

BACHELOR-THESIS: VERGLEICH VON KOMMUNIKATIONSTECHNOLOGIEN /-PROTOKOLLEN IN MICROSERVICE- LANDSCHAFTEN

BACHELOR-THESIS: Zur Erlangung des
akademischen Grades Bachelor of Science
(B.Sc.)Informatik

Betreuer:
Prof. Dr. Mäkiö (FH Emden)
Dipl.-Wirt.-Ing. Markus Zondler (Conemis)

Benjamin Leonhardt
Matrikel-Nr.: 7004553
benjaminleonhardt@gmx.de

Inhaltsverzeichnis

1.	Einleitung.....	2
1.1.	Motivation.....	2
1.2.	Ziel und Aufbau der Arbeit	4
2.	Grundlagen.....	5
2.1.	Was sind Microservices?	5
2.1.1.	Microservice vs. Monolith.....	6
2.1.2.	Abgrenzung zu SOA	7
2.2.	HTTP	8
2.2.1.	HTTP-Request	9
2.2.2.	HTTP-Response.....	11
2.2.3.	HTTP 1.1 vs. HTTP/2	12
2.3.	RPC und RMI.....	13
2.3.1.	Marshalling.....	14
2.3.2.	IDL – Interface Description Language.....	22
2.4.	REST - Representational State Transfer.....	27
2.4.1.	RESTful.....	27
2.4.2.	Jax-RS.....	28
2.5.	SOAP - Simple Object Access Protocol	30
2.5.1.	Jax-WS	31
2.5.2.	Skeleton erstellen	31
2.5.3.	Stubs aus WSDL-Dateien erstellen	31
2.6.	gRPC – gRPC Remote Procedure Call.....	32
2.6.1.	Skeleton aus der Proto-Datei erstellen	32
2.6.2.	Stub-Generierung mit Maven.....	34
3.	Verwandte Arbeiten.....	35
4.	Architektur des Benchmark-Systems	36
5.	Ergebnisse.....	39
6.	Fazit	59
	Abbildungsverzeichnis.....	60
	Listingverzeichnis.....	62
	Tabellenverzeichnis	63
	Verweise	64

1. Einleitung

Die zunehmende Größe und Komplexität von Software-Projekten stellt die Entwickler zunehmend vor Herausforderungen: Wie kann man große Projekte, die aus einem einzigen großen Programm bestehen (auch Monolith genannt), warten und erweitern? Wie können in monolithischen Systemen die Bindungen an veraltete Technologien und deren Aktualisierung vereinfacht werden? Je größer das Projekt wird, desto mehr Klassen und Abhängigkeiten zwischen diesen Klassen untereinander schleichen sich ein. Das macht einen Monolith auf Dauer unübersichtlich und schwer wartbar. Bei einem Monolith kann nur eine einzige Programmiersprache verwendet werden, daher lassen sich neue Features manchmal nur schwer oder schlecht umsetzen, weil zu Beginn eines Projekts eine bestimmte Programmiersprache gewählt wurde. Probleme könnten in anderen Programmiersprachen eventuell einfacher, besser oder performanter gelöst werden, je nach Problemfall. Die Verwendung einer einzigen Programmiersprache schränkt daher also die Lösbarkeit von Problemen stark ein. Ideal wäre es, jeden Vorteil jeder Programmiersprache flexibel nutzen zu können.

Der Trend zeigt, dass die Entwicklung von Projekten zunehmend in Richtung kleinere unabhängig entwickelbare Module geht [1]. Doch Modularisierung ist kein neues Konzept. Allerdings schleichen sich bei der Modularisierung innerhalb eines Monolith die oben genannten Probleme schnell ein. Der Microservice-Ansatz überzeugt dagegen in Sachen Modularisierung durch die komplette Abtrennung der Module mittels verteilter Kommunikation. Große Firmen wie Amazon oder Netflix haben diesen Vorteil für sich erkannt und ihre Systeme auf Microservice-Basis umgestellt.

Eine aktuelle Umfrage zeigt, dass etwa dreiviertel aller befragten Entwickler, entweder bereits mit Microservices arbeiten oder darauf umstellen möchten.

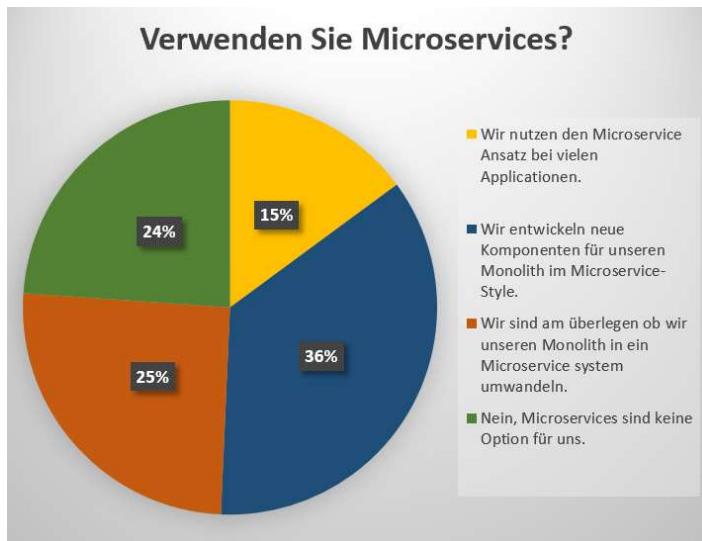


Abbildung 1: Umfrage, wie viele Entwickler den Microservice-Ansatz in Ihren Projekten verwenden

1.1. Motivation

Laut einer von der Seite JAXenter durchgeführten Umfrage zu den oben genannten Problemen bei monolithischen Programmen stimmen rund 60 % der befragten Entwickler zu mindestens 50 % zu, dass sie Probleme bei der Entwicklung von monolithischen Systemen haben (Abbildung 2) [1]. Auf der Suche nach einem Ansatz, diese Probleme zu mildern, sind erfahrene Entwickler dazu übergegangen,

kleine lose gekoppelte Programme zu schreiben. Diese sollen von kleinen Teams mit 6–8 Programmierern mühelos entwickelt, getestet, deployt und gewartet werden können.



Abbildung 2: Zustimmung zu Problemen bei der Entwicklung von monolithischen Programmen

Amazon begann etwa 2006 mit der Einführung von Microservices in der Amazon Cloud ([2] S. 1). Das darauf folgende positive Feedback, hat den Grundstein zur wachsenden Beliebtheit von Microservices gelegt.

Die Nutzung von Microservices bringt bemerkenswerte Vorteile mit sich:

- Bei einem Update eines Microservice, muss nur dieser neu kompiliert und deployt¹ werden. Alle anderen bleiben von der Änderung unberührt.
- Durch die Vorgabe von Schnittstellen werden die Microservices untereinander vollständig abgekoppelt. So entstehen weniger Abhängigkeiten als bei großen Monolithen, was die Wartbarkeit verbessert.
- Kleine Quellcodes helfen den Entwicklern, den Überblick über den Code zu behalten. Bugs lassen sich so schneller finden und einfacher beheben.
- Da die Microservices klein und nicht vom Gesamtsystem abhängig sind, wie bei einem Monolith, können Updates in kürzeren Intervallen deployt werden. So kann das Gesamtsystem agiler auf den Markt reagieren (Continuous Delivery²).
- Durch das Abkapseln der Module können Test parallel ablaufen, somit können alle Teams ihre Microservices gleichzeitig testen.
- Wenn ein Service ausfällt ist nur dieser betroffen. Der Rest des Systems läuft weiter.
- Mit Microservices können DevOps³ eingeführt werden, welche den Service entwickeln, testen, deployen und betreiben. So kennt der Verantwortliche bei einem Problem den ganzen Microservice. Bei Monolithen funktioniert das nur schlecht, weil diese zu groß sind.

¹ Webservice im Netzwerk verfügbar machen.

² Das ständige Versorgen mit Updates.

³ DevOps steht für Developer und IT Operation. Diese entwickeln einen Microservice und betreiben diesen auch.

Allerdings bringt die neue Technik nicht nur Vorteile mit sich. Mit einem der Probleme befasst sich diese Arbeit: Der Performance.

1.2. Ziel und Aufbau der Arbeit

Da Microservices die Kommunikation untereinander über das Netzwerk/Internet erledigen, wird die Bandbreite und Latenz zum Flaschenhals. Dies kann bei stark frequentierten Microservices zum Problem werden. Die Kommunikation wird über Protokolle erledigt, welche durch ihre Architektur maßgeblich Einfluss auf die Latenz und den Datendurchsatz haben.

Diese Arbeit stellt die Protokolle REST, SOAP und gRPC technisch vor und vergleicht deren Vor- und Nachteile miteinander. Die Protokolle haben bei der Übertragung von Objekten bzw. bei Aufrufen von entfernten Prozeduren (RPC – Remote Procedure Call) unterschiedliche Ansätze. Diese technischen Details sollen erklärt und anhand vergleichbarer Metriken gegenübergestellt werden. Dies soll Aufschluss darüber geben, warum ein Protokoll schneller ist und ob durch die Optimierungen andere Nachteile entstehen.

Als Programmiersprache wurde Java verwendet. Das System wird dabei in 3 Services aufgeteilt (Abbildung 3). Ein Persistenz-Service, der über eine SQL-Datenbank verfügt, einen Transformator-Service, der eine REST-Schnittstelle als Ein-/Ausgabeschnittstelle für den Benutzer besitzt, und einen UmdrehenMS-Service, der dazu geschalten werden kann. Vom Persistenz-Service werden Strings aus einer SQL-Datenbank gelesen und an den Transformator-Service gesendet. Dieser dreht entweder den String um und sendet ihn zum Persistenz-Service zurück oder er sendet ihn, falls dazu geschaltet, an den UmdrehenMS-Service weiter (Dies soll den Netzwerkverkehr für die Benchmark-Ergebnisse weiter erhöhen.). Wenn dazugeschalten, werden die Strings vom UmdrehenMS-Service gedreht und an den Transformator-Service wieder zurückgesendet. Der Transformator-Service sendet die gedrehten Strings dann, zum Speichern an den Persistenz-Service, welcher die Strings anschließend in die SQL-Datenbank schreibt.

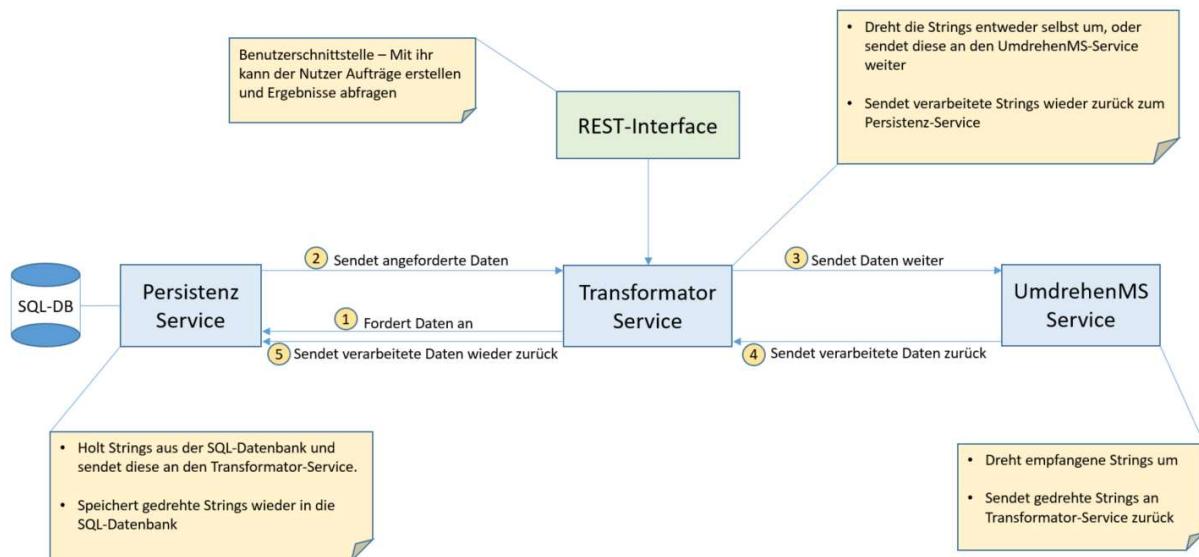


Abbildung 3: Aufbau des Projekts

Für die Implementierung des Benchmarks, wurde bei allen 3 Protokollen der Quellcode in jedem Service zusammen verschachtelt. Es wurden also in jedem Service der Code für REST, SOAP und gRPC zusammen gepackt. Dadurch sind die Quellcodes der „Microservices“ sehr groß (über 1000 Zeilen Code). Allerdings sind dadurch nur 3 Microservices statt 9 entstanden. Dies soll dem Leser, beim durchschauen des Quellcodes, das Gegenüberstellen der verschiedenen Implementierungen erleichtern.

In **Kapitel 2** werden die nötigen Verständnisgrundlagen geschaffen. Hierbei verlaufen die Themen vom Allgemeinen hin zum Konkreten. Erst wird der Begriff Microservice erläutert, danach wird alles Nötige erklärt, um die Protokolle und deren Unterschiede verstehen zu können.

In **Kapitel 3** werden ähnliche Arbeiten für weitere Literatur vorgestellt. Zudem sollen Fehler aus den verwandten Arbeiten vermieden werden und Grundlagen für die hier vorgestellte Arbeit geschaffen werden.

In **Kapitel 4** wird die Architektur des Benchmarksystems im Detail erklärt. Fragen wie, wo liegen die Messpunkte und wie werden die Ergebnisse berechnet, werden hier erklärt.

Kapitel 5 listet die Ergebnisse der Benchmark-Durchläufe auf. Zudem wird durch Review des in Kapitel 2.3.1 vorgestellten Beispielobjekts aus technischer Sicht erklärt, wieso die Testergebnisse so ausgefallen sind. Darüber hinaus wird eine Übersicht über die Stärken und Schwächen der einzelnen Protokolle gegeben.

Abschließend soll in **Kapitel 6** ein Fazit und einen Ausblick auf zukünftige Entwicklungen gegeben werden.

Konventionen

Für leichteres Lesen werden im Text besondere Stellen hervorgehoben: *Quelltexte* und *Befehle* werden immer in Consolas, kursiv, Größe 9 geschrieben, um die Listings der Quelltexte vom Fließtext abzuheben. Um die Leserlichkeit zu wahren, wurden lange Listings gekürzt. Gekürzte Stellen werden mit einem dreifach-Punkt, also dem Auslassungszeichen (...), dargestellt. Wenn aus einem Listing Passagen erklärt werden, sind diese Listingpassagen zur Verdeutlichung in einer eigenen Zeile und eingerücktem Text gekennzeichnet. Mathematische Formeln sind in Cambria Math Größe 11 geschrieben.

In dieser Arbeit werden die Begriffe Microservice, Service oder MS als synonym verwendet. Es ist in jedem Fall immer von einem Microservice die Rede. Sollte eine andere Art Service gemeint sein wird dies explizit erwähnt.

2. Grundlagen

2.1. Was sind Microservices?

Für den Begriff „Microservice“ gibt es keine offizielle, einheitliche Erklärung, allerdings kann man ihn folgendermaßen erklären: Microservices sind kleine eigenständig agierende Programme. Diese werden als Webservice betrieben und bekommen ihren Input per Netzwerk als HTTP-Anfrage (Kap. 2.2.1) und senden das Ergebnis auch wieder per Netzwerk als HTTP-Response (Kap. 2.2.2) zurück. Kommuniziert wird dabei per programmiersprachenunabhängigen Protokollen. Microservices werden ähnlich, wie bei der unter Linux bekannten Philosophie erstellt:

- „Ein Programm soll nur eine Aufgabe erledigen, und das soll es gut machen.“
- Programme sollen zusammenarbeiten können.
- Nutze eine universelle Schnittstelle. In Linux, sind das Textströme.“ ([2] S. 2). Bei Microservices sind diese Schnittstelle REST mit JSON, SOAP mit XML und gRPC mit Protobuf

Trotz des Namens ist die Größe eines Microservice nicht ausschlaggebend, ob dieser noch als Microservice anzusehen ist. Ziel von Microservices ist es, große Programme in kleine unabhängige Module zu teilen. Wie bereits in der Einleitung erwähnt, soll dies die Wartbarkeit gegenüber großen, komplexen, monolithischen Systemen verbessern. Das System, das programmiert werden soll, wird so aufgeteilt, dass die MS stets vollständig eigenständige Teile des Gesamtsystems darstellen, sodass jeder Service unabhängig von anderen entwickelt, getestet und deployed werden kann. Jedem Team wird immer nur ein Microservice zur Entwicklung und Wartung zugeteilt.

Die Microservices kommunizieren über programmiersprachenunabhängige Protokolle. Daher bietet es sich an, verschiedene Teams die an verschiedenen Problemstellen arbeiten, eine Technologiefreiheit zu geben. Das heißt, jedes Team darf selbst entscheiden, welche Programmiersprache es verwenden möchte. So kann ein Microservice in C++ geschrieben werden und ein anderer in Java. Da die Programmiersprachen zweier miteinander kommunizierender Microservices nicht einheitlich sein müssen, können die Vorteile jeder Sprache zum Lösen spezieller Probleme genutzt werden.

Durch das Deployen der Services auf Webservern und die losen Verbindungen über standardisierte Schnittstellen sind die Module direkt updatebar, ohne dass das ganze System heruntergefahren werden muss. Nur das betroffene Modul wird aktualisiert und steht sofort, mit der neueren Version, für alle anderen Microservices zur Verfügung.

2.1.1. Microservice vs. Monolith

Im Folgenden eine Auflistung der Vor- sowie Nachteile von Microservice-Programmierung abgebildet. Die Nachteile der Microservices sind gleichzeitig auch die Vorteile des monolithischen Ansatzes. Abbildung 4 verdeutlicht die Unterschiede beim Arbeiten mit Microservices im Vergleich zu einem monolithischen System (vgl. [2] S. 19).

<u>Vorteile</u>	<u>Nachteile</u>
<ul style="list-style-type: none"> • Strikte Trennung der Module verhindert Abhängigkeiten • Bessere Wartbarkeit • Bessere parallele Entwicklung durch unabhängige Module • Kleinere Teams verbessern Kommunikation • Schnelleres Deployen ermöglicht es, agiler auf den Markt zu reagieren (Continuous Delivery) • Bessere Skalierung. Services können doppelt deployt werden und Load-Balancer verteilen Last • Programmiersprachenunabhängig • Testen neuer Technologien ist einfacher und billiger 	<ul style="list-style-type: none"> • Komplexere Implementierung durch verteilte Kommunikation • Langsamer wegen Netzwerk-Latenz • Netzwerk-Pakete können evtl. verloren gehen • Erhöhte CPU-Last durch Marshalling und Unmarshalling (s. Kap. 2.3.1) • Änderungen an der Aufteilung der Services ist komplizierter

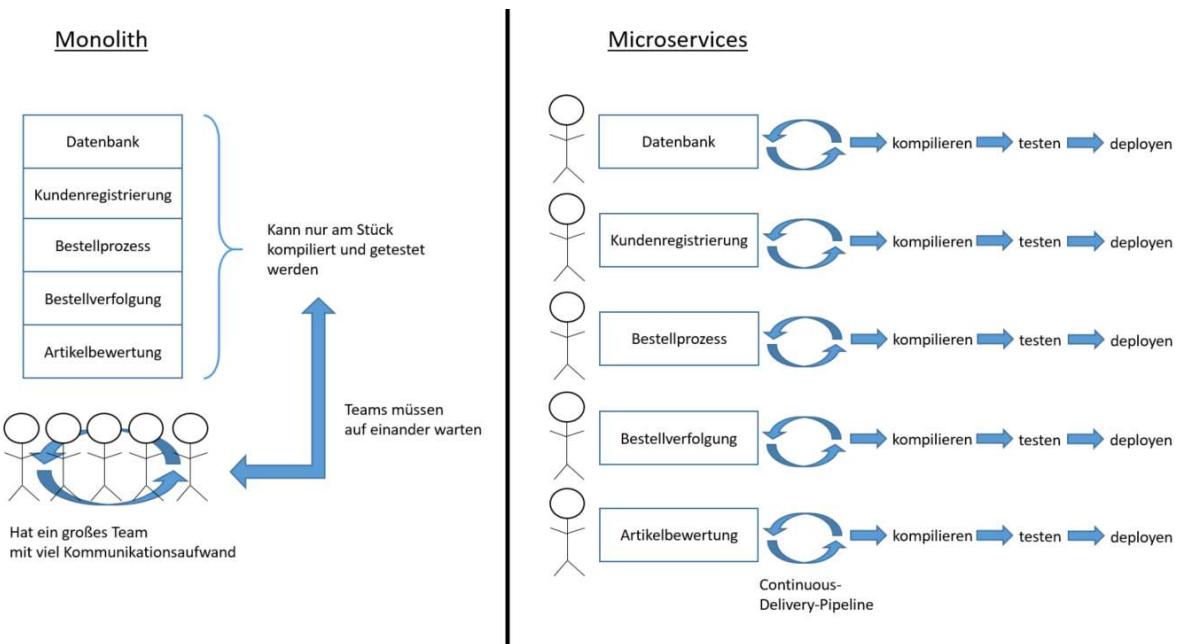


Abbildung 4: Monolith verglichen mit Microservices

Monolithische Systeme sind jedoch keineswegs überflüssig. Ob man Microservices einführen sollte, hängt ganz vom Einzelfall ab, und jeder muss für sich selbst abwägen, welches System besser für den gewünschten Gebrauch geeignet ist.

2.1.2. Abgrenzung zu SOA

Häufig fällt, wenn über Microservices gesprochen wird, auch der Begriff SOA. SOA steht für Service-Oriented Architecture. Wie der Begriff Microservice ist auch der Begriff Service-Oriented Architecture nicht einheitlich definiert. SOA ist anderer Ansatz, die Probleme die bei großen Monolithen auftreten zu lösen.

Bei SOA werden ebenfalls Services verwendet, um die Module vom Gesamtsystem zu lösen. Daher kommen hier auch dieselben Technologien zum Einsatz. Es kann also ein SOA-System mit den in dieser Arbeit verwendeten Protokollen implementiert werden. Bei SOA werden jedoch die Schnitte zwischen den Services anders gesetzt. So besitzt ein Service bei SOA keine eigene UI⁴. Ein Nutzer kommt nie direkt mit einem Service in Berührung.

Das UI wird beim SOA-Ansatz im sogenannten Portal implementiert. Kommt eine Benutzeranfrage über das Portal, reicht das Portal die Anfrage an einen Integrations- und Orchestrierungslayer weiter. Dieser verteilt die Anfrage dann an die Services. Darüber hinaus werden die Services im SOA-Ansatz nach Geschäftsprozess gruppiert. So ist es nötig bei der Einführung des SOA-Ansatzes die komplette IT auf SOA umzustellen, denn nur dann kann SOA all seine Vorteile entfalten. Die Entwicklerteams bekommen beim SOA-Ansatz eine komplette Gruppe zugewiesen, für die sie zuständig sind. Das

⁴ User Interface - Benutzeroberfläche

bedeutet, dass sie mehrere Services gleichzeitig pflegen müssen. Bei SOA kann ein Service nicht alleine geändert und deployed werden. Hier muss auch das Portal um das neue Feature erweitert werden und auch im Integrations- und Orchestrierungslayer müssen die Änderungen übernommen werden, damit andere Services diese nutzen können.

Microservices hingegen können ihre eigene UI besitzen. So kann ein Nutzer direkt mit einem MS kommunizieren. MS besitzen keinen Orchestrierungslayer, sondern organisieren sich selbst über ihre Schnittstellen. Da MS ihre eigene UI haben, können die Entwickler unabhängig von anderen Services ihr Update implementieren. Ein MS stellt immer einen eigenständigen Part des Gesamtsystems dar.

2.2. HTTP

HTTP steht für „Hypertext Transport Protokoll“ und ist ein Grundpfeiler des Internets. Die erste Version HTTP 1.0 wurde von Tim Berners-Lee 1991 offiziell vorgestellt. Die Definition ist in der RFC 1945 zu finden [3]. REST, SOAP und gRPC versenden ihre Nachrichten im Body von HTTP. HTTP ist ein Klartext-basiertes Protokoll, bei dem von einem Client an einen Server Anfragen (Request) gesendet werden, die im Klartext mitteilen, welche Information sie gerne hätten. Der Server sendet dann eine Antwort (Response) mit den gewünschten Informationen zurück (Abbildung 5).

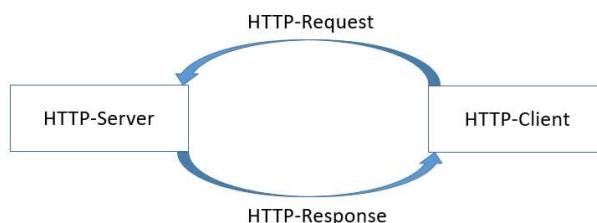


Abbildung 5: Funktionsweise von HTTP

Dabei gibt es für die Requests sogenannte HTTP-Verben, welche mitteilen, um was für eine Art der Anfrage es sich handelt. HTTP/2 ist zu HTTP 1.1 voll kompatibel (Kapitel 2.2.3), daher gelten alle Verben in beiden Versionen. Es gibt 9 Verben für HTTP. In dieser Arbeit sind nur 2 von Bedeutung, GET und POST.

Die Vollständigkeit halber folgenden nun alle Verben, die in der aktuellen Version in HTTP zur Verfügung stehen:

- | | |
|---------------|--|
| GET | – GET-Anfragen sind Anfragen wie sie ein Webbrower sendet. In der URI ⁵ der Anfrage wird alles an Information mitgesendet (Kap. 2.2.1). |
| HEAD | – Wie GET allerdings besitzt die Response keinen Body, sie besteht also nur aus dem Header. |
| PUT | – Dient zur Erstellung einer Ressource, sollte diese bereits existieren wird diese geändert. |
| POST | – POST-Anfragen eignen sich zum Übersenden von großen Datenmengen (Kap. 2.2.1). |
| DELETE | – Löscht eine Ressource. |
| OPTION | – Fragt nach verfügbaren Ressource eines Webservers. |

⁵ Universal Ressource Identifier – umfassen Adressen zu Ressourcen wie HTTP-URLs, E-Mail Adressen, FTP-Adressen und so weiter.

TRACE – Sendet einen Request zurück, um zu sehen, ob/wo sich bei zwischen Servern ändert.

CONNECT – Wandelt eine Verbindungs-Anfrage in eine transparente TCP/IP-Tunnel-Verbindung um.

PATCH – Ändert Teile einer Ressource.

2.2.1. HTTP-Request

GET ist eine Anfrage wie sie im Browser getätigt wird. Sie ist eine Anfrage nach einer Ressource. Sie ist idempotent, was bedeutet, dass egal, wie oft sie ausgeführt wird, sie stets zum gleichen Ergebnis führt. Alle Information wird bei einer GET-Anfrage in der URI übergeben.

Eine URI zu einer GET-Anfrage kann z. B. wie folgt aussehen:

`http://192.168.2.103:28080/microServices/webapi/Transformator/init?titel=test&bis=100000...`

Im Folgenden nun die Erklärung zu dieser URI. Dieser Link enthält alle Informationen, die gesendet werden. Beginnend mit dem

`http://`

welches dem Browser mitteilt, dass es sich um eine HTTP-Adresse handelt.

`192.168.2.103:28080`

ist in diesem Fall die IP-Adresse des eigenen Rechners inklusive Portnummer. Hier könnte auch beispielsweise „www.google.de“ stehen, wie man es vom Browsen im Internet gewohnt ist. Diese Adresse müsste dann noch von einem DNS-Server in eine IP-Adresse aufgelöst werden, damit die Anfrage ihren Weg zum Ziel auch finden kann. Der darauffolgende Teil:

`/microServices/webapi/Transformator/init`

ist die Angabe zur Ressource, die man haben möchte. In diesem Fall, da es sich um einen REST-Webservice handelt der angesprochen wird, entspricht das folgendem Schema: `/Projektname/Webservice_des_Projekts/Klassenname/Methodename`. In einer GET-Anfrage werden am Ende alle Variablen mit Namen hinter einem Fragezeichen an die URI angehängt, also:

`?titel=test&bis=100000`

Dabei kommen stets zuerst die Variablennamen, gefolgt von einem Gleichheitssymbol und dem Wert der Variable. Variablen werden mit einem Komma voneinander getrennt. Es ist also eine Key-Value-Liste. Der Browser wandelt diese URI dann, in die in Listing 1 dargestellte Form und sendet diese an den Server.

```
GET
/microServices/webapi/Transformator/init?titel=test&bis=1000000&stapelgroesse=10&protokoll=
3&parameter=2 HTTP/1.1
Accept: image/gif, image/jpeg, image/pjpeg, application/x-ms-application,
application/xaml+xml, application/x-ms-xbap, /*
Referer: http://192.168.2.103:28080/microServices/index.jsp
Accept-Language: de-DE
UA-CPU: AMD64
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0 (Windows NT 6.2; Win64; x64; Trident/7.0; rv:11.0) like Gecko
Host: 192.168.2.103:28080
Connection: Keep-Alive
Cookie: JSESSIONID=XiBCE1Ldow3FM0390H7DKmyQsdBpwRbpzjQLDWqs.desktop-21v7esd
```

Listing 1: HTTP-GET-Request

Eine GET-Anfrage ist von der Größe der Daten, die sie beinhalten darf, zwar nicht direkt beschränkt, allerdings wird im Hinblick auf die Abwärtskompatibilität empfohlen 255 Byte Länge nicht zu überschreiten. Darüber hinaus sind GET-Anfragen ASCII-codiert und können eventuell nicht alle Symbole korrekt übertragen. Es ist auch nicht möglich, durch einfache Aneinanderkettung von Key/Value-Paaren eine Datenstruktur darzustellen. GET-Anfragen eignen sich daher nicht für eine Datenübermittlung. Dort kommen POST-Anfragen zum Einsatz.

