn.microsoft.com/de-de/dotnet/architecture/grpc-for-wcf-developers/interface-definition-language, 2022. [Online; accessed: Oct. 26th, 2022]. 8

[8] Raheel Masood. Speaking http2. https://raheelkhan.github.io/2018/05/04/speaking-http2/, 2018. [Online; accessed: Dez. 3rd, 2022]. 9

[9] Report: State of the web. https://httparchive.org/reports/state-of-the-web, 2022. [Online; accessed: Dez. 3rd, 2022]. 9

[10] Http/2 und die hpack komprimierung mit static huffman. https://www.youtube.com/watch?v=aPQjko8uDGk, 2018. [Online; accessed: Dez. 3rd, 2022]. 9

[11] Richard Belleville. grpc python quick start. https://grpc.io/docs/languages/python/quickstart/, 2022. [Online; accessed: Dez. 3rd, 2022]. 10

[12] IBM Cloud Education. Rest-designprinzipien. https://www.ibm.com/de-de/cloud/learn/rest-apis#toc-rest-desig-OOvO-7Sb, 2021. [Online; accessed: Nov. 12th, 2022]. 13

[13] Rfc 6455. https://www.rfc-editor.org/rfc/rfc6455.html, 2011. [Online; accessed: Nov. 14th, 2022]. 14

[14] Websockets demystified, part 1: Understanding the protocol. https://levelup.gitconnected.com/websockets-demystified-part-1-understanding-the-protocol-fccca2ca75eb, 2021. [Online; accessed: Dez. 3rd, 2022]. 14

[15] Iso/iec 9126. https://de.wikipedia.org/wiki/ISO/IEC_9126. [Online; accessed: Dez. 3rd, 2022]. 15

[16] Mccall's quality model. https://www.geeksforgeeks.org/mccalls-quality-model/?ref=lbp. [Online; accessed: Mar. 05th, 2023]. 15

[17] Boehm's software quality model. https://www.geeksforgeeks.org/boehms-software-quality-model/. [Online; accessed: Mar. 05th, 2023]. 15

[18] Iso/iec 25010. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010. [Online; accessed: Feb. 15th, 2023]. 15, 42

[19] Roger S Pressman. *Software engineering: a practitioner's approach*. Palgrave macmillan, 2005. 16

[20] Miro - the leading visual collaboration platform. https://miro.com/about/. [Online; accessed: Apr. 01st, 2023]. 17

[21] Status codes and their use in grpc. https://grpc.github.io/grpc/core/md_doc_statuscodes.html. [Online; accessed: Apr. 02nd, 2023]. 23

[22] Mark Masse. *REST API design rulebook: designing consistent RESTful web service interfaces*. Ö'Reilly Media, Inc.", 2011. 23

[23] Kevin R Fall and W Richard Stevens. *TCP/IP illustrated, volume 1: The protocols*. addison-Wesley, 2011. 26

[24] Ilya Grigorik. *High Performance Browser Networking*. O'Reilly, 2013. 28

[25] Sandeep Verma. Http/1 to http/2 to http/3. https://medium.com/@sandeep4.verma/http-1-to-http-2-to-http-3-647e73df67a8, 2019. [Online; accessed: Feb. 15th, 2023]. 28

[26] What is openapi? https://swagger.io/docs/specification/about/. [Online; accessed: Apr. 01st, 2023]. 29

[27] Interoperability test case descriptions. https://github.com/grpc/grpc/blob/master/doc/interop-test-descriptions.md. [Online; accessed: Apr. 01st, 2023]. 29

[28] Autobahn—testsuite. https://swagger.io/solutions/api-documentation/. [Online; accessed: Apr. 01st, 2023]. 29

[29] Restful web api design. https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design#use-standard-protocols-and-media-types. 29

[30] Http response status codes. https://developer.mozilla.org/en-US/docs/Web/HTTP/Status. [Online; accessed: Feb. 23th, 2023]. 31

[31] grpc load balancing. https://grpc.io/blog/grpc-load-balancing/. [Online; accessed: Mar. 05th, 2023]. 32, 33, 36