POST-Anfragen besitzen einen Body, in dem Platz für beliebig viele Daten ist. POST-Anfragen können allerdings nicht einfach in einen Browser eingegeben werden.

```
POST /microServices/webapi/persistenzBA/anfordernBatch?
    Fortschritt=MSBenchmark.microServices.FortschrittgRPC%
    4079f82965 HTTP/1.1
Accept: application/json; charset=UTF-8
Accept-Encoding: gzip, deflate
Content-Type: application/json; charset=UTF-8
Content-Length: 692
Host: localhost:28080
Connection: Keep-Alive

{
  "parameter":2,
  "protokoll":1,
  "stapel":[],
  "stapelGedreht":[],
  "alsStapel":true,
  "transformatorMS":true,
  ...
  "startZeit":1498059885693
}
```



POST-Header

POST-Body

Listing 2: HTTP-POST-Request

Die in Listing 2 dargestellte POST-Anfrage, zeigt oben den POST-Header. In diesem Beispiel befindet sich im Body ein Objekt der Klasse FortschrittgRPC, das mit dem Mediatype JSON (Kapitel 2.3.1.1 auf S.15 und 2.3.1.2 S. 16), in eine übertragbare Form gebracht wurde.

Kommen wir nun zu den einzelnen Teilen der in Listing 2 dargestellten POST-Anfrage. Zuerst kommt das Schlüsselwort „POST“, welches die POST-Anfrage einleitet. Der nach POST kommende Teil

```
/microServices/webapi/persistenzBA/anfordernBatch?Fortschritt=MSBenchmark.microServices.FortschrittgRPC%4079f82965
```

lässt sich ähnlich wie die URI in der GET-Anfrage aufschlüsseln. Der erste Teil:

```
/microServices/webapi/persistenzBA/anfordernBatch
```

entspricht demselben Schema wie oben: /Projektnname/Webservice_des_Projekts/Klassenname/Methodenname. Darauf folgt genauso wie bei GET ein Fragezeichen und anschließend der Variablennamen.

```
?Fortschritt
```

Nach dem Gleichheitszeichen kommt die Klasse mit Package-Namen.

```
=MSBenchmark.microServices.FortschrittgRPC
```

Das Prozentzeichen leitet die Referenznummer des Objekts ein.

%4079f82965

Diese Referenznummer ist keine Pointer-Adresse, sondern eine hexadezimale Zahl und dient der Identifizierung des Objekts, das versendet wird.

Im Listing 2 kommt nach der Referenznummer „HTTP 1.1“, welches die Version des HTTP-Protokolls angibt, also in diesem Fall Version 1.1. Dies gehört nicht mehr zur URI. Die restlichen Angaben haben folgende Bedeutung:

- Accept:** Listet alle für eine Response akzeptierten Mediatypen auf. Der akzeptierte Mediatype ist hier „*application/json*“ mit dem Zeichensatz „*UTF-8*“.
- Accept-Encoding:** Enthält Informationen dazu, welche Komprimierungsverfahren unterstützt werden.
- Content-Type:** Gibt dem Mediatype der Daten an, die sich im Body befinden.
- Content-Length:** Größe der Daten im Body in Byte.
- Host:** Domain-Name des Servers. Dies ist seit HTTP 1.1 zwingend nötig, da unter einer IP mehrere Domain-Namen zu erreichen sein können.
- Connection:** Informationen darüber, ob die Verbindung beibehalten werden soll, oder ob sie beendet werden kann.

Der im Body befindliche Teil wird dann dem Mediatype, der unter Content-Type angegeben wurde, interpretiert. Somit sind alle nötigen Informationen vorhanden, um die Daten im Body wieder korrekt zu einem Java-Objekt der Klasse FortschrittgRPC unmarshalen (Kap. 2.3.1) zu können.

2.2.2. HTTP-Response

Wie bereits anfangs des Kapitels erwähnt, erfolgt auf einen HTTP-Request stets eine HTTP-Response (vgl. Abbildung 5 S. 8). Diese enthält einen Statuscode, der dem Client mitteilt, ob die Anfrage korrekt verarbeitet werden konnte. Darüber hinaus, werden die angeforderten Daten im Body der Response mitgesendet. Hierzu muss ebenso wie beim Request, im Header der Response angegeben werden, um was für einen Mediatype es sich bei den Daten im Body der Response handelt. Da im Request evtl. mehrere Mediatypen unter „Accept“ angegeben wurden (Content Negotiation).

```
HTTP/1.1 200 OK
Connection: keep-alive
X-Powered-By: Undertow/1
Server: WildFly/10
Content-Length: 4
Content-Type: application/json; charset=UTF-8
Date: Wed, 21 Jun 2017 15:44:46 GMT
```

Listing 3: HTTP Response

In Listing 3 ist der Header einer HTTP-Response abgebildet. Dieser ist ähnlich wie der Request aufgebaut. Dabei wird zuerst das Protokoll angegeben „*HTTP/1.1*“, gefolgt vom Statuscode „*200 OK*“.

Statuscodes beginnend mit 200 bedeuten, dass alles in Ordnung ist. Die Statuscodes sind dabei wie folgt gegliedert:

- 1xx – Information
- 2xx – Anfrage erfolgreich bearbeitet
- 3xx – Weitere Schritte nötig, um Anfrage bearbeiten zu können
- 4xx – Client-Fehler
- 5xx – Server-Fehler

Der Rest der Response bedeutet Folgendes:

Connection: Gibt wie beim Request an, ob die Verbindung aufrechterhalten werden soll oder geschlossen werden kann.

X-Powered-By: Ist eine Information zur Technologie, die vom Server verwendet wird. Undertow ist ein Teil des JBoss Frameworks

Server: Gibt an, welche Art Server antwortet. Hier ist es ein JBoss Wildfly Server Version 10.

Content-Length: Bytegröße der Daten im Body.

Content-Type: Gibt den Mediatype der Daten im Body an.

Date: Das Datum, wann die Nachricht versandt wurde.

2.2.3. HTTP 1.1 vs. HTTP/2

HTTP 1.1 wurde 1999 in der Form, in der es auch aktuell verwendet wird, definiert. Die Definition ist in der RFC 2616 zu finden [4]. Da HTTP selbst gut funktionierte und zur damaligen Zeit Webseiten noch weniger komplex waren, wurde der Fokus nicht auf Performance gelegt. So haben sich im Laufe der Jahrzehnte einige Probleme entwickelt.

Eines der größten Probleme, ist das so genannte Head-of-Line-Blocking-Problem. Welches Auftritt, wenn viele Verbindungsanfragen gestellt werden. Bei HTTP 1.1 wird für jedes Objekt auf einer Webseite eine separate GET-Anfrage gesendet. Dies kann bei heutigen Seiten bis zu 8 TCP-Verbindungen gleichzeitig aufbauen. In HTTP/2 wird durch ein Server Push dem Server ermöglicht, zusätzlichen Content, der zu einer GET-Anfrage gehört, an den Client zu senden. Er verwendet dabei die bestehende Verbindung [5]. Es wird also für eine Seite nur noch eine TCP-Verbindung geöffnet. Dies löst das Head-of-Line-Problem.

Abbildung 6 zeigt, wie sich der Verbindungsaufbau von HTTP 1.1 zu HTTP/2 unterscheidet (vgl. [5]). Während HTTP 1.1 zwei Verbindungen benötigt, kommt HTTP/2 mit einer Verbindung aus. Alle nötigen Dateien werden mit der ersten Verbindungsanfrage zum Client per Push gesendet.

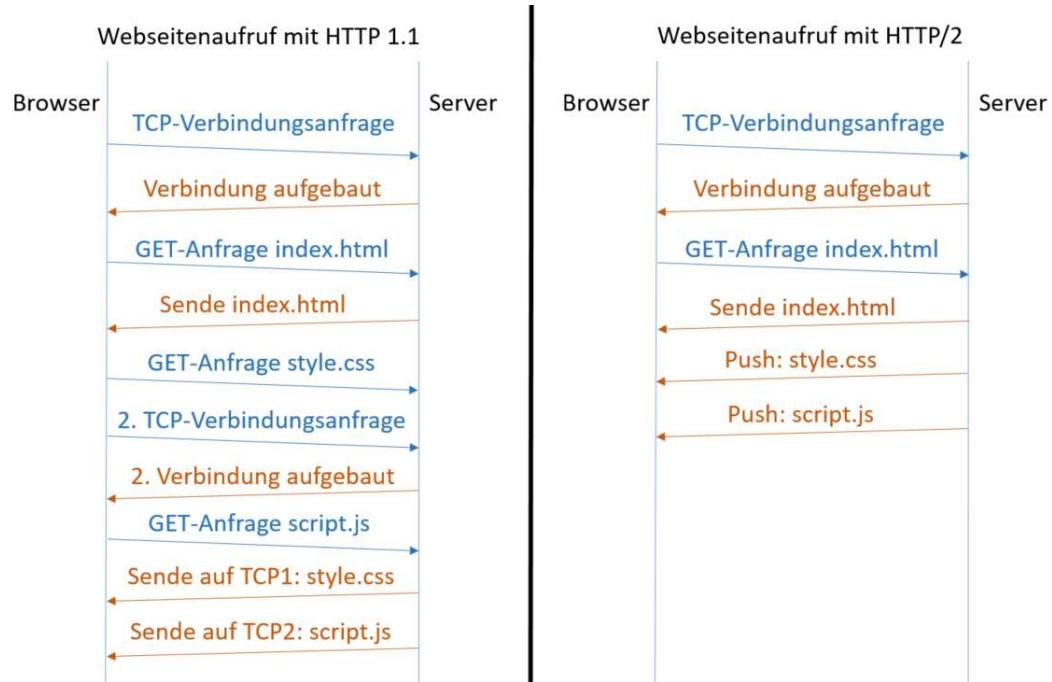


Abbildung 6: Vergleich von Verbindungsauflaufbau HTTP 1.1 und HTTP/2

Ein weiterer Vorteil ist, dass der HTTP/2-Header redundante Daten, wie Datum, Adresse oder Verbindungen, nicht jedes Mal aufs Neue mitsenden muss. Des Weiteren kann der Header komprimiert werden.

REST und SOAP verwenden HTTP 1.1. gRPC arbeitet mit dem neuen HTTP/2.

2.3. RPC und RMI

Zum Bewerkstelligen der verteilten Kommunikation bedarf es Techniken, die dabei helfen, Methodenaufrufe mit gewohnter Syntax auf entfernten Rechnern auszuführen. Dabei steht RPC für „Remote Procedure Call“ und RMI für „Remote Methode Invocation“. Beide bezeichnen genau so einen Ausführvorgang. RPC ist der allgemeine Oberbegriff und RMI ist der in Java verwendete. RPC und RMI sind Synonyme.

RPCs sind in der RFC 1057 [6] und RFC 5531 [7] definiert.

Bei einem RPC verbindet ein Client zu einem Server, um dort eine Methode aufzurufen (Abbildung 7). Dies soll mit gleicher Syntax für den Programmierer funktionieren, als würde er eine lokale Methode aufrufen.

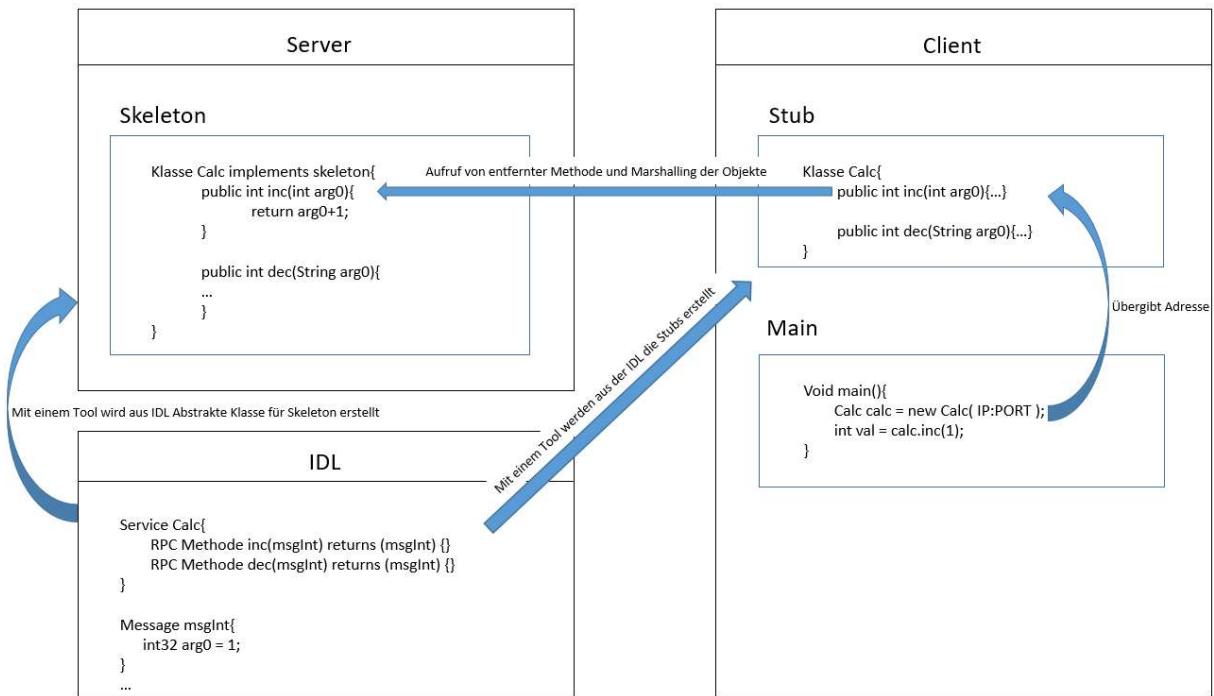


Abbildung 7: Allgemeiner Aufbau eines RPC-/RMI-Systems

Um dies zu bewerkstelligen, müssen auf dem Client und Server Klassen implementiert werden, die die Kommunikation übernehmen. Beim Server nennt man diese Klasse **Skeleton** und beim Client **Stub**. Beide Klassen werden in der Regel von externen Interface Compiler aus IDL-Dateien erstellt. Beim Server muss der Skeleton, der ein Interface oder eine abstrakte Klasse ist, mit Programmcode gefüllt werden. Der Stub dagegen, muss beim Client in das Projekt importiert werden und kann wie ein normales Objekt mit einem Konstruktor erstellt werden. Damit der Client eine Verbindung zu einem Server aufbauen kann, muss er dem Stub noch die Adresse zum Server übergeben. Die Kommunikation erfolgt dabei wie eine Blackbox für den Programmierer. Skeleton und Stub kümmern sich um Marshalling, Objektreferenzen und Kommunikationsprobleme.

Von den drei in dieser Arbeit implementierten Protokollen sind SOAP und gRPC die RPC-Protokolle. REST ist kein RPC-Protokoll, besitzt daher weder IDL (Kap. 2.3.2) noch Stubs oder Skeleton, muss allerdings das Marshalling-Problem trotzdem lösen.

2.3.1. Marshalling

Methodenaufrufe umfassen das Übersenden von Objekten als Argument oder als Rückgabewert. Dabei müssen die Objekte in eine übertragbare Form umgewandelt werden. Diese Transformation nennt man **Marshalling**. Objekte werden in einen Byte-Stream umgewandelt, übertragen und beim Empfänger wieder in ein Objekt gleichen Types zurückgewandelt (**Unmarshalled**). Dieser Vorgang ist in Abbildung 8 dargestellt. Dort wird ein Objekt der Klasse Kunde in den Mediatype JSON gemarshalled.

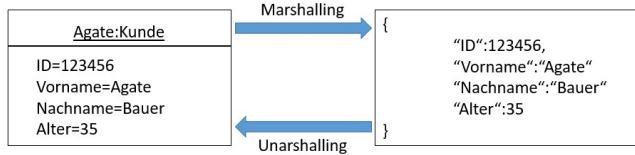


Abbildung 8: Marshalling/Unmarshalling

Der Begriff Serialisation, der in Java häufig für den gleichen Vorgang verwendet wird, beschreibt den gleichen Vorgang und kann als Synonym verstanden werden ([8] S. 138–144). Dieser stammt von der Java-Klasse Serializable und stellt eine konkrete Umsetzung eines Marshalling-Verfahrens dar. Dieses Verfahren wird in der „Java RMI“-API verwendet und ist nicht Teil dieser Arbeit.

Client und Server müssen sich beim Übertragen eines Objektes, auf eine Übertragungsform einigen. Diese Formen nennt man „Mediatypes“ (Kap. 2.3.1.1 S.15) und das Aushandeln, welcher Mediatype verwendet werden soll, nennt man Content Negotiation. HTTP verwendet hierfür den Header, um dem Gesprächspartner mitzuteilen, in welchem Mediatype die Daten im Body dargestellt sind und welche Mediatypen bei einer Antwort akzeptiert werden. Der Mediatype sagt also aus, wie die Bytes in der Nachricht zu interpretieren sind.

```

POST /microServices/webapi/persistenzBA/anfordernBatch?
    Fortschritt=MSBenchmark.microServices.FortschrittgRPC%
    4079f82965 HTTP/1.1
Accept: application/json; charset=UTF-8
Accept-Encoding: gzip, deflate
Content-Type: application/json; charset=UTF-8
Content-Length: 692
Host: localhost:28080
Connection: Keep-Alive

```

Listing 4: HTTP-POST-Header

Listing 4 zeigt nochmals den POST-Request-Header aus Kapitel 2.2.1. Unter „Content-Type:“ steht welcher Mediatype für die Daten im Body verwendet wurde. Nun kann der Server, der die Post-Anfrage erhält, überprüfen, ob er eine Methode „anfordernBatch“ besitzt, die ein Objekt der Klasse FortschrittgRPC als Argument nimmt. Des Weiteren muss der Server prüfen, ob er den Medientyp application/json versteht und wieder zu einem Objekt unmarshalen kann. Wenn er es kann, akzeptiert er die Nachricht und sendet eine Response mit Status 200 zurück zum Client. Wenn nicht, wird eine Response mit entsprechendem Fehlercode an den Client zurück gesendet.

2.3.1.1. *Mediatypes*

Um Objekte von und an verteilte Systeme übertragen zu können, benötigt man ein Medienformat, welches die Daten der Objektattribute in eine übertragbare Form bringt. Das Medienformat muss es ermöglichen, das Objekt erst beim Sender zu marshallen und später beim Empfänger wieder zu unmarshalen.

Der Body einer POST-Anfrage kann nach Belieben mit Daten gefüllt werden, um die Daten allerdings programmiersprachenunabhängig zu einem korrekten Objekt zurückwandeln zu können, muss man sich auf Standards einigen. Der ASCII-Zeichensatz verwendet z. B. 1 Byte pro Symbol und Unicode (UTF-8) 2 Byte. Unicode basiert zwar auf dem ASCII-Zeichensatz, wird allerdings anders interpretiert. Wenn der Empfänger in Unicode sendet und der Empfänger ASCII erwartet, könnte ein Objekt nicht wieder

korrekt hergestellt werden. Mediatypen für HTTP werden in der RFC 2046 festgelegt [9]. Eine vollständige Liste aller offiziell zugelassenen Medientypen ist in der IANAMIME zu finden [10].

Die Wahl des Mediatypen ist ausschlaggebend für die Performance beim Übertragen von Objekten. Mediatypen können Zusatzinformationen mit sich bringen. Diese können nützlich sein oder einfach nur unnötigen Overhead darstellen. Ein Format mit viel Overhead bremst die Übertragung aus. Allerdings bringt mehr Overhead evtl. andere Vorteile. Es muss also nicht der Medientyp, der am schlankesten/schnellsten überträgt, auch immer der am besten geeignete für ein bestimmtes Problem sein.

In den Folgenden 3 Unterkapiteln, werden nun mit allen drei in diesem Projekt verwendeten Mediatypes das in Abbildung 9 dargestellte UML-Objekt nachgebildet.

Samsung LCD4K:LCD-Bildschirm
ID=123456
Marke=Samsung
4K=true
Zubehör=Fernbedienung, Anleitung, Kabel

Abbildung 9: Beispielobjekt

2.3.1.2. JSON

JSON bedeutet **JavaScripT Object Notation** und ist ein human-readable⁶-Format. Der Standard ist in der RFC 7159 definiert [11]. Es wird im Header von HTTP als „application/json“ bezeichnet. JSON besitzt weder Namespaces noch schemabasierte Validierungen ([12] S. 89), ist daher das einfachste der drei Medienformate.

Die Datentypen werden in JSON folgendermaßen dargestellt:

Nullwert: null

Boolean: true und false

Zahl: Ziffern 0 bis 9; negativen Zahlen wird – vorangestellt; Exponenten werden durch den Buchstaben E dargestellt.

Zeichenkette: Werden von “ “ (Anführungszeichen) umklammert.

Array: In eckigen Klammern [] wird der Inhalt eines Arrays dargestellt.

Objekte: In geschweiften Klammern { } wird der Inhalt eines Objektes dargestellt. Attribute des Objekts werden von Kommas , getrennt. Ein Attribut ist ein Key-Value-Paar, getrennt durch ein Doppelpunkt : . Jeder Key darf in einem Objekt nur einmal vorkommen.

⁶ Menschenlesbar – Informationen werden im Klartext für Menschen lesbar versendet

Das Beispielobjekt aus Abbildung 9 wird in Listing 5 als der JSON Darstellung abgebildet.

```
{  
    "id":123456,  
    "marke":"Samsung",  
    "vierK":true,  
    "zubehoer":["Fernbedienung", "Anleitung", "Kabel"]  
}
```

Listing 5: Darstellung des Beispielobjekts in JSON

In Hexadezimal ist die JSON-Darstellung in Listing 6 abgebildet. Die farblichen Markierungen sollen zur Orientierung die einzelnen Attribute Highlighten.

```
0x00 7b226964223a3132333435362c226d61  
0x10 726b65223a2253616d73756e67222c22  
0x20 766965724b223a747275652c227a7562  
0x30 65686f6572223a5b224665726e626564  
0x40 69656e756e67222c22416e6c65697475  
0x50 6e67222c224b6162656c225d7d
```

Listing 6: Hexadezimaldarstellung der JSON-Darstellung des Beispielobjekts

Damit hat das Objekt in JSON eine Größe von 93 Bytes.

2.3.1.3. XML

XML steht für eXtensible Markup Language und wurde vom W3C definiert [13]. XML ist genauso wie JSON ein Klartextformat, es bietet allerdings die Möglichkeit zum Erstellen von Namespaces und Schemas. Um eine mit SOAP versendete Nachricht verstehen zu können, muss man zunächst erstmal die Syntax und die Namespaces verstehen. Schemas werden für die WSDL-Datei benötigt und in Kapitel 2.3.1.3 erklärt.

Die Syntax setzt sich bei XML wie folgt zusammen:

Element: Ein Element wird mit seinem Namen in spitzen Klammern eingeleitet z. B. `<elementname>`. Das Ende eines Elements wird auch mit dem Namen in spitzen Klammern markiert, allerdings wird ihm ein Slash vorangestellt z. B. `</elementname>`. In einem Element können beliebig viele weitere Elemente verschachtelt werden.

Array: Ein Array wird durch mehrere gleichnamige Elemente, die hintereinander kommen dargestellt.

Z. B.:

```
<fernseher>...</fernseher>  
<fernseher>...</fernseher>
```

Leere Elemente: Leere Elemente sind zulässig. Ein leeres Element kann verkürzt durch das Hinteranhangen des Slashes am Elementnamen dargestellt werden. Somit kann der zweite, abschließende Name des Elements weggelassen werden. Z. B.: `<fernseher/>`

- Attribut:** Attribute sind Begleitinformationen zu einem Element. Sie werden nach dem einleitenden Elementnamen, in spitzen Klammern angegeben, z. B.: `<fernseher laden="mediamarkt">`. Attribute sind immer Key/Value-Paare. Der Wert muss dabei in einfachen oder doppelten Anführungszeichen angegeben werden. Attribute können nicht wie Elemente geschachtelt werden.
- Kommentare:** Kommentare haben die gleiche Form wie in HTML. `<!--` leitet einen Kommentar ein und `-->` schließt einen Kommentar.

Nun folgen die Datentypen, welche jeweils als Zeichenkette dargestellt werden:

- Nullwert:** wird als leeres Element dargestellt.
- Boolean:** **true** und **false**
- Zahl:** Ziffern **0** bis **9**; negativen Zahlen wird – vorangestellt; Exponenten werden durch den Buchstaben **E** dargestellt.

Das in Abbildung 9 (S. 16) dargestellte Beispielobjekt, sieht in XML also folgendermaßen aus:

```
<arg0>
    <id>123456</id>
    <marke>Samsung</marke>
    <vierK>true</vierK>
    <zubehoer>Fernbedienung</zubehoer>
    <zubehoer>Anleitung</zubehoer>
    <zubehoer>Kabel</zubehoer>
</arg0>
```

Listing 7: XML-Kurzdarstellung des Beispielobjekts

SOAP benötigt allerdings zusätzliche Information, um aus diesem XML-Objekt wieder ein POJO⁷ machen zu können. Hierzu werden Namespaces verwendet.

- Namespace:** Namespaces sind wie Packages in Java zu verstehen. Sie dienen der Sortierung und Identifikation von Elementen und helfen, Namenskollisionen zu vermeiden. Sie geben dem Element einen Qualified Name oder auch QName genannt. Der Namespace wird wie ein Attribut mit dem Attributnamen xmlns angegeben und hat als Wert eine URI, welche den Namespace identifiziert. Die URI muss nicht zwingend auf ein existierendes Dokument zeigen. Sie muss nur eindeutig für die definierte Markup Language sein. Die Angabe des Namespace ist in Listing 8 in Fettschrift dargestellt.

```
<arg0 xmlns="http://BachelorThesis.Ausarbeitung/Artikel">
    <id>123456</id>
    <marke>Samsung</marke>
    <vierK>true</vierK>
    <zubehoer>Fernbedienung</zubehoer>
    <zubehoer>Anleitung</zubehoer>
    <zubehoer>Kabel</zubehoer>
</arg0>
```

Listing 8: XML-Namespace-Beispiel

⁷ Plain Old Java Object – Gewöhnliches Java Objekt

Namespaces können auch als Präfixe angegeben werden. Hierbei muss zuerst der Namespace einem Präfix zugeordnet werden. Dies geschieht durch Verwenden des xmlns, gefolgt von einem Doppelpunkt und der Angabe des Präfixes. Ihm wird dann noch die URI zugewiesen. Nun kann bei einem Element, durch Angabe des Präfixes vor dem Elementnamen, angegeben werden, ob es sich in diesem Namespace befindet. Dies ist in Listing 9 mit fetter Schrift abgebildet.

```
<bestellungen xmlns:artikel="http://BachelorThesis.Ausarbeitung/Artikel">
<artikel:arg0>
    <artikel:id>123456</artikel:id>
    <artikel:marke>Samsung</artikel:marke>
    <artikel:vierK>true</artikel:vierK>
    <artikel:zubehoer>Fernbedienung</artikel:zubehoer>
    <artikel:zubehoer>Anleitung</artikel:zubehoer>
    <artikel:zubehoer>Kabel</artikel:zubehoer>
</artikel:arg0>
</bestellungen>
```

Listing 9: XML-Namespace-Präfixbeispiel

SOAP sendet seine Nachrichten im XML-Format. Eine vollständige SOAP-Nachricht sieht wie folgt aus:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <ns2:getObject xmlns:ns2="http://BachelorThesis.Ausarbeitung/"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:type="ns2:getObject">
            <arg0>
                <id>123456</id>
                <marke>Samsung</marke>
                <vierK>true</vierK>
                <zubehoer>Fernbedienung</zubehoer>
                <zubehoer>Anleitung</zubehoer>
                <zubehoer>Kabel</zubehoer>
            </arg0>
        </ns2:getObject>
    </soap:Body>
</soap:Envelope>
```

Listing 10: SOAP-Darstellung des Beispielobjekts

Die in Listing 10 dargestellte SOAP-Nachricht wird in Listing 11 nochmals in Hexadezimal abgebildet. Die einzelnen Abschnitte wurden zur Übersichtlichkeit farblich hervorgehoben.

Die SOAP-Nachricht benötigt demnach 426 Byte. Damit nimmt SOAP für das Beispielobjekt aus Kapitel 2.3.1 den meisten Speicher ein.

0x000	3c 73 6f 61 70 3a 45 6e 76 65 6c 6f 70 65 20 78
0x010	6d 6c 6e 73 3a 73 6f 61 70 3d 22 68 74 74 70 3a
0x020	2f 2f 73 63 68 65 6d 61 73 2e 78 6d 6c 73 6f 61
0x030	70 2e 6f 72 67 2f 73 6f 61 70 2f 65 6e 76 65 6c
0x040	6f 70 65 2f 22 3e 3c 73 6f 61 70 3a 42 6f 64 79
0x050	3e 3c 6e 73 32 3a 67 65 74 4f 62 6a 65 63 74 20
0x060	78 6d 6c 6e 73 3a 6e 73 32 3d 22 68 74 74 70 3a
0x070	2f 2f 42 61 63 68 65 6c 6f 72 54 68 65 73 69 73
0x080	2e 41 75 73 61 72 62 65 69 74 75 6e 67 2f 22 20
0x090	78 6d 6c 6e 73 3a 78 73 69 3d 22 68 74 74 70 3a
0x0a0	2f 2f 77 77 77 2e 77 33 2e 6f 72 67 2f 32 30 30
0x0b0	31 2f 58 4d 4c 53 63 68 65 6d 61 2d 69 6e 73 74
0x0c0	61 6e 63 65 22 20 78 73 69 3a 74 79 70 65 3d 22
0x0d0	6e 73 32 3a 67 65 74 4f 62 6a 65 63 74 22 3e 3c
0x0e0	61 72 67 30 3e 3c 69 64 3e 31 32 33 34 35 36 3c
0x0f0	2f 69 64 3e 3c 6d 61 72 6b 65 3e 53 61 6d 73 75
0x100	6e 67 3c 2f 6d 61 72 6b 65 3e 3c 76 69 65 72 4b
0x110	3e 74 72 75 65 3c 2f 76 69 65 72 4b 3e 3c 7a 75
0x120	62 65 68 6f 65 72 3e 46 65 72 6e 62 65 64 69 65
0x130	6e 75 6e 67 3c 2f 7a 75 62 65 68 6f 65 72 3e 3c
0x140	7a 75 62 65 68 6f 65 72 3e 41 6e 6c 65 69 74 75
0x150	6e 67 3c 2f 7a 75 62 65 68 6f 65 72 3e 3c 7a 75
0x160	62 65 68 6f 65 72 3e 4b 61 62 65 6c 3c 2f 7a 75
0x170	62 65 68 6f 65 72 3e 3c 2f 61 72 67 30 3e 3c 2f
0x180	6e 73 32 3a 67 65 74 4f 62 6a 65 63 74 3e 3c 2f
0x190	73 6f 61 70 3a 42 6f 64 79 3e 3c 2f 73 6f 61 70
0x1a0	3a 45 6e 76 65 6c 6f 70 65 3e

Listing 11: Hexadezimaldarstellung der SOAP-Nachricht des Beispielobjekts

2.3.1.4. Protocol Buffer Wire Format

gRPC verwendet zum Marshallen das Wire Format von Protocol Buffer. Diese Darstellung ist im Gegensatz zu JSON und XML keine human readable Darstellung. Wie das Encoding funktioniert, ist auf der Developerseite von Google nachzulesen [14]. Da es sich um ein Binärformat handelt, muss man, um eine Nachricht verstehen zu können, erst begreifen, nach welchen Regeln Datentypen gemarshallt werden. An dieser Stelle müssen zum Verständnis ein wenig von der IDL vorweggenommen werden. Die IDL von gRPC wird im Detail im Kapitel 2.3.2.2 erklärt.

Eine Nachricht, die zwischen zwei gRPC-Services ausgetauscht werden kann, wird in der Protobuf-IDL wie folgt definiert:

```
message LCDBildschirm{
    int32 id = 1;
    string marke = 2;
    bool vierk = 3;
    repeated string zubehoer = 4;
}
```

Listing 12: Beispielobjekt als gRPC-Protobuf-IDL-Darstellung

Jedes Attribut der Nachricht bekommt hierbei eine FieldNr zugewiesen (Die Zahlen hinter den Gleichheitszeichen.). Jeder Datentyp hat zudem einen Wiretype, welcher einer Kategorie entspricht, in der sich ein Datentyp befindet.

Wiretype-Nummer	Bedeutung	Datentypen in der Kategorie
0	Varint	int32, int64, uint32, uint64, sint32, sint64, bool, enum
1	64-Bit	fixed64, sfixed64, double
2	Length-delimited	string, bytes, embedded messages, packed repeated fields
3	Start Group	groups (deprecated)
4	End Group	groups (deprecated)
5	32-Bit	fixed32, sfixed32, float

Tabelle 1: Wiretype-Nummern und ihre Bedeutung

Die Zahl 123456 aus dem Beispielobjekt wird dann folgendermaßen berechnet:

1. 123456 entspricht binär -> 0000 0001 1110 0010 0100 0000
2. Es werden immer 7 Bit genommen und vorne eine 0 (Null) angehängt, außer beim letzten Block, dort wird eine 1 angehängt.
-> 0000 0001 1110 0010 0100 0000 -> **0000 0111 0100 0100 1100 0000**
3. Danach werden die Bytes rückwärts genommen (Little endian)
-> 0000 0111 0100 0100 1100 0000 -> 1100 0000 0100 0100 0000 0111
In hexadezimaler Darstellung ist das dann **0x C0 C4 07**

Zur Identifikation wird dann noch ein Byte vorangestellt, welches sich mit ((FieldNr << 3) | Wiretyp-Nummer) berechnen lässt. Hier also ((1 << 3) | 0) = (8 | 0) = **0x 08**.

Damit ist die volle Darstellung der Zahl als Protobuf **08 C0 C4 07**. So kann die Hexzahl in der gRPC-Nachricht gefunden werden.

```
0x00 08 c0 c4 07 12 07 53 61 6d 73 75 6e 67 18 01 22
0x10 0d 46 65 72 6e 62 65 64 69 65 6e 75 6e 67 22 09
0x20 41 6e 6c 65 69 74 75 6e 67 22 05 4b 61 62 65 6c
```

Listing 13: Hexadezimale Darstellung des Beispielobjekts im gRPC-Protobuf-Wireformat

Bei einem String wird erst die Identifikationsnummer berechnet, anschließend folgt die Länge des Strings in Binärdarstellung und dann der String selbst. So setzt sich der Markenname des Beispielobjekts wie folgt zusammen ((FieldNr 2 << 3) | Typ 2) = 0x12 gefolgt von der Länge des Strings „Samsung“ in Binär (also 0x07), gefolgt vom String selbst. Daraus ergibt sich folgende Hexzahl 0x12 07 53 61 6d 73 75 6e 67.

Der boolesche Wert entspricht einem Byte mit 0x00 für false oder 0x01 für true. Ansonsten wird der Rest der Nachricht im gleichen Muster codiert. Zur Übersicht hier nochmals die Message als IDL definiert und der dazugehörige BLOB⁸ mit farblichen Markierungen in Listing 14.

```
message LCDBildschirm{
    int32 id = 1;
    string marke = 2;
    bool vierk = 3;
    repeated string zubehoer = 4;
}
```

⁸ Binary Long Object

0x00	08 c0 c4 07	12 07 53 61 6d 73 75 6e 67	18 01	22
0x10	0d 46 65 72 6e 62 65 64 69 65 6e 75 6e 67	22 09		
0x20	41 6e 6c 65 69 74 75 6e 67	22 05 4b 61 62 65 6c		

Listing 14: gRPC-Wireformat, farblich hervorgehoben

Damit ist die Länge der gRPC-Nachricht des Beispielobjekts 48 Byte lang. gRPC besitzt also die kleinste Darstellung des Beispielobjekts.

2.3.2. IDL – Interface Description Language

Eine Herausforderung des RPCs ist das Übertragen von Objekten. Diese haben wir mittels den Mediatypes im Kapitel 2.3.1 Marshalling gelöst. Eine andere nicht weniger wichtige Herausforderung ist zu wissen, über welche Methoden eine RPC-Klasse verfügt (vgl. Abbildung 7 S.14). Hier kommen IDLs (Interface Description Language) zum Einsatz. Mit ihrer Hilfe können so genannte Stubs erstellt werden, die in Java benötigt werden, um einen entfernten Funktionsaufruf mit normaler Java-Syntax zu bewerkstelligen. Ein Stub ist ein Teilabbild der Klasse, die sich auf dem Server befindet. Sie wird üblicherweise mit einem Compiler-Tool generiert (wsimport.exe bei SOAP und protoc.exe bei gRPC). Der Stub enthält Informationen, die benötigt werden, um die Verbindung herzustellen und Objekte zu übertragen, sowie alle entfernt aufrufbaren Methoden mit Argumenten und Rückgabetypen. Da REST kein RPC-Protokoll ist, besitzt REST keine IDL.

2.3.2.1. WSDL – Web Service Description Language

SOAP verwendet WSDL, um seine Services den Nutzern zur Verfügung zu stellen. Die WSDL ist eine IDL. Diese Beschreibungssprache basiert auf einem XML-Format. Aktuell sind 2 verschiedene Standards für diese Beschreibungssprache zu finden: WSDL 1.1 und WSDL 2. Der von JAX-WS⁹ verwendete Standard ist WSDL 2. WSDL 1.1 besitzt allerdings noch weite Verbreitung ([12] S. 160). WSDL ist im W3C definiert [15]. Der Komfort und Vorteil von SOAP gegenüber REST und gRPC liegt darin, dass die WSDL-Datei nach dem Deploy per URI zur Verfügung gestellt wird. Dies erleichtert das Erstellen der Stubs auf dem Client.

WSDL ist in 2 Teile gegliedert: einen abstrakten und einen konkreten Teil. Der abstrakte Teil beschreibt einen Webservice mit den Nachrichten, die er sendet oder empfängt. Der konkrete Teil unterscheidet WSDL vom ausschließlich abstrakten IDL. In diesem konkreten Teil werden die Binding- und Port-Adressen definiert, die für einen Verbindungsaufbau essentiell sind.

Wsimport ist das von Java mitgelieferte Programm, mit dem aus einer WSDL-Datei die Java-Stubs generiert werden können. Wsimport bekommt entweder eine URI zu einer WSDL-Datei oder eine Pfadangabe zu einer lokal gespeicherten WSDL-Datei als Argument übergeben. Die so generierten Stub-Dateien müssen anschließend in das bestehende Projekt importiert werden.

So können andere Programmierer die Stubs zur Nutzung für einen Webservice aus dieser WSDL-Datei generieren. Die WSDL-Datei ist, sofern nicht anders konfiguriert, unter folgender Adresse nach dem Deploy über einen Webbroweser abrufbar:

<http://<IP-Adr>:<Port>/<Projektname>/<Servicename>?wsdl>

⁹ JAX-WS ist die Java-Implementierung von SOAP.

Um nun eine WSDL-Datei verstehen zu können, müssen zunächst erst noch XML-Schemas erläutert werden. XML-Schemas beschreiben den Aufbau von Objekten mit komplexen Typen (Complex types) und einfachen Typen (Simple Types). Complex Types beschreiben, wie Elemente in Objekten organisiert und verschachtelt sind. Simple Types sind Datentypen wie Integer, Float oder String.

Das Beispielobjekt aus Kapitel 2.3.1.1 (S. 15) sieht als SOAP-Schema folgendermaßen aus:

```
<complexType name="LCD-Bildschirm">
    <sequence>
        <element name="id" type="integer"/>
        <element name="marke" type="string"/>
        <element name="4K" type="boolean"/>
        <element name="zubehör" maxOccurs="unbounded" minOccurs="0" type="string"/>
    </sequence>
</complexType>
```

Listing 15: SOAP-Schema

In Listing 15 ist ein ComplexType mit dem Namen der Klasse LCD-Bildschirm enthalten.

```
<complexType name="LCD-Bildschirm">
    ...
</complexType>
```

In ihm gibt es ein Element Sequence, welches alle SimpleTypes des Beispielobjekts enthält.

```
<sequence>
    ...
</sequence>
```

Die SimpleTypes sind als XML-Element-Attribut mit dem Namen „type“ angegeben.

```
<element name="id" type="integer"/>
<element name="marke" type="string"/>
<element name="4K" type="boolean"/>
<element name="zubehör" maxOccurs="unbounded" minOccurs="0" type="string"/>
```

Die zusätzlichen Attribute maxOccurs und minOccurs stehen für die Häufigkeit, wie oft dieses Element vorkommen darf. Dies steht also für ein Array. MaxOccurs hat hier den Wert unbounded, welcher für ein unbeschränkt häufiges Vorkommen an Elementen mit dem Namen Zubehör steht. MinOccur wurde auf 0 gesetzt. Das bedeutet, dass dieses Element auch leer sein darf. Wenn MinOccur und MaxOccur nicht angegeben sind, ist ihr Default-Wert 1.

```
<element name="zubehör" maxOccurs="unbounded" minOccurs="0" type="string"/>
```

Eine vollständige Liste aller SimpleTypes ist im Schema „Definition Language Part 2“ auf der W3C Webseite zu finden [16].

Schemas sind stets im Root-Element <Schema> zu finden, welches Member des Namespace <http://www.w3.org/2001/XMLSchema> ist. Die XML-Schema-Spezifikation beschreibt detailliert wie ein Objekt als XML-Schema dargestellt werden muss.

Die Grundstruktur einer WSDL-Datei besteht aus sieben Elementen: <import>, <types>, <messages>, <portType>, <operations>, <binding>, <service>, welche alle im <definitions> Element verschachtelt werden. Die Struktur wird in Listing 16 abgebildet.

```

</wsdl:definitions>
<wsdl:types>

...
<wsdl:message name="batchVerarbeitung">
...
<wsdl:portType name="TransformatorInterface">
    <wsdl:operation name="batchVerarbeitung">
...
<wsdl:binding name="TransformatorServiceSoapBinding"
type="tns:TransformatorInterface">
...
<wsdl:service name="TransformatorService">
...
</wsdl:definitions>

```

Listing 16: Grundstruktur einer WSDL-Datei

Types: Sie deklarieren mit Hilfe der XML-Schema-Sprache alle in der WSDL-Datei verwendeten complex Datatypes. Hier ein Beispiel der Methode batchVerarbeitung aus der Klasse Transformator, die ein Array aus Strings als Übergabeparameter akzeptiert.

```

<xss:element name="batchVerarbeitung" type="tns:batchVerarbeitung"/>...

<xss:complexType name="batchVerarbeitung">
    <xss:sequence>
        <xss:element maxOccurs="unbounded" minOccurs="0" name="arg0"
type="xs:string"/>
    </xss:sequence>
</xss:complexType>

```

Import: Wird verwendet, um Definitionen aus anderen WSDL-Dateien zu importieren. Dieses Element ist optional (in Listing 16 nicht abgebildet).

Message: Beschreibt mit Hilfe der XML-Schema-Sprache, wie eine Nachricht aufgebaut ist. Im Beispield wird das oben definierte Schema batchVerarbeitung verwendet.

```

<wsdl:message name="batchVerarbeitung">
    <wsdl:part element="tns:batchVerarbeitung" name="parameters">
    </wsdl:part>
</wsdl:message>

```

PortType: Zu PortType gehören auch die **Operationen**. Sie Beschreiben die Schnittstellen eines Web Services. Input sind Übergabeparameter und Output sind Rückgabewerte. Input und Output sind optional. Ein Weglassen der Elemente entspricht einem Void in Java. Im Beispiel wird die oben definierte Message als Übergabeparameter akzeptiert und eine Message vom Typ batchVerarbeitungResponse als Rückgabewert zurückgegeben.

```

<wsdl:portType name="TransformatorInterface">
    <wsdl:operation name="batchVerarbeitung">
        <wsdl:input message="tns:batchVerarbeitung" name="batchVerarbeitung"></wsdl:input>
        <wsdl:output message="tns:batchVerarbeitungResponse" name="batchVerarbeitungResponse"></wsdl:output>
    </wsdl:operation>
</wsdl:portType>

```

Binding: Ordnet einem PortType und dessen Operationen einem Protokoll und einer Encodierung zu. Im Beispiel wird unter <soap:binding> das Attribut „transport“ auf <http://schemas.xmlsoap.org/soap/http> gesetzt, was SOAP mitteilt, dass diese Operationen an HTTP gebunden werden.

```
<wsdl:binding name="TransformatorServiceSoapBinding"
  type="tns:TransformatorInterface">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="batchVerarbeitung">
    <soap:operation soapAction="" style="document"/>
    <wsdl:input name="batchVerarbeitung">
      <soap:body use="Literal"/>
    </wsdl:input>
    <wsdl:output name="batchVerarbeitungResponse">
      <soap:body use="Literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

Service: Ordnet eine Internet-Adresse einem bestimmten Binding zu. Im unteren Beispiel wird das im vorherigen Beispiel definierte Binding-Element, einem Service zugeordnet.

```
<wsdl:service name="TransformatorService">
  <wsdl:port binding="tns:TransformatorServiceSoapBinding"
    name="TransformatorPort">
    <soap:address
      location="http://localhost:8080/microServices/TransformatorService"/>
  </wsdl:port>
</wsdl:service>
```

Documentation: Gibt human readable Zusatzinformationen zur WSDL-Datei. Dieses Element ist optional (in Listing 16 nicht abgebildet).

2.3.2.2. *Protocol Buffer IDL*

Protocol Buffer (Protobuf) wurde ursprünglich von Google entwickelt. Protobuf wurde von Google 2008 mit der Version 2 (proto2) als Open-Source-Projekt auf Github zur freien Benutzung zur Verfügung gestellt [17]. Protobuf ist kein offizieller Standard, daher ist seine Definition auf der Google-Developer-Seite zu finden [18]. Die aktuelle Version ist Proto3. Proto3 ist die von gRPC verwendete Version.

Die von Protobuf verwendete IDL wird im Gegensatz zu WSDL nicht vom gRPC-Service als Download zur Verfügung gestellt. Sie muss über andere Kommunikationswege zum Nutzer übertragen werden. Aus ihr werden daraufhin die Klassen für die Services und alle Nachrichtenobjekte generiert. Um das Generieren aus der IDE¹⁰ heraus erledigen zu können, gibt es ein Maven-Plug-In, welches diese Aufgabe übernimmt. Dadurch muss für das Generieren der Skeletons und Stubs nicht in eine Command-Shell gewechselt werden.

¹⁰ Integrated Development Environment – Programm, das bei der Entwicklung neuer Programme hilft.

```

syntax = "proto3";
option java_multiple_files = true;
package grpcServices;

message FortschrittRequest {
    int32 parameter = 1;
    int32 protokoll = 2;
    repeated string stapel = 3;
    repeated string stapelGedreht = 4;
    bool alsStapel = 5;
    ...
}

message StapelMsg {
    repeated string message = 1;
}

...
message StringMsg {
    string message = 1;
}
...

service Transformator {
    rpc batchVerarbeitung(StapelMsg) returns (StringMsg);
}

```

Listing 17: Auszug aus einer Protobuf-Datei

In Listing 17 wird ein Teil der Protobuf-Datei, aus diesem Projekt abgebildet. Diese wird nun im Einzelnen erklärt. Für gRPC ist Protobuf Version 3 vorgegeben, daher wird in der ersten Zeile die Variable „syntax“ auf „Proto3“ gesetzt.

```
syntax = "proto3";
```

„Option java_multiple_files“ bedeutet, dass die Umwandlung der Protobuf-Datei in einen Output mit mehreren Java-Dateien erfolgen darf.

```
option java_multiple_files = true;
```

Das Schlüsselwort „Package“ bedeutet, dass die Dateien in den angegebenen Unterordner gepackt werden sollen, gleich bedeutend zu einem Java Package.

```
package grpcServices;
```

Eine „Message“ stellt ein zu übertragendes Objekt dar.

```
message FortschrittRequest {
    ...
}
```

Jedes Attribut in einer Message bekommt eine FieldNr zugewiesen, die von 1 beginnend und immer um 1 inkrementiert wird. Die FieldNr ist die Nummer hinter dem Gleichheitszeichen im Beispiel. „Int32“ ist der Datentyp Integer, „string“ ist ein String und „bool“ ein Boolean. „Repeated“ steht für eine Liste an Objekten.

```
int32 parameter = 1;
int32 protokoll = 2;
repeated string stapel = 3;
repeated string stapelGedreht = 4;
bool alsStapel = 5;
```

„Service“ steht für die Klasse, welche ihre Methoden für einen Fernaufruf bereitstellt.

```
service Transformator {  
    ...  
}
```

„Rpc“ stellt eine Methode des Service dar, die über das Netzwerk aufgerufen werden kann. RPC-Methoden bekommen die vorher definierten Messages als Argumente übergeben und geben Messages wieder als Rückgabewert zurück. Argumente und Rückgabewerte werden beide in Klammern angegeben. Dem Rückgabewert wird das Schlüsselwort „returns“ vorangestellt.

```
rpc batchVerarbeitung(StapelMsg) returns (StringMsg);
```

2.3.2.3. *Selbstbeschreibende Nachrichten*

REST folgt einem anderen Ansatz bei der Kommunikation. Dabei werden die Methoden als Webserver zur Verfügung gestellt und reagieren auf HTTP-Anfragen wie GET oder POST. REST verzichtet bei diesem Ansatz auf eine IDL, Skeleton und Stubs. Ohne eine IDL setzt REST dem Nutzer Wissen über die Schnittstellen des Webservices voraus. D. h., um eine Methode mit REST aufrufen zu können, benötigt man ihre URI und welche Mediatypes diese akzeptiert. Diese Information ist bei SOAP und gRPC in der IDL enthalten. Die IDLs generieren bei SOAP und gRPC die Stubs inkl. den aufrufbaren Methoden. Ohne Stubs sind die Methoden bei REST dem Kompiler nicht bekannt. Daher ist REST auch kein RPC-Protokoll.

Es werden bei REST für die Kommunikation HTTP-Clients erstellt, die an einen REST-Server HTTP-Requests senden. In einer POST-Nachricht können Objekte übertragen werden. Dabei wird im Header der Nachricht der Mediatype mitgeteilt, der angibt, wie das Objekt gemarshallt wurde. Für dieses Projekt wurde JSON für REST als Mediatype vorgegeben.

2.4. REST - Representational State Transfer

REST verwendet HTTP 1.1 zur Übertragung und ist das einzige der 3 Protokolle, das kein RPC-Protokoll ist. Dadurch ist REST das am einfachsten zu implementierende Protokoll. Es braucht keine Tools, um Stubs zu generieren, da es ohne Stubs auskommt. Zum Deployen eines REST-Service mit Jax-RS, muss ausschließlich die Java-Klasse annotiert werden (Kap. 2.4.1). Der Zugriff auf einen Service erfolgt über HTTP-Clients. Dabei wird dem Client eine Target-Adresse übergeben, welche auf die Ressource (Kap. 2.4.1) verweist, die man verwenden möchte. Da REST ohne IDL und Stubs arbeitet, muss dem Nutzer eines Service bekannt sein, unter welchen URI-Adressen er alle Ressourcen findet, die er nutzen möchte.

2.4.1. RESTful

Die Idee hinter REST ist es, über HTTP-Links auf Ressourcen zugreifen zu können. Eine vollkommen REST konforme Umsetzung einer URI-Hierarchie zu ihren Ressourcen nennt man RESTful. Dabei darf jede URI ausschließlich eine einzige Ressource identifizieren. Allerdings darf umgedreht eine Ressource

über mehrere URIs identifiziert werden. Eine Ressource kann eventuell durch eine andere Perspektive anders identifiziert werden. Zum Beispiel, kann ein Kunde auch gleichzeitig ein Mitarbeiter sein ([12] S. 40).

Eine gekonnte Wahl des Benennungsschemas erleichtert dem Nutzer, auf die gewünschten Ressourcen zugreifen zu können. REST nimmt ihm dabei die Gliederung der URIs mit Hilfe einer Annotation der zu identifizierenden Variable in geschweiften Klammern ab. Möchte man z. B. Objekte der Klasse Kunde zur Verfügung stellen, annotiert man eine Methode mit

```
@Path("/Kunde/{id}")
```

Eine Ressource Kunde ist dann unter www.beispiel.de/Kunde/1 erreichbar. URIs wie diese werden von REST dynamisch während der Laufzeit generiert. Sobald ein neuer Kunde in der Datenbank verfügbar ist, kann dieser unter der entsprechenden Adresse www.beispiel.de/Kunde/n+1 abgerufen werden. In der Annotation muss nicht unbedingt eine ID angegeben werden. Es können auch Namen oder ein Datum gegliedert in Jahr, Monat und Tag verwendet werden z. B. www.beispiel.de/Rechnung/2017/06/23. Dann würde die Annotation folgendermaßen aussehen:

```
@Path("/Rechnung/{Jahr}/{Monat}/{Tag}")
```

2.4.2. Jax-RS

Erstellen eines Webservice

Um einen Webservice mit Jax-RS generieren zu können, muss ausschließlich eine Klasse mit `@Path("Servicename")` annotiert werden. Sobald man dann den Service deployt, ist dieser unter der Adresse „<http://localhost:8080/RestServiceProjektname/webapi/Servicename/>“ verfügbar.

Um eine Methode per GET zur Verfügung zu stellen, gibt es die Annotation `@GET`. Für POST gibt es die `@POST`-Annotation.

Eine GET-Methode kann keine Objekte empfangen. Sie verfügt stattdessen über eine Liste mit String-Argumenten, die sie entgegen nehmen kann. Ein Argument wird in einer Methode mit `@QueryParam(„parametername“)` annotiert. Dies hilft Java dabei, die einzelnen Übergabewerte den Argumenten der Methode zu zuordnen. Die Rückgabewerte sind allerdings gleich wie bei POST, da ein Request immer einen Body besitzt, der jeden beliebigen Mediatype beinhalten kann.

```
@GET  
@Produces(MediaType.TEXT_HTML)  
@Path("/getMethode")  
public String getMethode(@QueryParam("tvname") String tvn) {  
    ...  
}
```

Listing 18: GET-Methode in Jax-RS

In Listing 18 ist eine GET-Methode abgebildet, diese empfängt einen Anfrageparameter „tvname“ und übergibt den String an das Methodenargument „tvn“. Der Rückgabewert kann hier vom Typ TEXT_HTML sein.

Weitere Annotationen sind für Übergabewerte @Consumes(<Mediatype>) und bei Rückgabewerte @Produces(<Mediatype>). Consumes gilt nur für POST-Methoden. So kann aus allen Annotationen folgende POST-Methode konstruiert werden:

```
@POST  
@Consumes(MediaType.APPLICATION_JSON)  
@Produces(MediaType.APPLICATION_JSON)  
@Path("/postMethode")  
public String postMethode(LCDBildschirm tv) {  
    ...  
}
```

Listing 19: POST-Methode in Jax-RS

Die in Listing 19 abgebildete Methode, kann durch die Annotationen mit POST-Nachrichten angesprochen werden. Ihr kann ein Objekt „LCDBildschirm“ mit dem Mediatype „JSON“ übergeben werden. Sie kann ein JSON-Objekt an den Requester zurückgeben. Erreichbar ist sie unter der Adresse „<http://localhost:8080/RestServiceProjektname/webapi/Servicename/postMethode>“.

Zugriff auf einen Webservice

Um nun einen Webservice mit Jax-RS ansprechen zu können, muss man einen HTTP-Client und eine POST- oder GET-Anfrage erstellen. Dazu gibt es die Klasse „Client“ im Package „javax.ws.rs.client“. Zum Generieren des Clients wird ein Builder verwendet, z. B.:

```
Client restClient = ClientBuilder.newClient();
```

Dem Client-Objekt muss nun das Ziel (auch Target genannt) übergeben werden. Dazu gibt es die Klasse „WebTarget“ aus demselben Package. Das Client-Objekt bekommt zunächst in die Methode target() die Basis-Adresse mit Port als String übergeben, also z. B. <http://localhost:8080> und gibt daraufhin ein Objekt der Klasse WebTarget zurück. Als Code sieht das dann so aus:

```
WebTarget restTarget = restClient.target("http://localhost:8080");
```

Mit dem Target können nun Anfragen gesendet und die Responsen empfangen werden. Zuerst muss dafür dem Target die restliche URI übergeben werden, die auf die gewünschten Ressource zeigt:

```
restTarget.path("/RestServiceProjektname/webapi/Servicename/getMethode")
```

Daraufhin müssen, wenn eine GET-Anfrage gesendet werden soll, die Übergabeparameter angegeben werden. Diese sind wie bei GET üblich ein Key-Value-Paar:

```
.queryParam("tvname", "Samsung")
```

Danach muss die Anfrage mit request() erstellt werden. Die GET-Nachricht wird durch den Aufruf der Methode get() versandt. Der Rückgabe-Typ wird der Methode get() als Klasse übergeben werden. In diesem Beispiel wird ein String als Response akzeptiert:

```
request().get(String.class);
```

Der ganze Code ist in ;

Listing 20 zusammengefasst dargestellt.

```

Client restClient = ClientBuilder.newClient();
WebTarget restTarget = restClient.target("http://localhost:8080");
restTarget.path("/RestServiceProjektname/webapi/Servicename/getMethode").queryParam("tvname",
", "Samsung").request().get(String.class);

```

Listing 20: GET-Anfrage mit Jax-RS

Eine POST-Nachricht funktioniert nach dem gleichen Prinzip. Auch hier wird ein Client erstellt und ein Target gebaut. Ihm wird die URI übergeben. Somit ist folgender Teil deckungsgleich:

```

Client restClient = ClientBuilder.newClient();
WebTarget restTarget = restClient.target("http://localhost:8080");
restTarget.path("/RestServiceProjektname/webapi/Servicename/postMethode")

```

Nun muss die POST-Nachricht erstellt werden. Dazu wird der Request-Methode zunächst der Mediatype der Daten, die sich in der POST-Nachricht befinden, übergeben.

```
.request(MediaType.APPLICATION_JSON)
```

Daraufhin muss das Objekt, das versendet werden soll, mit der Klasse Entity umgewandelt werden. Sie übernimmt dabei das Marshalling in den jeweiligen Mediatype. Dafür muss die Klasse Entity diesen mitgeteilt bekommen, also wird dem Konstruktor, wie unten zu sehen, das Objekt und der Mediatype übergeben.

```
Entity.entity(lcdTv, MediaType.APPLICATION_JSON)
```

Dieses Entity-Objekt kann nun der POST-Methode von Jax-RS übergeben werden. Darüber hinaus muss der Rückgabetype der Response als zweites Argument zusätzlich übergeben werden. Dies geschieht durch übergeben der Class Variable der Java-Klasse. In diesem Fall wird ein String als Response akzeptiert.

```
.post(Entity.entity(lcdTv, MediaType.APPLICATION_JSON), String.class);
```

Die komplette POST-Anfrage ist in Listing 21 nochmals dargestellt.

```

Client restClient = ClientBuilder.newClient();
WebTarget restTarget = restClient.target("http://localhost:8080");
restTarget.path("/RestServiceProjektname/webapi/Servicename/postMethode").request(MediaType.APPLICATION_JSON).post(Entity.entity(lcdTv, MediaType.APPLICATION_JSON), String.class);

```

Listing 21: POST-Anfrage mit Jax-RS

2.5. SOAP - Simple Object Access Protocol

SOAP wurde von Dave Winer und Microsoft entwickelt und 1999 in der Version 1.0 veröffentlicht. Im Jahr 2000 wurde SOAP in der Version 1.1 dem W3C zur Übernahme als Spezifikation eingereicht [19]. Das Ergebnis dieser Einreichung war die SOAP Version 1.2, welche 2003 als Empfehlung des W3C in das Basic Profile übernommen wurde [20]. Die Spezifikation von SOAP ist auf der W3C-Seite zu finden [21]. Laut dieser Spezifikation ist SOAP ein leichtgewichtiges Protokoll zum Austausch strukturierter Daten in dezentral verteilten Umgebungen. SOAP verwendet zur Übertragung HTTP 1.1.