[32] Neil Madden. *API Security in Action*. O'Reilly, 2021. 34

[33] The transport layer security (tls) protocol version 1.3. https://www.rfc-editor.org/rfc/rfc8446. [Online; accessed: Mar. 05th, 2023]. 35

[34] Amid Golmohammadi, Man Zhang, and Andrea Arcuri. Testing restful apis: A survey, 2022. 37

[35] Postman now supports grpc. https://blog.postman.com/postman-now-supports-grpc/. [Online; accessed: Apr. 01st, 2023]. 37

Abbildungsverzeichnis

Tabellenverzeichnis

Code Listings

Codelistings

```
# -*- coding: utf-8 -*-
# Generated by the protocol buffer compiler.  DO NOT EDIT!
# source: echo_example.proto
"""Generated protocol buffer code."""
from google.protobuf.internal import builder as _builder
from google.protobuf import descriptor as _descriptor
from google.protobuf import descriptor_pool as _descriptor_pool
from google.protobuf import symbol_database as _symbol_database
# @@protoc_insertion_point(imports)

_sym_db = _symbol_database.Default()




DESCRIPTOR = _descriptor_pool.Default().AddSerializedFile(b'\n\x12\x65\
    x63ho_example.proto\x12\x04\x65\x63ho\"\x1d\n\x0b\x45\x63hoRequest\x12\x0e
    \n\x06number\x18\x01 \x01(\x03\"\x1c\n\tEchoReply\x12\x0f\n\x07message\x18
    \x01 \x01(\x03\x32\x38\n\x04\x45\x63ho\x12\x30\n\x08SendEcho\x12\x11.echo.
    EchoRequest\x1a\x0f.echo.EchoReply\"\x00\x62\x06proto3')

_builder.BuildMessageAndEnumDescriptors(DESCRIPTOR, globals())
_builder.BuildTopDescriptorsAndMessages(DESCRIPTOR, 'echo_example_pb2',
    globals())
if _descriptor._USE_C_DESCRIPTORS == False:

  DESCRIPTOR._options = None
  _ECHOREQUEST._serialized_start=28
  _ECHOREQUEST._serialized_end=57
  _ECHOREPLY._serialized_start=59
  _ECHOREPLY._serialized_end=87
  _ECHO._serialized_start=89
  _ECHO._serialized_end=145
# @@protoc_insertion_point(module_scope)
```

Code Listing 1: echo_example_pb2.py: Code um Protobuf Nachrichten zu erstellen und in Binärdaten zu serialisieren

```
# Generated by the gRPC Python protocol compiler plugin. DO NOT EDIT!
"""Client and server classes corresponding to protobuf-defined services."""
import grpc

import echo_example_pb2 as echo__example__pb2


class EchoStub(object):
    """Interface exported by the Server
    """

    def __init__(self, channel):
        """Constructor.

        Args:
            channel: A grpc.Channel.
        """
        self.SendEcho = channel.unary_unary(
                '/echo.Echo/SendEcho',
                request_serializer=echo__example__pb2.EchoRequest.SerializeToString,
                response_deserializer=echo__example__pb2.EchoReply.FromString,
                )


class EchoServicer(object):
    """Interface exported by the Server
    """

    def SendEcho(self, request, context):
        """Missing associated documentation comment in .proto file."""
        context.set_code(grpc.StatusCode.UNIMPLEMENTED)
        context.set_details('Method not implemented!')
        raise NotImplementedError('Method not implemented!')


def add_EchoServicer_to_server(servicer, server):
    rpc_method_handlers = {
            'SendEcho': grpc.unary_unary_rpc_method_handler(
                    servicer.SendEcho,
                    request_deserializer=echo__example__pb2.EchoRequest.FromString,
                    response_serializer=echo__example__pb2.EchoReply.SerializeToString,
            ),
    }
    generic_handler = grpc.method_handlers_generic_handler(
            'echo.Echo', rpc_method_handlers)
    server.add_generic_rpc_handlers((generic_handler,))