2.5.1. Jax-WS

Ähnlich wie in REST werden bei SOAP ebenfalls die Java-Klassen annotiert, die als Webservice zur Verfügung gestellt werden sollen. Allerdings wird bei SOAP erst ein Interface annotiert, welches später in einer richtigen Klasse implementiert wird. Mit der Annotation @WebService wird Java übermittelt, dass es sich bei dieser Klasse um einen SOAP-Webservice handelt. Alle in dem Interface befindlichen public-Methoden sind nach der Annotation automatisch Webservice-Methoden des Webservices. In Listing 22 ist ein Beispielcode abgebildet.

```
@WebService  
public interface BestellungInterface {  
...  
}
```

Listing 22: SOAP-Annotation – Interface

Das Interface muss nun implementiert werden, damit der Service auch Funktionen ausführen kann. Der Implementationsklasse muss noch weitere Information per Annotation übergeben werden: zum Einen über den Servicenamen und zum Anderen das EndpointInterface. Der Servicename kann nach Belieben vergeben werden. Das EndpointInterface muss auf die Klasse verweisen, welche das Interface beinhaltet. Beide Parameter werden der Annotation @WebService in der Implementationsklasse übergeben. In Listing 23 ist ein Beispiel zu sehen.

```
@WebService(endpointInterface = "BABeispiel.SOAPServices.BestellungInterface", serviceName = "BestellungService")  
public class Bestellung implements BestellungInterface {  
...  
}
```

Listing 23: SOAP-Annotation – Implementationsklasse

2.5.2. Skeleton erstellen

Um diese Klasse nun deployen zu können, wird das Java-Tool „wsgen.exe“ verwendet. Dazu muss zunächst das Projekt kompiliert werden. Die Class-Dateien, die sich nach dem Kompilieren im Target-Ordner befinden, können dann zum Skeleton umgewandelt werden. Die Skeleton-Dateien müssen anschließend in den Projektordner importiert werden, damit die Webservices erreichbar werden. Ob der Service erreichbar ist, kann durch das Öffnen der URI, unter der die WSDL-Datei erreichbar ist, getestet werden (Kap. 2.3.2.1).

2.5.3. Stubs aus WSDL-Dateien erstellen

Zum Erstellen der Stubs muss das Java-Tool „wsimport.exe“ verwendet werden. Dem Tool muss die URI der WSDL-Datei übergeben werden. Es kann auch ein Pfad von einer lokal gespeicherten Datei angegeben werden. Daraufhin werden Java-Klassen aus dieser WSDL-Datei generiert. Die generierten Dateien müssen anschließend in das Projekt importiert werden.

Um nun mit den generierten Stubs einen RPC auf einen SOAP-Webservice ausführen zu können, muss ein Serviceport auf dem Client generiert werden. Dazu wird zuerst ein Objekt der Klasse URL erstellt. Die Klasse URL bekommt in den Konstruktor die URI oder den Pfad von einer WSDL-Datei, welcher auch oben zum Generieren der Stubs verwendet wurde, als String übergeben. Also z. B.:

```
URL soapserviceURLPersistenz = new URL(SOAPBestellServiceURL);
```

Als Nächstes benötigen wir ein Objekt der Klasse QName, welches einen qualified name einer XML-Datei darstellt (siehe Kap. 2.3.1.3). Ihm wird der TargetNamespace und der Servicename aus der WSDL-Datei der gewünschten Klasse übergeben.

```
QName soapQname = new QName(ServiceDataURL, SOAPBestellServiceName);
```

Aus diesen 2 Objekten wird nun ein Service generiert:

```
Service soapService = Service.create(soapserviceURLPersistenz, soapQname);
```

Mit diesem Service kann nun ein Port aus dem Stub generiert werden.

```
Stub.BestellInterface BestellPort = soapService.getPort(Stub.BestellInterface.class);
```

Mit dem Port kann nun wie mit einem normalen Java-Objekt gearbeitet werden. Der Rückgabewert entspricht hier der Response des Clients.

```
response = bestellPort.setArtikel(tvGeraet.getId());
```

In Listing 24 nochmals zur Übersichtlichkeit der ganze Code am Stück.

```
URL soapserviceURLPersistenz = new URL(SOAPBestellServiceURL);
QName soapQname = new QName(ServiceDataURL, SOAPBestellServiceName);
Service soapService = Service.create(soapserviceURLPersistenz, soapQname);
Stub.BestellInterface BestellPort = soapService.getPort(Stub.BestellInterface.class);
response = bestellPort.setArtikel(tvGeraet.getId());
```

Listing 24: SOAP-Service aufrufen

2.6. gRPC – gRPC Remote Procedure Call

gRPC ist genau genommen ein Framework und wurde ursprünglich von Google entwickelt [22]. Es ist der Nachfolger des Stubby-Protokolls, welches von Google intern verwendet wurde. 2015 entschied sich Google Stubby als Open-Source-Projekt weiterzuentwickeln. Daraus entstand das gRPC Projekt. gRPC ist ein bi-direktionales Streaming-Protokoll, das auf HTTP/2 basiert. Es wurde speziell auf hohe Effizienz und niedrige Latenz optimiert. Es nutzt die IDL und den Marshalling-Mechanismus von Protobuf (Kap. 2.3.1.4 und 2.3.2.2). Da gRPC kein Standard einer offiziellen Institution ist, ist die Spezifikation auf der gRPC.io Seite zu finden [23].

2.6.1. Skeleton aus der Proto-Datei erstellen

Zum Erstellen eines RPC-Services, muss bei gRPC zuerst die IDL geschrieben werden. Für dieses Beispiel wird die Proto-Datei aus Listing 25 verwendet. Diese besitzt 2 Messages und einen Service mit einer Methode.

```

...
message LCDBildschirm{
    int32 id = 1;
    string marke = 2;
    bool vierk = 3;
    repeated string zubehoer = 4;
}

message StringMsg {
    string message = 1;
}

service Bestellung {
    rpc neueBestellung(LCDBildschirm) returns (StringMsg) {}
}

```

Listing 25: Proto-Datei für gRPC-Bespiel

Um aus der Proto-Datei Skeleton und Stubs generieren zu können, muss sie mit dem „protoc.exe“-Tool kompiliert werden. gRPC bietet das protoc-Tool auch als Maven-Plug-in an. Die ArtifactID des Plug-ins ist `protobuf-maven-plugin`. Kompiliert werden kann nun mit `mvn protobuf:compile`. Nach dem Kompilieren sind die neuen Klassen im Target-Verzeichnis des Projekts zu finden. Diese sind so direkt verwendbar, da der Pfad des Target-Verzeichnisses automatisch durch gRPC dem Classpath des Projekts hinzugefügt wurde. Die Skeleton-Klassen sind abstrakte Klassen. Zum Implementieren muss diese abstrakte Klasse mit „`extends`“ eingebunden werden. Die einzubindende Klasse besitzt die Endung „`-ImplBase`“. In diesem Beispiel also:

```
public class Bestellung extends BestellungGrpc.BestellungImplBase {
```

Nun sind die vorher in der Proto-Datei definierten Methoden des Service in unserer Implementierungsklasse auch verfügbar und müssen mit der Annotation `@Override` überschrieben werden. Sie verfügen über einen zusätzlichen Übergabeparameter `responseObserver`, der dazu dient eine Response zurückzusenden. So kann gRPC auch asynchron Daten übertragen.

```
@Override
public void neueBestellung(grpcServices.LCDBildschirm grpcTV,
io.grpc.stub.StreamObserver<grpcServices.StringMsg> responseObserver) {
```

Nachdem alle Methoden des Skeletons überschrieben wurden, kann diese Klasse mit einem Server-Builder deployt werden. Dazu wird einem Server-Builder der Port übergeben, auf dem er lauschen soll.

```
Server server = ServerBuilder.forPort(51001)
```

Anschließend muss der für den Server gewünschte Service hinzugefügt werden. Dies geschieht mit der Methode `addService`. Sie bekommt als Übergabeparameter ein neues Objekt der Klasse des Skeletons. Hier also:

```
.addService(new Bestellung())
```

Da bei gRPC alle Messages, Server und sonstige Objekte mit Builder generiert werden, muss der Server nun noch abschließend mit dem Aufruf der Methode `Build()` fertig gebaut werden.

```
.build()
```

Zum Starten des Servers muss noch die Methode `Start` aufgerufen werden.

```
.start();
```

Damit der Server sich nicht nach einer Verbindung wieder schließt, muss man ihm durch eine weitere Methode mitteilen, dass er so lange laufen soll, bis er einen Terminierungsbefehl erhält. Dies geschieht mit der Methode `awaitTermination`.

```
server.awaitTermination();
```

Nun ist der Service auf dem Port zu erreichen. Der vollständige Code ist in Listing 26 abgebildet.

```
Server server = ServerBuilder.forPort(51001).addService(new Bestellung()).start();
server.awaitTermination();
```

Listing 26: gRPC-Server starten und Service hinzufügen

2.6.2. Stub-Generierung mit Maven

Die Stub-Generierung funktioniert mit Maven ebenso wie das Erstellen des Skeletons. Die Stubs werden also mit `mvn protobuf:compile` erstellt. In den Output-Dateien sind die Stubs als `BestellungStub` (kann alle Arten der Verbindungen), `BestellungBlockingStub` (Einzel- und Streaming-Output-Blocking-Calls) und als `BestellungFutureStub` (Einzel- und Streaming-Output-nonblocking-Calls) enthalten. Je nach Anwendung muss die entsprechende Stub-Art gewählt werden. In diesem Beispiel wird der `BestellungBlockingStub` verwendet, da dieser auch im Benchmark zum Einsatz kommt.

Um nun Verbindung zu einem gRPC-Server aufnehmen zu können, muss zuerst ein `ManagedChannel`-Objekt erstellt werden. Dieser Channel bekommt die Adresse und den Port übergeben.

```
ManagedChannel Channel = ManagedChannelBuilder.forName("127.0.0.1", 51001)
```

Dieser wird aus Debug-Gründen auf Plaintext umgestellt.

```
.usePlaintext(true)
```

Das Channel-Objekt muss nun abschließend gebaut werden, damit es benutzt werden kann.

```
.build();
```

Mit dem Channel kann nun ein Blockingstub erstellt werden. Dazu gibt es eine Methode `newBlockingStub`, welche den Channel als Argument übergeben bekommt.

```
BestellungGrpc.BestellungBlockingStub bbs = BestellungGrpc.newBlockingStub(Channel);
```

Mit dem Stub kann nun wie mit einem normalen Java-Objekt gearbeitet werden. Es besitzt alle Methoden, die im Skeleton verfügbar sind. Unter gRPC sind allerdings die Messages, die zwischen Server und Client versendet werden, mit Builder zu bauen. Um nun eine Message bauen zu können, hat die Message-Klasse eine Methode `newBuilder`. Sie gibt ein Builder-Objekt zurück.

```
LCDBildschirm tv = LCDBildschirm.newBuilder();
```

In diesem Builder-Objekt können die Variablen mit Werten gesetzt werden.

```
tv.setName("Samsung");
```

Danach ist die Message bereit, gesendet zu werden. Sie wird, wie in Java bei Übergabeparametern üblich, der Methode des Stubs übergeben. Das Marshalling und Senden wird von gRPC übernommen. Rückgabewerte sind auch Messages, daher ist der Rückgabewert immer eine Message, die in der Proto-Datei definiert wurde. In diesem Beispiel wird ein String in einer Message zurückgegeben.

```
StringMsg sm = bbs.neueBestellung(tv);
```

Sobald die Verbindung nicht mehr benötigt wird, kann sie durch die Methode shutdown geschlossen werden.

```
Channel.shutdown();
```

In Listing 27 nun zusammenfassend der ganze Code am Stück:

```
ManagedChannel Channel = ManagedChannelBuilder.forAddress("127.0.0.1", 51001)
    .usePlaintext(true).build();
BestellungGrpc.BestellungBlockingStub bbs = BestellungGrpc.newBlockingStub(Channel);
LCDBildschirm tv = LCDBildschirm.newBuilder();
tv.setName("Samsung");
StringMsg sm = bbs.neueBestellung(tv);
Channel.shutdown();
```

Listing 27: gRPC-Client-Verbindung zum Server

3. Verwandte Arbeiten

Google hat für die Entwicklung von gRPC ein Benchmark-System eingeführt [24]. Dieses testet die Latenz für Verbindungsaufrufe. Der Test erinnert dabei an eine Art Ping-Pong-Spiel. Es wird ein Request gesendet und die Reaktionszeit gemessen, die benötigt wird, bis die Response ankommt. Zusätzlich kann die Message-Rate pro Sekunde gemessen werden, wenn 2 Clients mit 64 Kanälen senden. Weiterhin kann die Anzahl der Messages pro Server-CPU-Kern gemessen werden, um die Hardware-Auslastung zu testen.

Da gRPC auf geringe Latenz und hohe Performance optimiert wird, gehören diese Tests zum Entwicklungsprozess und werden täglich durchgeführt. So kann bei einem Update direkt erkannt werden, wenn dieses fehlerhaft war. So kann das Update vor dem Veröffentlichen nochmals überarbeitet werden.

Die Ergebnisse dieser Tests werden auf einem Dashboard für die Öffentlichkeit zur Verfügung gestellt. Dabei sind Latenzen in Java von etwa 200µs gelistet [25]. In einem Video auf grpc.io wird über die Optimierungsfortschritte gesprochen, welche im April 2016 einen Stand von 350.000 streaming Messages/Sek auf einer 32 Kern-CPU vorweisen konten. Googles Ziel war es, 100.000/Sek./CPU-Kern zu erreichen. Im Juni 2016 haben sie bereits 2.000.000/Sek. auf 32 CPU-Kernen geschafft. Das entspricht 62.500/Sek./CPU-Kern.

Ein weiterer Benchmark zu gRPC wurde von der GopherAcademy für etcd erstellt [26]. Etcd ist ein Key-Value-Store für Microservices. Das Benchmark-Szenario wurde auf „wie lange benötigt ein Testlauf für das Speichern von 300.000 Key/Value-Paaren“ festgelegt. Verglichen werden JSONRPC mit gRPC. Während JSONRPC 8m7,270s benötigte, schloss gRPC den Test schon nach 36,715s ab.

Ein wohl privates Benchmark-Projekt wurde von Huso Bee geschrieben [27]. Es vergleicht, wie viele Operationen REST im Gegensatz zu gRPC schafft. In einer Sekunde schaffte gRPC 5000 HTTP-Posts innerhalb einer Sekunde. Das entspricht 224633 ns/Operation. REST hingegen schaffte nur 200 HTTP-Posts und benötigte 5748596 ns/Operation. REST ist also etwa um das 25-fache langsamer.

4. Architektur des Benchmark-Systems

Der Benchmark besteht aus 3 Microservices:

- einem Persistenz-Service zur Kommunikation mit der SQL-Datenbank
- einem Transformator-Service, der über eine REST-Schnittstelle zur Kommunikation mit dem Benutzer verfügt
- und einem UmdrehenMS-Service, der bei Bedarf hinzugeschaltet werden kann.

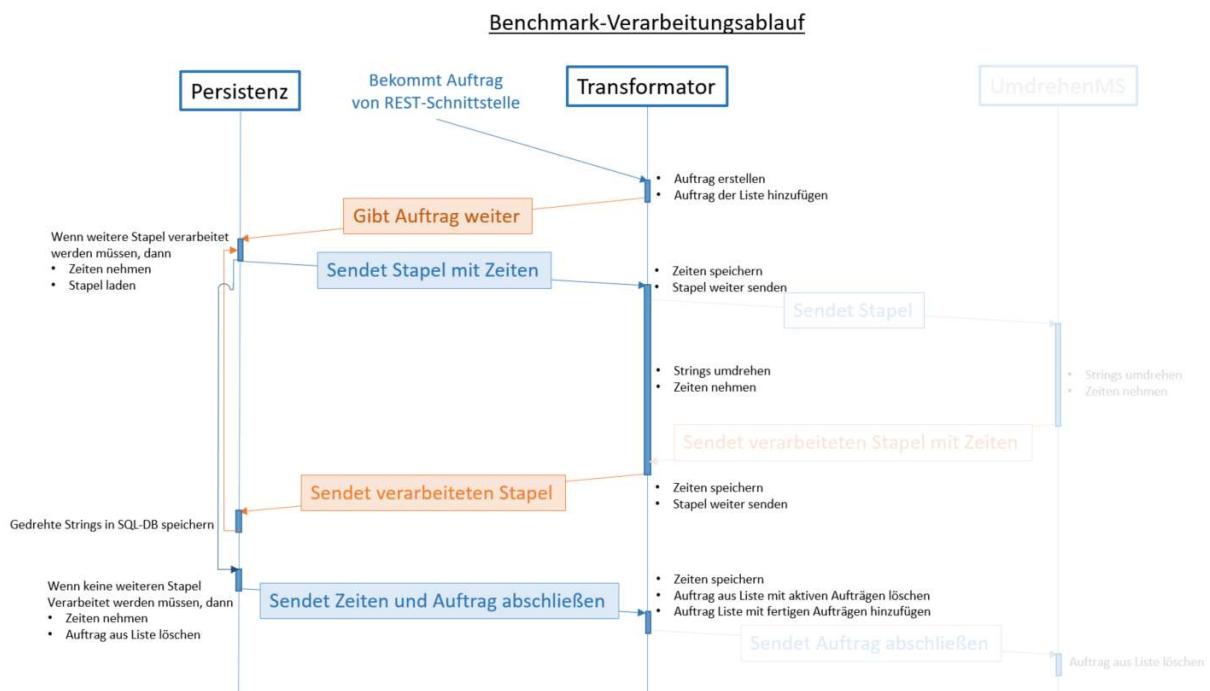


Abbildung 10: Benchmark-Verarbeitungsablauf ohne UmdrehenMS-Service

Wenn ohne den UmdrehenMS-Service ein Benchmark-Lauf gestartet wird, dann werden nur 2 Services verwendet: der Transformator-Service und der Persistenz-Service. Dieser Vorgang wird in Abbildung 10 dargestellt. Als Erstes erhält der Transformator-Service über die REST-Schnittstelle vom Benutzer einen Auftrag für einen Benchmark-Durchgang. Der Transformator speichert sich diesen in einer globalen Liste und gibt ihn an den Persistenz-Service weiter. Der Persistenz-Service sendet nun entsprechend den Einstellungen des Auftrags stapelweise Strings an den Transformator-Service. Dieser dreht die Strings um und sendet sie wieder zurück an den Persistenz-Service. Der Persistenz-Service speichert die Strings in der SQL-Datenbank ab und sendet, falls noch weitere Strings gesendet werden müssen, einen weiteren Stapel an den Transformator-Service. Sobald der Auftrag erledigt ist, wird vom Persistenz-Service, eine „Auftrag abschließen“-Nachricht an den Transformator gesendet. Dieser weiß dann, dass der Auftrag bearbeitet wurde, und verschiebt den Auftrag von der Liste der aktiven zur Liste der fertig bearbeiteten Aufträge.

Wenn ein Auftrag für einen Benchmark-Lauf mit hinzugeschaltetem UmdrehenMS-Service erstellt wird, arbeiten alle 3 Microservices (Abbildung 11). Wie oben wird der Auftrag über die REST-

Schnittstelle an den Transformator gesendet. Dieser leitet ihn weiter an den Persistenz-Service, welcher Strings aus der SQL-Datenbank holt und stapelweise an den Transformator-Service sendet. Der Transformator-Service dient diesmal als Vermittler und reicht die Stapel weiter an den UmdrehenMS-Service. Der UmdrehenMS-Service dreht die Stapel um und gibt die verarbeiteten Stapel zurück an den Transformator-Service. Der Transformator-Service spielt nochmals den Pivot und gibt die Stapel an den Persistenz-Service weiter. Anschließend kann der Persistenz-Service die Strings des Stapels in die SQL-Datenbank speichern. Danach prüft der Persistenz-Service, ob für den Auftrag weitere Strings gedreht werden sollen. Wenn ja, sendet er einen weiteren Stapel an den Transformator. Wenn nein, sendet er eine „Auftrag abschließen“-Nachricht an den Transformator, welcher wie oben die abschließenden Aufgaben erledigt, wie den Auftrag aus der aktiven Auftragsliste löschen und ihn der Liste der fertig bearbeiteten Aufträge hinzuzufügen. Zusätzlich sendet der Transformator an den UmdrehenMS eine weitere „Auftrag abschließen“-Nachricht, damit dieser den Auftrag aus seiner Liste löscht, um den Speicher wieder freizugeben.

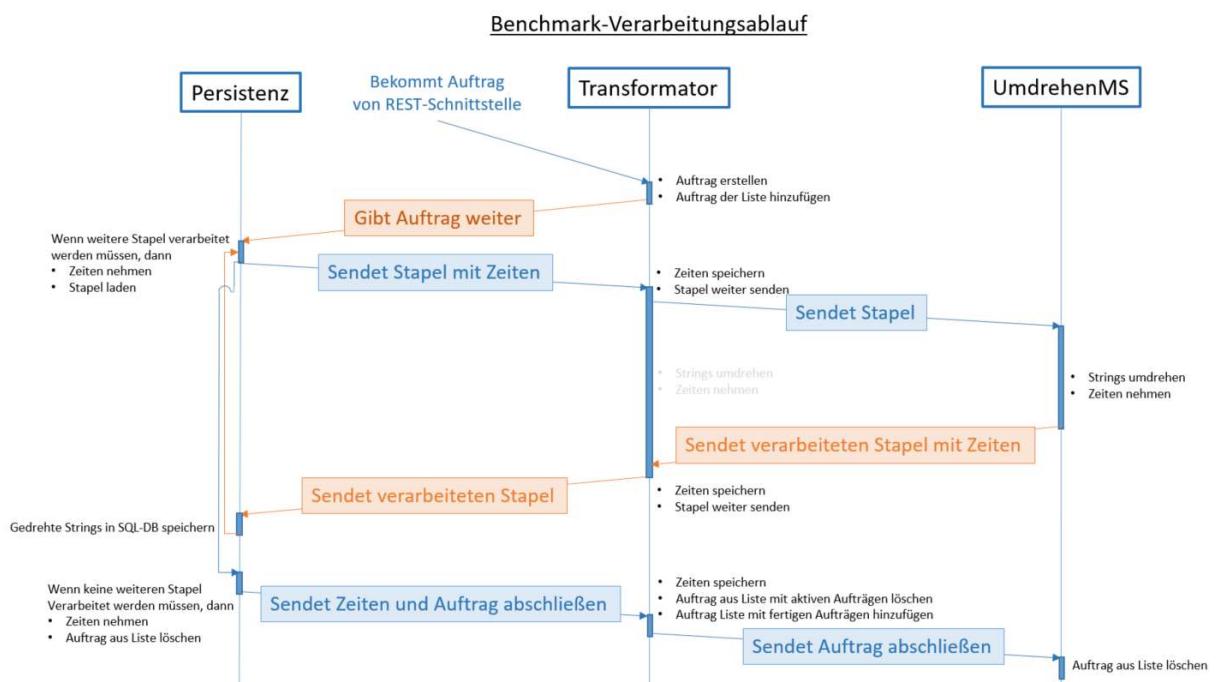


Abbildung 11: Benchmark-Verarbeitungsablauf mit UmdrehenMS-Service

Um die Zeiten für die Benchmark-Ergebnisse zu bekommen, wurde an 6 Stellen der Zeitstempel in Nanosekunden genommen (orange in Abbildung 11 markiert). Ein Zeitstempel in Millisekunden ist für diesen Benchmark nicht ausreichend, da gRPC weniger als 1 Millisekunde zum Senden eines Stapels benötigt. Die Microservices wurden zur Übersichtlichkeit in den Ergebnisdiagrammen abgekürzt: der Persistenz-Service mit SQL, der Transformator-Service mit Trans und der UmdrehenMS-Service mit Umdreh.

- Durch die Messpunkte 1 und 2 ergeben sich die Zeiten von SQL -> Trans
- Durch die Messpunkte 2 und 3 von Trans -> Umdreh
- Durch die Messpunkte 4 und 5 von Umdreh -> Trans
- Durch die Messpunkte 5 und 6 von Trans -> SQL

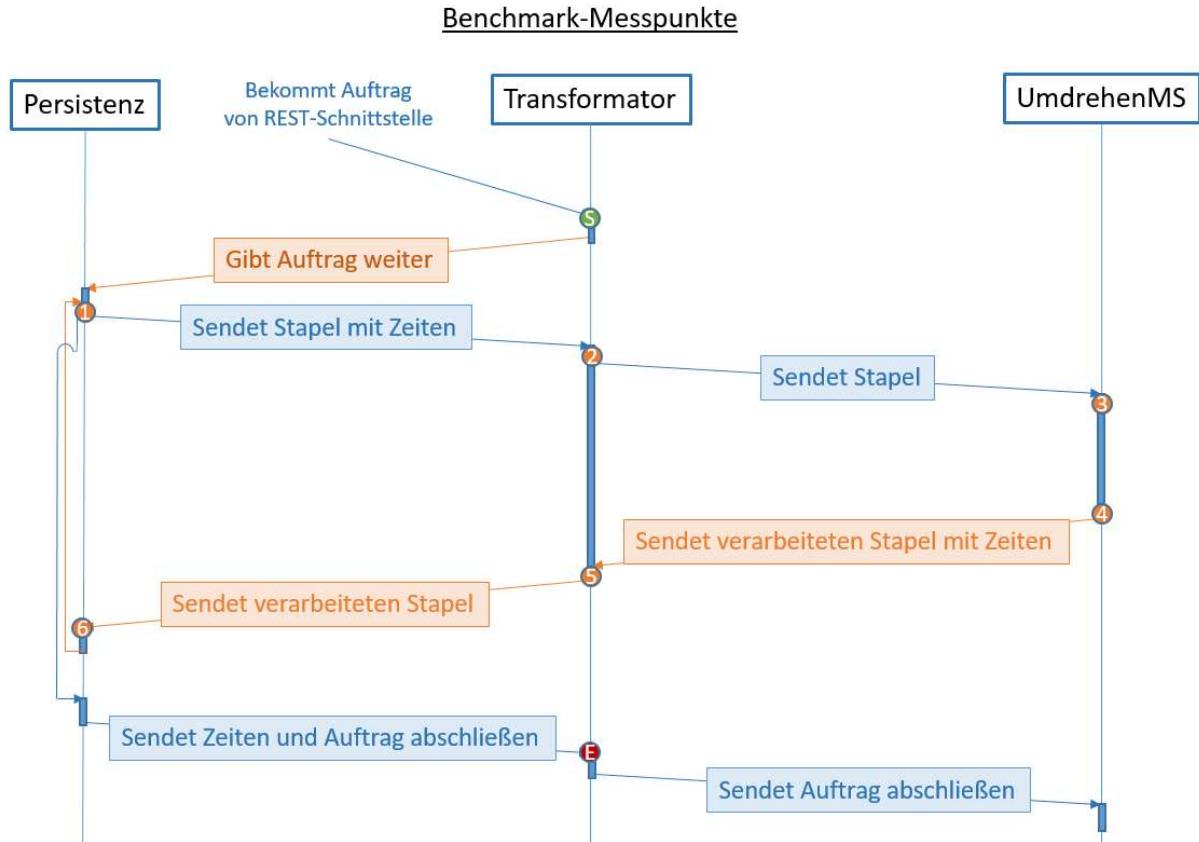


Abbildung 12: Messpunkte des Benchmarks

Es wird also immer direkt vor dem Senden und direkt nach dem Empfang ein Zeitstempel genommen. Aus der Differenz berechnet sich die gemessene Zeitdauer. Dabei wird der spätere Zeitstempel (Empfangszeitstempel) vom früheren (Sendezzeitstempel) abgezogen. Für die Gesamtzeit werden die Differenzen aller gemessenen Zeitstempel aufsummiert. Also errechnet sich die Gesamtzeit folgendermaßen:

$$\sum_{k=0}^{n-1} \text{Empfangszeitstempel}_k - \text{Sendezzeitstempel}_k$$

Zusätzlich zur Gesamtzeit wird noch der Durchschnitt für das Senden eines Stacks berechnet. Dabei ist n die Anzahl der Stacks, die verarbeitet wurden. Für n gilt $n \in \mathbb{N}$ und $n \leq 1.000.000$, da bei einer Stackgröße von 1 jeder String einzeln übertragen wird und sich 1.000.000 Strings in der SQL-Datenbank befinden. Der Durchschnitt berechnet sich wie folgt:

$$\frac{\sum_{k=0}^{n-1} \text{Empfangszeitstempel}_k - \text{Sendezzeitstempel}_k}{n - 1}$$

Die Durchschnittszeit für die Verarbeitung von 1000 Datensätzen ergibt sich aus folgender Formel:

$$\frac{\text{Zeitstempel } 6_{n-1} - \text{Zeitstempel } 1_0}{\frac{n - 1}{1000}}$$

Zeitstempel 6 ist dabei der in Abbildung 12 dargestellte Messpunkt 6. Von diesem wird der letzte, also der $n-1$ -te, gespeicherte Zeitstempel für die Berechnung verwendet. Hier gilt ebenso $n \in \mathbb{N}$ und

$n \leq 1.000.000$. Zeitstempel 1 ist der in der Abbildung 12 dargestellte Messpunkt 1. Von diesem wird der erste genommene Zeitstempel für die Berechnung verwendet. Diese werden dann durch das Tausendstel der Gesamtanzahl geteilt. Somit ergibt sich die Durchschnittszeit für die Verarbeitung von 1000 Datensätzen. Dadurch müssen nicht alle Zeitstempel einzeln aufaddiert werden, was Rechenzeit spart.

Zusätzlich wird zu den Sende- und Empfangszeiten noch die Startzeit (grün in Abbildung 12 markiert) und die Endzeit (rot in Abbildung 12 markiert) gespeichert. Diese werden zur anschaulichen Darstellung für den Nutzer in Strings in Millisekunden gespeichert. Die Java-Klasse SimpleDateFormat kann ausschließlich Zeiten in Millisekunden umwandeln. Durch die Verrechnung von *Zeitstempel Endzeit – Zeitstempel Startzeit* ergibt sich die Gesamtzeit, die für die Abarbeitung des Auftrags benötigt wurde.

5. Ergebnisse

In Tabelle 2 wurden die 3 verwendeten Protokolle nach für Entwickler wichtigen Merkmalen bewertet (grün bedeutet gut, gelb mittel und rot schlecht). Die Buchstaben haben folgende Bedeutung:

- a = Implementierung (Erstellung von Stub/Skeleton)
- b = Aufwand einen Service anderen Nutzern zur Verfügung zu stellen
- c = Menge des generierten Overheads
- d = Streaming-fähig
- e = Kann asynchrone Kommunikation

	a	b	c	d	e
REST	grün	grün	rot	rot	rot
SOAP	rot	grün	rot	rot	rot
gRPC	rot	rot	grün	grün	grün

Tabelle 2: Technikvergleich der Protokolle

Abbildung 13 zeigt die Größe des Beispielobjekts aus Kapitel 2.3.1.1 (S. 15) in Byte in den verschiedenen Mediatypes. Auffällig ist hierbei, dass XML mit großem Abstand den meisten Speicher für das Beispielobjekt benötigt. Auf Platz 2 kommt JSON und am wenigsten Platz benötigt Protobuf. Dies ist ausschlaggebend für die Resultate der Benchmark-Ergebnisse und sollte bereits zu diesem Zeitpunkt zu erkennen geben, welches der Protokolle am schnellsten die Daten verarbeitet. Für die Testläufe wird die Stapelgröße schrittweise erhöht, um zu sehen, ab welcher Größe der Overhead vernachlässigbar wird.

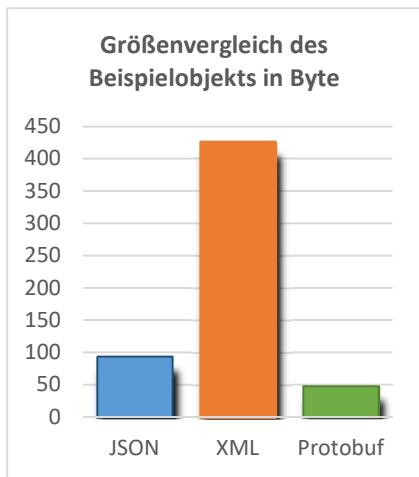


Abbildung 13: Größenvergleich des Beispielobjekts

Die Testreihen setzen sich sortiert nach Stapelgröße folgendermaßen zusammen:

Stapelgröße	String Anzahl	
1	30.000	ohne UmdrehenMS
1	30.000	mit UmdrehenMS
2	60.000	ohne UmdrehenMS
2	60.000	mit UmdrehenMS
5	125.000	ohne UmdrehenMS
5	125.000	mit UmdrehenMS
10	250.000	ohne UmdrehenMS
10	250.000	mit UmdrehenMS
50	500.000	ohne UmdrehenMS
50	500.000	mit UmdrehenMS
100	1.000.000	ohne UmdrehenMS
100	1.000.000	mit UmdrehenMS
1.000	1.000.000	ohne UmdrehenMS
1.000	1.000.000	mit UmdrehenMS
10.000	1.000.000	ohne UmdrehenMS
10.000	1.000.000	mit UmdrehenMS
100.000	1.000.000	ohne UmdrehenMS
100.000	1.000.000	mit UmdrehenMS

Tabelle 3: Zusammensetzung der Testreihen für die Benchmark-Durchläufe

Als Testsystem wird ein Lenovo-Laptop y70-50 mit Intel I7 4720-HQ (4 x 2,6 GHz) 8 GB RAM, 265 GB SSD und 1 Gb/s Ethernet-Karte verwendet. Die Tests laufen über die Localhost-Loopback-Schnittstelle. Deployt werden die Microservices in Eclipse über einen Wildfly-Webserver.

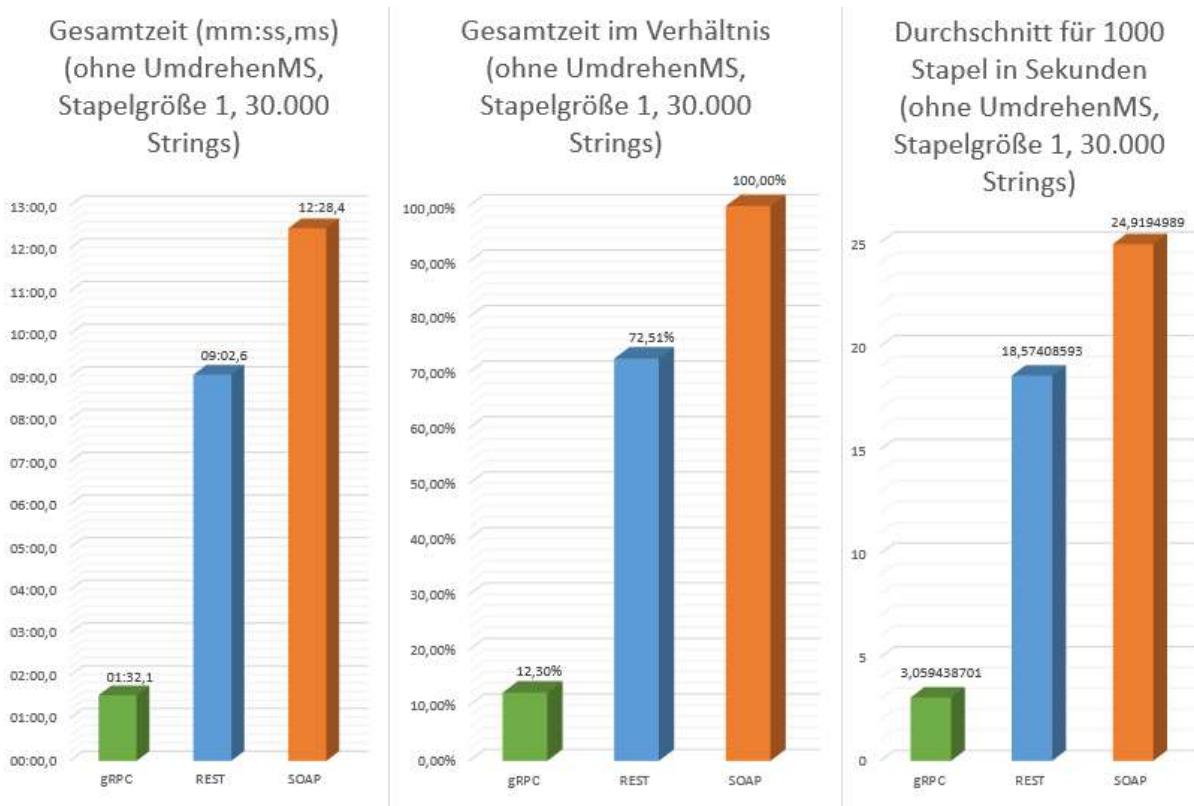


Abbildung 14: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 1, 30.000 Strings

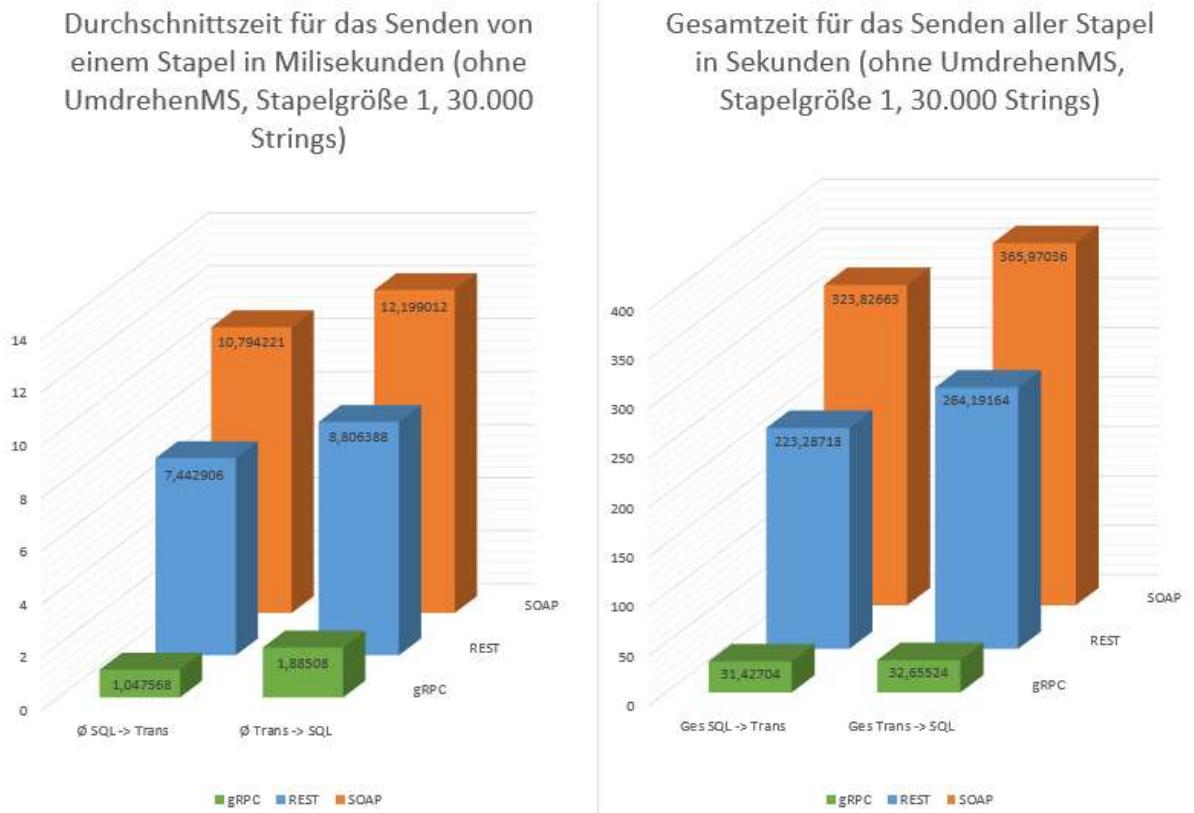


Abbildung 15: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 1, 30.000 Strings

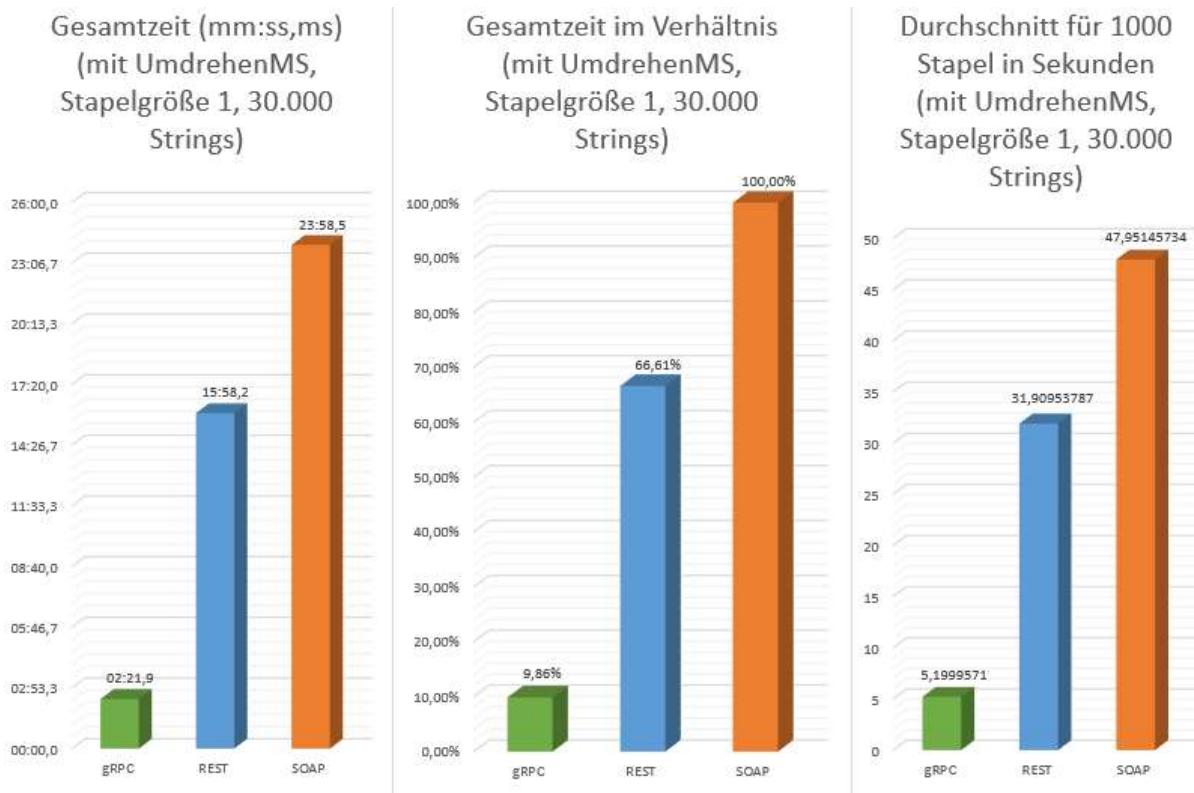


Abbildung 16: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 1, 30.000 Strings

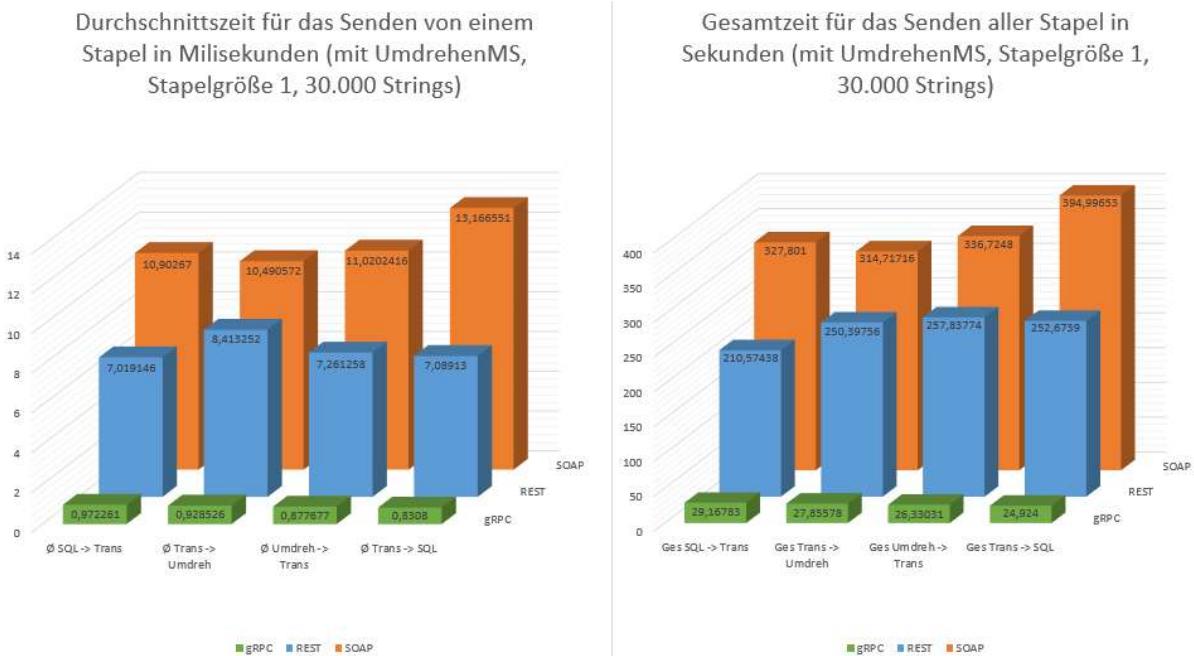


Abbildung 17: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 1, 30.000 Strings

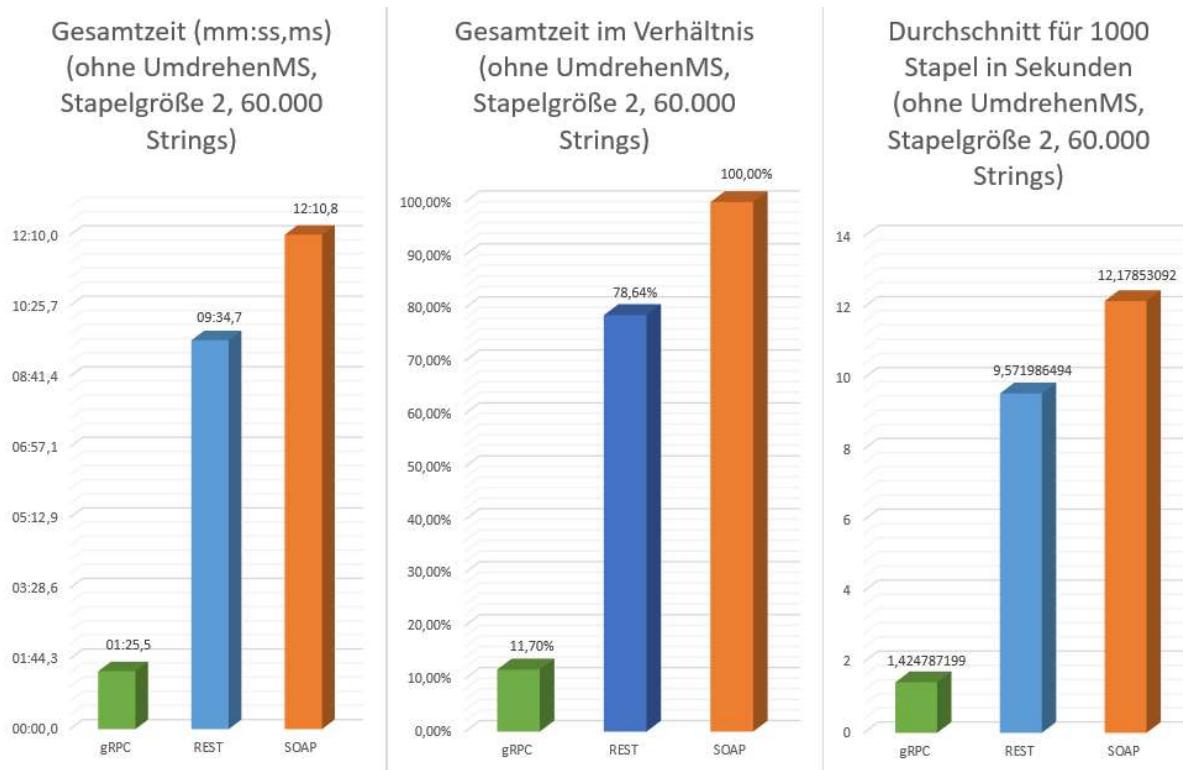


Abbildung 18: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 2, 60.000 Strings

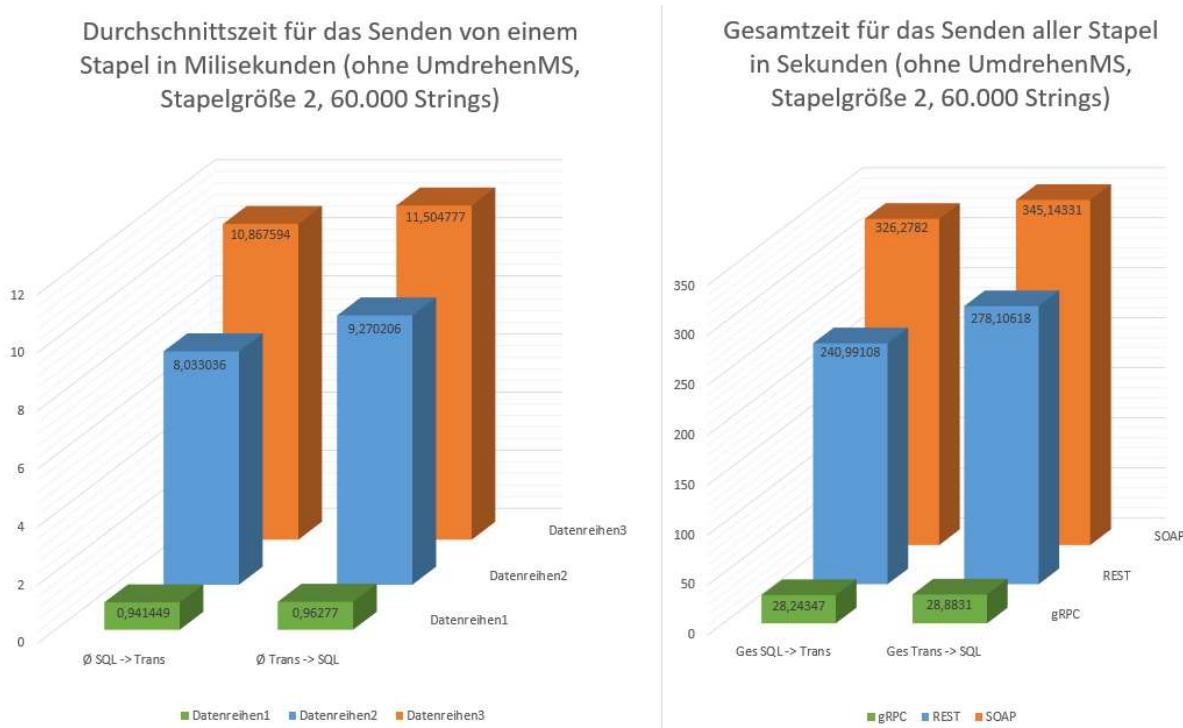


Abbildung 19: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 2, 60.000 Strings

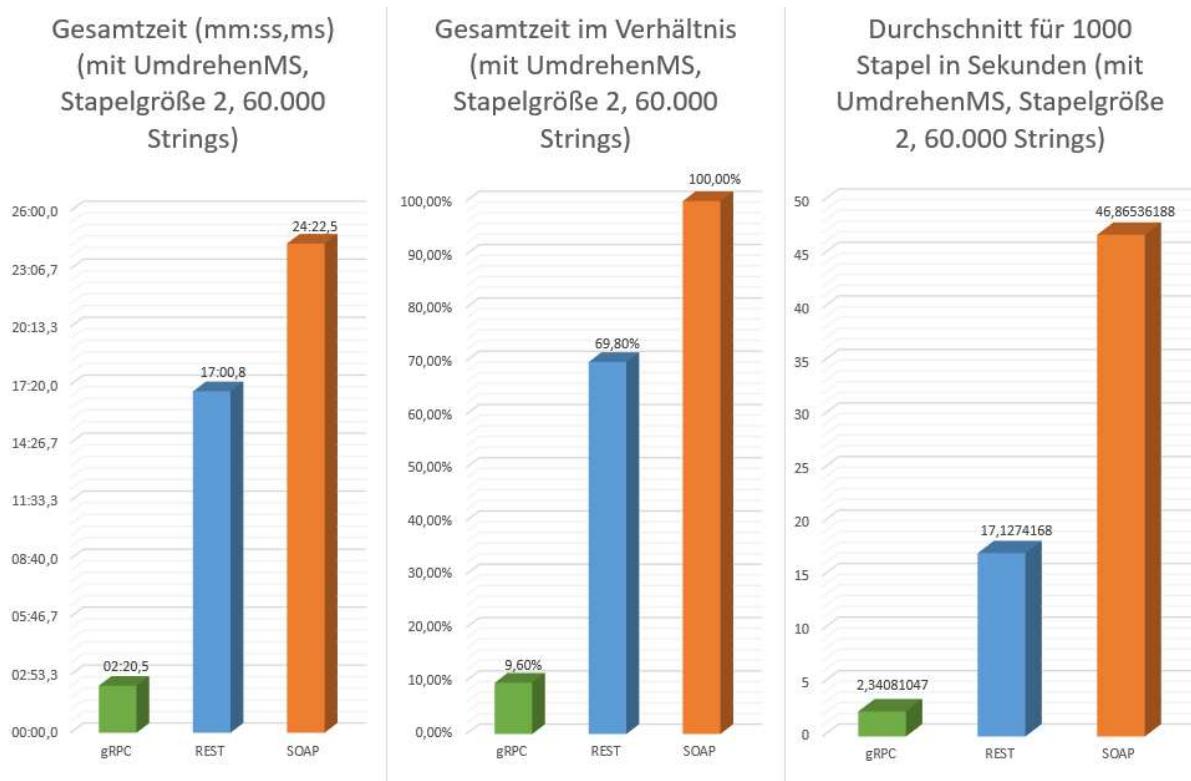


Abbildung 20: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 2, 60.000 Strings

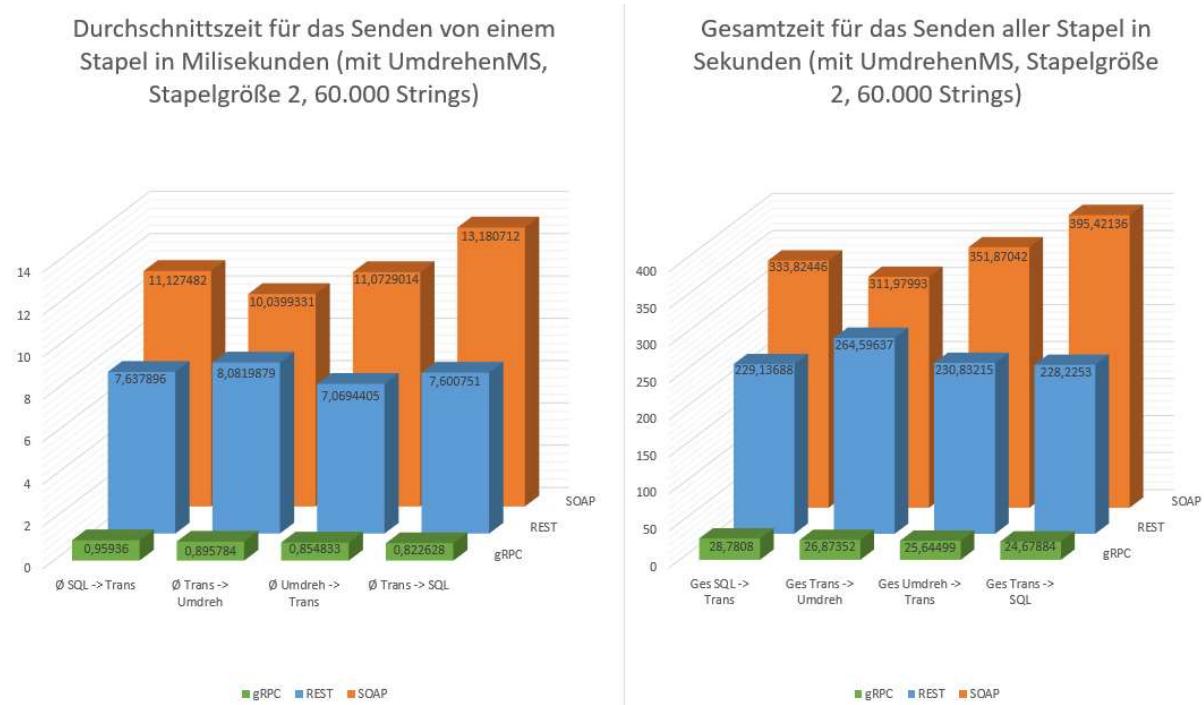


Abbildung 21: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 2, 60.000 Strings

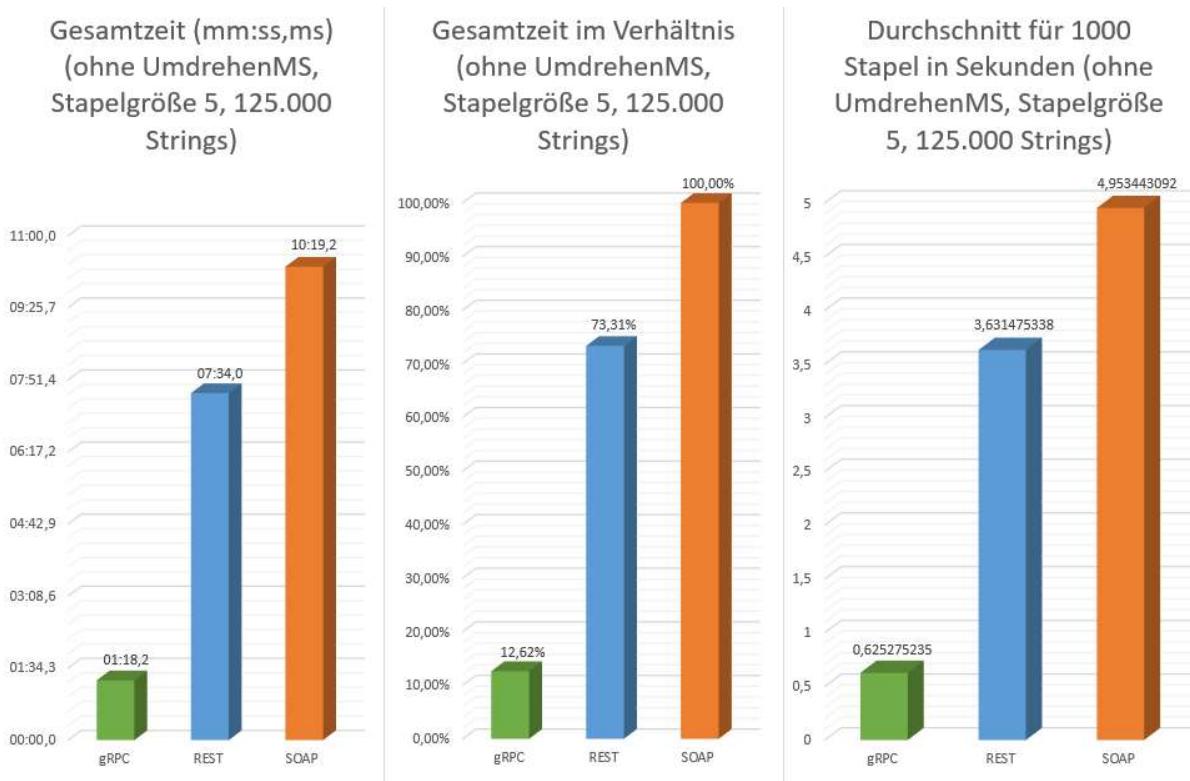


Abbildung 22: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 5, 125.000 Strings