```

```
49  # This class is part of an EXPERIMENTAL API.
50 class Echo(object):
51     """Interface exported by the Server
52     """
53 
54     @staticmethod
55     def SendEcho(request,
56             target,
57             options=(),
58             channel_credentials=None,
59             call_credentials=None,
60             insecure=False,
61             compression=None,
62             wait_for_ready=None,
63             timeout=None,
64             metadata=None):
65         return grpc.experimental.unary_unary(request, target, '/echo.Echo/
    SendEcho',
66             echo__example__pb2.EchoRequest.SerializeToString,
67             echo__example__pb2.EchoReply.FromString,
68             options, channel_credentials,
69             insecure, call_credentials, compression, wait_for_ready, timeout,
     metadata)
```

Code Listing 2: echo_example_pb2_grpc.py: Code um Objekte vom Typ Client- oder Server-Stub zu erstellen.

```python
from concurrent import futures
import grpc
import canvas_client_server_pb2
import canvas_client_server_pb2_grpc

drawn_lines = []
client_log = {}

class CanvasSync(canvas_client_server_pb2_grpc.CanvasSyncServicer):
    def getServerCanvas(self, request, context):
        for line in drawn_lines:
            yield canvas_client_server_pb2.Canvas(x=line["x"], y=line["y"], color=line["color"], width=line["width"])

    def updateServerCanvas(self, request_iterator, context):
        metadict = dict(context.invocation_metadata())
        clientid = metadict['clientid']
        if not client_log.get(clientid, None):
            client_log[clientid] = []
        for request in request_iterator:
            add_canvas_line_to_list(request)
        for line in drawn_lines:
            if not line in client_log[clientid]:
                client_log[clientid].append(line)
                yield canvas_client_server_pb2.Canvas(x=line["x"], y=line["y"], color=line["color"], width=line["width"])

def add_canvas_line_to_list(line):
    line_dict = {
            "x": line.x,
            "y": line.y,
            "color": line.color,
            "width": line.width
    }
    drawn_lines.append(line_dict)

if __name__=="__main__":
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=4))
    canvas_client_server_pb2_grpc.add_CanvasSyncServicer_to_server(CanvasSync(), server)
    server.add_insecure_port('192.168.0.241:50051')
    print("Starting gRPC Server")
    server.start()
    server.wait_for_termination()
```

Code Listing 3: grpc_server.py: Der gRPCbasierte Server.

```python
from fastapi import FastAPI
from pydantic import BaseModel


app = FastAPI()

class Line(BaseModel):
    x: list = []
    y: list = []
    color: str
    width: int
    clientId: str
client_log = {}
lines = []
clientId = ""
@app.get("/canvas/lines")
async def get_lines():
    return [i for i in lines if i not in client_log[clientId]]

@app.post("/canvas/lines", status_code=201)
async def add_line(line: Line):
    clientId = line.clientId
    if not client_log.get(clientId, None):
        client_log[clientId] = []
    if not line in client_log[clientId]:
        client_log[clientId].append(line)
    lines.append(line)
    return line
```

Code Listing 4: rest_server.py: Der Code der RESTAPI

```python
#!/usr/bin/env python

import asyncio
import websockets
import json

lines = []
client_log = {}
async def sync_canvas(websocket):
    async for message in websocket:
        line = json.loads(message)
        clientId = line.get("clientId", None)
        if not client_log.get(clientId, None):
            client_log[clientId] = []
        line.pop('clientId', None)
        if not line in client_log[clientId]:
            client_log[clientId].append(line)
        if line.get("width", 0) and line.get("x", []) and line.get("y", [])
    and line not in lines:
            lines.append(line)
        await websocket.send(json.dumps([i for i in lines if i not in
    client_log[clientId]]))
```

```
21
22 async def main():
23     while True:
24         try:
25             async with websockets.serve(sync_canvas, "192.168.0.241", 50051):
26                 await asyncio.Future()
27         except Exception as e:
28             print(e)
29
30 if __name__ == "__main__":
31     asyncio.run(main())
```

Code Listing 5: websocket_server.py: Die WebsocketAPI