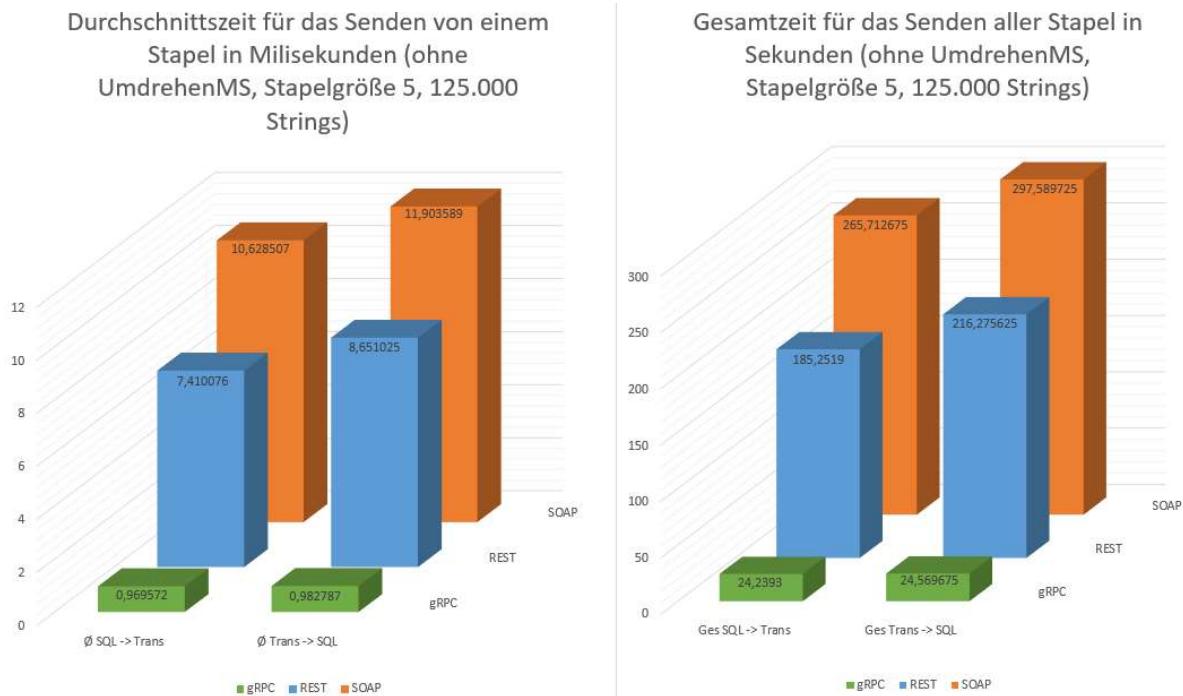


Abbildung 23: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 5, 125.000 Strings

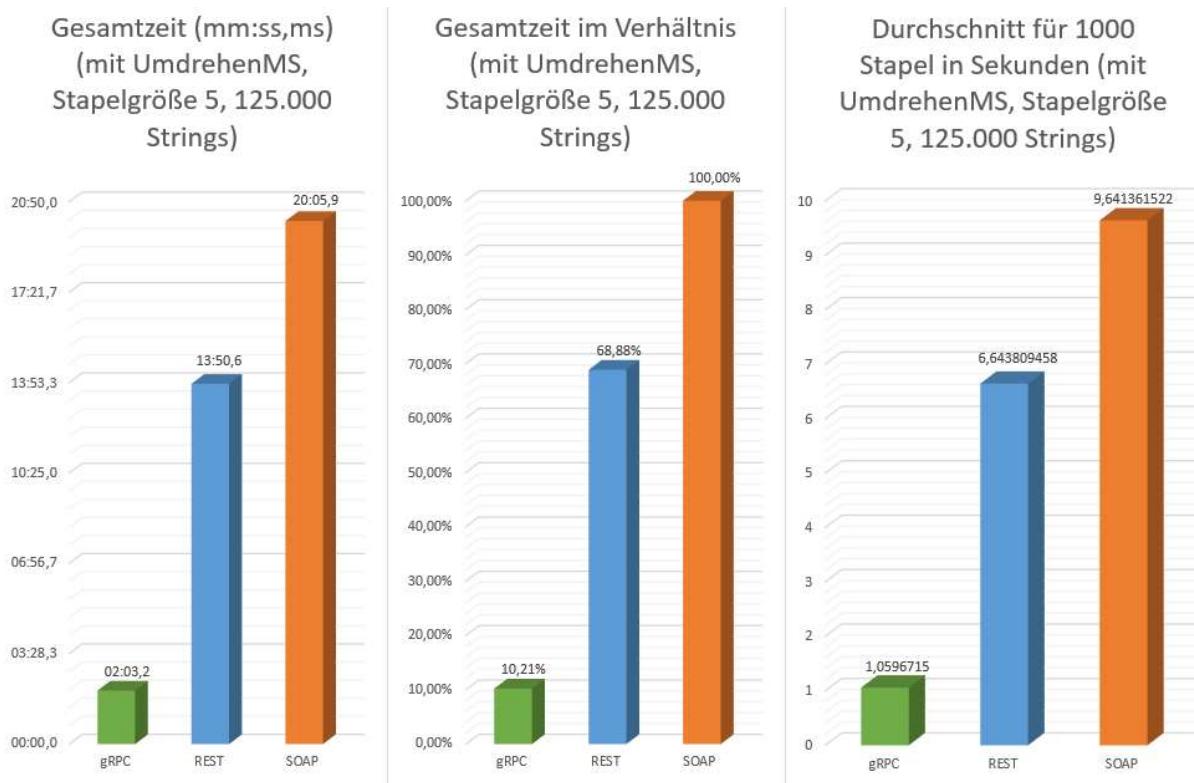


Abbildung 24: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 5, 125.000 Strings

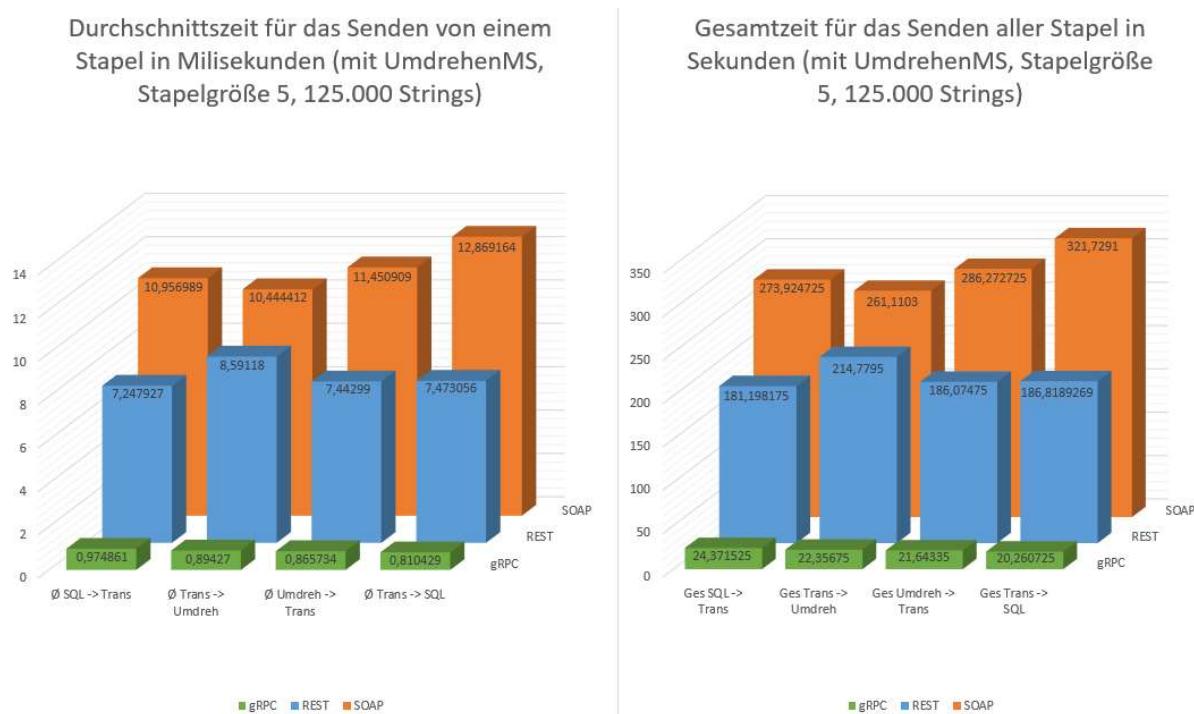


Abbildung 25: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 5, 125.000 Strings

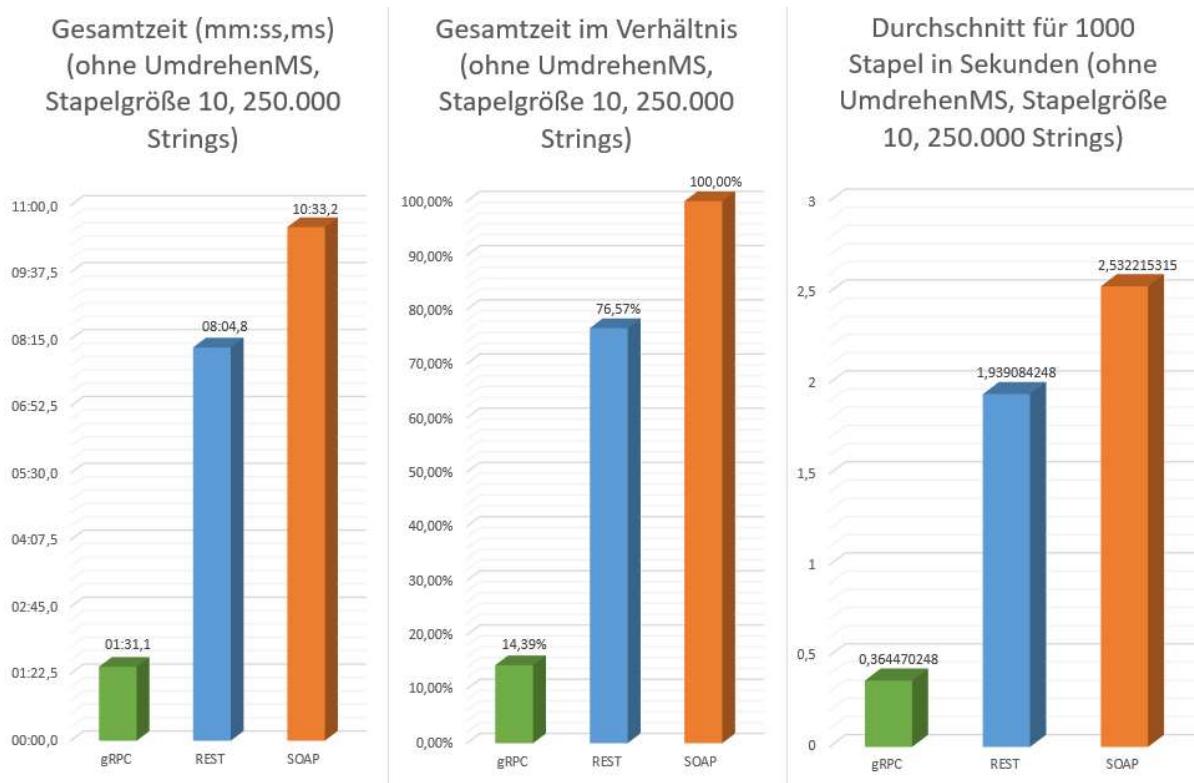


Abbildung 26: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 10, 250.000 Strings

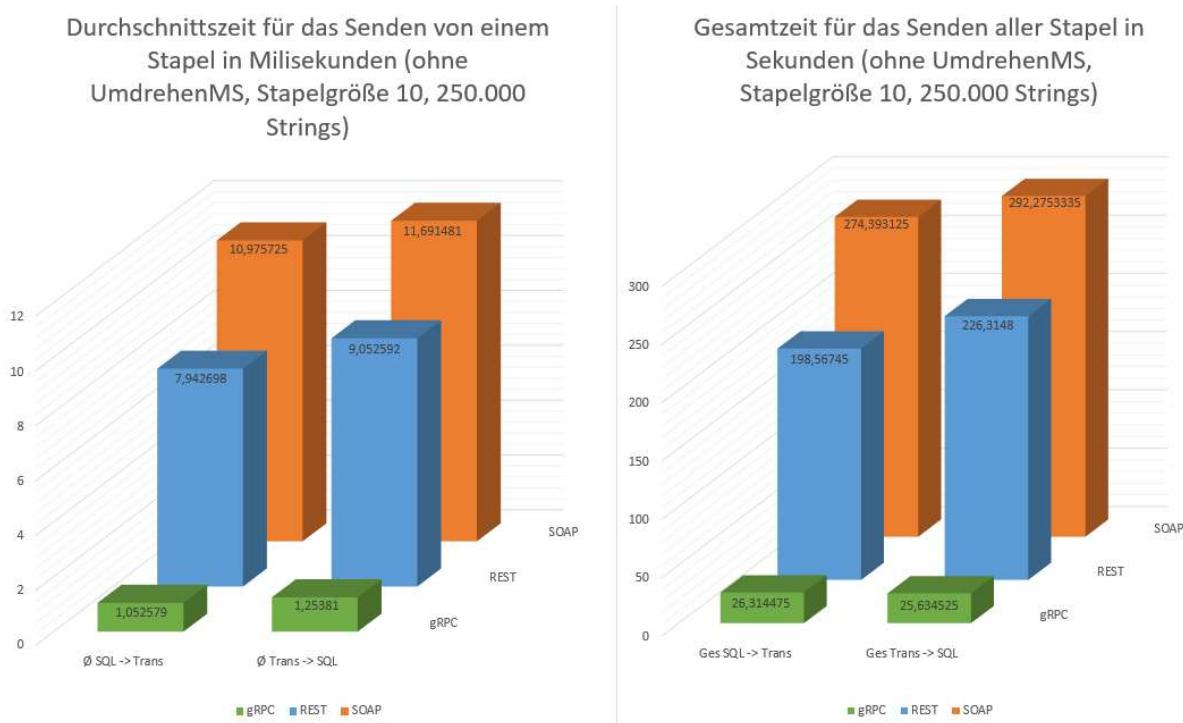


Abbildung 27: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 10, 250.000 Strings

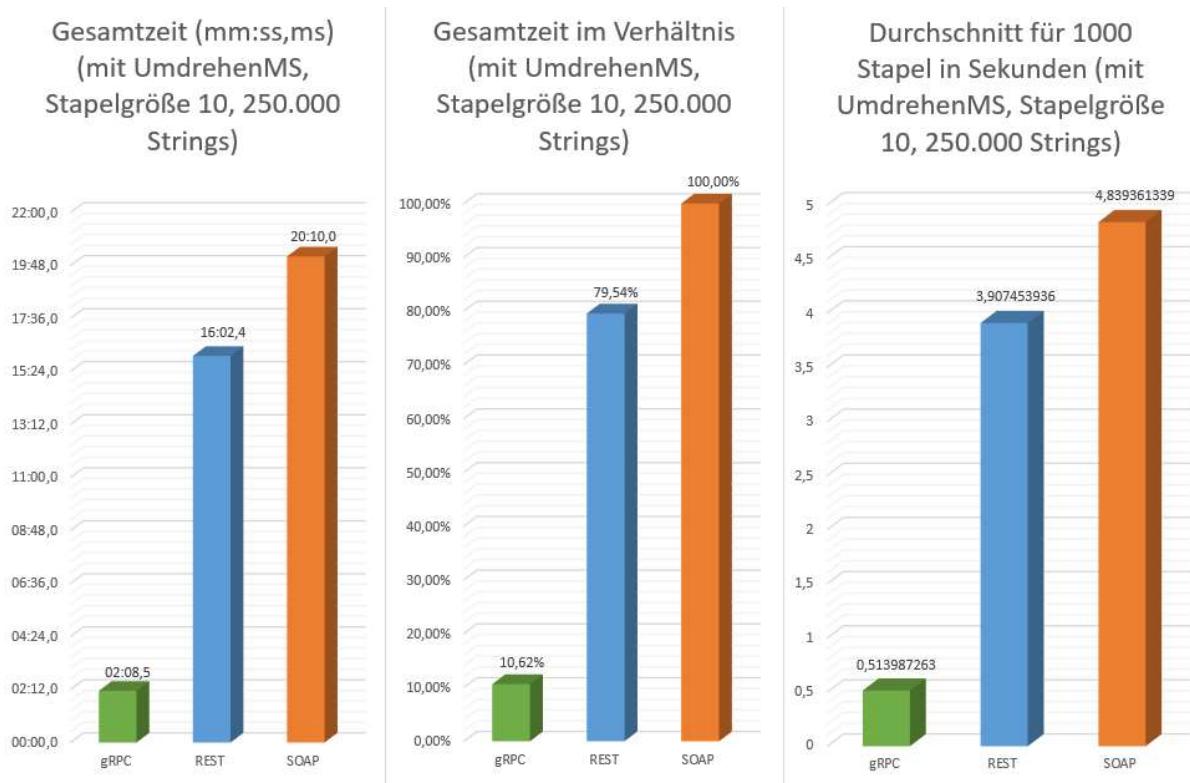


Abbildung 28: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 10, 250.000 Strings

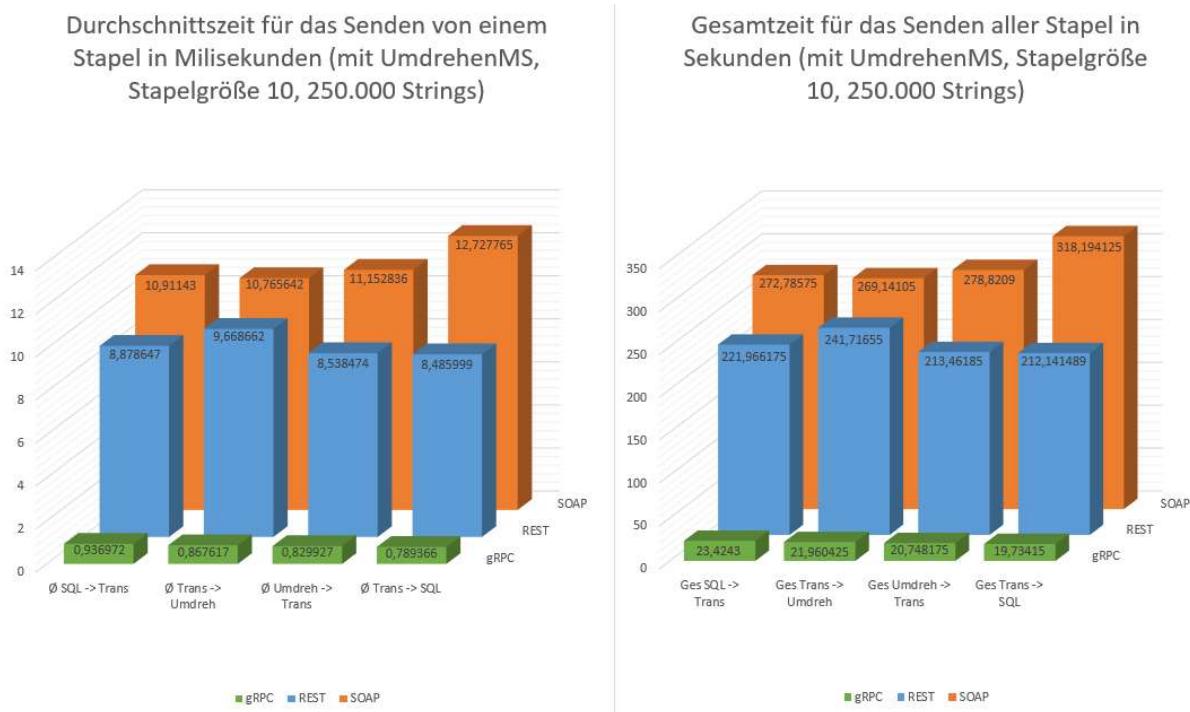


Abbildung 29: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 10, 250.000 Strings

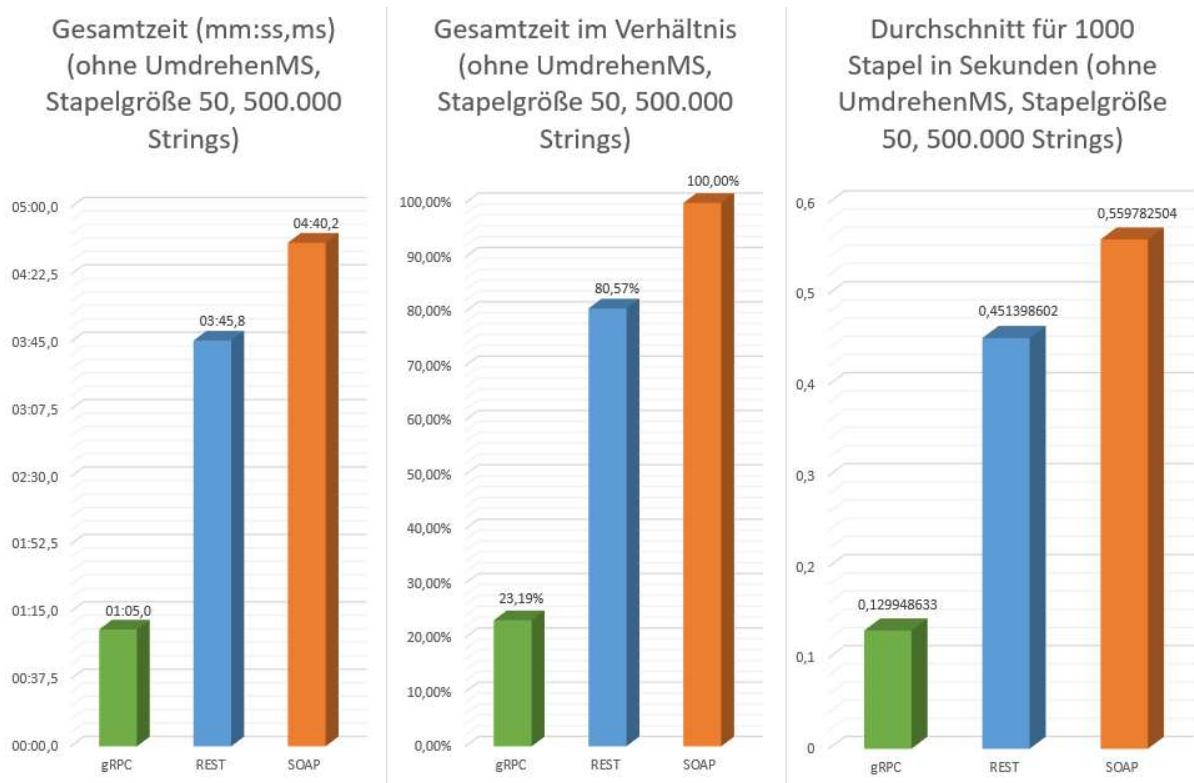


Abbildung 30: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 50, 500.000 Strings

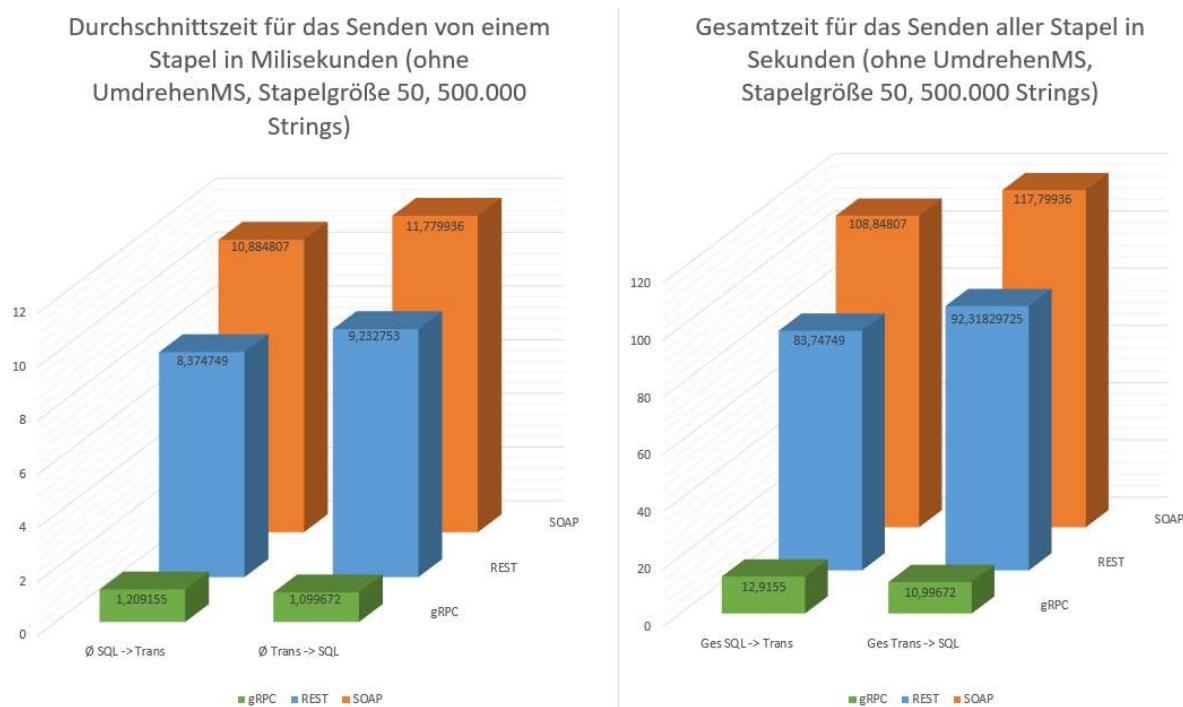


Abbildung 31: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 50, 500.000 Strings

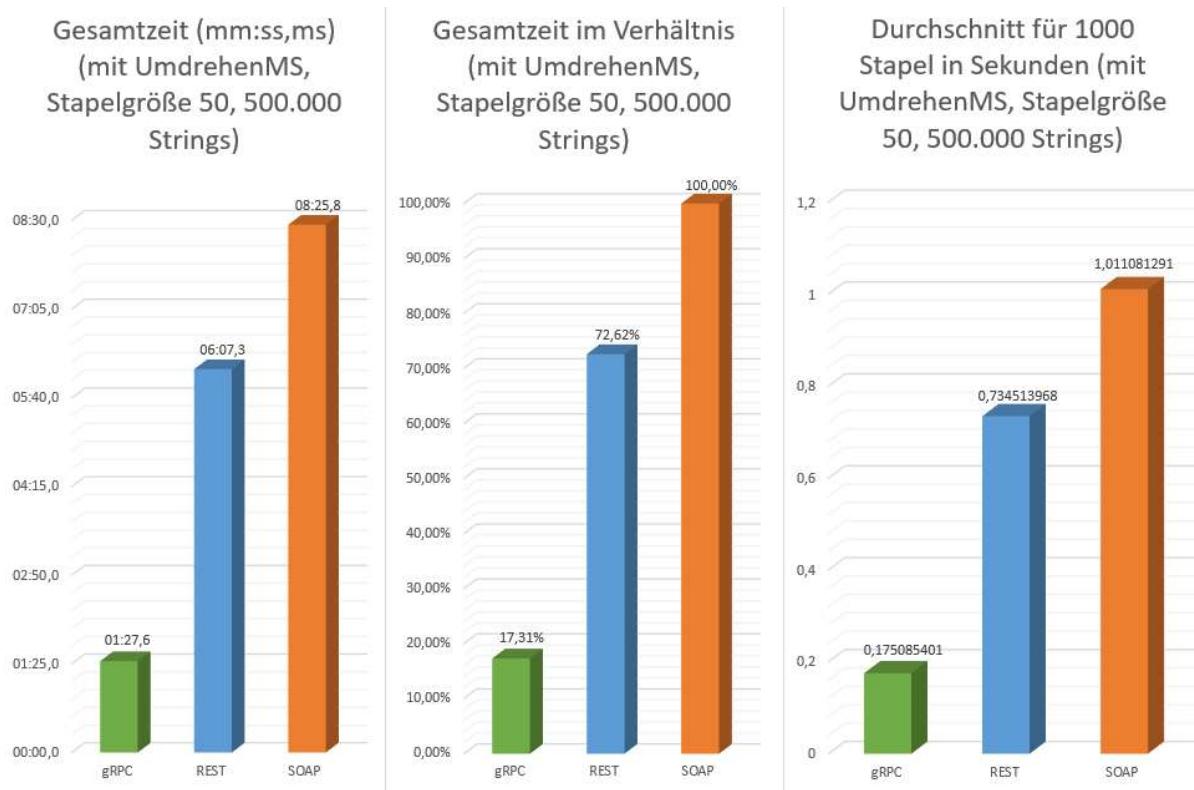


Abbildung 32: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 50, 500.000 Strings

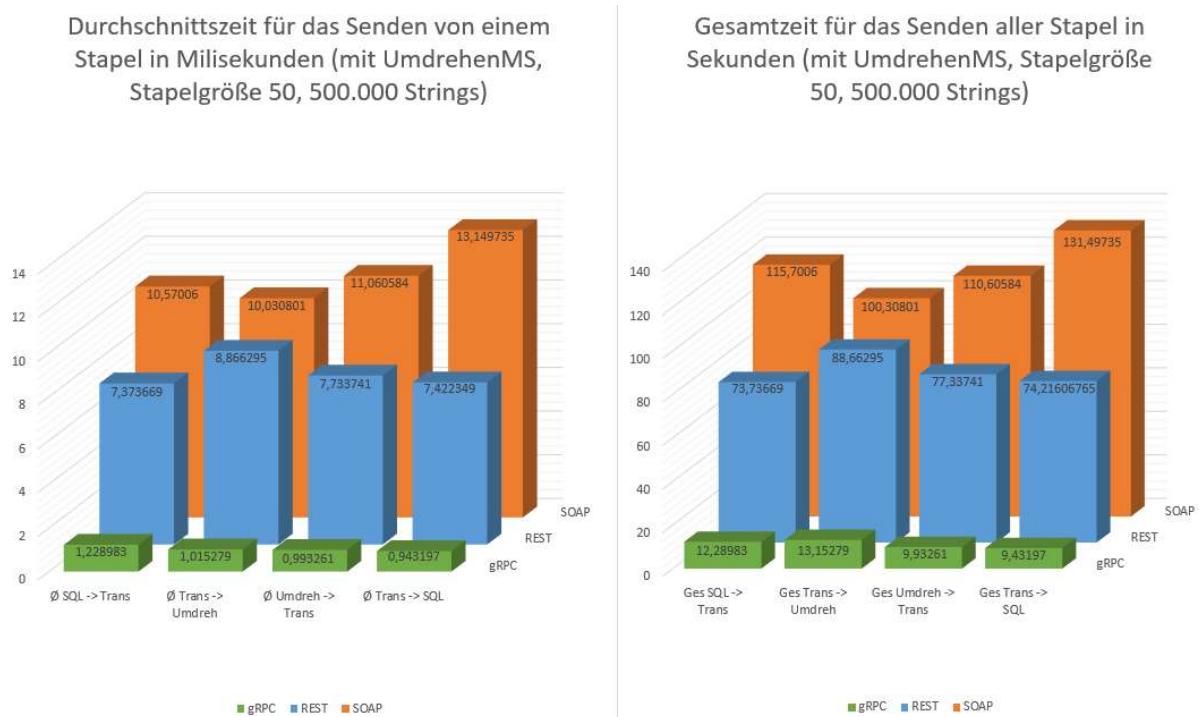


Abbildung 33: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 50, 500.000 Strings

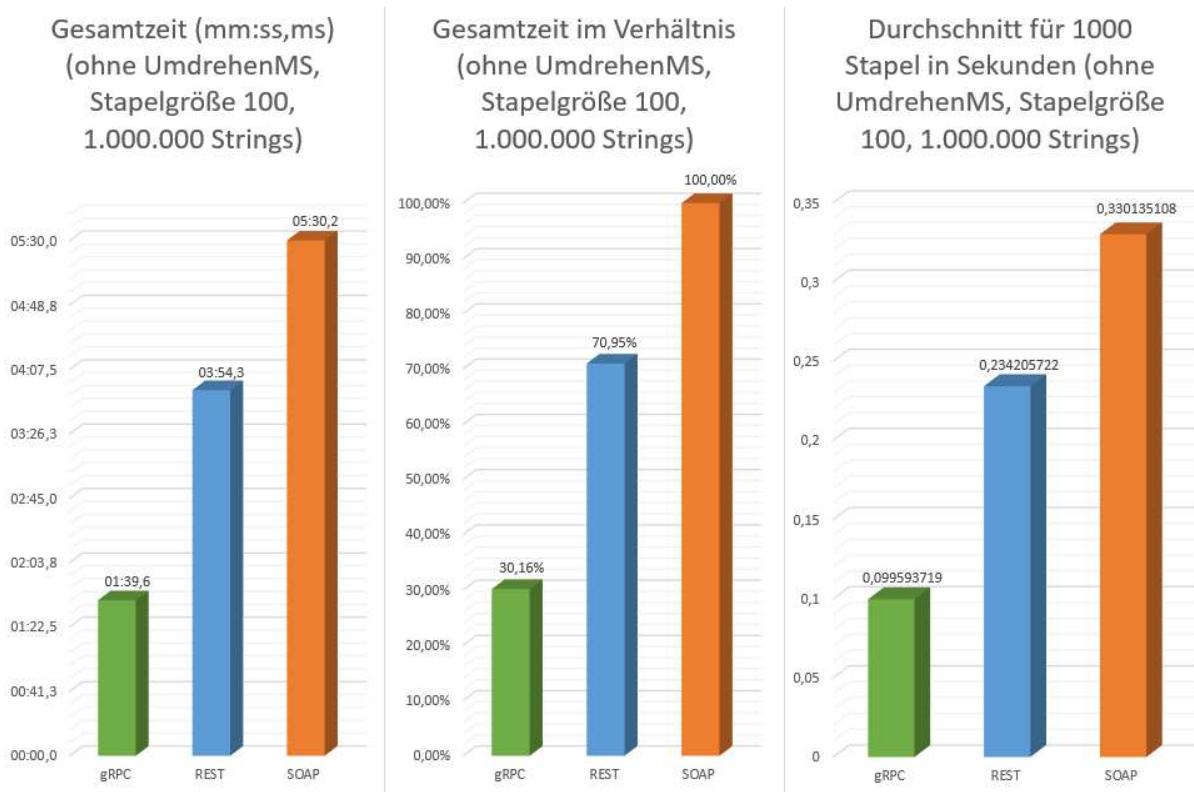


Abbildung 34: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 100, 1.000.000 Strings

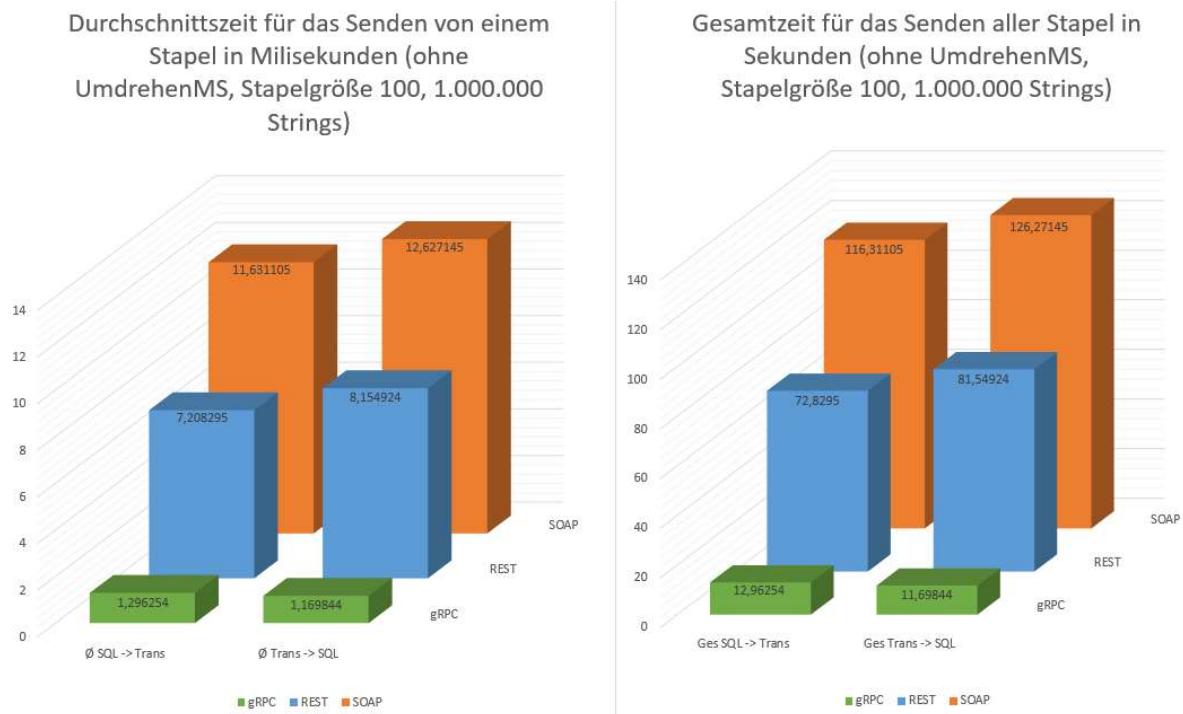


Abbildung 35: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 100, 1.000.000 Strings

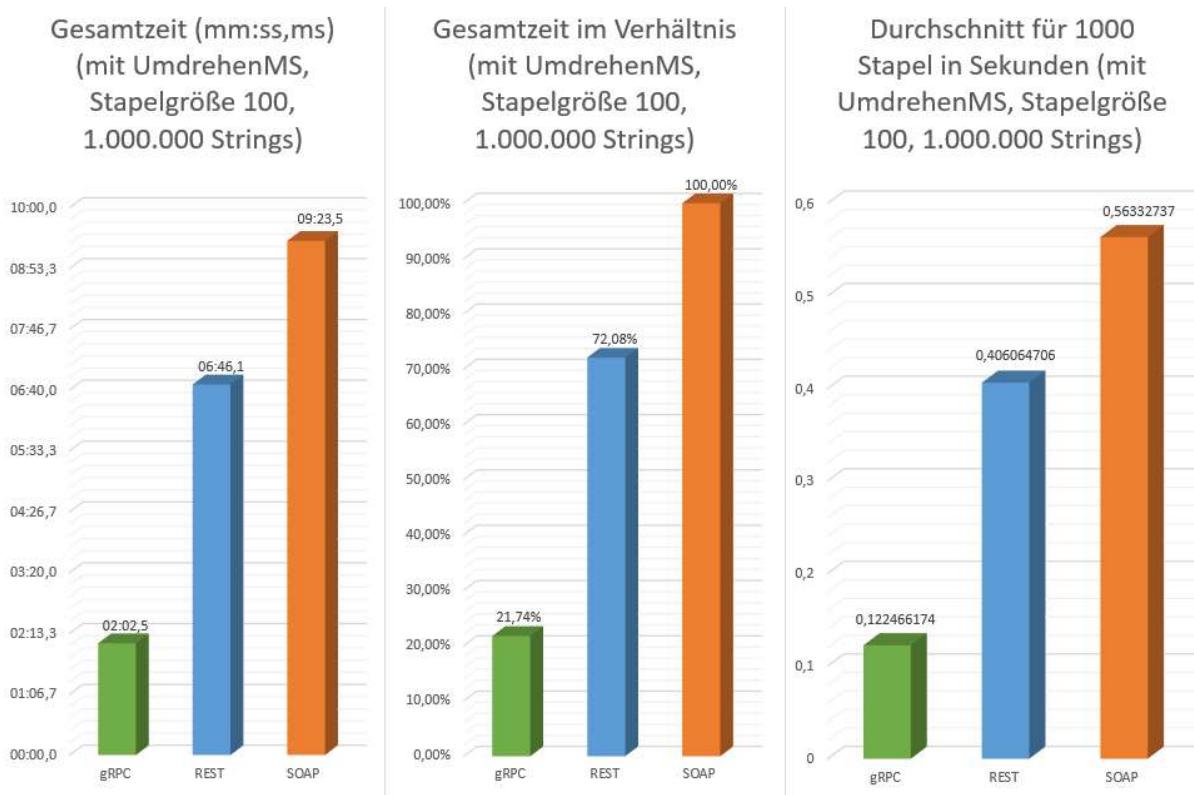


Abbildung 36: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 100, 1.000.000 Strings

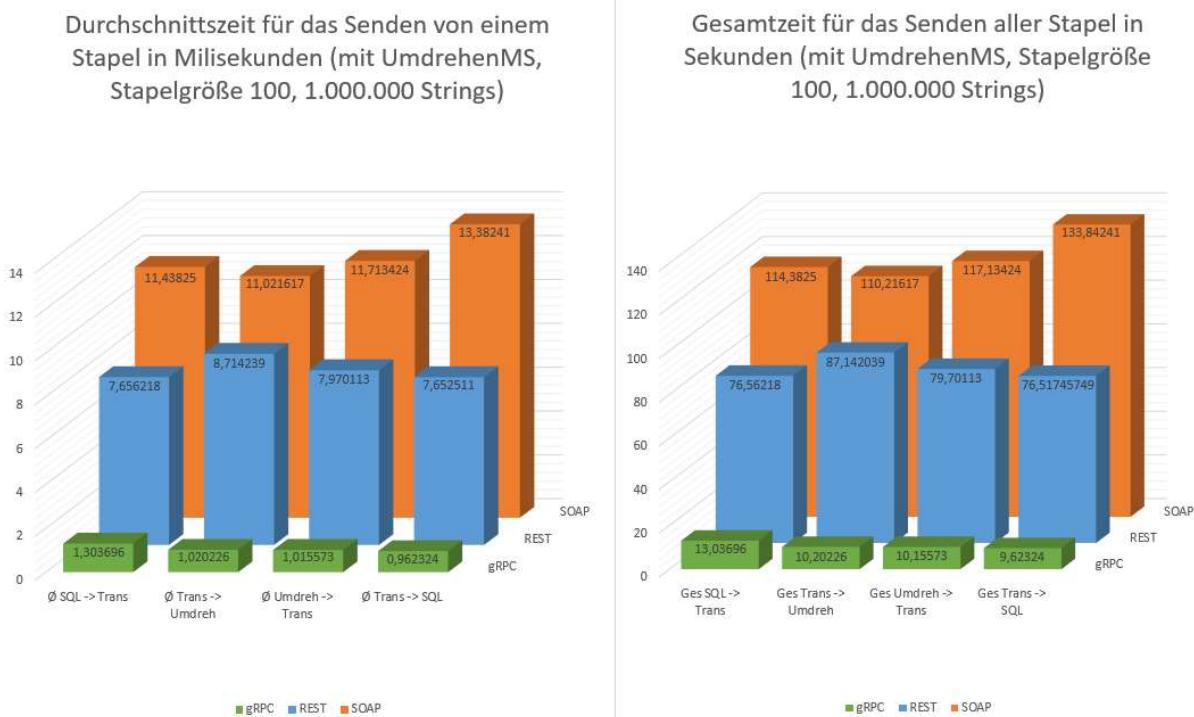


Abbildung 37: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 100, 1.000.000 Strings

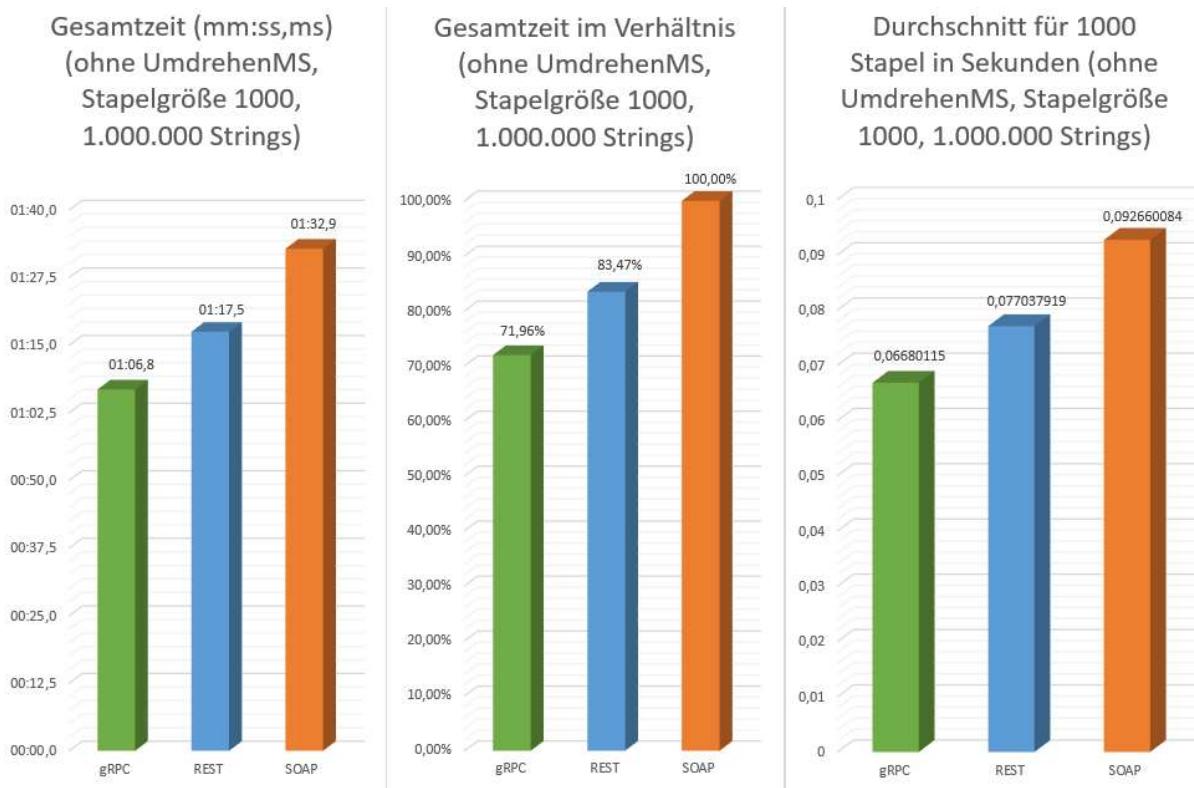


Abbildung 38: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 1.000, 1.000.000 Strings

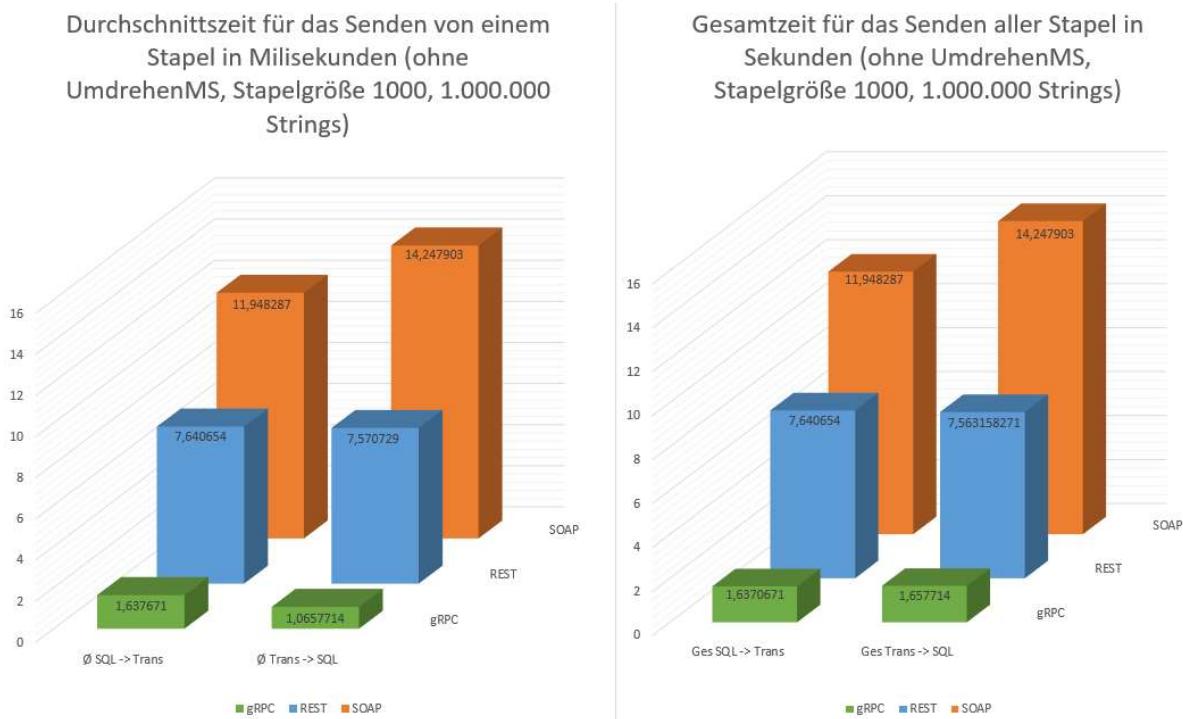


Abbildung 39: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 1.000, 1.000.000 Strings

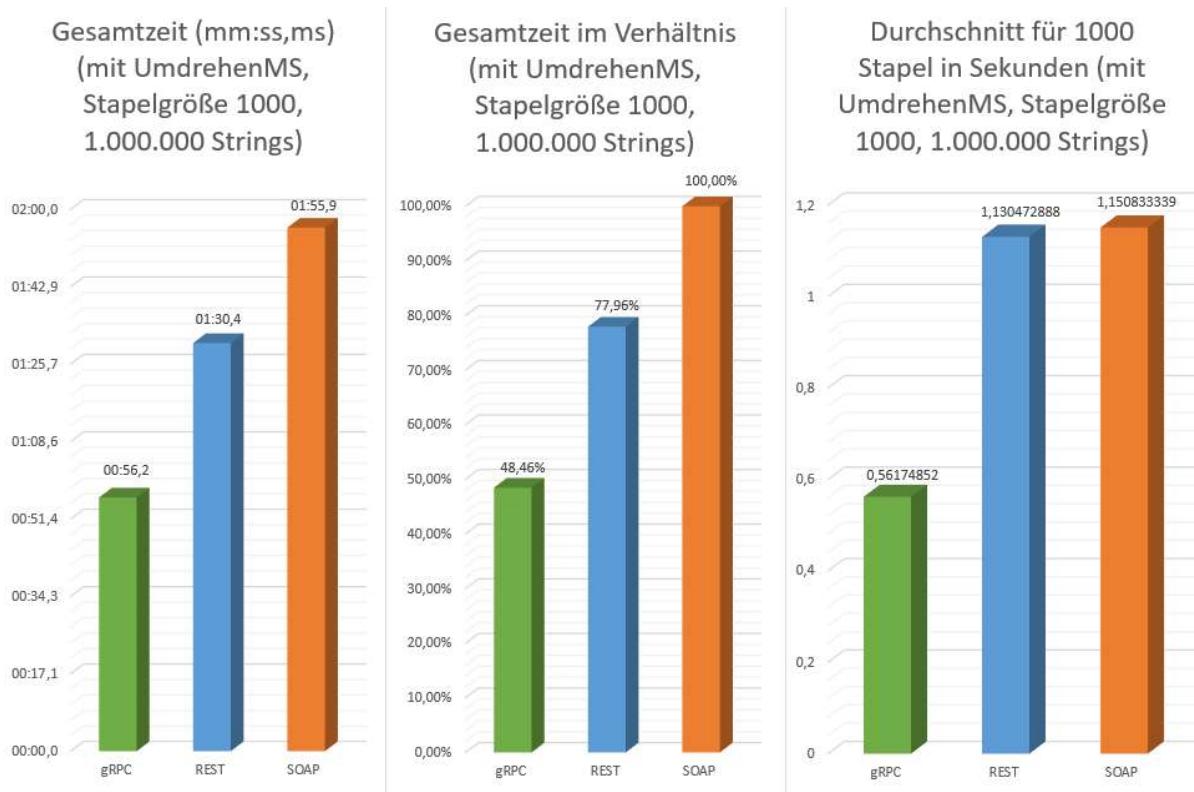


Abbildung 40: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 1.000, 1.000.000 Strings

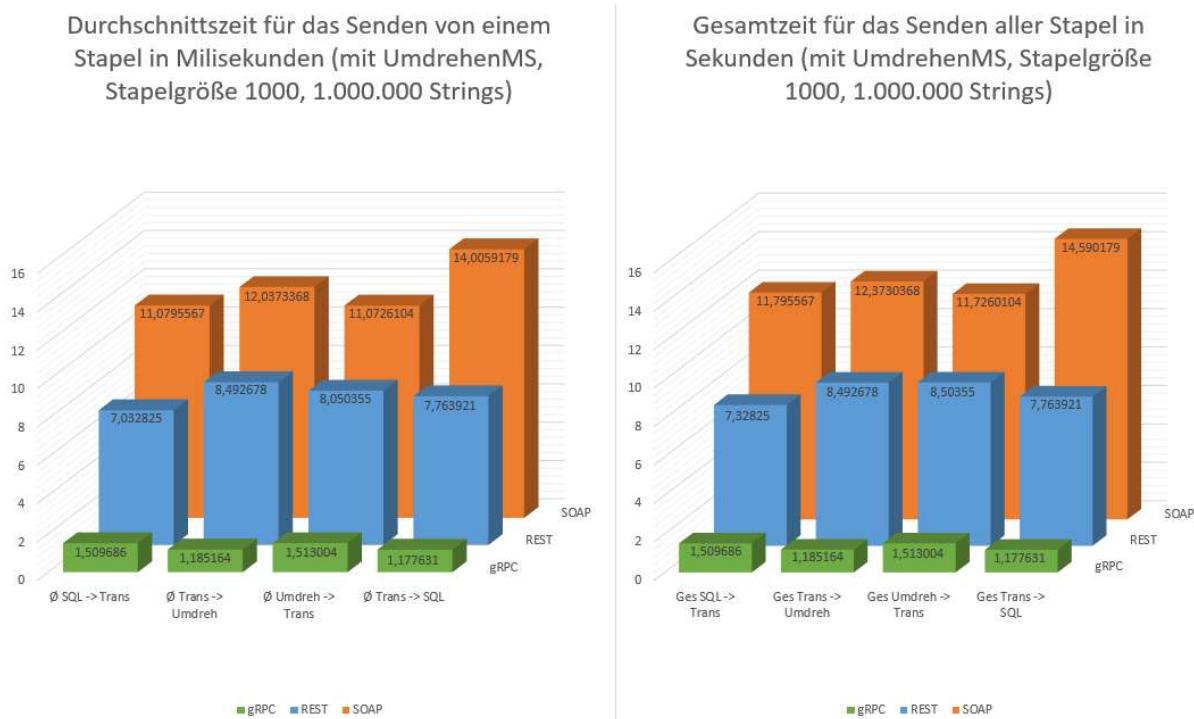


Abbildung 41: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 1.000, 1.000.000 Strings

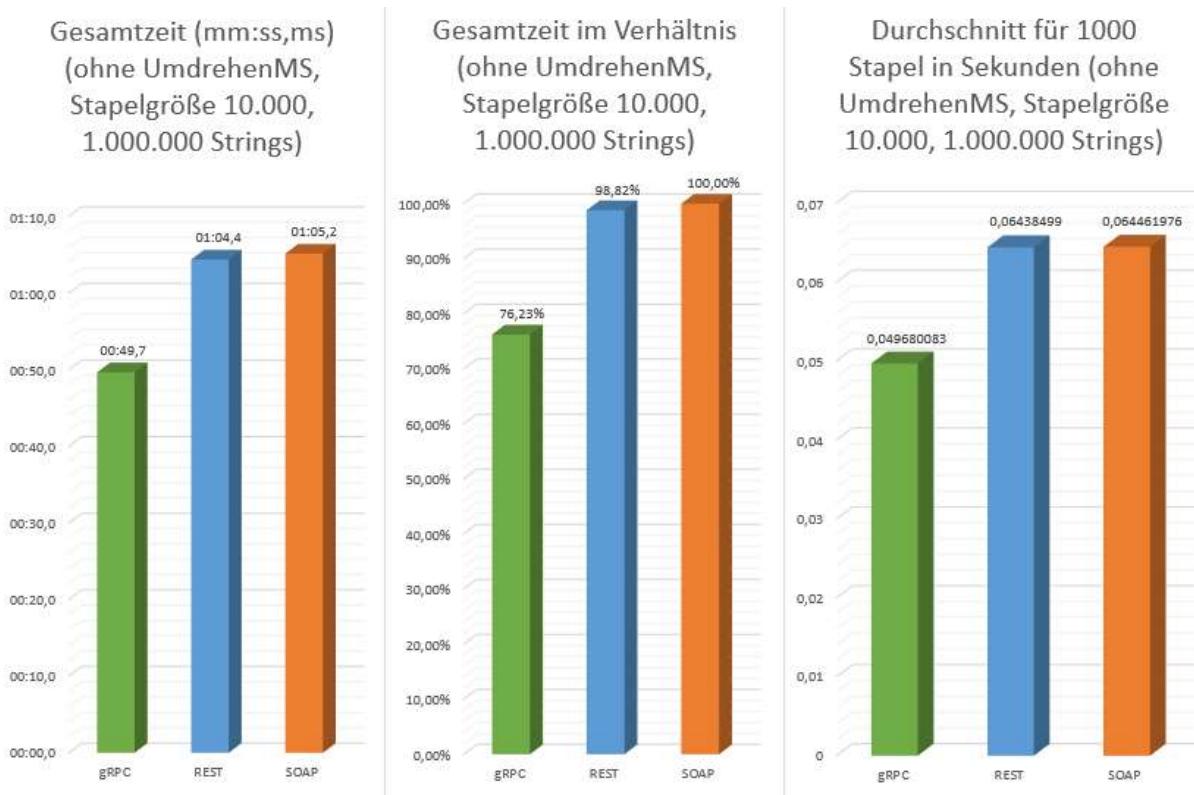


Abbildung 42: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 10.000, 1.000.000 Strings



Abbildung 43: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 10.000, 1.000.000 Strings

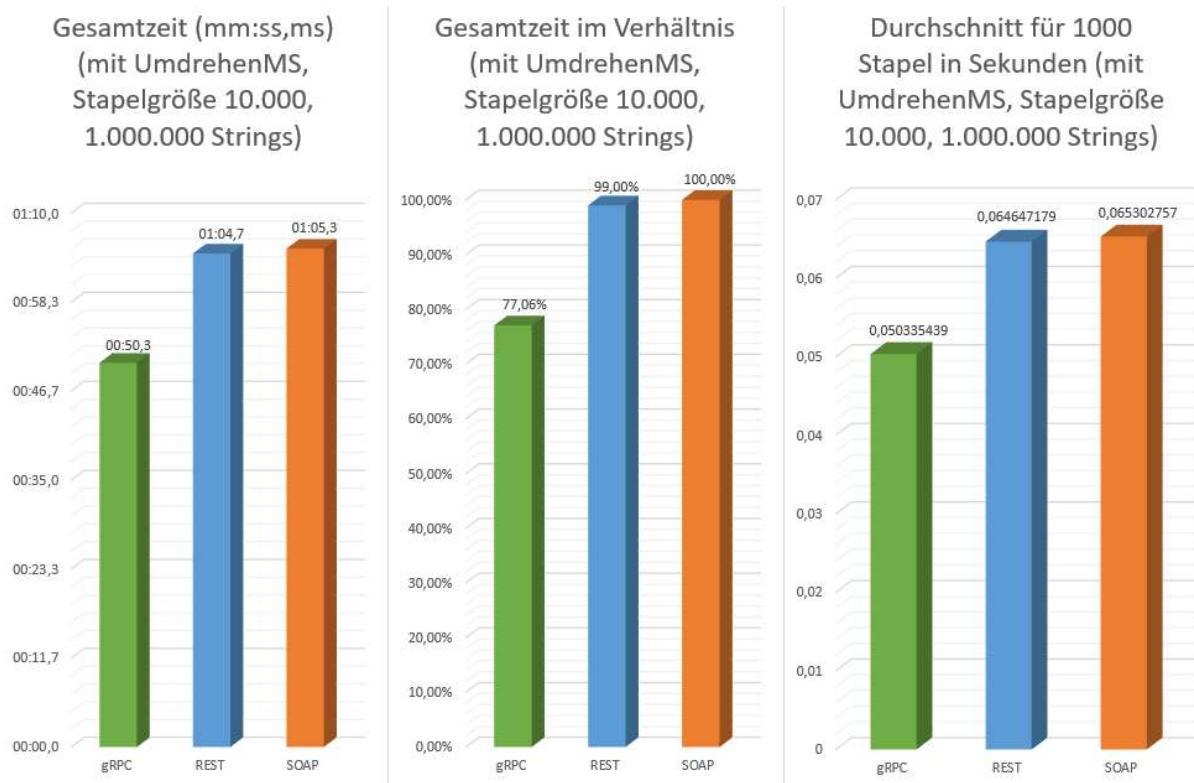


Abbildung 44: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 10.000, 1.000.000 Strings

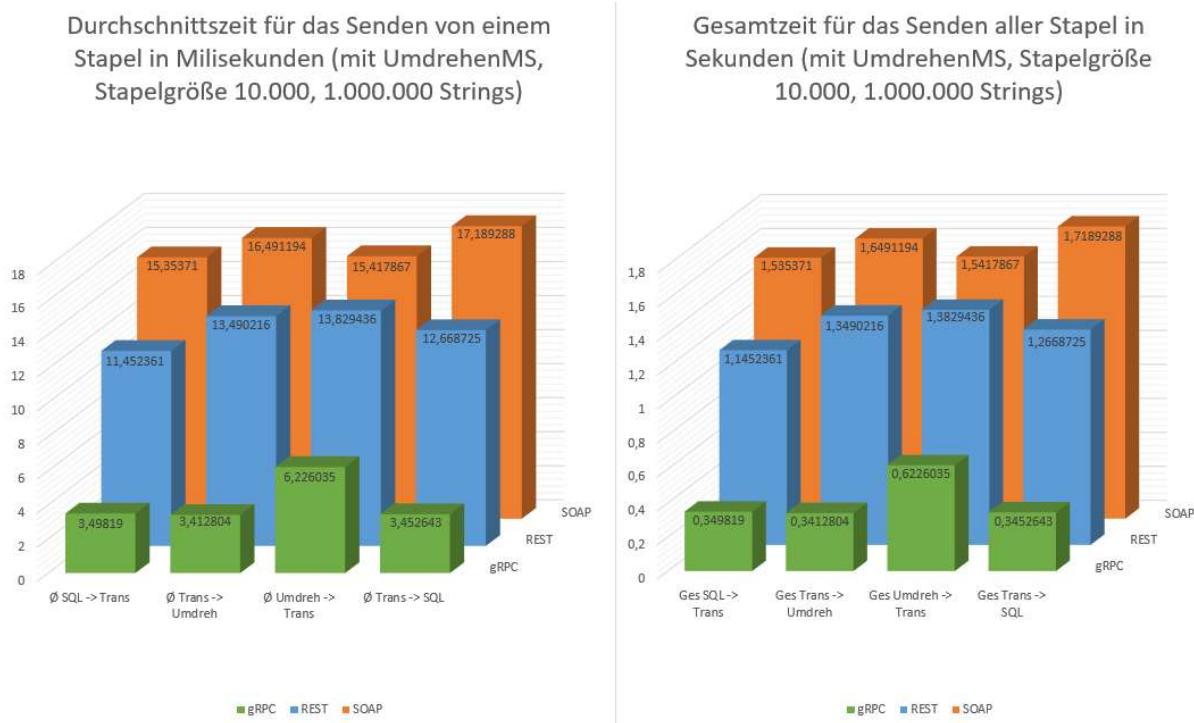


Abbildung 45: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 10.000, 1.000.000 Strings

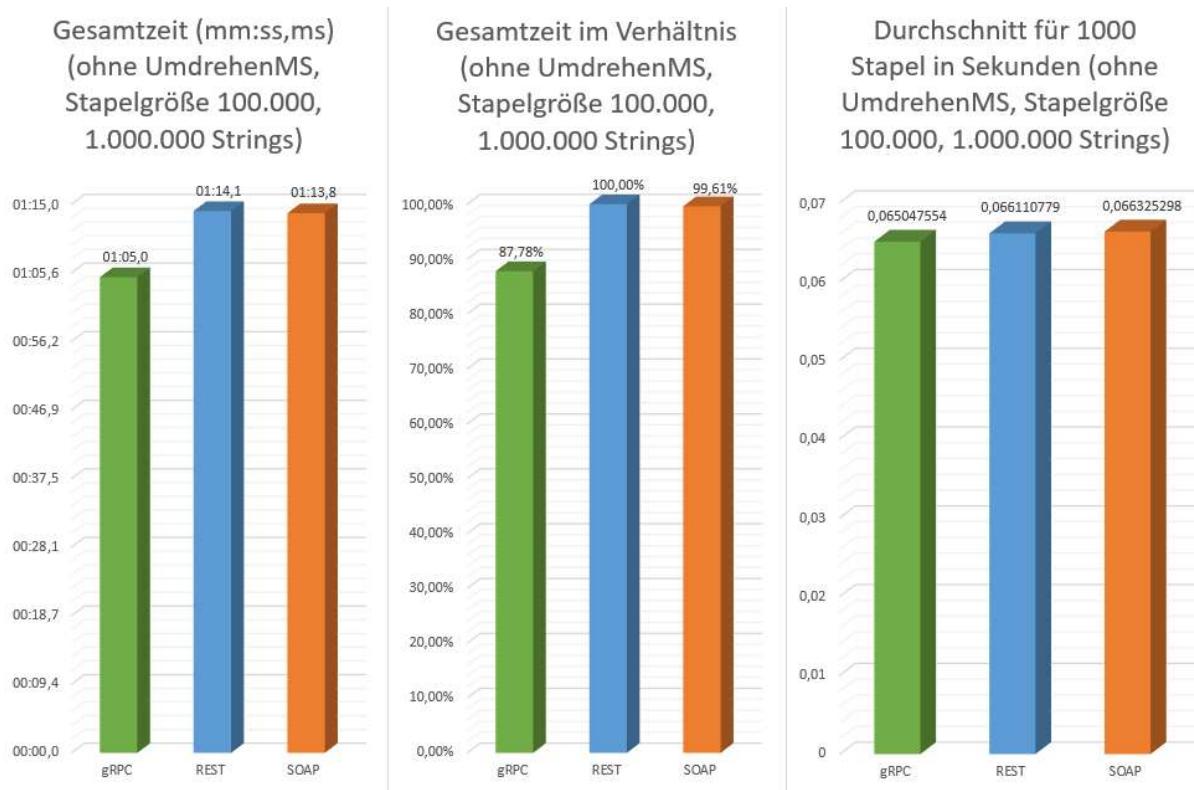


Abbildung 46: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 100.000, 1.000.000 Strings

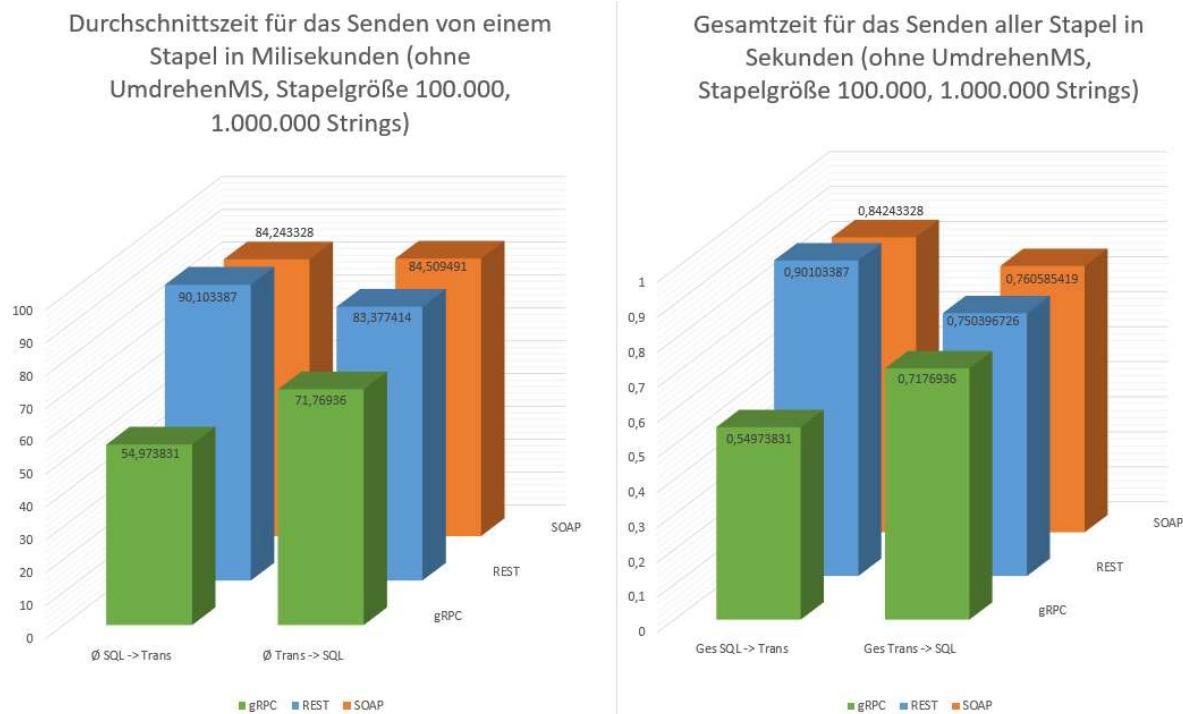


Abbildung 47: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 100.000, 1.000.000 Strings

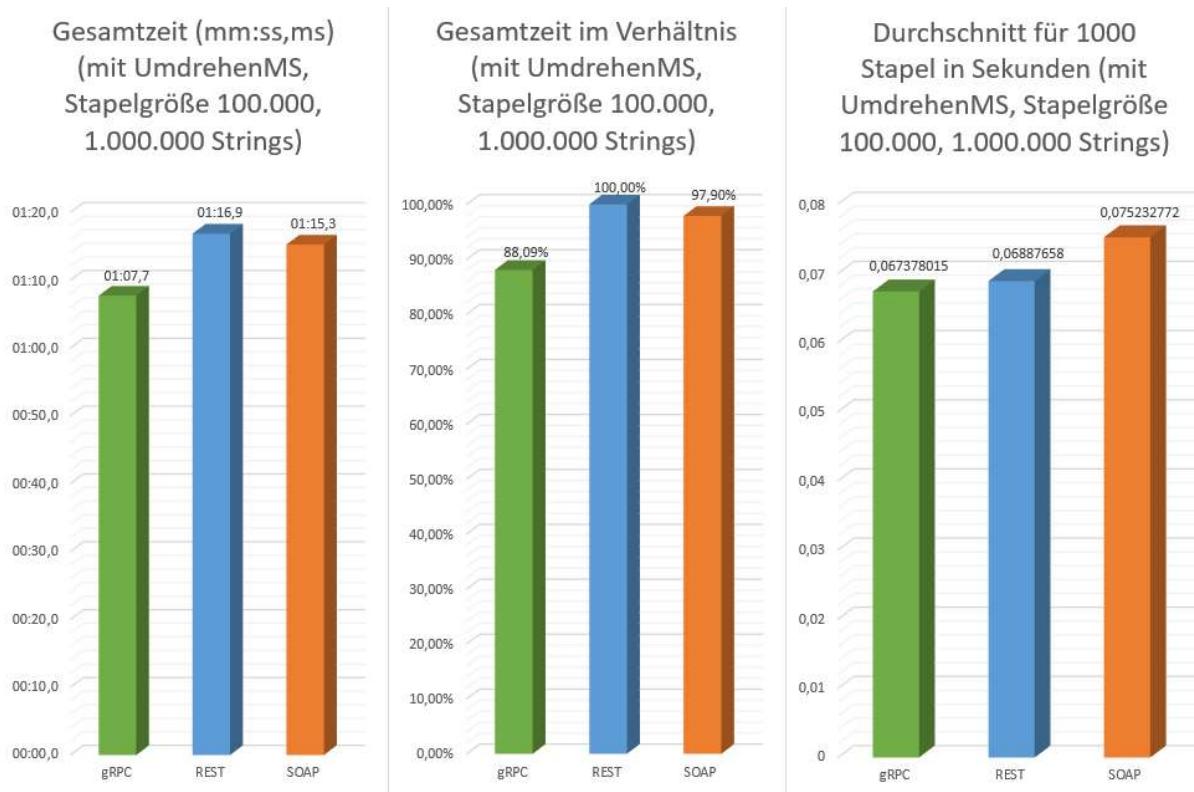


Abbildung 48: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 100.000, 1.000.000 Strings

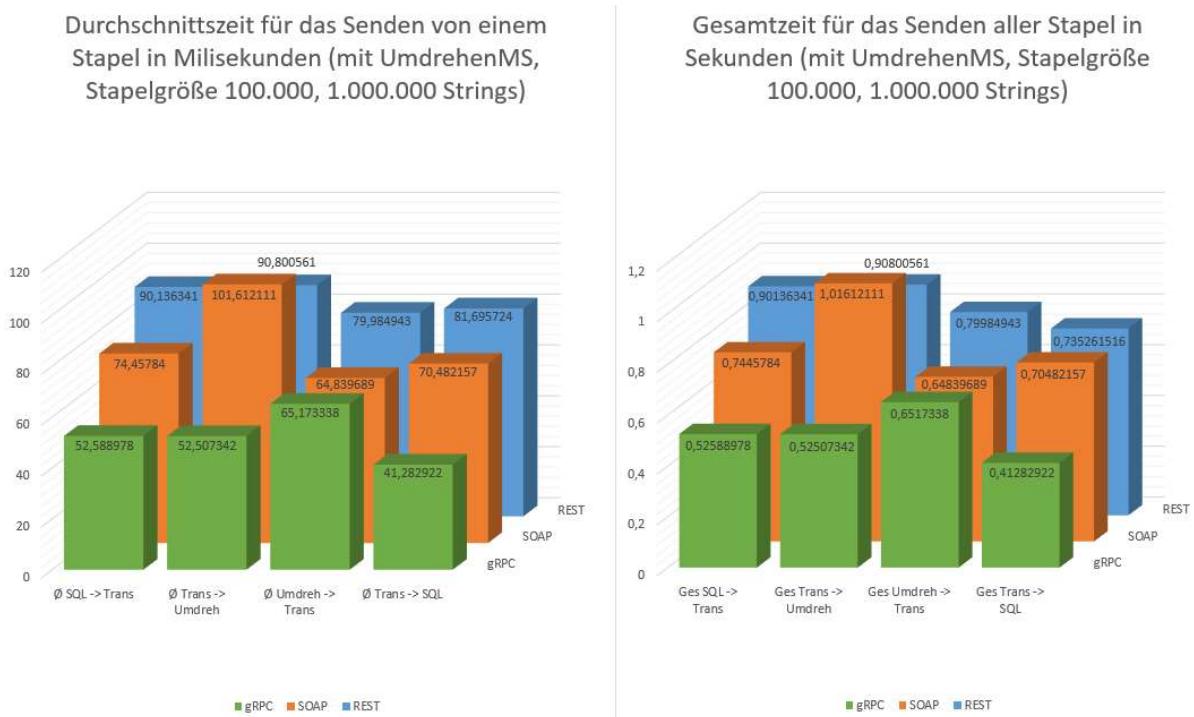


Abbildung 49: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 100.000, 1.000.000 Strings

6. Fazit

gRPC kann sich bis zu einer Stapelgröße von 1.000 Strings mühelos behaupten. Danach wird das Ergebnis nicht mehr so eindeutig. Das könnte eventuell auf zugewiesene Rechenzeit vom Windows-Scheduler zurückzuführen sein. Trotzdem schafft es weder REST noch SOAP, auch nur ein Mal die Aufträge schneller abzuarbeiten gRPC wie. Die technologischen Vorteile, die HTTP/2 und das Protobuf-Wire-Format mit sich bringen, verschaffen gRPC einen Geschwindigkeitsvorteil von bis zu 90,4 %. Dadurch bestätigt das Ergebnis, warum gRPC als „Game Changer“ unter den Microservice-Protokollen gilt.

Durch Google werden auch weiterhin Verbesserungen an gRPC durchgeführt. Zwar sind dort keine riesigen Sprünge in naher Zukunft zu erwarten, trotzdem wird Google versuchen das letzte Stück Performance aus gRPC rauszuholen.

REST kann allerdings in Sachen einfache Implementierung punkten. Ohne eine IDL, Stubs und Skeleton ist es für kleine Projekte sehr gut geeignet. Auch für Programmierneulinge ist dieses Protokoll um ein Vielfaches einfacher zu verstehen und daher für einen Einstieg bestens geeignet.

SOAP kann hingegen bei der Verfügbarkeit der Services punkten. Durch das online stellen der WSDL-Datei kann ein Nutzer schnell und bequem die Stubs für den gewünschten Service erstellen und diesen direkt im Anschluss auch nutzen.

REST und SOAP müssen allerdings, um konkurrenzfähig zu bleiben, auf das neue HTTP/2 upgraden. Human Readable Nachrichten zu versenden ist zwar ganz nett, allerdings nur zu debug Zwecken wirklich sinnvoll. Die dadurch vergeblichen Ressourcen sind mit einer lesbaren Nachricht nicht aufzuwiegen. Mehrere Verbindungen zum selben Ziel aufzubauen verbraucht viel Zeit und Ressourcen und führt zum Head-of-Line-Blocking Problem. Das neue HTTP/2 stellt hier eine wesentliche Verbesserung dar.

Durch gRPC sollte der Microservice-Ansatz noch einmal einen Schub bekommen, da eine der Kritikpunkte, die Performance, um eine Größenordnung von 10 verkleinert wurde. Somit könnte diese Schwäche von MS in absehbarer Zeit keiner mehr sein.

Abbildungsverzeichnis

Abbildung 1: Umfrage, wie viele Entwickler den Microservice-Ansatz in Ihren Projekten verwenden .	2
Abbildung 2: Zustimmung zu Problemen bei der Entwicklung von monolithischen Programmen	3
Abbildung 3: Aufbau des Projekts	4
Abbildung 4: Monolith verglichen mit Microservices	7
Abbildung 5: Funktionsweise von HTTP	8
Abbildung 6: Vergleich von Verbindungsaufbau HTTP 1.1 und HTTP/2.....	13
Abbildung 7: Allgemeiner Aufbau eines RPC-/RMI-Systems.....	14
Abbildung 8: Marshalling/Unmarshalling.....	15
Abbildung 9: Beispielobjekt.....	16
Abbildung 10: Benchmark-Verarbeitungsablauf ohne UmdrehenMS-Service	36
Abbildung 11: Benchmark-Verarbeitungsablauf mit UmdrehenMS-Service	37
Abbildung 12: Messpunkte des Benchmarks	38
Abbildung 13: Größenvergleich des Beispielobjekts.....	40
Abbildung 14: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 1, 30.000 Strings	41
Abbildung 15: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 1, 30.000 Strings	41
Abbildung 16: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 1, 30.000 Strings	42
Abbildung 17: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 1, 30.000 Strings.....	42
Abbildung 18: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 2, 60.000 Strings	43
Abbildung 19: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 2, 60.000 Strings	43
Abbildung 20: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 2, 60.000 Strings	44
Abbildung 21: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 2, 60.000 Strings.....	44
Abbildung 22: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 5, 125.000 Strings	45
Abbildung 23: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 5, 125.000 Strings	45
Abbildung 24: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 5, 125.000 Strings.....	46
Abbildung 25: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 5, 125.000 Strings.....	46
Abbildung 26: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 10, 250.000 Strings	47
Abbildung 27: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 10, 250.000 Strings	47
Abbildung 28: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 10, 250.000 Strings	48
Abbildung 29: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 10, 250.000 Strings	48
Abbildung 30: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 50, 500.000 Strings	49
Abbildung 31: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 50, 500.000 Strings	49
Abbildung 32: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 50, 500.000 Strings.....	50
Abbildung 33: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 50, 500.000 Strings	50
Abbildung 34: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 100, 1.000.000 Strings	51
Abbildung 35: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 100, 1.000.000 Strings	51
Abbildung 36: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 100, 1.000.000 Strings.....	52
Abbildung 37: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 100, 1.000.000 Strings	52
Abbildung 38: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 1.000, 1.000.000 Strings	53
Abbildung 39: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 1.000, 1.000.000 Strings	53
Abbildung 40: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 1.000, 1.000.000 Strings	54
Abbildung 41: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 1.000, 1.000.000 Strings.....	54
Abbildung 42: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 10.000, 1.000.000 Strings	55
Abbildung 43: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 10.000, 1.000.000 Strings	55
Abbildung 44: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 10.000, 1.000.000 Strings	56
Abbildung 45: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 10.000, 1.000.000 Strings.....	56
Abbildung 46: Ergebnisse 1/2 für ohne UmdrehenMS, Stapelgröße 100.000, 1.000.000 Strings.....	57

Abbildung 47: Ergebnisse 2/2 für ohne UmdrehenMS, Stapelgröße 100.000, 1.000.000 Strings.....	57
Abbildung 48: Ergebnisse 1/2 für mit UmdrehenMS, Stapelgröße 100.000, 1.000.000 Strings.....	58
Abbildung 49: Ergebnisse 2/2 für mit UmdrehenMS, Stapelgröße 100.000, 1.000.000 Strings.....	58

Listingverzeichnis

Listing 1: HTTP-GET-Request	9
Listing 2: HTTP-POST-Request	10
Listing 3: HTTP Response	11
Listing 4: HTTP-POST-Header	15
Listing 5: Darstellung des Beispielobjekts in JSON	17
Listing 6: Hexadezimaldarstellung der JSON-Darstellung des Beispielobjekts.....	17
Listing 7: XML-Kurzdarstellung des Beispielobjekts	18
Listing 8: XML-Namespace-Beispiel.....	18
Listing 9: XML-Namespace-Präfixbeispiel	19
Listing 10: SOAP-Darstellung des Beispielobjekts	19
Listing 11: Hexadezimaldarstellung der SOAP-Nachricht des Beispielobjekts	20
Listing 12: Beispielobjekt als gRPC-Protobuf-IDL-Darstellung.....	20
Listing 13: Hexadezimale Darstellung des Beispielobjekts im gRPC-Protobuf-Wireformat.....	21
Listing 14: gRPC-Wireformat, farblich hervorgehoben	22
Listing 15: SOAP-Schema	23
Listing 16: Grundstruktur einer WSDL-Datei.....	24
Listing 17: Auszug aus einer Protobuf-Datei	26
Listing 18: GET-Methode in Jax-RS	28
Listing 19: POST-Methode in Jax-RS	29
Listing 20: GET-Anfrage mit Jax-RS.....	30
Listing 21: POST-Anfrage mit Jax-RS.....	30
Listing 22: SOAP-Annotation – Interface	31
Listing 23: SOAP-Annotation – Implementationsklasse	31
Listing 24: SOAP-Service aufrufen	32
Listing 25: Proto-Datei für gRPC-Bespiel	33
Listing 26: gRPC-Server starten und Service hinzufügen.....	34
Listing 27: gRPC-Client-Verbindung zum Server.....	35

Tabellenverzeichnis

Tabelle 1: Wiretype-Nummern und ihre Bedeutung	21
Tabelle 2: Technikvergleich der Protokolle	39
Tabelle 3: Zusammensetzung der Testreihen für die Benchmark-Durchläufe.....	40

Verweise

- [1] H. Schlosser, „Microservices-Trends 2017: Strategien, Tools & Frameworks,“ Software & Support Media Group, 11. 04. 2017. [Online]. Available: <https://jaxenter.de/microservices-trends-55937>. [Zugriff am 11. 07. 2017].
- [2] E. Wolff, Microservices - Grundlagen flexibler Softwarearchitekturen, dpunkt.verlag, 2016.
- [3] T. Berners-Lee, „RFC1945 - Hypertext Transfer Protocol -- HTTP/1.0,“ 01. 05. 1996. [Online]. Available: <https://tools.ietf.org/html/rfc1945>. [Zugriff am 22. 06. 2017].
- [4] T. Berners-Lee, „RFC2616 - Hypertext Transfer Protocol -- HTTP/1.1,“ 01. 06. 1999. [Online]. Available: <https://tools.ietf.org/html/rfc2616>. [Zugriff am 22. 06. 2017].
- [5] M. Mandau, „www.chip.de,“ CHIP Digital GmbH, 17. 04. 2015. [Online]. Available: http://www.chip.de/artikel/HTTP_2-Der-neue-Web-Standard_75826963.html. [Zugriff am 21. 06. 2017].
- [6] S. Microsystems, „RPC: Remote Procedure Call Protokol Specification Version 2,“ 01. 06. 1988. [Online]. Available: <https://tools.ietf.org/html/rfc1057>. [Zugriff am 24. 06. 2017].
- [7] S. Microsystems, „RPC: Remote Procedure Call Protocol Specification Version 2,“ 01. 05. 2009. [Online]. Available: <https://tools.ietf.org/html/rfc5531>. [Zugriff am 24. 06. 2017].
- [8] G. Coulouris, Distributed Systems - Concepts and Design, Addison Wesley, 2001.
- [9] N. Freed, „Multipurpose Internet Mail Extension (MIME) Media Types,“ 01. 11. 1996. [Online]. Available: <https://www.ietf.org/rfc/rfc2046.txt>.
- [10] N. Freed, „Media Types,“ 12. 06. 2017. [Online]. Available: <https://www.iana.org/assignments/media-types/media-types.xhtml>.
- [11] E. Bray, „IETF - The JavaScript Object Notation (JSON) Data Interchange Format,“ Internet Engineering Task Force, 01. 04. 2014. [Online]. Available: <https://tools.ietf.org/html/rfc7159>. [Zugriff am 20. 06. 2017].
- [12] S. Tilkov, REST und HTTP - Entwicklung und Integration nach dem Architekturstil des Web, dpunkt.verlag, 2015.
- [13] T. Bray, „Extensible Markup Language (XML) 1.0 (Fifth Edition),“ 07. 10. 2013. [Online]. Available: <https://www.w3.org/TR/REC-xml/>. [Zugriff am 26. 06. 2017].
- [14] Google, „Protocol Buffers Encoding,“ 14. 04. 2016. [Online]. Available: <https://developers.google.com/protocol-buffers/docs/encoding>. [Zugriff am 27. 06. 2017].
- [15] E. K. MIT, „Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language,“ 26. 06. 2007. [Online]. Available: <https://www.w3.org/TR/wsdl20/>.

- [16] D. Peterson, „W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes,“ 05. 04. 2012. [Online]. Available: <https://www.w3.org/TR/xmlschema11-2/>. [Zugriff am 12. 07. 2017].
- [17] G. Inc., „Protocol Buffers - Google's data interchange format,“ 01. 01. 2008. [Online]. Available: <https://github.com/google/protobuf>. [Zugriff am 13. 07. 2017].
- [18] Google, „Protocol Buffers Version 3 Language Specification,“ 12. 05. 2017. [Online]. Available: <https://developers.google.com/protocol-buffers/docs/reference/proto3-spec>. [Zugriff am 13. 07. 2017].
- [19] Wikipedia, „SOAP,“ 13. 07. 2017. [Online]. Available: <https://en.wikipedia.org/wiki/SOAP>. [Zugriff am 15. 07. 2017].
- [20] K. Ballinger, „Basic Profile Version 1.1,“ 11. 04. 2006. [Online]. Available: <http://www.ws-i.org/profiles/basicprofile-1.1.html>. [Zugriff am 15. 07. 2017].
- [21] M. Gudgin, „SOAP Version 1.2 Part 1: Messaging Framework (Second Edition),“ 27. 04. 2007. [Online]. Available: <https://www.w3.org/TR/soap12/>. [Zugriff am 15. 07. 2017].
- [22] gRPC, „About gRPC,“ 01. 01. 2017. [Online]. Available: <https://grpc.io/about/>. [Zugriff am 15. 07. 2017].
- [23] gRPC, „gRPC Documentation,“ 01. 01. 2017. [Online]. Available: <https://grpc.io/docs/>. [Zugriff am 15. 07. 2017].
- [24] gRPC, „gRPC Benchmarking,“ 01. 01. 2017. [Online]. Available: <https://grpc.io/docs/guides/benchmarking.html>. [Zugriff am 15. 07. 2017].
- [25] gRPC, „gRPC Performance Multi-language Dashboard gRPC,“ 14. 07. 2017. [Online]. Available: <https://performance-dot-grpc-testing.appspot.com/explore?dashboard=5636470266134528&widget=1274284569&container=695104531>. [Zugriff am 15. 07. 2017].
- [26] G.-H. Lee, „etcd: distributed key-value store with grpc/http2,“ 09. 12. 2015. [Online]. Available: <https://blog.gopheracademy.com/advent-2015/etcd-distributed-key-value-store-with-grpc-http2/>. [Zugriff am 15. 07. 2017].
- [27] H. Bee, „REST v. gRPC,“ 28. 05. 2016. [Online]. Available: <https://husobee.github.io/golang/rest/grpc/2016/05/28/golang-rest-v-grpc.html>. [Zugriff am 15. 07. 2017].
- [28] G. Krüger, Handbuch der Java Programmierung, Addison-Wesley, 2009.